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 is growing. Currently, most cloud providers offer serverless services, in particular at Amazon, they have AWS Lambda to create serverless functions. There are at least two ways to implement a serverless function. One way is to compress the source code and required files in a compacted file in ZIP format, and the other one is through a container image, which has the running application and its dependencies. Depending on the approach chosen, the performance, the cost and the initialization time may vary. Considering these metrics, this paper wants to compare the two approaches mentioned regarding the implementations of serverless functions at AWS and aims to discover whether one of the approaches appears to be the most adequate. Experiments conducted at AWS Lambda show that functions created with compacted ZIP files present advantages, mainly in the initialization time of the function when it is in cold start mode.**

***Keywords*—Serverless Function, AWS Lambda, Container, 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* pelos seus clientes. Existem ao menos duas forma de implantação de funções *serverless*. Sendo assim, uma forma encapsula o código fonte e demais arquivos necessários em um arquivo compactado no formato ZIP, e outra onde a própria função executável e demais dependências estão em uma imagem de contâiner. 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 de funções *serverless* 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 arquivo compactado ZIP apresentam vantagens, principalmente no tempo de inicialização da função quando está em modo de partida fria.**

***Palavras-chave*—Função Serverless, AWS Lambda, 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 [**?**]. 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 atual *serverless* com aplicações monolíticas, este possui como principais diferenças: o tamanho e o tempo de execução das aplicações *serverless* são menores, não ~~precisam~~há necessidade de configurações de um servidor para executarem e são automaticamente escaláveis conforme a utilização dos recursos contratados e alocados. Já aplicações monolíticas compõem todo o código de um sistema ~~ou grande empresa~~, e por isso tendem a terem códigos fonte maiores, incluindo longos trechos de código, configurações de servidores e banco de dados. A escalabilidade, além de não ser padrão, pode ser um desafio devido ao grande tamanho da aplicação e necessidade de recursos ~~alocados~~disponíveis para alocação, e muitas vezes ociosos, para atendimento da demanda dos usuários. Visto que o sistema é executado todo em uma aplicação, para escalar ~~um~~esse tipo de sistema é preciso que toda a aplicação tenha recursos adicionados. Independente do quão grande ou pequeno o gargalo é no sistema, a aplicação como um todo precisa ser escalada, o que resulta em um desperdício computacional.

Apesar da maior facilidade de desenvolvimento, no sentido da abstração da infraestrutura necessária para a execução e

atendimento da demanda elástica da aplicação, a implantação da aplicação no ambiente *serverless* demanda cuidados, pois as configurações do ambiente são feitas pelo provedor e os desenvolvedores não tem acesso para alterá-las. Tais cuidados estão relacionados a a configuração dos recursos necessários para executá-la, bem como a forma que a aplicação é instanciada e desativada de acordo com o seu uso (pelos usuários) e ociosidade (períodos sem demanda por parte dos usuários). A medida que a aplicação fica ociosa por alguns minutes, o provedor desaloca os recursos computacionais da função, e a mesma fica inoperante. Quando uma nova ~~chamada~~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 ~~chamada a~~invocação (por parte do cliente) à função que se encontra no estado inoperante, o tempo de resposta também é maior na primeira requisição.

No provedor de nuvem AWS, particularmente, há duas maneiras de se implantar uma nova função manualmente: a maneira mais tradicional via arquivo compactado em formato zip, ou através de uma imagem de contêiner, opção introduzida pela empresa em 2020 [**?**].

Na implantação via arquivo compactado, o desenvolvedor precisa compactar em formato ZIP a pasta do projeto, incluindo o código fonte e os arquivos executáveis da aplicação, e deve realizar o *upload* direto para o serviço *serverless*, que é chamado AWS Lambda. Com o uso de imagem de contêiner, é necessário adicionar um arquivo Dockerfile com as dependências e configurações para realizar o *build* da imagem e para executar o projeto. A partir da imagem gerada no ambiente local do desenvolvedor, esta deve ser publicada no repositório de imagens do provedor. Na sequência, para se criar então a função *serverless*, deve ser selecionada a respectiva imagem disponível no repositório.

Independente do modelo de implantação escolhido, o desenvolvedor precisa implementar no código da aplicação uma interface fornecida pelo serviço AWS Lambda para receber as requisições e parâmetros de entrada do cliente a utilizará. Dessa forma, a função *serverless* sabe qual método do código invocar ao receber requisições.

Como contribuição científica, este trabalho busca responder as seguintes perguntas:

1) Dentre os modelos de implantação de função *serverless* ~~na Amazo~~no serviço AWS Lambda, qual é o modelo mais vantajoso, ~~dado as~~de acordo com métricas de desempenho e tempo de inicialização?
2) Como o custo impacta cada uma dessas implantações?

Para ~~corroborar com as~~subsidiar a resposta, o trabalho aborda as duas possibilidades de criar/implantar funções *serverless* no serviço AWS Lambda através de experimentos, e propõe uma comparação de desempenho, custo e tempo de inicialização entre as abordagens. Por fim, busca-se ~~concluir~~avaliar se há um modelo de implantação mais vantajoso do que outro.

Para ~~chegar nos~~obter os resultados de comparação,~~testes foram~~experimentos são executados nas funções *serverless* via chamada de API. As métricas ~~analisadas foram~~de desempenho para realizar tal comparação são as de consumo máximo de memória, o custo para implantar as funções e o tempo de inicialização da função/aplicação. Durante a execução ~~de testes~~dos experimentos, é utilizado o acesso *Free Tier* ao serviço AWS Lambda, e não houve cobrança nos serviços utilizados. Portanto, a 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~~e~~ apresenta~~r~~ características utilizadas na avaliação. A Seção III ~~lista~~elenca trabalhos relacionados pertinentes ao tema, bem como suas características e como se diferem desse artigo. Na Seção IV, é apresentada a proposta de avaliação de implantação de aplicações *serverless*, bem como os experimentos realizados e os valores das métricas coletadas. Por fim, a Seção V conclui o artigo.

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 o chamado *serverless*, que executa uma aplicação sem que o cliente necessite configurar o servidor e demais infraestruturas. Nesse sentido, esta seção trata de temas pertinentes a avaliação proposta passando pelo modelo de computação em nuvem; definindo funções *serverless*; os modelos de implantação *serverless*, no serviço AWS Lambda em particular, incluindo a conceituação de *containers*; e os serviços da nuvem computacional AWS da empresa Amazon, incluindo o AWS Lambda e correlacionados à disponibilização do serviço *serverless*.

A. Computação em Nuvem

A computação em nuvem é um modelo de disponibilização de serviços computacionais ~~com vastos recursos de computação~~ compartilhados, como rede, servidores, armazenamento, aplicações e serviços que podem ser provisionados e liberados sob demanda rapidamente e com baixa configuração e complexidade por parte dos seus usuários [**?**]. 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 querem 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 leva 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~~Funções~~ *serverless* é um modelo de serviço de computação em nuvem, também conhecido por funções *serverless* ou modelo *Function as a Service* (FaaS) que providencia a abstração dos servidores e demais infraestrutura, para a execução de aplicações em nuvem, sob demanda, sem necessidade de alocação (uso de recursos computacionais) prévia. Assim, os clientes do modelo, ou seja, os desenvolvedores de aplicações, precisam apenas implementar o código da aplicação, isto é, a função. Ao ter em suas instalações o código implementado da aplicação, o provedor toma conta de alocar/instanciar o servidor físico ou virtual, configurar o ambiente e disponibilizar uma *Application Program Interface* (API) para acesso da aplicação por parte do ~~cliente~~usuário final da mesma.

Uma das principais vantagens desse modelo é que o provedor de serviço *serverless* é responsável pela escalabilidade da aplicação. A escalabilidade nesse caso contempla a instanciação da função para sua utilização pelos ~~clientes~~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 o ~~dimensionamento~~a alocação dos recursostendo 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 sistemas. Quando a aplicação é novamente invocada por um usuário final, ela pode necessitar que os recursos sejam reativados e uma nova instância seja criada. Esse processo é chamado de partida lenta ou fria [**?**]. Após ter os recursos reativados, essa instância permanecerá ativa enquanto é utilizada, e por um período de tempo (minutos) após a sua última utilização, ou seja, mesmo não sendo utilizada por usuários finais. Nesse caso, se nesse período de tempo houver um novo acesso a função por usuários finais, a função *serverless* continua ativa, com os recursos computacionais ativos e alocados a ela. Assim, ela pode ser utilizada de imediato, processo que é chamado de partida quente, pois o provedor de serviço *serverless* possui a instância e os recursos operantes [**?**] para o atendimento de nova requisição, ou seja, continuam ativos (”quentes”), sem que seja necessária ~~instancie~~ uma nova instância da função. A função, após ficar alguns minutos ociosa, sem nenhuma execução, tem seus recursos desativados pelo provedor. Esse ciclo continua alternando entre partida fria e partida quente conforme a sua utilização.

O tempo de partida lenta é normalmente considerado uma desvantagem, e otimizações para diminuí-lo são estudados em busca de evitar grandes atrasos ao cliente final. Porém, há dificuldades em se alcançar soluções ideais, dado que as configurações acerca do gerenciamento dos recursos e das instanciações são mantidas pelos provedores de nuvem [**?**], [**?**], [**?**], [**?**], e não são disponíveis para experimentações e otimizações, por parte dos clientes desses provedores de serviço.

C. Contêineres

Contêineres fornecem às aplicações um ambiente computacional para a execução de aplicações, configurado com todas as suas dependências para ser executado. Eles isolam a aplicação de programas e processos externos que executam no sistema operacional hospedeiro do contêiner. Portanto, pode ser definido como um mecanismo de empacotamento de aplicações [**?**].

A utilização de contêineres se tornou amplamente popular pela facilidade de migrar aplicações de um ambiente para outro sem a ocorrência de problemas ~~ao~~ocasionados pela execu~~tar~~ção em máquinas diferentes. Sem isso, toda vez que as aplicações migravam para um novo ambiente, havia uma grande chance de ocorrerem erros, pois dois ambientes nunca são totalmente idênticos em questão de *software* e *hardware*, e portanto, podem não ter as configurações e dependências necessárias para a execução da aplicação [**?**]. 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, 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 tudo o que é necessário para executá-la. Com a imagem criada, 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 [**?**]. 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. Dentre os serviços ofertados, este trabalho utiliza três deles: AWS Lambda, API Gateway e AWS *Elastic Container Registry* (ECR).

O serviço de *serverless* da Amazon é chamado de AWS Lambda, e nele, consegue-se criar funções para executar aplicações sem precisar provisionar e gerenciar servidores. O serviço Lambda gerencia toda a configuração computacional, que oferece equilíbrio de memória, CPU, rede e outros recursos necessários para executar o código da aplicação (função). As funções são instanciadas sob demanda conforme requisições feitas por usuários finais, o que faz o serviço alocar parte da máquina virtual para a função. Conforme a função passa a ficar ociosa por alguns minutos, os recursos computacionais alocados são desativados. Isso possibilita que os clientes paguem apenas pelo tempo em que a aplicação está sendo utilizada, com os recursos alocados para a função *serverless* [**?**].

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 do serviço *serverless* disponibilizada pela biblioteca específica. Ao criar a função, na etapa de configuração, é necessário informar o caminho do método que implementa a interface, pois esse método é usado como ponto de entrada para a função invocar a aplicação e passar os parâmetros de entrada.

Particularmente, na AWS, há duas maneiras de implantar uma função *serverless*, que são: via arquivo compactado do código fonte ou com uma imagem de contêiner. Para a abordagem de arquivo compactado, é necessário compilar os arquivos fontes, 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 publicada na AWS ECR, que é o repositório de imagens de contêineres, e selecionar a respectiva imagem quando da criação da 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. Apesar da arquitetura x86_64 ser a padrão, a arquitetura arm64 se destaca por possuir menor custo de execução e atingir ~~melhores~~bons resultados de desempenho [**?**].

AWS ECR é um serviço de repositório para armazenar imagens de contêineres. Em particular o formato de contêiner utilizado é o formato *docker*. O desenvolvedor deve criar a imagem em sua máquina local a partir de um arquivo Dockerfile e publicá-la no repositório [**?**]. Assim, ao ter a imagem disponível no repositório, esta pode ser usada em outros serviços do provedor, como por exemplo para criar funções *serverless*.

Amazon API Gateway é um serviço para criar, publicar e gerenciar ~~APIs~~aplicações web??, ~~podendo ser~~aceitando o formato REST, HTTP ou WebSocket. É usado como um ponto de entrada para os serviços AWS, incluindo o AWS Lambda, e as APIs podem ser criadas para acessarem os serviços criados ou dados armazenados. Esse serviço lida com as tarefas de aceitar e processar as requisições, com capacidade de processar centenas de milhares de requisições concorrentes [**?**]. acho que poderias dizer que é por meio desse serviço, ou seja, geralmente por meio da URL HTTP disponibilizada por esse serviço que a função é invocada pelo usuário final, e que dá início ao processo de instanciação da mesma via partida fria ou quente

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 [**?**] aborda estratégias para diminuir o *cold start* 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 [**?**], 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 do *cold start*.

Na publicação [**?**], 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%.

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 características de alguns 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.

IV. Metodologia e Experimentos

Este trabalho propõe avaliar dois métodos de implantação de uma aplicação no serviço *serverless* da Amazon, AWS Lambda. A escolha da linguagem de programação para a aplicação (função) foi Java, devido a sua popularidade no meio Web e facilidade de implementar as interfaces requeridas para o serviço na AWS.

Os experimentos buscam auxiliar na descoberta se uma das abordagens é mais vantajosa que a outra. Para chegar em tal conclusão, métricas pertinentes ao serviço foram analisadas. As métricas levadas em consideração foram de desempenho com base no consumo de memória, de custo e de tempo de inicialização em partida lenta.

Para realizar a comparação das métricas, foi desenvolvido uma aplicação, implantada como função *serverless* no serviço AWS Lambda via código da função compactado como arquivo ZIP e imagem de contêiner, e fazendo requisições HTTP via Amazon Gateway.

~~Primeiro~~Inicialmente, é descrito o ambiente e as configurações utilizadas, e em seguida, são apresentados as métricas selecionadas e os dados da comparação entre os modelos de implantação.

A. Ambiente

Para realizar tal avaliação, foi realizada a implementação de uma aplicação simples utilizando a linguagem de programção Java, versão 11. A aplicação pode ser considerada simples, pois possui apenas umas interface de entrada de dados onde deve ser informada uma região geográfica do globo terrestre e é retornado a data e o horário da região informada.

Para implantar a aplicação no serviço da AWS, foi criada uma função utilizando arquivo ZIP compactado e outra função via imagem de contêiner. Para criar a função com o uso da imagem, uma imagem da aplicação foi criada a partir de um arquivo de configuração Dockerfile para o formato de contêiner docker, configurado para realizar o *build* (instalação do sistema operacional, dependências e a própria aplicação em uma imagem executável, no contêiner) e publicada no repositória do AWS ECR. ~~Refente a~~A arquitetura escolhida para as funções foi ~~escolhido~~ a arquitetura arm64 devido ao menor ~~preço~~custo, bem como a pouca necessidade de processamento, armazenamento ou memória da função (o que se reflete inclusive no custo).

Para fazer requisições às funções criadas, optou-se por usar APIs públicas REST na Amazon API Gateway, as quais foram configuradas para invocar as respectivas funções, dependendo do caminho do *endpoint*. É importante utilizar caminhos diferentes para as APIs, pois essas desconhecem o modelo de implantação de cada função *serverless* invocada.

O ambiente configurado é representado no diagrama da Figura 1. Nele, inicialmente, o usuário faz uma chamada de requisição REST. A requisição é recebida pela API Gateway, que possui um gatilho e redireciona para a respectiva função *serverless* criada. A função pode executar a aplicação implantada via arquivo compactado ZIP ou com uma imagem de contêiner armazenada no Amazon ECR. Ao executar o trecho de código da aplicação, a função *serverless* providencia uma mensagem de resposta para o serviço de API Gateway, o qual redireciona a mensagem para o usuário. quantas execuções da mesma função foram realizadas? acho importante destacar todos os aspectos dos experimentos aqui.

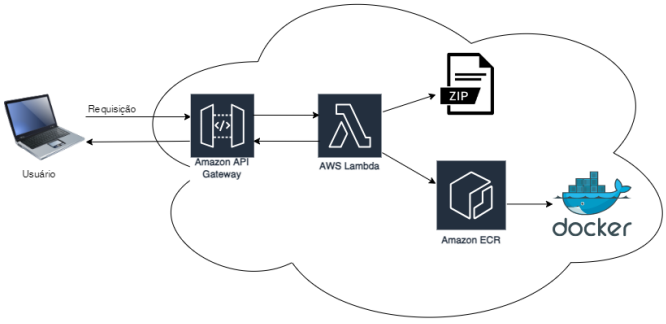

Fig. 1. Diagrama do ambiente de testes

B. Custo

Para a execução dos experimentos na AWS, 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,

TABLE I
COMPARAÇÃO DE TRABALHOS RELACIONADOS

Artigo	Arquivo compactado	Imagem de contêiner	Desempenho	Custo	Tempo de inicialização
Dantas [**?**]	Sim	Sim	Não	Sim	Sim
Elsakhawy [**?**]	Não	Sim	Sim	Sim	Não
Villamizar [**?**]	Não	Não	Não	Sim	Não
Trabalho atual	Sim	Sim	Sim	Sim	Sim

são usados os valores base de precificação do provedor de nuvem AWS.

O custo para implantar e manter a função *serverless* ativa é o mesmo, independente da abordagem escolhida. Outros fatores, como configuração de *hardware* e localização do servidor podem impactar no valor final, mas estão fora do escopo deste trabalho. Ao fazer o *upload* do projeto compactado com tamanho até 10 Mb, não há nenhuma cobrança para armazenar os arquivos. Porém, para projetos maiores, é necessário armazená-los no serviço S3, que é o serviço de armazenamento de dados de grande porte do provedor AWS. Para disponibilizar uma imagem de contêiner no serviço AWS ECR, há um custo, que é proporcional ao tamanho da imagem. A precificação na AWS ECR também varia dependendo da região [**?**] de localização do *data center*. em que região armazenaste a tua função. Acho que é importante comentar isso para reprodutibilidade.

C. Desempenho

~~Algumas~~Várias métricas podem estar relacionadas ao desempenho de uma aplicação. Neste trabalho, em relação ao desempenho foi analisado o consumo de memória RAM máximo no ambiente experimentado ao executar funções *serverless* que estavam em modo de partida lenta. Essa métrica é coletada no console de saída do AWS Lambda ao fazer uma requisição.

Conforme a Figura 2, nota-se que o uso de memória máximo com a abordagem de imagem de contêiner se mantém constante, enquanto via arquivo ZIP, há uma variação de uma execução para outra. A Figura 3 apresenta a média de ambas abordagens, corroborando os resultados da Figura 2, demonstrando que a implantação da função feita a partir da imagem de contêiner obteve melhores resultados em relação ao consumo de memória, ou seja, consome menos memória para executar a função.

D. Tempo de inicialização em partida fria

As Figuras 4 e 5 ~~mostram~~ilustram que o método de implantação via arquivo ZIP possui o menor tempo de inicialização em partida fria???. Particularmente, observa-se na Figura 4 o tempo de inicialização da função implantada via arquivo compactado e via contêiner, para várias inicializações. Cada inicialização ocorreu respeitando tempo suficiente para que os recursos fossem desalocados, obrigando uma nova instanciação da função. A Figura 5 apresenta a média dos tempos de inicialização apresentados na Figura 4.

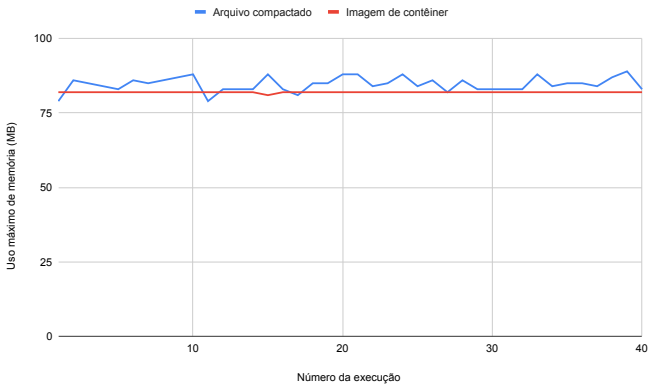

Fig. 2. Gráfico de uso de memória máximo em funções *serverless*

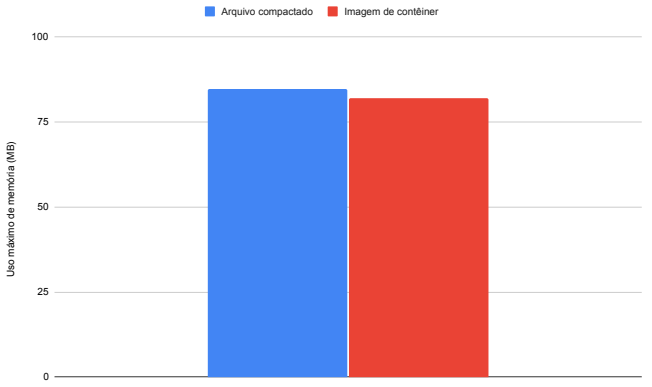

Fig. 3. Gráfico da média de uso de memória máximo em funções *serverless*

O tempo de inicialização também impacta no tempo de resposta quando a aplicação está em modo de partida lenta, portanto, funções executando com arquivo compactado obtém melhores resultados quanto a inicialização em partida fria. não testaste em partida quente? E a comparação entre as métricas, já que em contêiner tem um consumo de memória constante e melhor naquele sentido, segundo o que escreveste. Como fica

a conclusão final?

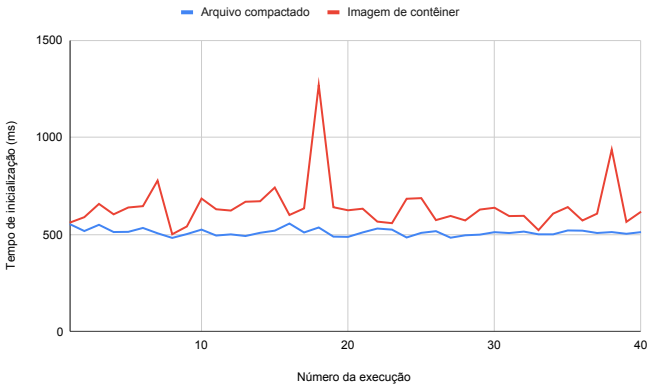

Fig. 4. Gráfico do tempo de inicialização em funções *serverless*

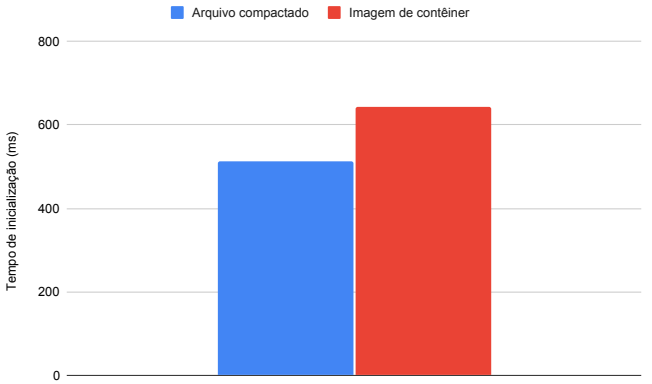

Fig. 5. Gráfico da média do tempo de inicialização em funções *serverless*

V. ~~Conclusão~~Considerações Finais

Esse trabalho avaliou os modelos de implantação de funções *serverless* disponíveis no serviço AWS Lambda. O serviço oferece duas possibilidades de implantação de funções: via arquivo compactado no formato ZIP e via imagem de contêiner. Com os arquivos compactados ZIP, o modelo de implantação é mais ~~fácil~~simplificado, pois ~~é necessário apenas fazer se~~basta efetuar o *upload* do projeto (código da função compactado) para o serviço AWS Lambda, enquanto que na segunda abordagem é necessário configurar um arquivo Dockerfile para ~~fazer~~executar o *build* da aplicação, gerar uma imagem de contêiner e publicá-la na AWS ECR.

Como resposta as perguntas estabelecidas ~~no Capítulo~~na Seção I, conclui-se que dependendo do modelo escolhido, o custo, o desempenho e o tempo de inicialização da partida lenta podem ser diferentes. Ao criar as funções *serverless* na AWS Lambda nos dois modelos de implantação, ambos não tiverem incidências de custos durante os testes realizados. Porém, quando a abordagem escolhida é com o uso de uma imagem de contêiner, é necessário utilizar outro serviço para armazenar a imagem, nesse caso o AWS ECR, e este pode vir a gerar custos conforme o tamanho da imagem. Ao analisar o uso máximo de memória RAM quando a aplicação está em modo de partida lenta, ambos apresentaram consumo similar, com a vantagem que via imagem de contêiner o uso da memória se manteve constante. A maior diferença se deu no tempo de inicialização da aplicação em partida lenta, no qual a implantação via arquivo ZIP mostrou-se mais eficiente para alocar os recursos e tornar a função ativa novamente.

Portanto, com base no cenário experimentado, nos dados e resultados obtidos e em sua análise, ~~é~~ pode-se~~r~~ inferir que a implantação via arquivo ZIP apresenta vantagens. As principais vantagens comparadas ao modelo de implantação com imagem de contêiner são: o custo para implantação, uma vez que não é necessário armazenar a pasta compacta em outro serviço e o tempo de inicialização quando em partida lenta que é menor.

Como trabalho futuros, pode-se estender a comparação para as outras linguagens de programação suportadas pelo provedor de nuvem computacionail AWS. Outro aspecto a ser comparado é a arquitetura na qual a função *serverless* é executada, x86_64 ou arm64. Neste trabalho, foi utilizado apenas a arquitetura arm64, havendo espaço para tratar da arquitetura x86_64. O escopo da aplicação também pode ser extrapolado para aplicações maiores ou mais complexas que demandem maior processamento computacional, e que ~~deve~~presumivelmente podem impactar no consumo máximo de memória e tempo de inicialização. ~~Aplicações maiores que 10 MB podem ter incidência de custos para disponibilizar o arquivo compactado, o que tornaria a comparação de custo mais justa.~~Ainda, o tamanho da aplicação é outra característica que pode ser explorada em trabalhos futuros, uma vez que a incidência de custo financeiro é diretamente ligada a ela.

Agradecimentos

Coloque aqui seus agradecimentos.