UNIVERSIDADE
FEDERAL RURAL
DE PERNAMBUCO

Edilson Alves de Andrade Júnior

Uma análise do impacto das linguagens de programação nos custos de execução no AWS Lambda em cenários de *cold start* e *warm start*

Recife

2023

Edilson Alves de Andrade Júnior

Uma análise do impacto das linguagens de programação nos custos de execução no AWS Lambda em cenários de *cold start* e *warm start*

Monografia apresentada ao Curso de Bacharelado em Ciência da Computação da Universidade Federal Rural de Pernambuco, como requisito parcial para obtenção do título de Bacharel em Ciência da Computação.

Universidade Federal Rural de Pernambuco – UFRPE

Departamento de Computação

Curso de Bacharelado em Ciência da Computação

Orientador: Robson Wagner Albuquerque de Medeiros

Recife

2023

Dados Internacionais de Catalogação na Publicação
Universidade Federal Rural de Pernambuco
Sistema Integrado de Bibliotecas
Gerada automaticamente, mediante os dados fornecidos pelo(a) autor(a)

J95a Júnior, Edilson Alves de Andrade
Uma análise do impacto das linguagens de programação nos custos de execução no AWS Lambda em cenários de cold start e warm start / Edilson Alves de Andrade Júnior. - 2023.
41 f. : il.

Orientador: Robson Wagner Albuquerque de Medeiros.
Inclui referências.

Trabalho de Conclusão de Curso (Graduação) - Universidade Federal Rural de Pernambuco, Bacharelado em Ciência da Computação, Recife, 2023.

1. Computação em nuvem. 2. Gerenciamento de custos na nuvem. 3. Serverless. 4. AWS Lambda. I. Medeiros, Robson Wagner Albuquerque de, orient. II. Título

CDD 004

MINISTÉRIO DA EDUCAÇÃO E DO DESPORTO
UNIVERSIDADE FEDERAL RURAL DE PERNAMBUCO (UFRPE)
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

http://www.bcc.ufrpe.br

Ficha de Aprovação do Trabalho de Conclusão de Curso

Trabalho defendido por Edilson Alves de Andrade Júnior às 14 horas e 00 minutos do dia 24 de abril de 2023, no link https://meet.google.com/mud-idzx-egc, como requisito para conclusão do curso de Bacharelado em Ciência da Computação da Universidade Federal Rural de Pernambuco, intitulado "Uma Análise do Impacto das Linguagens de Programação nos Custos de Execução no AWS Lambda em Cenários de Cold Start e Warm Start", orientado por Robson Wagner Albuquerque de Medeiros e aprovado pela seguinte banca examinadora:

Robson Wagner Albuquerque de Medeiros
DC/UFRPE

Érica Teixeira Gomes de Sousa
DC/UFRPE

Dedico este trabalho aos meus pais, Edilson Alves e Vilma Lopes.

Agradecimentos

A Deus, que me guiou em todos os momentos e me deu forças para seguir em frente com fé e perseverança.

Aos meus pais, Edilson Alves e Vilma Lopes, que sempre me apoiaram em todas as decisões da minha vida. Razões do meu viver.

Às minhas irmãs, Camila Lopes e Érica Patrícia, que me ensinaram o valor da união e do amor incondicional.

Aos meus sobrinhos, Pedro Vitor e Maria Cecilia, que sempre me trazem alegria.

Ao meu cunhado, Carlos Eduardo, por ser um grande amigo presente em todos os momentos.

À minha grande companheira, Rayanne Gusmão, que me apoiou em todas as fases deste trabalho acadêmico e da nossa jornada, sempre me incentivando e me motivando.

Ao meu amigo, Marcelino Chagas, que esteve comigo desde o primeiro dia da graduação, dividindo momentos de descontração.

A todos os professores da UFRPE, que contribuíram significativamente para a minha formação acadêmica, especialmente aqueles que me guiaram e me ensinaram de forma mais direta.

Ao corpo de funcionários da UFRPE, em especial à Sandra Xavier, por todo apoio prestado durante a graduação.

Por fim, ao meu orientador, Robson Medeiros, que esteve sempre presente, me orientando e me incentivando em todas as etapas deste trabalho. Sua orientação foi fundamental para que eu pudesse alcançar os resultados almejados.

"O maior inimigo do conhecimento não é a ignorância, é a ilusão do conhecimento."
(Stephen Hawking)

Resumo

Soluções de computação em nuvem pública ganharam destaque no mercado por oferecer grandes vantagens sobre os sistemas *on-premises*, mas o gerenciamento de fluxos de trabalho baseados na nuvem também traz preocupações. Assim como os problemas relacionados à segurança da informação e à falta de profissionais qualificados, a gestão de custos é um dos principais desafios enfrentados por usuários e organizações que migram ou já possuem suas operações na nuvem. Os provedores de nuvem definem variáveis que afetam diretamente nos comportamentos dos custos, além disso, fatores como as características fundamentais das linguagens de programação também podem contribuir com mudanças nesses comportamentos. Este trabalho teve como objetivo entender como as linguagens de programação se comportam em serviços na nuvem como o AWS Lambda, para que o gerenciamento de custos seja realizado de forma mais assertiva e eficiente, contribuindo diretamente com a redução de custos e desperdícios financeiros ao utilizar esse tipo de serviço. Os resultados evidenciaram que as características das linguagens de programação interferem significativamente nos custos financeiros de execução, elucidando que a escolha de uma determinada linguagem de programação deve ser considerada quando o custo é um requisito a ser atendido na utilização do AWS Lambda.

Palavras-chave: Computação em nuvem, gerenciamento de custos na nuvem, *serverless*, AWS Lambda.

Abstract

Public cloud computing solutions have gained visibility on the market for offering great advantages over on-premises systems. However, cloud-based management workflows also brings concerns. As well as problems related to information security and lack of skilled professionals, cost management is one of the main challenges faced by users and organizations that migrate or already have their operations on cloud. Cloud providers define variables that directly affect cost behaviors, in addition, factors such as key characteristics of programming languages can also contribute to change those behaviors. This work aimed to understand how programming languages behave in cloud services such as AWS Lambda, so that cost management is carried out more assertively and efficiently, directly contributing to the reduction of costs and financial waste when using this kind of service. The results showed that the characteristics of programming languages significantly interfere in the financial costs of execution, elucidating that the choice of a certain programming language should be considered when cost is a requirement to be met when using AWS Lambda.

Palavras-chave: Cloud computing, cloud cost management, serverless, AWS Lambda.

Lista de figuras

Figura 1 – Modelos de serviço e níveis de responsabilidade 17
Figura 2 – Fluxo de execução da máquina de estados 24
Figura 3 – Arquitetura do *framework* de testes (SPF) 25
Figura 4 – Médias dos tempos de execução no AWS Lambda 26
Figura 5 – Fluxograma de execução dos testes 27
Figura 6 – Diagrama da API de testes 29
Figura 7 – *Logs* no AWS CloudWatch 30
Figura 8 – Médias dos tempos de inicialização em cenário de *cold start* 33
Figura 9 – Médias dos tempos de faturamento em cenário de *cold start* 34
Figura 10 – Médias dos tempos de faturamento em cenário de *warm start* 35
Figura 11 – Simulação de custo mensal em cenário de *cold start* 36
Figura 12 – Simulação de custo mensal em cenário de *warm start* 37

Lista de tabelas

Tabela 1 – Linguagens de programação (*runtimes*) e versões 31
Tabela 2 – Ambiente com arquitetura de processamento x86_64 32
Tabela 3 – Ambiente com arquitetura de processamento arm64 32

Lista de abreviaturas e siglas

AoT	Ahead of Time
API	Application Programming Interface
ARM	Advanced RISC Machine
AWS	Amazon Web Services
CLI	Command Line Interface
CPU	Central Processing Unit
HTTP	Hypertext Transfer Protocol
IaaS	Infrastructure as a Service
JVM	Java Virtual Machine
PaaS	Platform as a Service
SaaS	Software as a Service
S3	Simple Storage Service
SAM	Serverless Application Model
SDK	Software Development Kit
SNS	Simple Notification Service
SO	Sistema Operacional
SPF	Serverless Performance Framework
TI	Tecnologia da Informação
UFC	Universidade Federal do Ceará
UFPB	Universidade Federal da Paraíba
UFRPE	Universidade Federal Rural de Pernambuco
VM	Virtual Machine
YAML	YAML Ain't Markup Language

Sumário

1 INTRODUÇÃO 12
1.1 Problema de pesquisa 13
1.2 Justificativa 13
1.3 Objetivos 14
1.3.1 Objetivo geral 14
1.3.2 Objetivos específicos 14
1.4 Estrutura do trabalho 14

2 FUNDAMENTAÇÃO TEÓRICA 15
2.1 Computação em nuvem 15
2.1.1 Características 15
2.1.2 Modelos de serviço 16
2.1.3 Modelos de implantação 18
2.2 Computação *serverless* 19
2.2.1 AWS Lambda 19
2.2.1.1 Linguagens de programação e *runtimes* 19
2.2.1.2 Arquiteturas de processamento 20
2.2.1.3 Casos de uso 20
2.2.1.4 Modelo de precificação 21
2.2.1.5 *Cold start* e *warm start* 21
2.2.2 AWS SAM 21

3 METODOLOGIA 23
3.1 Revisão bibliográfica 23
3.2 Hipóteses 28
3.3 Experimento 28
3.3.1 Linguagens de programação 31
3.3.2 Algoritmo 31
3.3.3 Ambientes 32

4 RESULTADOS 33
4.1 Cenário de *cold start* 33
4.2 Cenário de *warm start* 34
4.3 Simulação de custos 36

5 CONCLUSÃO 38

5.1 Trabalhos futuros 39

REFERÊNCIAS 40

1 Introdução

A computação em nuvem apresenta um paradigma de entrega de serviços bastante atraente, pois elimina ou reduz a necessidade de um maior planejamento sobre o provisionamento e gerenciamento de recursos de TI, principalmente no início de novos negócios. Ela permite algumas mudanças significativas na forma como produtos digitais, ou não, podem ser projetados e desenvolvidos (TAURION, 2009a). Com a computação em nuvem, em vez de comprar e manter *data centers* e servidores físicos, indivíduos e organizações podem ter acesso sob demanda a um *pool* compartilhado de recursos de TI hospedados e totalmente escaláveis (SUNYAEV, 2020), pagando apenas conforme o uso de tais recursos, semelhante ao que ocorre com serviços públicos básicos como fornecimento de água e energia elétrica.

Soluções de computação em nuvem pública ganharam destaque no mercado por oferecer grandes vantagens sobre os sistemas tradicionais, mas o gerenciamento de fluxos de trabalho baseados na nuvem também traz preocupações. Assim como os problemas relacionados à segurança da informação e à falta de profissionais qualificados, a gestão de custos é um dos principais desafios enfrentados por usuários e organizações que migram ou já possuem suas operações na nuvem (FLEXERA, 2022a). O desperdício em gastos com a nuvem é um problema importante e se torna mais crítico à medida que os custos da nuvem aumentam. As organizações estimam que 32% dos gastos com nuvem são desperdiçados, é necessário pensar em maneiras de otimizar a gestão de custos na nuvem de forma mais inteligente e assertiva (FLEXERA, 2022b).

Os principais provedores de computação em nuvem pública, AWS[1], Google Cloud[2] e Microsoft Azure[3], apresentam modelos de precificação de seus serviços baseados em diversos fatores. Esses modelos definem o comportamento dos custos envolvidos em cada serviço, variando conforme o tipo de utilização e finalidade. Os provedores definem variáveis que afetam diretamente nesses comportamentos, desde o número de requisições a uma API até o número de instâncias de leitura e escrita em um banco de dados. Além disso, fatores externos aos de domínio dos provedores de computação em nuvem, como as características fundamentais das linguagens de programação, podem contribuir direta ou indiretamente com a alteração de algumas dessas variáveis.

No AWS Lambda, um serviço computacional *serverless* que permite a execução de funções em diversas linguagens de programação, o custo é calculado conforme a duração total da execução do código, quantidade de memória alocada em cada execução,

[1] https://aws.amazon.com
[2] https://cloud.google.com
[3] https://azure.microsoft.com

quantidade de memória temporária para armazenamento, quantidade de solicitações e arquitetura de processamento utilizada, entre x86_64 e arm64 (ARM). Além disso, fatores sobre formas de provisionamento e a região escolhida também devem ser considerados (AMAZON, 2023a).

Este trabalho tem como objetivo avaliar se a escolha da linguagem de programação pode interferir no custo final de execução de funções no AWS Lambda. Entender como linguagens de programação se comportam em serviços como esse permite que o gerenciamento de custos na nuvem seja realizado de forma mais assertiva e eficiente, contribuindo diretamente com a redução de custos e desperdícios financeiros.

1.1 Problema de pesquisa

Este trabalho tem como principal objetivo responder ao problema de pesquisa definido a seguir:

- A escolha de uma determinada linguagem de programação interfere significativamente nos custos financeiros de execução de funções no serviço AWS Lambda?

1.2 Justificativa

O modelo de computação encontrado no AWS Lambda possui algumas características bem peculiares de operação. Compreender o funcionamento do ciclo de vida, desde a criação de uma aplicação até a etapa de execução, é primordial para o uso adequado e eficiente dos serviços que dispõem desse paradigma.

O autor (SILVA, 2018) realizou uma avaliação de desempenho visando investigar as etapas de pré-execução de funções no AWS Lambda. Os resultados obtidos contribuem para o entendimento do cenário de *cold start*, fenômeno que impacta diretamente na latência final da execução de funções.

Já os autores (JACKSON; CLYNCH, 2018) apresentaram o *design* e a implementação de uma estrutura de teste para análise das métricas de desempenho e custos financeiros nos serviços AWS Lambda e Azure Functions. O trabalho contribui no entendimento sobre o impacto da escolha de *runtimes*, das linguagens de programação, nesses serviços.

A autora (MOREIRA, 2020) avaliou as linguagens Python e Go em cenários de aplicações de computação de alto desempenho no AWS Lambda. Como resultado, o trabalho evidenciou desempenho superior na utilização da linguagem Python em comparação à Go quando há utilização de bibliotecas de terceiros. Já quando é considerado apenas o uso de recursos das próprias linguagens, Go apresentou melhor desempenho em relação à linguagem Python.

Os trabalhos citados contribuem para a compreensão de importantes características relacionadas ao modelo de computação encontrado no AWS Lambda e de algumas linguagens de programação executadas nesse serviço. Com base nessas informações, é crucial verificar se essas características podem interferir nos custos financeiros.

Ter conhecimento prévio, sobre os custos envolvidos no uso de serviços na nuvem, é de extrema importância para o sucesso de qualquer negócio, ajudando diretamente no planejamento, controle e tomada de decisões sobre situações operacionais e principalmente financeiras das organizações, reduzindo ou até mesmo eliminando custos desnecessários. Os benefícios decorrentes de um bom gerenciamento dos custos, assim como os problemas ocasionados por negligenciá-lo, são estendidos a todas as etapas do ciclo de vida de uma aplicação, desde o seu planejamento até a fase de execução (MARIN, 2012).

1.3 Objetivos

1.3.1 Objetivo geral

Analisar se a escolha de uma determinada linguagem de programação interfere significativamente nos custos financeiros de execução no AWS Lambda.

1.3.2 Objetivos específicos

1. Fornecer métricas sobre a execução de funções, em diferentes cenários, ambientes e linguagens de programação, no AWS Lambda.
2. Identificar pontos positivos e negativos, inerentes às características das linguagens de programação analisadas, quando executadas no AWS Lambda.
3. Possibilitar a definição de um modelo de cálculo dos custos envolvidos na execução de funções no AWS Lambda.

1.4 Estrutura do trabalho

No Capítulo 2 são apresentados os conceitos básicos e relevantes ao entendimento e desenvolvimento deste trabalho. No Capítulo 3 é detalhada a metodologia aplicada neste trabalho. No Capítulo 4 são expostos os resultados obtidos após a execução das etapas descritas na metodologia. No Capítulo 5 são realizadas as considerações finais e relacionados alguns pontos que podem ser abordados em trabalhos futuros.

2 Fundamentação teórica

Este capítulo apresenta conceitos básicos essenciais para o entendimento e desenvolvimento deste trabalho. Na Seção 2.1 são abordadas as definições de computação em nuvem, assim como suas principais características, modelos de serviço e implantação. Já na Seção 2.2 é apresentado o conceito de computação *serverless*, destacando pontos importantes sobre o serviço AWS Lambda e o *framework* AWS SAM.

2.1 Computação em nuvem

Há várias definições para o termo computação em nuvem, algumas delas se destacam por abordar pontos e aspectos em comum, como o compartilhamento de recursos de TI, escalabilidade, forma de acesso e provisionamento. A seguir, são apresentadas algumas definições que são amplamente aceitas.

Para (TAURION, 2009b), computação em nuvem é um termo que descreve um ambiente de computação baseado em uma grande rede de servidores, virtuais ou físicos, com disponibilização de recursos de TI, como capacidade de processamento, armazenamento de dados, redes, plataformas, aplicações e serviços, tudo via Internet.

Já (MELL; GRANCE, 2011) definem computação em nuvem como um modelo que permite acesso simplificado, conveniente e sob demanda a conjuntos de recursos de computação, incluindo servidores, redes, armazenamento, aplicativos e serviços. Tais recursos podem ser gerenciados de forma fácil e com o mínimo possível de intermediação dos provedores de serviços.

AWS (AMAZON, 2023d), Google Cloud (GOOGLE, 2023) e Microsoft Azure (MICROSOFT, 2023), três dos principais provedores de computação em nuvem pública, definem computação em nuvem como um modelo de serviços digitais e recursos de TI, sob demanda, por meio da Internet. Além disso, enfatizam o pagamento conforme o uso.

2.1.1 Características

As definições sobre computação em nuvem estão quase sempre ligadas às características essenciais desse modelo, e são essas mesmas características que diferenciam a computação em nuvem de outros paradigmas de computação. A seguir são destacadas as características essenciais, segundo (MELL; GRANCE, 2011), da computação em nuvem.

1. Autoatendimento sob demanda: Um usuário pode alocar recursos de computação de forma autônoma, com base em suas necessidades, sem qualquer intervenção humana

como provedores de serviços ou qualquer tipo de intermediário.

2. Amplo acesso à rede: Os recursos e serviços devem ser acessíveis de qualquer local e por variados dispositivos habilitados para Internet, como *tablets*, *smartphones*, *notebooks* ou estações de trabalho.

3. Agrupamento de recursos: Os provedores de serviços em nuvem utilizam um modelo multilocatário para agrupar seus recursos de computação e atender vários usuários, onde os recursos físicos e virtuais são alocados dinamicamente e reatribuídos com base na demanda. Os usuários geralmente não têm controle ou consciência da localização precisa desses recursos, mas podem especificar alguns detalhes com um nível maior de abstração, como país, estado ou *data center*. Os recursos disponíveis incluem largura de banda de rede, memória, processamento e armazenamento.

4. Elasticidade rápida: Os recursos disponíveis podem ser alocados e desalocados de forma elástica, com alguns casos sendo automatizados, para aumentar ou diminuir rapidamente conforme a demanda. Do ponto de vista do usuário, os recursos disponíveis para alocação podem parecer ilimitados e podem ser obtidos em qualquer quantidade a qualquer momento.

5. Serviço mensurado: Os sistemas em nuvem empregam uma capacidade de medição em um nível de abstração adequado ao tipo de serviço oferecido, como processamento, armazenamento, largura de banda e contas de usuários ativos, para gerenciar e otimizar automaticamente a utilização de recursos. É possível monitorar, controlar e reportar o uso dos recursos, aumentando a transparência tanto para o provedor quanto para o usuário do serviço.

2.1.2 Modelos de serviço

A computação em nuvem possui três modelos de serviços, são eles: SaaS, PaaS e IaaS (MELL; GRANCE, 2011).

SaaS é o modelo de serviço em que *softwares* são disponibilizados aos usuários através da Internet, como um serviço, sem que seja necessário instalá-los em computadores ou servidores locais. Nesse modelo, o provedor do serviço é responsável por manter o *software* atualizado e em funcionamento, além de garantir a segurança e a disponibilidade dos dados. Os usuários acessam o *software* através de uma interface *web* ou de aplicativos específicos, utilizando apenas um navegador ou dispositivo conectado à Internet.

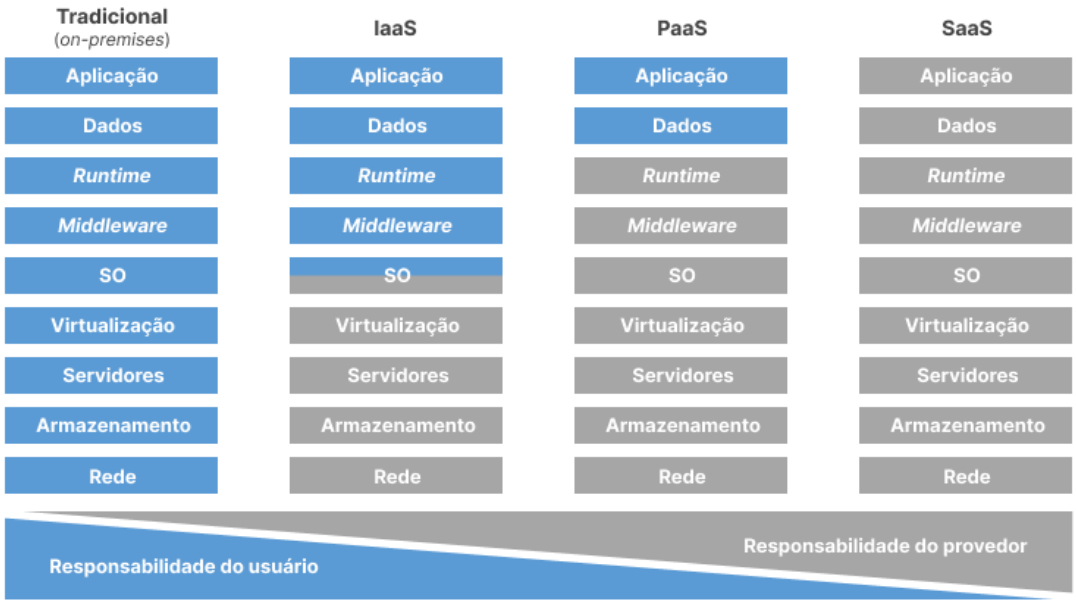
PaaS é o modelo de serviço em que uma plataforma de *software* é oferecida como serviço pela internet. Nele, o provedor do serviço oferece uma plataforma completa de desenvolvimento e execução de aplicativos, com recursos como linguagens de programação,

bancos de dados, SDKs, CLIs e SOs. Os usuários podem utilizar as plataformas desse modelo para criar, testar, implantar e gerenciar seus próprios aplicativos, sem a necessidade de se preocuparem com a infraestrutura necessária para suportá-los.

Já IaaS é o modelo de serviço que permite que empresas e usuários finais aluguem recursos de infraestrutura de TI, como servidores, armazenamento, redes e outros recursos computacionais, de um provedor de serviços em nuvem. Ao optar por um serviço IaaS, os usuários finais têm acesso a recursos de infraestrutura escaláveis e flexíveis, que podem ser configurados e gerenciados de forma remota. Isso significa que os usuários não precisam se preocupar com manutenções de *hardware* e outros recursos, pois tudo é gerenciado pelo provedor de serviços em nuvem.

A Figura 1 relaciona os três modelos citados, mais o modelo tradicional *on-premises*, comparando em níveis de responsabilidade de gerenciamento entre os usuários e os provedores de serviços em nuvem.

Figura 1 – Modelos de serviço e níveis de responsabilidade



Os níveis de responsabilidade de gerenciamento aumentam conforme o modelo de serviço. No modelo tradicional, mais à esquerda, não há provedor de nuvem e todos os níveis são gerenciados pelo usuário, como em infraestruturas de TI locais. À medida que se avança, da esquerda para a direita, cada modelo de serviço aumenta os níveis de responsabilidade de gerenciamento dos provedores de serviços em nuvem e, consequentemente, reduz esses níveis para os usuários.

2.1.3 Modelos de implantação

Os modelos de implantação em nuvem são as diferentes formas pelas quais as soluções de computação em nuvem podem ser implantadas e disponibilizadas para os usuários finais. Eles podem ser classificados em nuvem pública, nuvem privada, nuvem comunitária e nuvem híbrida (MELL; GRANCE, 2011).

Nuvem pública é um modelo de implantação onde os serviços são oferecidos por provedores de nuvem pública, como AWS, Google Cloud e Microsoft Azure. Nesse modelo, os recursos computacionais, como servidores, armazenamento e aplicativos, são compartilhados entre diferentes organizações e indivíduos que os alugam por meio da Internet. O acesso aos recursos é geralmente escalável e elástico, o que significa que os usuários podem aumentar ou diminuir sua capacidade conforme as necessidades do negócio. Os provedores de nuvem pública geralmente cobram pelos serviços com base no consumo, oferecendo uma opção mais acessível para empresas que não têm orçamento ou recursos para manter seus próprios *data centers* e infraestrutura de TI.

Nuvem privada é um modelo operado exclusivamente para uma única organização. Ao contrário da nuvem pública, onde os recursos são compartilhados entre várias organizações, a nuvem privada fornece uma infraestrutura dedicada que pode ser gerenciada internamente pela organização ou por um provedor de serviços de nuvem privada. Isso pode oferecer maior controle sobre a segurança, conformidade e desempenho da nuvem, mas também pode exigir um investimento mais significativo em recursos e gerenciamento. A nuvem privada pode ser implantada nas instalações da organização ou em uma infraestrutura de nuvem privada externa.

Já no modelo de nuvem comunitária, a infraestrutura é compartilhada por várias organizações com interesses em comum, como requisitos de segurança, conformidade regulatória ou necessidades de aplicativos específicos. A nuvem comunitária é gerenciada e utilizada por essas organizações, em vez de ser aberta ao público. Ela pode ser hospedada interna ou externamente e pode ser gerenciada pelas organizações que a utilizam ou por um terceiro provedor de serviços. A nuvem comunitária oferece maior controle e privacidade em comparação com a nuvem pública, mas pode não ser tão escalável e acessível quanto a nuvem privada ou pública.

Por fim, a nuvem híbrida combina elementos de duas ou mais nuvens distintas (pública, privada ou comunitária). Nesse modelo, as empresas usam serviços de nuvens distintas, em conjunto, para atender a diferentes requisitos de negócios. Em um exemplo prático, pode ser feito o uso da nuvem pública para serviços de uso geral, como armazenamento de arquivos, enquanto a nuvem privada é usada para processamento de dados altamente sensíveis. A nuvem híbrida permite que as empresas maximizem a eficiência operacional, reduzam os custos de TI e, simultaneamente, mantenham o controle sobre

dados críticos e confidenciais.

2.2 Computação *serverless*

Serverless é um paradigma de computação em nuvem que possibilita a criação e execução de aplicativos sem a necessidade de gerenciamento de servidores ou infraestrutura por parte dos desenvolvedores. Nesse modelo, a responsabilidade pelo gerenciamento da infraestrutura é transferida para o provedor de nuvem, permitindo que os desenvolvedores foquem na lógica do aplicativo. A execução do código é desencadeada por eventos específicos, como solicitações HTTP, envio de mensagens em filas ou atualizações em bancos de dados, ao invés de ser executada continuamente em um servidor. Algumas das implementações mais populares de serviços *serverless* incluem AWS Lambda e Google Cloud Functions.

2.2.1 AWS Lambda

O AWS Lambda é um serviço de computação *serverless* onde os usuários podem escrever funções em diversas linguagens de programação e executá-las em um ambiente altamente escalável e resiliente, sem precisar se preocupar com a configuração ou gerenciamento de servidores.

2.2.1.1 Linguagens de programação e *runtimes*

O AWS Lambda conta com suporte nativo à execução de funções desenvolvidas nas linguagens de programação C# (.NET), Go, Java, JavaScript (Node.js), Python e Ruby. Além disso, permite a criação de *runtimes* personalizados para que funções em outras linguagens também sejam contempladas (AMAZON, 2023c), concedendo ao usuário um maior poder de escolha a partir de características que tornam as linguagens de programação únicas e adequadas para diferentes tipos de tarefas e aplicações. Uma dessas características está relacionada à forma de como o código-fonte é traduzido em código de máquina, criando duas grandes categorias de linguagens de programação, as compiladas e as interpretadas.

Em resumo, linguagens compiladas são aquelas onde o código-fonte é transformado em código de máquina pelo compilador, que é um programa específico destinado a essa tarefa. O código resultante é executado diretamente pelo SO ou pelo processador, sem necessidade de interpretação adicional. Exemplos de linguagens compiladas, com suporte nativo para execução no AWS Lambda, incluem: C# (.NET), Java e Go. Já as linguagens interpretadas são aquelas onde o código-fonte é executado por um interpretador, que é um programa que lê o código-fonte e executa cada instrução à medida que as traduz

para código de máquina. O AWS Lambda conta com suporte nativo a algumas linguagens interpretadas, como: JavaScript (Node.js), Python e Ruby.

2.2.1.2 Arquiteturas de processamento

As seguintes arquiteturas de processamento são disponibilizadas no AWS Lambda: x86_64, sendo a arquitetura padrão para processadores baseados em x86 de 64 *bits*, e arm64, sendo a arquitetura de 64 *bits* para o processador AWS Graviton (ARM).

Segundo a AWS (AMAZON, 2023b), funções que utilizam a arquitetura arm64 geram menores custos de execução quando comparados aos de funções equivalentes executadas em uma CPU baseada em x86. Contudo, funções que dependem do conjunto de instruções da CPU baseada em x86, necessitam da criação de um novo pacote de implantação para a arquitetura arm64, com dependências recompiladas para o conjunto de instruções ARM.

2.2.1.3 Casos de uso

O AWS Lambda é amplamente utilizado em situações que necessitam de processamento de eventos em tempo real, integrações de aplicativos, automação de tarefas e desenvolvimento de aplicativos *serverless* em geral, como:

1. Processamento de eventos em tempo real: O AWS Lambda pode ser usado para processar eventos gerados em tempo real a partir de outros serviços da AWS, como o Amazon S3, Amazon Kinesis, Amazon DynamoDB, Amazon API Gateway e Amazon SNS.

2. Integração de aplicativos: O AWS Lambda pode ser usado para integrar aplicativos e serviços em nuvem, como o Salesforce, Slack, Zendesk, entre outros.

3. Processamento de mídia: O AWS Lambda pode ser usado para processar e transformar mídias em diferentes formatos, tamanhos e resoluções, para atender às necessidades de diferentes dispositivos e plataformas.

4. Automação de tarefas: O AWS Lambda pode ser usado para automatizar tarefas de rotina, como *backups* de bancos de dados, limpeza de arquivos antigos, entre outras.

5. Desenvolvimento de aplicativos *serverless*: O AWS Lambda pode ser usado como uma plataforma para desenvolver e implantar aplicativos *serverless*, onde a infraestrutura e a escalabilidade são gerenciadas automaticamente pela plataforma.

2.2.1.4 Modelo de precificação

O modelo de precificação do AWS Lambda é estabelecido com base na duração total da execução do código, quantidade de memória alocada em cada execução, quantidade de memória temporária para armazenamento, quantidade de solicitações e arquitetura de processamento utilizada, entre x86_64 e arm64 (ARM). Além disso, fatores sobre formas de provisionamento e a região escolhida também devem ser considerados. Mensalmente, a AWS disponibiliza 1 milhão de invocações ou até 3,2 milhões de segundos de computação gratuitamente (*free tier*), logo, os custos começam a ser contabilizados apenas após o esgotamento desse benefício (AMAZON, 2023a).

O AWS Lambda registra uma solicitação toda às vezes que uma determinada função é executada em resposta a invocações, como eventos de notificação com o Amazon SNS, *cron jobs* com o Amazon EventBridge e chamadas HTTP com o Amazon API Gateway.

A duração é calculada a partir do momento em que o código começa a ser executado até seu término ou retorno, arredondando o tempo de faturamento para o 1 ms mais próximo. A capacidade da CPU e outros recursos são alocados proporcionalmente conforme a memória alocada para execução, um aumento no tamanho da memória aciona um aumento correspondente na CPU disponível para a função.

2.2.1.5 *Cold start* e *warm start*

Quando uma função no AWS Lambda é invocada, o tempo que leva para a função ser executada pela primeira vez é conhecido como *cold start*. Isso ocorre porque o ambiente de execução precisa ser inicializado, incluindo o provisionamento de recursos e a carga do código da função na memória. Por outro lado, quando a mesma função é invocada novamente, é possível que o ambiente de execução ainda esteja ativo e a função possa ser iniciada mais rapidamente, sem a necessidade de inicializar novamente o ambiente. Isso é conhecido como *warm start*.

A diferença entre a *cold start* e a *warm start* pode ser significativa em termos de tempo de resposta da função. É importante considerar essa diferença ao projetar e dimensionar arquiteturas de aplicativos *serverless*.

2.2.2 AWS SAM

O AWS SAM é um *framework* de código aberto que ajuda a criar aplicativos *serverless* na nuvem da AWS. Ele fornece uma sintaxe simplificada para definir recursos comuns, como funções no AWS Lambda, APIs no Amazon API Gateway, tabelas no Amazon DynamoDB e *buckets* no Amazon S3. Com o AWS SAM, é possível definir,

testar e implantar facilmente aplicativos *serverless*, permitindo que os desenvolvedores se concentrem no código em vez da infraestrutura subjacente.

O provisionamento da infraestrutura necessária para a execução dos testes realizados neste trabalho, assim como a implantação de todos os códigos, foi realizada utilizando o AWS SAM.

3 Metodologia

Neste capítulo são abordados todos os aspectos metodológicos deste trabalho, descrevendo os procedimentos necessários e úteis na análise do impacto de algumas linguagens de programação nos custos de execução no AWS Lambda em cenários de *cold start* e *warm start*.

3.1 Revisão bibliográfica

Uma revisão bibliográfica foi realizada para construção deste trabalho. As buscas foram efetuadas em fontes que incluem o Google Scholar[1] e sites de busca que levaram aos repositórios de trabalhos acadêmicos da UFC[2] e UFPB[3]. Nas fontes citadas, foram realizadas consultas por meio de termos de pesquisa relevantes para o tema abordado, como computação em nuvem, *cloud computing*, gerenciamento de custos na nuvem, *serverless*, AWS Lambda e linguagens de programação.

Com base em uma pesquisa exploratória, com intuito de investigar as características do modelo de computação *serverless*, (SILVA, 2018) realizou um experimento investigando o fenômeno de *cold start* no AWS Lambda. As execuções e avaliações foram realizadas através do serviço AWS Step Functions, possibilitando a utilização de uma máquina de estados responsável por orquestrar um fluxo de invocações a funções. Em cada estado foram criadas tarefas que acionam uma função ou que aguardam um determinado período, definido pelo usuário, para o estado ser alterado.

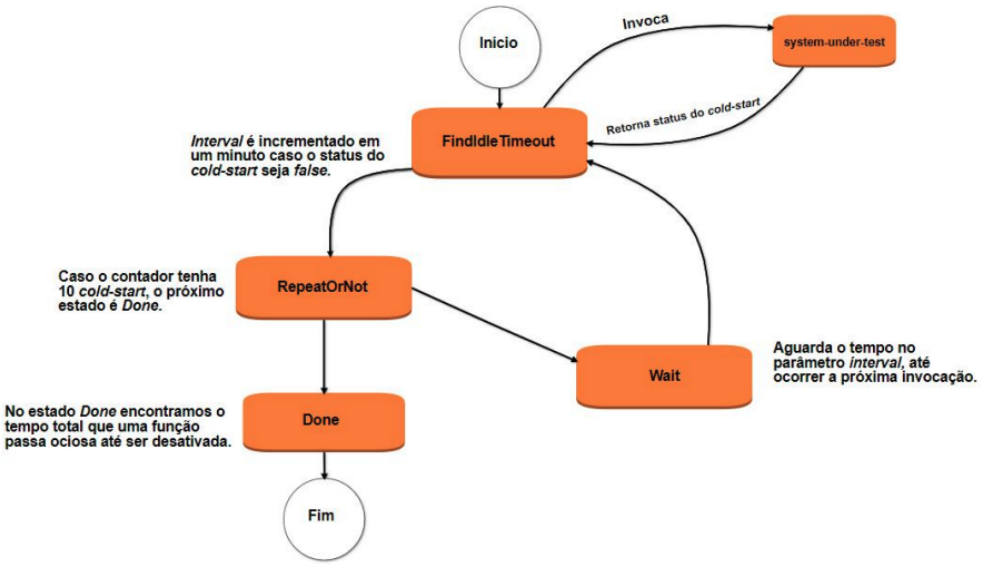
Como ambientes de teste, cinco versões de funções foram criadas, com 128 MB, 256 MB, 512 MB, 1024 MB e 2048 MB de memória alocada para cada. As funções, todas contando com a mesma estrutura de código desenvolvida em JavaScript (Node.js), possuíam uma variável do tipo *boolean*, nomeada de *isColdStart*, inicializada com o valor *false*. Após a chegada da primeira requisição e identificação de *cold start*, o valor da variável era alterado para *true*. Funções no AWS Lambda guardam o estado global enquanto estão ativas, com isso, o valor atribuído à variável *isColdStart* era mantido até ocorrer um novo *cold start*.

A Figura 2 ilustra o fluxo executado nos testes que durou cerca de um dia até encontrar o tempo máximo de ociosidade das funções. O AWS Step Function era responsável por avaliar o retorno da função executada e caso a variável voltasse a retornar *false*, 1 min era incrementado até a próxima execução. Após 10 registros de *cold start*, o

[1] https://scholar.google.com.br
[2] https://repositorio.ufc.br
[3] https://repositorio.ufpb.br

AWS Step Functions alterava o fluxo para o estado *Done*, concluíndo o tempo máximo de ociosidade das funções.

Figura 2 – Fluxo de execução da máquina de estados



Fonte: (SILVA, 2018)

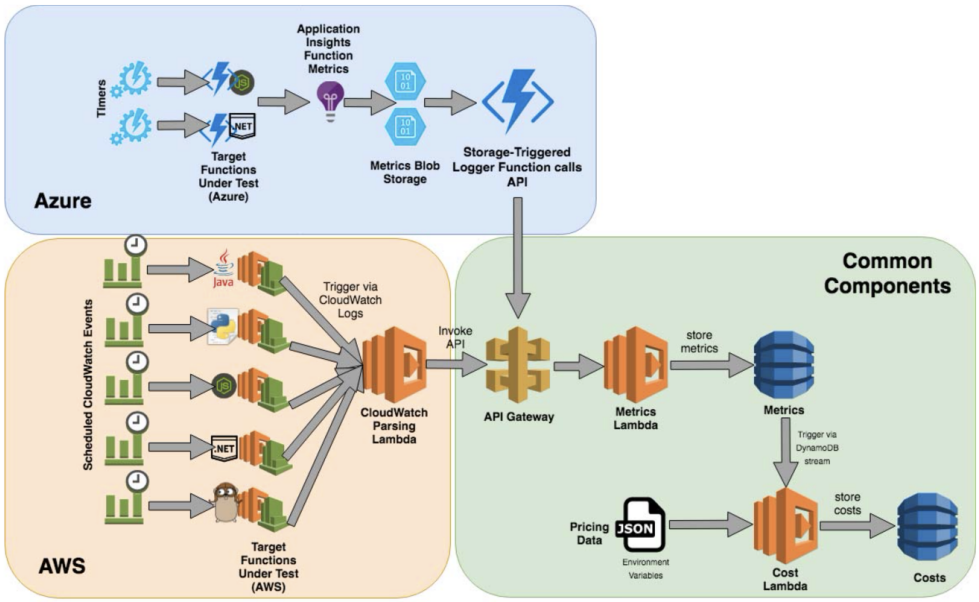
O experimento concluiu que o limite de tempo de ociosidade das funções no AWS Lambda, até serem desativadas, foi entre 30 min e 42 min; evidenciando que aplicações com frequência de requisições abaixo dos 25 min raramente enfrentarão cenários de *cold start*. Além disso, o autor elencou como possível trabalho futuro a realização de experimentos utilizando diferentes linguagens de programação, como os propostos por este trabalho.

Visando entender o impacto da escolha de *runtimes* das linguagens de programação, no desempenho e custo financeiro da execução de funções *serverless*, (JACKSON; CLYNCH, 2018) apresentaram o *design* e a implementação de uma estrutura de teste para análise das métricas de desempenho e custo nos serviços AWS Lambda e Azure Functions. O trabalho contou com a execução de um experimento envolvendo todos os *runtimes* (.NET Core 2, Go, Java, Node.js e Python) disponíveis no AWS Lambda no momento do desenvolvimento (2018), além de dois *runtimes* adicionais (Node.js e .NET C#) no Azure Functions.

Os testes foram projetados em torno de funções sem implementação de código, o objetivo dos autores foi de realizar medições do tempo necessário para que o ambiente de execução das funções fossem configurados pelos serviços. Além disso, uma estrutura de teste foi desenvolvida com a responsabilidade de coletar, de forma automática, as métricas resultantes provenientes dos testes.

A Figura 3 ilustra a arquitetura criada para realização dos testes, com módulos definidos conforme suas respectivas responsabilidades.

Figura 3 – Arquitetura do *framework* de testes (SPF)



Fonte: (JACKSON; CLYNCH, 2018)

- Azure: Módulo com os *runtimes* Node.js e .NET C#. As funções de teste (*Target Functions Under Test*) foram configuradas para integração com o serviço Azure Application Insights (*Application Insights Function Metrics*), responsável pela coleta de *logs* e telemetria. Esses dados foram armazenadas no serviço Azure Blob Storage (*Metrics Blob Storage*) e repassados por meio de uma nova função (*Storage-Triggered Logger Function Calls API*) para a API do módulo *Common Components*.

- AWS: Módulo com os *runtimes* .NET Core 2, Go, Java, Node.js e Python. As funções de teste (*Target Functions Under Test*) foram configuradas para integração com o serviço AWS CloudWatch (*CloudWatch Parsing Lambda*), responsável pela coleta de *logs* e telemetria. Diferente do módulo Azure, no módulo AWS não foi necessário armazenar os dados em um serviço de *storage*, repassando-os diretamente para a API do módulo *Common Components*.

- *Common Components*: Módulo com a estrutura responsável pelo recebimento, processamento e armazenamento dos dados disponibilizados pelos módulos Azure e AWS. Por meio de uma API padrão, que conta com um único *endpoint*, uma função é acionada (*Metrics Lambda*) e armazena as métricas fornecidas em uma tabela no DynamoDB (*Metrics*). A função *Cost Lambda* é acionada, por meio de um evento, combina as métricas recebidas com os dados de preços mais recentes (*Pricing Data*) e calcula o custo estimado de execução dessa função. Os dados de custos são armazenados na tabela *Costs*, também no DynamoDB.

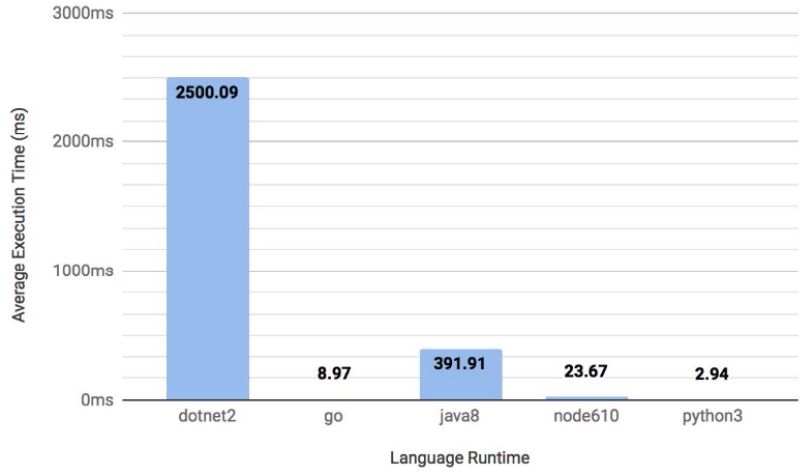
Os testes em cenário de *cold start* foram realizados em um período que chegou a um total de 6 dias, envolvendo 144 invocações a cada função correspondente aos *runtimes*

testados. A estratégia dos autores foi de medir o desempenho em diferentes condições de ambiente que podem ocorrer conforme horário ou dia da semana. Já em cenário de *warm start*, foram realizados 4 testes de 1 h cada, totalizando 248 testes individuais em cada um dos cinco *runtimes*.

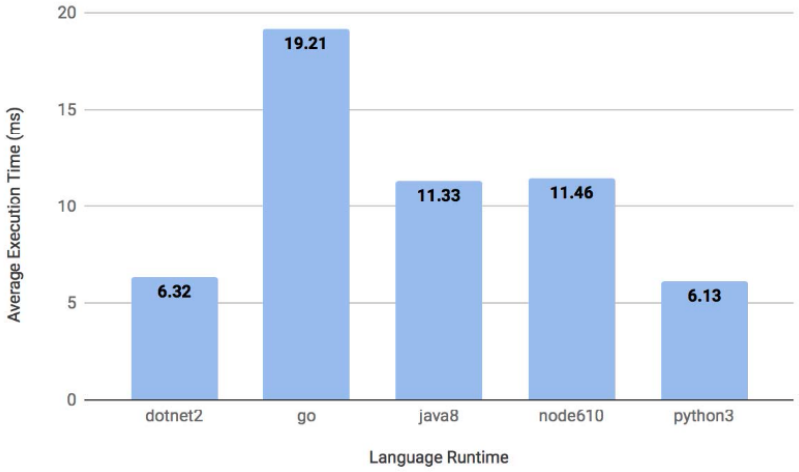
Como este trabalho consiste em uma análise com foco no serviço AWS Lambda, serão apresentados apenas os resultados e conclusões relacionadas aos testes realizados na AWS, conforme as Figuras 4a e 4b.

Figura 4 – Médias dos tempos de execução no AWS Lambda

(a) Resultado em *cold start*



(b) Resultado em *warm start*



Fonte: (JACKSON; CLYNCH, 2018)

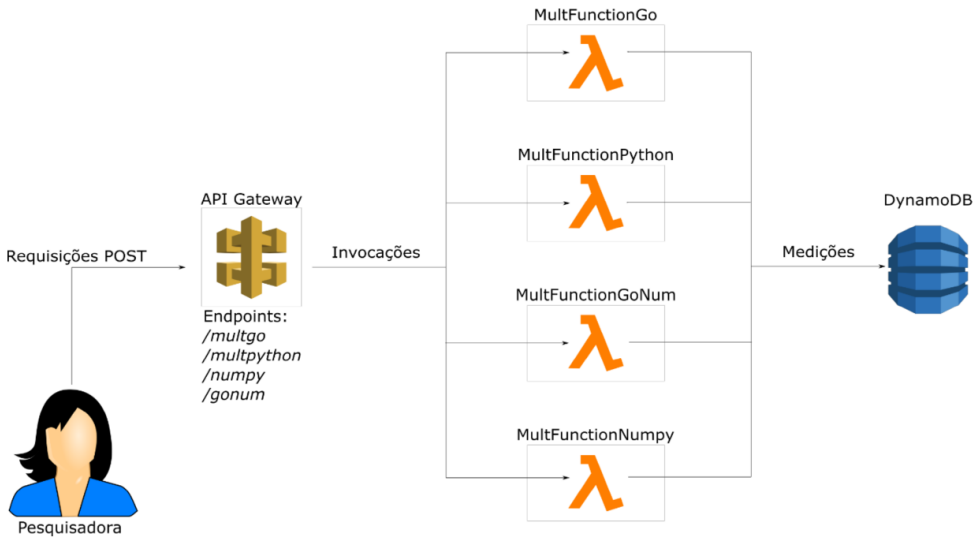
O trabalho concluiu que o *runtime* definido para a linguagem de programação Python é a melhor escolha para uso no AWS Lambda, ao apresentar o menor tempo médio de execução tanto em cenário de *cold start* quanto em cenário de *warm start*, 2,94 ms e 6,13 ms, respectivamente. Além disso, o trabalho expôs um problema na utilização dos *runtimes* .NET Core 2 e Java, ambos com tempos médios de execução bem elevados em cenário de *cold start*, 2500,09 ms e 391,91 ms, respectivamente.

Vale salientar que todos os testes foram realizados em funções que não contavam com implementação de código, os autores ressaltaram que o objetivo foi de medir o desempenho do modelo de computação *serverless*, na criação e execução de *runtimes*, com o mínimo possível de intervenção das linguagens de programação. Além disso, apenas a arquitetura de processamento x86_64 foi contemplada nos testes.

Buscando conhecer os benefícios e limitações do modelo de computação *serverless*, (MOREIRA, 2020) realizou uma análise das linguagens de programação Python e Go em cenários de aplicações de alto desempenho no AWS Lambda. A autora visou responder qual das duas linguagens fornece melhor desempenho em operações envolvendo multiplicação de matrizes serial.

O experimento foi realizado por meio da execução de um algoritmo de multiplicação de matrizes desenvolvido em ambas as linguagens de programação. Como pré-requisito, foram utilizadas matrizes densas preenchidas totalmente com valores de ponto flutuante (1.0). A autora usou o AWS SAM para definir uma API no API Gateway, funções que implementam o algoritmo no AWS Lambda e uma base de dados no DynamoDB responsável por armazenar as informações sobre cada requisição, como o tempo de execução, linguagem de programação utilizada e a ordem da matriz multiplicada. A Figura 5 ilustra o fluxograma de execução dos testes realizados.

Figura 5 – Fluxograma de execução dos testes



Fonte: (MOREIRA, 2020)

Conforme o *endpoint* especificado pelo usuário durante a chamada HTTP, retratada pela pesquisadora, o API Gateway invoca a função correspondente. Utilizando a linguagem de programação Go foram desenvolvidas duas funções, a *MultFunctionGo* implementa o algoritmo de multiplicação de matrizes nativamente e a *MultFunctionGoNum* com auxílio da biblioteca Gonum[4]. Semelhantemente, foram desenvolvidas duas funções

[4] https://github.com/gonum/gonum

utilizando a linguagem de programação Python, *MultFunctionPython* implementando o algoritmo de multiplicação de matrizes nativamente e *MultFunctionNumpy* através da biblioteca NumPy[5].

As requisições foram executadas com os seguintes parâmetros: linguagem, ordem, memória e tempo máximo. Por exemplo, uma requisição configurada como (python, 100, 2GB, 6min) representa a execução da multiplicação de uma matriz 1000x1000, utilizando a função *MultFunctionPython*, 2 GB de memória RAM e limite máximo de execução de 6 min.

Como resultado, nos cenários onde as execuções contaram apenas com o uso das bibliotecas padrão de cada linguagem de programação, a linguagem Go obteve resultados superiores aos da linguagem Python, ou seja, menores tempos de execução. Já em cenários em que bibliotecas externas foram utilizadas, a linguagem Python, com auxílio da biblioteca NumPy, obteve um melhor desempenho nos tempos de execução.

3.2 Hipóteses

Através das revisões descritas na seção anterior, algumas hipóteses foram levantadas:

1. A escolha de uma determinada linguagem de programação, para execução de funções no AWS Lambda, interfere significativamente nos custos financeiros.

2. Linguagens de programação compiladas possuem um melhor desempenho, quando executadas no AWS Lambda, resultando em menores tempos de execução e consequentemente menores custos financeiros.

3.3 Experimento

Foram desenvolvidas duas etapas de experimentação, com a primeira contemplando execuções em um cenário de *cold start* e a segunda em um cenário de *warm start*. Em cada etapa, 30 execuções foram realizadas para *endpoints* que apontam para as funções desenvolvidas em cada linguagem de programação descrita na Subseção 3.3.1. As funções receberam o evento de execução e retornaram o valor resultante correspondente ao algoritmo descrito na Subseção 3.3.2. Além disso, cada teste foi realizado em dois ambientes distintos descritos na Subseção 3.3.3, totalizando 720 execuções.

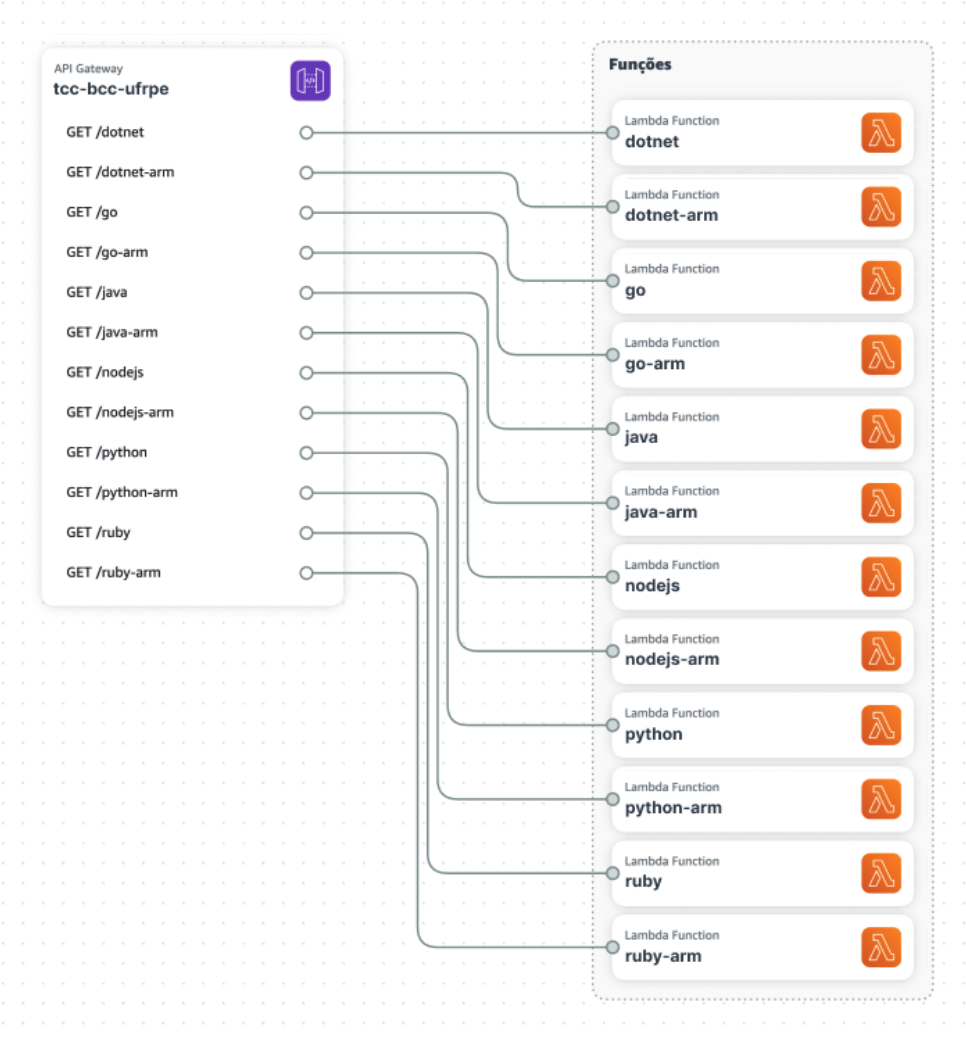
Com conhecimento prévio sobre os tempos de ociosidade das funções no AWS Lambda, descrito no trabalho realizado por (SILVA, 2018), e buscando evitar a espera

[5] https://github.com/numpy/numpy

pela desativação das funções testadas, foram forçadas reimplantações através do SAM. Essa ação desliga as funções de imediato e cria o cenário ideal para testes em *cold start*. Já para os testes em cenário de *warm start*, a primeira execução de cada função foi ignorada, sendo considerados apenas os resultados das execuções subsequentes, quando as funções já estão ativadas.

A estrutura de código desenvolvida para realização do experimento, correspondente ao diagrama apresentado na Figura 6, assim como os *logs* contendo as métricas geradas durante os testes e as estimativas de custos calculadas, podem ser acessados através do seguinte repositório público: <https://github.com/edilsonalves/tcc-bcc-ufrpe>.

Figura 6 – Diagrama da API de testes



As funções foram testadas para vários termos (n) diferentes, de 1 até 30, sendo uma execução para cada termo (n), totalizando 30 execuções. Os valores foram passados via *query string*, conforme a seguinte estrutura:

```
api_gateway_invoke_url/resource?term=n
```

Exemplo com a linguagem JavaScript (Node.js) para o trigésimo termo (n) da Sequência de Fibonacci:

```
api_gateway_invoke_url/nodejs?term=30
```

O valor correspondente à `api_gateway_invoke_url` é disponibilizado após a etapa implantação do código, na AWS, pelo serviço API Gateway.

As métricas resultantes das execuções foram coletadas através do serviço AWS CloudWatch (Logs Insights), a Figura 7 demonstra como os *logs* de *report* podem ser dispostos.

Figura 7 – *Logs* no AWS CloudWatch

#	@duration	@billedDuration	@initDuration	@memorySize	@maxMemoryUsed
▶ 1	911.91	912.0	209.71	1.28E8	6.1E7
▶ 2	949.99	950.0	235.99	1.28E8	6.1E7
▶ 3	915.0	916.0	234.46	1.28E8	6.1E7
▶ 4	890.04	891.0	248.18	1.28E8	6.1E7
▶ 5	894.4	895.0	222.59	1.28E8	6.1E7
▶ 6	856.57	857.0	225.56	1.28E8	6.1E7
▶ 7	911.46	912.0	311.04	1.28E8	6.1E7
▶ 8	894.25	895.0	251.41	1.28E8	6.1E7
▶ 9	900.28	901.0	221.22	1.28E8	6.1E7
▶ 10	869.52	870.0	233.55	1.28E8	6.1E7

Algumas outras informações podem ser obtidas através dos *logs* coletados pelo AWS Cloud Watch, mas para o objeto de estudo deste trabalho as principais métricas a serem observadas são:

- *Duration*: Tempo, em milissegundo (ms), necessário para o processamento do evento.
- *Billed duration*: Tempo de faturamento da invocação arredondado para o 1 ms mais próximo (valor usado no cálculo do custo).
- *Memory size*: Quantidade de memória, em *megabyte* (MB), alocada para a função.
- *Max memory used*: Quantidade máxima de memória, em *megabyte* (MB), usada pela função durante a execução.
- *Init duration*: Tempo de inicialização, em milissegundo (ms), para a primeira solicitação atendida (cenário de *cold start*). Esse valor só é incorporado ao tempo de faturamento caso ultrapasse 10 s (MAHANTY, 2022). Logo, é possível ocorrer execuções com tempo de faturamento menor que o de inicialização.

3.3.1 Linguagens de programação

As linguagens de programação foram escolhidas visando contemplar diferentes características, separando-as em grupos descritos como: compiladas e interpretadas. Discussões inerentes a essa classificação não são abordadas neste trabalho. Além disso, essas são as linguagens de programação oficialmente suportadas pelo serviço AWS Lambda, sem a necessidade da criação de *runtimes* personalizados.

Tabela 1 – Linguagens de programação (*runtimes*) e versões

	Linguagem	**Versão**
Compiladas	C# (.NET)	6.0
	Go	1.20
	Java	11
Interpretadas	JavaScript (Node.js)	18.x
	Python	3.9
	Ruby	2.7

3.3.2 Algoritmo

O algoritmo usado no experimento consiste no cálculo do enésimo termo da Sequência de Fibonacci. Trata-se de uma sequência infinita de números, em que cada um de seus valores é o resultado da soma dos dois anteriores. Sendo esses os primeiros números da sequência (OEIS, 2022):

`0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, …`

Na matemática, a Sequência de Fibonacci é definida recursivamente pela seguinte fórmula:

`F(n) = F(n-1) + F(n-2), com F(0) = 0 e F(1) = 1`

E a própria definição da Sequência de Fibonacci pode ser tomada como base para implementar um algoritmo que retorna o enésimo termo da sequência, como o seguinte:

```
function fibonacci(n)
  if n <= 1 then
    return n
  else
    return fibonacci(n - 1) + fibonacci(n - 2)
```

3.3.3 Ambientes

As implantações das funções e ambientes foram realizadas através da CLI do *framework* SAM, que permite a criação de aplicações *serverless* através de uma sintaxe simplificada e definições de *templates* em YAML, atribuindo as funções aos seus devidos ambientes.

Para realização dos testes, foram definidos 2 ambientes conforme tabelas abaixo:

Tabela 2 – Ambiente com arquitetura de processamento x86_64

Região	**Arquitetura**	**Memória**
Oregon (us-west-2)	x86_64	128 MB

Tabela 3 – Ambiente com arquitetura de processamento arm64

Região	**Arquitetura**	**Memória**
Oregon (us-west-2)	arm64	128 MB

Os ambientes foram criados na região de Oregon (us-west-2), contando com 128 MB de memória alocada para execução e arquiteturas de processamento x86_64 e arm64.

4 Resultados

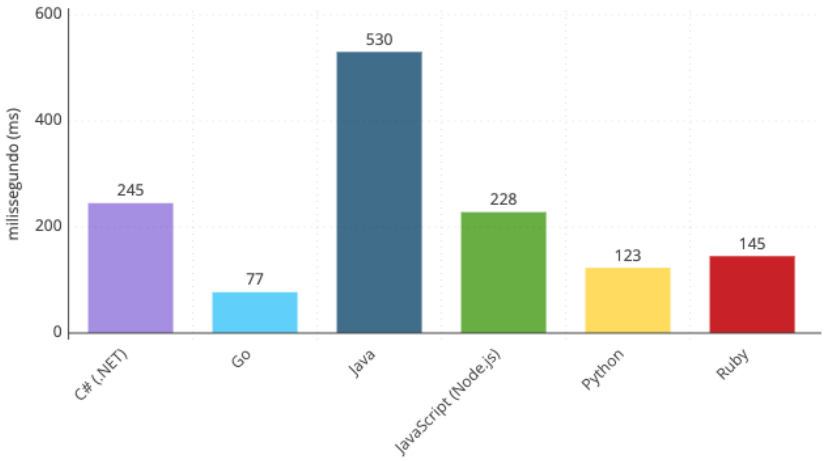
Este capítulo apresenta os resultados organizados em seções que contemplam os cenários de *cold start* e *warm start*, além de uma simulação de custos. Os valores correspondem às médias, dispensando casas decimais, para os tempos de inicialização e faturamento.

4.1 Cenário de *cold start*

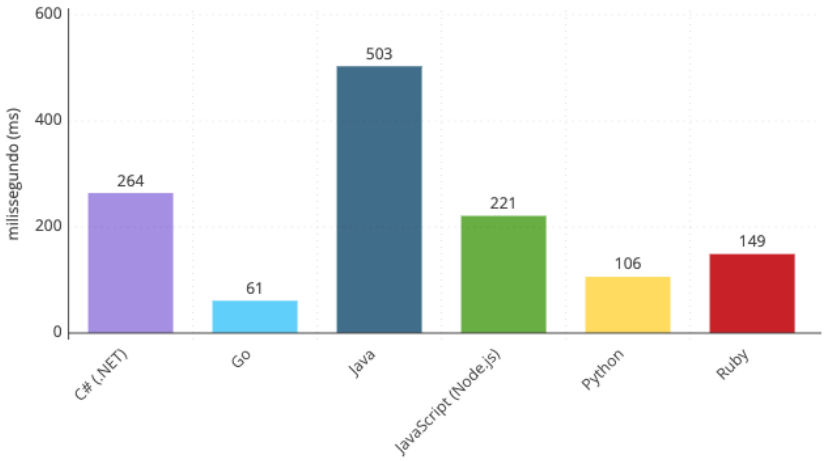
As funções executadas com linguagens de programação que utilizam camadas intermediárias de *software*, como C# com o .NET e Java com a JVM, foram notoriamente impactadas. Essas funções apresentaram médias dos tempos de inicialização maiores que 245 ms. As Figuras 8a e 8b ilustram bem os resultados das médias dos tempos de inicialização.

Figura 8 – Médias dos tempos de inicialização em cenário de *cold start*

(a) Resultado em ambiente com arquitetura de processamento x86_64



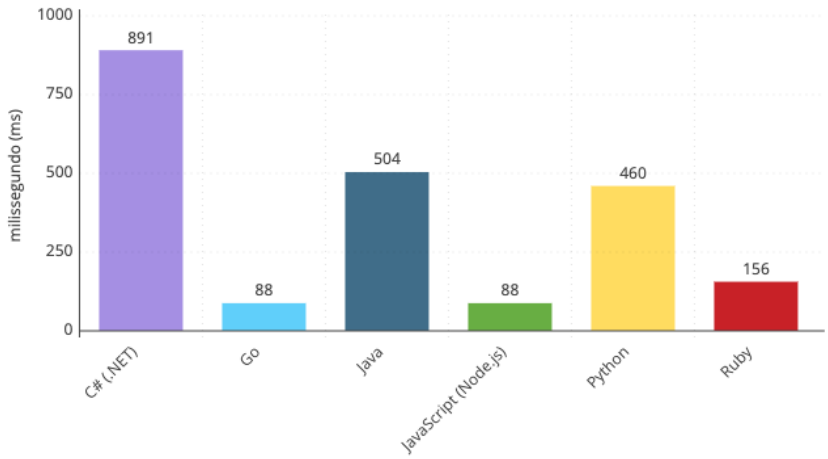
(b) Resultado em ambiente com arquitetura de processamento arm64



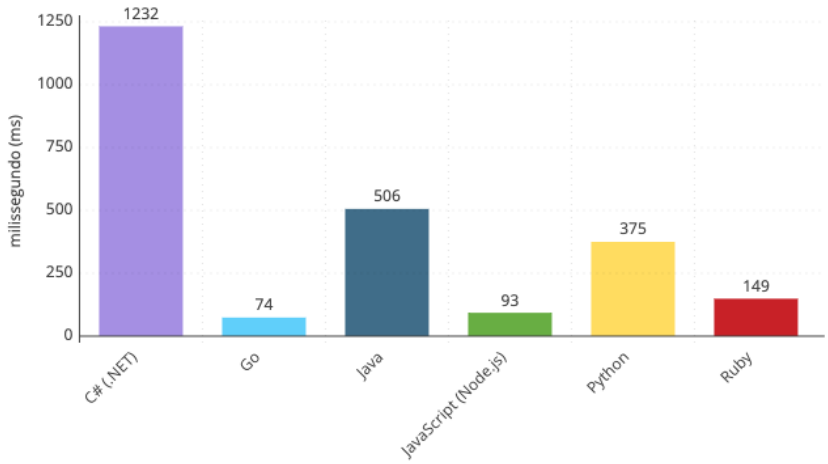
As médias dos tempos de faturamento, em cenário de *cold start*, foram mais favoráveis às linguagens interpretadas, muito devido aos péssimos resultados das funções executadas em C# (.NET) e Java. Ainda assim, a função em Python não obteve médias satisfatórias, com valores acima dos 375 ms. Os resultados estão apresentados nas Figuras 9a e 9b.

Figura 9 – Médias dos tempos de faturamento em cenário de *cold start*

(a) Resultado em ambiente com arquitetura de processamento x86_64



(b) Resultado em ambiente com arquitetura de processamento arm64



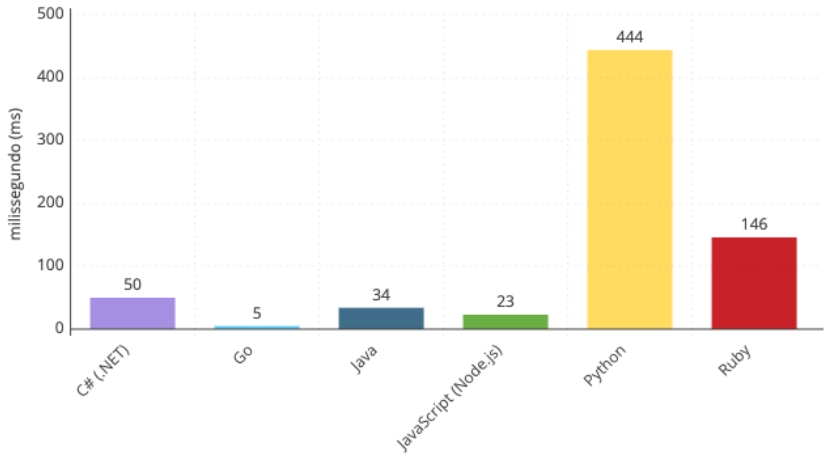
Das linguagens compiladas, apenas Go se destacou positivamente, obtendo as menores médias incluindo todas as outras linguagens, 88 ms em ambiente com arquitetura de processamento x86_64 e 74 ms em ambiente com arm64.

4.2 Cenário de *warm start*

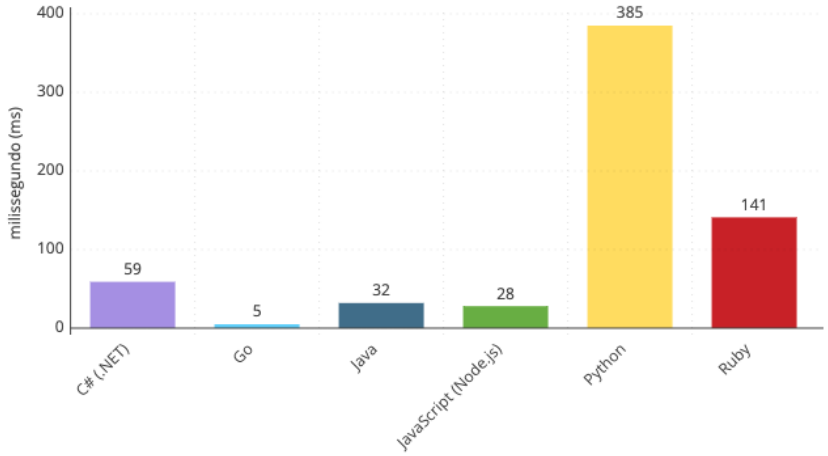
Em cenário de *warm start* os resultados foram mais favoráveis às linguagens compiladas, como mostram as Figuras 10a e 10b.

Figura 10 – Médias dos tempos de faturamento em cenário de *warm start*

(a) Resultado em ambiente com arquitetura de processamento x86_64



(b) Resultado em ambiente com arquitetura de processamento arm64



Go se destacou por obter médias que não passaram dos 5 ms. C# (.NET) e Java melhoraram seus respectivos tempos médios de faturamento, quando comparados aos obtidos em cenário de *cold start*. C# (.NET) com médias dos tempos de faturamento abaixo dos 59 ms e Java com médias abaixo dos 34 ms, uma redução que chega a aproximadamente 95% e 93%, respectivamente. Ainda assim, JavaScript (Node.js) alcançou o segundo melhor resultado (23 ms e 28 ms), perdendo apenas para Go. A linguagem Ruby obteve tempos medianos (146 ms e 141 ms) e Python o pior dos resultados, com tempos médios de faturamento maiores que a soma dos obtidos por todas as outras linguagens, 444 ms em ambiente com arquitetura de processamento x86_64 e 385 ms em ambiente com arm64.

4.3 Simulação de custos

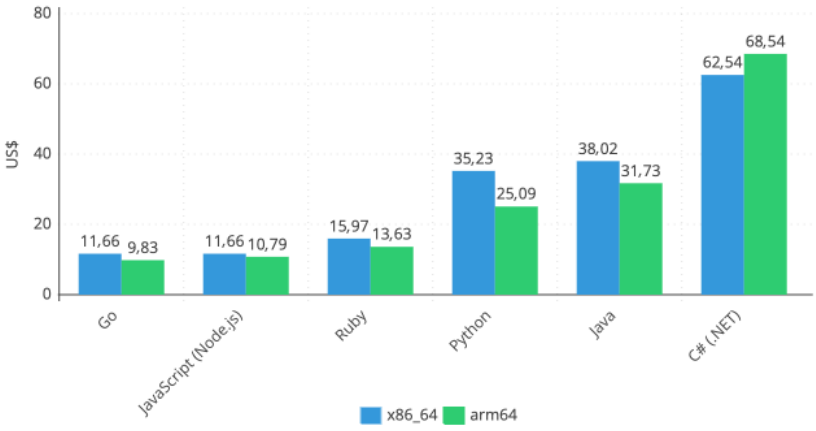
Para simulação de custos foi utilizado o serviço AWS Pricing Calculator[1], uma calculadora para estimativa de custos disponibilizada gratuitamente pela AWS. Os cálculos são realizados considerando a definição de preço do AWS Lambda (AMAZON, 2023a).

As estimativas foram realizadas respeitando alguns pontos:

- Foram utilizados os mesmos tempos de faturamento apresentados nos gráficos das Seções 4.1 e 4.2 deste mesmo capítulo.
- Foi considerada a mesma região utilizada nos ambientes de teste (us-west-2).
- Foram consideradas as mesmas arquiteturas de processamento utilizadas nos ambientes de teste (x86_64 e arm64).
- Foi considerada a mesma quantidade de memória alocada nos ambientes de teste (128 MB).
- Foram desconsiderados os benefícios do nível gratuito[2] da AWS, que inclui uma quantidade considerável de solicitações e de tempo de computação por mês para o AWS Lambda.
- A quantidade de armazenamento temporário alocada escolhida foi de 512 MB, essa quantidade não implica em custo adicional.
- A quantidade de solicitações escolhida foi de 1.000.000 por dia.

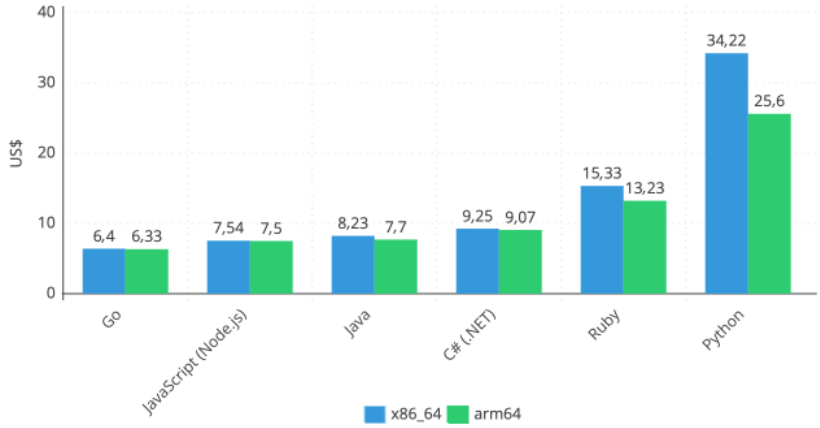
As Figuras 11 e 12 apresentam os resultados da simulação de custo mensal nos cenários de *cold start* e *warm start*.

Figura 11 – Simulação de custo mensal em cenário de *cold start*



[1] https://calculator.aws
[2] https://aws.amazon.com/pt/free

Figura 12 – Simulação de custo mensal em cenário de *warm start*



Da esquerda para a direita, os resultados estão ordenados da linguagem que apresentou os menores custos mensais, para a linguagem que apresentou os maiores custos. Os valores evidenciam uma maior economia para execuções de funções, no AWS Lambda, com a linguagem de programação Go.

Em cenário de *cold start*, Go apresentou custos mensais de US$ 11,66 em ambiente com arquitetura de processamento x86_64 e US$ 9,83 em ambiente com arm64. Os custos são ainda menores em cenário de *warm start*, onde a linguagem apresentou custos mensais de US$ 6,40 em ambiente com arquitetura de processamento x86_64 e US$ 6,33 em ambiente com arm64.

Os resultados apresentados neste capítulo foram de grande importância para identificar que as características das linguagens de programação interferem significativamente nos custos financeiros de execução, portanto, a escolha de uma determinada linguagem de programação deve ser considerada quando o custo é um requisito a ser atendido na utilização do AWS Lambda.

5 Conclusão

A proposta deste trabalho foi de realizar uma análise do impacto das linguagens de programação nos custos de execução de funções no AWS Lambda. Para tal, foram explorados os cenários de *cold start* e *warm start* em dois 2 ambientes que se distinguem pela arquitetura de processamento utilizada.

Os resultados coletados são de extrema importância para elucidar o objetivo geral deste trabalho e trazer respostas às hipóteses levantadas através das revisões bibliográficas. Além disso, contribuem com o entendimento de como funções no AWS Lambda se comportam em cenários e ambientes específicos quando executadas com diferentes linguagens de programação.

Ficou evidente que a escolha de uma determinada linguagem de programação deve ser considerada quando o custo é um requisito a ser atendido. Na simulação de custos realizada, Go se mostrou a mais barata para execução de funções no AWS Lambda em todos os cenários e ambientes. Em cenário de *cold start*, por exemplo, os custos de execução com a linguagem Go mostraram uma economia superior a 80% quando comparados aos custos com a linguagem C# (.NET). E em cenário de *warm start*, uma economia superior a 75% quando comparados aos custos com a linguagem Python.

Com relação à hipótese de que linguagens de programação compiladas possuem melhor desempenho no tempo de execução e, consequentemente, menor custo financeiro, isso não é verdade. Em cenário de *cold start*, as linguagens C# (.NET) e Java apresentaram resultados piores que todas as linguagens interpretadas, resultando nos maiores custos de execução. E em cenário de *warm start*, C# (.NET) e Java também tiveram desempenhos inferiores ao da linguagem JavaScript (Node.js).

Outro ponto respondido está relacionado à diferença de custo entre execuções usando arquitetura de processamento x86_64 e arm64. Apenas um dos resultados se mostrou desfavorável à utilização da arquitetura arm64, o da linguagem C# (.NET) em cenário de *cold start*. Em alguns resultados, algumas linguagens apresentaram menor tempo de faturamento no ambiente com arquitetura de processamento x86, porém com menor custo no ambiente com arquitetura de processamento arm64. Esse detalhe pode ser percebido nos resultados da linguagem Java em cenário de *cold start*, por exemplo. A linguagem apresentou médias dos tempos de faturamento de 504 ms e 506 ms, nas arquiteturas de processamento x86_64 e arm64, respectivamente, mas os custos finais foram de US$ 38,02 em x86_64 e US$ 31,73 em arm64.

5.1 Trabalhos futuros

Em novembro de 2022 a AWS anunciou o suporte nativo à compilação AoT, permitindo que códigos em C# (.NET) sejam compilados antecipadamente para binários nativos, visando *cold starts* até 86% mais rápidas em comparação aos tempos de execução gerenciados pela versão 6 do .NET (BESWICK, 2022). Para o Java, há a possibilidade da utilização de outra VM como a GraalVM, que possibilita a compilação AoT em um executável otimizado contendo tudo o que é necessário para execução. Isso possibilita obter tempos mais rápidos de inicialização em comparação com a JVM tradicional (WANG, 2021).

Visando o entendimento de como essas melhorias destacadas se comportariam, é de extrema importância realizar uma nova análise contemplando essas mudanças na forma como códigos em C# (.NET) e Java são compilados.

Referências

AMAZON. *Definição de preço do AWS Lambda*. 2023. Disponível em: <https://aws.amazon.com/pt/lambda/pricing>. Acesso em: 15 de fev. 2023. Citado 3 vezes nas páginas 13, 21 e 36.

AMAZON. *Lambda instruction set architectures*. 2023. Disponível em: <https://docs.aws.amazon.com/lambda/latest/dg/foundation-arch>. Acesso em: 15 de fev. 2023. Citado na página 20.

AMAZON. *Lambda runtimes*. 2023. Disponível em: <https://docs.aws.amazon.com/lambda/latest/dg/lambda-runtimes>. Acesso em: 15 de fev. 2023. Citado na página 19.

AMAZON. *O que é a computação em nuvem?* 2023. Disponível em: <https://aws.amazon.com/pt/what-is-cloud-computing>. Acesso em: 15 de fev. 2023. Citado na página 15.

BESWICK, J. *Building serverless .NET applications on AWS Lambda using .NET 7*. 2022. Disponível em: <https://aws.amazon.com/pt/blogs/compute/building-serverless-net-applications-on-aws-lambda-using-net-7>. Acesso em: 15 de fev. 2023. Citado na página 39.

FLEXERA. *State of the Cloud Report*. [S.l.]: Flexera, 2022. 40-44 p. Citado na página 12.

FLEXERA. *State of the Cloud Report*. [S.l.]: Flexera, 2022. 45-50 p. Citado na página 12.

GOOGLE. *O que é a computação em nuvem?* 2023. Disponível em: <https://cloud.google.com/learn/what-is-cloud-computing?hl=pt-br>. Acesso em: 15 de fev. 2023. Citado na página 15.

JACKSON, D.; CLYNCH, G. *An Investigation of the Impact of Language Runtime on the Performance and Cost of Serverless Functions*. [S.l.]: 2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion), 2018. 154-160 p. Citado 4 vezes nas páginas 13, 24, 25 e 26.

MAHANTY, K. *AWS Lambda function and its valuable nuances*. 2022. Disponível em: <https://awstip.com/aws-lambda-function-and-its-valuable-nuances-15434642a4b2>. Acesso em: 15 de fev. 2023. Citado na página 30.

MARIN, L. A. P. *Custos em projetos de TI: Como Estimar, Determinar e Controlar Custos para Projetos de Softwares*. São Paulo, Brasil: Universidade Presbiteriana Mackenzie, 2012. 17-18 p. Citado na página 14.

MELL, P.; GRANCE, T. *The NIST Definition of Cloud Computing*. Gaithersburg, EUA: Special Publication (NIST SP), National Institute of Standards and Technology, 2011. Citado 3 vezes nas páginas 15, 16 e 18.

MICROSOFT. *O que é computação em nuvem?* 2023. Disponível em: <https://azure.microsoft.com/pt-br/resources/cloud-computing-dictionary/what-is-cloud-computing>. Acesso em: 15 de fev. 2023. Citado na página 15.

MOREIRA, R. de S. *Avaliação das Linguagens Python e Go na Plataforma AWS Lambda para Aplicações de Computação de Alto Desempenho*. Quixadá, Brasil: Universidade Federal do Ceará, 2020. Citado 2 vezes nas páginas 13 e 27.

OEIS. *Fibonacci numbers*. 2022. Disponível em: <https://oeis.org/A000045>. Acesso em: 15 de fev. 2023. Citado na página 31.

SILVA, M. N. da. *Análise de Mecanismos de Serverless Computing em Ambientes de Nuvens Computacionais*. Rio Tinto, Brasil: Universidade Federal da Paraíba, 2018. Citado 4 vezes nas páginas 13, 23, 24 e 28.

SUNYAEV, A. *Internet Computing: Principles of Distributed Systems and Emerging Internet-Based Technologies*. 1. ed. [S.l.]: Springer International Publishing, 2020. 195 p. Citado na página 12.

TAURION, C. *Cloud Computing: Computação em Nuvem: Transformando o Mundo da Tecnologia da Informação*. 1. ed. Rio de Janeiro, Brasil: Brasport, 2009. 6-7 p. Citado na página 12.

TAURION, C. *Cloud Computing: Computação em Nuvem: Transformando o Mundo da Tecnologia da Informação*. 1. ed. Rio de Janeiro, Brasil: Brasport, 2009. 2 p. Citado na página 15.

WANG, Z. *GraalVM Native Image Support no AWS SDK for Java 2.x*. 2021. Disponível em: <https://aws.amazon.com/pt/blogs/developer/graalvm-native-image-support-in-the-aws-sdk-for-java-2-x>. Acesso em: 15 de fev. 2023. Citado na página 39.