Universidade Federal de Pernambuco

Centro de Informática

Graduação em Ciência da Computação

# Em direção a um Framework Serverless para Docker Swarm

Ricardo Robson Mendes da Silva (rrms@cin.ufpe.br)

Trabalho de Graduação

Recife, Dezembro de 2018

Universidade Federal de Pernambuco

Centro de Informática

Graduação em Ciência da Computação

# Em direção a um Framework Serverless para Docker Swarm

Trabalho apresentado ao Programa de Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Bacharel em Ciência da Computação.

**Orientador:** Vinicius Cardoso Garcia

Recife, Dezembro de 2018

*Aos meus pais, Rui Robson e Severina Maria (Lia).*
*Ao meu irmão, José Reginaldo (Sargento Silva Neto).*
*Aos meus avós, José Reginaldo, Maria Dalva, Antônio Arruda (in memoriam) e Maria Francisca.*
*Ao meu grande amigo, Jack (in memoriam).*

**Agradecimentos**

À santíssima trindade que me guia.

À minha família pelo apoio e confiança que me deram nessa jornada de 5,5 anos de graduação, e por toda jornada antes dessa. Em especial aos meus pais, Rui Robson e Severina Maria (Lia), pelas suas renúncias, orações, amparos e por nunca terem me deixado só. À meu amado irmão, José Reginaldo (Sargento Silva Neto), pela inspiração de garra e honra. À minha querida tia Rubia Regina, pelos sacrifícios que fez por mim e pelas demonstrações de fraternidade.

À minha amada namorada, Leilane Neves, pela paciência, companheirismo e por me ajudar a direcionar o leme para o futuro quando este sofria com o árduo cotidiano.

À meus queridos amigos, que tornaram toda essa jornada divertida. Em especial Avyner Henrique, Deyvson Lazaro, Pedro Henrique, Guilherme Henrique, Luiz Felipe e Thayonara Alves, pela hombridade. Sonhamos e realizamos juntos.

À meus orientadores e referências, deste trabalho e da vida. Em especial Vinícius Garcia, Sérgio Soares, Adriano Gomes, Ernani Azevedo, Samuel Romeiro, Alberto Trindade, Lauro Augusto, Luiz Carlos e Rosberg Rodrigues. Vocês ajudam pessoas a serem melhores.

À todos que compõe o CIn-UFPE. É o empenho de cada profissional, de cada docente e de cada discente, que torna este local acolhedor, inovador e produtivo.

À toda turma do ISITICs. Em especial a Sala Raiz, por tudo que aprendemos, rimos e construímos.

À todos que passaram pela República do Senhor P.

*Visita Interiorem Terrae, Rectificando, Invenies Occultum Lapidem.*

**Resumo**

O crescimento de serviços conectados à internet, bem como as novas demandas de tecnologias por parte de usuários e entre serviços interconectados, proporcionou o surgimento da *cloud computing*. Novas tecnologias surgiram como implementações de serviços em *cloud computing*, entre elas o *Serverless*. Trata-se de uma função implantada como um serviço que respeita alguns princípios, como ser *stateless*, efêmera e com responsabilidade única. Grandes provedores de cloud computing já oferecem serviços de Serverless, como a AWS Lambda, Azure Functions, Google Cloud Functions e Oracle Cloud. Em paralelo ao crescimento dessas tecnologias, novas tecnologias de virtualização estão surgindo, entre elas o Docker Swarm - um gerenciador de *cluster* e *containers*. Este trabalho apresenta o sl-handler, uma ferramenta capaz de provê Serverless em um *cluster* Docker Swarm e é fortemente atrelada aos princípios de *Serverless* apresentados neste trabalho. Por fim apresenta também experimentos com funções de exemplo para avaliar o desempenho do sl-handler frente aos principais provedores de *Serverless* do mercado. O sl-handler é uma ferramenta preliminar, que dá seu primeiro passo em direção a um *framework Serverless* para Docker Swarm.

**Palavras-chave:** Serverless, Docker Swarm, Framework, Estudo Comparativo, Cloud Computing, Function as a Service.

**Abstract**

The growth of internet-connected services, as well as the new demands of technologies by users and between interconnected services, has led to the emergence of cloud computing. New technologies have emerged as cloud service implementations, among them Serverless. It is a function implanted as a service that respects some principles, such as being stateless, ephemeral and with single responsibility. Bigs cloud computing providers already offer Serverless services, such as AWS Lambda, Azure Functions, Google Cloud Functions and Oracle Cloud. In parallel to the growth of these technologies, new virtualization technologies are emerging, among them Docker Swarm - a cluster and container manager. This work presents sl-handler, a tool capable of providing Serverless in a Docker Swarm cluster and is strongly tied to the principles of Serverless presented in this work. Finally, it also presents experiments with example functions to evaluate the performance of sl-handler against the main Serverless providers of the market. The sl-handler is a preliminary tool, which takes its first step towards a Serverless framework for Docker Swarm.

**Keywords:** Serverless, Docker Swarm, Framework, Comparative Study, Cloud Computing, Function as a Service.

**Lista de tabelas**

Tabela 1 - Lista de linguagens suportadas por provedor de Serverless 26

Tabela 2 - Versões da função helloworld em cada provedor 32

Tabela 3 - Função Fibonacci e Matriz implementada no AWS Lambda 33

Tabela 4 - Resultados dos testes síncronos da função Hello World 36

Tabela 5 - Resultados dos testes síncronos da função Fibonacci(25) 37

Tabela 6 - Resultados dos testes síncronos da função Matriz(5) 38

Tabela 7 - Resultados dos testes assíncronos da função Matriz(5), sendo 10 requisições com 2 em concorrentes, 100 requisições com 10 concorrentes e 1000 requisições com 100 concorrentes 39

Tabela 8 - Resultados dos testes assíncronos da função Matriz(5), sendo 10 requisições com 2 em concorrentes, 100 requisições com 10 concorrentes e 1000 requisições com 100 concorrentes 40

**Lista de figuras**

Figura 1 - GQM do trabalho 22

Figura 2 - Arquitetura do sl-handler 29

Figura 3 - Arquitetura do cluster Docker Swarm com o sl-handler implantado 29

Figura 4 - Exemplo de código Serverless para o sl-handler. 31
Expõe uma rota /helloworld que retorna a mensagem “hello world”

Figura 5 - Código do servidor Node Express que executa junto ao 30
container Docker do Serverless

Figura 6 - Comparação de desempenho entre os provedores 43
Serverless com requisições sequenciais usando a função Hello World

Figura 7 - Comparação de desempenho entre os provedores 43
Serverless com requisições simultâneas usando a função Fibonacci

## Sumário

**1 Introdução** **10**
1.1 Contexto e motivação 10
1.2 Objetivos gerais 12
1.3 Objetivos específicos 13
1.4 Organização do trabalho 13

**2. Revisão da literatura** **14**
2.1 Conceitos 14
2.1.1 Cloud Computing 14
2.1.2 Modelos de serviços de Cloud Computing 14
2.1.3 Software as a Service (SaaS) 15
2.1.4 Platform as a Service (PaaS) 15
2.1.5 Infrastructure as a Service (IaaS) 16
2.1.6 Function as a Service (FaaS) 16
2.1.7 Serverless 17
2.1.8 Modelos de implantação de Cloud Computing 17
2.1.9 Nuvem pública 17
2.1.10 Nuvem privada 18
2.1.11 Nuvem híbrida 19
2.2 Estado da arte sobre Cloud Computing e Function as a Service 19

**3. Metodologia** **22**
3.1 Goal Question Metric (GQM) 22
3.2 Proposta de ferramenta 23
3.3 Forma de comparação 24

**4. Provedores Serverless** **25**
4.1 A escolha dos provedores Serverless 25
4.2 Provedores escolhidos 27
4.3 Ferramenta proposta (sl-handler) 28

**5. Experimentos** **32**
5.1 Funções de exemplo 32
5.2 Experimento com requisições 36

**6. Resultados e discussão** **37**
6.1 Resultados 37
6.2 Discussão 44

**7. Conclusão e trabalhos futuros** **46**
7.1 Conclusão 46
7.2 Trabalhos futuros 46

**8. Referências** **48**

# 1 Introdução

Este capítulo apresenta o contexto e a motivação deste trabalho e esclarece seus objetivos (gerais e específicos) e a organização da estrutura do documento. Este capítulo está dividido em quatro seções. A primeira seção, apresenta o contexto e motivação para o estudo realizado, assim como para a ferramenta desenvolvida (sl-handler, detalhada no capítulo 4). Os objetivos deste trabalho são apresentados na segunda seção e especificados na terceira seção. E por fim, a estrutura deste trabalho é descrita na quarta seção.

## 1.1 Contexto e motivação

A forma com que as pessoas interagem entre si, se expressam, buscam, compartilham e guardam seus conteúdos vêm passando por uma evolução significativa. Da mesma forma também evolui a maneira que serviços de computação interagem com outros serviços, plataformas e dispositivos sem que a interação parta necessariamente de uma ação humana. Isso decorre principalmente das mudanças da *World Wide Web*, que se dirige para um modelo que vai além do cenário em que foi proposta - que era compartilhar e prover dados em rede.

Essas mudanças na rede mundial de computadores (*World Wide Web*) é denominada Web 3.0 e torna possíveis sistemas que atuem a partir da já comum ação humana, mas também a partir de regras de negócios próprias ou ação de outros sistemas através da *internet*. Isso é possível devido a transposição do modelo tradicional de distribuição de *softwares* e sistemas (cliente-servidor) para um modelo em que são distribuídos na nuvem, no modelo computacional chamado de *Cloud Computing*. Desta forma um *software* não precisa mais passar por um processo de configuração ou atualização sempre que novas funcionalidades estiverem disponíveis. O *software* está disponível como um recurso na *internet*, basta acessá-lo e o terá com a versão mais recente disponível [1].

Com a tendência de *Cloud Computing* alguns conceitos ganharam adeptos com casos de sucesso, como a Salesforce com sua API já no início do século XXI. A sigla API vêm do inglês *Application Programming Interface*, que trata de fornecer formas de acesso à dados e regra de negócios de um sistema disponível em nuvem

através de uma interface via *internet*. Com o uso de API é possível que desenvolvedores do próprio sistema ou terceiros possam criar aplicações independentes ou acopladas à API. E essas aplicações por sua vez podem também serem disponibilizadas em nuvem [2]. Dessa maneira os sistemas passam a ter uma forma orgânica de crescimento nas funcionalidades e nos mercados que representam. Outros conceitos veremos no decorrer deste trabalho.

A rápida adoção do uso de API e a transposição de sistemas tradicionais para a nuvem trouxeram novos conceitos que impactam drasticamente a forma de se pensar, implantar e disponibilizar *softwares*. Pesquisas na área ganharam força a partir de 2007, onde buscam compreender e propor conceitos, alternativas, melhorias e otimizações na forma de implantar, disponibilizar e cobrar por esses serviços (tipicamente através de modelos de *SLA* - onde a cobrança é baseada no uso de recursos computacionais) [3]. Ao longo dessas pesquisas e dos casos propostos por grandes empresas (como Salesforce, Google e Amazon), conceitos sobre novos modelos computacionais implantados em nuvem ganharam relevância, como: *Software as a Service* (SaaS), *Storage as a Service* (StaaS), *Backend as a Services* (BaaS), *Function as a Service* (FaaS), entre outros.

Os trabalhos do *National Institute of Standards and Technology* (NIST) [4] e do Ian Sommerville [5, 6] foram importantes para elucidar os conceitos, desafios e problemas que cercam o modelo de *Cloud Computing*. Sommerville afirma que Sistemas Complexos de TI de Grande Escala (da sigla LSCITS em inglês) [7] são sistemas usados por grande número de usuários, com fluxos e intenções de uso distintas. Esses usuários, além da própria natureza dos sistemas, podem ocasionar comportamentos inesperados por parte dos sistemas ou infraestrutura em que os mesmos estão implantados. Um sistema nesse cenário tende a ter uma manutenção e evolução complexa, além de custos altos. Ian Sommerville destaca também dificuldades e problemas enfrentados por quem gerencia ou implanta projetos que serão disponibilizados em nuvem, e evidencia que os principais tratam de escalabilidade e alta disponibilidade de sistemas.

A escalabilidade e a alta disponibilidade são essenciais em sistemas implantados em nuvem, e a forma em que os sistemas alocam recursos para garantir essas características varia de acordo com suas funcionalidades e as

requisições feitas à eles. Manter recursos alocados desnecessariamente pode gerar custos sem motivos, enquanto a falta de recursos previamente alocados pode gerar gargalos em picos de acessos [8, 9]. É natural perceber também que a granularidade do módulo computacional é importante também para definir uma estratégia de escalabilidade e disponibilidade. Isto é, garantir essas características para SaaS e para FaaS demanda abordagens distintas, uma vez que a natureza conceitual das características nessas abordagens são distintas [10].

Serviços como o AWS Lambda[1], Azure Functions[2] e Google Cloud Functions[3] oferecem interfaces que permitem implantar FaaS de modo que a escalabilidade e a alta disponibilidade sejam asseguradas por eles. Há projetos de código aberto que também permitem a implantação de FaaS, como o OpenFaaS[4] e Fn Project[5], porém com uma abstração menor - uma vez que é preciso previamente instalar e configurar tal projeto em uma máquina ou *cluster*. Os serviços que abstraem o processo de configuração e disponibilidade das funções implantadas são chamados de *Serverless*.

*Serverless* é uma implementação de FaaS que segue alguns princípios, como ser *stateless*, efêmero, dirigido a evento e com alta abstração em sua implantação e disponibilidade. Dessa forma, espera-se que serviços que ofereçam *Serverless* permitam uma implementação em que o desenvolvedor ou o responsável não precise conhecer detalhes da máquina em que a função estará e nem da estratégia usada para manter tal função escalável e disponível [11].

## 1.2 Objetivos gerais

Este trabalho visa investigar a adequação aos princípios de *Serverless* por parte dos serviços da AWS Lambda, Azure Functions, Google Cloud, além de propor uma ferramenta que permita total adequação a esses conceitos e portabilidade entre os principais serviços de *Cloud Computing*.

[1] https://aws.amazon.com/pt/lambda/
[2] https://azure.microsoft.com/pt-br/services/functions/
[3] https://cloud.google.com/functions/
[4] https://github.com/openfaas
[5] https://fnproject.io

## 1.3 Objetivos específicos

- Desenvolver uma ferramenta para implantação, via API REST, de Serverless através do Docker Swarm
- Analisar o estado da prática dos principais provedores de Serverless do ponto de vista da aderência aos princípios de Serverless
- Comparar o desempenho entre a ferramenta proposta (sl-handler) e os principais serviços de *Serverless*

## 1.4 Organização do trabalho

- Capítulo 2: apresenta conceitos necessário para compreensão da ferramenta e experimentos propostos e uma revisão do estado da arte acerca de *Cloud Computing*, *FaaS* e *Serverless.*
- Capítulo 3: apresenta a metodologia utilizada no trabalho, isto é, os objetivos e a forma que o trabalho é feito para atingi-los.
- Capítulo 4: apresenta os principais serviços de *Serverless* e detalha a ferramenta proposta (sl-handler).
- Capítulo 5: detalha e apresenta a execução dos experimentos realizados.
- Capítulo 6: apresenta os resultados dos experimentos e uma discussão sobre eles.
- Capítulo 7: apresenta a conclusão do trabalho e perspectivas de trabalhos futuros.

## 2. Revisão da literatura

Este capítulo tem como objetivo apresentar o conteúdo necessário para o entendimento deste trabalho como um todo. Este capítulo está dividido em duas seções: a primeira aborda conceitos relacionados a área de *Cloud Computing* e a segunda apresenta o estado da arte em *Cloud Computing*, FaaS e *Serverless*.

### 2.1 Conceitos

Esta seção elucida os principais conceitos da área de *Cloud Computing* a fim de permitir ao leitor uma compreensão deste trabalho como um todo. Focamos nos conceitos essenciais da área, notando suas principais características e pequenas discussões sobre seus usos.

#### 2.1.1 *Cloud Computing*

*Cloud Computing* é um modelo que permite implantação e acesso à um módulo computacional sob demanda, com alta disponibilidade e via *internet*. Este modelo tipicamente permite que os recursos computacionais envolvidos (como por exemplo, armazenamento de dados, redes e aplicativos) sejam configuráveis. Dessa forma os recursos podem ser provisionados rapidamente para uso do módulo computacional, com pouco esforço de gerenciamento - ou nenhum, em casos que esse provisionamento é automático [4].

Dessa forma, o modelo de *Cloud Computing* possui as seguintes características:

- Serviço sob demanda
- Alta disponibilidade
- Recursos computacionais disponíveis
- Escalabilidade
- Serviço de medição de consumo do serviço e recursos utilizados

#### 2.1.2 Modelos de serviços de *Cloud Computing*

Atualmente existem vários modelos de serviços de *Cloud Computing,* cada um com suas particularidades, indicações de uso e desafios no estado da arte.

Neste trabalho iremos comentar os cincos principais modelos, sendo os três primeiros adequadamente definidos na publicação de referência “The NIST Definition of Cloud Computing”, em 2011, e por fim dois recentes modelos de *Cloud Computing* - a saber, *Serverless* como implementação de *Function as a Service*.

### 2.1.3 *Software as a Service* (SaaS)

Este modelo oferece um *software* para o usuário final e tipicamente é acessado por qualquer dispositivo capaz de visualizar páginas *web*. O acesso a ele é via *internet* e a disponibilidade e escalabilidade é abstraída pelo usuário. Dessa forma, em caso de picos de acesso ou atenuações na infraestrutura onde o sistema está disponível para a rede, fica sob responsabilidade do provedor do sistema provisionar a estabilidade [4, 13].

O usuário final já está familiarizado com este modelo, usa no cotidiano ao acessar suas redes sociais (como Facebook e Twitter), suas ferramentas de armazenamento de dados (como OneDrive e Dropbox) e até suas contas de email (como Gmail e Hotmail).

### 2.1.4 *Platform as a Service* (PaaS)

Assim como no modelo de SaaS, em *Platform as a Service* o usuário abstrai detalhes da infraestrutura que garante que o serviço se mantenha estável, além de abstrair a máquina virtual ou *container* em que a plataforma contratada está disponível. A diferença é que neste modelo o serviço final é configurável e programável pelo usuário ou contratante [4, 13].

PaaS é um modelo comumente usado por empresas desenvolvedoras de *software*. Como exemplo podemos citar a plataforma de venda Magento ou a de gerenciamento de projetos Jira Platform [14]. A ideia por trás desse modelo é a possibilidade de prover um serviço programável às características do próprio contratante, com a liberdade de implantar no provedor de *Cloud Computing* que desejar.

### 2.1.5 *Infrastructure as a Service* (IaaS)

Este modelo tem nível mais baixo de abstração. Aqui o usuário gerencia a nível de recursos computacionais o que irá utilizar na nuvem. Largura de banda da rede, capacidade de armazenamento em disco, memória alocada, tipo de processador (quantidade de núcleos de processamento, modelo do processador ou uso de GPU, por exemplo) entre outros recursos. Comumente o que é oferecido ao usuário é uma virtualização de uma máquina, seja através de métodos mais tradicionais de virtualização, ou mais modernos - como o uso de *Containers* via ssh [4, 13].

Neste modelo cabe ao usuário gerenciar o que acontece em situações de picos ou instabilidades na infraestrutura. Isto é, a forma que os recursos serão provisionados ou liberados para atender as adversidades da infraestrutura é configurada pelo usuário. Vale notar que neste modelo o usuário pode implantar PaaS, SaaS e IaaS, uma vez que já dispõe da infraestrutura que este será hospedado. Cabe apenas a responsabilidade de gerenciar os recursos consumidos pela plataforma. Em alguns casos os provedores de IaaS oferecem mecanismo de automação do dimensionamento desses recursos [14].

### 2.1.6 *Function as a Service* (FaaS)

Este modelo tem um alto nível de abstração. Aqui o módulo computacional implantado ou solicitado pelo usuário é uma função. Isto é, um módulo que para um determinado evento (tipicamente uma requisição HTTP) a função é executada - podendo ou não disparar um evento de resposta. Mais uma vez, assim como em SaaS e PaaS, os recursos computacionais e a disponibilidade da FaaS são de responsabilidade do provedor.

Em comparação com os demais modelos de *Cloud Computing*, o FaaS é mais recente, sendo lançado nos principais provedores entre 2014 e 2016 [15, 16]. Com uma vasta quantidade de eventos que podem ser configurados para executar a FaaS, surgiu uma vasta possibilidade de arquitetar sistemas na *web*. Por exemplo, inspirado em sistemas distribuídos, a partir de FaaS podemos arquitetar sistemas

como microsserviços. Esse modelo ganhou força com o caso de sucesso da Pivotal [17].

Josef Spillner, em “Snafu: Function-as-a-Service (FaaS) Runtime Design and Implementation” (2017), define:

> “Function-as-a-Service (FaaS) is therefore the technological concept to subdivide software applications into functions and to invoke these functions through network protocols with explicitly specified or externally triggered messages. The functions are deployed as FaaS units, encompassing the callable functions along with their dependencies, on dedicated FaaS runtimes. The runtimes are platform-level host services whose tasks are the blazingly fast allocation of resources for the function execution and the precise accounting of the associated duration and processing load.“

### 2.1.7 *Serverless*

O modelo FaaS passou a ser difundido nos últimos anos pelos principais provedores de *Cloud Computing* através de uma implementação chamada de *Serverless*. Essa implementação é tipicamente FaaS com alguns princípios intrínsecos. São eles [19, 20, 21]:

- A função deve executar em contexto efêmero,
- A função deve ter uma execução *stateless,*
- A função deve ter responsabilidade única,
- Recursos computacionais da infraestrutura, disponibilidade e escalabilidade são responsabilidade do provedor de *Serverless.*

### 2.1.8 Modelos de implantação de *Cloud Computing*

Existem três modelos de organização de fornecimento de *Cloud Computing*, esses ditam as características e responsabilidades no gerenciamento da infraestrutura e dos serviços implantados. São eles: Nuvem pública, nuvem privada e nuvem híbrida. Os três modelos oferecem vantagens semelhantes, no entanto é importante notar suas particularidades para uma escolha adequada entre eles.

### 2.1.9 Nuvem pública

Este é o modelo de nuvem mais usado. Adequado para a implantação de FaaS e SaaS. Os recursos são disponibilizados via *internet* pelos provedores,

através de abstrações com ambientes virtualizados. Esses recursos são compartilhados entre os usuários ou locatários, e a gerência do compartilhamento, bem como o provisionamento dos recursos, é de responsabilidade dos provedores. Normalmente há um acordo prévio entre o usuário e o provedor que estipula os recursos que estarão disponíveis e como serão escalonados - e, evidentemente, como o usuário será cobrado por eles [4, 37, 38].

As características que levam à aderência desse modelo são:

- Redução de custos - pois recursos de *hardware*, *software* e infraestrutura são abstraídos pelos provedores,
- Sem manutenção - pois os próprios provedores se encarregam das estratégias de manutenção de seus recursos,
- Escalabilidade - tipicamente usando um modelo de escalabilidade por replicação, os provedores se comprometem em garantir a escala e disponibilidade dos serviços implantados pelos usuários,
- Confiabilidade - a oferta e correções de falhas ocasionadas por infraestrutura são altas, uma vez que os provedores tem uma vasta rede espalhada globalmente para garantir a entrega e manutenção das infraestruturas que oferta os serviços dos usuários.

### 2.1.10 Nuvem privada

Neste modelo, os recursos podem ser disponibilizados em *datacenters* próprios, distribuindo assim os serviços via rede ou *internet* para filiais, ou a partir de provedores. Porém, este modelo não compartilha recursos entre outros usuários ou contratantes. A ideia por trás desse modelo é isolar os recursos para que sejam dedicados unicamente ao usuário e para que possam ser configurados conforme as necessidades dos serviços implantados, além das necessidades da cultura do usuário ou contratante [4, 37, 38].

Este modelo mantém as características da nuvem pública, e adiciona:

- Mais flexibilidade - uma vez que os recursos agora podem ser especificados para atender aos serviços do usuário ou contratante,

- Mais segurança - os recursos não são compartilhados com outros usuários, assim ficam restritos ao contratante e evita que problemas de terceiros afetem os recursos,
- Mais escalabilidade - uma vez que os recursos são dedicados aos serviços que são implantados pelo contratante, as medidas e heurísticas usadas para escalabilidade podem ser mais específicas.

### 2.1.11 Nuvem híbrida

A nuvem híbrida toma as características da nuvem pública e da nuvem privada, pois é caracterizada pelo uso de ambas. É ideal para negócios que tem parte de seus sistemas trabalhando com regras de negócios e dados críticos, demandando um uso de nuvem privada, e outra parte das regras de negócios e dados de caráter não sigiloso, podendo ser implantado em nuvem pública [4, 37, 38].

Vale notar que implantar um serviço em nuvem pública não significa que seus dados estão públicos, ou vulneráveis. A noção de que dados sigilosos devem ser tratados apenas em recursos computacionais próprios em uma rede privada está mais atrelada à cultura do contratante do que características de segurança desses modelos.

Este modelo mantém as características da nuvem pública e da nuvem privada, e adiciona:

- Controle dos custos e qualidade dos serviços - uma vez que nesse modelo o contratante pode estrategicamente migrar seus sistemas entre o ambiente público e o privado,
- Controle de módulos sigilosos ou críticos - o contratante pode usar a rede privada para implantar apenas os serviços que deseja implantar em um ambiente com recursos e configurações próprias.

## 2.2 Estado da arte sobre *Cloud Computing* e *Function as a Service*

Uma vez que elasticidade e disponibilidade são as características fundamentais em *Cloud Computing* [4, 5, 19, 20, 21, 22], focamos em elucidar o estado da arte delas. É importante lembrar que ao falarmos do estado da arte de

FaaS estamos falando também do estado da arte de *Serverless*, mas o inverso não é necessariamente verdade.

A abordagem de replicar as máquinas virtuais ou *Containers*, de forma reativa aos picos de requisições ou consumo de recursos, é a mais utilizada para garantir a escalabilidade dos serviços. Nessa abordagem, o provedor de Cloud Computing monitora os serviços e recursos alocados para eles de forma que quando uma determinada medida é atingida, o processo de escalonamento é iniciado. Isto é válido tanto para instanciar uma nova máquina quando as atuais estão com carga alta, quanto para desligar alguma máquina quando não há carga em demasiado nos serviços [22].

A escalabilidade via replicação também é usada de forma manual em alguns provedores de *Cloud Computing*, ou ferramentas de virtualização usadas por esses provedores - como Kubernetes e Docker Swarm. Essa abordagem requer um nível de *Cloud Computing* de IaaS [22]. Trabalhos futuros podem propor uma abstração da replicação manual para o nível de PaaS.

Em ambos os casos, escalar de forma automática ou manual, a quantidade de falsos positivos que levam a escalar em hora indevida, e falsos negativos que levam a não escalar quando era necessário, são preocupantes. Por exemplo, há cenários em que a máquina passa mais de 50% de seu tempo com consumo de recurso além de seu ideal, demorando demais para escalar - podendo comprometer a experiência do usuário que requisita o serviço [22]. Desta forma, há iniciativas de pesquisas para identificar limiares no *threshold* [23, 24, 25].

Especificamente no que se trata de *Serverless* ainda há discussão conceitual em aberto. Embora tenham princípios atribuídos à esta abordagem [19, 20, 21], verifica-se que o estado da prática nos principais provedores de serverless não os respeitam necessariamente [35, 36]. Em decorrência disso, podemos reforçar algumas questões em aberto:

- O código legado pode ser convertido para *Serverless*? Se sim, como?
- *Serverless* é fundamentalmente *stateless*?
- Haverá padrões para criar soluções *Serverless*?

- O *Serverless* se estende além das plataformas de nuvem tradicionais? Se sim, poderia *Serverless* ser adotado por outras tecnologias, como *Fog Computing*?

.

## 3. Metodologia

Este capítulo tem como objetivo apresentar a metodologia utilizada neste trabalho e está dividido em três seções: a primeira explica brevemente a metodologia GQM e como ela é utilizada neste trabalho, a segunda seção descreve a proposta da ferramenta desenvolvida e a terceira seção descreve a necessidade das comparações entre a ferramenta proposta e os principais provedores de *Serverless*.

### 3.1 *Goal Question Metric* (GQM)

Neste trabalho fizemos uso do método GQM, que é útil para planejar medições e comparações focadas em um objetivo. A ideia dessa abordagem é pensar no estudo de cima para baixo. Isto é, partindo da meta, realizar questões que queremos responder e por fim listar medidas que podemos tomar e avaliar para responder essas respostas [39, 40].

A Figura 1 apresenta o GQM utilizado para este trabalho. Com o fundo verde podemos ver as metas, em fundo azul temos as questões que são associadas às metas por traços pretos. Por fim, em rosa temos as medidas úteis para responder as questões, com a noção de utilidade representada pelos traços pretos mais uma vez.

Para atingirmos as metas desse trabalho precisaremos de uma proposta de ferramenta *Serverless* e responder as questões listadas no GQM. Para responder às questões precisamos coletar as medidas listadas. É importante perceber que o GQM utilizado satisfaz os objetivos gerais e específicos descritos nas seções 1.2 e 1.3 do capítulo 1. A forma que fizemos  isso é descrita nas seções 3.2 e 3.3.

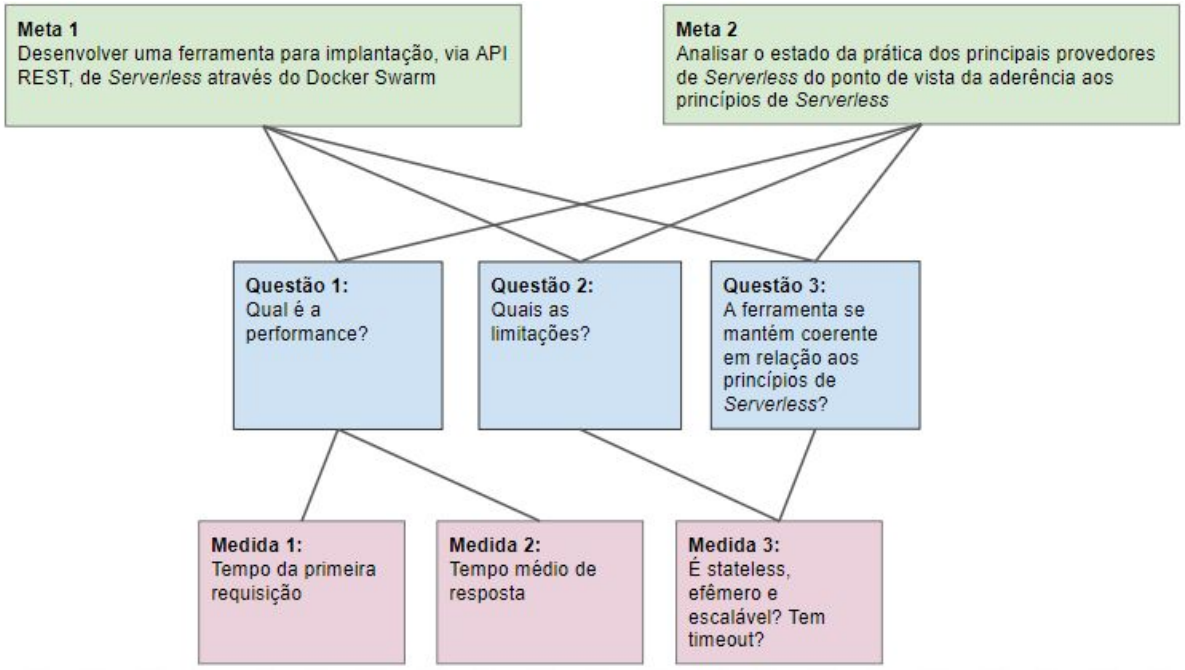



*Figura 1 - GQM do trabalho*

### 3.2 Proposta de ferramenta

A meta 1 do GQM (Figura 1) é a motivação deste trabalho. Trata-se de propor e desenvolver uma ferramenta que permita a implantação de *Serverless* em um cluster de Docker Swarm. Em posse dessa ferramenta, podemos dar seguimento ao trabalho comparando-a com os principais provedores de *Serverless* - meta 2.

Para o desenvolvimento dessa ferramenta precisamos levantar o estado da prática dos principais provedores de *Serverless*. Isso é útil para especificarmos as *features* necessárias para que a ferramenta seja comparável com as implementações dos principais provedores, e que cumpra o objetivo deste trabalho ao se adequar aos princípios de *Serverless* [11, 35, 36].

Os principais provedores de Serverless usados neste trabalho são detalhados no capítulo 4, junto com a justificativa de suas escolhas. Ainda no capítulo 4, detalharemos o sl-handler, a ferramenta proposta.

### 3.3 Forma de comparação

Para realizar as comparações entre a ferramenta sl-handler e os principais provedores de Serverless precisamos definir parâmetros de comparação. Para isso analisamos as informações disponíveis nesses provedores sobre as funções implantadas como *Serverless*, bem como as informações de suas execuções - estas são úteis para medir o desempenho das ferramentas.

Fazendo uma interseção entre as informações disponíveis pelos provedores, chegamos a conclusão que iremos analisar o desempenho a partir do tempo que uma requisição a um Serverless dura. Para isso faremos vários experimentos para medir o comportamento com alto volume de carga e com requisição única. Usaremos para isso funções com propriedade de consumo de recursos computacionais distintos.

## 4. Provedores *Serverless*

Este capítulo apresenta como os principais provedores de Serverless usados para avaliação e comparação foram escolhidos, além de apresentar suas características. O capítulo está dividido em duas três seções. A primeira apresenta a tomada de decisão sobre quais provedores iremos usar. A segunda apresenta brevemente os principais provedores escolhidos e, por fim, a terceira apresenta a ferramenta proposta.

### 4.1 A escolha dos provedores *Serverless*

Existem muitos provedores de *Serverless* atualmente, entre eles podemos citar AWS Lambda, Azure Functions, DigitalOcean, Fn-An, Google Cloud Functions, OpenWhisk, entre outros. Além de ferramentas *open sources* que permitem a criação de um provedor *Serverless*, como o OpenFaaS, Fn Project e o Kubeless. Para que seja possível elucidar como a ferramenta desenvolvida neste trabalho se enquadra com o que é entregue no mercado, no estado da prática, e também nos princípios de *Serverless*, precisamos identificar as características dos principais provedores.

Começamos então selecionando os principais provedores de *Serverless*. Elegemos os principais provedores semelhantes e analisamos suas adequações aos princípios de Serverless citados na seção 2.1.2.5, seguindo um raciocínio descrito abaixo. Para lembrar, segue os princípios de *Serverless*:

- A função deve executar em contexto efêmero,
- A função deve ter uma execução *stateless,*
- A função deve ter responsabilidade única.

Começamos então decidindo ignorar, para este trabalho, os projetos *open sources*, como OpanFaaS e o Fn Project. Esses projetos exigem contratação de IaaS e suas instalações e configurações, com isso acabam adicionando complexidade que fogem do escopo deste trabalho. Pretendemos apenas avaliar a forma de implantar *Serverless* e seus desempenhos. Reforçando essa decisão,

temos também que os projetos *open sources* não aparecem nas listas dos mais usados [41, 42, 43].

Uma vez que excluímos os *open sources*, analisamos então os principais provedores das listas dos mais usados. São eles: AWS Lambda, Azure Functions e Google Cloud Functions. Ao entrarmos nos sites oficiais desses projetos podemos identificar *features* e restrições que apresentam [44, 45, 46, 47]. As principais *features* apresentadas são: Linguagens de programação suportadas, tempo máximo de execução por chamada ao *Serverless*, forma de cobrança pelo serviço e se são ou não *Stateless* (normalmente justificado por uma abordagem de otimização de desempenho dos Serverless).

Decidimos ignorar as features sobre forma de pagamento e tempo máximo de execução. Elas não fazem parte dos princípios *Serverless*, de forma que sua existência não fere nem acrescenta adequação aos princípios.

Identificamos também que os provedores otimizam o tempo de execução dos Serverless, quando estão em picos de requisições, através de estratégias que podem ferir o princípio de execuções *stateless*. Um exemplo de estratégia é descrito na página da AWS Lambda [58]:

> "Para melhorar o desempenho, o AWS Lambda pode escolher reter uma instância de sua função e reutilizá-la para atender a uma solicitação posterior em vez de criar uma nova cópia."

Estratégias assim ferem também o princípio de ser efêmero - se considerarmos esse princípio sobre todas as execuções do *Serverless*. Resolvemos então ignorar as implementações dessas estratégias por parte dos provedores, uma vez que estão de acordo que podem levar a ferir o princípio de execuções stateless e de serem efêmeros, porém orientam os usuários a tomarem medidas para evitar problemas inesperados nesse contexto. Uma outra razão é que o princípio de execução *stateless* não diz quem é o responsável por garantir isso. Assumimos então que os provedores citados acima, cientes disso, implementam boas estratégias de escalabilidade e disponibilidade e orientam os usuários para garantir execução *stateless*. Ou seja, assim como garantir responsabilidade única em função é dever de quem desenvolve, ser *stateless* também é. Ademais, a ferramenta proposta irá garantir uma execução stateless, tornando toda execução efêmera.

Analisando as linguagens suportadas, na Tabela 1, temos em destaque - fundo azul - o Node.js (JavaScript). Dessa forma fica decidido que a ferramenta proposta oferece suporte a Node.js. Concluímos então que a ferramenta proposta não tem tempo limite para execução dos *Serverless*, é efêmera para cada requisição (e por isso, *stateless*) e dá suporte a Node.js.

| | **AWS Lambda** | **Azure Functions** | **Google Cloud Functions** |
|---|---|---|---|
| **Node.js (JavaScript)** | Sim | Sim | Sim |
| **Java** | Sim | Sim | Não |
| **C#** | Sim | Sim | Não |
| **Python** | Sim | Experimental | Não |
| **PHP** | No | Experimental | Não |
| **Go** | Sim (Parcial) | Não | Não |
| **F#** | Não | Yes | Não |
| **Swift** | Não | Não | Não |

Tabela 1 - Lista de linguagens suportadas por provedor de *Serverless*

### 4.2 Provedores escolhidos

Os provedores escolhidos estão entre os mais utilizados para implantação de *Serverless*. É comum vermos análises de desempenho entre eles na comunidade, além dos próprios fazerem ajustes nas suas estratégias de cobranças e escalabilidade [55, 56, 57]. Neste trabalho não levamos em conta avaliações passadas, uma vez que no momento presente podem ter alterado suas estratégias em relação à data das avaliações. Fizemos experimentos próprios, incluindo na ferramenta proposta.

Os provedores escolhidos foram, então, os seguintes:

- **AWS Lambda**

  "*Execute códigos sem pensar sobre os servidores. Pague apenas pelo tempo de computação que você utilizar.*", assim se apresenta o AWS Lambda em seu site oficial [44]. Além do tradicional HTTP, esse provedor permite que o *Serverless* implantado seja executado a partir de evento interno de outro serviço da Amazon, como o projeto Alexa, algum evento de um DynamoDB, Kinesis, SNS e até CloudWatch.

  O AWS Lambda, um serviço da Amazon, tem casos de uso que se concentram em processamento de arquivos em tempo real, processamento de dados e processamento de *stream*. Isso faz com que seja adequado para *back-ends* de IOT [47]. Grandes empresas, como Coca-Cola Company, iRobot, Benchling e Thomson Reuters fazem uso desse serviço para atender suas necessidades desde páginas *web* até seus projetos de IOT [48, 49, 50, 51].

- **Azure Functions**

  O Azure Functions é um serviço de *Serverless* da Microsoft. Seus casos de uso são comparáveis aos dos outros provedores de *Serverless*, no entanto em seu site oficial destacam-se a possibilidade de agendar execuções e arquitetar as funções como extensões de SaaS. Com essas *features* tem clientes com casos de uso bastante distintos [52, 53, 54].

- **Google Cloud Functions**

  O Google Cloud Functions tem casos de uso semelhantes aos demais. No entanto traz novidade ao oferecer características em sua infraestrutura capaz de oferecer *Serverless* para análise de sentimento, análise de vídeo e imagem e *chatbots*. Faz isso oferecendo uso de GPU compartilhada, em nuvem pública.

### 4.3 Ferramenta proposta (sl-handler)

Após a análise da seção 4.1, concluímos que a ferramenta proposta não tem tempo limite para execução dos Serverless, é efêmera para cada requisição (e por isso, *stateless*) e dá suporte a Node.js. Essa feramenta está disponível no repositório git em https://github.com/ricardorobson/sl-handler, sob licença ISC, e foi

desenvolvido em Golang sob a motivação de ser uma linguagem que se adequa bem ao contexto de muitas requisições e processamento paralelo.

O sl-handler é uma ferramenta que pode ser executada em um cluster Docker Swarm e passa então a gerenciar Serverless nesse cluster. O usuário pode enviar código Node.js via API, envolto ao módulo Express do Node, para que o sl-handler possa criar uma imagem Docker com esse código e associar a ela uma rota HTTP. Dessa forma quando a rota associada for chamada, o sl-handler cria o *container* Docker a partir da imagem, e passa para ele a requisição. O *container*, que é contém função Serverless enviada pelo usuário, retorna então sua resposta para o sl-handler, que destrói o *container* e devolve a resposta para quem requisitou.

A Figura 2 representa a arquitetura do sl-handler. À esquerda vemos clientes, que podem ser qualquer dispositivos que faça requisição HTTP para o módulo principal do sl-handler. O módulo principal é responsável por receber requisições e direcionar para a regra de negócio associada, como chamar o módulo Database para salvar um novo *Serverless* ou chamar o módulo Docker para executar um *Serverless* existente. O módulo Database é responsável por cuidar do CRUD (*Create*, *Read*, *Update* e *Delete*) dos *Serverless* implantados nessa ferramenta, para isso ele acessa um banco de dados parametrizado - ou seja, pode ser alterado sob a escolha do usuário. O módulo docker é responsável por se comunicar com a API Docker do *cluster*, podendo criar imagens Docker dos *Serverless*, iniciar e destruir *containers*. Por fim, à direita, temos o *cluster* em Docker Swarm onde os *Serverless* são implantados e executados. É importante notar que o sl-handler é implantado como um *container Docker* dentro do próprio *cluster*.

Ainda sobre a Figura 2, as linhas em vermelho representam as comunicações que são feitas em HTTP. As latências envolvidas nessa comunicação fogem do escopo do trabalho, uma vez que a infraestrutura entre o dispositivo que requisita o sl-handler e o *cluster* é de responsabilidade desconhecida.

A Figura 3 apresenta a arquitetura do *cluster* Docker Swarm com o sl-handler implantado. Vemos então que o sl-handler é um *container* Docker, e os *Serverless* são imagens Docker que eventualmente são instanciadas como *containers* e em seguida destruídas.

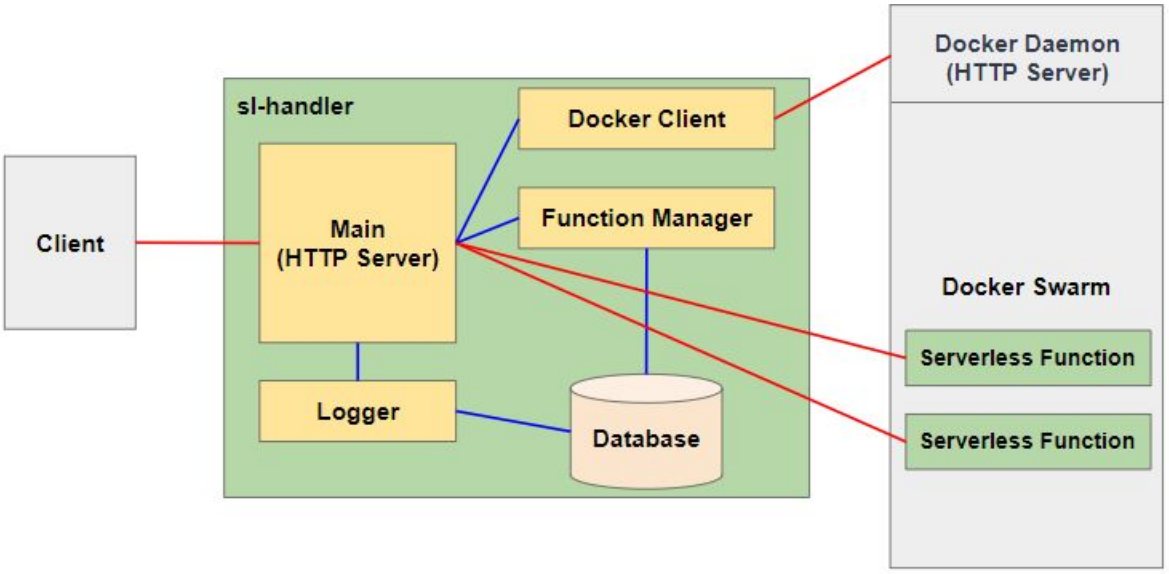



Figura 2 - Arquitetura do sl-handler

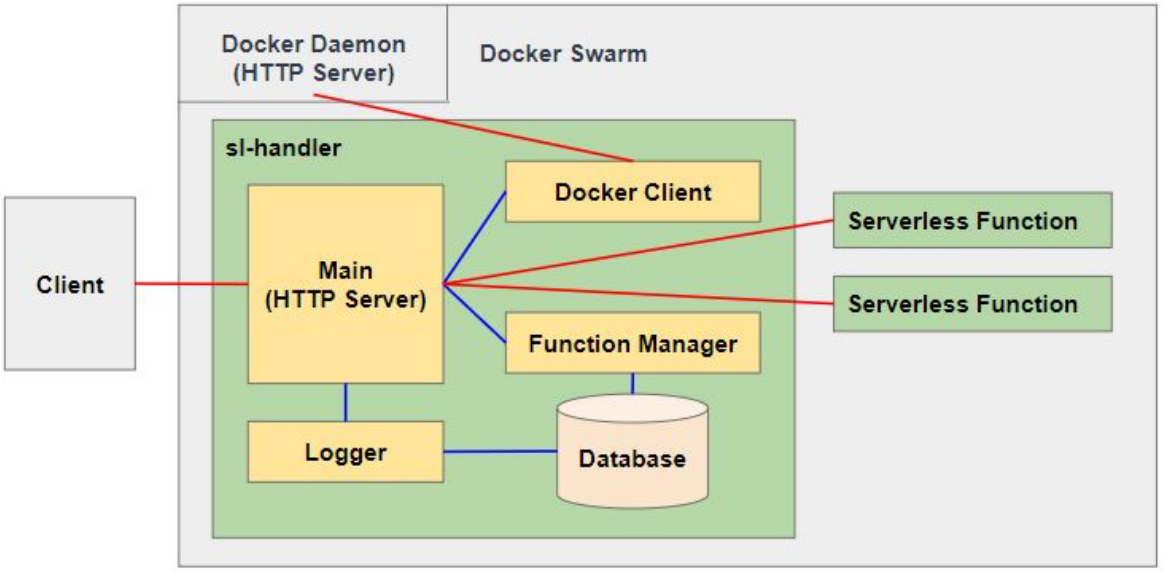



*Figura 3 - Arquitetura do cluster Docker Swarm com o sl-handler implantado*

Assim como nos principais provedores de *Serverless*, que definem uma estrutura básica de código para o *Serverless* receber a variável *req* (que contém a requisição HTTP que o chamou) e *res* (que define a resposta HTTP do Serverless), o sl-handler também tem seu padrão. A Figura 4 ilustra um “hello world” *Serverless*

para o sl-handler. Note que o exemplo utiliza o *module.exports*, do Node Express. Isto se dá por que, por padrão, todo *container Serverless* do sl-handler utilida o Node Express para iniciar um servidor HTTP que permita o *Serverless* receber e responder requisições HTTP. O código do servidor Node Express que executa no *container Serverless* está disponível na Figura 5.

```
module.exports.helloWorld = (req, res) => {
    console.log("hello world")
    res.send(req.method)
}
```

*Figura 4 - Exemplo de código Serverless para o sl-handler. Expõe uma rota /helloworld que retorna a mensagem "hello world".*

```
'use strict';
const express = require('express');
const functions = require('./code.js');

// Constants
const PORT = 8080;
const HOST = '0.0.0.0';

// App
const app = express();

Object.keys(functions).map( name => app.all(`/${name}`, functions[name] ))

app.listen(PORT, HOST);
```

*Figura 5 - Código do servidor Node Express que executa junto ao container Docker do Serverless.*

## 5. Experimentos

Este capítulo apresenta os experimentos executados para avaliar o desempenho da ferramenta proposta. O capítulo está dividido em duas seções. A primeira seção comenta sobre as funções que serão usadas nos experimentos e a segunda seção apresenta a como foi feita a execução do experimento. Em linhas gerais, este capítulo oferece as ferramentas para responder a questão 1 do QGM (Figura 1), o desempenho dos principais provedores Serverless.

### 5.1 Funções de exemplo

Para a etapa de experimento, trabalhamos com 3 funções *Serverless*. Um “hello world”, Fibonacci recursivo (para consumir memória) e multiplicação de matriz (para consumir processador). Cada função escrita pelo autor e foi implantada nos provedores de Serverless e no sl-handler e foi testada individualmente. Esse experimento mede o tempo de resposta da primeira requisição e o tempo médio das demais requisições a cada uma das funções em cada um dos provedores com cargas de requisições distintas.

O sl-handler foi executado em uma máquina com uma distribuição Linux Oracle Server, de 15GB de RAM e 128GB de armazenamento. Esta máquina não dispõe de escalonamento ou provisionamento automático de recursos. As únicas aplicações instaladas foram o Docker-Engine e o git, para preparar e executar o sl-handler.

As funções foram implementadas com códigos semelhantes nos provedores e no sl-handler, preservando suas propostas e diferindo apenas no padrão de captura do *trigger* HTTP e de envio de resposta de cada provedor. A Tabela 2 lista a função helloworld implementada em cada provedor para exemplificar a diferença citada.

As demais funções estão listadas na Tabela 3 apenas na versão do AWS Lambda, uma vez que as diferenças entre as dos outros provedores estão apenas na chamada à função principal e na forma de envio da resposta, devidamente ilustradas na Tabela 2. A motivação da função Fibonacci está em medir como se

comporta os provedores em uma função recursiva, e a função Matriz em como de comporta com muitas operações aritmética no processador.

| | Função |
|---|---|
| **AWS Lambda** | *exports.handler = async (event) => {*<br>*  const response = {*<br>*  statusCode: 200,*<br>*  body: "hello world",*<br>*  };*<br>*  return response;*<br>*};* |
| **Azure Functions** | *module.exports = async function (context, req) {*<br>*  context.res = {*<br>*  status: 200,*<br>*  body: "hello world"*<br>*  };*<br>*};* |
| **Google Cloud Functions** | *exports.helloWorld = (req, res) => {*<br>*  let message = 'Hello World!';*<br>*  res.status(200).send(message);*<br>*};* |
| **sl-handler** | *module.exports.helloWorld = (req, res) => {*<br>*  res.send("hello world")*<br>*}* |

*Tabela 2 - Versões da função helloworld em cada provedor*

| | AWS Lambda |
|---|---|
| **Fibonacci** | *exports.handler = async (event, context, callback) => {*<br>*  const response = {*<br>*  statusCode: 200,*<br>*  body: fib(event["queryStringParameters"]["param"]),*<br>*  };*<br>*  return response;*<br>*};*<br><br>*function fib(n){*<br>*  if(n < 2) return n;*<br><br>*  return fib(n - 1) + fib(n - 2);*<br>*}* |
| **Matriz** | *exports.handler = async (event) => {*<br>*  let length = event["queryStringParameters"]["param"];*<br>*  let a = new Array(length);*<br>*  let b = new Array(length);*<br>*  for(i=0;i<length;i++){*<br>*  var aLine = new Array(length);*<br>*  var bLine = new Array(length);* |

```
        for(j=0;j<length;j++){
            aLine[j]=j;
            bLine[j]=length-j;
        }
        a[i]=aLine;
        b[i]=bLine;
    }
    let m=multiply(a, b);
    for(i=0;i<length;i++){
        m=multiply(multiply(a, m), multiply(b, m));
    }
    const response = {
        statusCode: 200,
        body: display(m),
    };
    return response;
};

function multiply(a, b) {
  var aNumRows = a.length, aNumCols = a[0].length,
      bNumRows = b.length, bNumCols = b[0].length,
      m = new Array(aNumRows);
  for (var r = 0; r < aNumRows; ++r) {
    m[r] = new Array(bNumCols);
    for (var c = 0; c < bNumCols; ++c) {
      m[r][c] = 0;
      for (var i = 0; i < aNumCols; ++i) {
        m[r][c] += a[r][i] * b[i][c];
      }
    }
  }
  return m;
}

function display(m) {
  var ret = m[0];
  for (var r = 1; r < m.length; ++r) {
    ret+=", "+m[r];
  }
  return ret;
}
```

*Tabela 3 - Função Fibonacci e Matriz implementada no AWS Lambda*

### 5.2 Experimento com requisições

Para avaliar como se comportam os provedores de *Serverless* usamos um módulo Node.JS chamado Loadtest. Com este módulo é possível enviarmos requisições síncronas e assíncronas (definindo a quantidade de requisições paralelas por iteração). Com o Loadtest podemos definir quantas requisições serão feitas e quantas serão concorrentes, e ao final ele nos retorna um relatório do teste com a quantidade de requisições feitas, a quantidade de requisições que falharam, o tempo total do teste e o tempo médio de latência. A máquina utilizada para o experimento com o Loadtest é um Notebook Asus X550LA, com 8GB de RAM, processador Intel I5.

Decidimos então, para cada função de cada provedor, avaliar o tempo de resposta para a primeira requisição, para requisições sucessivas síncronas e para requisições sucessivas e concorrentes. Fizemos então os seguintes testes, limitando o tempo máximo para 1 hora (cancelando o teste e classificando-o como “tempo excedido” quando esse tempo fosse atingido):

i. 1 requisição
ii. 10 requisições sucessivas e síncronas.
iii. 100 requisições sucessivas e síncronas.
iv. 1000 requisições sucessivas e síncronas.
v. 10 requisições sucessivas e assíncronas com 2 concorrentes.
vi. 100 requisições sucessivas e assíncronas com 10 concorrentes.
vii. 1000 requisições sucessivas e assíncronas com 10 concorrentes.

## 6. Resultados e discussão

Este capítulo apresenta e discute os resultados dos experimentos. O capítulo é dividido em duas seções. A primeira apresenta os resultados dos experimentos e a segunda discute os resultados.

### 6.1 Resultados

A Tabela 4 apresenta os resultados dos experimentos síncronos na função Hello World. A Tabela 5 faz o mesmo para a função Fibonacci e a Tabela 6 para a função Matriz. Para Fibonacci os testes foram feitos com o parâmetros 25. Para Matriz os testes foram feitos com o parâmetro 5.

Já os resultados dos experimentos assíncronos da função Fibonacci com parâmetro 25 estão na Tabela 7, representando os itens v, vi e vii da seção 5.2 do capítulo 5. Os resultados da função Matriz com parâmetro 5 estão na Tabela 8. Não realizamos experimentos assíncronos para a função Hello World.

| Hello World | | 1 requisição | 10 requisições | 100 requisições | 1000 requisições |
|---|---|---|---|---|---|
| **AWS Lambda** | **Falhas** | 0 | 0 | 0 | 0 |
| | **Latência média** | 1230,7 ms | 574,2 ms | 477,2 ms | 499,2 ms |
| | **Tempo total** | 1,234 s | 5,746 s | 47,734 s | 499,281 s |
| **Azure Functions** | **Falhas** | 0 | 0 | 0 | 0 |
| | **Latência média** | 6888 ms | 946,5 ms | 766,7 ms | 768,2 ms |
| | **Tempo total** | 6,891 s | 9.469 s | 76,681 s | 768,292 s |
| **Google Cloud Functions** | **Falhas** | 0 | 0 | 0 | 0 |
| | **Latência média** | 3348,4 ms | 519,1 ms | 485,2 ms | 479,8 ms |
| | **Tempo total** | 3,351 s | 5,194 s | 48,532 s | 479,866 s |
| **sl-handler** | **Falhas** | 0 | 0 | 0 | 0 |
| | **Latência média** | 3512,5 ms | 3071,5 ms | 3598,7 ms | 3472,4 ms |
| | **Tempo total** | 3,516 s | 30,718 s | 359,878 s | 3472,477 s |

*Tabela 4 - Resultados dos testes síncronos da função Hello World*

| Fibonacci(25) | | 1 requisição | 10 requisições | 100 requisições | 1000 requisições |
|---|---|---|---|---|---|
| **AWS Lambda** | **Falhas** | 0 | 0 | 0 | 0 |
| | **Latência média** | 606,5 ms | 589,2 ms | 503,1 ms | 499,8 ms |
| | **Tempo total** | 0,610 s | 5,896 s | 50,320 s | 499,862 s |
| **Azure Functions** | **Falhas** | 0 | 0 | 0 | 0 |
| | **Latência média** | 7003,3 ms | 864,1 ms | 788 ms | 767,1 ms |
| | **Tempo total** | 7,006 s | 8,646 s | 78,819 s | 767,169 s |
| **Google Cloud Functions** | **Falhas** | 0 | 0 | 0 | 0 |
| | **Latência média** | 698,7 ms | 488,2 ms | 457,8 ms | 473,4 ms |
| | **Tempo total** | 0,702 s | 4,886 s | 45,792 s | 473,424 s |
| **sl-handler** | **Falhas** | 0 | 0 | 0 | * Tempo excedido |
| | **Latência média** | 3413,1 ms | 3415,6 ms | 3431,8 ms | |
| | **Tempo total** | 3,416 s | 34,160 s | 343,195 s | |

*Tabela 5 - Resultados dos testes síncronos da função Fibonacci(25)*

| Matriz(5) | | 1 requisição | 10 requisições | 100 requisições | 1000 requisições |
|---|---|---|---|---|---|
| **AWS Lambda** | **Falhas** | 0 | 0 | 0 | 0 |
| | **Latência média** | 841,3 ms | 547,4 ms | 496,8 ms | 497,9 ms |
| | **Tempo total** | 0,844 s | 5,478 s | 49,693 s | 497,961 s |
| **Azure Functions** | **Falhas** | 0 | 0 | 0 | 0 |
| | **Latência média** | 7865,5 ms | 767,4 ms | 765 ms | 757,1 ms |
| | **Tempo total** | 7,868 s | 7,678 s | 76,514 s | 757,164 s |
| **Google Cloud Functions** | **Falhas** | 0 | 0 | 0 | 0 |
| | **Latência média** | 794,1 ms | 444,8 ms | 458,2 ms | 461,8 ms |
| | **Tempo total** | 0,797 s | 4,452 s | 45,826 s | 461,828 s |
| **sl-handler** | **Falhas** | 0 | 0 | 0 | 0 |
| | **Latência média** | 3589,8 ms | 3349,3 ms | 3452,2 ms | 3421,8 ms |
| | **Tempo total** | 3,593 s | 33,497 s | 345,227 s | 3421,855 s |

*Tabela 6 - Resultados dos testes síncronos da função Matriz(5)*

| Fibonnaci(25) | | 10 requisições com 2 concorrentes | 100 requisições com 10 concorrentes | 1000 requisições com 100 concorrentes |
|---|---|---|---|---|
| **AWS Lambda** | **Falhas** | 0 | 0 | 0 |
| | **Latência média** | 517,2 ms | 518 ms | 563,2 ms |
| | **Tempo total** | 2,637 s | 5,328 s | 5,956 s |
| **Azure Functions** | **Falhas** | 0 | 0 | 0 |
| | **Latência média** | 860,6 ms | 801,3 ms | 837,2 ms |
| | **Tempo total** | 4,377 s | 8,574 s | 8,797 s |
| **Google Cloud Functions** | **Falhas** | 0 | 0 | 0 |
| | **Latência média** | 476,8 ms | 485,3 ms | 694,3 ms |
| | **Tempo total** | 2,457 s | 5,092 s | 7,367 s |
| **sl-handler** | **Falhas** | 0 | 3 | 22 |
| | **Latência média** | 5254,3 ms | 22849,5 ms | 253799,2 ms |
| | **Tempo total** | 27,443 s | 238,396 s | 2640,342 s |

*Tabela 7 - Resultados dos testes assíncronos da função Matriz(5), sendo 10 requisições com 2 em concorrentes, 100 requisições com 10 concorrentes e 1000 requisições com 100 concorrentes*

| Matriz(5) | | 10 requisições com 2 concorrentes | 100 requisições com 10 concorrentes | 1000 requisições com 100 concorrentes |
|---|---|---|---|---|
| **AWS Lambda** | **Falhas** | 0 | 0 | 0 |
| | **Latência média** | 501,2 ms | 500,1 ms | 541,6 ms |
| | **Tempo total** | 2,558 s | 5,306 s | 5,718 s |
| **Azure Functions** | **Falhas** | 0 | 0 | 0 |
| | **Latência média** | 830,5 ms | 863,4 ms | 847,3 ms |
| | **Tempo total** | 4,196 s | 8,954 s | 8,940 s |
| **Google Cloud Functions** | **Falhas** | 0 | 0 | 0 |
| | **Latência média** | 488,6 ms | 473,1 ms | 645,3 ms |
| | **Tempo total** | 2,492 s | 4,873 s | 6,802 s |
| **sl-handler** | **Falhas** | 0 | 5 | 9 |
| | **Latência média** | 5326,4 ms | 23659,4 ms | 256935,2 ms |
| | **Tempo total** | 27,769 s | 248,138 s | 2653,280 s |

*Tabela 8 - Resultados dos testes assíncronos da função Matriz(5), sendo 10 requisições com 2 em concorrentes, 100 requisições com 10 concorrentes e 1000 requisições com 100 concorrentes*

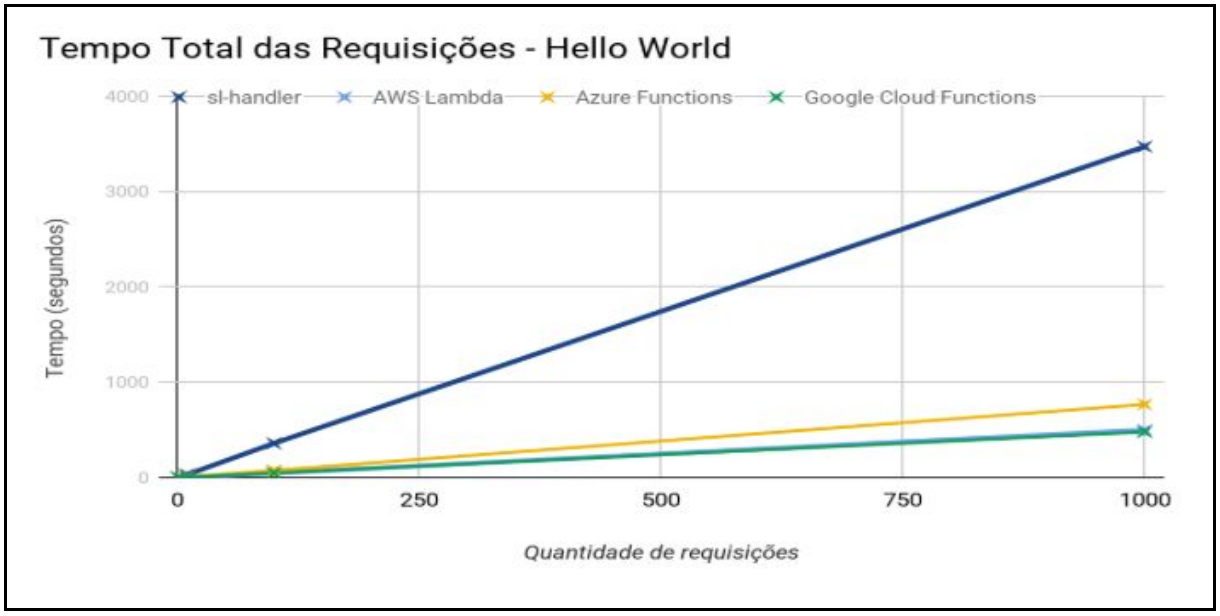



Figura 6 - Comparação de desempenho entre os provedores Serverless com requisições sequenciais usando a função Hello World

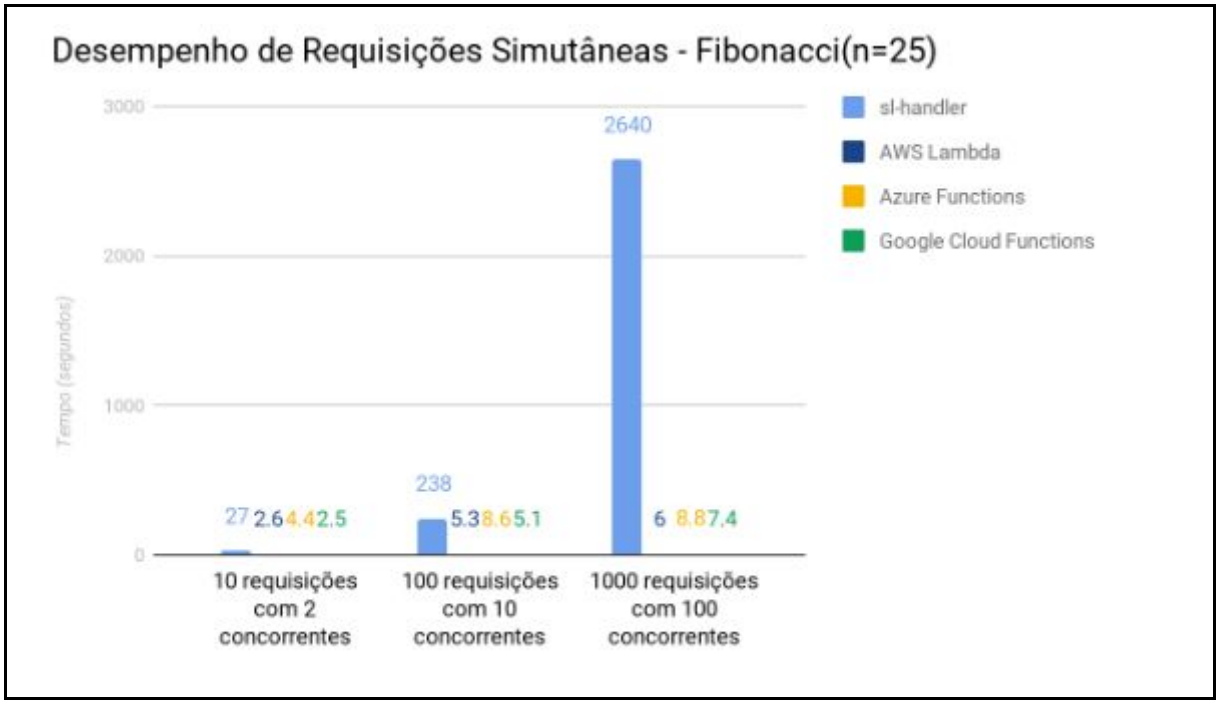



Figura 7 - Comparação de desempenho entre os provedores Serverless com requisições simultâneas usando a função Fibonacci

## 6.2 Discussão

Nossa discussão começa com a primeira requisição feita aos *Serverless* em cada provedor. Vimos que o sl-handler responde com cerca da metade do tempo do Azure Functions, no entanto dura mais de três vezes o tempo do AWS Lambda e o Google Cloud Functions. Isso se deve a forma que esses provedores provisionam os *Serverless* implantados. A AWS Lambda mantém disponível o *Serverless* assim que é implantada ou atualizada, de forma que as próximas requisições serão atendidas por instâncias do *Serverless* já disponíveis, isto diminui a latência [58]. A Google Cloud Functions implementa de forma semelhante. A Azure Functions não tem essa abordagem para a primeira requisição.

Conforme discutimos na seção 4.3 do capítulo 4, o sl-handler assume os princípios Serverless de maneira conservadora, de forma que considera o princípio de ser efêmero para cada requisição - isto é, inicia o *container*, executa a função e deleta o *container* para cada requisição. Ao considerar isso, garante que toda execução é *stateless*.

Ao contrário do sl-handler, a AWS Lambda, a Azure Functions e a Google Cloud Functions mantém a instância da função ligada para atender requisições simultâneas e sucessivas. Essa instância só é desligada se passar um período de tempo sem requisição. Essa abordagem é justificada pelo nicho de mercado que atendem e para que possam manter alta disponibilidade e escalabilidade dos *Serverless* implantados.

Seguimos nossa discussão agora sobre os experimentos de requisições simultâneas. Podemos observar que conforme aumentamos a carga de requisições simultâneas, o sl-handler se deparou com falhas. Nos experimentos de 1000 requisições com 100 simultâneas, cerca de 2,2% das requisições Fibonacci(25) falharam, e cerca de 0,9% das requisições para Matriz(5) falharam. Essas falhas se devem a uma restrição em código do próprio sl-handler, que espera 2 segundos para que o *container* da do *Serverless* seja iniciado. Caso ultrapasse esses 2 segundos e o *container* não estiver pronto, a chamada é considerada como falha. O motivo para que algumas requisições tenham apresentados esse cenário carece de investigação,

mas picos de consumo de memória na máquina em que o sl-handler foi testado, deixando-a lenta, é um provável motivo.

O tempo de resposta do sl-handler nos experimentos, por consequência de sua implementação, se mostrou muito alto quando comparado com os outros provedores. Cada requisição no experimento concorrente durou, no sl-handler, cerca de cinco vezes mais que nos demais provedores.

## 7. Conclusão e trabalhos futuros

Este capítulo apresenta a conclusão deste trabalho e perspectivas de trabalhos futuros. Este capítulo é dividido em duas seções. A primeira seção conclui o trabalho e a segunda apresenta tópicos para trabalhos futuros.

### 7.1 Conclusão

Este trabalho então entrega as metas 1 e 2 do GQM apresentado na seção 3.1 do capítulo 3. São elas: “Desenvolver uma ferramenta para implantação, via API REST, de *Serverless* através do Docker Swarm” e “Analisar o estado da prática dos principais provedores de *Serverless* do ponto de vista da aderência aos princípios de *Serverless”*, respectivamente.

O trabalho entrega o sl-handler fortemente alinhado aos princípios Serverless. Esses princípios, no entanto, têm sido questionado no estado da arte. Com isto, e com os resultados de desempenho do sl-handler frente aos provedores que implementam estratégias para melhora de performance que ferem os princípios, é notável que há possibilidade de alterações e contribuições nas estratégias do sl-handler em direção à melhorar seu desempenho.

### 7.2 Trabalhos futuros

- **Implementar estratégia para reutilizar os containers Serverless instanciados em várias requisições**
  Assim como os principais provedores de Serverless, é interessante adicionar ao sl-handler uma estratégia para atender requisições simultâneas e sucessivas. Isso pode levar o sl-handler a responder as requisições com melhor desempenho. No entanto, é necessário também investigar o que acontece com o consumo de recursos com cluster com essa mudança de abordagem e discutir novamente a adequação ao estado da arte.

- **Alterar o módulo Docker.go do sl-handler para utilizar outra forma de comunicação com o Docker-engine, que atualmente usa a API HTTP do Docker**
  Atualmente o sl-handler utiliza comunicação HTTP com o Docker-engine, tanto para gerenciar as imagens Docker e os *containers*, quanto para enviar aos *containers* suas requisições e ouvir suas respostas. Uma possibilidade de trabalho futuro é propor outra forma de comunicação entre esses módulos,

como, por exemplo, um volume Docker compartilhando arquivos com trocas de mensagens entre os Serverless e o próprio sl-handler.

- **Implementar Dashboard para acompanhar o consumo de recursos do sl-handler e dos Serverless.**
  Os principais provedores de Serverless cobram pelo serviço a partir de SLA.
  De forma que para eles é essencial monitorar o consumo de recursos computacionais dos Serverless. Adicionar essa feature ao sl-handler o torna mais próximo do estado da prática e permite que seja feito trabalhos de investigação mais profundo quando ao desempenho e consumo de recursos computacionais.

- **Investigar as falhas encontradas nas requisições dos experimentos no sl-handler.**
  A Tabela 7 e a Tabela 8 apresenta algumas requisições que falharam no sl-handler. Embora já sabemos que foi devido à restrição em código, que determina que um *container* tem no máximo 2 segundos para estar pronto para atender à requisição, cabe investigar o que levou os *containers* a ultrapassarem esse limite nas requisições simultâneas.

## 8. Referências

[1] [Hayes, Brian. (2008) “Cloud computing”. In: Communications of the ACM, vol. 51, 2008, p.9.].

[2] [JACOBSON, Daniel; BRAIL, Greg; WOODS, Dan. APIs: A Strategy Guide. 1. ed. Estados Unidos da América: O‟Reilly Media, 2012. 136 p]

[3] [Alhamad, Mohammed, Tharam Dillon, and Elizabeth Chang. "Conceptual SLA framework for cloud computing." Digital Ecosystems and Technologies (DEST), 2010 4th IEEE International Conference on. IEEE, 2010.]

[4] Mell, Peter, and Tim Grance. "The NIST definition of cloud computing." (2011).

[5] Sommerville, Ian. Software Engineering. 9th Edition. Addison Wesley, 2010.

[6] Ian Sommerville, Dave Cliff, Radu Calinescu, Justin Keen, Tim Kelly, Marta Kwiatkowska, John Mcdermid, and Richard Paige. 2012. Large-scale complex IT systems. Commun. ACM 55, 7 (July 2012), 71-77.

[7] Large Scale Complex IT Systems, The UK Research & Training Initiative, URL: http://lscits.cs.bris.ac.uk, Last access 19-Sept-2012.

[8] Chieu, Trieu C., et al. "Dynamic scaling of web applications in a virtualized cloud computing environment." E-Business Engineering, 2009. ICEBE'09. IEEE International Conference on. IEEE, 2009.

[9] Dillon, Tharam, Chen Wu, and Elizabeth Chang. "Cloud computing: issues and
challenges." Advanced Information Networking and Applications (AINA), 2010 24th IEEE International Conference on. Ieee, 2010.

[10] Papazoglou, M. P. (2003) Service-Oriented Computing: Concepts, Characteristics and Directions, 4th Intl. Conf. on Web Information Sys. Engineering (WISE'03), 2003.

[11] Serverless Architectures - https://martinfowler.com/articles/serverless.html - acessado em 15/11/2018

[12] Docker Containerization Unlocks the Potential for Dev and Ops - https://www.docker.com/why-docker- acessado em 15/11/2018

[13] IaaS, PaaS e SaaS: entenda os modelos de nuvem e suas finalidades - http://brasil.softlinegroup.com/iaas-paas-saas-nuvem/ - acessado em 16/11/2018

[14] Exemplos de Computação em nuvem: SAAS, IAAS e PAAS - https://ecoit.com.br/exemplos-de-computacao-em-nuvem-saas-iaas-e-paas/ - acessado em 16/11/2018

[15] "Release: AWS Lambda on 2014-11-13". *Amazon Web Service*. Retrieved 26 February 2017.

[16] Thomsen, Jakob (January 19, 2016). "Using Triggers On Schemaless, Uber Engineering's Datastore Using MySQL".

[17] What are Microservices? - https://pivotal.io/microservices - acessado em 16/11/2018

[18] Josef Spillner, 2017. “Snafu: Function-as-a-Service (FaaS) Runtime Design and Implementation”

[19] O que é arquitetura Serverless (sem servidor)? - http://viniciusgarcia.me/development/o-que-eh-arquitetura-serverless/ - acessado em 16/11/2018

[20] The Benefits of Serverless Computing and its Impact on DevOps - acessado em 16/11/2018 - https://hackernoon.com/the-benefits-of-serverless-computing-and-its-impact-on-devops-e75d82c47ac4 - acessado em 16/11/2018

[21] What is Serverless Architecture? What are its Pros and Cons? - https://hackernoon.com/what-is-serverless-architecture-what-are-its-pros-and-cons-cc4b804022e9 - acessado em 16/11/2018

[22] Rodrigo da Rosa Righi, Revista Brasileira de Computação Aplicada (ISSN 2176-6649), Passo Fundo, v. 5, n. 2, p. 2-17, out. 2013 2. “Elasticidade em cloud computing: conceito, estado da arte e novos desafios”

[23] BEERNAERT, L.; MATOS, M.; VILACA, R.; OLIVEIRA, R. Automatic elasticity in openstack. In: . SDMCMM ’12. New York, NY, USA: ACM, c2012. p. 2:1–2:6.

[24] RAVEENDRAN, A.; BICER, T.; AGRAWAL, G. A framework for elastic execution of existing

[34] RAJAN, D.; CANINO, A.; IZAGUIRRE, J. A.; THAIN, D. Converting a high performance application to an elastic cloud application. In: . CLOUDCOM ’11. Washington, DC, USA: IEEE Computer Society, c2011. p. 383–390.

[35] Baldini, Ioana, et al. "Serverless computing: Current trends and open problems." Research Advances in Cloud Computing. Springer, Singapore, 2017. 1-20.

[36] ASAF YIGAL, 2017. “Should You Go ‘Serverless’? The Pros and Cons “

[37] O que são nuvens públicas, privadas e híbridas? - https://azure.microsoft.com/pt-br/overview/what-are-private-public-hybrid-clouds/ - acessado em 16/11/2018

[38] Nuvem pública, privada ou híbrida? Entenda as diferenças - https://computerworld.com.br/2017/06/14/nuvem-publica-privada-ou-hibrida-entenda-diferencas/ - acessado em 16/11/2018

[39] Uso de medições com GQM (Goal Question Metric) no SCRUM - http://blog.myscrumhalf.com/2013/08/uso-de-medicoes-com-gqm-goal-question-metric-no-scrum/ - acessado em 17/11/2017

[40] Van Solingen, Rini, et al. "Goal question metric (gqm) approach." Encyclopedia of software engineering (2002).

[41] O que é e para quem é indicado a Serverless Computing - https://canaltech.com.br/infra/o-que-e-e-para-quem-e-indicado-a-serverless-computing-119782/ - acessado em 17/11/2018

[42] Serverless Comparison: Lambda vs. Azure vs. GCP vs. OpenWhisk - https://www.twistlock.com/2018/07/10/serverless-comparison-lambda-vs-azure-vs-gcp-vs-openwhisk/ - acessado em 17/11/2018

[43] Serverless Framework Alternatives - https://siftery.com/serverless-framework/alternatives - acessado em 17/11/2018

[44] AWS Lambda - https://aws.amazon.com/pt/lambda/ - acessado em 17/11/2018

[45] Google Cloud Platform - https://cloud.google.com/gcp/ - acessado em 17/11/2018

[46] Azure Functions - https://azure.microsoft.com/pt-br/services/functions/ - acessado em 17/11/2018

[47]
Serverless Reference Architecture for creating an IoT Backend
- https://github.com/aws-samples/lambda-refarch-iotbackend - acessado 17/11/2018

[48] Things Go Better With Step Functions - https://aws.amazon.com/pt/blogs/aws/things-go-better-with-step-functions/ - acessado 17/11/2018

[49] iRobot Ready to Unlock the Next Generation of Smart Homes Using the AWS Cloud - https://aws.amazon.com/pt/solutions/case-studies/irobot/ - acessado 17/11/2018

[50] Benchling Case Study - https://aws.amazon.com/pt/solutions/case-studies/benchling/ - acessado em 17/11/2018

[51] Thomson Reuters Case Study - https://aws.amazon.com/pt/solutions/case-studies/thomson-reuters/ - acessado em 17/11/2018

[52] CarMax drives online innovation in the cloud - https://customers.microsoft.com/pt-br/story/carmax - acessado em 18/11/2018

[53] Preluxe case - https://customers.microsoft.com/pt-br/story/plexure-dev-azure - acessado em 18/11/2018

[54] Fujifilm case - https://customers.microsoft.com/pt-br/story/fujifilm-software-co-ltd - acessado em 18/11/2018

[55] AWS Lambda vs Azure Functions vs Google Cloud Functions: Comparing Serverless Providers - https://www.simform.com/aws-lambda-vs-azure-functions-vs-google-functions/ - acessado em 19/11/2018

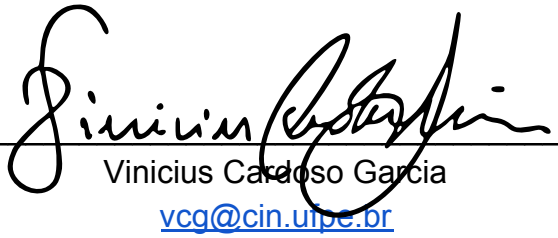
[56] AWS Lambda vs. Azure Functions vs. Google Functions - https://logz.io/blog/serverless-guide/ - acessado em 19/11/2018

[57] Microsoft Azure Functions vs. Google Cloud Functions vs. AWS Lambda - https://cloudacademy.com/blog/microsoft-azure-functions-vs-google-cloud-functions-fight-for-serverless-cloud-domination-continues/ - acessado em 19/11/2018

[58] Perguntas frequentes sobre o AWS Lambda - https://aws.amazon.com/pt/lambda/faqs/ - acessado em 19/11/2018

## 9. Assinaturas

Ricardo Robson Mendes da Silva
rrms@cin.ufpe.br
Orientando



Vinicius Cardoso Garcia
vcg@cin.ufpe.br
Orientador