Análise de Mecanismos de Serverless Computing em Ambientes de Nuvens Computacionais

Matheus N. da Silva, Marcus Carvalho

Departamento de Ciências Exatas – Universidade Federal da Paraíba (UFPB), Campus IV – Av. Santa Elizabeth, S/N, Centro – CEP 58297-000 – Rio Tinto – PB – Brazil

{matheus.nicolas, marcuswac}@dcx.ufpb.br

Abstract. *The serverless computing model is trending nowadays, because of its easy adoption and possibility of reducing costs. However, as users are not in charge of managing servers, they may face performance issues related to coldstarts, when the provider disables an idle application and new requests have to wait until it is deployed in a server and enabled again. The objective of this work is to analyze coldstarts in serverless computing, aiming to understand the time an application stays idle until the provider disables it, the overhead implied by coldstarts on response times and the impact of application's memory size on these metrics.*

Resumo. *O modelo de serverless computing tem se tornado tendência nos últimos anos, devido à facilidade de adoção e possibilidade de redução de custos. Porém, como usuários não têm controle sobre os servidores, eles podem ter problemas de desempenho relacionados a coldstarts, quando o provedor torna inativa uma aplicação ociosa e novas requisições precisam esperar até que ela seja implantada em um servidor e se torne ativa novamente. O objetivo deste trabalho é analisar o coldstart em serverless computing, buscando entender o tempo de ociosidade de uma aplicação até o provedor torná-la inativa, o overhead imposto pelo coldstart no tempo de resposta e se a quantidade de memória alocada para a aplicação afeta essas métricas.*

1. Introdução

A computação em nuvem permite a seus usuários reduzir despesas na aquisição de hardware e em sua manutenção, investindo por outro lado em serviços terceirizados pelos provedores. Inicialmente, ainda cabia ao usuário da nuvem realizar tarefas operacionais como: criar instâncias (servidores) e determinar sua capacidade (e.g. CPU, memória e disco); implantar a aplicação nos servidores; e decidir quando e quantas instâncias estarão em execução com base na demanda da aplicação [Savage 2018].

O *serverless computing* surgiu como alternativa a este modelo tradicional, no qual os usuários não precisam gerenciar os servidores que rodam suas aplicações. Neste novo modelo, o usuário pode focar de fato na sua aplicação, deixando para o provedor a implantação da aplicação em instâncias, escalabilidade e tolerância a falhas. O usuário paga apenas pelo uso de suas aplicações com base na demanda e não mais pelo tempo em que os servidores estiveram rodando [Fowler 2012].

Um dos principais modelos associados ao *serverless* é o *FaaS* (ou função como serviço, do inglês *Function as a Service*). Neste modelo, o usuário carrega o código da função no provedor, que é responsável por toda a gerência de servidores, implantação da função em um ambiente de execução e provisionamento automático de recursos para atender à demanda. A execução da função é acionada por eventos definidos pelo usuário no provedor – por exemplo, uma requisição HTTP que chega em um certo caminho pode acionar uma função que faz o seu processamento [Fowler 2012]. Provedores de *FaaS* da atualidade incluem Amazon AWS Lambda[1] e Google Cloud Functions[2], que dão

[1] https://aws.amazon.com/pt/lambda/

suporte a funções implementadas em linguagens de programação e ambientes de execução como: Node.js, Python, Go, Java, etc.

Um dos diferenciais do modelo *serverless* é que, para reduzir custos, o provedor pode desativar aplicações após um tempo de ociosidade (i.e. em períodos sem demanda). Porém, para processar requisições que chegam para uma aplicação inativa, o provedor precisa antes implantar esta aplicação em um servidor e iniciá-la em um ambiente de execução. Este tempo de inicialização de aplicações inativas em um ambiente *serverless* é chamado de *coldstart*, que pode impactar o desempenho das aplicações [Baldini et al 2016].

O objetivo geral deste trabalho é investigar o *coldstart* em um ambiente de *serverless computing,* analisando o seu impacto no tempo de resposta das aplicações e os fatores que podem afetar a sua frequência e duração. Para isto, foram realizados experimentos de medição em um provedor de *FaaS,* usando funções e cargas sintéticas. Como objetivos específicos, buscou-se responder às seguintes perguntas de pesquisa:

- Com quanto tempo de ociosidade uma função é desativada pelo provedor, tendo como consequência o *coldstart* em requisições que vierem a seguir?
- Qual o tempo adicional (*overhead*) que o *coldstart* impõe no tempo de resposta das requisições, comparado a requisições que não passam por *coldstart*?
- Qual o impacto da quantidade de memória alocada para a função no *overhead* do *coldstart* e no intervalo de ociosidade para que ocorra um *coldstart*.

O restante do artigo está estruturado da seguinte forma: a seção 2 apresenta os trabalhos relacionados; a seção 3 apresenta a metodologia da avaliação; a seção 4 apresenta os resultados; e a seção 5 discute as conclusões e trabalhos futuros.

2. Trabalhos relacionados

Baldini et al. (2016) faz um levantamento das tendências e problemas em aberto na área de *serverless computing*. Um dos desafios apontados é exatamente o fenômeno do *coldstart*, pois ao mesmo tempo que pode-se reduzir custos escalando uma função para zero servidores, desativando-a, pode-se também piorar seu desempenho por causa do *coldstart*. Por isso, seu trabalho aponta como crítico o estudo de técnicas para minimizar o *coldstart,* ao mesmo tempo que aplicações ociosas ainda possam ser desativadas. Apesar da relevância do tema, encontramos poucas publicações acadêmicas que realizam análises voltadas ao *coldstart* durante nossa revisão da literatura. Desta forma, o trabalho corrente visa preencher um pouco esta lacuna, contribuindo com uma avaliação de desempenho de *coldstarts* em um ambiente de *serverless computing*.

Os trabalhos encontrados mais relacionados foram os de Cui (2017a) e Cui (2017b), que apesar de não terem sido publicações acadêmicas, apresentam uma metodologia clara e resultados bem descritos. Em seu primeiro estudo, Cui (2017a) propõe uma metodologia de detecção de *coldstarts* e analisa um provedor de *FaaS* (AWS Lambda) quanto tempo uma função fica ociosa até que o provedor torne-a inativa, resultando em seguida em um *coldstart*. Este estudo concluiu que o AWS Lambda tornou uma função inativa após aproximadamente 40 a 60 minutos de ociosidade, mas que eventualmente podiam ocorrer *coldstarts* antes desse período de ociosidade, para liberar recursos. Também observou-se que funções com mais memória alocada tendem a ser desativadas com um tempo menor de ociosidade. Com base nesta análise inicial, nosso trabalho buscou reproduzir os experimentos de Cui (2017a) para medir quando os *coldstarts* ocorrem e, além disso, expandir a análise para medir o *overhead* imposto pelo *coldstart* no tempo de resposta das funções, dependendo da quantidade de memória alocada para ela.

---

[2] https://cloud.google.com/functions/

Em outro estudo, Cui (2017b) mediu como a linguagem de programação, o tamanho do código e a quantodade de memória alocada afetam o *coldstart*. Em seus resultados, observou-se que: linguagens de programação diferentes podem apresentar diferenças significativas no *overhead* de *coldstarts*, sendo os *coldstarts* para *Python* e *NodeJS* bem menores que para *Java* e *C#*; quanto mais memória alocada para a função, menor a duração do *coldstart*; quanto maior o código, menor o *coldstart*. O nosso trabalho buscou fazer uma análise mais aprofundada da duração e frequência de *coldstarts* para diferentes tamanhos de memória, focando no framework *NodeJS*. Ao contrário do trabalho de Cui (2017b), observamos que a relação do tempo do *coldstart* com o tamanho da memória alocada não é linear, o que será discutido nos resultados.

3. Metodologia

Neste trabalho, foram realizados experimentos de medição para analisar a frequência e duração de *coldstarts* em um ambiente de *serverless computing*. O experimento foi baseado na metodologia usada por Cui (2017a) para criação de funções, medição de desempenho e geração de carga.

O ambiente de *FaaS* do AWS Lambda foi usado para a medição de desempenho e análise de *coldstarts*. Uma função lambda chamada *system-under-test* foi criada no ambiente de execução NodeJS, sendo configurada para executar ao chegarem requisições HTTP em um caminho específico. Esta função informa como retorno se a requisição sendo executada está se deparando ou não com um *coldstart* da seguinte forma: em seu primeiro acesso, a função sendo executada armazena uma *flag* em memória indicando que já foi acessada; se um acesso posterior à função não observar tal *flag* ativada, é porque a função passou por um *coldstart* e teve que ser iniciada novamente, perdendo seu estado prévio [Cui 2017a]. Com o intuito de avaliar o impacto da alocação de memória no *coldstart*, foram executados diferentes cenários de alocação de memória para a função *system-under-test*, com os seguintes valores: 128MB, 256MB, 512MB, 1024MB e 2048MB.

A geração de carga e medição de desempenho foi realizada utilizando o serviço AWS Step Functions[3], que possibilita a criação de um fluxo de execução de funções e a gravação de medições em serviços de monitoramento. A Figura 1, demonstra as funções criadas e o fluxo de execução para a geração de carga periódica dos experimentos.

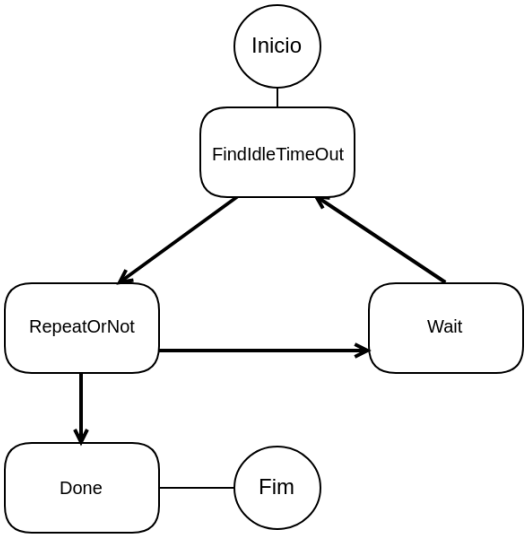

Figura 1. Fluxo de execução de funções de geração de carga dos experimentos.

[3] https://aws.amazon.com/pt/step-functions/

Durante a execução de um experimento, o fluxo de execução mantém uma máquina de estados com os três atributos abaixo:

- *target:* função alvo a ser chamada pelo gerador de carga e para a qual o desempenho será medido;
- *coldstarts:* contador da quantidade de *coldstarts* consecutivos encontrados*;*
- *wait*: intervalo de tempo entre requisições submetidas pelo gerador de carga.

A função *FindIdleTimeout* é responsável por gerar a carga e medir o tempo de resposta, enviando requisições periodicamente à função *system-under-test* informada no parâmetro *target*. Cada chamada à função *system-under-test* retorna o status do *coldstart* para aquela chamada: se houve *colstart*, o parâmetro *coldstarts* do experimento é incrementado. Em seguida, o fluxo do experimento passa para a função *RepeatOrNot*.

A função *RepeatOrNot* verifica se o experimento chegou ao fim, ou se ainda serão necessários mais testes. Além disso, ela pode ajustar o parâmetro *wait* para determinar o tempo de espera até a próxima requisição ser gerada. Um dos objetivos do experimento é saber após quanto tempo de ociosidade o provedor torna a função inativa – ou seja, quando os *coldstarts* começam a acontecer sistematicamente. Nos nossos experimentos, o parâmetro *wait* é iniciado com o valor de 10 minutos, sendo incrementado ao longo do experimento até que seja encontrado o tempo de ociosidade que causa *coldstarts* de forma sistemática. Como *coldstarts* inesperados podem acontecer mesmo com pouco tempo de ociosidade, considera-se que o tempo de ociosidade sistemático para tornar a função inativa é encontrado quando 10 *coldstarts* consecutivos são observados. Desta forma, se a função *RepeatOrNot* verificar que a variável *coldstarts* chegou a 10, o fluxo do experimento segue para o estado *Done* para ser finalizado. Caso contrário, se não houve *coldstart* na última requisição, tenta-se um tempo de ociosidade maior incrementando o atributo *wait* em 1 minuto e zerando o contador de *coldstarts* para este novo período. No fim, o fluxo passa para a função *Wait*.

A função *Wait* representa um simples estado de espera, onde a variável *wait* é observada para saber quantos minutos o fluxo de execução do experimento ficará aguardando naquele estado, até voltar novamente à função *FindIdleTimeout* para gerar mais uma requisição e continuar o ciclo.

Esse fluxo de execução do experimento foi rodado uma vez para cada cenário de alocação de memória, um cenário de cada vez. A execução de todos os cenários durou aproximadamente um dia para executar por completo, devido as várias rodadas necessárias para encontrar o tempo de ociosidade *(wait)* que cause 10 *coldstarts* consecutivos.

As métricas avaliadas no experimento foram:

- *tempo máximo ocioso*: tempo máximo que uma função permanece ociosa, sem receber requisições, até ser desativada pelo provedor. Essa métrica foi calculada como o maior valor da variável *wait*, para o qual foram observados 10 *coldstarts* consecutivos.
- *tempo de resposta das requisições*: tempo observado pelo gerador de carga entre a submissão de uma requisição à função *system-under-test* e o recebimento de sua resposta. Associada a esta medição também está a identificação da requisição enviada ter ou não passado por um *coldstart*.

A próxima seção apresenta os resultados da avaliação, discutindo as métricas para cada cenário descrito na metodologia.

4. Resultados

A Tabela 1 apresenta os resultados do tempo máximo ocioso para diferentes cenários de memória alocada para a função. Observa-se que, nos experimentos, o AWS Lambda desativou a função de forma sistemática após um tempo de ociosidade que variou de 30 a 42 minutos. O tempo máximo ocioso observado nos experimentos foram menores do que os encontrados por Cui (2017a), 1 ano antes, que reportou valores entre 48 e 63 minutos para os mesmos cenários de memória alocada. Este resultado sugere que o tempo de ociosidade para desativar uma função não é constante e que, além do tamanho da memória, ele pode depender de fatores externos como a carga do provedor. Apesar de Cui (2017a) sugerir que existe um padrão de quanto mais memória alocada, menor o tempo máximo ocioso, este fenômeno não foi observado nos nossos experimentos. O tempo máximo ocioso de fato variou ao mudar o total de memória alocada, mas não conseguimos observar nenhum padrão linear. Uma quantidade maior de experimentos devem ser realizados, em diferentes dias e horas, para tentar capturar apenas o efeito da alocação de memória no tempo máximo de ociosidade.

Tabela 1. Tempo máximo ocioso da função para o qual acontecem *coldstarts* sistemáticos, para diferentes cenários de memória alocada.

Total de memória alocada (MB)	Tempo máximo ocioso (minutos)
128	30
256	42
512	30
1024	30
2048	40

A Figura 2 apresenta o tempo de resposta das requisições ao variar o intervalo entre requisições *(wait)*, em diferentes cenários de memória alocada para a função. Para cada requisição, também é identificado se a mesma passou por um *coldstart* (em azul) ou se não passou por *coldstart* (em vermelho). Nota-se que sempre há um *coldstart* no intervalo inicial entre requisições, de 10 minutos. Isto acontece porque a primeira requisição sempre passa por um *coldstart*, pois ela que ativa a função pela primeira vez. Também pode-se notar claramente que o tempo de execução com *coldstart* é significativamente maior do que o tempo de resposta sem *coldstart*, reforçando a importância de minimizar ao máximo *coldstart*. Também se observa que os *coldstarts* tendem a aparecer aproximadamente após 25 minutos de ociosidade, mas nem sempre a função é desativada de forma sistemática em um intervalo fixo. Uma hipótese é que em momentos de sobrecarga no provedor, ele pode decidir desativar funções que estão ociosas a mais de 25 minutos, dando prioridade a funções que estão ociosas a mais tempo.

A função de maior alocação de memória (2048MB) foi a que mais apresentou *coldstarts* antes de atingir os 10 consecutivos, havendo: 5 *coldstarts* com ociosidade de 26 minutos; 2 *coldstarts* com ociosidade de 35 minutos; e 10 *coldstarts* com ociosidades de 10 minutos, quando considerou-se o tempo de ociosidade sistemática. Uma outra hipótese para estes *coldstarts* mais frequentes em intervalos mais curtos é que o provedor pode tentar desativar primeiro funções que estão ociosas por algum tempo e que possuem mais memória alocada, para liberar mais recursos em períodos de sobrecarga no servidor.

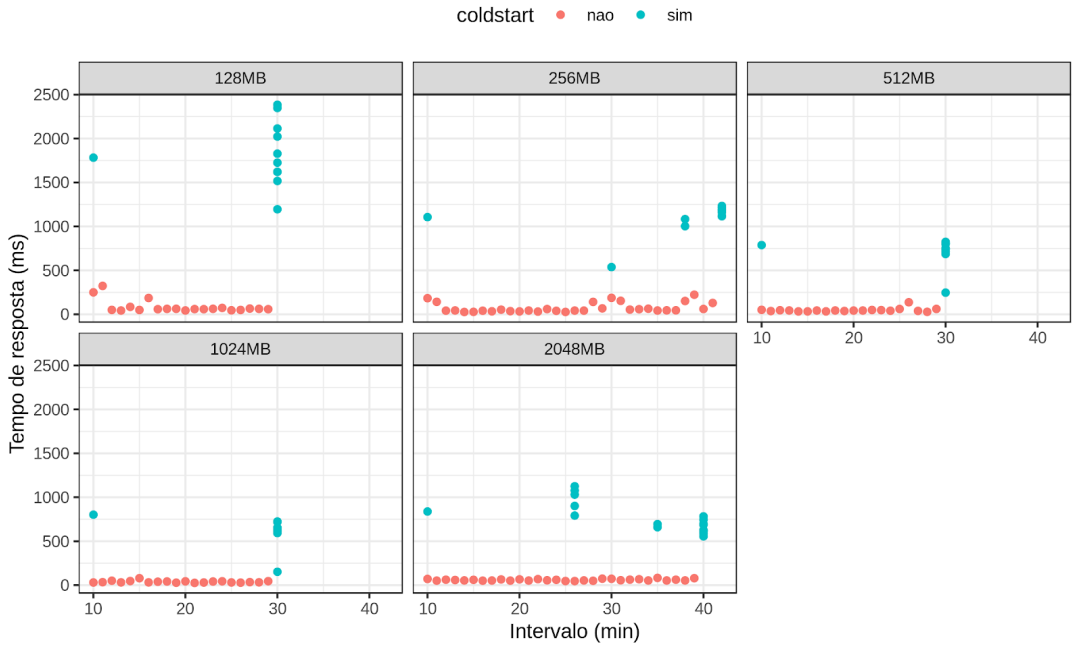

Figura 2. Tempo de resposta das requisições com e sem *coldstart*.

A Figura 3 apresenta o *boxplot* utilizado para analisar o tempo de resposta das requisições para todos os cenários. Podemos comparar as medianas e os quartis do tempo de resposta quando há ou não *coldstart*. Os pontos dispersos presentes na figura são chamados de *outliers*; são os valores discrepantes no nosso gráfico e a presença deles nos experimentos é a razão de fazermos a comparação através da mediana e não da média. Observa-se que o tempo de resposta com coldstart foi significativamente maior do que sem coldstart. A mediana do tempo de resposta sem *coldstart* foi de 51ms, enquanto com *coldstart* a media foi 802ms. Ou seja, o tempo adicional gasto com o *coldstart* foi de aproximadamente 751ms para os cenários avaliados.

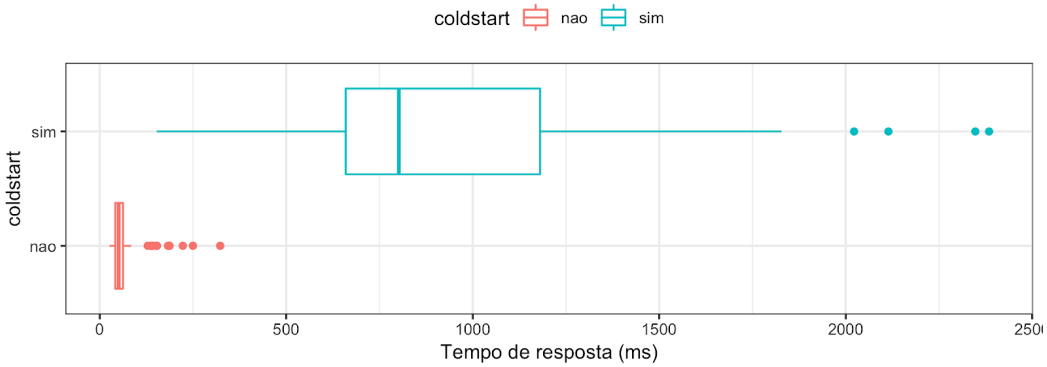

Figura 3. Boxplot do tempo de resposta total com e sem *coldstart*.

A Figura 4 apresenta o boxplot do tempo de resposta com e sem *coldstart*, agora dividindo para cada cenário de memória alocada. Observando o gráfico do tempo de execução quando ocorrem *coldstarts* (à direita), é possível identificar uma tendência: quanto mais memória alocada, menor é o tempo de *coldstart*. Este comportamento é inesperado, pois imaginava-se que conforme a alocação de memória fosse aumentando, maior seria o impacto do carregamento da função durante o *coldstart*.

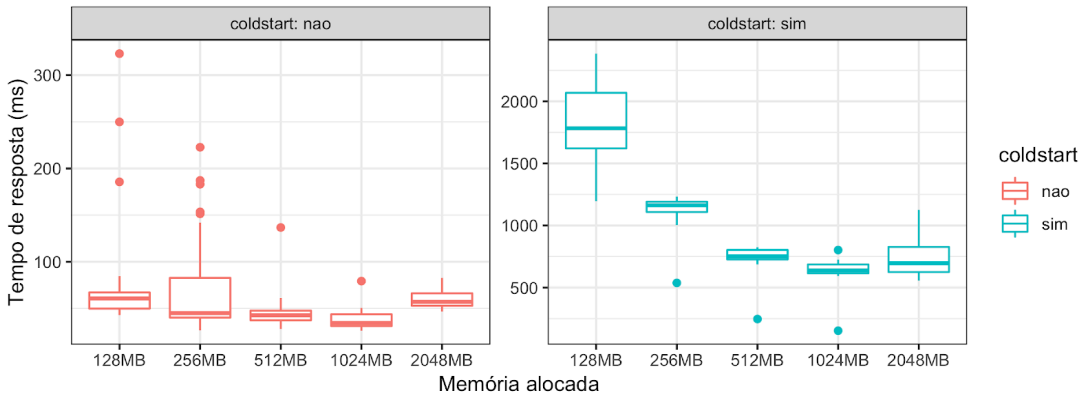

Figura 4. Boxplot do tempo de resposta com e sem *coldstart*, para diferentes cenários de memória alocada para a função.

Este fenômeno pode ser explicado através da documentação do AWS Lambda[4], que diz que a capacidade de CPU alocada para a função é proporcional à quantidade de memória alocada para a mesma. Portanto, uma hipótese é que funções que alocam mais memória, como também possuem maior capacidade de CPU, carregam a função e respondem às requisições mais rapidamente. Uma ressalva é que ao aumentar a memória alocada de 1024MB para 2048MB, não se observou uma diminuição no tempo de resposta; houve, na verdade, um pequeno acréscimo na mediana. Uma possível explicação, no entanto, é que a partir deste ponto, CPU deixa de ser um gargalo e não melhorar mais o desempenho do carregamento e execução da função, enquanto a quantidade maior de memória a ser alocada passa a fazer algum efeito no tempo de carregamento, mesmo que pouco significativo. Além disso, quando não há *coldstart* (gráfico há esquerda), o tempo de resposta apresenta mais *outliers* com valores maiores, mostrando que a capacidade menor de CPU também pode afetar o tempo de resposta da função mesmo sem *coldstart*.

5. Conclusão

O modelo de *serverless computing* está se tornando bastante atrativo para usuários de nuvem, devido à menor complexidade operacional por não precisar gerenciar servidores; da possível redução de custos ao ser cobrado apenas pelo uso da aplicação. Porém, ao desativar funções após um período de ociosidade para reduzir custos, o provedor gera um novo problema para a aplicação chamado *coldstart*, que é o tempo necessário para inicializar e implantar uma aplicação inativa ao chegar uma nova requisição. Neste trabalho, analisamos o *coldstart* em um provedor de *serverless computing*, medindo para diferentes cenários de memória alocada para a função: o tempo que uma função passa ociosa até que o provedor a torne inativa (havendo *coldstart*); o tempo de resposta das requisições com e sem *coldstart*; e o *overhead* que o coldstart impõe no tempo de resposta dependendo da quantidade de memória alocada.

Os resultados dos experimentos mostraram que: (1) *Coldstarts* aconteceram de forma sistemática após 30 a 42 minutos de ociosidade, dependendo da memória alocada para a função. Alguns *coldstarts* isolados também ocorrem com tempos de ociosidade menor, de 25 minutos, principalmente para quantidades maiores de memória alocada, o que supõe que em momentos de sobrecarga o provedor pode desativar funções que ocupam mais recursos mais cedo. (2) O *overhead* imposto pelo *coldstart* foi significativo para os experimentos realizados, sendo a diferença nas medianas do tempo de resposta com e sem *coldstart* de aproximadamente 751ms. (3) Quanto maior a

[4] https://docs.aws.amazon.com/lambda/latest/dg/resource-model.html

memória alocada, menor tende a ser o *overhead* do *coldstart*. A hipótese é que como o provedor aloca CPU proporcionalmente à memória, a função é inicializada e executada mais rapidamente quando há mais memória. Porém, quando se aumentou a memória de 1024MB para 2048MB, o *overhead* não diminuiu. A possível explicação é que a CPU deixou de ser um gargalo e aumentar sua capacidade já não afeta o tempo de resposta significativamente.

Com base nos resultados obtidos no experimento, sugerimos que: aplicações que passam muito tempo ociosas podem reduzir custos por não serem cobradas nestes períodos. Porém, se estes períodos forem superiores a 25 minutos e a aplicação for sensível à latência, precisando de tempos de resposta inferiores a 1 segundo para uma grande porcentagem das suas requisições, o *overhead* imposto pelo *coldstart* pode afetar a qualidade de serviço da aplicação, sendo mais adequado um serviço tradicional baseado em máquinas virtuais ou containers, ou estratégias para que funções não sejam desativadas. Por outro lado, se as aplicações se mantém ativas e raramente apresentam longos períodos de atividade, a facilidade de uso do modelo de *serverless computing* pode ser um grande atrativo para os usuários executarem e escalarem suas aplicações sem precisar gerenciar servidores.

Como possíveis trabalhos futuros, pretende-se investigar o impacto no *coldstart* adotando diferentes provedores de serviços, diferentes linguagens de programação e com experimentos mais exaustivos em dias da semana e horas diferentes para eliminar efeitos externos da análise.

Referencias

Baldini, I., Castro, P., Chang, K., Cheng, P., Fink, S., Ishakian, V., Mitchell, N., Muthusamy, V., Rabbah, R, Slominski, A. and Suter, P. (2017) "Serverless Computing: Current Trends and Open Problems". In: Research Advances in Cloud Computing. Edited by Chaudhary S., Somani G. and Buyya R, Springer, Singapore.

Cui, Y. “Finding coldstarts: how long does AWS Lambda keep your idle functions around?” (2017a) in: https://theburningmonk.com/2017/06/finding-coldstarts-how-long-does-aws-lambda-keep-your-idle-functions-around/. Acesso em: 28 set. 2018.

Cui, Y “How does language, memory and package size affect cold starts of AWS Lambda?” (2017b) in: https://read.acloud.guru/does-coding-language-memory-or-package-size-affect-coldstarts-of-aws-lambda-a15e26d12c76/. Acesso em: 22 out. 2018

Fowler, M. “Serverless Architectures” (2018). In: martinfowler.com. Disponível em: https://martinfowler.com/articles/serverless.html. Acesso em: 14 ago. 2018.

Gancarz, R. “Serverless Still Requires Infrastructure Management” (2018). In: InfoQ. Disponível em: https://www.infoq.com/articles/serverless-infrastructure-management. Acesso em: 14 ago. 2018.

Ken, F. "Why The Future Of Software And Apps Is Serverless" (2012). In: readwrite. Disponível em: https://readwrite.com/2012/10/15/why-the-future-of-software-and-apps-is-serverless. Acesso em: 08 ago. 2018.

Mell, P., Grance, T. (2011). The NIST definition of cloud computing.

McGrath, G., Brenner, P.R. Serverless Computing: Design, Implementation, and Performance, IEEE 37th International Conference on Distributed Computing Systems Workshops, 2017

Perez, C. “Serverless e AWS Lambda” (2018). In: Elo7 Tech. Disponível em: https://engenharia.elo7.com.br/serverless/. Acesso em: 20 ago. 2018.

Savage, N. (2018). Going serverless. Communications of the ACM, 61(2), 15-16.