Análise de Mecanismos de Serverless Computing em Ambientes de Nuvens Computacionais

Matheus Nícolas da Silva, Marcus Carvalho

Bacharelado em Sistemas de Informação – Universidade Federal da Paraíba (UFPB) Campus IV – Rua da Mangueira, S/N – Companhia de Tecido Rio Tinto CEP 58297-000 – Rio Tinto – PB - Brasil

{matheus.nicolas[1], marcuswac}@dce.ufpb.br

Abstract. *The objective of this work is to present the analysis of one of the most promising cloud application paradigms that emerged in this model - Serverless Computing. Being a new approach to cloud computing, it is much discussed because of its characteristics in the world of computer systems industry. Based on the exploratory research, this work was developed with the purpose of investigating the characteristics of serverless computing and performing a quantitative experiment in the AWS Lambda environment, investigating a phenomenon present in this approach, cold-start.*

Resumo. *O objetivo deste trabalho trabalho é apresentar a análise de um dos paradigmas de aplicações em nuvem mais promissores que surgiu neste modelo - Serverless Computing. Sendo uma nova abordagem de computação em nuvem, é muito discutido devido suas características no mundo da indústria de sistemas de computação. Com base na pesquisa exploratória, este trabalho foi desenvolvido com o intuito de investigar as características de serverless computing e realizar um experimento quantitativo no ambiente AWS Lambda, investigando um fenômeno presente nesta abordagem, o cold-start.*

1. Introdução

Desde o surgimento da computação em nuvem, há cada vez mais adesão a este modelo que inovou a forma com que se consome tecnologia. Há alguns anos, muitas pessoas e empresas habituaram-se a entregar suas necessidades computacionais para um provedor de serviços, sem precisar se preocupar com despesas de comprar e manter os próprios servidores. Porém, inicialmente ainda cabia exclusivamente ao usuário da nuvem determinar a capacidade dos recursos alocados em suas máquinas virtuais, como núcleos de CPU, espaço em memória e armazenamento em disco [Savage 2018].

Com o crescimento do consumo de computação em nuvem, diferentes modelos e estratégias surgiram para suprir essas necessidades, ampliando o espectro de serviços com relação à facilidade de uso e flexibilidade. Segundo Mell (2011), existem três modelos principais de computação em nuvem, sendo eles: Infraestrutura como Serviço (*IaaS*), no qual o consumidor tem controle sobre o sistema operacional, capacidade de recursos e aplicativos implantados em máquinas virtuais; Plataforma como Serviço (*PaaS*), no qual não há preocupação em gerenciar a infraestrutura como os recursos de hardware e o sistema

[1]"Trabalho de Conclusão de Curso (TCC) na modalidade Artigo apresentado como parte dos pré-requisitos para a obtenção do título de Bacharel em Sistemas de Informação pelo curso de Bacharelado em Sistemas de Informação do Centro de Ciências Aplicadas e Educação (CCAE), Campus IV da Universidade Federal da Paraíba, sob a orientação do professor Marcus Williams Aquino de Carvalho."

operacional; e o Software como Serviço (*SaaS*), que é a aplicação executada e gerenciada pelo provedor de serviços.

No entanto, novas soluções em nuvem não param de surgir. Um dos paradigmas de aplicações de nuvem mais promissores que emergiram neste contexto foi o chamado *Serverless computing*, em sua tradução livre "computação sem servidor", que tem como princípio deixar transparente ao usuário a gerência de servidores que hospedam suas aplicações. Um dos principais modelos de computação que surgiram no paradigma *serverless* é o de Funções como Serviço (*FaaS*), que fornece uma plataforma que permite desenvolver, executar e gerenciar funcionalidades de aplicações sem a complexidade de criar e manter uma infraestrutura que normalmente é associada ao desenvolvimento e lançamento de uma aplicação [Fowler 2012].

Este novo paradigma representa uma evolução do modelo de programação, abstração e implantação de aplicações e serviços no ambiente de *cloud computing*, tratando-se de uma prova de maturidade e ampla adoção de tecnologias em nuvem [Baldini et al. 2017]. Considerando os aspectos mencionados, é um assunto que se encontra em destaque na indústria de sistemas de computação, tendo em vista que os principais provedores de serviços de nuvem como Amazon, Google e Microsoft vêm investindo de forma massiva, bem como também tem surgido muitos projetos de código aberto, conferências e fornecedores de softwares voltados a esta nova abordagem [Fowler 2012].

Se opondo ao modelo tradicional no qual a aplicação é controlada pelo usuário e hospedada em servidores se mantendo em execução o tempo todo, as aplicações *FaaS* gerenciadas pelo provedor podem não estar sempre ativas, o que pode impactar no seu desempenho. Nesta abordagem durante a execução de uma função pode ser preciso associá-la a um servidor. Um cenário que contempla este ocorrido, é quando feito o *deploy* da função, ocorre uma primeira requisição, então o primeiro tempo de resposta é maior. Esse fenômeno é conhecido como *cold-start*.

Foi realizada uma pesquisa exploratória em busca de se familiarizar com o modelo de computação *serverless*, sendo possível notar que há poucos estudos voltados a abordagem. Uma hipótese para esta falta, é devido o modelo ter emergido recentemente na área de computação em nuvem. A proposta desta pesquisa é poder preencher um pouco da escassez de trabalhos voltados a *serverless computing*, com o objetivo de levantar os principais provedores de serviços e ferramentas *serverless*, realizar uma avaliação para medir o impacto no desempenho de um serviço *FaaS* - onde a aplicação não permanece em execução durante todo o tempo - medindo o atraso no tempo de resposta causado por funções que se tornam inativas após algum período de ociosidade - o fenômeno chamado de *cold-start*, uma das características deste modelo - inerente de um ambiente de *FaaS*.

As demais seções deste artigo estão estruturadas da seguinte forma: a seção 2 apresenta a fundamentação teórica, descrevendo os conceitos básicos de *serverless computing* e *FaaS*. A seção 3 apresenta um estudo de natureza exploratória identificando os diferentes tipos de provedores de serviços e ferramentas, seção 4 a avaliação de desempenho, metodologia e resultados e a seção 5 a conclusão.

2. Serverless Computing

Segundo Savage (2018), o termo *serverless* pode parecer confuso, pois mesmo neste paradigma as aplicações necessitam de processos de servidores para serem hospedadas e executar. A diferença deste novo modelo com relação ao modelo tradicional baseado em servidor é que o operador da aplicação *serverless* não é o responsável por gerenciar o hardware ou processos do servidor, já que essas responsabilidades são terceirizadas para o provedor de nuvem. Ou seja, o termo remete à ideia do provisionamento e gerência dos servidores que hospedam a aplicação não serem de responsabilidade do usuário. Portanto, os servidores estão presentes na abordagem *serverless* de forma transparente, suprindo todas as variedades de aplicações *serverless*, mas sem que o usuário consiga interagir ou gerenciar esses servidores. Além do provedor de nuvem se encarregar do gerenciamento dos servidores e contêineres para o usuário, eles também oferecem serviços que providenciam blocos comuns de construção para criação de arquiteturas *cloud* nativas, como por exemplo armazenamento, gerenciamento de API e monitoramento [Gancarz 2018].

O termo *serverless* se popularizou depois do seu lançamento no Amazon AWS Lambda[2] em 2014, e cresceu ainda mais após o lançamento do Amazon API Gateway[3] em julho de 2015. No final do mesmo ano, o projeto de código aberto Javascript Amazon Web Services foi renomeado para Serverless Framework[4], continuando esta tendência. Em meados de 2016, *serverless* já havia se tornado um assunto dominante nessa área, dando espaço à origem da série Serverless Conference[5] e a vários fornecedores adotando o termo em tudo, desde marketing de produtos até descrições de cargos.

Em comparação com o modelo tradicional baseado em servidores e máquinas virtuais, *serverless computing* é mais abstrato. No modelo tradicional, uma máquina virtual (VM) emula um sistema completo do computador dentro de outro computador, ainda existindo o conceito de memória para gerenciar e de drivers na VM que pode chegar ao hardware [Savage 2018 apud Brenner 2017]. Ao executar uma função com o modelo *serverless*, a única responsabilidade do desenvolvedor é codificar suas funções; necessário alocar a quantidade de memória que a função vai ter para sua execução antes ser feito deploy, apenas quando alguns provedores possuem esta restrição como por exemplo, AWS Lambda e Google Functions; essa quantidade pode variar de acordo com o provedor de serviço e outra observação é que como o provedor é quem fica responsável pela infraestrutura, um meio de economizar recursos é mantendo a aplicação que utiliza esta abordagem em execução durante um certo período de tempo, e que ao levantar a função quando não está ativa, impacta no tempo de resposta durante sua primeira execução, o *cold-start*.

2.1. Function as a Service (FaaS)

Além dos três modelos principais citados anteriormente (*Iaas*, *Saas*, *Paas*), Savage (2018) também menciona que *serverless* é uma quarta categoria de modelo de computação em nuvem chamada Function as a Service (*FaaS*) ou Função como Serviço. O *FaaS* possui características como execução de código *backend* sem gerenciar seus próprios sistemas de servidor ou suas próprias aplicações de servidor de longa duração; não requer codificar para um *framework* ou biblioteca específica, por exemplo as funções do AWS Lambda podem ser implementadas em Java, Node.js, Python, Go, entre outras linguagens; a implantação é

[2] https://aws.amazon.com/lambda/
[3] https://aws.amazon.com/api-gateway/
[4] https://serverless.com/
[5] http://serverlessconf.io/

diferente dos modelos tradicionais, já que o código da função é carregado para o provedor que é responsável por toda parte de provisionar recursos, instanciar VMs e gerenciar processos, por exemplo; escalonamento horizontal é totalmente automático, elástico e gerenciado pelo servidor; são tipicamente acionadas por tipos de eventos definidos pelo provedor; e a maioria dos provedores permitem que as funções responda a solicitações HTTP [Fowler 2012].

Para falar um pouco da aplicação do *FaaS* com base no exemplo citado por Fowler (2012), a Figura 1 ilustra a arquitetura de um servidor de anúncios onde o usuário clica em um anúncio e é redirecionado para o seu destino. Ao mesmo tempo é necessário coletar os eventos de clique que ocorrem para poder cobrar do anunciante. O servidor de anúncios responde ao usuário de forma síncrona e envia a mensagem de clique para o canal de mensagem, então essa mensagem é processada de forma assíncrona por uma aplicação processador de cliques, que atualiza um banco de dados como forma de economizar no orçamento do anunciante. Imagine o processador de cliques como uma aplicação de modelo tradicional baseado em servidores e VMs; ela permanece em execução durante um longo período de tempo. Essa aplicação pode facilmente ser substituída por uma função *FaaS,* pois essa aplicação não precisa de um servidor gerenciável e nem estar em execução durante todo o tempo. Como citado anteriormente, funções em um ambiente *FaaS* podem ser acionadas por eventos, onde cada clique no canal de mensagem pode acionar a função. Seu ambiente também pode processar várias mensagens simultaneamente, instanciando várias cópias do código da função. Sobre o código, a única mudança para o *FaaS* seria o método principal de inicialização e provavelmente o código específico para o manipulador de mensagens, o restante do código como por exemplo, salvar no banco de dados não é diferente em um mundo *FaaS*.

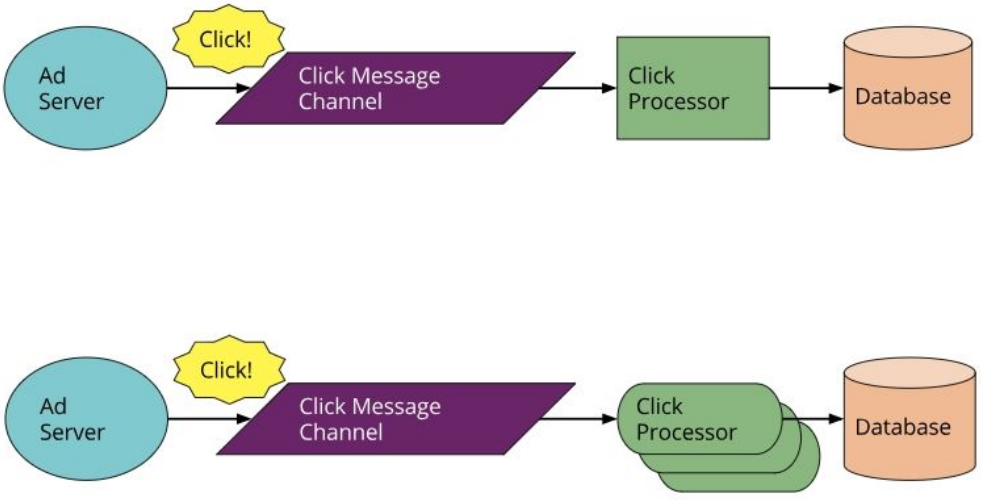

Figura 1. Servidor de Anúncios no modelo FaaS.

Fonte: https://martinfowler.com/articles/serverless.html

Caso um dia a arquitetura da Figura 1 tivesse seu uso aumentado para dez vezes, a aplicação de cliques conseguiria lidar com várias mensagens por vez, executando instâncias suficientes de funções para processar várias mensagens com escalonamento automático sem precisar reconfigurar manualmente. Basta escrever a função com antecedência para assumir o

paralelismo de escala horizontal, a partir desse ponto o provedor de *FaaS* automaticamente lida com todas as necessidades de dimensionamento.

Vale lembrar que as funções *FaaS* possuem restrições significativas quando o assunto é seu estado em relação a instância de máquina, como por exemplo, dados que você armazena em variáveis na memória ou que escreve no disco local. Por mais que se tenha esses armazenamentos disponíveis, não quer dizer que há garantia de que tal estado é persistente em várias chamadas e também não se deve assumir que o estado de uma invocação estará disponível sempre para outra invocação da mesma função; portanto elas são chamadas de *stateless* ou sem estado.

FaaS possui algumas restrições significativas referente ao tempo de execução de suas funções e, nem sempre aplicações que utilizam este modelo estão em execução. Muitas vezes ao executar uma função, ela precisará ser associada a um servidor e passará por todo um processo que acarreta em um tempo adicional no tempo de resposta - *cold-start* - até permanecer ativa durante um período de tempo (em execução, o tempo de resposta das requisições seguintes são menores). Com este fenômeno presente no *FaaS*, realizar o experimento viabilizando esta característica entre as demais é importante para analisar como pode impactar em aplicações que adotam este modelo de computação em nuvem.

3. Serviços e Ferramentas de Serverless Computing

Esta seção apresenta um estudo de natureza exploratória, que busca promover o aprimoramento e ideias sobre *serverless computing*, identificando os diferentes tipos de serviços e ferramentas de *serverless* para fazer um comparativo entre elas, discutir as definições e suas propriedades.

3.1 Serviços

A primeira etapa ao utilizar a abordagem *serverless* é escolher o provedor de serviços e/ou as ferramentas. Savage (2018) menciona que todos os principais provedores de serviços em nuvem fornecem *serverless computing*, onde a Amazon vem com o AWS Lambda, Google com Cloud Functions e Microsoft com o Azure Functions. Todos esses provedores possuem as características já mencionadas na seção 2 sobre *serverless* e *FaaS* como: o provedor é responsável por toda parte de provisionar recursos, escalabilidade, entre outras. A Tabela 1 apresenta um comparativo sobre esses provedores de serviços para a abordagem *serverless*.

Provedor	Recursos	Linguagens	Precificação	Diferencial
Amazon: AWS Lambda	Memória: 128MB até 3GB. Tempo máximo de execução: 300 segundos (5 minutos).	Node.js, .NET, Java, Python, Go.	Primeiro 1 milhão de requisições grátis, $0,20 por 1 milhão de requisições depois e $0,00001667 USD para cada GB-segundo[6].	-

[6] "AWS Lambda – Definição de preço - Amazon AWS." https://aws.amazon.com/pt/lambda/pricing/ Acessado em 20 out. 2018.

Google: Cloud Functions	Memória: 128MB até 2GB. Tempo máximo de execução: 540 segundos (9 minutos).	Node.js, Python.	Primeiro 2 milhões de requisições grátis, depois $0.40 por cada milhão de requisições, e a cobrança varia de acordo com a frequência e quantidade de memória também[7].	-
Microsoft: Azure Functions	Memória: 128MB até 1,5GB (1532MB). Tempo máximo de execução: 5 minutos.	C#, F#, Node.js, Java, Python e PHP.	Plano de consumo incluem um subsídio gratuito mensal de 1 milhão de requisições 400.000 GB-s e de consumo de recursos por mês[8].	Durable Functions

Tabela 1. Algumas características dos serviços *serverless* de cada um dos principais provedores serviços em nuvem.

Cada provedor tem um limite de memória para seus serviços *serverless* que deve ser configurado manualmente para cada função, e dentre eles a Amazon permite alocar a maior quantidade de memória. Além dessas informações, todos os serviços não cobram pelo tempo ocioso da função, possuem tolerância a falha, execuções paralelas, entre outros recursos. Em questão de diferencial, *Azure Functions* possui uma extensão chamada *Durable Functions*. Como mencionado na seção 2.1, não há garantia de que o estado de uma função é persistente em várias invocações, então durante a pesquisa foi possível identificar uma forma de poder manter o estado de uma função. Durable Functions é uma extensão do serviço da Microsoft que permite escrever funções *stateful* ou cheia de estado que basicamente garante que o estado local da função não seja perdido, mesmo com o processo reiniciando ou a VM sendo reinicializada.

Tomando como base as afirmações de Perez (2018), as situações de uso a seguir descreve quando é recomendável adotar a abordagem *serverless* e quando não é:

Recomendável:

- Aplicações pequenas e rápidas;
- Processamento assíncrono em resposta a eventos;
- Aplicações com padrão de consumo que possui picos apenas durante um período de tempo no dia a dia. Por exemplo, durante todo o dia a aplicação se mantém com o uso de 20% e apenas durante um período do dia seu uso é mais intenso, chegando aos

[7] "Pricing | Cloud Functions Documentation | Google Cloud." https://cloud.google.com/functions/pricing. Acessado em 20 out. 2018.
[8] "Preço - Funções | Microsoft Azure." https://azure.microsoft.com/pt-br/pricing/details/functions/. Acessado em 20 out. 2018.

90% de uso, neste cenário é recomendável adotar uma abordagem *serverless*;

- Funcionalidades que passam boa parte do tempo com recurso ocioso.

Não recomendável:

- Mesmo havendo um gerenciamento fácil e de rápido desenvolvimento sem infraestrutura para se preocupar, não é recomendável para casos de uso muito intensos como número de requisições muito alto. No final do mês, pode pesar no bolso;
- Aplicações com processamentos longos, pois uma característica dessa arquitetura é o limite de tempo que varia de acordo com cada provedor;
- Processamentos com grande consumo de memória.

3.2 Ferramentas

Quando se pensa em desenvolver uma aplicação *serverless*, primeiro por padrão é pensado qual o provedor em nuvem escolher, já que esta abordagem é ofertada desta forma com sua infraestrutura gerenciada por um dos demais provedores. Depois de dado este primeiro passo, escolhemos qual a ferramenta utilizaremos como mecanismo de desenvolvimento para nos auxiliar no decorrer desse processo. Existem inúmeras ferramentas que podem ser adotadas ao contexto de desenvolvimento, entre elas: Claudia.js[9], OpenFaaS[10], Serverless Framework, existe um acervo delas e todas essas mencionadas são *Open Source.*

Uma observação é que as ferramentas não se limitam ao contexto de desenvolvimento. Também há as que são voltadas para o contexto de experimentos como o Charles Proxy[11], Hey[12], Step Functions[13], entre outras. De forma geral, são bem produtivas e possuem utilidades diferentes dependendo da necessidade. Para poder realizar o nosso experimento, foram adotadas duas ferramentas: Serverless Framework para tornar produtivo o processo de desenvolvimento, gerenciamento e *deploy* das funções para o experimento e o Step Functions para podermos realizar as medições sobre o fenômeno do *cold-start.*

Concluindo esta seção, o provedor de serviços escolhido para o experimento foi o AWS Lambda com o objetivo de realizar o experimento. A ferramenta para realizar as medições escolhida foi o AWS Step Functions, outro serviço da Amazon que além de ter integração com as funções Lambda, nos permite realizar uma análise mais precisa devido ao acesso interno no mesmo provedor. Na seção 4 há mais detalhes sobre o experimento.

4. Avaliação de Desempenho

Esta seção apresenta uma avaliação de desempenho realizada através de experimentos de medição, fazendo o *deployment* de funções em um ambiente *serverless computing* que permitam realizar medições avaliando o fenômeno do *cold-start.* A metodologia usada na avaliação e os resultados dos experimentos são apresentados nas seções a seguir.

[9] "Claudia.js." https://claudiajs.com/
[10] "GitHub - openfaas/faas: OpenFaaS - Serverless Functions Made ...." https://github.com/openfaas/faas.
[11] "Charles Proxy." https://www.charlesproxy.com/
[12] "rakyll/hey - GitHub." https://github.com/rakyll/hey
[13] "AWS Step Functions." https://aws.amazon.com/pt/step-functions/

4.1 Metodologia

A avaliação de desempenho realizada teve como principal objetivo investigar o fenômeno do *cold-start* em um ambiente de *serverless computing,* apresentando os resultados de forma quantitativa através de experimentos de medição. Os experimentos foram realizados usando o serviço AWS Lambda, com funções desenvolvidas em Node.js. É importante deixar claro o ambiente de execução e a linguagem de programação utilizados no escopo da avaliação, pois experimentos apresentados por Cui (2017), mostraram que existe uma diferença no *cold-start* de acordo com a linguagem e ambientes utilizados.

Como objetivos específicos de pesquisa, buscou-se responder às seguintes questões com base nos experimentos:

1. Depois de quanto tempo de inatividade uma função é desativada, tendo como consequência o *cold-star*t em uma requisição seguinte?

2. Com que frequência são observados *cold-starts*, ao se variar o intervalo entre requisições?

3. Qual o tempo gasto (*overhead*) adicional que o *cold-start* impõe em média no tempo de resposta das requisições, em comparação ao tempo de resposta de requisições que não sofrem *cold-start*?

4. Qual o impacto da quantidade de memória alocada para a função no *overhead* do *cold-start* e no intervalo de inatividade para que ocorra um *cold-start*?

   Com esses objetivos observados, foi definido a pesquisa.

Baseado na metodologia de avaliação de Cui (2017), foi utilizado o serviço AWS Step Functions para gerar a carga periódica e realizar as medições. Este serviço possibilita a criação de máquina de estados e um fluxo de execução de tarefas, que pode ser integrada com as funções Lambda e, em cada estado definido no serviço, permite criar uma tarefa que invoca uma função, ou uma tarefa que aguarda por um tempo especificado com base na entrada do usuário para em seguida mudar de estado. O Step Functions também permite executar tarefas simultâneas, detectar erros, entre outras possibilidades. A medição foi realizada utilizando como base em funções em Node.js disponibilizadas pelo autor do experimento Cui (2017): *system-under-test* e *find-idle-timeout*[14].

Foram criadas cinco versões da função *system-under-test* com alocações de memória diferente: 128MB, 256MB, 512MB, 1024MB e 2048MB, para avaliar o impacto da alocação de memória no *cold-start*. Vale salientar que a capacidade de CPU[15] não é escolhida pelo usuário, mas a Amazon indica que ela é proporcional à capacidade de memória escolhida; ou seja, se uma função aloca duas vezes mais memória que outra, ela também terá duas vezes mais capacidade de processamento. Também é necessário um mecanismo para aumentar

[14] "GitHub - theburningmonk/lambda-when-will-i-coldstart" https://github.com/theburningmonk/lambda-when-will-i-coldstart/tree/master/functions. Acessado em 22 out. 2018.

[15] "Configuração de funções Lambda - AWS Lambda." https://docs.aws.amazon.com/pt_br/lambda/latest/dg/resource-model.html. Acessado em 22 out. 2018.

gradativamente o intervalo entre as invocações até atingir dez *cold-starts* consecutivos, onde vai indicar o tempo máximo que a função passa ociosa.

Para executar a máquina de estados, é necessário passar como parâmetro um JSON com três atributos: target, a função *system-under-test* com o tamanho de memória alocada a ser executada; coldstarts, necessário para contar o total de *cold-starts* ocorridos no experimento e *wait*, o intervalo que o estado *Wait* presente na Figura 2, irá aguardar até executar a próxima requisição. Então foram feitas cinco execuções no AWS Step Functions, uma para cada função com os atributos no JSON sendo a descrição da função a ser executada, coldstarts com o valor 0 e o wait com o valor de 600 que é o total de segundos que inicialmente a máquina de estados irá aguardar quando chegar ao estado *Wait.* Então, no estado *FindIdleTimeOut* temos uma função Lambda responsável por gerar as requisições a função *system-under-test*, responsável por detectar se houve ou não *cold-start.*

FindIdleTimeout também é responsável por coletar as métricas de desempenho e determinar quanto tempo se deve aguardar para gerar a requisição seguinte. Foi modificada para que fosse possível coletar as métricas com os resultados da medição sendo elas: *target*, com a descrição da função (exemplo, *system-under-test-128); interval*, o intervalo em que a função foi invocada; *requestId*, que é o contador de *cold-starts* para cada intervalo; *isColdStart*, que é um booleano que identifica se ocorreu ou não *cold-start* para cada requisição gerada; e *responseTime* para medir o tempo de resposta de cada requisição.

A função *system-under-test* possui uma variável booleana chamada *isColdStart* que inicia como *false*, mas ao chegar a primeira requisição ela muda para *true*, registrando um *cold-start* para esta função. Enquanto a função se mantém ativa e novas requisições chegam, a variável *isColdStart* mantém seu estado e o *cold-start* não acontece. No momento que a instância da função é desativada, após um certo período de inatividade, a função perde seu estado e uma nova requisição sofrerá um *cold-start*, iniciando novamente a variável *isColdStart* como false e passando pelo mesmo ciclo até se tornar inativada novamente

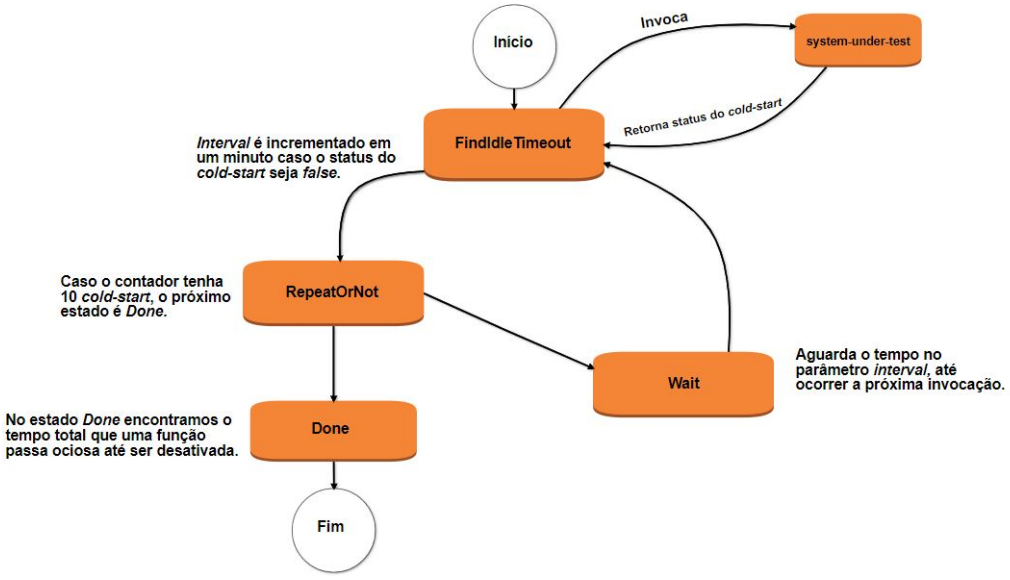

Figura 2. Fluxo de execução do experimento na máquina de estados AWS Step Functions.

Observando a Figura 2, a execução do fluxo da máquina de estados ocorre da seguinte forma:

- Recebendo o JSON descrito anteriormente, *FindIdleTimeout* recebe os atributos que indicam qual a função *system-under-test* ele irá invocar.
- É feito a requisição e *system-under-test* retorna o status do *cold-start*, caso o resultado seja *true*, o contador *requestId* incrementa 1 no contador, caso contrário, o atributo wait recebe um incremento de 60 segundos.
- Chegando no estado *RepeatOrNot*, há uma verificação para caso o contador *requestId* seja 10, se atingiu este total, então a função tem o seu tempo máximo ocioso encontrado que será o valor do wait no momento, assim encerrando o experimento para aquela função, caso contrário, o próximo estado será o *Wait* que aguardará o tempo que consta no atributo wait até gerar uma próxima execução.

Esse fluxo de execução é um *loop* que nos testes durou cerca de um dia para poder executar as funções até encontrar o tempo máximo ocioso de cada uma. Imagine que você contou 10 execuções, onde todas o status retornado por *system-under-test* foi de *cold-start* sendo *false,* então cada vez que o status chegava a *FindIdleTimeout,* era incrementado um minuto no atributo wait que ao chegar no estado *Wait,* aguarda 11 minutos até a próxima requisição, depois 12 minutos, depois 13 e assim sucessivamente.

Executando a máquina de estados para cada função, a análise foi realizada e os resultados estarão descritos na seção 4.2 a seguir.

4.2 Resultados

A avaliação de desempenho foi executada com base nas configurações especificadas na seção anterior. Na Tabela 2 podemos observar o tempo máximo que uma função permanece ociosa, sem receber requisições, até ser desativada pelo provedor para diferentes cenários de capacidade de memória alocada para a função. Este tempo é o último valor guardado no parâmetro wait que foi sendo incrementado no Step Functions, indicando as 10 requisições de *cold-starts* consecutivos em minutos.

Total de memória alocada (MB)	Tempo máximo ocioso (minutos)
128	30
256	42
12	30
1024	30
2048	40

Tabela 2. Tempo de ociosidade para o provedor desativar a função e acontecer um *cold-start*.

Então, para nossa pergunta de número 1 e número 2 respectivamente: Depois de quanto tempo de inatividade uma função é desativada, tendo como consequência o *cold-star*t em uma requisição seguinte? Com que frequência são observados *cold-starts*, ao se variar o intervalo entre requisições? Analisando o tempo ocioso que leva para uma função ser desativada e começar a acontecer o *cold-star*t com base na Tabela 3, por volta dos 30 a 42

minutos foi o tempo medido para finalizar uma função Lambda com base nos experimentos, com 10 *cold-starts* consecutivos para estes intervalos.

A Figura 3 apresenta o tempo de resposta de requisições ao variar o intervalo entre requisições e a quantidade de memória alocada para cada função, identificando também se houve ou não *cold-start* naquela requisição. Ao observar os gráficos, consegue-se identificar uma tendência comum de alguns *cold-starts* no intervalo de 25 a 30 minutos em todos cenários. Uma hipótese é que nesse intervalo de tempo a Amazon faz algum tipo de verificação para checar se a função está ociosa e decide desativá-la dependendo de fatores externos (exemplo, carga total na máquina onde a função roda ou no cluster inteiro da Amazon).

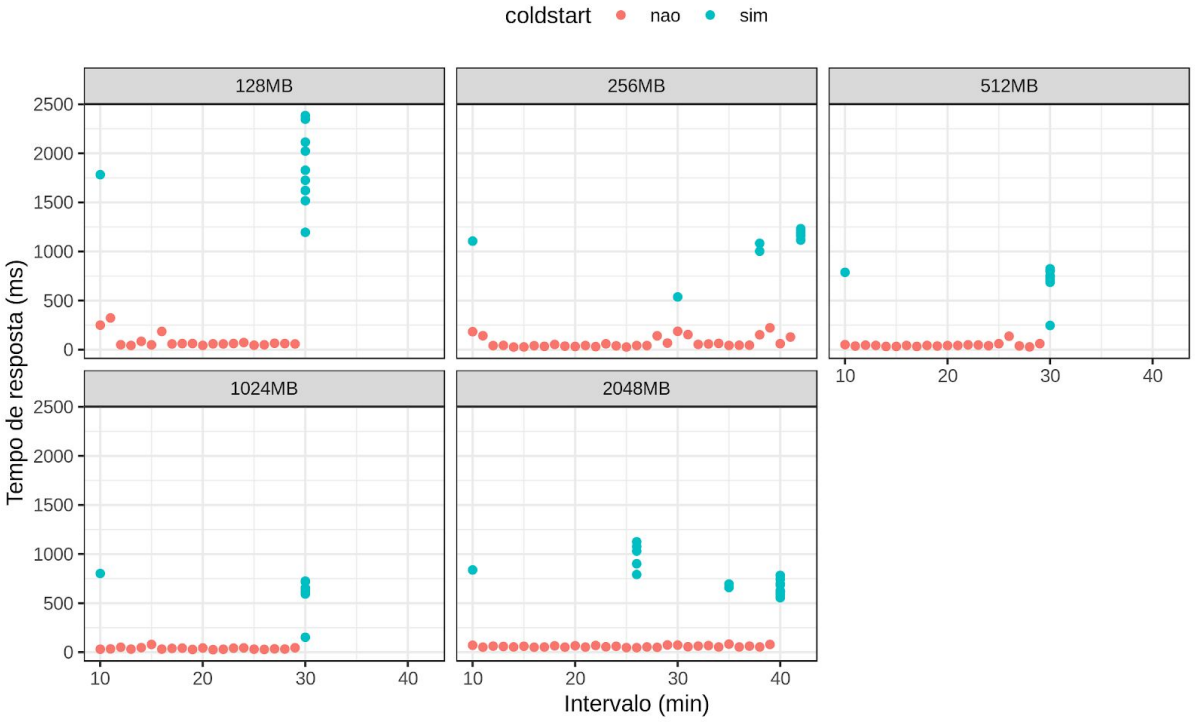

Figura 3. Tempo de resposta das requisições com e sem *cold-start*.

A função de maior alocação de memória (2GB) foi a que mais apresentou *cold-start* antes de atingir os 10 consecutivos, onde no intervalo de 26 minutos houveram 5 *cold-starts* na tentativa de encerrá-la, e novamente alguns *cold-starts* foram observados no intervalo de 35 minutos. Uma outra hipótese para estes *cold-starts* mais frequentes em intervalos não tão longos é devido à sua grande quantidade de recursos alocados; a Amazon pode tentar desativar primeiro funções que estão ociosas e consomem mais recursos em momentos de pico de carga na máquina ou cluster.

Na Figura 4, temos o gráfico de *boxplot* utilizado para avaliar a distribuição empírica dos dados, que no nosso caso é a mediana do tempo de resposta de todas as requisições de todas as funções. Observando as medianas para todas as funções, podemos comparar o tempo de execução quando há e quando não há *cold-start*, a nossa pergunta de número 3: Qual o tempo gasto (*overhead*) adicional que o *cold-start* impõe em média no tempo de resposta das requisições, em comparação ao tempo de resposta de requisições que não sofrem *cold-start*? Os pontos dispersos presentes na figura são chamados de *outliers*; são os valores discrepantes

no nosso gráfico e eles são a causa de adotarmos a mediana na comparação. A mediana do tempo de execução quando há *cold-start* é de 802 milissegundos, enquanto sem *cold-start* este tempo é de 50 milissegundos. A diferença entre as medianas, consideramos como sendo o *overhead* médio de desempenho imposto pelo *cold-start* para as funções avaliadas.

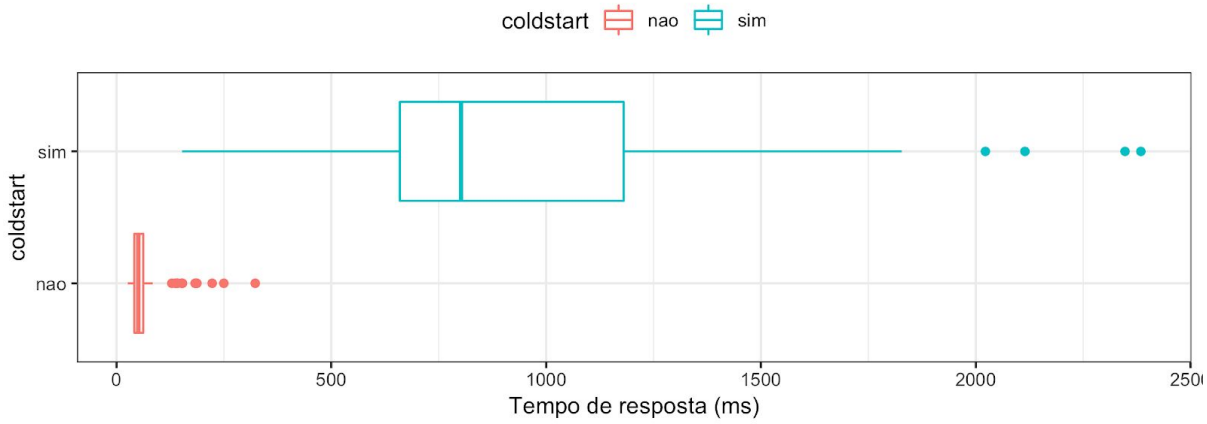

Figura 4. Gráfico de *boxplot* com a mediana do tempo de resposta total com e sem *coldstart.*

A análise anterior comparou as medianas de forma genérica, juntando os resultados para todos os cenários de alocação de memória. Porém, queremos saber também se a quantidade de memória que a função aloca tem impacto no tempo de resposta e nos *cold-starts*, no caso, a última pergunta de número 4: Qual o impacto da quantidade de memória alocada para a função no *overhead* do *cold-start* e no intervalo de inatividade para que ocorra um *cold-start*?.

A Figura 5 apresenta o boxplot do tempo de resposta para cada cenário de memória alocada e dividindo entre requisições com e sem *cold-start*. Observando o gráfico, é possível identificar uma tendência: quanto mais memória alocada, menor é o tempo de *cold-start.* Este comportamento é inesperado, pois esperava-se que conforme a alocação de memória fosse aumentando, maior seria o impacto de carregamento da função no *cold-start*. Porém, este acontecimento pode ser explicado na documentação da Amazon, que diz que quanto mais memória alocada, mais CPU[16] proporcionalmente tem alocada para a função. Por exemplo, caso uma função teve 512MB de memória alocada, ela possui quatro vezes mais capacidade de CPU que uma função de 128MB, então provavelmente com mais capacidade de CPU o carregamento e execução da função se torna mais rápido, principalmente porque a função usada não exige muita capacidade de memória para executar.

Uma ressalva é que ao aumentar a memória alocada de 1024MB para 2048MB, não se observou uma diminuição significativa no tempo de resposta; houve, na verdade, um pequeno acréscimo na mediana do tempo da função. Uma possível explicação, no entanto, é que a partir desse ponto, adicionar mais CPU não irá mais melhorar o desempenho do carregamento e execução da função, enquanto a quantidade maior de memória alocada passa a fazer algum efeito, mesmo que pouco significativo.

[16] "Configuração de funções Lambda - AWS Lambda." https://docs.aws.amazon.com/pt_br/lambda/latest/dg/resource-model.html. Acessado em 22 out. 2018.

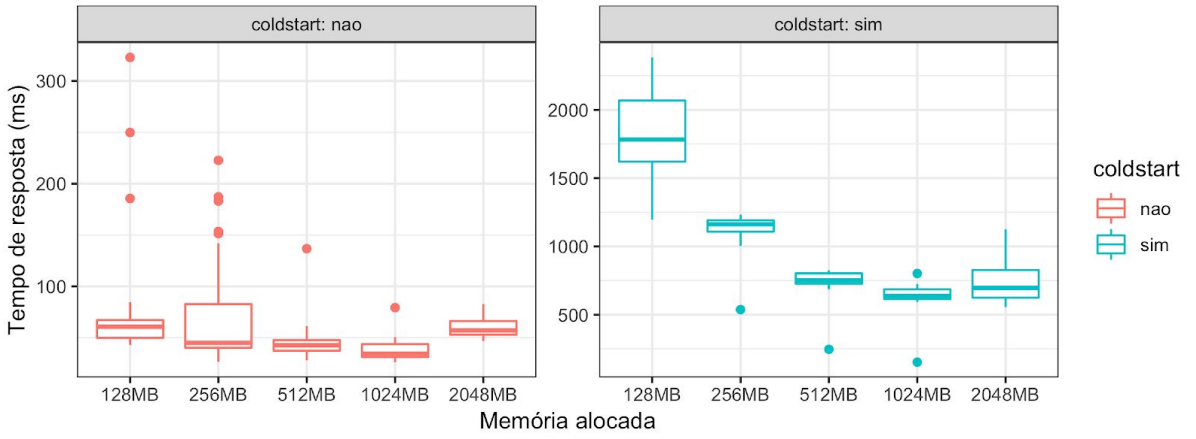

Figura 5. Mediana do tempo de resposta para cada função.

O gráfico da Figura 6 é um complemento da Figura 5, para podermos ver melhor a diferença entre as medianas quando há *cold-start* e quando não há (*overhead)* para cada função de acordo com sua alocação de memória. Como os gráficos da Figura 5 estão em escalas diferentes, não teria como observar a diferença de forma clara a diferença no mesmo gráfico.

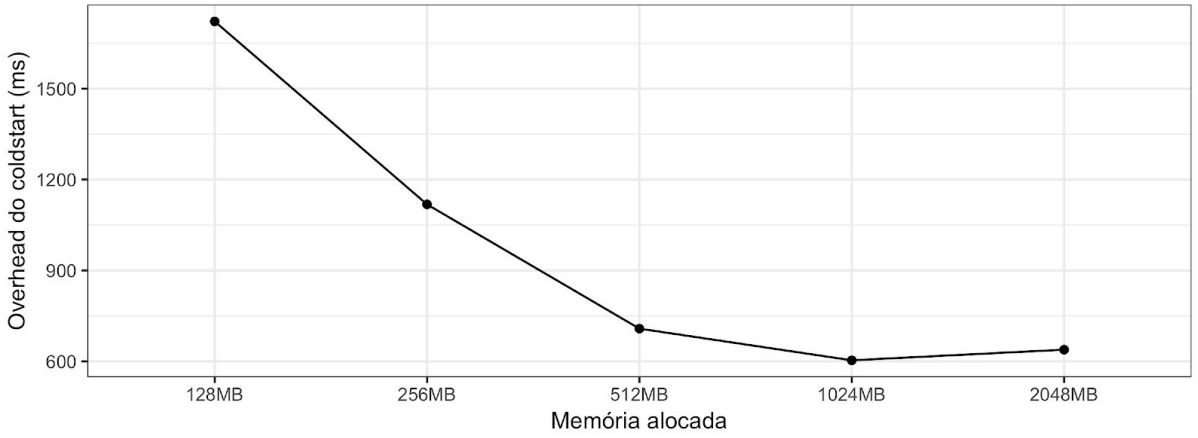

Figura 6. *Overhead* para cada função com base na alocação de memória.

Conforme o aumento da quantidade de memória na Figura 6, é possível ver que o gráfico segue a tendência da diferença de mediana para cada função. De acordo com a quantidade de memória alocada, o *overhead* tende a diminuir e este acontecimento se dá pelo fato de quanto mais memória alocada, mais CPU para processamento. Apenas não há um impacto significativo ao se tratar do aumento de 1GB para 2GB, onde o gráfico não segue a tendência, em vez de diminuir o *overhead*, ele tem um pequeno aumento.

5. Conclusão

Serverless computing é uma nova abordagem de computação em nuvem muito promissora, que possui uma forma diferente de arquitetura em relação a outros modelos de *cloud computing*, tornando mais fácil o desenvolvimento de aplicações que adotam este modelo. Investigar os mecanismos deste ambiente é de grande valia, definindo suas características,

provedores de serviço, ferramentas utilizadas e realizando uma análise de desempenho neste meio, com a motivação de poder contribuir cientificamente para um trabalho de uma tecnologia recente, que possui poucas publicações relacionadas ao seu contexto.

O experimento identificou que o fenômeno do *cold-start* existe no ambiente do AWS Lambda de acordo com as características deste modelo de computação em nuvem. Tratando-se do limite de tempo que as funções passam ociosas até serem desativadas pelo provedor, o intervalo de tempo para acontecer *cold-starts* nos experimentos foi entre 30 e 42 minutos, que é um tempo bem elevado de ociosidade e que indicam que aplicações com grande frequência de requisições raramente enfrentarão *cold-starts*.

Com base nos resultados obtidos no experimento, levando o *overhead* em consideração, caso o tempo adicional entre 600 milissegundos para cenários com muita memória alocada e de quase dois segundos para cenários com pouca memória para algumas requisições impacta negativamente sua aplicação de forma significativa, em casos que a função permanece mais de 25 minutos ociosa para ser desativada, não é recomendado utilizar esta abordagem, sendo mais adequado um serviço tradicional baseado com máquinas virtuais ou containers, ou estratégias para que funções não sejam desativadas. No geral, enquanto sua função estiver ociosa, mas recebendo requisições em intervalos menores que 25 minutos, a probabilidade de ocorrer um *cold-start* é muito baixa; então. para esse caso, é recomendável adotar esta abordagem, caso contrário com períodos de mais de 25 minutos sendo crítico para sua aplicação, é melhor procurar outro modelo de computação em nuvem.

Como possíveis trabalhos futuros, pode-se investigar: o impacto no *cold-start* adotando diferentes provedores de serviços e diferentes linguagens de programação, já que os provedores dão suporte a diversos ambientes de execução, sendo de grande contribuição para quem busca melhor performance ao adotar a abordagem *serverless*. Também pode-se avaliar o impacto no *cold-start* ao adotar uma grande quantidade de dependências, identificando se quanto mais memória de fato usada maior será o overhead do *cold-start*.

Referências

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?” (2017) 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?” (2017) in: https://read.acloud.guru/does-coding-language-memory-or-package-size-affect-cold-starts-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.

Google Cloud. “Google Cloud Functions Documentation”. Disponível em <https://cloud.google.com/functions/docs/>. Acessado em: 19 de agosto de 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.

Microsoft Azure. “Documentação do Azure Functions”. Disponível em: <https://docs.microsoft.com/pt-br/azure/azure-functions/>. Acessado em: 20 de agosto de 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.