Avaliação de modelos de implantação de funções serverless no serviço AWS Lambda

Gabriel Duessmann
Programa de Pós-Graduação em Computação Aplicada
Departamento de Ciência da Computação - DCC
Universidade do Estado de Santa Catarina - UDESC
Joinville, Brasil
gabriel.duessmann@edu.udesc.br

Adriano Fiorese
Programa de Pós-Graduação em Computação Aplicada
Departamento de Ciência da Computação - DCC
Universidade do Estado de Santa Catarina - UDESC
Joinville, Brasil
https://orcid.org/0000-0003-1140-0002

***Abstract*—With the advancement of computing and serverless services in the last couple of years, this area has been growing rapidly. Currently, most cloud providers offer serverless services, in particular at Amazon, they have AWS Lambda to create Functions as a Service (FaaS). There are at least two ways to implement it: by compressing the source code and files into a compacted folder in a ZIP format; the second way is through a container image, which has the running application and its dependencies. Based on the approach selected, the function's performance, cost and initialization time may vary. This paper takes into account these metrics and compares the aforementioned ways of deployment. Furthermore, it aims to discover which approach is the most adequate. Experiments conducted at AWS Lambda show that functions created with compressed ZIP folders present advantages, regarding their initialization time during cold start mode, and cost.**

***Keywords*—Serverless Function, AWS Lambda, Container Image, Evaluation, Performance, Cost, Start Time**

***Resumo*—Com o avanço da computação em nuvem e serviços *serverless*, mais foco essa área vem ganhando nos últimos anos. Provedores de nuvem oferecem serviços relacionados a *serverless*, e em particular, a Amazon disponibiliza o AWS Lambda para a criação de funções *serverless*. Existem ao menos duas formas de implantá-los: através da compressão da pasta do projeto, que contém o código fonte e arquivos executáveis, em formato ZIP; e a segunda maneira na qual a aplicação e suas dependências estão em uma imagem de contêiner. Dependendo da abordagem escolhida, o desempenho, o custo e o tempo de inicialização podem variar. Levando em consideração essas métricas, este trabalho visa compará-las entre as duas abordagens de implantação mencionadas e tem como objetivo descobrir se uma das abordagens apresenta ser mais adequada do que outra. Experimentos conduzidos visando tal comparação demonstram que a criação de funções utilizando pastas compactadas apresentam vantagens, principalmente no tempo de inicialização da função quando está em modo de partida fria, e no custo.**

***Palavras-chave*—Função Serverless, AWS Lambda, Imagem de Contêiner, Avaliação, Desempenho, Custo, Tempo Inicialização**

I. Introdução

Serverless é um modelo de computação em nuvem na qual os provedores ofertam serviços de provisionamento dinâmico de servidores configurados para que seus clientes executem suas aplicações. Sendo assim, o provedor do serviço em nuvem *serverless* é quem tem a responsabilidade do provisionamento, escalabilidade e segurança das aplicações implantadas nesse modelo [1]. Isso traz maior praticidade para que desenvolvedores e companhias implementem suas aplicações sem a preocupação em contratação e manutenção de infraestrutura necessária para sua execução. Cada aplicação implantada nesse modelo, é chamada de função *serverless*, a qual deve executar independentemente da infraestrutura ofertada pelo provedor.

Ao comparar o modelo mais recente *serverless* com aplicações monolíticas que já estão difundidas no mercado há anos, se diferem de tal forma que as funções *serverless* possuem um escopo de código e funcionalidades menores, não há necessidade de configurações de servidor para serem executadas e são automaticamente escaláveis conforme à demanda dos recursos contratados e alocados. As aplicações monolíticas por sua vez, compõem todo o código de um sistema, incluindo configurações de servidores e banco de dados, e portanto seus códigos e estruturas tendem a ser mais extensos. A escalabilidade, além de não ser padrão, pode ser um desafio devido a grande quantidade de recursos computacionais necessários que precisam ser alocados à aplicação.

Apesar da maior facilidade de desenvolvimento utilizando funções *serverless*, no sentido da abstração da infraestrutura necessária para a execução e atendimento da demanda elástica da aplicação, a implantação da aplicação nos provedores de nuvem demanda cuidados, pois as configurações do ambiente são feitas por eles e os desenvolvedores não têm acesso para alterá-las. Tais cuidados estão relacionados a configuração dos recursos necessários para executá-las, bem como a forma em

que a aplicação é instanciada e desativada de acordo com o seu uso (requisições feitas pelos usuários) e ociosidade (período sem requisições). A medida que a aplicação fica ociosa por alguns minutos, o provedor desaloca os recursos computacionais da função, e a mesma fica inoperante. Quando uma nova execução é requisitada à função, e a mesma se encontra nesse estado inoperante, o serviço a instancia novamente com os recursos alocados, o que é chamado de partida lenta. Quando é feita uma invocação (por parte do cliente) à função que se encontra no estado inoperante, o tempo de resposta também é maior na primeira requisição.

O provedor de nuvem AWS oferece o modelo *serverless* em seu serviço chamado de AWS Lambda. Esse serviço permite a implantação de aplicações como funções *serverless*, testá-las, realizar chamadas às funções, e integrá-las com os outros serviços da plataforma. Algumas maneiras para implantar as funções são ofertadas de forma visual e manual através do *website* da AWS. Dentre essas maneiras, foram selecionadas particularmente duas: a maneira mais tradicional (similar a outros provedores) através de uma pasta compactada em formato ZIP; e a partir de uma imagem de contêiner, opção introduzida pela empresa em 2020 [2].

Dados esses modelos de implantação no serviço AWS Lambda, este trabalho busca avaliá-los a partir de experimentos realizados na plataforma e analisar suas características com base nos dados coletados. Os dados selecionados para essa avaliação foram: custo, desempenho e tempo de inicialização em partida fria. Como contribuição científica, este trabalho busca responder as seguintes perguntas referente aos dois modelos analisados:

- **RQ-1:** Qual apresenta ser mais vantajoso, com base nas métricas de desempenho e tempo de inicialização?
- **RQ-2:** Como a escolha do modelo pode impactar no custo?

Para o ambiente dos experimentos, foi desenvolvida uma aplicação própria em Java e implantada como função *serverless* nos dois modelos. Essa função espera como entrada do usuário uma região no mapa terrestre, executa o código da aplicação (função) para descobrir a data e hora local, e retorna o resultado. Os dados das execuções foram coletados pelo próprio serviço AWS Lambda, que disponibiliza uma aba com console de testes. Não houve incidência de custos para realizar os experimentos, pois foi utilizado o acesso *Free Tier* aos serviços da AWS, que permite o acesso a diversos serviços de forma gratuita. Nesse contexto, a análise e comparação de custo se deu com base nos valores estimados que são informados pela empresa Amazon.

Esse artigo é estruturado da seguinte maneira: A Seção II apresenta o referencial teórico para esclarecer termos relacionados a nuvem e ao provedor AWS, incluindo o serviço AWS Lambda, bem como apresenta características utilizadas na avaliação. A Seção III elenca trabalhos relacionados pertinentes ao tema, bem como suas características e como este trabalho se difere dos demais. Na Seção IV, é apresentada a metodologia, as configurações dos ambientes, a proposta das avaliações, bem como os experimentos realizados e os valores das métricas coletadas. Por fim, a Seção V conclui o artigo respondendo as perguntas estabelecidas.

II. Referencial Teórico

Com o avanço da computação em nuvem, cada vez mais serviços estão sendo migrados ou implementados nesse modelo. Os provedores de nuvem, por sua vez, estão dedicados a criarem e disponibilizarem recursos que melhoram a qualidade e praticidade de seus serviços a fim de atender um maior número de clientes. Um desses serviços é chamado *serverless*, que executa uma aplicação sem que o cliente necessite configurar o servidor e demais infraestruturas para executar uma aplicação. Nesse sentido, esta seção trata de temas pertinentes a avaliação proposta passando pelo modelo de computação em nuvem; definindo funções *serverless*; e os modelos de implantação de uma função *serverless*, em particular, no serviço AWS Lambda, e seus correlacionados.

A. Computação em Nuvem

A computação em nuvem é um modelo de disponibilização de serviços computacionais compartilhados, como rede, servidores, armazenamento, aplicações e serviços que podem ser provisionados e publicados sob demanda rapidamente e com baixa configuração e complexidade por parte dos seus clientes [3]. Os provedores de nuvem são responsáveis por cuidar, controlar e ofertar serviços em nuvem para usuários finais ou entidades que necessitam de poder computacional, mas que não pretendem ter o custo de propriedade e operação dos equipamentos físicos envolvidos. Conforme os clientes utilizam os serviços ofertados, pagam sob demanda de acordo com a utilização dos mesmos ou conforme acordo firmado com o provedor. Com a computação em nuvem, *data centers* e infraestruturas de empresas começaram a ser migrados para nuvem devido a agilidade e praticidade dos serviços ofertados pelos provedores. Como consequência, as aplicações também precisaram ser migradas, o que levou a novas formas de aproveitamento dos recursos computacionais disponíveis, tendo em vista a otimização dos mesmos e do custo, dado o modelo de precificação baseado no pagamento do que é usado sob demanda.

B. Função serverless

O modelo de computação *serverless*, conhecido por funções *serverless* ou modelo *Function as a Service* (FaaS), é

um modelo de serviço de computação em nuvem no qual os provedores disponibilizam ambientes pré-configurados em diversas linguagens de programação para executarem aplicações em nuvem. Esses ambientes são configurados de maneira que atendam principalmente a demanda elástica da aplicação, garantindo a boa execução das funções. Ainda é necessário que o usuário do serviço de computação *serverless* faça a configuração inicial, configurando os componentes de *hardware* e especificações desejadas. Entretanto, aplicar e controlar as configurações mais robustas fica a cargo do provedor, como processamento paralelo e escalabilidade de recursos.

Visto que o provedor gerencia grande parte das configurações do ambiente para os usuários no modelo *serverless*, torna-se possível que os desenvolvedores se especializem nas suas implementações de códigos, implantando-os como funções. Essa facilidade proporciona agilidade para o sistema, principalmente podendo entregar valor e funcionalidades mais rapidamente, se comparado ao modelo monolítico, ou outros serviços ofertados pelos provedores de nuvem, por exemplo, instanciação de máquinas virtuais.

Além da praticidade na entrega de funcionalidades como funções *serverless*, esse modelo gerencia todo a parte da escalabilidade. A escalabilidade nesse caso contempla a instanciação da função para sua utilização pelos usuários finais da aplicação, de forma a atender a demanda de requisições, bem como o gerenciamento dessas instâncias quando não estão necessariamente atendendo às requisições dos clientes finais. É possível utilizar estratégias de otimização com vistas a reutilização dos recursos para o atendimento da escalabilidade. Uma das estratégias utilizadas para gerenciar a alocação dos recursos tendo em vista a escalabilidade é chamada de *cold start*, ou partida lenta. Nesse caso, conforme a função deixa de ser utilizada pelos usuários finais, os recursos de *hardware* alocados passam a ficar ociosos, e portanto, os provedores desalocam parte desses recursos para obter uma maior economia em seus servidores, possibilitando o uso desses recursos em outras funções *serverless* ou serviços por outros clientes. Quando a aplicação é invocada novamente através de uma requisição feita por um usuário, esta pode necessitar que seus recursos sejam realocados, de tal forma que uma nova instância seja criada, procedimento chamado de partida lenta, ou partida fria [4]. Após ter os recursos reativados, essa instância permanece ativa enquanto é utilizada através das chamadas à função, e por mais alguns minutos em modo ocioso sem ter requisições. Se nesse período durante o tempo ocioso houver um novo acesso à função por usuários finais, a função *serverless* continua ativa, com os recursos computacionais disponíveis e alocados a ela. Assim, ela pode ser utilizada de imediato, pois possui seus recursos operantes para o atendimento de nova requisição, sem que seja necessária uma nova instância da função. Esta, após passar minutos em modo ocioso sem execuções, tem seus recursos desativados pelo provedor. Esse ciclo continua alternando entre partida fria e partida quente conforme a sua utilização. Apesar dos provedores não disponibilizarem o tempo exato em que uma função precisa estar ociosa para passar do estado de partida quente para a partida fria, o trabalho [5] chegou empiricamente no valor aproximado de 20 minutos. Adicionando uma tolerância de 10 minutos, essa trabalho considera que uma função que está pelo menos 30 minutos ociosa, deixa de estar ativa e tem seus recursos desalocados.

C. *Contêineres*

Contêineres fornecem às aplicações um ambiente computacional para a execução de aplicações, configurado com todas as suas dependências para ser executado. Os contêineres isolam a aplicação desejada dos programas e processos externos que executam no sistema operacional principal da máquina. Portanto, pode ser definido como um mecanismo de empacotamento de aplicações [6].

A utilização de contêineres se tornou amplamente popular devido a sua portabilidade para de migrar aplicações entre diferentes ambientes, sem a ocorrência de problemas ocasionados por configurações distintas nas máquinas hospedeiras. Sem esse modo de virtualização, toda vez que as aplicações migravam para um novo ambiente, havia chance de ocorrer erros, pois dois ambientes raramente são totalmente idênticos em questão de *software* e *hardware*, e portanto, podem não ter as configurações e dependências necessárias para a execução da aplicação [6]. Sem o uso de contêineres ou alguma técnica de virtualização, ou seja, em *bare metal*, para cada vez que a aplicação executasse em uma máquina diferente, seria necessário antes disso, instalar suas dependências, programas de terceiros, configurar o ambiente, e entre outros.

Para que uma aplicação seja portável entre máquinas com contêiner é necessário que o *host* hospedeiro, em seu sistema operacional, esteja preparado para a execução de contêineres. Geralmente, alguma instalação e configuração de um *software* gerenciador de contêineres é necessária. Em um ambiente de nuvem computacional, a depender do modelo de serviço (ex: *Infrastructure as a Service* (IaaS), *Software as a Service* (SaaS), ou mesmo *Function as a Service* (FaaS)) é necessário que o cliente faça tal instalação, ou a mesma é disponibilizada como parte do serviço pelo provedor de nuvem. Além disso, é preciso de uma imagem computacional da aplicação com as especificações que são necessárias para executá-la. Com a imagem gerada, o contêiner pode ser replicado para outras máquinas de diferentes *hardwares* e sistemas operacionais (SOs), e eventualmente diferentes provedores de nuvem, mantendo o mesmo funcionamento da aplicação. Isso é possível porque os

contêineres são uma forma de virtualização leve, que pode incluir seu próprio SO [7]. Sendo assim, pode-se ter uma imagem de contêiner para cada função *serverless* que se deseja criar e especificar qual imagem utilizar ao criar uma nova função no serviço AWS Lambda.

D. Serviços AWS

Um dos maiores provedores de nuvem da atualidade é a AWS (Amazon Web Service), que oferta centenas de serviços disponíveis na Internet, sendo que este trabalho utiliza os serviços: AWS Lambda e AWS ECR (*Elastic Container Registry*). O serviço *serverless* da Amazon é chamado de AWS Lambda [8], e nele, consegue-se criar funções para executar aplicações sem precisar provisionar e gerenciar servidores. Para implantar uma aplicação em função *serverless*, é necessário que o desenvolvedor importe em seu projeto as bibliotecas da AWS Lambda, específicas para cada linguagem de programação, e implemente a interface disponibilizada pela biblioteca específica. Essa interface serve para definir um método principal dentro da aplicação. Ao criar a função, na etapa de configuração, é necessário informar o caminho e o nome desse método principal, para que seja usado como ponto de entrada para a função invocar a aplicação e passar os parâmetros de entrada.

A AWS disponibiliza diversas maneiras de implantar (criar) funções através do seu *website* de forma visual e intuitiva. Em particular, como objeto de estudo deste trabalho, são analisados os modelos através de: pasta compactada em formato ZIP e imagem de contêiner. Há um terceiro modelo, através da IDE (*Integrated Development Environment*) integrada ao *website* na AWS, entretanto foi escolhido removê-lo dos experimentos, pois IDEs de navegadores não são tão intuitivas e não facilitam tanto o desenvolvimento como IDEs tradicionais, e portanto não são utilizadas com frequência por desenvolvedores e empresas do setor. Outro fator que levou a essa escolha é que a IDE integrada está disponível apenas para poucas linguagens de programação na plataforma (NodeJS, Python e Ruby), o que limita o seu escopo de utilização.

Sobre os dois modelos de implantação escolhidos, para abordagem de pasta compactada, é necessário compilar os arquivos fontes em um arquivo executável, e compactar a pasta do projeto em formato ZIP e fazer o *upload* diretamente no serviço AWS Lambda. Com o uso de uma imagem de contêiner, é necessário que o desenvolvedor tenha a imagem da aplicação publicada no serviço AWS ECR, que é o repositório de imagens de contêineres, e selecionar a respectiva imagem quando criar a função.

Além das características citadas acerca do servico AWS Lambda, e seus coadjuvantes relacionados ao modelo *serverless*, a arquitetura de *hardware* onde é executada a função também pode ser escolhida junto ao provedor AWS. Assim, ao criar uma nova função, esta pode ser criada sob arquitetura *arm64* ou *x86_64*. Embora o provedor AWS seleciona a arquitetura *x86_64* como padrão, a arquitetura *arm64* se destaca por possuir menor custo de execução e atingir bons resultados de desempenho [9].

AWS ECR é um serviço de repositório para armazenar imagens de contêineres. Algumas dessas ferramentas de contêineres que podem ser usadas para tal finalidade são: Podman [10] e Docker [11]. Dado a grande popularidade de Docker, ele foi escolhido como ferramenta de uso nos experimentos. O desenvolvedor deve criar a imagem para executar a aplicação em sua máquina local a partir de um arquivo *Dockerfile* e publicá-la no repositório da nuvem [12], para que fique disponível na AWS. Ao ter a imagem disponível no repositório AWS ECR, pode ser utilizada em diversos serviços do provedor, e para este trabalho em específico, para criar funções na AWS Lambda.

III. Trabalhos Relacionados

Modelos FaaS não são tão recentes, e apesar de trazerem praticidade para implementação de aplicações, há ainda espaço para estudos analisarem pontos de melhoria.

O trabalho [13] aborda estratégias para diminuir o *cold start* (partida lenta) e compara o impacto no tempo ao instanciar uma função através de arquivo compactado e via imagem de contêiner. Apesar de propor soluções para minimizar o tempo de inicialização, o trabalho citado não aborda custo.

Em [14], são investigados os fatores que afetam o desempenho de funções *serverless* e como também são examinados os resultados em opções de contêineres, diferentes linguagens de programação e alternativas de compilações. Porém, os autores do trabalho citado não levam em consideração o custo para executar a função e tempo de inicialização da partida fria.

Na publicação [15], os autores compararam os custos de executar aplicações em monolito, microserviço e função Lambda na AWS. Das três arquiteturas, AWS Lambda obteve o menor custo, reduzindo os custos de infraestrutura em 70%.

Em [16] é conduzido um estudo de caso na AWS Lambda que avalia o uso de memória, a escalabilidade e a partida lenta conforme a escolha das linguagens de programação. Os experimentos são conduzidos a partir de operações de CRUD (escrita, leitura, atualização e remoção). Os resultados apontaram que as linguagens Python e NodeJS obtiveram os melhores resultados. Entretanto, não é destacado o modelo de implantação das funções *serverless*.

Apesar de trabalhos anteriores já terem comparado os modelos de implantação entre arquivo compactado e imagem de contêiner, o trabalho atual busca relacionar suas diferenças considerandos os fatores de desempenho, custo e tempo de inicialização.

A Tabela I compara as características dos trabalhos relacionados e pontua como esse artigo se difere dos demais. As colunas de arquivo compactado e imagem de contêiner são referentes ao modo de implantação da aplicação. As demais colunas apresentam desempenho, custo e tempo de inicialização como métricas referentes às funções, e por fim as linguagens de programação utilizadas nos experimentos.

IV. Metodologia e Experimentos

Este trabalho propõe avaliar dois métodos de implantação de uma função *serverless* no serviço AWS Lambda: através de pasta compactada (formato ZIP) e imagem de contêiner. Através dos modelos implantados, foram executados experimentos, e coletados os dados para realizar a análise.

Os experimentos buscam auxiliar na descoberta se uma das abordagens apresenta ser mais vantajosa e corroborar com as questões de pesquisa (RQ-1 e RQ-2). Para chegar em tais conclusões, métricas pertinentes ao serviço foram analisadas, sendo elas: o custo, o desempenho com base no consumo de memória, e o tempo de inicialização em partida lenta.

A. Função e ambiente

Primeiramente, foi desenvolvida uma aplicação para ser implantada como função *serverless* [1]. Essa aplicação espera uma região geográfica como um parâmetro de entrada (ex.: *Europe/Berlin*), obtém a data e horário atual da região, e retorna-os como resposta. A aplicação foi desenvolvida na linguagem de programação Java 21. Essa escolha se deu devido a sua popularidade no meio Web e facilidade de implementar a interface da biblioteca requerida para o uso como função no serviço AWS Lambda.

Como o objetivo desse trabalho não é comparar o uso de recursos computacionais (processamento e memória, sem uso de disco) entre funções, foi desenvolvida uma função relativamente simples, que demandasse baixo processamento dos recursos computacionais, portanto, foi possível alocar os recursos de *hardware* mínimos disponíveis (memória de 128 MB) para a função no serviço AWS Lambda, sem impactar nos resultados dos experimentos realizados.

As etapas para implantação de uma função no serviço AWS Lambda são semelhantes, porém possuem algumas particularidades:

Arquivo compactado: Gera-se o arquivo executável da aplicação, que para a linguagem Java é chamado de JAR (Java ARchive). A pasta que contém o código fonte, juntamente com o arquivo JAR deve ser compactada em formato ZIP. Ao criar a função no serviço AWS Lambda, deve ser feito o *upload* dessa pasta diretamente para o serviço.

[1] https://github.com/gabrielduessmann/date-hour-lambda-aws

Imagem de contêiner: Adiciona-se um arquivo extra no projeto contendo as configurações e dependências para gerar o *build* da imagem de contêiner (geralmente nomeado *Dockerfile*). Nele deve constar as instruções para a instalação do sistema operacional e as dependências do ambiente. Essa imagem gerada no ambiente local do desenvolvedor, deve ser publicada no serviço AWS ECR (serviço de repositório de imagens) para que fique disponível na nuvem. Ao criar a função, deve-se escolher a opção por imagem de contêiner, e informar o respetivo *link*. Esse *link* é uma referência única da imagem disponível no serviço AWS ECR, e dessa forma a função *serverless* é criada a partir da imagem.

Além de selecionar o modelo de implantação, é necessário escolher algumas configurações de recursos computacionais. Em particular, nesse trabalho foram selecionadas apenas as configurações de arquitetura e memória. Para a arquitetura, optou-se pela *arm64* devido ao menor custo [9], bem como a pouca necessidade de processamento. A memória escolhida foi a mínima, de 128 MB, visto que as funções não demandam armazenamento efêmero e grande poder computacional para operarem efetivamente. A CPU é alocada automaticamente e proporcionalmente à memória escolhida, sem opção de alteração por parte dos clientes.

Para realizar os experimentos, foi utilizado o console de testes disponível dentro do serviço AWS Lambda através do *website* da AWS. Ao visualizar as configurações da função, há uma aba para testá-la, sendo possível passar parâmetros de entrada e visualizar a saída. Além disso, o console de testes disponibiliza as métricas da função a cada invocação, as quais são coletadas para análise deste trabalho. Por ser uma ferramenta interna do serviço, obtém-se os valores reais das métricas a cada execução, não sendo afetados por fatores externos, como a conexão de Internet do usuário, ou o tempo de ida e volta da requisição entre o caminho do origem e destino.

O fluxo de testes, bem como o ambiente configurado é apresentado no diagrama da Figura 1. No cenário, inicialmente, o usuário faz a requisição a uma função (através do console de testes) pelo seu navegador, a função executa a aplicação (seja via pasta compactada ou imagem de contêiner) e retorna o resultado ao usuário, bem como as métricas daquela execução em particular.

Para a coleta de dados, foram realizadas 40 chamadas às funções em modo de partida lenta, modo em que a função precisa realocar seus recursos, e consequentemente tem um maior atraso no tempo de resposta. Conforme definido na Seção II-B, esse trabalho considera 30 minutos como o tempo em que uma função deixa de estar operante no estado de partida quente e passa para a partida lenta. Logo, chamadas à função foram realizadas com um intervalo de 30 minutos.

Tabela I
COMPARAÇÃO DE TRABALHOS RELACIONADOS

Artigo	Arquivo compactado	Imagem de contêiner	Desempenho	Custo	Tempo de inicialização	Linguagens
Dantas [13]	Sim	Sim	Não	Sim	Sim	NodeJS, Python e Java
Elsakhawy [14]	Não	Sim	Sim	Sim	Não	-
Villamizar [15]	Não	Não	Não	Sim	Não	Java 7
Rodríguez [16]	Não	Não	Sim	Não	Sim	NodeJS, Python e Java
Trabalho atual	Sim	Sim	Sim	Sim	Sim	Java 21

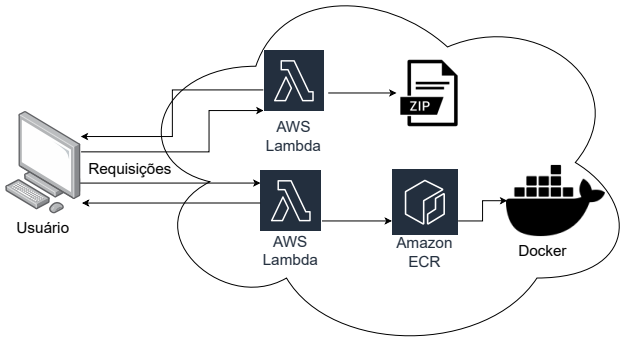

Figura 1. Diagrama do ambiente de testes

B. Custo

Para a execução desses experimentos na AWS, incluindo os serviços utilizados, não houve incidência de cobrança do provedor por terem sido utilizados apenas serviços e configurações dentro do nível *Free Tier*. Esse nível possibilita que clientes utilizem serviços de forma gratuita, desde que atendam as restrições existentes. Portanto, para comparar o custo entre as duas abordagens de implantação, são usados os valores base de precificação do provedor de nuvem AWS.

Os fatores que podem refletir nos custos do serviço AWS Lambda são: a escolha do modelo de implantação das funções, a configuração do ambiente, a quantidade de chamadas ou requisições a função, o tempo de execução por requisição, e a região geográfica de disponibilização da função. Dentre esses fatores, a quantidade de requisições e o tempo de execução são desconsideradas da análise, pois são muito particulares de cada implementação e regra de negócio a ser seguida.

A análise de custos relacionado ao modelo de implantação das funções pode ser dividido entre as aplicações com tamanhos de até 10 MB, e superiores a esse tamanho. O serviço AWS Lambda oferece as aplicações que se enquadram na primeira divisão (de até 10 MB) e em formato de pasta compactada (ZIP), o *upload* direto para o serviço e isento de custos de armazenamento. Aplicações maiores de 10 MB ou geradas através de imagens de contêineres (independente do tamanho) precisam ser armazenadas em seus respectivos serviços do provedor. Para pastas compactadas, o respectivo serviço na AWS é o S3 (*Simple Storage Service*); e AWS ECR para imagens de contêiner. Ambos os serviços AWS S3 e ECR [17] precificam seu custo de armazenamento baseado no tamanho da pasta ou imagem. Vale ressaltar que as imagens de contêiner tendem a terem tamanhos maiores, pois compõem as configurações do ambiente necessárias para executarem a função, incluindo o sistema operacional, e portanto, podem ter uma significância maior no custo final.

As funções na AWS Lambda também são cobradas conforme a escolha dos recursos computacionais alocados para o ambiente. Os recursos são: arquitetura (*arm64* ou *x86_64*), memória disponível (entre 128 MB e 10240 MB), e armazenamento efêmero (entre 512 MB e 10240 MB). Entre as arquiteturas oferecidas, a *arm64* possui um menor custo [9]. Para alocação de memória e armazenamento efêmero, o custo é proporcional ao tamanho alocado.

Em geral, a zona geográfica selecionada para o provisionamento dos serviços em nuvem influencia nos valores finais, entretanto, aplica-se a todos os serviços ofertados pelo provedor (não apenas a AWS Lambda), e como este trabalho utilizou apenas a região *us-east-1*, (localizada em Virgínia, nos EUA), não pode ser utilizada como base para comparação entre os modelos implantados. E caso estivessem disponibilizados em regiões distintas, também seria injusto fazer tal comparação, visto que os preços dos serviços variam por região.

C. Desempenho

Como métrica de desempenho da aplicação, o trabalhou levou em consideração o consumo de memória RAM máximo no ambiente experimentado ao executar funções *serverless* que estavam em modo de partida lenta.

Conforme a Figura 2, nota-se que o uso de memória máximo com a abordagem de imagem de contêiner se manteve constante, enquanto que a pasta compactada, há um desvio padrão maior, o que indica que a função no modelo de imagem de contêiner apresenta um controle do uso de memória mais estável.

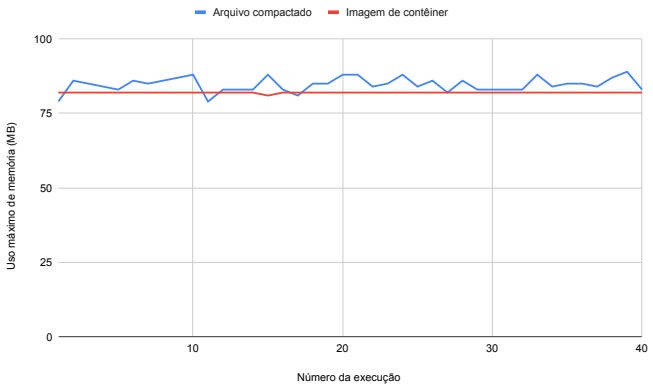

Figura 2. Gráfico de uso de memória máximo em funções *serverless*

A Figura 3 apresenta a consolidação da média de ambas as abordagens, corroborando os resultados da Figura 2, e demonstra que a implantação da função feita a partir da imagem de contêiner obteve melhores resultados em relação ao consumo de memória, isto é, consome menos memória para executar a função. Entretanto, a diferença é muito baixa para ser considerada relevante.

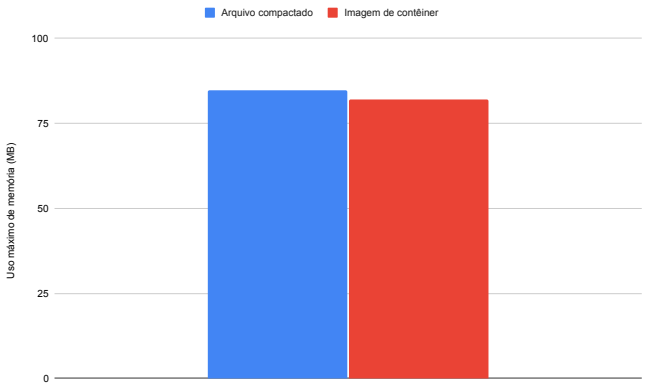

Figura 3. Gráfico da média de uso de memória máximo em funções *serverless*

A diferença no uso de memória das funções entre os dois modelos, apesar de ser relativamente baixa, não altera o valor do custo final, pois a precificação é baseada na memória alocada para a função, que nos experimentos foram de 128 MB, e a memória utilizada durante as execuções não é considerada.

D. Tempo de inicialização em partida fria

A Figura 4 apresenta o tempo de inicialização das funções em cada modelo de implantação, para 40 execuções, chamadas com um intervalo de 30 minutos entre cada execução, garantido que as funções deixassem de estar ativas (em partida quente) e passassem para o modo de partida fria. Particularmente, observa-se a constância nos resultados obtidos para as funções em pasta compactada, resultado contrário obtido na Figura 2 para o uso de memória, o que pode indicar que o provedor AWS realiza otimizações no ambiente para funções implantadas nesse modelo, para obter maior estabilidade e assim conseguirem menores tempos de inicialização.

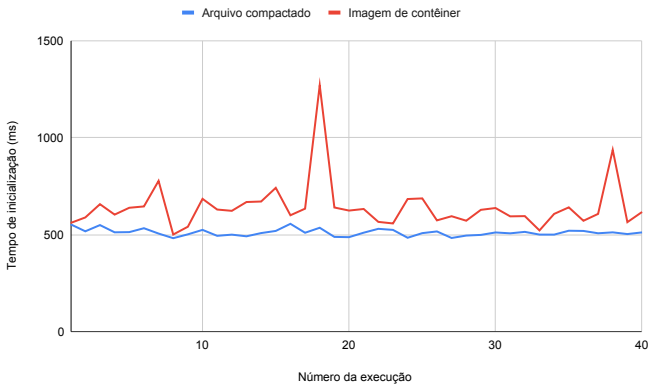

Figura 4. Gráfico do tempo de inicialização em funções *serverless*

A Figura 5 apresenta a média dos tempos de inicialização em partida fria. Essa métrica está diretamente ligada ao custo, pois o tempo total de execução é um dos fatores de precificação para o serviço AWS Lambda, e quanto maior for o tempo para inicialização, maior é o tempo de execução total. Portanto, as funções com menor tempo de inicialização tem um menor custo, bem como um melhor tempo de resposta aos usuários finais. Com base nos experimentos, o modelo de implantação através de pasta compactada apresentou o melhor tempo de inicialização.

V. Considerações Finais

Esse trabalho avaliou dois modelos de implantação de funções *serverless* no serviço AWS Lambda. No modelo de pasta compactada (ZIP), o modelo de implantação é mais simplificado, pois é possível realizar o *upload* da aplicação direto para o serviço AWS Lambda, enquanto que na abordagem com imagem de contêiner é necessário configurações extras, como a geração do *build* da imagem e publicação no serviço AWS ECR.

Experimentos foram realizados no ambiente do provedor AWS para corroborar com as perguntas de pesquisa estabelecidas, referente as métricas de desempenho, tempo de inicialização, e custo. Com base nos cenários experimentado e

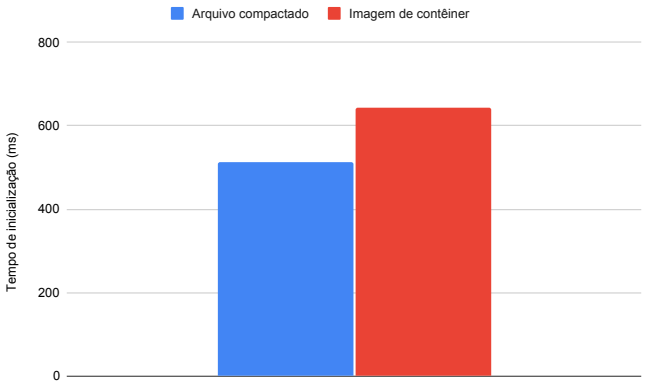

Figura 5. Gráfico da média do tempo de inicialização em funções *serverless*

dados obtidos em cada modelo de implantação, pode-se inferir que a implantação via pasta compactada (ZIP) apresentou as principais vantagens, sendo elas: o menor custo de implantação, visto que pode ser feito o *upload* da aplicação direto para o serviço AWS Lambda, sem custo para o armazenamento da função na nuvem; e o menor tempo de inicialização quando em partida lenta, oferecendo tempo de resposta mais rápido aos usuários finais, e reduzindo o custo por tempo de execução nas chamadas à função. Nesse sentido, ambas RQ-1 e RQ-2 são respondidas.

Como trabalho futuros, pode-se estender a comparação para as demais linguagens de programação suportadas pelo serviço AWS Lambda, assim como comparar os modelos de implantação em diferentes provedores de nuvem, como Google Cloud e Microsoft Azure. Outro aspecto a ser avaliado é a arquitetura na qual a função *serverless* é executada, podendo ser *x86_64* ou *arm64*. Neste trabalho, utilizou-se apenas a arquitetura *arm64*, havendo espaço para explorar a arquitetura *x86_64*. O escopo da aplicação também pode ser extrapolado para aplicações maiores ou mais complexas que demandem maior processamento computacional, e que presumivelmente podem impactar no consumo de memória e no tempo de inicialização. Tamanhos maiores também podem impactar no custo final, podendo comparar com aplicações maiores que 10 MB em fomato ZIP, que são armazenas no serviço AWS S3.

Agradecimentos

O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Código de Financiamento 001.

Referências

[1] J. Nupponen and D. Taibi, "Serverless: What it is, what to do and what not to do," in *2020 IEEE International Conference on Software Architecture Companion (ICSA-C)*, 2020, pp. 49–50.

[2] AWS, "AWS Lambda now supports container images as a packaging format," 2020, accessed: 2024-08-17. [Online]. Available: https://aws.amazon.com/about-aws/whats-new/2020/12/aws-lambda-now-supports-container-images-as-a-packaging-format

[3] T. G. Peter Mell, "The NIST Definition of Cloud Comptuing," 2011, accessed: 2024-08-10. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/legacy/sp/nistspecialpublication800-145.pdf

[4] P. Vahidinia, B. Farahani, and F. S. Aliee, "Cold start in serverless computing: Current trends and mitigation strategies," in *2020 International Conference on Omni-layer Intelligent Systems (COINS)*, 2020, pp. 1–7.

[5] J. Manner, M. Endreß, T. Heckel, and G. Wirtz, "Cold Start Influencing Factors in Function as a Service," in *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*, Dec. 2018, pp. 181–188.

[6] T. Siddiqui, S. A. Siddiqui, and N. A. Khan, "Comprehensive analysis of container technology," in *2019 4th International Conference on Information Systems and Computer Networks (ISCON)*, 2019, pp. 218–223.

[7] M. J. Scheepers, "Virtualization and containerization of application infrastructure: A comparison," in *21st twente student conference on IT*, vol. 21, 2014, pp. 1–7.

[8] AWS, "What is aws lambda?" 2024, accessed: 2024-08-17. [Online]. Available: https://docs.aws.amazon.com/lambda/latest/dg/welcome.html

[9] ——, "Lambda instruction set architectures (arm/x86)," 2024, accessed: 2024-08-17. [Online]. Available: https://docs.aws.amazon.com/lambda/latest/dg/foundation-arch.html?icmpid=docs_lambda_help

[10] ——, "Using Podman with Amazon ECR - Amazon ECR," 2024, accessed: 2024-09-12. [Online]. Available: https://docs.aws.amazon.com/en_us/AmazonECR/latest/userguide/Podman.html

[11] ——, "Pushing a Docker image to an Amazon ECR private repository," 2024, accessed: 2024-09-12. [Online]. Available: https://docs.aws.amazon.com/AmazonECR/latest/userguide/docker-push-ecr-image.html

[12] ——, "What is Amazon Elastic Container Registry?" 2024, accessed: 2024-08-17. [Online]. Available: https://docs.aws.amazon.com/AmazonECR/latest/userguide/what-is-ecr.html

[13] J. Dantas, H. Khazaei, and M. Litoiu, "Application Deployment Strategies for Reducing the Cold Start Delay of AWS Lambda," in *2022 IEEE 15th International Conference on Cloud Computing (CLOUD)*, 2022, pp. 1–10.

[14] M. Elsakhawy and M. Bauer, "Performance analysis of serverless execution environments," in *2021 International Conference on Electrical, Communication, and Computer Engineering (ICECCE)*, 2021, pp. 1–6.

[15] M. Villamizar, O. Garcés, L. Ochoa, H. Castro, L. Salamanca, M. Verano Merino, R. Casallas, S. Gil, C. Valencia, A. Zambrano, and M. Lang, "Cost comparison of running web applications in the cloud using monolithic, microservice, and AWS Lambda architectures," *Service Oriented Computing and Applications*, vol. 11, 06 2017.

[16] N. Rodríguez, M. Murazzo, A. Martín, and M. Rodríguez, "Evaluation of programming languages for memory usage, scalability, and cold start, on aws lambda serverless platform as a case study," in *Computer Science–CACIC 2023: 29th Argentine Congress of Computer Science, Lujan, Argentina, October 9-12, 2023, Revised Selected Papers*, vol. 2123. Springer Nature, 2024, p. 33.

[17] AWS, "Amazon Elastic Container Registry pricing," 2024, accessed: 2024-08-20. [Online]. Available: https://aws.amazon.com/ecr/pricing