UNIVERSIDADE ESTADUAL DE CAMPINAS

Faculdade de Tecnologia

LEONARDO VIVEIROS SANTOS

IMPACTOS DA ARQUITETURA FAAS NOS CUSTOS DE COMPUTAÇÃO EM NUVEM

LIMEIRA/SP

2019

LEONARDO VIVEIROS SANTOS

IMPACTOS DA ARQUITETURA FAAS NOS CUSTOS DE COMPUTAÇÃO EM NUVEM

FAAS ARCHITECTURE IMPACTS IN CLOUD COMPUTING COSTS

Trabalho de Conclusão de Curso apresentado à Faculdade de Tecnologia da Universidade Estadual de Campinas como parte dos requisitos exigidos para a conclusão do curso superior em Análise e Desenvolvimento de Sistemas.

Orientador:
Prof. Dr. André Franceschi De Angelis

ESTE TRABALHO CORRESPONDE AO TEXTO PRINCIPAL ELABORADO PELO ALUNO LEONARDO VIVEIROS SANTOS, E ORIENTADO PELO PROF. DR. ANDRÉ FRANCESCHI DE ANGELIS.

LIMEIRA/SP
2019

Resumo

Desde a popularização das aplicações distribuídas pela Internet, estas consumiram mais e mais recursos, dada a grande escala de usuários. No tempo em que a maioria das empresas fornecedoras de *software* gerenciavam seu próprio *hardware*, a cada vez que se falava em aumentar recursos, muito investimento era necessário para compra de mais *datacenters* e locação ou compra de espaços físicos. Com os adventos de fornecedores de *Infrastructure-as-a-Service*, a compra de *hardware* deixou de ser obrigatória. Tornou-se possível provisionar máquinas virtuais de qualquer capacidade e pagar pelo tempo em que esta foi utilizada, eliminando investimento inicial em aquisição de *hardware* e consequentemente reduzindo o custo de infraestrutura em projetos de software. Entretanto, com o grande número atual de usuários na Internet, a demanda recebida por aplicações pode variar bruscamente por conta de eventos fora do controle da empresa fornecedora da aplicação. Então foram provisionadas máquinas virtuais de maior capacidade, capaz de lidar com toda a demanda recebida esporadicamente. Porém, não são todas as horas do dia em que há pico de tráfego, o que resultou na maior parte do tempo em máquinas virtuais de alto poder computacional e com baixa utilização, para atender algumas horas de alta demanda no dia. Este cenário de desperdício de recursos prometeu ser resolvido com o lançamento em 2014 da arquitetura *FaaS* (*Function-as-a-Service*), plataforma que possibilitou escrever funções que escalam sua capacidade com base na demanda recebida em tempo real, e não gera custos se não são executadas. Esta arquitetura promete ser a solução para aplicações cuja demanda varie bruscamente, eliminando desperdícios com infraestrutura em nuvem. Entretanto, ao realizar experimentos, autores chegaram em conclusões diferentes, contradizendo os resultados entre si. Para investigar o motivo da divergência na literatura atual, neste trabalho foram comparados os custos das máquinas virtuais com os custos das *FaaS*. Portanto, este trabalho conta com o desenvolvimento de duas aplicações *webservice*, com uma baseada em máquinas virtuais, e a outra baseada em *FaaS*. Estas aplicações foram submetidas a cargas de trabalho que começaram em 5 e foram até 1250 requisições *por segundo*, com cada cenário de carga sendo executado por uma hora, a fim de detectar o comportamento de cada infraestrutura, sua latência média e os custos gerados. Em linhas gerais, concluímos que o custo de máquinas virtuais é até 150 vezes menor do que o das *FaaS* quando a demanda é grande, pelo fato de as funções terem seu custo baseado no número de requisições, e as máquinas virtuais não se importarem para esta métrica, tendo custo fixo. Entretanto, ao atingir o maior cenário de requisições por segundo, a latência observada na máquina virtual foi 20 vezes maior do que o mesmo número de requisições na aplicação *FaaS.* Portanto, caso seja considerado o custo total para manter aplicações baseadas em máquinas virtuais com resiliência, segura e *online*, o custo observado nas *FaaS* não é tão grande, por automatizar todo este processo. Mas quando comparados custos absolutos das duas arquiteturas, as máquinas virtuais são mais baratas.

Abstract

Since the beginning of the development of web applications, more and more resources were needed due to the large scale of their user base. Back when software provider companies used to manage their own hardware, big investments took place to acquire more datacenters and rent or buy room for them. When cloud providers became reality, it wasn't mandatory to buy and manage hardware anymore. It became possible to rent virtual machines with the desired resources and only pay for the used time, cutting off the investment to buy hardware and consequently reducing infrastructure costs in software projects. However, given the big number of Internet users nowadays, the demand received by the applications can vary abruptly due to events out of control from the software provider company. So, companies provisioned more powerful virtual machines, able to deal with the increase in demand. But the spike in demand is sporadic, resulting in a lot of resources that just sits idle most of the time. It soon became a problem of wasting resources, therefore, money. The solution was promised in 2014 with the release of Function-as-a-Service. This platform made it possible to write functions that scales based on real-time demand, and don't charge when not being used. It may be the solution to applications that suffer spikes in traffic, reducing waste of infrastructure resources. Although, when experimenting with costs of both virtual machines and *FaaS*, recent authors achieved substantially different results. In order to clarify the reason of given divergence in current literature, this work compares the costs of virtual machines and Functions-as-a-Service. So, here is detailed the development of two webservice applications, one based on virtual machines and the other in *FaaS*. These applications were put under workloads that started in 5 and went to 1250 requests *per second*. Each of the workloads were run for one hour, in order to observe the availability of each infrastructure, the mean latency and their costs. In conclusion, the use of virtual machines is up to 150 times cheaper than *FaaS* when operating in high demand workload. It is due to the pricing of *FaaS*, that charge per execution. Virtual machines have a fixed price per hour, independently of the use. However, the latency of the virtual machines was more than 20 times higher than *FaaS* when experimenting with the highest workload. So, if considered the total cost to maintain virtual machines infrastructures with backups, safety and online, *FaaS* is not too expensive, because all this are on behalf of the cloud provider. However, if comparing absolute costs of both architectures, virtual machines are cheaper.

Lista de Figuras

Lista de Quadros

Lista de Tabelas

Lista de Gráficos

1. Introdução

Até o início dos anos 1990, muitas empresas de tecnologia possuíam *data centers* locais para prover suas aplicações com recursos computacionais e armazenamento de dados. Isto obrigava a construção ou aluguel de espaço físico com refrigeração apropriada, contratação de pessoal especializado para manutenção, segurança, para garantir a disponibilidade dos servidores, elevando os custos fixos com TI. Caso a demanda por recursos computacionais da aplicação aumentasse, era necessário comprar mais *hardware*, impossibilitando escalar os recursos computacionais e armazenamento de forma rápida [1].

Colocation data centers são empresas que fornecem espaço físico para instalação dos *data centers* de muitos clientes, fornecendo energia, refrigeração, segurança física e acesso à Internet. A principal vantagem em levar seus *data centers* para *Colocations* é gerenciar somente seu *hardware*, reduzindo os custos com energia elétrica e segurança física. Estas empresas também oferecem acesso à Internet de altíssima velocidade e otimização de refrigeração graças a escala de suas operações [2].

Uma máquina virtual é um programa de computador que executa um sistema operacional e aplicações neles instaladas, sendo um computador virtual independente de seu hospedeiro. Esta técnica possibilita a execução de vários computadores virtuais em um único *data center*, possibilitando a diminuição dos custos. A partir de 2000 [3], os avanços em virtualização permitiram otimizar os recursos computacionais de cada *data center* para sua capacidade total, pois várias aplicações distintas puderam utilizar o mesmo *hardware,* distribuindo os custos de cada servidor por aplicação [1].

IaaS (Infrastructure as a Service) são serviços de infraestrutura oferecidos por *cloud providers*, como armazenamento, rede, balanceadores de carga e máquinas virtuais. Em meados de 2006, a *Amazon Web Services* lançou o primeiro serviço *IaaS*, o EC2 *(Amazon Elastic Compute Cloud)* [4]. Atualmente, existem várias empresas que

oferecem *IaaS*, com serviços como *Microsoft Azure*[1], *Google Cloud Platform*[2], *IBM Cloud*[3], *Oracle Cloud*[4] e *Cisco Cloud Infrastructure Solutions*[5].

Fornecedores de *IaaS*, também chamados de *cloud providers*, tarifam seus clientes mensalmente com base nos recursos computacionais provisionados, normalmente horas de uso da CPU, espaço de armazenamento utilizado, volume do tráfego de rede, entre outros [4]. Este modelo permitiu que empresas de software foquem seus esforços no gerenciamento das aplicações, sendo de responsabilidade do provedor de *IaaS* a infraestrutura e *data centers* necessários para disponibilizar os recursos contratados [1].

A flexibilidade nos custos é um dos principais motivos para empresas de software adotarem os serviços de computação em nuvem, pois pagam somente os recursos provisionados, evitando o investimento inicial de criar e manter *data centers* próprios. Caso exista demanda por mais recursos computacionais, basta provisionar uma máquina virtual com maior capacidade de armazenamento, CPU e/ou memória RAM [5].

Aplicações que sofrem de aumento repentino de uso, como serviços de *streaming* de vídeo, processamento de imagem, treino de inteligência artificial, venda de ingressos, entre outros, se beneficiam da fácil expansão de recursos computacionais em utilizar *IaaS*. Entretanto, caso os recursos computacionais provisionados sejam maiores do que a necessidade da aplicação, é um sinal de desperdício nos custos de infraestrutura em nuvem [1] [6].

Com o intuito de minimizar o desperdício de recursos computacionais, a Amazon Web Services, pioneira na implementação de *FaaS (Function-as-a-Service),* ou *serverless computing*, anunciou em 2014 o *AWS Lambda*, serviço acionado por eventos em que cada função é executada em um contexto [7]. Assim como no *IaaS*, hoje as *FaaS* são oferecidas por muitos *cloud providers*, tendo destaque os serviços *AWS Lambda* [7], *Google Cloud Functions* [8], *Microsoft Azure Functions* [9] e *IBM Cloud Functions* [10].

As funções são executadas em um ambiente conteinerizado. Diferente das máquinas virtuais que duplicam a camada do sistema operacional, na conteinerização

[1] https://azure.microsoft.com/en-us/
[2] https://cloud.google.com/
[3] https://www.ibm.com/cloud/
[4] https://cloud.oracle.com/home
[5] https://dcloud.cisco.com/

não é necessário um novo sistema operacional por aplicação servida. Em aplicações conteinerizadas, são definidos ambientes de execução, como Node.js, Java, Python, Ruby, e o código fonte da função ou aplicação. Sendo assim, imagens conteinerizadas de aplicações são muito menores do que as máquinas virtuais contendo as aplicações, permitindo maior agilidade na criação e destruição destes ambientes. [11]

Em *FaaS*, as funções são disparadas por gatilhos de execução baseados em eventos que podem ser requisições HTTP, comandos de voz [12], novos objetos criados no *AWS S3* [13], novas mensagens em filas de mensageria, novos e-mails, entre outros [14].

Ao implementar o *backend* de um aplicativo móvel ou sistema web utilizando a arquitetura *FaaS*, todo trabalho de escalar o sistema com balanceamento de carga, resiliência a falhas e réplicas multi-região tornam-se responsabilidade do *cloud provider*. Isto pode possibilitar desenvolvedores a criarem aplicações de escala global com uma equipe muito reduzida [7].

Coldstart é o termo dado para a grande latência na primeira execução de uma função, que ocorre quando o contêiner da função não foi iniciado e não está em *cache*. Execuções seguintes se beneficiam do contêiner já iniciado, por isso os *cloud providers* não destroem o contêiner assim que cada função termina sua execução. O container da função permanece em *cache*, mas após certo tempo do último acionamento da função, o *coldstart* acontece novamente [15].

Entretanto, há algumas desvantagens em adotar o *serverless computing (FaaS)* quando comparado às máquinas virtuais, por questões como *coldstart*, difícil versionamento e implantação de diversas funções por aplicação no lugar de um único projeto [15].

A arquitetura *serverless* também apresenta desafios no momento de versionar e implantar novas versões da aplicação pois devem ser gerenciadas várias funções distintas, e com objetivo de simplificar esta tarefa, foi criado o *Serverless framework* [16]. Este framework permite controlar diversas funções *FaaS* de maneira automatizada por uma interface de linhas de comando, sendo possível apontar qual *cloud provider* está usando (*AWS, Azure, Google, IBM*...) e o *Serverless framework* se encarrega de implantar as novas versões das funções, abstraindo as diferenças entre as *FaaS* de diferentes *cloud providers* existentes [16].

Aplicações reais muitas vezes precisam lidar com autenticação e autorização de usuários finais. A autenticação também é um fator impactado ao substituir o

backend por *FaaS*, devido não ser possível manter um estado em memória, dada a volatilidade dos contêineres de execução. Em uma aplicação onde uma ação do usuário final dispara várias funções, a autenticação deve ser feita em cada uma das funções seguintes, uma vez que não é possível distinguir uma requisição feita por outra função já autenticada de uma requisição de um usuário final ainda não autenticado. Para contornar este problema, existem serviços de "autenticação-como-serviço" (*Auth as a Service)*, como por exemplo *Auth0*[6], *AuthRocket*[7] e *Amazon Cognito*[8], que permitem escalar a autenticação de usuários automaticamente assim como as *FaaS* [17].

Ao criar uma função *FaaS*, é necessário informar a quantidade de memória RAM alocada para cada execução, partindo de 128 Megabytes, até 3008 Megabytes no AWS Lambda [7]. Outros *cloud providers* possuem limites similares. Diferente do provisionamento de máquinas virtuais à custos mensais fixos, nas *FaaS* os *cloud providers* cobram pelo número de execuções e Gigabytes/segundo de memória RAM provisionados para a função. Isso significa que caso não exista nenhum evento acionando a função, nenhum custo será gerado. Porém caso existam 100 execuções em simultâneo da mesma função, o custo Gigabytes/segundos será multiplicado por 100 [6].

A quantidade de memória provisionada dita o tempo de CPU disponível para a função, portanto uma função configurada com 256 Megabytes de RAM possui duas vezes o poder computacional de uma função com 128 Megabytes RAM, influenciando diretamente no tempo de resposta da função, mesmo que esta não utilize toda a memória provisionada [7].

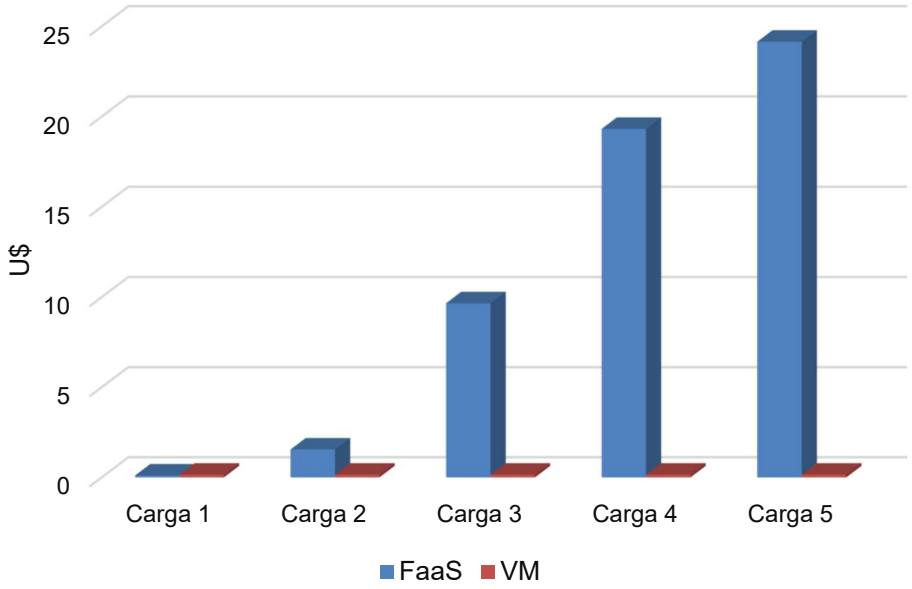
Dada suas características, *FaaS* aparenta ser uma boa maneira de reduzir o desperdício de recursos computacionais para aplicações em geral. Um *webservice* cujo tráfego aumente de forma imprevisível pode sofrer lentidão caso seja hospedado em uma máquina virtual. Para contornar este problema, pode ser provisionada uma máquina virtual capaz de suportar todo o aumento de tráfego, mas que passará a maior parte do tempo com baixa utilização de seus recursos. Este é o cenário em que *FaaS* aparenta ser uma boa alternativa por escalar os recursos pelo aumento de tráfego da aplicação automaticamente, sem desperdiçar memória e CPU. Entretanto,

[6] https://auth0.com/
[7] https://authrocket.com/
[8] https://aws.amazon.com/cognito/

para aplicações cuja latência é um fator crítico, *FaaS* não é adequado devido a imprevisibilidade da ocorrência dos *Coldstarts*.

O impacto nos custos é incerto ao adotar a infraestrutura *FaaS*. Recentes publicações [18] [19] chegaram a resultados divergentes sobre qual arquitetura possui menor custo de operação. São comparadas máquinas virtuais convencionais às *FaaS*. Villamizar [18] defende que a utilização de *FaaS* reduz em até 77% o custo de infraestrutura em nuvem. Entretanto a carga de trabalho utilizada em sua simulação é muito baixa, com 7 requisições por segundo, e não reflete o cenário da demanda de uma aplicação real. Já Eivy calculou que para uma aplicação que receba 150 requisições por segundo é ligeiramente mais barato usar o *AWS Lambda*. Porém, se a demanda da aplicação crescer para 30 mil requisições por segundo, o custo de usar *FaaS* é até três vezes maior do que máquinas virtuais.

Gráfico 1 – Custos por Carga de Trabalho em cada Aplicação

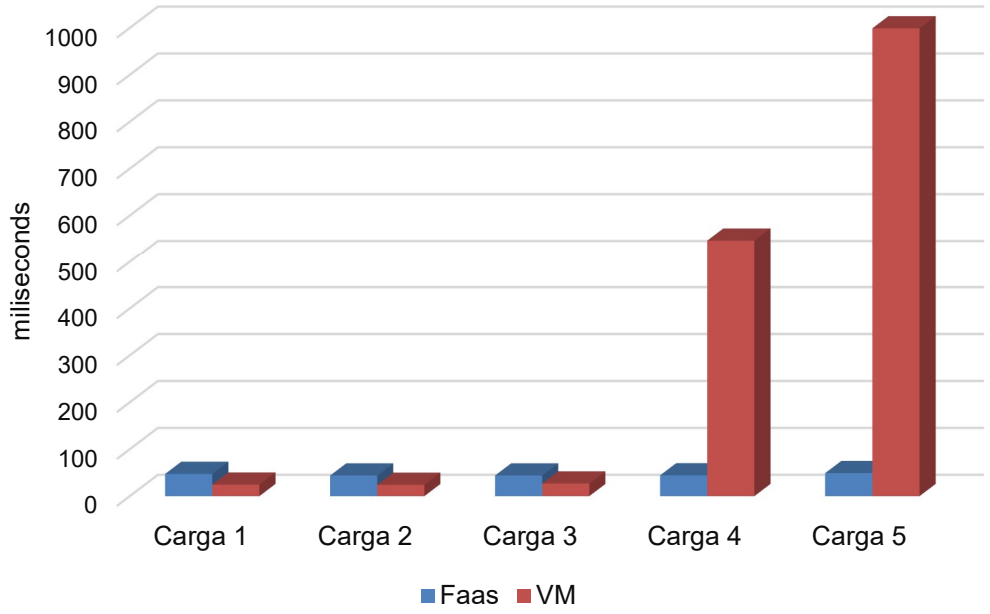


Com o objetivo de comparar os impactos no custo de computação em nuvem ao adotar o *FaaS* para um *webservice* escrito em Node.js, foram desenvolvidas duas aplicações, uma baseada em máquinas virtuais, e a outra baseada em funções *FaaS*. As aplicações possuem a funcionalidade de armazenamento de dados textuais, sendo possível obter um ou todos os registros. Os dados foram armazenados em um banco de dados *"no-sql"*, e foram realizadas simulações de carga de trabalho de 5 a 1250 requisições por segundo em cada aplicação, a fim de verificar seu comportamento a

diferentes níveis de estresse. Foram coletados os dados de latência e custo total de cada solução em cada carga de trabalho.

Após análise dos dados, tornou-se evidente que o custo com máquinas virtuais é inferior aqueles obtidos com as *FaaS* para aplicações que possuam uma demanda constante. Como as *FaaS* compõem seu preço baseado no total de chamadas realizadas, caso seja constante o número de requisições esperados, o custo de manter máquinas virtuais de capacidade adequada é menor do que utilizar as funções como serviço. Os resultados de custos podem ser observados no **Gráfico 1**.

Gráfico 2 – Latência média por Carga de Trabalho em cada Aplicação



Entretanto, nos cenários de carga de trabalho superiores a mil requisições por segundo, a máquina virtual provisionada neste experimentou mostrou-se incapaz de suportar todo o tráfego. Isso levou a latência média para mais de 1000 milissegundos no cenário de maior carga, enquanto as *FaaS* mantiveram a média de 45 milissegundos em todos os cenários realizados, conforme **Gráfico 2**.

2. Revisão Bibliográfica

Recentes publicações como Villamizar [18] e Eivy [19] abordam o custo de *FaaS* quando comparado a outras arquiteturas. Entretanto, os autores chegam a resultados muito divergentes, com Eivy afirmando que a arquitetura *FaaS* pode ser até 300% mais cara do que provisionamento de máquinas virtuais, enquanto Villamizar destaca a redução de custos de até 77% ao utilizar *FaaS*, quando comparada as aplicações monolíticas.

Em [18], Villamizar implementa três aplicações que contém dois serviços cada, um serviço de escrita e outro de leitura de dados. Uma das aplicações é desenvolvida na arquitetura monolítica, em que a mesma aplicação é responsável por todos os serviços. Aplicações monolíticas são servidas por um único computador, o que torna o *deploy* mais simplificado, porém limita as possibilidades de escalar os recursos computacionais. A segunda aplicação é implementada utilizando microsserviços, em que cada projeto é responsável por um único serviço da aplicação. Aplicações microsserviços podem ter cada serviço executando em um computador diferente, otimizando os custos à medida que a demanda de certos serviços aumenta, eliminando a necessidade de escalar a aplicação como um todo. A terceira aplicação utiliza a arquitetura *FaaS*, implementada usando a *FaaS AWS Lambda*. Villamizar simula o uso das aplicações fazendo requisições HTTP nos serviços de leitura e escrita das aplicações. [18]

Em seus resultados, Villamizar concluiu que uma aplicação *serverless* utilizando o serviço *AWS Lambda* para atender 450 requisições *por minuto* pode ter seu custo até 77% menor quando comparado a uma aplicação desenvolvida em arquitetura monolítica para atender a mesma demanda, e 20% menor do que microsserviços. Villamizar não cita a configuração de memória RAM utilizada pelas suas funções, dificultando replicar seus resultados uma vez que o custo está diretamente relacionado ao Gigabytes/Segundos utilizado, além do total de execuções. [18]

É importante ressaltar que Villamizar usa a medida de requisições *por minuto*, assumindo que o tráfego é distribuído constantemente neste período. Nem sempre

este cenário se replica em aplicações reais, pois a carga da aplicação pode variar drasticamente em um minuto, conforme ilustra **Figura 1**. Nesta figura, são exemplificadas duas distribuições, denominadas *P1* e *P2*, que resultam no mesmo total de requisições no período de 60 segundos, mas com comportamentos diferentes. *P2* é a exemplificação de uma distribuição constante de requisições por segundo, conforme assumida por Villamizar em sua simulação. Já *P1* representa um comportamento de aumento e queda repentinos nas requisições *por segundo*, porém chegando ao mesmo número de requisições *por minuto* de *P2*. Caso uma aplicação seja desenhada para suportar um número de requisições *por minuto* e o comportamento do tráfego não seja constante, mas varia conforme *P2,* a aplicação estará sujeita a períodos de instabilidade, portanto a medida requisições *por segundo* é a mais utilizada em simulação de carga.

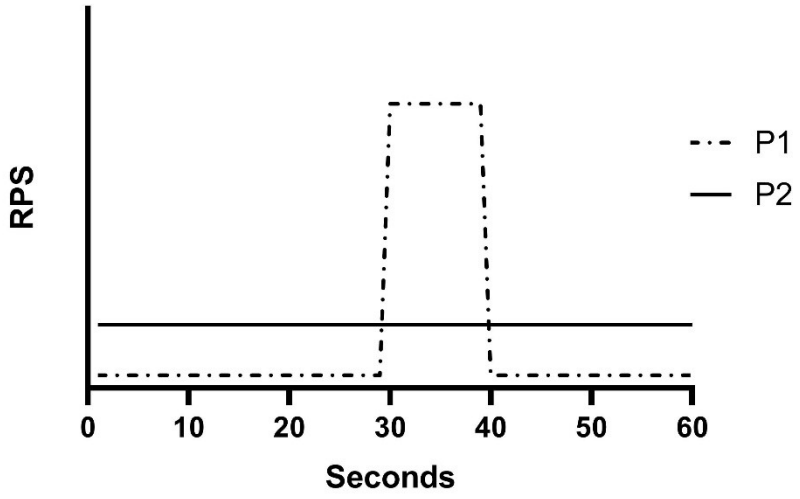


Figura 1 – 60 mil requisições por minuto com comportamentos diferentes

Eivy alerta para a complexidade de estimar os custos de infraestrutura para soluções envolvendo a arquitetura *FaaS*, exemplificando uma aplicação que receba 150 requisições por segundo, com cada função executando em 100ms com 128 Megabytes de memória RAM, a *FaaS* possui custo mensal pouco menor do que provisionar máquinas virtuais para atender a mesma demanda. Porém caso a demanda da aplicação cresça para 30 mil requisições por segundo, adotar a arquitetura *FaaS* pode ser até três vezes mais caro, quando comparado às máquinas virtuais. Eivy não usa simulação de uso das aplicações, mas calcula analiticamente

com base em dados de aplicações reais, porém, não indica quais máquinas virtuais levou em conta para chegar neste resultado. [19]

A divergência na conclusão dos autores se dá provavelmente pela diferente escala de suas operações. Villamizar [18] fez seu experimento com 450 requisições *por minuto*, que se distribuídas constantemente conforme é citado em seu trabalho, resultam em 450 / 60 = 7 requisições *por segundo*. Esta é uma demanda muito inferior à calculada por Eivy [19], ao afirmar que os custos com *FaaS* são três vezes superiores as máquinas virtuais ao utilizar 30 mil requisições *por segundo*.

Existem cenários de uso onde uma ação que leva alguns milissegundos para se concluir deve ser executada em dado intervalo de tempo, por exemplo a cada cinco minutos. Se utilizando uma infraestrutura tradicional, uma máquina virtual deverá ser provisionada para atender a demanda deste acionamento, e em contrapartida ficará a maior parte do tempo ociosa. Este cenário é explorado por Gojko [17] em seus experimentos. Ele conclui que uma função *AWS Lambda* que é acionada a cada cinco minutos, com latência média de 200 milissegundos e 512 Megabytes de memória RAM é até 99.8% mais barata do que provisionar uma máquina virtual *AWS EC2* com 512 Megabytes de memória RAM cujo custo é U$0.0059 por hora. Caso seja levado em conta uma réplica para a máquina virtual para aumentar a resiliência do serviço, seria então 99.95% mais barato utilizar o *AWS Lambda*, que já conta com estratégias de replicação que são de responsabilidade do *cloud provider*.

Neste trabalho não foram considerados os cenários de uso intermitente conforme citado por Gojko [17], mas foram abordados os impactos de substituir o *back-end* de uma aplicação *web* que receba cargas regulares de trabalho por uma solução *FaaS*.

Como mencionado por Eivy [19], além de infraestrutura, a adoção da arquitetura *FaaS* também proporciona impactos nos custos de pessoal especializado em administração de sistemas *(sysadmins)*, uma vez que qualquer falha, atualização de segurança, redundância e disponibilidade são de responsabilidade do provedor da *FaaS*. Portanto não serão considerados os custos com *sysadmins*, focando o texto especificamente em infraestrutura.

Ao implementar uma aplicação real utilizando *FaaS*, os custos não se limitam aos de acionamento e execução das funções, pois os *cloud providers* cobram também pelos dados trafegados na rede, armazenamento do código fonte das funções, *AWS API Gateway* para funções cujo mecanismo de acionamento são requisições HTTP

[7], custos que não incidem na utilização de máquinas virtuais para infraestrutura da aplicação.

3. Metodologia

Este trabalho contou com o desenvolvimento de duas aplicações, uma baseada em funções *FaaS* e outra em máquinas virtuais. Foram realizadas simulações de carga com diferentes quantidades de requisições por segundo em cada aplicação, a fim de identificar seu comportamento.

Para realização deste experimento, foi necessário um microcomputador de uso geral, conexão à Internet banda larga e uma conta em serviço de computação em nuvem que possuísse os serviços de *FaaS*, provisionamento de máquinas virtuais e banco de dados *no-sql*. Foram desenvolvidas duas aplicações, µS1 e µS2, que se comunicam por *webservices*, e ambas implementam as funcionalidades de listar um registro e listar todos os registros, uma aplicação baseada em máquinas virtuais, µS1, e a outra µS2 em arquitetura *FaaS*.

As aplicações foram submetidas a simulação de uso com diferentes quantidades de requisições por segundo. Com o resultado da simulação, foram apurados os custos gerados para cada cenário ao utilizar a arquitetura *FaaS* em comparação a infraestrutura tradicional, assim como a latência de cada aplicação a fim de observar seu comportamento.

3.1. Escolha de Ferramentas

O *cloud provider* utilizado foi o Amazon Web Services, por ser o primeiro a distribuir para uso público o modelo de *Function-as-a-Service* com *AWS Lambda*, e *Infrastructure-as-a-Service* com *AWS EC2*. A conta utilizada na simulação possuia mais de doze meses após sua criação, portanto, não elegível a qualquer crédito de *free-tier* oferecido pela AWS.

Node.js foi escolhido como plataforma de execução das aplicações por não bloquear *I/O* quando executando operações como acesso a base de dados e escrita de arquivos, uma vez que a maior demanda das aplicações será em entrada e saída (input/output) [20]. Esta característica do Node.js permite que sejam desenvolvidas aplicações capazes de atender grande demanda de usuários. *Websites* como *twitter.com*, *bbc.com* e *aliexpress.com*, rodam Node.js em seus servidores e juntos

atendem mais de 5 bilhões de visitas mensais [21]. A versão do Node.js utilizada em ambas as aplicações foi 8.10, por ser a versão mais atual disponível para uso no serviço *AWS Lambda* no momento da experimentação.

A aplicação µS1 foi hospedada em uma máquina virtual do serviço *AWS EC2* do tipo *t2.large* [4], que possui 8 Gigabytes de memória RAM e duas CPUs. Esta aplicação é executada em uma máquina com sistema operacional Linux, em uma variante desenvolvida pela *Amazon Web Services*. O motor de execução do Node.js é *single-thread*, mas foi utilizado o gerenciador de processos *PM2* [22] para obter duas réplicas e assim usar as duas CPUs disponíveis na instância *t2.large*. O balanceamento de carga entre as réplicas realizado pelo *Node Cluster* [23].

A sintaxe utilizada para o desenvolvimento da aplicação foi o Typescript[9], um superconjunto de Javascript que adiciona tipagem para a linguagem. Como Node.js não executa nativamente código Typescript, é realizada a transformação para Javascript no momento de compilação pela ferramenta *ts-node*[10]. Nesta aplicação foi utilizada a biblioteca *Express* [24] para descrever os pontos de acesso HTTP.

A aplicação *FaaS* foi desenvolvida na linguagem de programação Javascript, e é executada em um container que possui a plataforma Node.js. Para abstrair o *deploy* e atualização da função, foi utilizado o *framework Serverless* [16], que automatiza este processo. Como a aplicação *FaaS* teve seus pontos de acesso disponíveis por requisições HTTP, foi necessário a utilização do *AWS API Gateway*, onde são definidos os pontos de acesso de cada função. A configuração do *AWS API Gateway* é abstraída pelo *framework Serverless,* que interpreta as configurações e as aplica no *cloud provider*. Para execução em ambiente de desenvolvimento, foi instalado o *plugin* do *Serverless Framework* chamado *serverless-offline*[11]. Este plugin permite a emulação do ambiente *AWS Lambda*, e portanto, facilita o desenvolvimento, pois elimina a necessidade de fazer o *deploy* para testar a função.

Para armazenar os dados retornados pelas aplicações, foi utilizado o serviço de banco de dados *no-sql AWS DynamoDB*. Este serviço foi escolhido por ser considerado um banco de dados altamente escalável [25], portanto exercendo pouco impacto negativo no desempenho das aplicações. No desenvolvimento das aplicações, foi utilizado um emulador do *DynamoDB*, chamado *DynamoDB Local* [26],

[9] https://www.typescriptlang.org/
[10] https://github.com/TypeStrong/ts-node
[11] https://github.com/dherault/serverless-offline

disponibilizado pela *AWS* para permitir a integração com o banco de dados em ambiente local. A *Amazon Web Services* disponibiliza um *SDK* [27] para desenvolvimento Javascript, que contém funções utilizadas para acessar o *DynamoDB* e realizar as operações de leitura e escrita, e este foi utilizado em ambos os projetos.

A ferramenta *loadtest* [28] foi empregada na simulação de carga das aplicações. Antes da escolha da ferramenta *loadtest,* foram realizados testes com a ferramenta *locust* [29]. Porém, esta falhava ao atingir 50 requisições por segundo, número muito abaixo do total esperado para os testes. Este comportamento foi contornado na ferramenta *loadtest* ao utilizar o parâmetro *keep-alive* [30] de requisições HTTP na simulação de carga, fazendo com que as conexões sejam reutilizadas e não seja aberta uma nova conexão a cada requisição realizada.

Para realização da simulação de carga, foi utilizado um computador executando sistema operacional baseado no kernel GNU/Linux, com CPU Quad core Intel Core i7-7700HQ e 16 Gigabytes de memória RAM.

Para obter os dados e analisar os custos de cada plataforma, foi utilizado o serviço *AWS Cost Explorer*, que possibilita detalhar por dia cada serviço utilizado, e o custo gerado.

Portanto, as ferramentas utilizadas na aplicação µS1 hospedada em máquinas virtuais foram:

- *Node.js 8.10*
- *AWS EC2 t2.large*
- *PM2 - Advanced Node.js process manager*
- *Express.js*
- *Typescript* e *ts-node*
- *AWS SDK for JavaScript in Node.js*
- *AWS DynamoDB*
- *DynamoDB Local*

O desenvolvimento e execução da aplicação µS2 *FaaS* contou com as ferramentas:

- *Serverless Framework*
- *Serverless Offline*
- *Javascript*

- *AWS Lambda*
- *AWS API Gateway*
- *AWS SDK for JavaScript in Node.js*
- *AWS DynamoDB*
- *DynamoDB Local*

Ferramenta utilizada para as simulações de carga de trabalho:

- *loadtest*

3.2. Procedimentos

Os procedimentos para a realização do experimento foram divididos em duas etapas: 1) desenvolvimento das aplicações, e 2) simulação de uso com diferentes cargas de trabalho.

3.2.1. Desenvolvimento

Foram desenvolvidas duas aplicações, sendo a primeira aplicação denominada µS1 baseada em máquinas virtuais e a outra aplicação baseada em funções *FaaS,* denominada µS2. Ambas as aplicações possuem as mesmas funcionalidades e os mesmos pontos de acesso, sendo acessados por requisições HTTP conforme descrito no **Quadro 1**.

No desenvolvimento da aplicação µS1, foram definidos os pontos de acesso HTTP utilizando a biblioteca *Express*. Foram definidas duas camadas na arquitetura do projeto, *controller* e *repository.* O *controller* foi responsável pelas validações de entrada das funções, assim como tratamento de erros retornados pela camada de dados, o *repository*. No *repository* foram definidos métodos para obtenção de um registro do banco de dados com base em uma chave de identificação, obtenção de todos os registros do banco de dados, além de inserção e atualização de registros. Não foram testadas as funcionalidades de inserção e atualização de dados, somente

a obtenção de um ou mais registros. Foram usados mecanismos de leitura *eventualmente consistente*[12] da *API* do *DynamoDB* [27].

No arquivo *package.json*, utilizado para gerenciar dependências e tarefas de uma aplicação baseada no *node package manager (NPM)*, foram definidas as tarefas *build, dev* e *prod.*

Para iniciar o *DynamoDB* localmente, portanto não dependendo de conexão à Internet para seu funcionamento, deve ser executado o *jar*[13] disponibilizado pela *AWS*, com o comando abaixo:

```
java -jar DynamoDBLocal.jar
```

Para iniciar a aplicação em modo de desenvolvimento, deve ser executado o comando:

```
npm run dev
```

Ao executar esta tarefa, a aplicação será iniciada em ambiente de desenvolvimento, ouvindo requisições na porta *3000*, com *live-reloading*[14] habilitado para detectar mudanças no código fonte, e conexão ao *DynamoDB* local.

Para realizar a *transpilação e minificação*[15] do código fonte foi utilizado o comando:

```
npm run build
```

Este comando transforma todos os arquivos de extensão *Typescript* para gerar um arquivo *bundle* chamado *entry.js* em Javascript, sintaxe executável pelo Node.js. O comando abaixo executa o mesmo processo de *build,* e inicia a aplicação executando o arquivo *entry.js* em modo *single-thread*:

---

[12] No DynamoDB, leitura eventualmente consistente não garante que sejam retornados os dados mais atualizados deste registro em caso de alteração concorrente à leitura

[13] https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/DynamoDBLocal.DownloadingAndRunning.html

[14] Ao realizar alterações no código fonte e salvar o arquivo, a aplicação é recarregada automaticamente com o código alterado

[15] O código Typescript não é executado nativamente pelo Node.js sendo necessário sua transformação para Javascript, e este processo é chamado Transpilação

```
npm run prod
```

Após realizado o *build,* o arquivo *entry.js* foi enviado por *SSH* para a máquina virtual. A inicialização da aplicação em modo *produção* é feita utilizando o gerenciador de processos *PM2*, configurado com 2 réplicas da aplicação, utilizando o comando:

```
pm2 start entry.js -i 2
```

A distribuição de carga de trabalho entre as duas instâncias da aplicação é realizada pelo *PM2* utilizando o *Node Cluster* [23], do próprio Node.js. A arquitetura da aplicação µS1 foi exemplificado na **Figura 2**. Para permitir conexões na porta 3000, porta que a aplicação está escutando, e porta 22 para conexão *SSH,* foi necessário alterar as regras de entrada (*inbound rules)* do grupo de segurança do *AWS EC2*. O código fonte da aplicação com suas instruções para execução está disponível em https://github.com/leonardoviveiros/microservice.

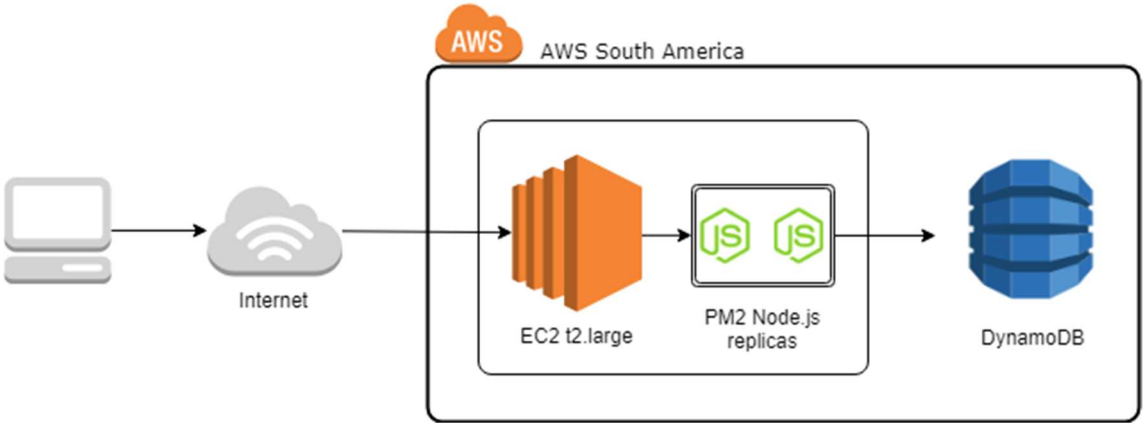


Figura 2 – Diagrama de Arquitetura da Aplicação µS1

A aplicação *FaaS*, µS2, foi desenvolvida na linguagem de programação Javascript, e é executada na plataforma Node.js. As aplicações *FaaS* se resumem a funções que juntas compõem os serviços oferecidos pela aplicação, que são acionadas por eventos. São assuntos deste trabalho as funções denominadas *getOne* e *getAll*, para obtenção de um ou todos os registros respectivamente, conforme **Figura 3**. A arquitetura da aplicação foi dividida em duas camadas, sendo elas *controller* e *repository.* O *controller* é responsável por publicar as funções *AWS Lambda*, aplicando validações para as requisições recebidas. O *repository* foi implementado contendo

funções para listagem de um e todos os registros, além de inserção de novos registros na base de dados *DynamoDB*.

Para iniciar o *DynamoDB* localmente deve ser executado o *jar* disponibilizado pela *AWS*, com o comando abaixo:

```
java -jar DynamoDBLocal.jar
```

Para executar a aplicação localmente em ambiente de desenvolvimento, conectando-se ao *DynamoDB Local*, foi utilizado o comando:

```
serverless offline start
```

As definições de eventos que acionam as funções são realizadas no arquivo *serverless.yml.* Após finalizado o desenvolvimento e todas as definições das funções, foi realizado o *deploy* com o comando:

```
serverless deploy
```

Assim o S*erverless Framework* se encarrega de aplicar as configurações nos respectivos serviços do *cloud provider,* como criar ou atualizar as *AWS Lambda* com o código escrito*,* criar regras de acesso no *AWS API Gateway*, criar tabelas no *DynamoDB*, entre outros. É importante ressaltar que para o *Serverless Framework* conseguir se autenticar com o *cloud provider* devem ser configurados os acessos e permissões utilizando algum serviço de gerenciamento de acesso, como o *AWS Identity and Access Management (AWS IAM)*. O código fonte das funções que consistem na aplicação µS2 está disponível em https://github.com/leonardoviveiros/lambda-architecture.

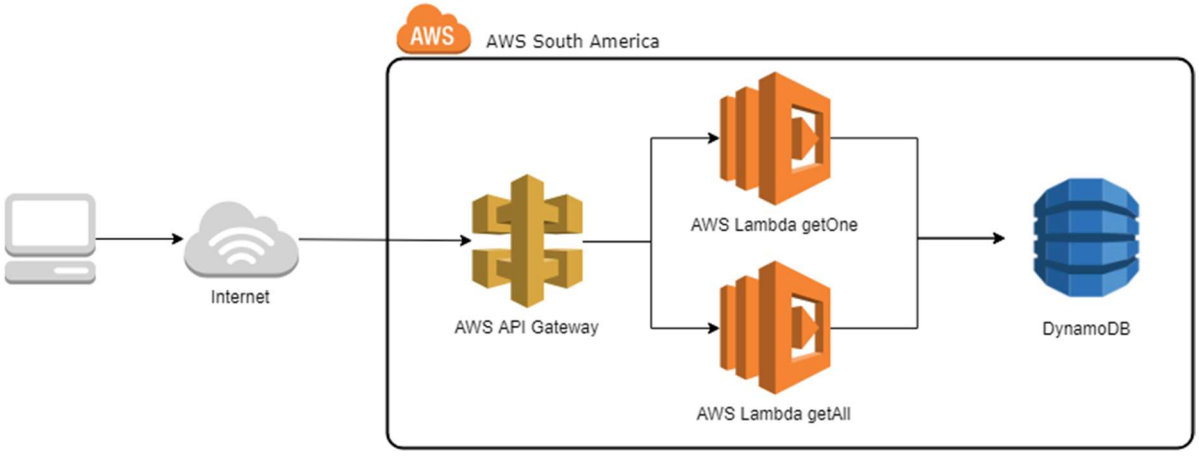


Figura 3 – Diagrama de Arquitetura da Aplicação µS2

3.2.2. Simulação de Carga

A metodologia de testes consiste em realizar requisições HTTP nos pontos de acesso descritos no **Quadro 1**. Foram desenhadas as cargas de trabalho distribuídas conforme **Tabela 1**, iniciando em cinco requisições *por segundo* na *Carga 1*, e indo até 1250 requisições *por segundo* na *Carga 5*. Os dados trafegados foram textos com tamanho de 2048 bytes. Do total de requisições *por segundo* de cada cenário de carga de trabalho, 20% das requisições foram direcionadas para o ponto de acesso que retorna todos os registros da base de dados, e 80% foram direcionadas para o ponto de acesso que retorna um registro. Das requisições à função que retorna um registro, foram divididas entre 20% em *GetOne1*, 20% em *GetOne2*, 20% em *GetOne3* e 20% em *GetOne4*, onde cada uma destas funções retornam um registro diferente.

A ferramenta utilizada para os testes foi *loadtest,* utilizada por linha de comando, com os parâmetros:

```
loadtest {URL} -k --rps 1000 -t 3600
```

Estes parâmetros indicam: *-k* reaproveitar conexões abertas, usando o agente de conexão Keep-alive [30]; *--rps 1000* indica realização de mil requisições por segundo, e *-t 3600*, para a simulação ser executada por uma hora. Além disso, é necessário informar a *URL* que as requisições serão apontadas. Para cada cenário de carga de trabalho, foram realizadas requisições por uma hora, a fim de gerar custos

reais na *Amazon Web Services* e ser possível detectar o comportamento das aplicações as diferentes cargas de trabalho.

Tabela 1 – Cargas de trabalho e requisições por segundo

Cargas de Trabalho	Requisições *por segundo*
Carga 1	5
Carga 2	80
Carga 3	500
Carga 4	1000
Carga 5	1250

Quadro 1 – Pontos de Acesso das Aplicações

Descrição do *webservice*	Método HTTP	Ponto de Acesso
Obtém um registro pelo identificador	GET	/{id}
Obtém todos os registros existentes	GET	/

Para a execução dos cenários em sequência, foi escrito um *shell script* que executa cada cenário por uma hora, escrevendo os resultados em arquivo. Este procedimento foi realizado para a simulação nas aplicações µS1 e µS2.

```
./loadtest_faas_carga_1 && \
  loadtest_faas_carga_2 && \
  loadtest_faas_carga_3 && \
  loadtest_faas_carga_4 && \
  loadtest_faas_carga_5
```

Cada chamada é semelhante ao exemplo abaixo, diferenciando principalmente a URL e o número de requisições por segundo, assim como o nome do arquivo que foram gravados os resultados:

```
loadtest http://52.67.6.51:3000/ -t 3600 -k --rps 200 >> ec2_getall_1h_200rps
```

Como é possível observar, os cenários foram executados em seguida, do menos para o mais custoso. Uma vez que o banco de dados foi configurado de modo *on-demand*, sua performance escala conforme as operações realizadas. Começar a simulação do cenário de menor carga de trabalho permitiu que o provisionamento de recursos feito pelo *autoscaling* do banco de dados se adapte à demanda recebida. Portanto, foram minimizados os atrasos por parte do banco de dados.

A simulação realizada não representou fielmente o tráfego que uma aplicação real receberia, pois seriam recebidas solicitações de diferentes dispositivos, espalhados geograficamente. A simulação foi feita de uma só rede, portanto sujeita a congestionamento de tráfego nos nós que a conexão à Internet possa possuir da máquina cliente até o servidor da *Amazon Web Services*.

Em aplicações reais seriam realizadas operações de criação, edição e deleção de registros e não somente recuperação como utilizadas nesta simulação. Também é comum ser necessário algum processamento de dados que faça uso intensivo de *CPU*, além do armazenamento, e este cenário não foi refletido na simulação.

4. Avaliação dos Custos

Para o cálculo de custo foram desconsiderados os custos do banco de dados, armazenamento de *logs*, custo de *bytes* trafegados para fora da rede *Amazon Web Services* entre outros eventuais custos menores.

Na aplicação de máquinas virtuais, µS1, o único custo considerado foi o preço *por hora* de utilização da instância *AWS EC2* do tipo *t2.large*. Para a aplicação µS2 foram considerados os custos dos serviços *AWS Lambda* e *AWS API Gateway*.

Uma vez que cada cenário foi executado durante 60 minutos, o custo gerado pela máquina virtual foi U$ 0,1488 em cada carga de trabalho, independentemente da quantidade de requisições atendida, resultando no custo total de U$ 0,744

Diferente das máquinas virtuais, os custos de aplicações *FaaS* escalam à medida que também escala sua demanda. Na **Tabela 2** é possível observar os custos *por hora* de operação em cada cenário da aplicação *FaaS*. Na carga 1 os custos em *FaaS* são 30% inferiores aqueles obtidos com a aplicação baseada em máquinas virtuais. Porém no cenário de carga 5, os custos podem ser mais de 150 vezes superiores do que o custo por hora da instância *AWS EC2* utilizada na aplicação µS1.

Tabela 2 – Custo por Carga de Trabalho da Aplicação µS2

	Carga 1 ***(U$)***	**Carga 2** ***(U$)***	**Carga 3** ***(U$)***	**Carga 4** ***(U$)***	**Carga 5** ***(U$)***
AWS Lambda	0,03	0,48	3	6	7,5
AWS API Gateway	0,075	1,07	6,66	13,32	16,65
Total	0,105	1,55	9,66	19,32	24,15

Estes dados podem fazer parecer que aplicações *FaaS* resultam em custos muito superiores do que provisionar instâncias de máquinas virtuais para o *backend* de aplicações. Entretanto, a simulação foi feita com cargas de trabalho constantes. Em aplicações reais, caso o *workload* recebido pela aplicação seja conhecido, o custo sempre será menor em utilizar máquinas virtuais de capacidade adequada do que as *FaaS*.

Entretanto, ao provisionar máquinas virtuais, a escalabilidade de sua aplicação não é automatizada e pode ficar comprometida. Mesmo com serviços de *autoscaling* para máquinas virtuais sendo uma realidade, estes dependem de

configurações complexas e não reagem ao aumento da demanda em tempo real como as *FaaS* conseguem fazer.

A maior parte do custo gerado pela aplicação µS2 é proveniente do serviço *AWS API Gateway*. O faturamento deste serviço se baseia no número de chamadas e no volume de dados trafegados pela rede. Até a data do desenvolvimento deste experimento, este serviço era a única maneira de usar gatilhos HTTP para funções *AWS Lambda*, portanto todas as aplicações *FaaS* hospedadas na *AWS* precisariam utilizá-lo. Entretanto, recentemente, foi anunciada a possiblidade de publicar as funções *AWS Lambda* pelo *AWS Application Load,* serviço que possui um preço muito inferior ao *AWS API Gateway,* eliminando a necessidade deste serviço para expor as funções na web [31]. Esta alternativa com certeza irá reduzir significativamente os custos de aplicações *FaaS* hospedadas na *Amazon Web Services*.

O comportamento das aplicações em diferentes cargas de trabalho resultou em latências esperadas. A função *GetAll*, responsável por retornar todos os dados do banco de dados, foi a função que levou mais tempo para executar quando comparada às outras funções da mesma carga de trabalho. Este comportamento era esperado, devido a esta função retornar o mesmo conteúdo das outras quatro funções juntas. Portanto, além do tempo de execução da função *GetAll*, também contribuiu com a latência o tempo de download dos dados, por serem ligeiramente maiores. As funções *GetOne* retornavam 2048 bytes cada por chamada, já a função *GetAll* retornava 8192 bytes a cada chamada que é o tamanho dos quatro registros da base de dados.

O custo da aplicação µS2 se mostrou muito superior à instância de máquina virtual, quando operada em cargas de trabalho maiores. Entretanto, a latência observada na aplicação µS1 foi muito superior à latência da aplicação µS2, quando realizando a simulação nas cargas de trabalho 4 e 5, com 1000 e 1250 requisições *por segundo* respectivamente, conforme pode ser observado na **Tabela 3**.

As duas *CPUs* disponíveis chegaram em 100% de uso nestes cenários, ou seja, foi atingido o limite que esta máquina virtual é capaz de atender. A latência média chegou a meio segundo no cenário de carga 4, e a cerca de um segundo no cenário de carga 5. Ao demorar um segundo para dar *feedback* às ações dos usuários, é percebida muita lentidão na utilização da aplicação e uma latência de um segundo ou mais atrapalha a experiência dos usuários, podendo levar a desistirem de usar a aplicação [32].

Tabela 3 – Latência Média por Carga de Trabalho da Aplicação μS1

Ponto de acesso	Carga 1 *(ms)*	Carga 2 *(ms)*	Carga 3 *(ms)*	Carga 4 *(ms)*	Carga 5 *(ms)*
GetAll	28,90	27,90	33,20	557,50	1096,30
GetOne1	23,70	23,90	25,50	545,20	949,30
GetOne2	21,40	21,90	25,40	546,60	988,70
GetOne3	24,80	22,20	24,50	541,10	957,50
GetOne4	24,20	24,40	25,20	542,40	1004,10
Média	24,60	24,06	26,76	546,56	999,18

Para serviços que reagem às ações dos usuários, é ideal minimizar a latência para não prejudicar a experiência do usuário. Nesta situação, fica evidente que usar uma máquina virtual com maior capacidade resolveria o problema da latência demasiada grande. Uma opção seria usar uma máquina virtual do tipo *AWS EC2 t2.xlarge* com 4 *CPUs* e 16 *Gigabytes* de memória RAM para aumentar o número de requisições por segundo atendidas sem a ocorrência de grandes latências.

Na aplicação μS2 baseada em *FaaS*, é possível observar dois comportamentos com base na latência média por carga de trabalho disponibilizada na **Tabela 4**.

A aplicação apresenta latência média nas cargas de trabalho 1 a 3 de 45 milissegundos face aos 25 milissegundos da aplicação μS1. A diferença de latência provavelmente se dá pelo fato de as funções estarem sendo executadas em contêineres, e não diretamente no Sistema Operacional. Por estar em um nível de abstração maior, há, portanto, *delay* na inicialização do container, execução do código e retorno dos dados. Assim como o *Coldstart* referido anteriormente, que aumenta consideravelmente o tempo de resposta para a primeira execução.

Tabela 4 – Latência Média por Carga de Trabalho da Aplicação μS2

Ponto de acesso	Carga 1 *(ms)*	Carga 2 *(ms)*	Carga 3 *(ms)*	Carga 4 *(ms)*	Carga 5 *(ms)*
GetAll	55,30	48,30	53,50	46,90	54,30
GetOne1	45,20	42,90	39,80	42,60	51,20
GetOne2	46,10	42,90	40,00	40,80	42,30
GetOne3	45,30	42,80	46,40	40,40	47,40
GetOne4	45,00	42,50	40,30	40,70	50,10
Média	47,38	43,88	44,00	42,28	49,06

Porém, o *delay* de 45 milissegundos em média, mesmo sendo quase duas vezes maior que a mesma latência média dos mesmos cenários da aplicação µS1, é muito baixo. Portanto é impossível para usuários finais da aplicação notarem diferença entre respostas que levam 25 ou 45 milissegundos.

Entretanto, para as cargas de trabalho 4 e 5, a latência apresentada pela aplicação µS2 é muito inferior do que aquelas da aplicação µS1 baseada em máquinas virtuais, mantendo a média de 45 milissegundos, mostrando que a escalabilidade das funções se aplica automaticamente e independentemente da carga de trabalho utilizada, enquanto a aplicação µS1 sofre com latências de 550 a 1000 milissegundos nestes mesmos cenários.

Quando comparamos as duas aplicações no cenário de carga de trabalho 3, onde ambas responderam em tempo médio similar, é possível identificar que o custo resultante em *FaaS* é muito superior do que aqueles observados na aplicação µS1.

5. Conclusões

Foram desenvolvidas duas aplicações, com a primeira baseada em máquinas virtuais, e a segunda em funções como serviço. Ao realizar simulações de uso em cada aplicação, foi possível identificar a característica de custos de cada arquitetura. Também foram coletados os dados de latência de cada função das aplicações, de modo a identificar variações de disponibilidade para cada carga de uso. Com isso, tornou-se evidente que as aplicações baseadas em funções como serviço possuem custo absoluto maior do que aquelas baseadas em máquinas virtuais no cenário testado. Porém, aplicações baseadas em *FaaS* mostraram-se estáveis em qualquer carga de trabalho aplicada no teste.

Aplicações que recebam tráfego imprevisível, ou que a demanda varia muito, podem se beneficiar do uso das *FaaS*, uma vez que os recursos disponíveis serão ditados pela demanda das funções em tempo real, evitando provisionar recursos superiores ao necessário, e também mitigando falhas ou lentidões na aplicação por falta de recursos, caso a demanda ultrapasse o poder computacional provisionado.

Também são ótimos candidatos às *FaaS* serviços que podem ser executados de maneira assíncrona. Uma tarefa que seja executada em determinado intervalo de tempo, precisaria de uma máquina virtual com agendamento de sua execução, entretanto este é um cenário ideal para o uso de funções como serviço, pois só seria gerado custo quando a tarefa executar.

As *FaaS* permitem alocar grande poder computacional quase que instantaneamente, e podem ser paralelizadas praticamente infinitamente, portanto tarefas que façam uso intensivo de CPU como redimensionamento de imagens, cálculo de rotas, edição de vídeo, criptografia, descriptografia e compressão de arquivos são mais adequadas para *FaaS.* Já tarefas que possuem principal latência em entrada/saída *(I/O)* não são adequadas pois neste caso estaria sendo pago pelo tempo de CPU que estaria ociosa esperando alguma entrada de dado.

Conforme é possível identificar, utilizar as máquinas virtuais para servir aplicações que possuem demandas conhecidas resultam em custos menores do que atender a mesma demanda com *FaaS* nos cenários testados. Entretanto a garantia de resiliência, backup, atualizações de segurança, balanceamento de carga, provisionamento de mais recursos e outras tarefas de administração de sistemas se

tornam responsabilidade dos desenvolvedores da aplicação. Empresas que possuem time de *DevOps* e *SysAdmins* podem obter menores custos ao utilizarem máquinas virtuais, pois já possuem mão de obra capaz de realizar estas tarefas.

Entretanto, em aplicações cuja demanda de tráfego recebida varie bruscamente, as aplicações *FaaS* se apresentam como solução para manter a latência média aceitável em qualquer escala, sem intervenção por parte dos desenvolvedores. Isto possibilita a redução ou até extinção de departamento focado em infraestrutura, dando aos desenvolvedores as chances de criarem as aplicações, fazer o *deploy* e deixar toda manutenção de infraestrutura nas mãos do *cloud provider*.

Isto possibilita *startups* com times enxutos criarem aplicações globais em pouco tempo, reduzindo o *time to market* pela escala estar sendo gerenciada pelo *cloud provider*.

Todavia, ao desenvolver soluções *FaaS* baseada nos serviços da *Amazon Web Services*, boa parte das regras de negócio serão associadas com este fornecedor. Isto é chamado de *vendor lock-in.* Em um cenário em que mais e mais partes dos produtos de uma empresa são baseados em *FaaS*, o *vendor lock-in* se torna um fator preocupante. No futuro, mudar de *cloud provider* envolve repensar todas as soluções desenvolvidas, procurando alternativas em outros fornecedores existentes. Por se tratar de uma tecnologia nova, ainda não há padrões definidos entre os fornecedores de *FaaS*. O *Serverless Framework* auxilia na distinção entre o código e a infraestrutura. Porém, os gatilhos de função existentes em um *cloud provider* podem não existir em outro.

Acredita-se que em um cenário onde existam pontos de acesso que disparam tarefas de processamento de dados, a aplicação baseada em máquinas virtuais atingiria seu limite muito antes das 1000 requisições *por segundo*, com pouca ou nenhuma alteração no resultado da aplicação baseada em *FaaS.*

Como foram disponibilizadas alternativas ao serviço *AWS API Gateway*, que compõe cerca de 68% do custo observados na aplicação µS2, é possível que em aplicações futuras *FaaS* seja possível reduzir os custos com este tipo de serviço.

Bibliografia

[1] M. Breeeding, “The Systems Librarian: The Advance of Computing From the Ground to the Cloud,” Vanderbilt University Libraries , November 2009. [Online]. Available: http://www.infotoday.com/cilmag/nov09/Breeding.shtml. [Acesso em 12 Abril 2019].

[2] M. A. Islam, H. Mahmud, S. Ren e X. Wang, “Paying to save: Reducing cost of colocation data center via rewards,” em *IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, Burlingame, CA, USA, 2015.

[3] VMWare, “vmware_infrastructure_wp.pdf,” VMWare, Inc., 15 Dezembro 2018. [Online]. Available: https://www.vmware.com/pdf/vmware_infrastructure_wp.pdf. [Acesso em 20 Abril 2019].

[4] “Amazon Elastic Compute Cloud,” Amazon, 10 Fevereiro 2018. [Online]. Available: http://aws.amazon.com/ec2/. [Acesso em 10 Fevereiro 2018].

[5] S. J. Berman, L. Kesterson-Townes, A. Marshall e R. Srivathsa, “How cloud computing enables process and business,” *Strategy & Leadership,* vol. 40, pp. 27-35, 2012.

[6] M. Fowler, “Serverless Architectures,” 22 Maio 2018. [Online]. Available: https://martinfowler.com/articles/serverless.html. [Acesso em 14 Outubro 2018].

[7] Amazon Web Services, “Amazon Lambda - AWS,” Amazon Web Services, 2019. [Online]. Available: https://aws.amazon.com/pt/lambda/. [Acesso em 12 Janeiro 2019].

[8] Google Cloud, “Cloud Functions - Event-driven Serverless Computing | Cloud Functions,” Google, Abril 2019. [Online]. Available: https://cloud.google.com/functions/. [Acesso em 23 Abril 2019].

[9] Microsoft Azure Functions, “Azure Functions—Develop Faster With Serverless Compute | Microsoft Azure,” Microsoft, 2019. [Online]. Available: https://azure.microsoft.com/en-us/services/functions/. [Acesso em 23 Abril 2019].

[10] IBM Cloud, “IBM Cloud Functions,” IBM, 2019. [Online]. Available: https://console.bluemix.net/openwhisk/. [Acesso em 23 Abril 2019].

[11] A. Pérez, G. Moltó, M. Caballer e A. Calatrava, “Serverless computing for container-based architectures,” *Future Generation Computer Systems,* vol. 83, pp. 50-59, 2018.

[12] Amazon Web Services, “Alexa for Business - empower your organization with Alexa,” Amazon Web Services, [Online]. Available: https://aws.amazon.com/alexaforbusiness/. [Acesso em 10 Abril 2019].

[13] Amazon Web Services, “Cloud Object Storage | Store & Retrieve Data Anywhere | Amazon Simple Storage Service,” Amazon Web Services, [Online]. Available: https://aws.amazon.com/s3/. [Acesso em 10 Abril 2019].

[14] Amazon Web Services, “Using AWS Lambda with Other Services - AWS Lambda,” Amazon Web Services, 2018. [Online]. Available: https://docs.aws.amazon.com/lambda/latest/dg/lambda-services.html. [Acesso em 03 Abril 2019].

[15] J. Manner, M. Endreß, T. Heckel e G. Wirtz, “Cold Start Influencing Factors in Function as a Service,” em *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*, Zurich, Switzerland, 2018.

[16] Serverless, Inc., “Serverless Framework - Build applications on AWS Lambda, Google CloudFunction, Azure Function, AWS Flourish and more,” Serverless, Inc., 2019. [Online]. Available: https://serverless.com/framework/. [Acesso em 03 Fevereiro 2019].

[17] G. Adzic e R. Chatley, “Serverless Computing: Economic and Architectural Impact,” em *European Software Engineering Conference*, Paderborn, Germany, 2017.

[18] M. Villamizar, “Infrastructure Cost Comparison of Running Web Applications in the Cloud Using AWS Lambda and Monolithic and Microservice Architectures,” em *6th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, Cartagena, Colombia, 2016.

[19] A. Eivy, “Be Wary of the Economics of "Serverless" Cloud Computing,” *IEEE Cloud Computing,* pp. 6-12, 26 Abril 2017.

[20] Node.js, “Node.js,” Node.js Foundation, 2019. [Online]. Available: https://nodejs.org/en/. [Acesso em 12 Janeiro 2019].

[21] SimilarTech, “NodeJS Market Share and Web Usage Statistics,” SimilarTech, 08 Abril 2019. [Online]. Available: https://www.similartech.com/technologies/nodejs. [Acesso em 09 Abril 2019].

[22] PM2, “Overview | PM2 Documentation,” PM2, 2018. [Online]. Available: https://github.com/Unitech/pm2. [Acesso em 20 Março 2019].

[23] Node.js Foundation, “Cluster | Node.js v8.10 Documentation,” Node.js Foundation, 2019. [Online]. Available: https://nodejs.org/docs/latest-v8.x/api/cluster.html#cluster_how_it_works. [Acesso em 18 Março 2019].

[24] Express, “Express - Node.js web applications,” Node.js Foundation, 2019. [Online]. Available: https://expressjs.com/. [Acesso em 04 Fevereiro 2019].

[25] S. Sivasubramanian, “Amazon dynamoDB: a seamlessly scalable non-relational database service,” em *ACM SIGMOD International Conference on Management of Data*, Scottsdale, Arizona, USA, 2012.

[26] Amazon Web Services, “DynamoDB (Downloadable Version) on Your Computer - Amazon DynamoDB,” Amazon Web Services , 2019. [Online]. Available: https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/DynamoDBLocal.DownloadingAndRunning.html. [Acesso em 18 Maio 2019].

[27] Amazon Web Services, “AWS SDK for JavaScript in Node.js,” Amazon, 2019. [Online]. Available: https://aws.amazon.com/sdk-for-node-js/. [Acesso em 07 Maio 2019].

[28] A. Fernandez, “Loadtest,” 2019. [Online]. Available: https://github.com/alexfernandez/loadtest. [Acesso em 11 Março 2019].

[29] locust.io, “Locust - A modern load testing framework,” locust.io, [Online]. Available: https://locust.io/. [Acesso em 02 Abril 2019].

[30] Internet Engineering Task Force, “RFC 2068 - Hypertext Transfer Protocol -- HTTP/1.1,” Internet Engineering Task Force, Janeiro 1997. [Online]. Available: https://tools.ietf.org/html/rfc2068#section-8.1. [Acesso em 09 Abril 2019].

[31] J. Thomerson, “Saving Money By Replacing API Gateway With Application Load Balancer's Lambda Integration - Serverless Training by Jeremy Thomerson,” 13 Dezembro 2018. [Online]. Available: https://serverless-training.com/articles/save-money-by-replacing-api-gateway-with-application-load-balancer/. [Acesso em 24 Maio 2019].

[32] F. F.-H. Nah, “A study on tolerable waiting time: how long are Web users willing to wait?,” *Behaviour & Information Technology,* pp. 153-163, 03 Fevereiro 2007.