Linguagem Node como uma alternativa a latência cold-start de FaaS de aplicações de baixa frequência de requisições desenvolvidas em Java

Paulo Otávio Ferreira dos Santos[1]

[1]Instituto Federal de Educação Ciência e Tecnologia do Rio Grande do Sul (IFRS)
BR-470, Km 172, 6.500. Bairro Sapopema. Veranópolis - RS

otaviopfs@gmail.com

Abstract. *Serverless applications are those that do not require the developer to provide or manage any servers. The developer can focus on core product logic and development and just release their code into a container at the service provider. However, when initializing a container there may be a latency called cold start. This article aims to use the Node.js language as an alternative to the Java language as a prevention against the cold start scenario in applications that have a low frequency of use. With the development of lambda functions with the same functionality in both languages. Noting that the node.js language had an 82% reduction in startup time compared to java.*

Resumo. *Aplicações serverless são aquelas que não exigem que o desenvolvedor provisione ou gerencie quaisquer servidores. O desenvolvedor pode se concentrar na lógica e desenvolvimento do produto principal e apenas lançar seu código em um container no provedor de serviço. Contudo, ao inicializar um container pode haver uma latência chamada cold start. Este artigo utiliza a linguagem Node.js como uma alternativa a linguagem Java como uma prevenção ao cenário do cold start em aplicações que possuem uma baixa frequência de utilização. Com o desenvolvimento de funções lambda com a mesma funcionalidade nas duas linguagens. Sendo notado que a linguagem node.js obteve uma redução de 82% no tempo de inicialização em relação ao java.*

1. Introdução

Hospedar um software na internet geralmente envolve o gerenciamento de algum tipo de infraestrutura de servidor. Normalmente, isso significa um servidor virtual ou físico que precisa ser gerenciado, bem como o sistema operacional e outros processos de hospedagem de servidores web para que um aplicativo seja executado. Desde o surgimento da computação em nuvem ou computação serverless, usar um servidor virtual de um provedor de nuvem como Amazon AWS (Amazon Web Services) significa a eliminação das preocupações físicas de hardware [Jonas et al. 2019], mas ainda requer algum nível de gerenciamento do sistema operacional e dos processos de software do servidor web.

A computação sem servidor (Serverless Computing) é um paradigma de computação em nuvem em que os aplicativos são hospedados por um provedor de nuvem, eliminando a necessidade de software em servidor e o gerenciamento de hardware pelo desenvolvedor [Bargmann 2019]. Os aplicativos são divididos em funções individuais que podem ser chamadas e escaladas individualmente. Os principais benefícios desses

tipos de arquiteturas incluem a redução dos custos operacionais, complexidade e tempo de desenvolvimento [Trilles et al. 2020]. Segundo [Castro et al. 2019], a computação serverless favorece as unidades pequenas e autônomas de computação para facilitar a gestão e escalar na nuvem.

A função como serviço (FaaS) é uma maneira de executar partes modulares de código na computação serverless. A FaaS permite que os desenvolvedores escrevam e atualizem um pedaço de código em tempo real, que pode ser executado em resposta a um evento. Isso facilita o dimensionamento do código e é uma maneira econômica de implementar microsserviços, porém a latência é variável de acordo com a linguagem escolhida. Segundo [Hosseini and Sahragard 2019], uma função na linguagem Java apresenta um desempenho pior e inconsistente em comparação com outras linguagens. Essa inconsistência se dá também devido ao número de CPUs que são usados no processamento, por exemplo, funções de 512 MB, cujo é tamanho padrão da AWS, é utilizado apenas um núcleo de CPU para processamento [Aaron Harris 2021].

Dentre os problemas de inconsistência, há uma latência que ocorre ao iniciar uma função que está ociosa. O *cold start* como é chamado, é um problema característico à arquitetura serverless. As funções são executadas em contêineres, a primeira execução de uma função passa por uma ”inicialização a frio”, pois o contêiner deve ser iniciado antes da execução. As execuções subsequentes usam os contêineres gerados, mas já existentes, para aproveitar os ambientes de execução ”quentes”[Manner et al. 2018]. As funções lambda permanecem quentes por um tempo médio de 15 minutos[1], após isso um novo contêiner deve ser gerado, ocasionando uma nova latência.

Por se tratar de um curto tempo de atividade da função, ”softwares com uma grande frequência de requisição raramente irão enfrentar o cold start”[Silva and Carvalho 2019]. Por outro lado, softwares com finalidades mais módicas onde não possuirão operações requisitadas a todo tempo, poderão enfrentar a latência. Baseado nisso, o objetivo deste trabalho foi realizar uma comparação de desempenho entre duas das linguagens mais populares na AWS Lambda, sendo elas java e node.js [Dantas 2022]. Desenvolvendo duas funções Lambda para leitura e persistência de dados, sendo acionadas por uma chamada de API HTTP. Dentre os artigos tidos como referência, nenhum trata do *cold start* especificamente para aplicações de pequeno porte. A proposta desta pesquisa é poder encontrar uma alternativa sem a complexidade de ferramentas robustas. E que por ventura, não traga um impacto temporal tão grande, mediante as configurações básicas da AWS Lambda.

As demais seções deste artigo estão estruturadas da seguinte forma: a seção 2 apresenta os trabalhos relacionados, comparando trabalhos recentes sobre o assunto. A seção 3 apresenta a fundamentação teórica, descrevendo os conceitos básicos de serverless computing e FaaS, seção 4 os materiais e métodos, metodologia e resultados, na seção 5 a discussão e na seção 6 a conclusão.

2. Trabalhos Relacionados

O problema do *cold start* é um dos maiores desafios que surgem ao usar funções sem servidor [Dantas 2022]. Com base nisso, foram selecionados artigos relevantes dos últimos

[1]https://aws.amazon.com/pt/about-aws/whats-new/2018/10/aws-lambda-supports-functions-that-can-run-up-to-15-minutes

três anos por meio de uma pesquisa usando três bases de dados científicas, representados na Tabela 1. As bases utilizadas foram a Scopus (Elsevier), Web of Science e Google Scholar. Para ambas as buscas as palavras-chave usadas foram: 'cold-start', 'serverless', 'faas' e 'problems'.

Trabalho	Solução para o Cold-start	Plataforma utilizada	Ano
Using Application Knowledge to Reduce Cold Starts in FaaS	Técnica de middleware, tem um baixo custo monetário a mais	IBM; AWS	2020
Reducing cold starts during elastic scaling of containers in Kubernetes	Escalonamento com Kubernetes	-	2021
Prebaking Functions to Warm the Serverless Cold Start	Usa técnica de clonagem CRIU (Linux) e prebaking	Linux	2020
A Time Series Forecasting Approach to Minimize Cold Start Time in Cloud-Serverless Platform	Mitigar o cold start com modelo de predição, junto de ferramentas do Kubernetes	-	2022
Análise de Mecanismos de Serverless Computing em Ambientes de Nuvens Computacionais	Realização um experimento quantitativo, investigando o fenômeno cold-start	AWS	2019

Tabela 1. Comparação de trabalhos relacionados.

No trabalho de [Bermbach et al. 2020] é proposto um middleware de coreografia multiplataforma leve que pode ser implantado junto com as funções lambda e que evita componentes de orquestração centralizados. Para o funcionamento é necessário implantar um código na função, sendo assim, ao chamar a função invoca o código que delega a execução de um manipulador de etapas, onde o middleware verifica se um estado de fluxo de trabalho foi passado. Atrelado a isso, os autores propõem três abordagens implementadas como parte do referido middleware, que reduzem o número de cold starts, consideram a plataforma FaaS uma caixa preta. Após um grande número de experimentos, os autores concluem que suas abordagens removem uma média de 30-40% dos cold starts das aplicações, porém a um acréscimo de baixo custo monetário por parte do provedor serverless.

No artigo de [Beni et al. 2021], a tratativa para solucionar a latência cold start foi a utilização do orquestrador de containers Kubernetes, aninhado ao compartilhamento de bibliotecas em camadas e gerenciamento de configuração imperativa. O Kubernetes é um plataforma, portável e extensiva para o gerenciamento de cargas de trabalho e serviços distribuídos em contêineres, que facilita tanto a configuração quanto a automação. Os autores concluem que a abordagem de compartilhamento de bibliotecas atrelado ao Kubernetes de fato teve uma redução de tempo de inicialização significativa em servidores como JVM e Tomcat.

O objetivo proposto no artigo de [Silva et al. 2020], se iguala no quesito de mitigar o problema do *cold start*. O autor cita que o tamanho do código influencia na inicialização da JVM do Java. Os autores utilizaram uma técnica de clonagem baseada em Checkpoint/Restore In Userspace (CRIU). A solução proposta mantém o estado do processo de uma instância serverless pronta para servir para recuperar esse estado em um processo clonado posteriormente, quando a plataforma precisar ser dimensionada. A técnica remove cerca de 47% a 71% da sobrecarga de inicialização, proporcionais ao tamanho do código da função.

No trabalho de [Jegannathan et al. 2022] é usado um modelo de previsão de séries temporais SARIMA (Seasonal Auto Regressive Integrated Moving Average), para prever o momento em que a solicitação recebida chega e, consequentemente, aumentar ou

diminuir a quantidade de contêineres necessários, reduzindo o tempo de lançamento da função. Após é implantado o PBA (Prediction Based Autoscaler), comparado com o HPA (Horizontal Pod Autoscaler), ferramentas do kubernetes. Ao comparar os dois, o HPA sofreu com o cold start, enquanto o PBA foi capaz de subjugar o efeito do cold start. O PBA também consumiu 18% menos recursos do que o HPA.

No artigo de [Silva and Carvalho 2019] o objetivo foi realizar uma pesquisa exploratória sobre a computação serverless. E também medir o impacto no desempenho de um serviço FaaS medindo o atraso no tempo de resposta causado pelo cold start. Silva aponta que o tempo que uma função permanece ociosa até acontecer o *cold start* é de 30 a 42 minutos. O autor descreve que funções de 128MB a 1024MB, conforme o aumento de memória o tempo de cold start foi decrescendo.

Baseado na comparação de trabalhos relacionados da arquitetura proposta, os trabalhos de Bermback (2020) e Silva (2020) apresentam tratativas similares, porém as soluções podem de fato ter um acréscimo monetário por parte do provedor, devido a chamadas adicionais a função. No trabalho de Beni (2021) é utilizada uma solução singular com o Kubernetes. Bem como o de Jegannathan (2022), onde além de kubernetes faz uso da ferramenta SARIMA (Seasonal Auto Regressive Integrated Moving Average), um modelo de previsão de séries temporais visando a otimização dos recursos. Levando em conta a proposta atual, um software com uma baixa frequência de requisições e utilizando uma baixa alocação de memória para a execução da lambda, essa solução acaba se mostrando demasiada robusta. Já no artigo de Silva (2019) é realizada uma análise dos impactos e métricas do cold start apenas na linguagem Node.js.

3. Serverless Computing

O termo serverless se popularizou depois do seu lançamento nos serviços da Amazon AWS (Amazon Web Services) em 2014 [Silva and Carvalho 2019], nesse modelo é o provedor de nuvem que fica responsável pelas tarefas de provisionamento, manutenção e escala da infraestrutura do servidor, dimensionando a infraestrutura para cima e para baixo sob demanda, conforme necessário. Os desenvolvedores só precisam empacotar o código em containers para fazer a implantação. O provedor de nuvem também é responsável por todo o gerenciamento e manutenção de infraestrutura de rotina, como atualizações e patches do sistema operacional, gerenciamento de segurança, planejamento de capacidade, monitoramento do sistema, entre outros.

Segundo [Buyya 2018], a computação em nuvem possibilita o estabelecimento de novos negócios em um período de tempo mais curto, facilitou a expansão de empresas em todo o mundo, acelerou o ritmo do progresso científico e levou à criação de vários modelos de computação para aplicações difundidas e ubíquas, entre outros benefícios. Para [Pelle 2020], a programação nativa em nuvem, microsserviços e as arquiteturas serverless fornecem uma nova maneira de desenvolvimento e gerenciamento de software.

A redução de custos de infraestrutura e recursos humanos é a vantagem básica da arquitetura serverless, “a solução serverless é aquela que não custa nada para ser executada se ninguém a estiver usando (excluindo o armazenamento de dados)” [Castro et al. 2019]. O provedor de nuvem aumenta e provisiona os recursos de computação necessários sob demanda quando o código é executado e os reduz novamente, chamado de ”escalonamento para zero”, quando a execução é interrompida.

3.1. FaaS

Além dos três modelos principais (Iaas, Saas, Paas), [Savage 2018] também menciona que serverless é uma quarta categoria de modelo de computação em nuvem chamada *Function as a Service* (FaaS) ou Função como Serviço. O FaaS é o alicerce fundamental do serverless, responsável por executar a lógica que determina como os recursos são alocados em um determinado cenário. Essas funções irão ler o banco de dados quando o usuário acionar um evento e extrair e fornecer uma resposta. Os principais provedores de nuvem pública têm uma ou mais ofertas de FaaS. Eles incluem Amazon Web Services com AWS Lambda, Microsoft Azure com Funções Azure, Google Cloud com várias ofertas, e IBM Cloud com Funções de Nuvem IBM, entre outros.

Um ponto importante é como se dá o processamento de funções. Na Tabela 2, é demonstrado a capacidade de processamento por memória das funções da AWS Lambda (provedor usado neste trabalho), a quantidade de processamento é designada de acordo com a quantidade de memória alocada da função, algo diferente de outros serviços que se auto escalam conforme a demanda. As funções da AWS tem disponibilidade de alocação de 128 a 10240 MB, sendo 512 MB um tamanho suficiente para baixos processamentos e operações simples com serviços externos [Beswick 2021], como operação em banco de dados.

Memória (MB)	vCPUs
128 - 1769	1
1770 - 3538	2
3539 - 5307	3
5308 - 7076	4
7077 - 8845	5
8846 - 10240	6

Tabela 2. Memória da Lambda x CPUs para processamento.

As funções possuem um tempo máximo de atividade para evitar que a função Lambda seja executada de maneira interminável. Como o usuário está pagando pelos recursos da AWS que são usados para a execução da função, isso acaba sendo útil. Ao atingir o tempo limite o AWS Lambda encerra a execução, sendo inicializada novamente quando houver a próxima chamada. Consequentemente, ocasionando um *cold start* na próxima inicialização.

Em relação ao custo de execução de uma função lambda, a AWS cobra pelo número de solicitações de suas funções e pela duração, que é o tempo que o código precisa para ser executado. Quando o código começa a ser executado em resposta a um evento, o AWS Lambda conta uma solicitação. Ele cobrará o número total de solicitações em todas as funções usadas. A duração é calculada quando o código começa a ser executado até retornar ou ser encerrado, arredondado para o 1 milissegundo mais próximo. E a definição de valores do AWS Lambda depende da quantidade de memória que o usuário usou para alocar à função. Um exemplo desses valores pode ser aplicado a função de 512 MB, onde o valor cobrado é de 0,0000000067 por 1 milissegundo[2]. Vale dizer também que o tipo de

[2]https://aws.amazon.com/pt/lambda/pricing/

processador utilizado tem influência no valor, como os do tipo *Arm* que são mais baratos que os do tipo *x86*.

4. Materiais e Métodos

A comparação de desempenho teve como principal objetivo analisar o fenômeno cold start após as funções entrarem em inatividade. Os experimentos foram realizados usando o serviço AWS Lambda da AWS, com funções de leitura e persistência desenvolvidas em Java e Node.js, com memória alocada da função de 512 MB, um tamanho de alocação considerado mais que suficiente para as operações propostas. Quanto a CPU de processamento, será utilizada para ambas as funções a arquitetura arm64. Tendo como banco de dados o DynamoDB e a interface API Gateway para fazer a comunicação da função com a base de dados. A infraestrutura é fornecida pela AWS, através dos recursos disponibilizados na conta gratuita, esta que tem duração de 1 ano. Todos os testes foram realizados na região *us-east-1* da AWS, e todas as configurações foram realizadas através do console da AWS.

4.1. Metodologia

As funções deste experimento foram desenvolvidas através do serviço AWS Lambda da AWS. Os usuários do AWS Lambda podem criar funções, que são aplicativos independentes escritos em uma das linguagens disponíveis. Após o processo de deploy, o AWS Lambda executa essas funções de maneira eficiente e flexível. As funções podem executar qualquer tipo de tarefa de computação, desde servir páginas da Web e processar fluxos de dados até chamar APIs e integrar-se a outros serviços da AWS como o banco de dados.

O banco de dados utilizado para a interação das funções lambda foi o Amazon DynamoDB. É um serviço de banco de dados em nuvem NoSQL que oferece suporte a desempenho rápido e previsível em qualquer escala. O DynamoDB é usado para alimentar várias propriedades e sistemas de alto tráfego da Amazon, como Alexa, os sites Amazon.com e todos os centros de atendimento da Amazon. O DynamoDB é também um serviço de nuvem totalmente gerenciado. Usando a API do DynamoDB, os aplicativos criam tabelas, leem e gravam dados sem considerar onde essas tabelas são armazenadas ou como são gerenciadas. O DynamoDB é capaz de escalonar as tabelas de forma ilimitada. Não há limites predefinidos para a quantidade de dados que cada tabela pode armazenar, elas crescem elasticamente para atender a demanda das aplicações dos clientes [Perianayagam et al. 2022].

Para realizar a integração entre a função lambda e o banco de dados, foi utilizada a API Gateway da AWS. Um gateway de API (Interface de Programação de Aplicações) é uma ferramenta de gerenciamento de APIs que fica entre o cliente e os serviços de back-end. Ele funciona como um proxy inverso, que aceita todas as chamadas da API. Um gateway de API é uma maneira de desacoplar a interface do cliente da sua implementação de back-end. Quando um cliente faz uma solicitação, o gateway a divide em várias solicitações, as direciona para os locais adequados e produz uma resposta e faz o monitoramento. O serviço de gateway da AWS aceita apenas APIs do tipo REST e HTTP.

Para falar da aplicação geral das ferramentas da AWS citadas acima, a (Figura 1) ilustra a arquitetura serverless no momento que um cliente faz uma requisição à API. Quando o usuário invoca o endereço de requisição da API HTTP, o API Gateway trata

o tipo da requisição (GET, PUT, POST, DELETE) e encaminha a solicitação para a rota previamente configurada a sua função Lambda. A função do Lambda interage com o banco de dados DynamoDB e retorna uma resposta ao API Gateway. O API Gateway retorna a resposta para o usuário que a solicitou.

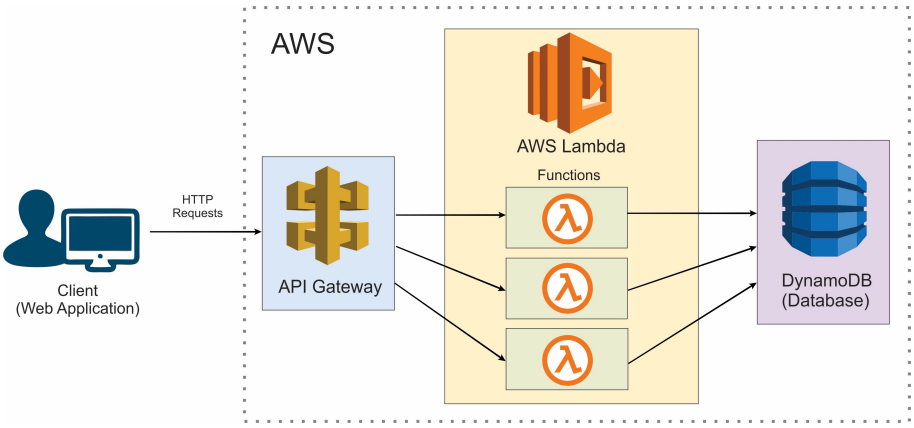


Figura 1. Estrutura da arquitetura serverless AWS utilizada no projeto.

O objetivo foi desenvolver duas APIs em Java e Node.js do tipo HTTP para leitura e persistência de dados, sendo estas integradas às funções lambda e ao banco de dados. O experimento se iniciou com a configuração do banco DynamoDB para armazenamento dos dados fictícios. Sendo originada uma base de dados com uma tabela contendo quatro colunas, sendo elas id, idade, nome e força; o id representa identificador único do registro chamado de chave primária. Apenas uma tabela será utilizada para as operações em Java e Node.

Após a criação da base de dados, iniciou-se o desenvolvimento das funções lambdas, vale ressaltar que as duas funções possuirão memória de alocação igual a 512 MB, a primeira a ser desenvolvida foi a da linguagem Node. Para algumas linguagens mais simples, é possível escrever o código da função diretamente no console da AWS, node é um exemplo destas. A função Node desenvolvida é capaz de abranger tanto a parte da persistência quanto a da consulta dos dados, sem a necessidade de uma segunda função. Essa distinção no código é feita através de um evento passado pela API Gateway, onde essa manda um valor representando o tipo da requisição solicitada, sendo elas GET ou PUT.

Para o desenvolvimento da lambda Java, algumas tratativas no desenvolvimento da função foram diferentes. Nesse caso o código da função não pôde ser escrito diretamente no console AWS. Foi necessário criar um projeto *Maven* em uma IDE de desenvolvimento, ser compilado, e após ser gerado um arquivo *jar*, sendo necessário fazer upload desse arquivo na função lambda. A forma como o evento do API Gateway é interpretado pela lambda Java é similar a lambda Node.

Dada a criação das funções, seguiu-se para a configuração do API Gateway, ponto responsável pela interação do usuário com a função. Neste foram definidas as rotas, para cada tipo de requisição, onde essas podem ser do tipo GET e PUT apenas e cada rota recebe um vínculo à sua função lambda responsável.

O fluxo de configuração da infraestrutura iniciou-se pelo DynamoDB, seguindo

para a função lambda e após o API Gateway. Ao criar uma API HTTP um link de acesso é disponibilizado, e ao digitá-lo em um navegador por exemplo, é obtido um retorno de acordo com a rota informada. A principal métrica a ser analisada é o tempo de resposta das funções lambda. Para obter métricas mais precisas, as requisições da API foram realizadas através da ferramenta Postman. Essa ferramenta simula requisições HTTP e ainda permite que as mesmas sejam armazenadas para uso em outras ocasiões. Além disso, a ferramenta é capaz de dar suporte no que diz respeitos às documentações e testes requisitados ao trabalhar com APIs.

5. Resultados

Nesta seção será apresentado os resultados das execuções das funções com base nas configurações especificadas na seção anterior. Foi avaliado o desempenho do tempo de serviço das funções para processar a primeira e as demais requisições.

O objetivo do trabalho teve como base várias referências de um período mais recente como [Silva and Carvalho 2019], [Fuerst and Sharma 2021], [Aapakallio-Autio 2021], que citam que o tempo máximo ocioso de uma lambda é de cerca de 15 minutos ou mais, porém não foi o cenário encontrado neste teste. Com as configurações padrões de 512 MB da AWS Lambda, as funções foram desativadas após um período de 6 a 7 minutos de inatividade. Essa incoerência acaba tendo uma relevância ainda maior quando relacionada a softwares com uma frequência menor de requisições.

Através da ferramenta Postman, foram realizadas as requisições pelo endereço disponibilizado na API Gateway, sendo feitas dez chamadas com cada uma das funções lambdas ainda frias. As Figuras 2 e 3 representam o tempo total de obtenção da resposta da API.

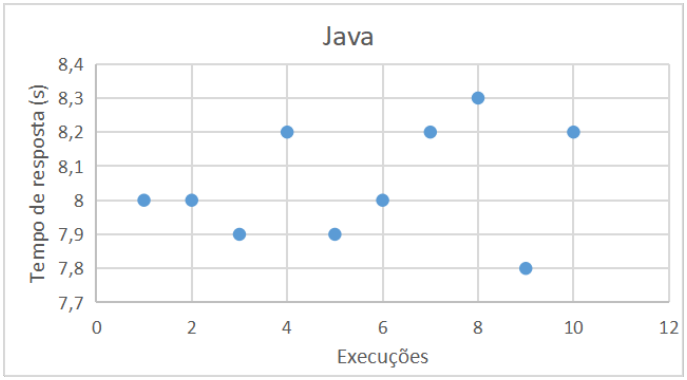


Figura 2. Tempo de execuções em segundos da lambda Java.

Nos gráficos também é possível notar que o tempo de cold start é variável nas duas linguagens, mas sendo mais volátil na função java. Essa acaba por ter uma latência maior devido a inicialização de sua maquina virtual JVM[3]. A duração do cold start aumenta ou diminui conforme a quantidade de memória alocada da lambda para ambas as funções. Uma fator que pode influenciar o cold start é a importação de uma quantidade maior que

[3]https://aws.amazon.com/pt/premiumsupport/knowledge-center/lambda-improve-java-function-performance/

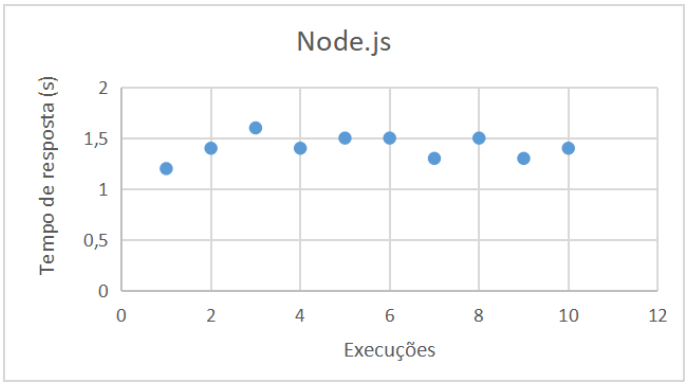
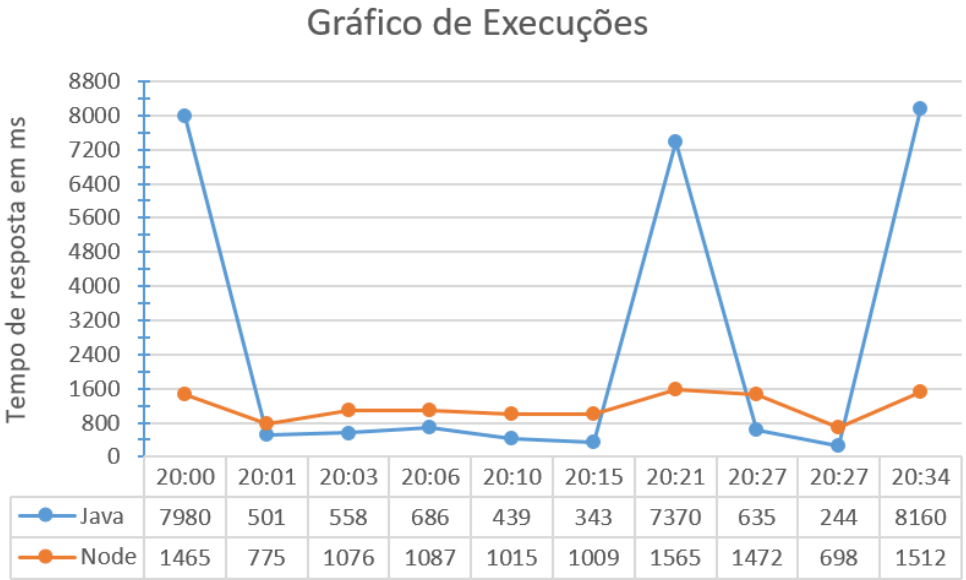


Figura 3. Tempo de execuções em segundos da lambda Node.

o necessário de dependências [Jignesh Solanki 2021], nas funções desenvolvidas foram implementadas apenas as dependências necessárias para utilização dos recursos da AWS.

Outra métrica obtida deu-se através de 10 execuções realizadas em um intervalo de 1 e 7 minutos, como é representado na Figura 4. A lambda java, quando em estado *warm*, obteve seu tempo de resposta mais rápido com 244 ms no menor intervalo, contra 698 ms da função node. Durante esse teste, também foi notado que o tempo de resposta da lambda node permanece regular por quase todas as execuções, enquanto a java permaneceu regular nas primeiras e foi decrescendo em suas últimas execuções *warm* antes do primeiro cold start. Foram realizadas duas execuções com o intervalo de 6 minutos, na primeira delas ambas as funções enfrentaram o cold start. Já na segunda execução, a função java permaneceu ativa enquanto a node entrou em inatividade. Quando as funções enfrentaram o cold start, ambas permaneceram na mesma média citada anteriormente.



	20:00	20:01	20:03	20:06	20:10	20:15	20:21	20:27	20:27	20:34
Java	7980	501	558	686	439	343	7370	635	244	8160
Node	1465	775	1076	1087	1015	1009	1565	1472	698	1512

Figura 4. Intervalo de execuções das funções entre 1 e 7 minutos.

6. Discussão

Após o processo de desenvolvimento, deploy, e testes das funções, foi possível determinar e analisar seus desempenhos de inicialização. Sendo notado que a função na linguagem node de fato obteve métricas superiores à função desenvolvida na linguagem java em relação ao tempo de cold start. O resultado da avaliação da latência do cold start ao inicializar uma função lambda é tido como satisfatório para o escopo deste trabalho. Ambas as funções obtiveram métricas de tempo de resposta claras para a definição de qual das duas linguagens se adéqua melhor ao objetivo do trabalho.

Um dado de suma relevância obtido foi o tempo real de atividade de uma função lambda. Em todos os testes das duas funções, o tempo de atividade ficou dentro do intervalo de 6 a 7 minutos, um tempo bem menor do que o previsto na elaboração da proposta deste trabalho que era de 15 minutos ou mais.

Uma das métricas mais importantes obtidas é a relação de execuções das duas funções. Nesta, os resultados trouxeram boas visões das funções, sendo uma a velocidade de inicialização da linguagem node, ficando com uma média de 1.4 segundos de inicialização. Significando um baixo tempo de espera para a resposta quando uma função enfrentar o *cold start*. Em relação a função java, esta atingiu a alta média de 8 segundos para obter a resposta. Por outro lado, a linguagem java obteve uma melhor resposta quando a função se encontra em estado *warm*, possuindo uma resposta mais performática que a lambda node em todas as execuções.

7. Conclusão

O padrão serverless tem diversos benefícios, dentre eles o custo e o escalonamento conforme o uso. Porém desenvolvedores e usuários de softwares criados nesse formato enfrentam um problema conhecido, tempos de resposta altos quando o tratamento de requisições precisa aguardar a inicialização da função solicitada. Esse problema é comumente conhecido como "*cold start*", que tem como contribuintes significativos a sobrecarga de orquestração da plataforma e a inicialização do ambiente virtualizado (contêiner ou VM) [Mohan et al. 2019]. E softwares onde sua utilização é reduzida esse problema torna-se mais pertinente, no qual suas funções não sendo requisitadas a todo momento, acabam por entrar em ociosidade mais facilmente.

Para analisar esse problema, foi elaborada uma proposta de desenvolvimento e análise sobre os impactos causados pelo tempo de execução da linguagem java, comparada ao desempenho de uma função lambda com as exatas mesmas funcionalidades desenvolvidas em node. O objetivo foi definir a melhor opção para softwares de menor porte, onde por não serem requisitados a todo momento, enfrentam o cold start com mais facilidade.

O experimento identificou que o tempo de atividade máximo no ambiente AWS Lambda é de 7 minutos, um tempo relativamente baixo que pode trazer impactos de inicialização caso uma função não seja requisitada novamente nesse período. Em relação ao tempo de resposta quando uma função enfrenta o cold start, a linguagem node obteve um desempenho consideravelmente melhor, tendo uma resposta até 82% mais rápida que a linguagem java. Por outro lado, a linguagem java obteve um melhor tempo de resposta que o node.js enquanto a função ainda permanece em atividade (*warm*).

Contudo, deve ser analisado os requisitos de um software serverless ao escolher uma linguagem de desenvolvimento. Softwares onde possuirão uma utilização menos frequente, como intervalos de requisições maiores que 6 minutos, a linguagem node se mostrou ser a melhor opção para este, oferecendo o benefício de um tempo de espera curto ao acionar uma função.

Referências

Aapakallio-Autio, R. (2021). From Elastic Beanstalk to Lambda: A comparative case study on the AWS tools. Master's thesis, Aalto University. School of Science.

Aaron Harris (2021). Multithreading in lambda? you'll need to use this much memory. Acesso em: 20 set. 2022.

Bargmann, C. (2019). An introduction to serverless computing. *Grundseminar M.Sc.*

Beni, E. H., Truyen, E., Lagaisse, B., Joosen, W., and Dieltjens, J. (2021). Reducing cold starts during elastic scaling of containers in kubernetes. In *Proceedings of the 36th Annual ACM Symposium on Applied Computing*, pages 60–68.

Bermbach, D., Karakaya, A.-S., and Buchholz, S. (2020). Using application knowledge to reduce cold starts in faas services. In *Proceedings of the 35th Annual ACM Symposium on Applied Computing*, SAC '20, page 134–143, New York, NY, USA. Association for Computing Machinery.

Beswick, J. (2021). Operating lambda: Performance optimization – part 2.

Buyya, R. e. a. (2018). A manifesto for future generation cloud computing: Research directions for the next decade. In *ACM Computing Surveys, V.51(5)*, pages 1–38. ACM.

Castro, P., Isahagian, V., Muthusamy, V., and Slominski, A. (2019). The rise of serverless computing. *Communications of the ACM*, 62:44–54.

Dantas, Jaime; Khazaei, H. L. M. (2022). Application deployment strategies for reducing the cold start delay of aws lambda.

Fuerst, A. and Sharma, P. (2021). Faascache: Keeping serverless computing alive with greedy-dual caching. ASPLOS '21, page 386–400, New York, NY, USA. Association for Computing Machinery.

Hosseini, M. and Sahragard, O. (2019). Aws lambda language performance.

Jegannathan, A. P., Saha, R., and Addya, S. K. (2022). A time series forecasting approach to minimize cold start time in cloud-serverless platform. In *2022 IEEE International Black Sea Conference on Communications and Networking (BlackSeaCom)*, pages 325–330.

Jignesh Solanki (2021). How to avoid aws lambda cold starts? Acesso em: 29 set. 2022.

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

Manner, J., Endreß, M., Heckel, T., and Wirtz, G. (2018). Cold start influencing factors in function as a service. In *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*, pages 181–188. IEEE.

Mohan, A., Sane, H., Doshi, K., Edupuganti, S., Nayak, N., and Sukhomlinov, V. (2019). Agile cold starts for scalable serverless. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*, Renton, WA. USENIX Association.

Pelle, I. e. a. (2020). Operating latency sensitive applications on public serverless edge cloud platforms. In *IEEE Internet of Things Journal, V.8(10)*, pages 7954–7972. IEEE.

Perianayagam, S., Vig, A., Terry, D., Sivasubramanian, S., Sorenson III, J. C., Mritunjai, A., Idziorek, J., Gallagher, N., Elhemali, M., Gordon, N., et al. (2022). Amazon dynamodb: A scalable, predictably performant, and fully managed nosql database service. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 1037–1048.

Savage, N. (2018). Going serverless. *Commun. ACM*, 61(2):15–16.

Silva, M. and Carvalho, M. (2019). Análise de mecanismos de serverless computing em ambientes de nuvens computacionais. In *Anais Estendidos do XXXVII Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos*, pages 225–232, Porto Alegre, RS, Brasil. SBC.

Silva, P., Fireman, D., and Pereira, T. E. (2020). Prebaking functions to warm the serverless cold start. In *Proceedings of the 21st International Middleware Conference*, Middleware '20, page 1–13, New York, NY, USA. Association for Computing Machinery.

Trilles, S., González-Pérez, A., and Huerta, J. (2020). An iot platform based on microservices and serverless paradigms for smart farming purposes. *Sensors*, 20(8).