PONTIFÍCIA UNIVERSIDADE CATÓLICA DE GOIÁS
ESCOLA POLITÉCNICA E DE ARTES
GRADUAÇÃO EM ENGENHARIA DE COMPUTAÇÃO

DESENVOLVIMENTO DE UM SISTEMA DISTRIBUIDO COM DIFERENTES LINGUAGENS DE PROGRAMAÇÃO PARA COMPARAR O ATRASO DO *COLD START* EM APLICAÇÕES *SERVERLESS*

OMAR DE ARAUJO LOPES

GOIÂNIA
2024

OMAR DE ARAUJO LOPES

DESENVOLVIMENTO DE UM SISTEMA DISTRIBUIDO COM DIFERENTES LINGUAGENS DE PROGRAMAÇÃO PARA COMPARAR O ATRASO DO *COLD START* EM APLICAÇÕES *SERVERLESS*

Trabalho de Conclusão de Curso apresentado à Escola Politécnica e de Artes, da Pontifícia Universidade Católica de Goiás, como parte dos requisitos para a obtenção do título de Bacharel em Engenharia de Computação.

Orientador: Me. Alexandre Ribeiro

GOIÂNIA

2024

OMAR DE ARAUJO LOPES

DESENVOLVIMENTO DE UM SISTEMA DISTRIBUIDO COM DIFERENTES LINGUAGENS DE PROGRAMAÇÃO PARA COMPARAR O ATRASO DO *COLD START* EM APLICAÇÕES *SERVERLESS*

Trabalho de Conclusão de Curso aprovado em sua forma final pela Escola Politécnica e de Artes, da Pontifícia Universidade Católica de Goiás, para obtenção do título de Bacharel em Engenharia de Computação, em 22 de junho de 2024.

_____

Prof. Ma. Ludmilla Reis Pinheiro dos Santos

Coordenador de Trabalho de Conclusão de Curso

Banca examinadora:

_____

Orientador Prof. Me. Alexandre Ribeiro

_____

Prof. Me. André Luiz Alves

_____

Prof. Me. Max Gontijo de Oliveira

GOIÂNIA

2024

AGRADECIMENTOS

Agradeço, sobretudo, ao meu Deus por ter me conduzido até aqui.

À minha mãe, pelo carinho e amor incondicional, pelo cuidado, apoio e força que me inspirou a nunca desistir. Por ter comprado, em 2012, meu primeiro computador, mesmo sem condições financeiras, ascendendo a fagulha que queima até hoje.

À minha primeira família, em especial meu padrasto, meus irmãos, cunhados e sobrinhas.

À minha segunda família, em especial meus sogros, cunhados e sobrinha.

Aos meus amigos, que me acompanharam em toda essa jornada, em especial aqueles que me ajudaram na elaboração deste trabalho: Lara, Lucas e Ana Carolina.

Ao meu orientador e amigo Alexandre Ribeiro, por todos os ensinamentos, que não se limitaram às fronteiras da faculdade, pela paixão a disseminação de conhecimento, por ter me feito perceber que fiz a escolha certa desde o primeiro dia de aula na universidade, por ser referência de profissional e ser-humano.

À minha esposa Isadora, por ter me acompanhado durante toda essa jornada, por ter tornado isso tudo possível. Pelas conceções, pela compreensão habitual, por ter os ouvidos abertos sempre que precisei, pela inspiração diária e exemplo de determinação.

A todos os meus mais sinceros agradecimentos.

“ Quem não vê outro objetivo no jogo além de dar xeque-mate, nunca será um bom jogador.”

- Max Euwe

RESUMO

A computação em nuvem já se consolidou como uma poderosa ferramenta utilizada no mercado e entre os serviços oferecidos, a computação *serverless,* que é uma arquitetura de *software* disposta numa plataforma de computação em nuvem que provê um ambiente de hospedagem e de execução, tem se destacado pela forma otimizada de consumo de recursos computacionais, utilizando-os apenas quando há demanda e liberando-os quando não são necessários. Além disso, abstrai a complexidade de configurações de infraestruturas para os desenvolvedores, permitindo que seus esforços se concentrem na solução dos problemas. É um conceito recente e tem ganhado notória popularidade, com previsões de crescimento significativo até 2025. No entanto, enfrenta desafios, como o problema do *cold start,* que é o atraso decorrente da inicialização de uma instancia em serveless, que causa prejuízos no processamento das aplicações. Desse modo, o objetivo desse estudo é examinar empiricamente como as diferentes linguagens de programação influenciam a intensidade do *cold start* em aplicações *serverless*. Para isso, foi desenvolvido um sistema distribuído, implementado em diferentes linguagens de programação. Foram conduzidos experimentos computacionais no sistema desenvolvido, e os dados gerados foram analisados e comparados.

Palavras-Chave: computação *serverless*, *cold start*, computação em nuvem, linguagens de programação.

ABSTRACT

Cloud computing has established itself as a powerful tool utilized in the market, and among the services offered, serverless computing— a software architecture deployed on a cloud computing platform that provides a hosting and execution environment—has stood out for its optimized consumption of computational resources, using them only when there is demand and releasing them when they are not needed. Additionally, it abstracts the complexity of infrastructure configurations for developers, allowing their efforts to focus on problem-solving. It is a recent concept and has gained notable popularity, with significant growth forecasts until 2025. However, it faces challenges, such as the cold start problem, which is the delay caused by the initialization of an instance in serverless, leading to processing drawbacks for applications. Thus, the objective of this study is to empirically examine how different programming languages influence the intensity of cold start in serverless applications. To this end, a distributed system was developed and implemented in different programming languages. Computational experiments were conducted on the developed system, and the generated data were analyzed and compared.

Keywords: Serverless computing, cold start, cloud computing, programming language.

LISTA DE FIGURAS

LISTA DE QUADROS

LISTA DE SIGLAS

AWS – *Amazon Web Services*

HTTP – *Hypertext Transfer Protocol*

YAML – *yet another markup language*

IDE - *Integrated Development Environment*

API - *Application Programming Interface*

JSON - *JavaScript Object Notation*

NOP – Número da Ordem de Processamento

CLI – *Command line interface*

SAM – *Serverless Application Model*

VSCODE – Visual Studio Code

C/S – Cliente-Servidor

SUMÁRIO

1 INTRODUÇÃO

A computação em nuvem é uma abordagem revolucionária para fornecimento de infraestrutura de tecnologia da informação que transformou a maneira como indivíduos e organizações gerenciam, armazenam e acessam seus dados e aplicações. Suas vantagens incluem redução de custos financeiros, escalabilidade, acessibilidade, flexibilidade e confiabilidade (SANTOS et al., 2023).

Além disso, ela oferece serviços como computação *serverless* que utiliza os recursos computacionais de forma mais otimizada, consumindo-os apenas enquanto houver demanda, e quando não há, os recursos são liberados, portanto, não são cobrados.

Apesar de ser um conceito recente, esta abordagem tem atraído a atenção da indústria. Em 2025, espera-se que os gastos com esse serviço atinjam cerca de vinte e dois bilhões de dólares, em comparação aos três bilhões de dólares pagos em 2017. Prevê-se também que cerca de cinquenta por cento das empresas empregará computação *serverless* até 2025 (JINFENG et al., 2023).

No entanto, com a crescente utilização deste método, surgiu também o problema do *cold start*, que implica no atraso das aplicações *serverless* em processar uma demanda computacional, gerando efeitos negativos na funcionalidade deste serviço.

Isso posto, o presente trabalho examinou empiricamente como diferentes linguagens de programação podem influenciar a intensidade do atraso causado pelo problema de *cold start* em aplicações *serverless* hospedadas na nuvem. Com esse objetivo, foi implementado um sistema distribuído no qual uma aplicação é executada localmente e outras são hospedadas na nuvem, utilizando a arquitetura *serverless* da AWS Lambda e se comunicando de forma síncrona via protocolo HTTP.

O critério de escolha de linguagem de programação foi selecionado para ser explorado na gestão do atraso causado pelo *cold start*, pois é a primeira decisão e responsabilidade dos desenvolvedores da aplicação. A decisão sobre o provedor de computação em nuvem pode ser influenciada por fatores que vão além das questões técnicas, como custos ou licenças, por exemplo.

Ao final deste trabalho, têm-se a comparação de desempenho das diferentes linguagens de programação em relação ao atraso provocado pelo *cold start*, visando, de maneira empírica, validar se existe alguma linguagem de programação, das escolhidas para este trabalho, que

tenha desempenho menor ou maior em relação ao atraso provocado pelo *cold start* em aplicações *serverless*.

O presente trabalho está organizado da seguinte forma: o capítulo 2 apresenta o referencial teórico, explicando alguns conceitos preliminares sobre computação em nuvem, computação *serverless*, modelos de execução de linguagem de programação e sistemas distribuídos; O capítulo 3 descreve a metodologia do processo de desenvolvimento do sistema cliente/servidor, das aplicações *serverless* e da aplicação de execução local, e também como foi o processo dos experimentos computacionais para gerar os dados de insumo para análise; o capitulo 4 apresenta os resultados dos experimentos computacionais realizados; e, por fim, o capítulo 5 apresenta as conclusões finais.

1.1 OBJETIVO GERAL

Implementar um sistema distribuído com comunicação HTTP, para comparar o atraso provocado pelo *cold start* em aplicações *serverless* implementadas em diferentes linguagens de programação, evidenciando como sistemas concebidos com o mesmo padrão de solução, porém implementados em linguagens de programação distintas, podem manifestar atrasos decorrentes do "*cold start*" em diferentes magnitudes.

1.2 Objetivos específicos

• Projetar e implementar um sistema distribuído, utilizando comunicação HTTP.

• Implementar um *software* a ser implantado em uma arquitetura *serverless* utilizando AWS Lambda para receber requisições HTTP

• Implementar um *software* que seja executado localmente para produzir requisições HTTP.

• Implementar o *software* que será implantado na nuvem em três modelos de execução de linguagens de programação diferentes, em linguagem compilada, interpretada e mista.

• Comparar os atrasos ocasionados pelo "*cold start*" nas três soluções desenvolvidas.

3 REFERENCIAL TEÓRICO

3.1 Computação em nuvem

Computação em nuvem é uma forma alternativa de distribuição e consumo de recursos de tecnologia da informação (TI), em que tais recursos, sejam eles de hardware ou *software*, são oferecidos sob demanda com pagamento baseado no uso. Surgiu da necessidade de infraestruturas de TI complexas, e é baseada na abstração, que oculta a complexidade de infraestruturas e manutenção. Essa abstração oferece cada parte dessa infraestrutura como serviço, essas estruturas são normalmente alocadas em centros de dados, utilizando hardware compartilhado, tanto para processamento, quanto armazenamento (SOUSA et al., 2009).

A computação em nuvem representa uma alternativa ao emprego de recursos computacionais locais e fisicamente presentes. Por exemplo, considere a situação em que surge a necessidade de processar uma vasta base de dados, demandando poder de processamento e espaço de armazenamento superiores aos disponíveis no momento. Nesse contexto, recorre-se à computação em nuvem, mediante alocação de uma máquina virtual equipada com os recursos necessários. O pagamento, nesse caso, é efetuado somente pelo período de utilização da máquina virtual. Outro cenário ilustrativo envolve a execução ininterrupta de processos, imunes a instabilidades decorrentes de variações na energia elétrica, falhas humanas ou circunstâncias imprevistas. Tal necessidade também pode ser atendida pela computação em nuvem, através de alocação de uma máquina virtual, permitindo a execução contínua do processo. O acesso a esse processo é disponibilizado por meio de conexões seguras e/ou protocolos padronizados.

O *National Institute of Standards and Technology* (NIST) destaca cinco características essenciais de qualquer provedor que oferece soluções de computação em nuvem:

> a) self-service sob demanda: Os usuários podem adquirir recursos computacionais, como processamento e memória, na medida em que necessitem e sem precisar de interação humana com os provedores de cada serviço, inclusive podendo ser configurados e orquestrados automaticamente, de forma transparente para os usuários.
> b) amplo acesso: Recursos são disponibilizados por meio da rede e acessados através de mecanismos padronizados que possibilitam o uso por diferentes fontes.
> c) pooling de recursos: Os recursos computacionais do provedor são organizados em um pool para servir múltiplos usuários. Estes usuários não precisam ter conhecimento da localização física dos recursos computacionais, podendo somente especificar a localização em um nível mais alto de abstração, tais como o país, estado ou centro de dados.
> d) elasticidade rápida: Recursos podem ser adquiridos de forma rápida e elástica, em alguns casos automaticamente, caso haja a necessidade de escalar com o aumento da demanda, e liberados, na retração dessa demanda. Para os usuários, os recursos disponíveis para uso parecem ser ilimitados e podem ser adquiridos em qualquer quantidade e a qualquer momento.

> e) serviço medido: Sistemas em nuvem automaticamente controlam e otimizam o uso de recursos por meio de uma capacidade de medição. A automação é realizada em algum nível de abstração apropriado para o tipo de serviço, tais como armazenamento, processamento, largura de banda e contas dos usuários ativas. O uso de recursos pode ser monitorado e controlado, possibilitando transparência para o provedor e o usuário do serviço utilizado (SOUSA et al., 2009, tradução nossa).

Sob esse ponto de vista, segundo Bello et al., (2021), a computação em nuvem é uma mudança de paradigma na forma como os recursos de hardware e *software* são gerenciados e utilizados, permitindo que se compartilhem os aspectos físicos e não físicos de uma infraestrutura de TI, tornando-a reutilizável e assim distribuindo os custos computacionais, reduzindo-os tanto no momento de investimento inicial, quanto os custos operacionais em infraestrutura de computação.

Nesse sentido, de acordo com Gupta et al., (2021) empresas como *International Business Machine* (IBM), *Amazon Web Services* (AWS), *Google Cloud Plataform* e *AZURE* disputam para prover os melhores serviços em computação em nuvem atualmente e a AWS tem liderado a disputa em termos de quantidade de clientes.

3.2 Computação *serverless*

Computação *Serverless* ou função-como-um-serviço (*function-as-a-service*) é uma arquitetura de *software* em que a aplicação é descomposta em gatilhos (eventos) e ações (funções), disposta numa plataforma de computação em nuvem que provê um ambiente de hospedagem e de execução. Nesse sentido, o desenvolvedor da aplicação concentra a preocupação em desenvolver apenas funções leves e sem estado (sem reter contexto em diferentes execuções), que possam ser executadas por meio de uma API. A aplicação consome os recursos da plataforma de computação em nuvem somente enquanto estiver em execução e, posteriormente, os recursos são liberados. O modelo de preço inclui apenas a quantidade de tempo em que os recursos estiverem em uso, não sendo cobrado por eles até que, novamente, haja demanda de processamento, por isso é chamado de *serverless* (sem servidor), (RAJAN, 2020).

Apesar de ser um conceito recente, a computação *serverless* tem sido muito usada na indústria, adotada em muitas aplicações como *machine*/*deep learning*, computação numérica, processamento de vídeo, internet das coisas, análises de *big data*, sistemas web e outros (SHAFIEI et al., 2021). Além disso, a indústria deve pagar por computação *serverless* em 2025 cerca de vinte e dois bilhões de dólares, contra três bilhões de dólares que foram pagos por esse

serviço em 2017. E prevê-se que a computação *serverless* deve ser empregada em cerca de cinquenta por cento das empresas em 2025 (JINFENG et al., 2023).

Ademais, soluções computacionais que envolvam computação em nuvem, *serverless* e micro serviços são ecologicamente sustentáveis, uma vez que priorizam otimização de recursos, escalabilidade sob demanda, e sustentabilidade na gestão dos *data-centers*. (ATADOGA, 2024).

Em resumo, as diversas vantagens do uso de uma arquitetura *serverless* derivam do fato dela otimizar a utilização dos recursos computacionais disponíveis na nuvem. Uma vez que o provedor de computação em nuvem garante a disponibilidade do serviço quando há demanda e que, durante períodos de ociosidade, ou seja, quando o serviço não está sendo utilizado, tais recursos não são empregados e não implicam em cobrança. Isso, aliado à abstração de toda a infraestrutura necessária para esse comportamento, permite que os esforços se concentrem exclusivamente na lógica da solução, sem a necessidade de se preocupar com a infraestrutura subjacente.

O primeiro exemplo de computação *serverless* é a plataforma AWS Lambda introduzida pela *Amazon* em 2014. Nessa plataforma, os desenvolvedores de código projetam seus sistemas baseado em funções. Essas funções são processadas a partir de gatilhos, como uma requisição HTTP, por exemplo, operando numa arquitetura orientada a eventos. Desse modo, os usuários da AWS Lambda podem focar apenas no desenvolvimento lógico do código-fonte, abstraindo questões como gerenciamento do servidor, além das vantagens como escalabilidade dinâmica, e um modelo de pagamento conforme o uso. Isso significa que os recursos em nuvem podem aumentar ou diminuir de acordo com a demanda, e somente serão cobrados pelo tempo em que estiverem em uso. (GOLEC et al., 2023a).

Na computação *serverless*, os recursos em nuvem ociosos são liberados, a fim de se manter a eficiência energética e a otimização da política de preços favoráveis. Este processo é conhecido como escalar para zero. Entretanto, os recursos que escalam para zero levam um tempo para serem instanciados novamente para o reuso quando surge uma demanda nova por processamento. Isso pode gerar um atraso no processamento desta requisição. Este atraso é conhecido como *cold start*. *Cold start* tem efeitos negativos, como: atraso na resposta em aplicações que são sensíveis ao tempo (Aplicações em tempo real), performance inconsistente, e prejuízo à experiencia de usuário. Minimizar os problemas advindos do *cold start* pode contribuir para uma mais rápida adoção da indústria a este modelo de serviço, aumentando a confiança na computação *serverless* (GOLEC et al., 2023a).

Ainda, segundo Golec et al., (2023b), alguns fatores podem afetar e ter influência sobre o problema de *cold start*, como segue:

a) plataforma de computação em nuvem e ambiente de execução: cada plataforma usa diferentes técnicas de isolamento e ambiente de execução de contêiners específicos para sua arquitetura, portanto os tempos de latência de *cold start* diferem em cada uma;

b) tamanho do pacote de implantação da função (a ser detalhado na seção 4.2.2): com o aumento do arquivo de implantação da função, espera-se que a latência de *cold start* aumente, porque levará mais tempo para carregar os pacotes grandes em contêiners do que pacotes pequenos;

c) linguagem de programação: plataformas de computação *serverless* oferecem suporte nativo para várias opções de linguagens de programação, como Python, Java e C#. Como as linguagens têm diferentes sobrecargas de ambiente, paradigmas e arquiteturas, a latência de *cold start* também será diferente em cada uma. Linguagens de alto nível como Python e Java tem atrasos adicionais na inicialização;

d) concorrência: com o recurso de escalabilidade na computação *serverless*, uma nova instancia pode ser iniciada para atender a demanda de requisições simultâneas. Como resultado, picos de carga ocorrem devido ao uso excessivo de recursos, e picos de carga desencadeiam *cold start* (Figura 1);

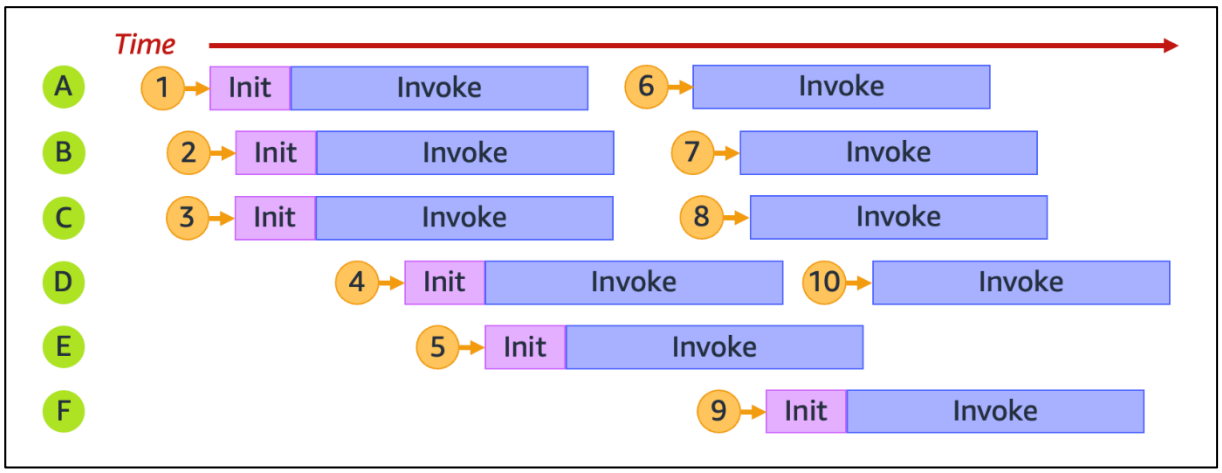
e) dependências das funções: antes da função ser implantada, as dependências, como bibliotecas necessárias para a execução da função, devem ser carregadas nos contêiners. Como esse processo de carregamento leva tempo, ele tem efeito direto no *cold start*;

f) alocação de recursos: em plataformas de provisionamento de computação *serverless*, recursos como RAM e CPU podem ser ajustados manualmente nas configurações de ambiente. Dependendo da quantidade de recursos, o tempo de implantação dos contêiners também varia, e é esperado que esse tempo afete no *cold start*.

Como pode ser observado na Figura 1, tomando como exemplo a AWS Lambda, que é o serviço oferecido pela Amazon Web Services para apoiar computação em serverless, de acordo com a documentação oficial da Amazon Web Services (2024b), por ser um paradigma altamente escalável, ao receber uma requisição e as instancias ainda estiverem ocupadas processando requisições antigas, uma nova instancia é inicializada para processar a requisição corrente. Na Figura 1, as letras de A a F representam as instancias que são inicializadas a cada requisição (números de 1 a 10) ao longo do tempo. "*Invoke*" significa o tempo de processamento de cada requisição, e *Init* o tempo necessário para a etapa de inicialização do processo. Repare

que as requisições processadas por instancias que já estavam inicializadas não tem tempo adicional de inicialização (*Init)* associadas.

Figura 1 – Diagrama de gestão de concorrência – AWS Lambda



Fonte: Amazon Web Services (2024b)

3.3 Modelos de execução de linguagens de programação

De acordo com Golec et al., (2023b), os diferentes modelos de execução de linguagens de programação têm influência sobre o comportamento das aplicações *serverless*. As linguagens de programação podem ser compiladas, interpretadas ou mistas.

3.3.1 Compilação e interpretação

De acordo com Scott (2005), a compilação de um programa é a tradução realizada pelo compilador do código-fonte, escrito em uma linguagem de alto nível - amigável para seres humanos - para um programa-alvo equivalente, geralmente em linguagem de máquina, que é, posteriormente, executado pelo sistema operacional. Uma alternativa a este processo é a interpretação, que, ao contrário da compilação, permanece presente durante a execução da aplicação. O interpretador implementa uma máquina virtual, cuja "linguagem de máquina" é a própria linguagem de programação de alto nível. O interpretador lê instruções nessa linguagem, podendo ser uma ou mais por vez, e as executa à medida que avança.

Em geral há vantagens inerentes às linguagens interpretadas em comparação às compiladas, principalmente em termos de flexibilidade e capacidade de diagnóstico, como a geração de mensagens de erro mais claras e informativas. No entanto, as linguagens compiladas

tendem a oferecer melhor desempenho durante a execução e uma gestão mais eficiente da memória, uma vez que o processo de compilação ocorre apenas uma vez, ao passo que a interpretação é realizada a cada execução do programa. (SCOTT, 2005). Linguagens como C, C++ e Go são exemplos de linguagens compiladas, enquanto Python, Perl e Ruby são exemplos de linguagens interpretadas

É importante observar que algumas linguagens adotam um processo misto de compilação e interpretação, como é o caso notável da linguagem Java. No paradigma Java, o código-fonte é primeiramente compilado pelo compilador Java (javac), resultando na geração de um formato intermediário conhecido como *bytecode*. Este *bytecode* é então interpretado pela *Java Virtual Machine* (JVM) durante a execução do programa. Essa abordagem mista combina as vantagens da compilação, que incluem a otimização do código e a detecção de erros em tempo de compilação, com as capacidades dinâmicas da interpretação, permitindo a portabilidade do código através da execução em diferentes plataformas sem a necessidade de recompilação (SCOTT, 2005).

Dentro do contexto da arquitetura *serverless*, exemplificado pela documentação oficial da AWS Lambda (Amazon Web Services, 2024a), é observado que funções escritas em linguagens compiladas não são recompiladas a cada instância, mesmo quando são escaladas para zero devido à falta de demanda. Isso ocorre devido ao processo de implantação, que envolve apenas o programa-alvo, previamente compilado. Quando uma função precisa ser instanciada, o ambiente de execução da AWS Lambda a executa diretamente, sem a necessidade de recompilação. Por outro lado, no caso de funções escritas em linguagem interpretada, como ocorre quando precisam ser instanciadas, é realizado um processo de interpretação a cada instanciação.

3.4 Sistemas distribuídos

Segundo Tanenbaum e Van Stenn (2007, p. 01), “um sistema distribuído é um conjunto de computadores independentes que se apresenta a seus usuários como um sistema único e coerente”. De acordo com Lewis e Martin (2014), um sistema distribuído baseado numa arquitetura em microsserviços é uma abordagem para o desenvolvimento de uma única aplicação construída por um conjunto de aplicações menores, ou funções, cada uma executando em seu próprio processo e comunicando-se, geralmente, a partir de uma API por meio do protocolo HTTP.

Existem diferentes modelos de sistemas distribuídos, mas neste trabalho o sistema foi implementado utilizando a arquitetura cliente/servidor (C/S). O modelo C/S é composto por um ou mais aplicações clientes e um ou mais aplicações servidoras. As aplicações clientes e servidoras interagem de maneira concorrente. O cliente é uma aplicação hospedada num ambiente apartado do servidor, e emite requisições para uma aplicação de servidor e aguarda a resposta da solicitação. O cliente não responde a requisições. O servidor é uma aplicação hospedada em um ambiente que oferece infraestrutura computacional coerente aos serviços que disponibiliza, e não pode nunca iniciar a comunicação com nenhuma aplicação, apenas responder a requisições das aplicações clientes.

3.4.1 Protocolo HTTP

Bilhões de imagens JPEG, páginas HTML, arquivos de textos, vídeos MPEG, arquivos de áudio WAV, Java applets e mais navegam pela internet todos os dias. O HTTP transfere a maior parte dessas informações de forma rápida, conveniente e confiável de servidores web em todo o mundo (GOURLEY; TOTTY, 2002).

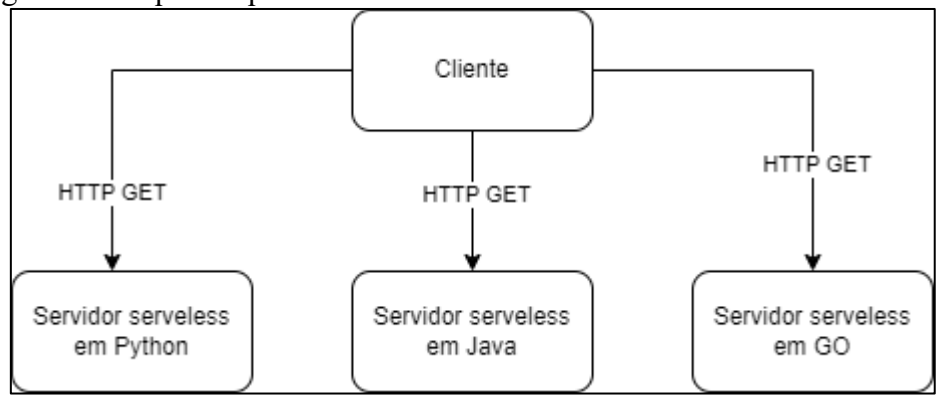
As transações em HTTP são compostas por requisições e respostas. Cada requisição HTTP está associada a um método HTTP que identifica a natureza da solicitação. Por exemplo, requisições com o método GET implicam na leitura de informações, enquanto requisições com o método POST tratam da escrita de informações. As respostas HTTP, por sua vez, estão associadas a um código de status que descreve o comportamento da aplicação receptora diante da solicitação. Por exemplo, um código de status da classe 2xx indica sucesso na requisição, enquanto a classe 4xx indica um erro na solicitação por parte do cliente, e a classe 5xx indica um erro por parte do servidor (GOURLEY; TOTTY, 2002).

4 METODOLOGIA

4.1 Sistema distribuído

Para mensurar o atraso causado pelo *cold start*, neste trabalho foi desenvolvido um sistema distribuído, no modelo cliente-servidor, composto por quatro aplicações: uma aplicação cliente e três aplicações servidoras (Figura 2). A aplicação cliente foi projetada para ser executada localmente, enquanto as aplicações servidoras são funções Lambda, também mencionadas apenas como *lambda*, hospedadas na AWS, utilizando uma arquitetura *serverless*. A aplicação cliente envia requisições HTTP para um *endpoint* do tipo GET disponibilizado pelas aplicações servidoras. Cada aplicação servidora foi implementada em uma linguagem de programação diferente: Python, Java e Go. Essas linguagens foram escolhidas pois cada uma representa um modelo de execução de linguagem de programação distinto: Python sendo uma linguagem interpretada; Go sendo uma linguagem compilada; Java sendo uma linguagem mista.

Figura 2 – Esquema que demonstra o Sistema Cliente/Servidor desenvolvido.



Fonte: Elaborado pelo autor.

Sob esse contexto, o sistema foi projetado para que a rota HTTP GET em cada aplicação servidora retorna à requisição o mesmo corpo de resposta. Em caso de sucesso, é retornado um código de status HTTP 200, indicando que houve sucesso na requisição, acompanhado de um arquivo no formato JSON com dois campos, sendo eles:

- Identificação da requisição: um id gerado pela aplicação servidora para identificar a requisição.
- Número da ordem de requisição (NOP): um número inteiro indicando em que ordem essa requisição foi processada pela instancia que recebeu essa requisição, desde quando esta instancia foi inicializada pela primeira vez.

Em caso de erro, é retornado um código de status HTTP 500, indicando uma falha inesperada na requisição.

As aplicações servidoras implementam o mesmo algoritmo, diferenciando apenas na linguagem de programação utilizada.

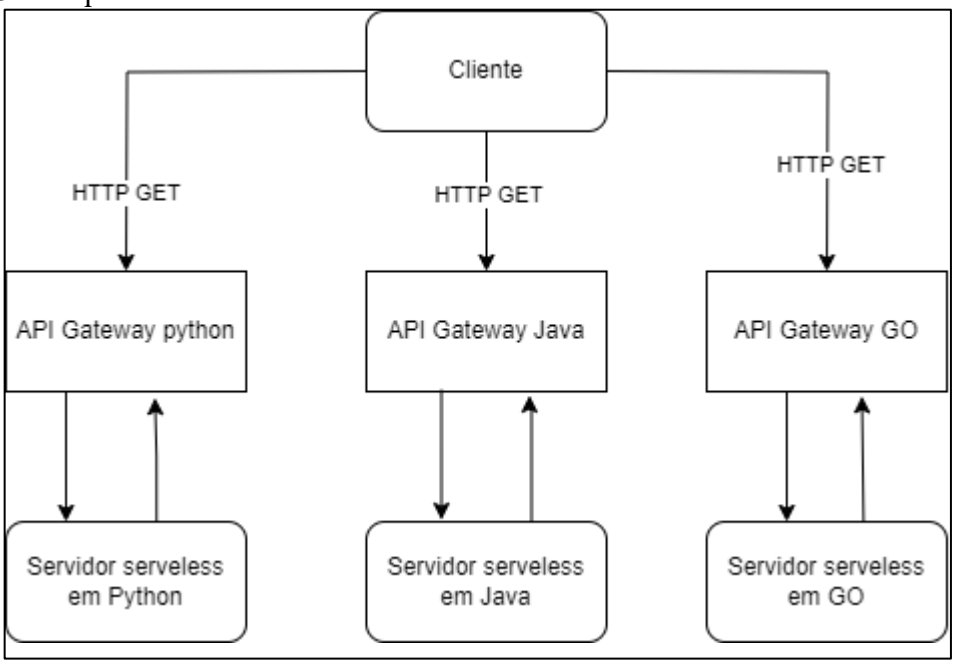
É importante ressaltar que o sistema desenvolvido neste trabalho tem fins exclusivos de testes, não se comprometendo em solucionar problemas reais, apenas em gerar dados que possam ser analisados, visando mensurar o tempo de resposta das aplicações *serverless*.

4.2 Desenvolvimento das aplicações servidoras

Aplicações servidoras foram projetadas para executarem em arquitetura *serverless* e implantadas no AWS *lambda*.

Lambda é um serviço de computação *serverless* oferecido pela AWS, e o gatilho de execução pode ser disparado por diferentes maneiras, no presente trabalho, o gatilho de execução foi o recebimento de uma requisição HTTP. Para isso, utilizou-se o recurso de API Gateway oferecido pela AWS.

Figura 3 – Esquema do sistema cliente/servidor desenvolvido - Inclusão do API Gateway.



Fonte: Elaborado pelo autor.

Segundo Amazon Web Services (2024a), API Gateway é um serviço para criação, publicação, manutenção, monitoramento e proteção de APIs em qualquer escala. O API

Gateway serviu como um mediador entre as requisições disparadas pela aplicação cliente e as aplicações servidoras. A Figura 3 demonstra como é o comportamento das requisições com a inclusão do API Gateway entre a aplicação cliente e as servidoras.

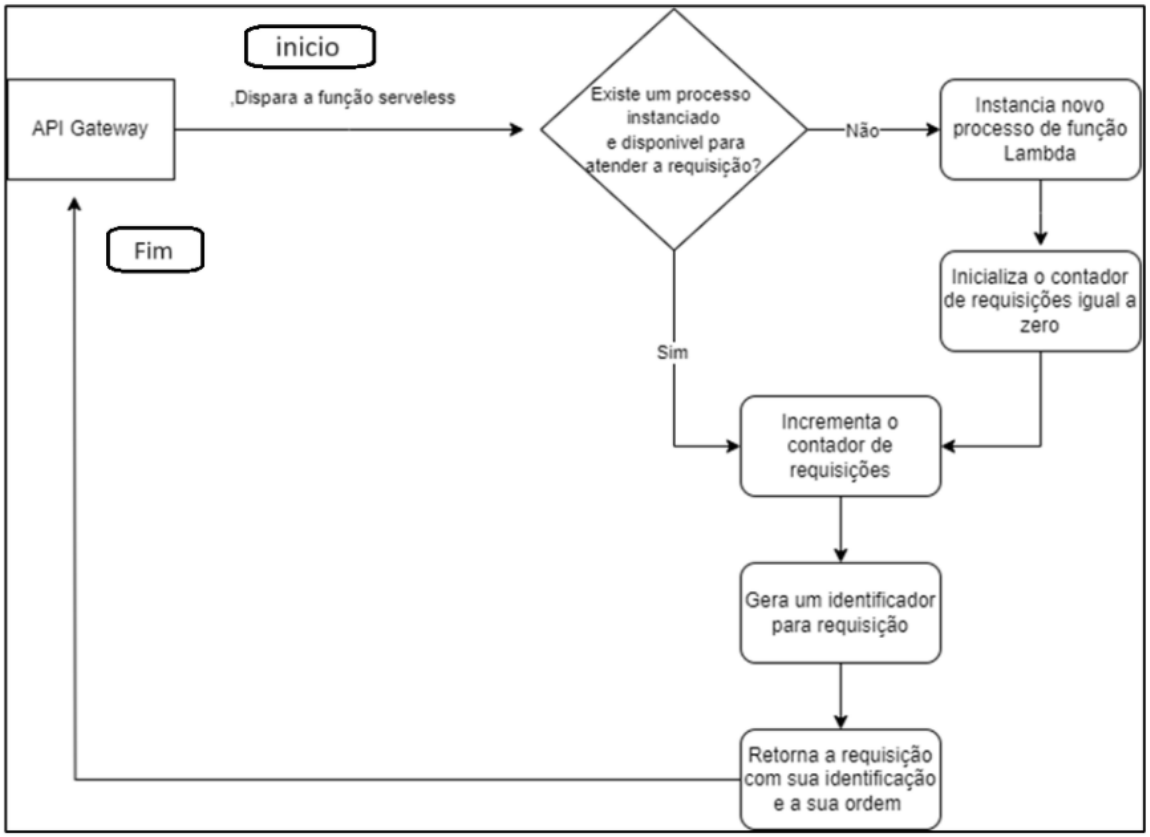
Nesse sentido, a requisição que a aplicação cliente dispara foi encaminhada para o serviço de API Gateway, que faz a mediação da comunicação entre a requisição da aplicação cliente e a resposta da aplicação servidora.

O processamento da requisição, apesar de ser em três linguagens de programação diferentes, é o mesmo, tendo todos a seguinte estrutura:

1) Um contador é incrementado a cada requisição recebida. Este contador é inicializado na instanciação da *lambda* e zerado sempre que a *lambda* escala para zero.
2) Foi gerado um id para identificar a requisição recebida.
3) A requisição foi retornada com a sua identificação, e com um número inteiro ditando qual a sua ordem desde que a aplicação que a processou foi instanciada pela última vez, definido pelo contador.

A Figura 4 exemplifica o fluxo de processamento das aplicações servidoras na AWS Lambda.

Figura 4 - Fluxo de processamento das aplicações servidoras



Fonte: Elaborado pelo autor.

Instanciar *lambda* pode ser mencionado, significando inicializar um processo de instancia da sua respectiva aplicação *serverless*. Como foi mencionado no referencial teórico, escalar para zero é um comportamento de aplicações *serverless*, que significa finalizar as instâncias visando economizar recursos em períodos de ociosidade computacional.

4.2.1 Implementação das aplicações servidoras

Cada aplicação foi desenvolvida e testada localmente, e depois implantada na AWS Lambda. Para desenvolvimento e testes locais, foram utilizadas algumas ferramentas que serão descritas nos tópicos seguintes.

4.2.1.1 AWS *Command line interface* (CLI) e AWS *serverless application model* (SAM)

AWS CLI e AWS SAM são ferramentas fornecidas pela AWS para facilitar o gerenciamento e desenvolvimento de aplicações em nuvem. O AWS CLI é uma ferramenta de linha de comando que permite interagir com os serviços da AWS usando comandos no terminal. Ela foi necessária para realizar as configurações de integração entre o ambiente local, onde as aplicações foram desenvolvidas, e a conta da AWS onde foram implantadas.

Segundo a Amazon Web Services (2024a), O AWS SAM é um *framework* fornecido pela AWS para construir e implantar aplicações serverless, oferecendo uma maneira simplificada de definir os recursos necessários para essas aplicações, através da implementação de um arquivo template no formato YAML. Esse arquivo pode incluir configurações para funções *lambda* e informações sobre as definições das APIs do API Gateway. O AWS SAM foi necessário para criar no ambiente local uma réplica do ambiente em nuvem, para que as aplicações pudessem ser testadas.

4.2.1.2 Docker

O *Docker* é uma plataforma de código aberto que automatiza a implantação de aplicações dentro de contêineres de *software*, permitindo que as execute de forma consistente em qualquer ambiente. Container é uma unidade leve e portátil que contêm tudo o que uma aplicação precisa para funcionar. É uma entidade de virtualização no nível de sistema operacional que permite a execução de aplicações em um ambiente isolado, utilizando o mesmo

kernel do sistema operacional *host*, mas com espaços de usuário, sistemas de arquivos, processos e redes separados (DOCKER, 2024).

Tal plataforma é necessária para utilizar certos recursos do AWS SAM, porque ela permite a criação de um ambiente de execução local que emula o ambiente da AWS Lambda, permitindo que o desenvolvimento e os testes fossem feitos de forma concisa e eficiente.

No sistema operacional Windows, o Docker oferece uma interface gráfica, que simplifica a instalação e gerenciamento do mesmo, conhecido por Docker Desktop.

4.2.1.3 Visual studio code (VSCode)

De acordo com Microsoft (2024), VSCode é um leve, porém poderoso editor de código, oferece suporte nativo javascript, typescript e node.js, mas possui um ecossistema de extensões rico para outras linguagens, como Java, Python e GO. Importante salientar a diferença entre o editor de código VSCode da IDE Visual Studio. A IDE Visual Studio é uma ferramenta completa de desenvolvimento integrada, ideal para grandes projetos e múltiplas linguagens, oferecendo uma ampla gama de funcionalidades como depuração avançada, design de interface, e integração profunda com o .NET. Já o Visual Studio Code (VSCode) é um editor de código-fonte leve e extensível, focado na simplicidade e na flexibilidade, suportando diversas linguagens e extensões, adequado para desenvolvimento ágil e rápido.

4.2.2 Implantação das aplicações servidoras

Para o processo de implantação das funções Lambda, foram utilizados os recursos de criação e integração de pacotes de implantações, disponibilizados no AWS SAM.

De acordo com Amazon Web Services (2024a), pacote de implantação é um arquivo compactado, contendo todo o contexto necessário para a execução da aplicação, como o código fonte da aplicação, ou o artefato resultante da compilação, arquivos de configurações e suas dependências.

4.3 Desenvolvimento da aplicação Cliente

No presente trabalho, a aplicação cliente foi projetada para ser executada localmente, diferente das aplicações servidoras que foram projetadas para serem hospedas em ambiente de

nuvem, numa arquitetura *serverless*. Isso se deve ao fato de que a aplicação cliente tem a responsabilidade de mensurar o atraso de resposta das aplicações servidoras, portanto é ideal que ela seja hospedada num ambiente apartado, simulando um comportamento comum, semelhante a realidade, como o comportamento de um sistema *web*, por exemplo, em que os clientes são computadores espalhados ao redor do mundo.

A aplicação cliente foi projetada para produzir requisições HTTP através dos *endpoints* implementados nas aplicações servidoras, calcular o tempo de resposta de cada requisição, e guardar esses dados num arquivo de forma estruturada. De modo que, posteriormente, esses dados pudessem ser analisados e gerassem informações sobre o comportamento das aplicações servidoras.

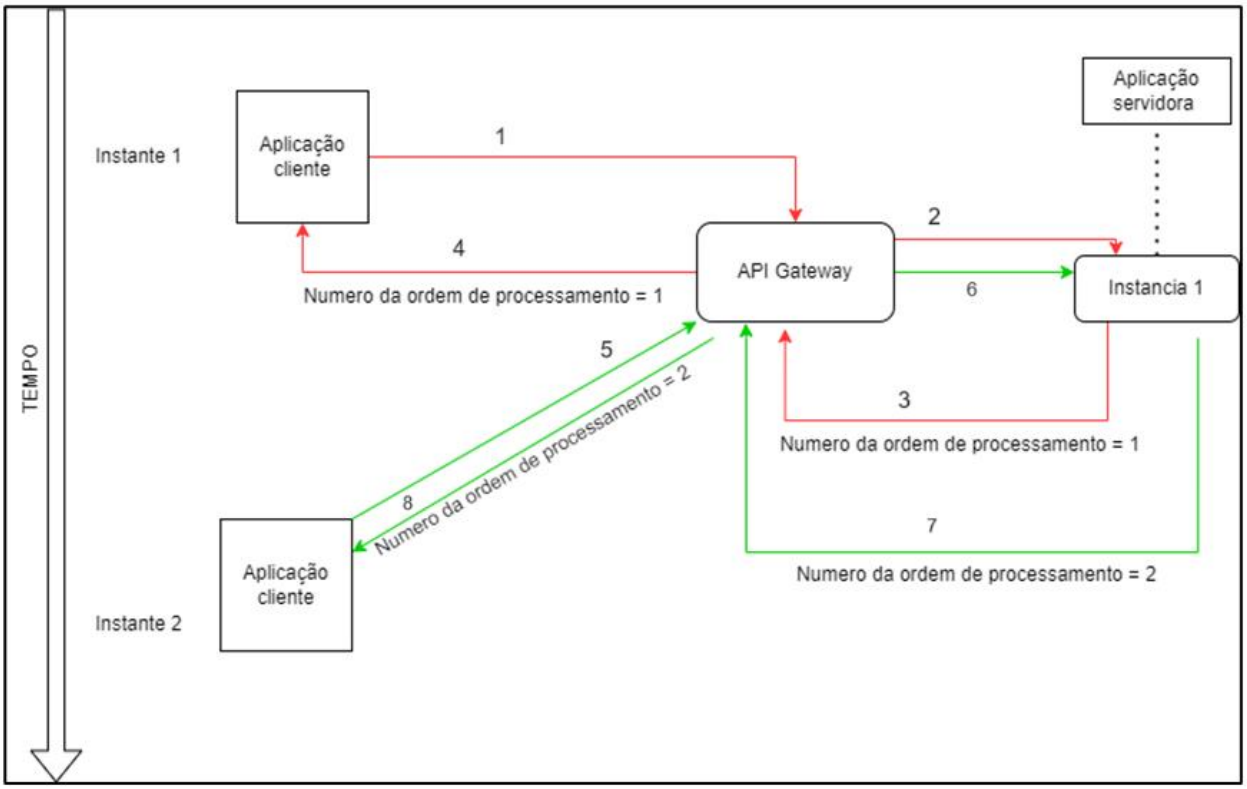
Desse modo, a aplicação cliente tem flexibilidade quanto ao consumo dos recursos das aplicações *serverless*, podendo variar na carga de requisições, bem como nos intervalos entre elas. Assim é possível gerar cenários de picos de requisições, seguidos de intervalos de ociosidade que precedem outro pico de requisições. Com isso, as aplicações serveless podem ser escaladas para zero.

Sabendo que em cada requisição as aplicações servidoras devolvem um número que indica a ordem em que essa requisição foi processada pela aplicação *serverless* desde que esta foi instanciada pela última vez, a aplicação cliente é capaz de guardar esse indicador para cada requisição, com seu respectivo tempo de resposta calculado. Isso implica que, sempre que uma dada requisição tiver como resposta um NOP igual a 1 (um), trata-se da primeira requisição que a sua respectiva aplicação *serverless* processou desde que foi instanciada. Esse dado é relevante, pois este trabalho visa mensurar o tempo de atraso provocado pelo *cold start*, que impacta justamente na primeira requisição de uma instancia de função *lambda*, pois foi esta que provocou a instanciação da respectiva *lambda*.

Sob essa perspectiva, a aplicação cliente guardou os registros de cada requisição, identificando cada uma quanto à ordem em que foi processada na aplicação servidora e seu respectivo tempo de resposta. De posse do tempo e do NOP de cada requisição, foi possível, posteriormente, filtrar os registros de requisições por NOP, para calcular o tempo médio de resposta em requisições de mesmo NOP nas aplicações servidoras. Por exemplo, é possível identificar as requisições que tem NOP igual a 1 (requisições impactadas pelo *cold start*) e as requisições que têm NOP diferente de 1 (requisições que foram processadas sem impacto do *cold start*), e comparar o tempo de resposta entre essas duas classes de requisições, para analisar se o *cold start* provocou algum atraso no tempo de resposta das requisições com NOP igual a 1, em comparação com o tempo de resposta das requisições com NOP diferente de 1.

Para facilitar no entendimento do comportamento da aplicação cliente, a Figura 5 levanta um seguinte possível cenário: a aplicação cliente dispara duas requisições para uma mesma *lambda* em dois instantes distintos, e a *lambda* não foi escalada para zero no intervalo entre as duas requisições. Neste caso, como a requisição foi processada pela mesma instância da *lambda*, a aplicação cliente irá guardar dois registros, sendo: um registro com NOP igual a 1 (um); e um registro com NOP igual a 2 (dois).

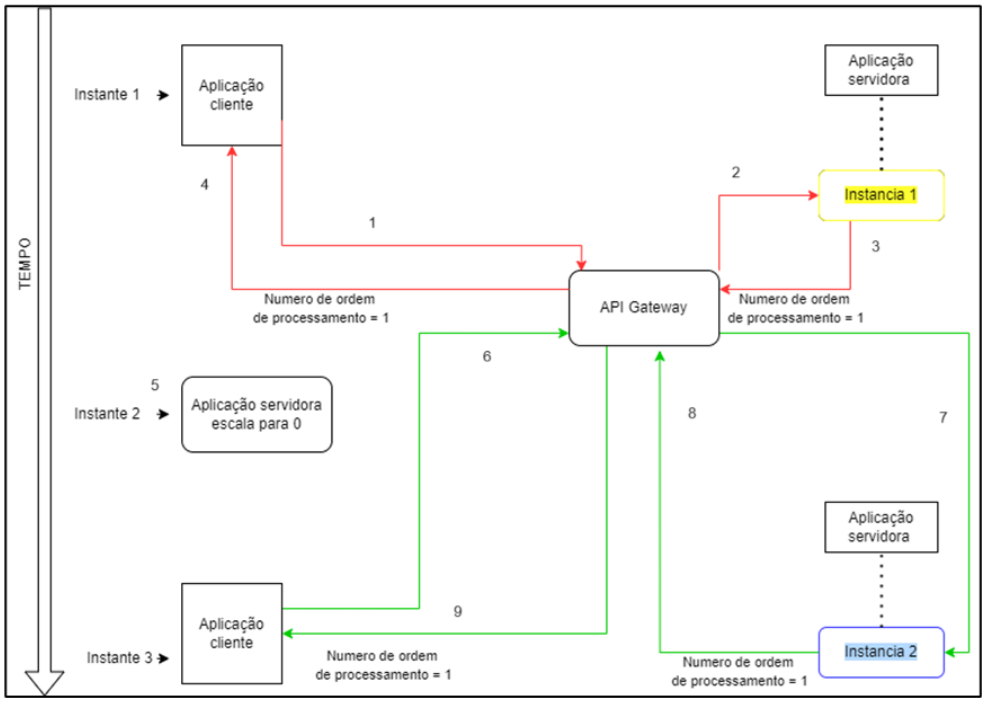
Figura 5 - Exemplo de duas requisições para a mesma instância.



Fonte: Elaborado pelo autor.

No entanto, outro cenário também é possível, como exemplifica a Figura 6: a aplicação cliente dispara duas requisições para a mesma *lambda*, mas no intervalo entre as duas requisições, a *lambda* foi escalada para zero, fazendo com que as requisições sejam processadas por instancias diferentes. Neste caso, a aplicação cliente irá guardar dois registros de resposta, ambos com o NOP igual a 1 (um), indicando que as duas requisições foram as primeiras a serem processadas por suas respectivas instancias da *lambda*.

Figura 6 - Exemplo de duas requisições para instâncias diferentes.



Fonte: Elaborado pelo autor.

A aplicação cliente foi implementada para realizar requisições em uma *lambda* por execução, salvando os registros de resposta em um arquivo no formato CSV, de modo que cada arquivo de registro de resposta representa os dados de uma execução. A quantidade de requisições varia em cada execução, bem como a quantidade de rodadas de requisições, se houver necessidade. Por exemplo, caso se queira produzir 100 requisições a uma *lambda*, fazendo isso a 10 disparos de requisições por rodada, a aplicação cliente recebe esses dados como entrada para a execução, e divide as 100 requisições totais, por 10 rodadas, resultando em 10 requisições por rodada. O intervalo de tempo entre uma rodada e outra é um valor aleatório entre 1 e 30 minutos, de modo que esse intervalo entre as rodadas de requisições gere ociosidade, possibilitando que as *lambdas* possam ser escaladas para zero, para que quando outra rodada de requisições for disparada, haja demanda computacional para que se instanciassem novas *lambdas* para as processarem.

O Algoritmo 1 mostra em pseudocódigo o comportamento da aplicação cliente. O *looping* da linha 10 garante que todas as rodadas sejam processadas, e o da linha 11 que todas

as requisições das rodada seja disparada. A linha 17 descreve o intervalo entre uma rodada entre duas rodadas.

Quadro 1 – Algoritmo 1 – Fluxo de processamento da aplicação cliente

```
1   Entrada:
2   endpoint da aplicação servidora
3   quantidade total de requisições
4   quantidade de requisições por rodada
5   nome do arquivo de resposta
6   Saída:
7   arquivo no formato CSV com os registros de respostas
8   Inicialização
9   calcular quantidade de requisições por rodada
10   Enquanto x < quantidade de rodadas
11     Enquanto y < quantidade de requisições por rodada
12       faz a requisição para aplicação servidora
13       calcula o tempo de resposta da requisição
14       salva os dados no arquivo de registros
15       atualiza y = y + 1
15     Fim enquanto
16     atualiza x = x + 1
17     espera um intervalo aleatório de 1 a 30 minutos
18   Fim enquanto
19   salva arquivo com registros de resposta
```

4.4 Descrição do ambiente local

As aplicações foram desenvolvidas num computador com processador 13th *Gen* Intel(R) *Core*(TM) i5-1342H 2.10 GHz, utilizando o sistema operacional *Windows* 11 *Home* (64 bits).

As aplicações servidoras foram desenvolvidas utilizando as linguagens de programação:

Python versão 3.8.0.

Java versão 11.0.23.

GO versão go1.22.3.

A aplicação cliente foi desenvolvida com a linguagem de programação Python versão 3.8.0.

As aplicações servidoras foram hospedadas na região sa-east-1 da AWS, que representa os servidores alocados na América do Sul, mais precisamente no estado de São Paulo – BR.

A aplicação cliente foi hospedada e executada no mesmo computador em que foi desenvolvida.

4.5 Experimentos computacionais

Visando mensurar o atraso provocado pelo *cold start* nas aplicações *serverless* desenvolvidas, foi efetuada uma carga de testes no sistema descrito neste trabalho, com o propósito de gerar dados que pudessem ser analisados, para gerar informações sobre o comportamento de cada aplicação *serverless*, implementada na sua respectiva linguagem de programação.

Após gerados os dados decorrentes do comportamento das aplicações *serverless*, foi possível agrupar os registros de resposta das aplicações pelo NOP, e extrair informações estatísticas a respeito do tempo de resposta das requisições, como a média, a mediana e a moda do tempo de resposta.

Com as medidas estatísticas a respeito do tempo de resposta de cada aplicação *serverless*, agrupadas pelo NOP, foi possível, então, realizar comparações tanto entre os tempos de resposta de cada agrupamento por NOP, quanto entre as diferentes aplicações *serverless* implementadas, cada uma em sua respectiva linguagem.

Ao realizar a comparação das informações de tempo de resposta entre as requisições agrupadas por NOP, numa mesma aplicação *serverless*, foi mensurado o impacto do *cold start* nessa aplicação, levando em consideração o tempo de resposta das requisições com NOP igual a 1 (um) e as requisições com NOP diferente de 1 (um). Essa comparação evidencia o atraso provocado pelo *cold start* em cada aplicação.

Ainda, ao realizar as comparações das informações de tempo de resposta entre as três aplicações *serverless*, agrupadas pelo NOP, pôde-se então mensurar o desempenho de cada implementação em relação as outras.

Para produzir esses dados, foram realizadas três rodadas de execução da aplicação cliente (Algoritmo 1) para cada aplicação servidora, resultando, portanto, em nove rodadas de testes ao todo. Em cada rodada de testes, foram realizadas oitocentas requisições, disparadas de dez em dez, com um intervalo aleatório de 1 a 30 minutos. Portanto, no total foram disparadas 2.400 requisições por aplicação servidora, ou 7.200 requisições ao todo. O intervalo entre as rodadas de 800 requisições por aplicação foi de 24 horas.

Gerou-se por este processo de testes nove *datasets*, um *dataset* por execução da aplicação cliente. Os *datasets* foram os seguintes:

1) Primeira_rodada_teste_aplicação_Java
2) Primeira_rodada _teste_aplicação_Python
3) Primeira_rodada_teste_aplicação_GO
4) Segunda_rodada_teste_aplicação_Java
5) Segunda_rodada_teste_aplicação_Python
6) Segunda_rodada_teste_aplicação_GO
7) Terceira_rodada_teste_aplicação_Java
8) Terceira_rodada_teste_aplicação_Python
9) Terceira_rodada_teste_aplicação_GO

Sendo as colunas de cada *dataset* as seguintes:

1) Identificação da requisição
2) Número da ordem de processamento da requisição
3) Tempo de resposta de cada requisição (em segundo)

É possível que os dados de tempo de resposta das aplicações, gerados pelos experimentos computacionais, estejam com algum ruído provocado pelo comportamento de rede, porque o protocolo que rege a comunicação entre as aplicações requer um tempo para estabelecer conexões e transportar os dados. Para minimizar o impacto desses ruídos, os testes foram feitos em 3 rodadas, com intervalo de tempo entre as rodadas de 24 horas, visando evitar situações circunstanciais. Além disso, todos os dados foram gerados por aplicações submetidas às mesmas condições de ambiente e rede, para que eventuais atrasos provocados por motivo diferente do *cold start* fossem igualmente distribuídos entre as três aplicações. Como o presente trabalho visa comparar o atraso entre as aplicações, considerando que esses ruídos foram uniformemente distribuídos, a comparação não é impactada.

5 RESULTADOS

Neste capítulo são descritas as técnicas empregadas para análise dos dados gerados pelos experimentos computacionais, visando mensurar e comparar o atraso provocado pelo *cold start* nas aplicações *serverless* que foram desenvolvidas. Além disso, os dados mensurados são exibidos e comparados por meio de tabelas e gráficos.

5.1 Coleta e tratamento dos dados

Os dados das requisições de cada *dataset*, gerados pelos experimentos computacionais, foram agrupados pelo NOP, para extrair informações estatísticas do tempo de resposta das requisições. De modo que, para cada NOP, fosse possível calcular a média, a mediana e a moda do tempo de resposta em cada aplicação, por rodada.

Por exemplo, para calcular a média de tempo de resposta das requisições geradas na primeira rodada de teste, na aplicação implementada pela linguagem Java, com o NOP igual a 1, usou-se o *dataset* Primeira_rodada_teste_aplicação_Java, filtrou-se todos os registros de resposta cujo NOP seja igual a 1, e calculou-se a média do campo Tempo de resposta de cada requisição.

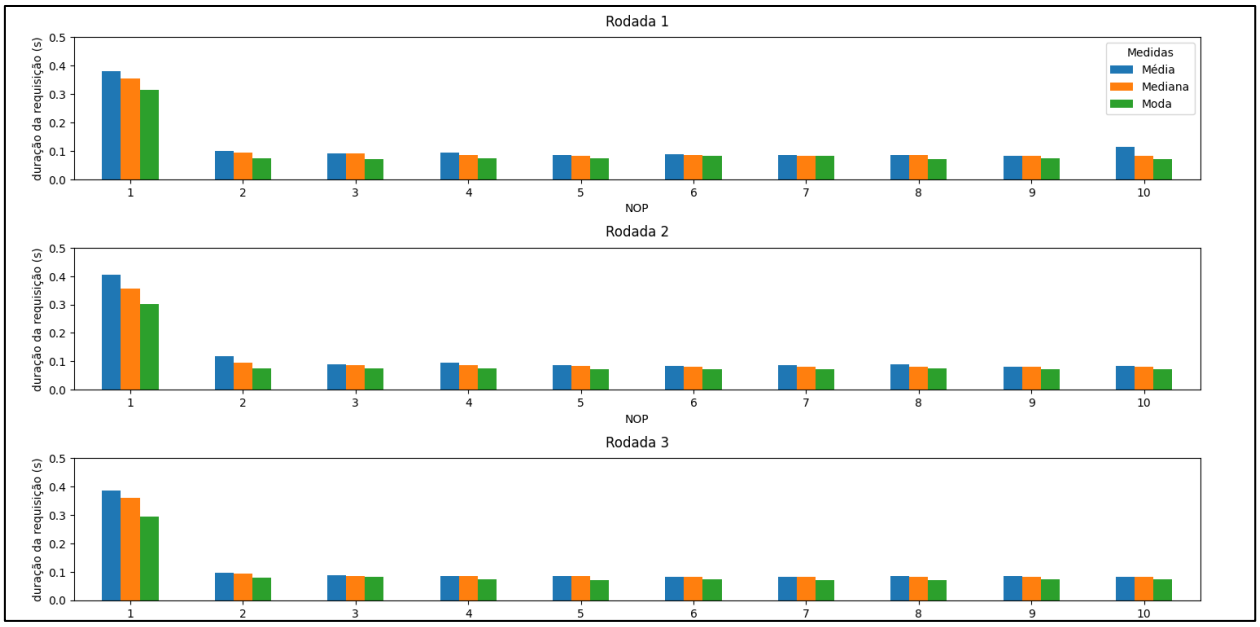
As razões pelas quais as medidas estatísticas média, moda e mediana foram escolhidas para mensurar os tempos de resposta das requisições são as seguintes: a média oferece uma representação de fácil interpretação do comportamento geral das aplicações testadas. A mediana oferece uma medida mais confiável do ponto central dos dados, sendo resistente aos valores extremamente altos ou baixos, ocasionados por alguma indisponibilidade de rede, por exemplo. A moda é capaz de identificar tendencias ou padrões comuns no comportamento das aplicações testadas. Ao realizar uma análise comparativa utilizando tais medidas, é possível ter uma compreensão mais completa dos dados. Todos os dados de tempo aqui apresentados estão na unidade de segundos.

5.2 Apresentação dos dados

A Figura 7 exibe o gráfico do desempenho da aplicação desenvolvida utilizando a linguagem Python, nas três rodadas. As primeiras requisições (NOP igual a 1) processadas pelas instancias da *lambda* em Python obtiveram, nas três rodadas, tempo de resposta nas três medidas estatísticas mensuradas entre 0.3 e 0.4 segundos, enquanto as demais requisições (NOP diferente de 1) obtiveram tempo de resposta, nas medidas estatísticas, menor que 0.1 segundo.

Portanto, a diferença de tempo de resposta das requisições afetadas pelo *cold start* e das requisições não afetadas, da *lambda* em Python, foi cerca de 3 a 4 vezes maior.

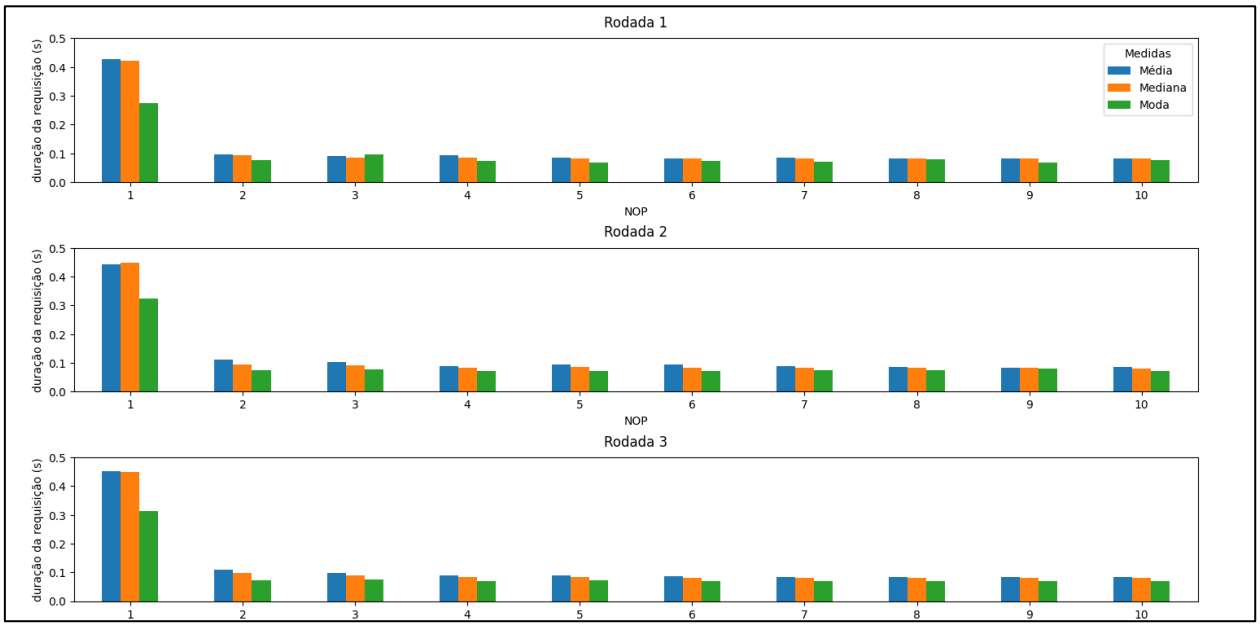
Figura 7 - Tempo de respostas das requisições ao longo das rodadas - aplicação em Python



Fonte: Elaborado pelo autor.

A Figura 8 exibe o gráfico de desempenho da aplicação desenvolvida utilizando a linguagem de programação Go nas três rodadas. Também ocorreu uma diferença de tempo de resposta das primeiras requisições processadas pelas instancias da aplicação desenvolvida em Go e as demais requisições, sendo o tempo das primeiras requisições, nas três medidas estatísticas, entre 0.27 e 0.45 segundos, e as demais requisições, nas medidas estatísticas, na faixa de 0.1 segundo. Logo, na *lambda* em Go, a diferença de tempo entre as requisições afetadas pelo *cold start* e as que não foram afetadas é semelhante à de Python, variando de 3 a 4 vezes maior.

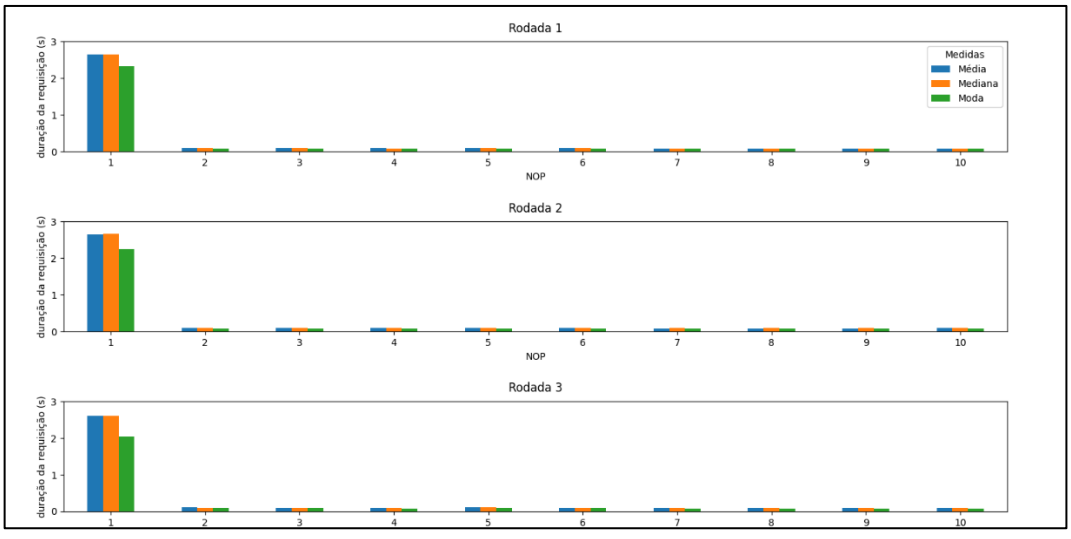
Figura 8 - Tempo de resposta das requisições ao longo das rodadas - aplicação em Go



Fonte: Elaborado pelo autor.

A Figura 9 exibe o gráfico de desempenho da aplicação desenvolvida utilizando a linguagem Java. Nessa *lambda*, as primeiras requisições processadas por suas instancias teve tempo de resposta, em medidas estatísticas, de 2.32 a 2.65 segundos, já as demais requisições obtiveram tempo de resposta abaixo de 0.1 segundo. Nesse sentido, Java foi a que demonstrou maior diferença de tempo das requisições afetadas pelo *cold start* das demais, mais de 20 vezes maior.

Figura 9 - Tempo de resposta das requisições ao longo das rodadas - aplicação em Java

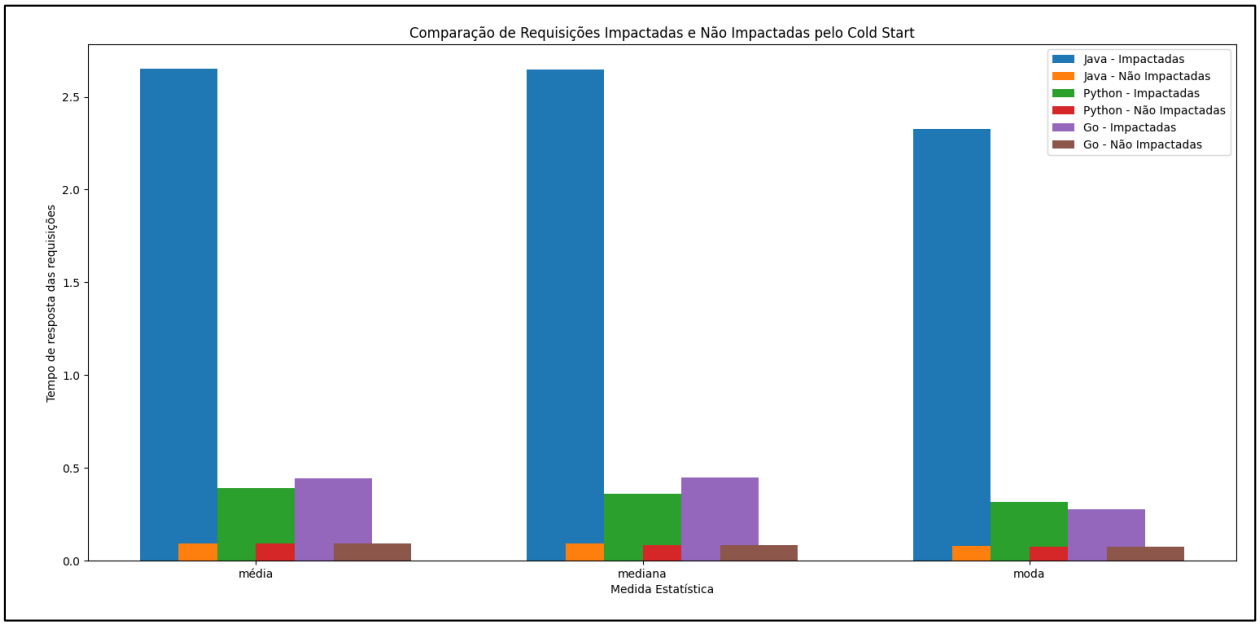


Fonte: Elaborado pelo autor.

5.3 Comparação de desempenho entre as aplicações

A Figura 10 apresenta um gráfico comparativo entre o tempo de resposta das requisições impactadas e não impactadas pelo *cold start* nas três aplicações *serverless* desenvolvidas em Python, Java e Go. Para isso, os dados das três rodadas de teste foram agrupados e os valores foram calculados conforme as medidas estatísticas. As requisições descritas como não impactadas pelo *cold start* são aquelas cujo NOP é diferente de 1, enquanto as consideradas impactadas são aquelas com o NOP igual a 1.

Figura 10 - Tempo de resposta das requisições impactadas x não impactadas pelo *cold start* nas aplicações desenvolvidas



Fonte: Elaborado pelo autor.

Comparando o desempenho, em termos de tempo de resposta das três aplicações, foi possível mensurar que a aplicação implementada em Java foi a que teve pior desempenho no tempo de resposta para processar as requisições com NOP igual a 1, mas levou tempo semelhante às outras aplicações para processar as demais requisições (NOP diferente de 1) , como mostra a Figura 10.

Dado que a aplicação em Java teve um desempenho menor que as demais, os Quadros 2 e 3 demonstram a diferença absoluta entre o tempo de resposta da aplicação em Java em

relação a Python e Go, respectivamente, levando em consideração apenas as requisições que pudessem ser afetadas pelo *cold start*.

Quadro 2 - Diferença de tempo de resposta - Python

Rodada	Medida estatística	Diferença do tempo de resposta em relação a aplicação desenvolvida em Java
1	média	2,26
	mediana	2,28
	moda	2,1
2	média	2,30
	mediana	2,29
	moda	1,94
3	média	2,21
	mediana	2,23
	moda	1,73

Fonte: Elaborado pelo autor.

Quadro 3 - Diferença de tempo de resposta - Go

Rodada	Medida estatística	Diferença do tempo de resposta em relação a aplicação desenvolvida em Java
1	média	2,22
	mediana	2,22
	moda	2,05
2	média	2,21
	mediana	2,21
	moda	1,92
3	média	2,15
	mediana	2,15
	moda	1,71

Fonte: Elaborado pelo autor.

Como demonstra os Quadros 2 e 3, nas três rodadas de testes, considerando todas as medidas estatísticas, não houve diferença entre os tempos de resposta das aplicações implementadas em Python e em Go superior a 0.01 segundos, considerando as requisições que pudessem ser impactadas pelo *cold start*.

O Quadro 4 exibe o tempo de resposta das requisições não impactadas pelo *cold start* nas três aplicações. Os dados de todas as rodadas foram agrupados para calcular as medidas estatísticas. Foi percebido que não houve diferença de tempo de resposta para essas requisições maior que 0.01 segundo.

Quadro 4 – Tempo de resposta das requisições não impactadas pelo *cold start.*

Linguagem	Medida estatística	Requisições não impactadas pelo cold start
Java	média	0.089
	mediana	0.089
	moda	0,078
Python	média	0.089
	mediana	0.083
	moda	0.072
Go	média	0.089
	mediana	0.083
	moda	0.073

Fonte: Elaborado pelo autor.

Portanto, como pode ser observado no Quadro 5, considerando o atraso calculado, houve um pequeno melhor desempenho das aplicações em Python em relação as aplicações em Go, Tendo Go, em média, um atraso 16,56% maior que Python. No entanto, as aplicações em Java demonstraram ter um atraso substancialmente maior que as demais aplicações. Em média, o atraso de Java foi 8.41 vezes maior que em Python e 7,21 vezes maior que em Go.

Por fim, como foi levantado na subseção de experimentos computacionais, é possível que o comportamento de rede tenha influência no tempo de resposta das requisições impactadas e não impactadas pelo *cold start*. No entanto, como todas as requisições foram submetidas ao mesmo ambiente e rede, esse tempo adicional foi distribuído entre elas. Soma-se a isso, que os tempos de resposta das requisições processadas pela *lambda* em Java obteve atraso substancial em relação as demais aplicações, evidenciando, portanto, o impacto do *cold start* nesse atraso (Quadro 5).

Quadro 5 - Tempo de atraso em todas as rodadas

Linguagem	Medida estatística	Requisições impactadas pelo *cold start*	Requisições não impactadas pelo *cold start*	Atraso em segundos
Java	média	2.653	0.089	2.54
	mediana	2.649	0.089	2.56
	moda	2.327	0,078	2.249

Python	média	0.391	0.089	0.302
	mediana	0.357	0.083	0.274
	moda	0.314	0.072	0.242
Go	média	0.441	0.089	0.352
	mediana	0.447	0.083	0.364
	moda	0.276	0.073	0.203

Fonte: Elaborado pelo autor.

6 CONCLUSÃO

A análise dos dados produzidos pelos experimentos computacionais revelou que a aplicação em Java apresentou pior desempenho em relação ao atraso provocado pelo *cold start* em aplicações *serverless*, nas 3 rodadas de testes, e em todas as medidas estatísticas utilizadas neste estudo: média, mediana e moda. Em média, Go obteve um atraso 16.56% maior que em Python, no entanto, o atraso de Java foi 8.41 vezes maior que em Python e 7,21 vezes maior que em Go. Portanto, este estudo sugere que a linguagem de programação Java tem maior incidência de atraso provocado pelo *cold start* em aplicações *serverless*, em comparação às linguagens Python e Go.

Como sugestões de trabalhos futuros propõe-se: desenvolver e comparar aplicações com outras linguagens de programação; investigar os motivos que levaram Java a ter pior desempenho em relação ao atraso provocado pelo cold start; adotar uma estratégia que mensure o tempo de resposta das requisições de modo que se extraia os impactos de rede, mensurando os tempos de resposta das requisições através de recursos de observabilidade da provedora de computação em nuvem.

REFERÊNCIAS

AMAZON WEB SERVICES. **AWS Lambda - Documentação**. 2024a. Disponível em: <https://docs.aws.amazon.com/pt_br/code/>. Acesso em: 20 mai. 2024.

AMAZON WEB SERVICES. **Escalabilidade da função do Lambda**. 2024b. Disponível em: <https://docs.aws.amazon.com/pt_br/lambda/latest/dg/lambda-concurrency.html> Acesso em: 16 jun. 2024

ATADOGA, Akoh, et al. Tools, techniques, and trends in sustainable software engineering: A critical review of current practices and future directions. **World Journal of Advanced Engineering Technology and Sciences**, v. 11, n. 1, p. 231-239, 2024.

BELLO, Sururah A. et al. Cloud computing in construction industry: Use cases, benefits and challenges. **Automation in Construction**, v. 122, p. 103441, 2021.

DOCKER. **Docker Documentation**. 2024. Disponível em: <https://docs.docker.com/>. Acesso em: 5 jun. 2024.

GOLEC, M., GILL, S. S., PARLIKAD, A. K., UHLIG, S. Healthfaas: Ai based smart healthcare system for heart patients using serverless computing. **IEEE Internet of Things Journal**, v. 10, n. 21, 18469-18476, 2023a. Disponível em: <https://doi.org/10.1109/JIOT.2023.3277500> Acesso em: 20 mai. 2024.

GOLEC, Muhammed, et al. Cold start latency in serverless computing: A systematic review, taxonomy, and future directions. **Association for Computing Machinery**. v. 1, n. 1, p. 1-34, 2023b. arXiv:2310.08437v1

GOURLEY, D.; TOTTY, B. **HTTP**: The Definitive Guide. 1. ed. Sebastopol: O'Reilly Media, 2002. 635 p.

GUPTA, B.; MITTAL, P.; MUFTI, T. A review on amazon web service (aws), microsoft azure & google cloud platform (gcp) services. **ICIDSSD**, New Delhi, v. 27, n. 28, 2021. DOI 10.4108/eai.27-2-2020.2303255

JINFENG, Wen; ZHENPENG, Chen; XIN, Jin; XUANZHE, Liu. Rise of the planet of serverless computing: A systematic review. **ACM Transactions on Software Engineering and Methodology**, v. 32, n. 5, p 1-61, 2023. Disponível em: <https://doi.org/10.1145/3579643>. Acesso em: 20 mai. 2024.

LEWIS, J.; FLOWER, M. **Microservices, a definition of this new architectural term**. 2014. Disponível em: <https://martinfowler.com/articles/microservices.html> Acesso em: 5 jun 2024.

MICROSOFT. **Visual Studio Code Documentation**. 2024. Disponível em: <https://code.visualstudio.com/docs>. Acesso em: 5 jun. 2024.

RAJAN, Arokia Paul. A review on serverless architectures-function as a service (FaaS) in cloud computing. **Telkomnika**, v. 18, n. 1, p. 530-537, 2020.

SAABITH, A. S.; FAREEZ, M. M. M.; VINOTHRAJ, T. Python current trend applications-an overview. **International Journal of Advance Engineering and Research Development**, v. 6, n. 10, 2019.

SANTOS, A. et al. Factors affecting cloud computing adoption in the education context-Systematic Literature Review. **IEEE Access**, v. 1, 2023. DOI 10.1109/ACCESS.2024.3400862

SCOTT, M. L. **Programming Language Pragmatics**. 2. ed. San Francisco: Morgan Kaufmann, 2005. 855 p.

SHAFIEI, H.; KHONSARI, A.; MOUSAVI, P. Serverless Computing: A Survey of Opportunities, Challenges, and Applications. **ACM Comput. Surv**., v. 1, n. 1, p. 1-27, 2021. DOI 10.1145/nnnnnnn.nnnnnnn [online]

SOUSA, F. R.C.; MOREIRA, L. O.; MACHADO, J. C. Computação em nuvem: Conceitos, tecnologias, aplicações e desafios. **II Escola Regional de Computação Ceará, Maranhão e Piauí (ERCEMAPI**), p. 150-175, 2009.

TANENBAUM, Andrew S.; VAN STEEN, Maarten. **Distributed Systems**: Principles and Paradigms. 2. ed. Upper Saddle River: Pearson, 2007. 686 p.

WEN, J. et al. Rise of the planet of serverless computing: A systematic review. **ACM Transactions on Software Engineering and Methodology**, v. 32, n. 5, p. 1-61, 2023.