Universidade Federal de Pernambuco
Centro de Informática

Graduação em Ciência da Computação

Serverless Broker

Sergio Torres Teixeira Filho (sttf@cin.ufpe.br)

Trabalho de Graduação

Recife, Julho de 2021

Universidade Federal de Pernambuco

Centro de Informática

Graduação em Ciência da Computação

Serverless Broker

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
Coorientador: Ricardo Robson Mendes da Silva

Recife, Julho de 2021

Aos meus pais, Sergio e Karin.
Aos meus irmãos, Matheus e Arthur.
Aos meus avós, José Airton, Eliane,Volgran
e Julia (in memoriam).

Agradecimentos

Primeiramente, eu expresso minha imensa gratidão a minha família que sempre me apoiou e me deu o suporte necessário para que eu realizasse este trabalho.

Gostaria de agradecer também a todo corpo docente do Centro de Informática da Universidade Federal de Pernambuco, por me incentivar a sempre buscar evoluir tanto como profissional quanto como ser humano. Em especial ao meu professor e orientador Vinicius Garcia que sempre me apoiou e acreditou em mim e ao professor Leopoldo Teixeira por se disponibilizar a fazer parte da banca avaliadora deste trabalho.

Agradeço também a todos colegas alunos que me acompanharam nessa incrível jornada, aprendemos juntos, sofremos juntos e nos divertimos juntos. Em especial a turma de 2016.1 da qual faço parte e a Ricardo Robson Mendes que se dispôs a ser meu co-orientador neste trabalho.

E finalmente sou grato a todas as pessoas que de alguma forma me ajudaram a ser quem sou hoje, sem elas nada disso seria possível.

Resumo

O avanço da tecnologia e serviços tecnológicos sob demanda, juntamente com a crescente inclusão digital, transformaram a maneira como a sociedade funciona assim como seus serviços. *Cloud Computing* vem dominando o mercado de tecnologia por apresentar diversas vantagens e facilidades para as empresas. Um modelo de *Cloud Computing* que demonstra grande potencial de crescimento é o *Serverless Computing*, que pode ser definido como o provisionamento de funções como serviço, que seguem alguns padrões como ser stateless, ter responsabilidade única e duração efêmera. Os maiores provedores de *Cloud Computing* já oferecem serviços de *Serverless Computing* como a AWS Lambda, Azure Functions e a Google Cloud Functions. Um problema conhecido no desenvolvimento de sistemas serverless usando estes grandes provedores é a falta de padronização na tecnologia, assim cada provedor tem sua abordagem e consequentemente o sistema possui vendor lock-in. Este trabalho busca a implementação de um broker que seja capaz de portar funções escritas em Javascript no formato do Express.js para o formato específico aceito por cada um dos provedores escolhidos(AWS e Google Cloud) e fazer o processo de implantação das mesmas.

Palavras-chave: Serverless, Broker, Cloud Computing, Function as a Service.

Abstract

The advancement of technology and technological services on demand, together with the growing digital inclusion, have transformed the way society works as well as its services. Cloud Computing has dominated the technology market as it presents several advantages and facilities for companies. A Cloud Computing model that demonstrates great growth potential is Serverless Computing, which can be defined as the provisioning of functions as a service, which follow certain standards such as being stateless, having sole responsibility and ephemeral duration. The largest providers of Cloud Computing already offer Serverless Computing services such as AWS Lambda, Azure Functions and Google Cloud Functions. A known problem in the development of serverless systems using these large providers is the lack of standardization in the technology, so each provider has its own approach and consequently the system has a vendor lock-in. This work aims to implement a broker that is capable of porting functions written in Javascript in the Express.js format to the specific format accepted by each of the chosen providers (AWS and Google Cloud) and deploy the functions.

Keywords: Serverless, Broker, Cloud Computing, Function as a Service.

Lista de tabelas

Lista de figuras

Sumário

1 Introdução

1.1 Contexto e motivação

A sociedade atual está cada vez mais conectada, principalmente por conta do aumento no acesso à tecnologia e da facilidade de adoção da mesma. Segundo o Pew Research Center, estimava-se que no ano 2000 aproximadamente metade dos adultos americanos usavam a internet, em contrapartida ao ano de 2021 onde esse percentual chegou a cerca de noventa e três por cento [1].

A crescente integralização digital demanda que a infraestrutura que a sustenta também acompanhe esse crescimento, tanto com inovações tecnológicas quanto com aumento de capital humano. Empresas precisaram se adaptar para atender as necessidades de seus clientes, assim passaram a informatizar seus serviços e fazer uso da tecnologias para aumentar sua produtividade.

Um dos principais custos que diversas empresas tinham ao se modernizar e adentrar nesse mundo de serviços conectados à internet foi o de comprar e fazer a manutenção de várias máquinas usadas como *datacenters* e servidores, conhecido como modelo *on-premise*. Dado esse contexto surgiu uma forma de serviço inovadora que mudou como as empresas criam e gerenciam seus *softwares* e sistemas, chamada *Cloud Computing*. Com a ajuda de inovações tecnológicas como a virtualização [2], tornou-se possível a entrega de serviços de tecnologia e infraestrutura sob demanda. Este modelo traz muitas comodidades para seus clientes, onde as responsabilidades de manutenção, segurança, provisionamento podem ser terceirizadas para o provedor do serviço [3].

Para garantir resiliência e escalabilidade, os sistemas e aplicações modernas passaram a adotar novas arquiteturas, dividindo de forma cada vez mais granular o domínio de cada módulo. A arquitetura monolítica(previamente predominante no ecossistema de *enterprise software*), que consistia em um modelo arquitetural onde os módulos não podem ser separados ou executados de forma independente, demonstrou gerar uma complexidade grande de manutenção e evolução à medida que o sistema cresce [4].

Um modelo de *Cloud Computing* em específico vem ganhando a atenção do mercado é o *Serverless Computing*, que tem como base tornar a configuração e uso de servidores transparente para o desenvolvedor, ao contrário do que a tradução literal do inglês sugere, que seria algo como "sem servidor", na verdade o objetivo seria abstrair essa preocupação com o servidor [5]. Uma aplicação Serverless funciona de forma elástica, alocando recursos e escalando sob demanda, sendo cobrada apenas pelos recursos utilizados.

Uma plataforma de *Serverless Computing* ou modelo de serviço *Serverless* em destaque seria a *FaaS*[1] (*Function as a Service*) onde a unidade computacional consiste em funções que são executadas após determinados gatilhos, como por exemplo uma requisição HTTP. Essa plataforma vem desafiando a maneira como se constroem sistemas, com uma mudança na filosofia de uma aplicação orientada a serviços para uma orientada a eventos [6].

Os maiores provedores de serviços de *Cloud Computing* já oferecem serviços no modelo *FaaS* como a AWS (Amazon Web Services) [7], a GCP (Google Cloud Platform) [8], Microsoft Azure [9] a IBM Cloud [10]. Sendo seus serviços respectivamente: AWS Lambda [11], Google Cloud Functions [12], Azure Functions [13] e IBM Cloud Functions [14]. Um problema comum no universo de *Cloud Computing* é o fenômeno chamado de *vendor lock-in* que consiste na forte dependência no provedor a ponto de não ser possível a mudança do mesmo sem que existam custos significativos para essa migração, sendo assim os serviços citados anteriormente não são exceção, na realidade o grau de dependência é ainda maior para os serviços *FaaS*. Existem diversos fatores que influenciam no *vendor lock-in* nestes serviços, dentre eles podemos citar a interface das funções, a estrutura do eventos recebidos, suporte a diferentes linguagens de programação dentre outros [6].

Os modelos de Serverless Computing e FaaS ainda são muito recentes e por conta disso não existe um padrão na indústria e na comunidade de como a plataforma deve funcionar, fazendo com que existam interfaces proprietárias e distintas para cada provedor. Não obstante, torna-se necessário que o engenheiro

[1] https://www.ibm.com/cloud/learn/faas

de software lide com essas diferenças em seu processo de desenvolvimento quando há necessidade do uso de diferentes provedores.

Esta falta de padronização faz com que aplicações que se beneficiam de ambientes Multi-Cloud tenham dificuldade e receio de adotar a arquitetura Serverless. Aplicações como Sistemas Críticos, por exemplo um sistema de infraestrutura hospitalar, se beneficiariam de propriedades tanto da Arquitetura Serverless como a alta capacidade de escalonamento, quanto de ambientes Multi-Cloud, como a redundância de serviços e dados em diferentes provedores e a diminuição de dependência em um único provedor.

1.2 Objetivos gerais

Este trabalho tem como objetivo investigar os desafios do desenvolvimento de soluções *Multi-Cloud* dos serviços FaaS e *Serverless*, especificamente os serviços da AWS Lambda, Azure Functions, Google Cloud e também propor uma ferramenta que permita a construção de funções *Serverless* de maneira agnóstica ao provedor.

1.3 Objetivos específicos

- Desenvolver uma ferramenta para desenvolvimento de funções *Serverless* agnóstico ao provedor
- Prover uma solução alternativa viável e compatível com os serviços FaaS: AWS Lambda e Google Cloud Functions

1.4 Organização do trabalho

- Capítulo 2: introduz os conceitos e tecnologias fundamentais para o entendimento do trabalho e uma breve revisão do estado atual do desenvolvimento *Multi-Cloud* de funções *Serverless.*
- Capítulo 3: apresenta o processo de elaboração e idealização do projeto, juntamente com seu funcionamento e arquitetura final.
- Capítulo 4: apresenta o processo metodológico, as experimentações feitas e discute seus resultados.

- Capítulo 5: apresenta a conclusão do trabalho, suas possíveis limitações e as perspectivas de trabalhos futuros.
- Capítulo 6: referências bibliográficas.

2. Revisão da literatura

Este capítulo busca apresentar os conceitos fundamentais para entendimento do trabalho e compreensão do contexto atual no qual ele faz parte.

2.1 Conceitos

2.1.1 *Cloud Computing*

Cloud Computing pode ser definida como um modelo de computação que envolve diversos tipos de serviços computacionais como armazenamento de dados, gerenciamento de servidores e aplicações, estes são oferecidos sob demanda e de maneira remota [15]. Dentro do universo de *Cloud Computing* existem algumas ramificações de serviços que entregam diferentes funcionalidades para o cliente, como principais podem ser citadas a IaaS, SaaS, PaaS e a FaaS.

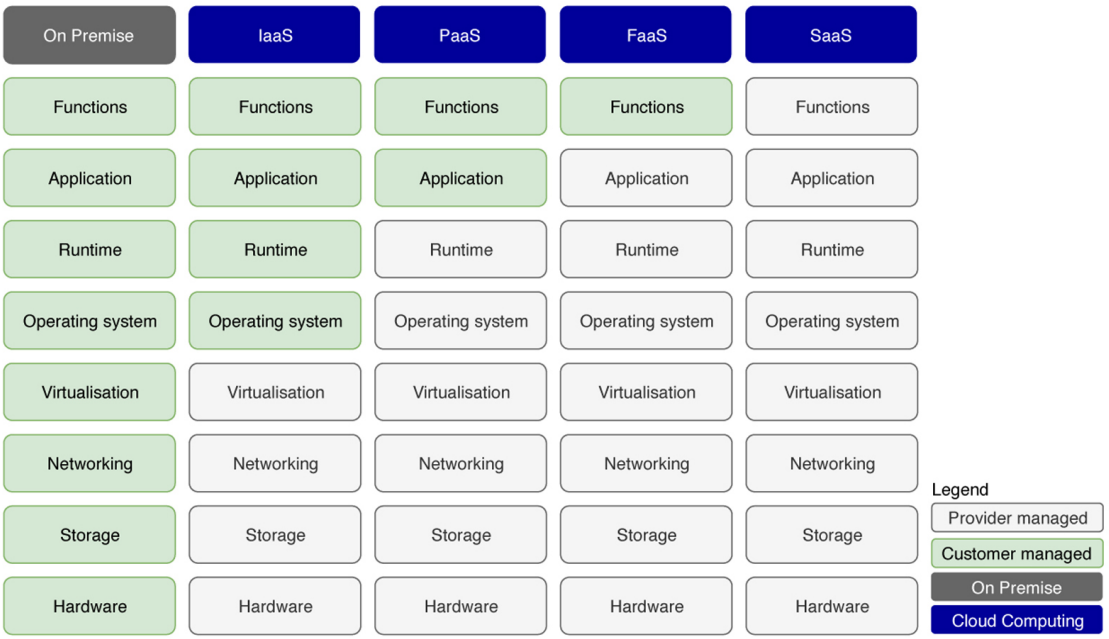

Figura 1 - Short comparison: On premise vs. IaaS vs. PaaS vs. FaaS vs. SaaS [16]

As principais características do modelo *Cloud Computing* geralmente consistem em prover serviços computacionais sob demanda, de maneira escalável, com acesso remoto e alta disponibilidade.

A adoção de *Cloud Computing* permitiu que empresas evoluíssem sua infraestrutura com muito mais facilidade, terceirizando muitos serviços que antes eram sua responsabilidade e tendo acesso a inovações tecnológicas que os provedores oferecem [3, 15].

2.1.2 *Infrastructure as a Service* (IaaS)

O modelo de IaaS ou infraestrutura como serviço oferece ao cliente um ambiente virtualizado no qual ele tem controle apenas do sistema operacional e suas camadas superiores, abstraindo sua preocupação com a infraestrutura física e as camadas de mais baixo nível. Geralmente, os provedores de serviços IaaS oferecem uma plataforma de gerenciamento onde seus clientes podem definir as especificações técnicas do *hardware* alocado para o serviço, como também configurações de rede [17].

A interface de virtualização que é usada mais comumente na indústria de IaaS é na forma de *Containers*, mas também pode ser oferecida na forma de máquinas virtuais menos modernas.

2.1.3 *Platform as a Service* (PaaS)

O modelo de PaaS adiciona uma camada a mais de abstração em comparação ao IaaS, abstraindo a preocupação com o sistema operacional ou ambiente de virtualização, fazendo com que o cliente consiga focar apenas na camada de aplicação. Esse modelo de Cloud Computing permite que empresas de software construam aplicações com mais facilidade, mas ainda sim com certa flexibilidade e configurabilidade [17].

Exemplos populares de serviços PaaS que podem ser citados seriam o Google App Engine [18] e o AWS Elastic Beanstalk [19], pois ambos fornecem

ambientes programáveis e customizáveis para o desenvolvimento de aplicações na nuvem.

2.1.4 *Software as a Service* (SaaS)

O modelo de SaaS consiste em serviços de softwares ou aplicações completas disponíveis na nuvem. A maioria dos serviços SaaS são aplicações web onde o cliente paga para ter acesso a suas funcionalidades, sem ter que se preocupar com toda infraestrutura por trás desse sistema e geralmente possui apenas algumas pequenas configurações específicas de usuário [17].

Essa modalidade de serviço de *Cloud Computing* já é amplamente utilizada na sociedade, alguns exemplos populares seriam plataformas de email como Gmail[20], redes sociais como o Facebook [21] e ferramentas colaborativas online como o Google Docs [22].

2.1.5 *Function as a Service* (FaaS) e *Serverless Computing*

Os conceitos de FaaS e *Serverless Computing* estão bastante interligados, o que acaba fazendo com que autores usem os dois nomes para expressar a mesma ideia, além do fato da tecnologia ser recente e não estar tão bem padronizada ainda, intensificando essa confusão.

Serverless Computing é um conceito mais amplo que se refere ao próximo passo de evolução na *Cloud Computing*, seus serviços abstraem a preocupação com o servidor de maneira que a responsabilidade de gerenciar isso passa para o provedor. Já o modelo FaaS é uma implementação do *Serverless Computing* que potencializa a abstração de servidores ao fragmentar a aplicação em funções escaláveis e gerenciadas pelo provedor [6].

O modelo FaaS consiste de uma plataforma de *Serverless Computing* onde a unidade computacional é uma função, esta segue alguns princípios *Serverless* como ser *stateless*, ter duração efêmera e ser acionada por algum gatilho como por exemplo uma requisição HTTP ou uma inserção em um banco de dados. Respeitando estes princípios, resulta em um modelo que consegue um alto grau de escalabilidade, caso as funções necessitem de manter algum tipo de estado seria necessário usar algum serviço externo [6].

A principal diferença entre *Serverless Computing* e as outras formas de *Cloud Computing* está em como o serviço é entregue ao cliente, se existir algum tipo de preocupação por parte do cliente em como escalar, provisionar e gerenciar o servidor e seus processos, o serviço não se encaixaria nos padrões *Serverless* [5]. Alguns exemplos de serviços *Serverless* que não são caracterizados como FaaS seriam o AWS Cognito [23], Firebase Cloud Firestore [24] e Auth0 [25].

2.1.6 *Multi-Cloud*

O termo *Multi-Cloud* se refere ao uso de soluções e serviços de *Cloud Computing* provenientes de múltiplos provedores ou nuvens. Esse modelo arquitetural tem sido bastante utilizado na indústria [26] por possuir algumas vantagens. Dentre os principais motivos para a adoção do Multi-Cloud podem ser citados [27]:

- Diminuir a dependência em apenas um provedor,
- Replicar serviços e aplicações em diferentes clouds para garantir maior resiliência,
- Fazer uso de serviços específicos particulares a um provedor,
- Otimização de custos ou melhorar qualidade dos serviços.

2.1.7 *Cloud Broker*

Com a complexidade das plataformas de *Cloud Computing*, pode se tornar difícil para um cliente integrar e gerenciar os serviços oferecidos. Um *Cloud Broker* é uma entidade que gerencia o uso, performance e entrega de serviços *Cloud*, além de administrar os relacionamentos entre provedor *Cloud* e consumidor. O Cloud Broker passa a ser o intermediário entre o cliente e a plataforma *Cloud* [28].

Em geral, um Cloud Broker pode prover serviços em três categorias [28]:

- *Service Intermediation*: Nesta categoria os *Cloud Brokers* focam em aperfeiçoar alguma capacidade específica de um serviço e provê esse valor

agregado aos consumidores. O aperfeiçoamento pode ser no gerenciamento de acesso, gerenciamento de identidade, melhoria em segurança e etc.

- *Service Aggregation*: Nesta categoria os *Cloud Brokers* agregam e integram múltiplos serviços em um ou mais novos serviços. O *Broker* provê integração de dados e garante uma movimentação segura dos dados entre o consumidor e múltiplos provedores *Cloud*.
- *Service Arbitrage*: Nesta categoria os *Cloud Brokers* são similares aos de *Service Aggregation* com a diferença que não necessariamente os serviços vão ser agregados, dando liberdade ao *Broker* selecionar entre os serviços de diversos provedores. Um exemplo seria um *Broker* que mediria pontuações para cada serviço e selecionaria estes de acordo com essa medida.

2.2 Estado da arte do desenvolvimento *Multi-Cloud* de Funções *Serverless*

2.2.1 Introdução

As aplicações Serverless são predominantemente implantadas em nuvem públicas gerenciadas por provedores de Cloud Computing, estes tendem a ser semelhantes na forma como oferecem as tecnologias Serverless, sendo a sua principal diferença presente no ecossistema de serviços que sua infraestrutura oferece. As plataformas atuais apenas se preocupam com facilitar a integração de serviços dos seus próprios ecossistemas, fazendo com que o cliente fique preso aos serviços nativos daquela plataforma, ou em outras palavras, gerando o *vendor lock-in* [29, 30].

2.2.2 Frameworks

Tendo em vista este cenário, algumas soluções Open Source foram surgindo para mitigar esse problema que afeta o desenvolvimento de aplicações *Serverless* em ambientes *multi-cloud*. Dentre elas podemos citar o OpenFaaS [31], o Kubeless [32], o Fn Project [33] e o Apache OpenWhisk [34], estas plataformas buscam oferecer uma abordagem agnóstica não só para o desenvolvimento de aplicações *Serverless* em *Clouds* públicas mas também para seu provisionamento *on premise* com o uso de um orquestrador de *containers* como o Kubernetes [35] (O Apache

OpenWhisk não necessita de um orquestrador, embora seja compatível com o mesmo). Porém, essas plataformas estão longe de ser as mais utilizadas nos ecossistemas Serverless, além de não fazer uso dos serviços *FaaS* ofertados pelas provedoras de *Cloud Computing* [36, 37].

O Serverless Framework [38] é uma ferramenta bastante utilizada no processo de desenvolvimento e implantação de aplicações *FaaS* para diversos provedores [36, 37], simplificando o gerenciamento e manutenção dessas aplicações. Esta ferramenta permite integrações de plugins criados pela comunidade de desenvolvedores, provendo extensibilidade a suas funcionalidades.

2.2.3 Linguagens de Programação

Quanto a linguagem de programação, as mais utilizadas são Javascript (Node.js) e Python, além de serem linguagens populares [39], seus ambientes de *runtime* tem curta duração de *coldstart*(tempo necessário para a instância ser inicializada) [40]. Outro fator importante que também determinou a predominância destas linguagens foi que seus *runtimes* estavam disponíveis em quase todos os provedores.

2.2.4 Desafios

Quando se fala em desenvolvimento de aplicações *Serverless Multi-cloud*, existem diversos desafios envolvidos que devem variar dependendo do contexto dessas aplicações, mas alguns destes podem ser destacados [41]:

- Interfaces,
- Runtimes,
- Eventos,
- SDKs,
- Serviços Exclusivos.

Cada provedor de serviços *FaaS* tem uma interface de *handler* específica para cada runtime(linguagem de programação) que ele oferece, o *handler* nada mais é do que o método executado ao ser acionado por algum gatilho, como por exemplo uma chamada HTTP. A Tabela 1 ilustra as diferentes interfaces do handler para o runtime de Node.js para os principais provedores de *Cloud Computing*.

	Handler
AWS Lambda	*exports.handler = async function(event, context) {* * const response = {* * statusCode: 200,* * body: "hello world",* * };* * return response;* *}*
Azure Functions	*module.exports = async function (context, req) {* * context.res = {* * status: 200,* * body: "hello world"* * };* *};*
Google Cloud Functions	*exports.helloWorld = (req, res) => {* * res.status(200).send('hello world');* *};*

Tabela 1 - Versões do handler para cada provedor

Uma diferença comum entre os provedores Serverless está na oferta de runtimes disponíveis que podem variar não só em Linguagem de Programação mas também na versão da Linguagem utilizada. Alguns provedores como a AWS oferecem uma opção de runtime personalizado, onde é possível utilizar um runtime específico não suportado nativamente, porém no exemplo citado você precisa implementar a interface de runtimes utilizada pelo provedor. A Tabela 2 ilustra os diferentes runtimes aceitos pelos principais provedores.

	AWS Lambda	Azure Functions	Google Cloud Functions
Node.js (JavaScript)	Node.js 14 Node.js 12 Node.js 10	Node.js 14 Node.js 12 Node.js 10 Node.js 8 Node.js 6	Node.js 14 Node.js 12 Node.js 10 Node.js 8 Node.js 6
Java	Java 8 Java 11	Java 8 Java 11	Java 11

C#	.NET Core 2.1 .NET Core 3.1	.NET Framework 4.8 .NET Core 2.1 .NET Core 3.1 .NET 5.0	NET Core 3.1
Python	Python 3.9 Python 3.8 Python 3.7 Python 3.6	Python 3.9 Python 3.8 Python 3.7 Python 3.6	Python 3.9 Python 3.8 Python 3.7
PHP	-	Experimental	PHP 7.4
Go	Go 1.x	-	Go 1.13 Go 1.11
F#	-	.NET Core 3.1 .NET Core 2.1 .NET Framework 4.8	-
Swift	-	-	-
Ruby	Ruby 2.7 Ruby 2.5	-	Ruby 2.7 Ruby 2.6
Runtime Personalizado	Sim, com implementação de interface própria	Sim, com implementação de um servidor HTTP	-

Tabela 2 - Lista de linguagens suportadas por provedor de Serverless

Em plataformas de *Cloud Computing* existe uma gama de eventos que ocorrem, seja dentro de aplicações ou em em serviços oferecidos no ecossistema desta plataforma. Alguns destes eventos podem ser usados como *triggers* ou gatilhos para acionar e invocar funções Serverless. No entanto, estes eventos não são padronizados entre diferentes provedores, até pela diferença nos serviços oferecidos. Outro desafio também se encontra na integração de eventos externos à plataforma na invocação de funções. A Tabela 3 apresenta resumidamente os principais eventos aceitos como *triggers* para funções *Serverless* nos principais provedores de *Cloud Computing*.

	AWS Lambda	Azure Functions	Google Cloud Functions
HTTP	Sim	Sim	Sim
Storage	Sim	Sim	Sim
Database	Sim	Sim	Sim
Message Queue	Sim	Sim	Sim
Cron Job	Sim	Sim	Sim

Tabela 3 - Lista de eventos aceitos como trigger por cada provedor

No desenvolvimento de funções Serverless às vezes se torna necessário interagir com alguns recursos e serviços oferecidos pela plataforma de Cloud Computing, para isso estes provedores fornecem bibliotecas e SDKs que facilitam essa integração. Um desafio de aplicações Serverless em ambientes multi-cloud, é conciliar o uso dessas diferentes bibliotecas, com diferentes interfaces e funcionalidades.

E por fim, uma grande diferença entre os provedores de *Cloud Computing* está no ecossistema de serviços e soluções que ele oferece, esse rico portfólio de ferramentas proprietárias, geralmente gerenciadas pelo próprio provedor, são muito atrativas para seus clientes. Ambientes de desenvolvimento *multi-cloud* possuem o risco de depender de um provedor específico ao integrar um serviço exclusivo daquela plataforma.

2.3 Sumário do Capítulo

Neste capítulo, exploramos os conceitos por trás de Cloud Computing e seus principais modelos de oferta de serviços. Também vimos os conceitos relacionados a Multi-Cloud, elucidando as vantagens desse tipo de arquitetura, e Cloud Broker, apresentando suas diferentes categorias. Por fim, discutimos o estado da arte do desenvolvimento Multi-Cloud de Funções Serverless, apresentando o ecossistema desse meio e seus desafios.

Após o entendimento dos conceitos que envolvem o tema deste trabalho e o seu contexto atual, no próximo capítulo vamos expor a proposta de Solução e o processo de decisões que levaram a ela.

3. Projeto

Este capítulo tem como objetivo apresentar o processo de desenvolvimento e as tomadas de decisões feitas no decorrer do projeto, além de apresentar a versão final da solução desenvolvida.

3.1 Escolha da linguagem de programação

Para construir uma solução que seja relevante para o problema, analisamos o estado atual do ecossistema de aplicações Serverless para decidir em qual linguagem de programação(runtime) focar nossos esforços, uma vez que diferentes linguagens têm diferentes interfaces de handlers e suas próprias peculiaridades.

Entre as linguagens mais populares no ecossistema Serverless, se destacam Python e Node.js (Javascript) [36], como já citado no tópico 2.2.3 do capítulo 2. Estas linguagens são bastante populares também fora deste ecossistema [40].

A Figura 2 ilustra os *runtimes* mais populares para funções Serverless na AWS Lambda segundo pesquisa realizada pela empresa Datadog.

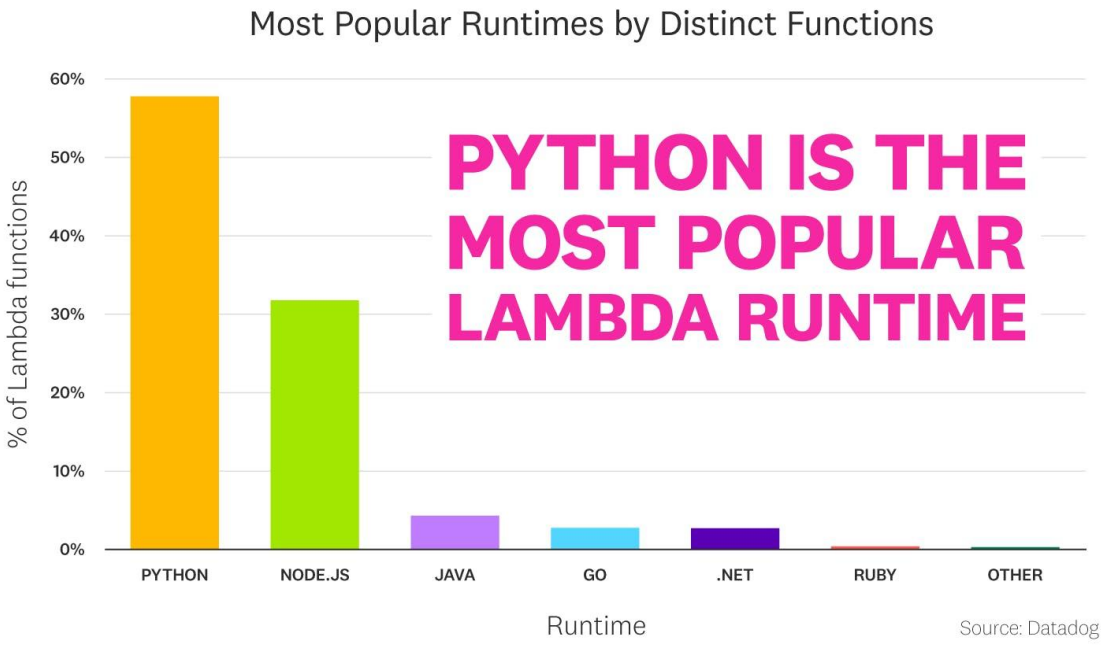

Figura 2 - Most Popular Runtime by Distinct Functions [36]

Como pode ser observado na Figura 2, Python e Node.js se destacam quando em comparação às outras linguagens.

Finalmente, dentre as duas opções foi selecionada a linguagem Node.js pois havia um conhecimento prévio do responsável pelo trabalho nesta linguagem.

3.2 Escolha dos provedores Serverless

No cenário atual, existem diversos provedores de *Cloud Computing* que oferecem serviços *Serverless* e *FaaS*, dentre os quais podemos destacar Amazon Web Services (AWS)[2], Microsoft Azure[3], Google Cloud Platform[4] e IBM Cloud[5] como sendo as principais plataformas deste ecossistema [36, 37]. A Figura 3 apresenta a popularidade dos principais provedores no ecossistema *Serverless*.

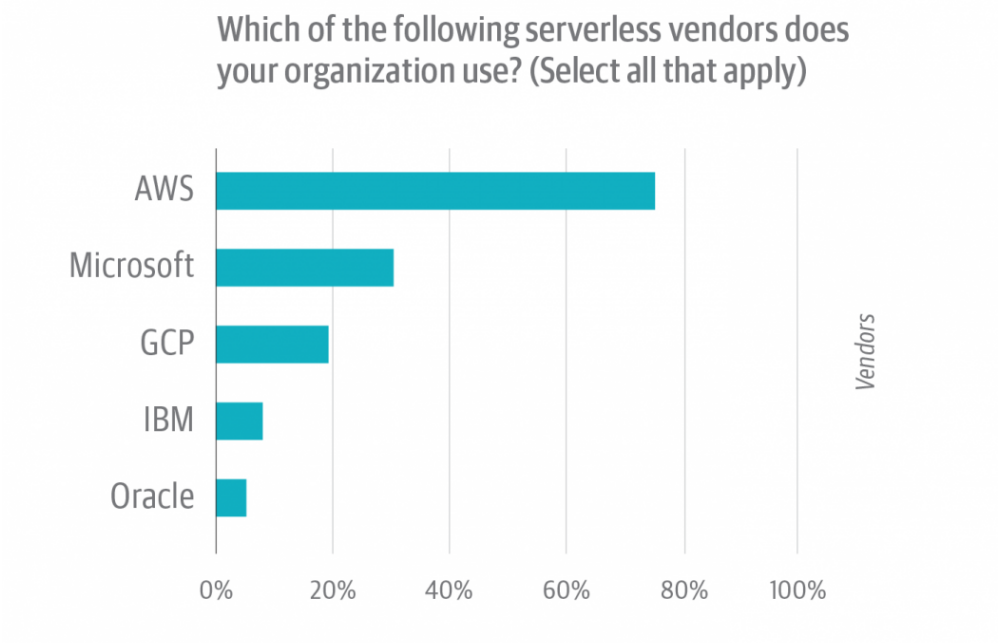

Figura 3 - Serverless vendors used by survey respondents' organizations.[37]

[2] https://aws.amazon.com/
[3] https://azure.microsoft.com/
[4] https://console.cloud.google.com/
[5] https://cloud.ibm.com/

Inicialmente foram selecionados os três grandes players do mercado, *AWS*, *Microsoft Azure* e *Google Cloud Platform*, sendo seus serviços de FaaS respectivamente *AWS Lambda*, *Azure Functions* e *Google Cloud Functions*. Todas estas plataformas possuem um *runtime* com Node.js.

Todavia, o serviço *Serverless Azure Functions* funciona de uma maneira um pouco diferente do *AWS Lambda* e *Google Cloud Functions*, ele tem uma funcionalidade bastante única chamada *Bindings* ou Associações que são uma maneira de você configurar fluxos de dados independente do trigger utilizado por aquela função, permitindo você receber ou enviar dados de serviços do ecossistema [43]. Essa complexidade a mais, juntamente com a dificuldade que o autor encontrou ao tentar mapear o objeto de evento HTTP fornecido pela Azure para a estrutura do objeto Request do framework Express.js, fez com que este provedor não tenha sido escolhido para o escopo deste trabalho. Muito embora seja sim possível e viável a sua inclusão no futuro, uma vez que o fator tempo para realização do trabalho foi o que obrigou este escopo a ser limitado.

3.3 Solução proposta

3.3.1 Cloud Broker

Após as escolhas da linguagem de programação e do provedor *Serverless*, foi idealizado construir a solução na forma de um *Cloud Broker*, por este formato possuir algumas características que se encaixam na nossa proposta, como prover interoperabilidade na *Cloud*, portabilidade na *Cloud* e reduzir a dependência em apenas um provedor *Cloud* [44]. O *Cloud Broker* implementado neste projeto foi estruturado para ser executado no ambiente local de desenvolvimento, embora este padrão também possa ser implementado para ser executado na *Cloud*.

3.3.2 Interface das Funções

Outra decisão importante, seria a escolha da interface usada por nossa solução, para escrita da aplicação *Serverless*. Existiam, em termos gerais, duas alternativas: escolher utilizar alguma das interfaces já usadas por um dos provedores, ou estabelecer uma diferente destas já fornecidas pelos provedores.

Para não mudar a maneira como desenvolvedores web constroem aplicações usando Node.js, foi escolhido usar a interface do framework web mais popular deste ecossistema, o Express.js [39].

```
1  const express = require('express')
2  const app = express()
3  const port = 3000
4
5  app.get('/', (req, res) => {
6    res.send('Hello World!')
7  })
8
9  app.listen(port, () => {
10   console.log(`Example app listening at http://localhost:${port}`)
11 })
```

Figura 4 - Exemplo de uma aplicação Express.js e sua interface

3.3.3 Ferramental utilizado

Para portar aplicações que usam a interface do Express.js para o formato aceito pelos provedores, foi usada a biblioteca Open Source chamada *serverless-http* [45] que engloba a aplicação Express.js, fazendo o roteamento dos endpoints e a conversão do objeto recebidos para o formato aceito pelo framework, isto para o caso da AWS Lambda. Já no caso do Google Cloud Function, foram aproveitados a compatibilidade do framework usado pela infraestrutura da Google Cloud Platform, o Functions Framework [46], este usa o Express.js em sua implementação, permitindo uma integração dessa interface com poucas modificações no código.

Toda alteração no código executada pelo *Broker* é feita por uma ferramenta Open Source chamada *jscodeshift* [47], desenvolvida internamente pela empresa Facebook, ela permite a execução de *codemods* ou alterações na AST(*Abstract Syntax Tree*) de arquivos Javascript para executar transpilações (*source-to-source*

compilations). Dessa forma o *Broker* consegue alterar o código necessário para adaptar a aplicação para o formato aceito pelo provedor. A Figura 5 ilustra um *codemod* simples que renomeia uma variável com nome “foo” para o nome “bar”.

```
module.exports = function(fileInfo, api) {
  return api.jscodeshift(fileInfo.source)
    .findVariableDeclarators('foo')
    .renameTo('bar')
    .toSource();
}
```

Figura 5 - Exemplo de um codemod para renomear uma variável.

Por fim, para integrar ao Broker uma infraestrutura de implantação das funções *Serverless*, foi utilizado a ferramenta Open Source chamada Serverless Framework [38], que além de ser popular entre os desenvolvedores de Aplicações *Serverless* [36, 37], ela permite extensibilidade para diferentes funcionalidades através de plugins e de maneira agnóstica.

3.3.4 Arquitetura da Solução

O Cloud Broker [28] foi desenvolvido como um plugin do Serverless Framework, integrando todos os componentes citados nos tópicos 3.3.1, 3.3.2 e 3.3.3. Este plugin tem dois comandos essenciais, o de compilação e o de implantação. O comando de compilação gera os arquivos do projeto no formato de cada provedor especificado. O comando de implantação executa a implantação na plataforma *Cloud* de cada provedor especificado.

A Figura 6 ilustra a arquitetura do Broker e como ele se encaixa no fluxo de desenvolvimento.

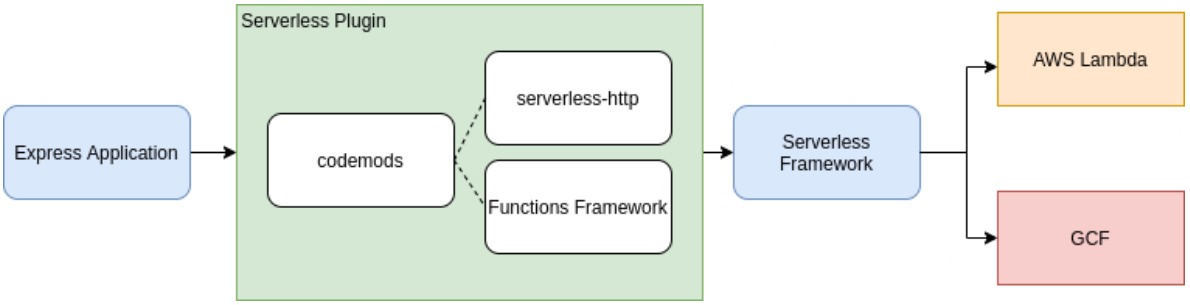

Figura 6 - Arquitetura do serverless-agnostic.

Podemos notar na Figura 6 que aplicação Express.js usa o Broker para gerar os formatos específicos de cada provedor e por fim implantar com o Serverless Framework em cada provedor.

A Figura 7 ilustra a arquitetura interna do serverless-agnostic, ilustrando os três módulos de serviço.

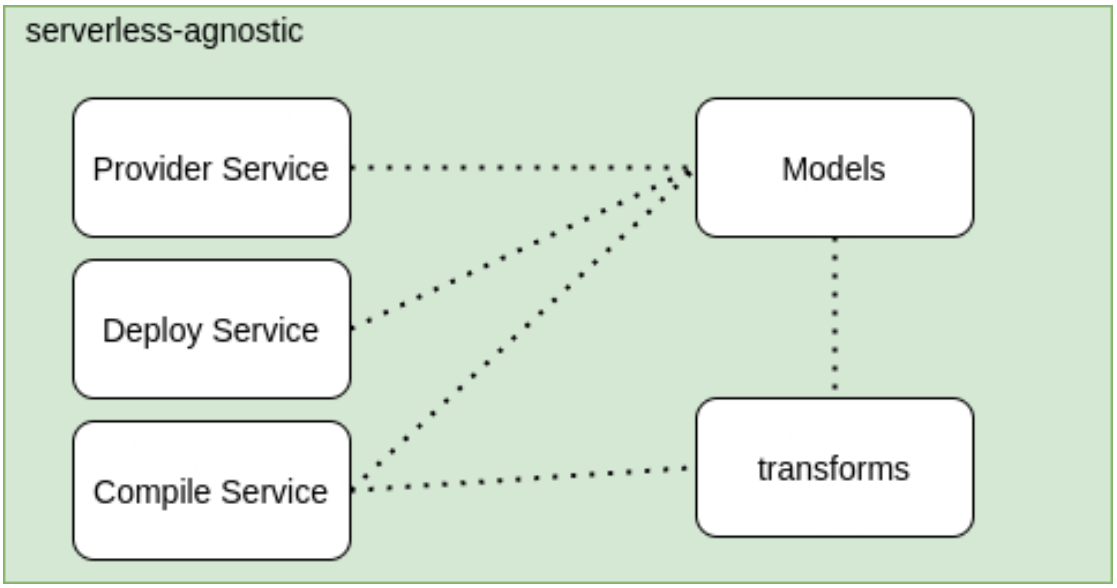

Figura 7 - Arquitetura interna do serverless-agnostic.

O *Compile Service*, responsável pela compilação do projeto para o formato aceito por cada provedor, o *Deploy Service*, responsável pela implantação dos

projetos gerados para cada provedor e por fim o *Provider Service* responsável por armazenar as informações dos provedores. Ainda na Figura 7, o módulo *transforms* armazena os *codemods* necessários para compilação.

3.4 Configuração e Uso da Ferramenta

Para instalação da ferramenta, o usuário precisa ter instalado algum gerenciador de pacotes para Node.js como *npm* ou *yarn*. Após navegar com o terminal até o arquivo do projeto escolhido, executar o seguinte comando:

- **npm install serverless-agnostic**

Para utilizar a ferramenta *serverless-agnostic* você precisa de um arquivo nomeado *.serverless.yml* que vai conter as configurações do plugin, assim como também outros arquivos no formato yaml para especificar as configurações do Serverless Framework de cada provedor. A Figura 8 abaixo apresenta um exemplo da configuração necessária do plugin para o arquivo *.serverless.yml* principal.

```yaml
custom:
  serverlessAgnostic:
    serverFile: index.js
    handlerName: myHandler
    serverName: app
    providers:
      - name: aws
        slsFilePath: serverless.aws.yml
      - name: gcp
        slsFilePath: serverless.gcp.yml
```

Figura 8 - Exemplo de arquivo .serverless com o plugin configurado.

Na Figura 8 o campo *serverlessAgnostic* é o objeto onde ficam as configurações do plugin. O campo *serverFile* se refere ao arquivo que contém o

objeto do servidor Express.js, o campo *handlerName* se refere ao nome do *handler* ou nome da função que é exportada como *handler*, o campo *serverName* se refere ao nome da variável do servidor Express.js e por fim, o campo *providers* se refere aos provedores de *Cloud Computing*, onde para cada um você indica o nome e caminho para o arquivo de configuração do Serverless Framework.

O plugin possui dois comandos principais para seu funcionamento:

- **serverless compile:** compila o projeto para as versões específicas de cada provedor, deixando tudo pronto para a implantação.
- **serverless agnosticDeploy:** faz a implantação do projeto compilado para cada provedor configurado.

Por fim o plugin possui uma funcionalidade que opera como uma espécie de compilação condicional onde o desenvolvedor pode declarar que certos trechos de códigos vão ser compilados para apenas um provedor específico. Esta funcionalidade permite que o desenvolvedor faça uso de funcionalidades específicas de um provedor caso algum caso de uso ou circunstância exija isto. A Figura 9 mostra um trecho de código que faz uso desta funcionalidade.

```
1  if (PROVIDER.aws) {
2    console.log('Execute AWS specfic code');
3  }
```

Figura 9 - Exemplo de código compilado pelo plugin apenas para a AWS.

3.5 Disponibilização da Ferramenta

A ferramenta *serverless-agnostic* está disponível no repositório git em https://github.com/SergioTTF/serverless-agnostic e no repositório npm (*node package manager*) em https://www.npmjs.com/package/serverless-agnostic, sob a licença MIT.

3.6 Sumário do Capítulo

Neste capítulo destacamos as tomadas de decisões feitas sobre o desenvolvimento do trabalho e da solução, posteriormente apresentamos a solução implementada e sua arquitetura, e por fim exibimos como configurar, utilizar e onde encontrar a solução final desenvolvida.

No próximo capítulo vamos apresentar o processo metodológico usado para avaliar esta solução, evidenciando como foi feito toda a experimentação e seus respectivos resultados.

4. Metodologia, Experimentos e Resultados

Este capítulo apresenta a metodologia utilizada neste trabalho para gerar as métricas do experimento, os experimentos executados para avaliar a ferramenta proposta e também a discussão dos resultados destes experimentos. O capítulo começa explicando o conceito de GQM e como ele foi usado no projeto, posteriormente detalha como foi feita a experimentação e quais aplicações foram usadas de base para o estudo e por fim discute os resultados e as possíveis limitações da ferramenta. Este capítulo busca oferecer as ferramentas necessárias para responder a Questão 1 do GQM (Figura 10).

4.1 *Goal Question Metric* (GQM)

A metodologia escolhida para ser utilizada neste trabalho foi a *Goal Question Metric* (GQM), que consiste em uma abordagem de cima para baixo (*top-down*) para elaborar um sistema de medição orientado a metas. O processo do GQM se inicia com a elaboração de metas, posteriormente levantam-se questões que abordam os objetivos e por fim se identificam as métricas que proporcionam respostas para as questões levantadas [42].

A Tabela 4 apresenta a estruturação da Meta estabelecida para o GQM

Análise	serverless-agnostic
Finalidade de	avaliação, melhoria, caraterização
No que diz respeito ao	esforço de desenvolvimento, portabilidade entre provedores
Do ponto de visto do	desenvolvedor
No seguinte contexto	projetos possivelmente multi-cloud

Tabela 4 - Caracterização da Meta do GQM

A meta estabelecida para o modelo GQM foi: Analisar o serverless-agnostic para avaliar o impacto sobre o esforço de desenvolvimento e portabilidade entre

provedores em projetos possivelmente multi-cloud do ponto de vista do desenvolvedor.

O modelo GQM utilizado neste trabalho está caracterizado na Figura 10. Onde, destaco que no topo do diagrama e com fundo verde se encontra a Meta definida para o projeto, na segunda linha em fundo azul se encontram as Questões levantadas provenientes da meta e por fim na base do diagrama em fundo vermelho se encontram as Métricas escolhidas para responder às questões levantadas.

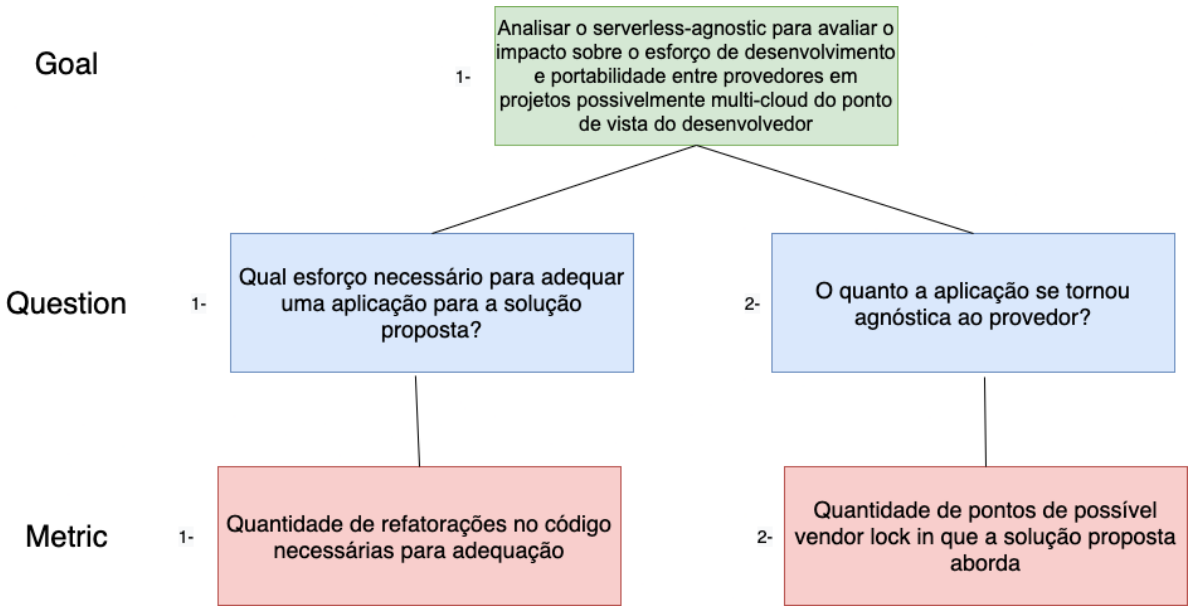

Figura 10 - GQM do trabalho

A meta definida para este trabalho aborda um tema complexo, que é o desenvolvimento de software em ambientes *Multi-cloud*, dessa maneira torna-se necessário limitar o escopo da solução para que esta se encaixe no contexto do trabalho. Esclarecendo isto, fica mais fácil entender a Questão de número 2 do GQM que busca investigar o quanto a solução facilitou o desenvolvimento Multi-cloud com Serverless, através da Métrica de número 2 que quando cita "*pontos de possível vendor lock in*" se refere aos desafios listados na seção 2.2.4 do Capítulo 2.

Já a Questão de número 1 do GQM investiga a dificuldade de utilização da solução, fazendo uso da Métrica de número 1. Esta métrica é utilizada na experimentação descrita na seção seguinte.

4.2 Experimento

Tendo conhecimento do contexto da ferramenta e seu funcionamento apresentado no Capítulo 3, foram selecionadas 26 aplicações web, que usam Express.js, para experimentação. Estas, são fornecidas pelo próprio repositório git do framework, são aplicações criadas pela comunidade que abrangem exemplos de diferentes casos de uso do framework [48]. As aplicações e suas descrições são detalhadas na Tabela 5 abaixo.

Aplicação	Descrição
auth	Authentication with login and password
content-negotiation	HTTP content negotiation
cookie-sessions	Working with cookie-based sessions
cookies	Working with cookies
downloads	Transferring files to client
ejs	Working with Embedded JavaScript templating (ejs)
error-pages	Creating error pages
error	Working with error middleware
hello-world	Simple request handler
markdown	Markdown as template engine
multi-router	Working with multiple Express routers
multipart	Accepting multipart-encoded forms
mvc	MVC-style controllers
online	Tracking online user activity with online and redis packages
params	Working with route parameters

resource	Multiple HTTP operations on the same resource
route-map	Organizing routes using a map
route-middleware	Working with route middleware
route-separation	Organizing routes per each resource
search	Search API
session	User sessions
static-files	Serving static files
vhost	Working with virtual hosts
view-constructor	Rendering views dynamically
view-locals	Saving data in request object between middleware calls
web-service	Simple API service

Tabela 5 - Lista de aplicações Express.js exemplo

O experimento consistiu do autor do trabalho usar a solução desenvolvida(o plugin *serverless-agnostic*) para migrar todas as aplicações para os serviços Serverless da AWS Lambda e da Google Cloud Functions e analisar o esforço necessário e os possíveis problemas que poderiam surgir. O experimento não contou com usuários externos devido à disponibilidade de tempo para realização do trabalho que foi realizado em um semestre atípico na universidade, onde teve duração de apenas três meses.

Os arquivos de configuração utilizados por todas as aplicações, foram os mesmos. Estes definindo que para a função implantada tanto na AWS Lambda quanto na Google Cloud Functions, seriam escolhidos o ambiente de runtime Nodejs 12 e a região do servidor *us-east-1*. Segue na Tabela 6 abaixo, o conteúdo dos arquivos de configuração utilizados para todas as aplicações.

	Conteúdo do Arquivo
.serverless.yml	*service: express-simple-project* *frameworkVersion: '2'*

	provider: * name:* *plugins:* * - serverless-agnostic* *custom:* * serverlessAgnostic:* * serverFile: index.js* * handlerName: myHandler* * serverName: app* * providers:* * - name: aws* * slsFilePath: serverless.aws.yml* * - name: gcp* * slsFilePath: serverless.gcp.yml*
.serverless.aws.yml	*service: lambda-express-example* *frameworkVersion: '2'* *provider:* * name: aws* * runtime: nodejs12.x* *functions:* * myHandler:* * handler: index.myHandler* * events:* * - http:* * path: /* * method: ANY* * cors: true* * - http:* * path: /{proxy+}* * method: ANY* * cors: true*
.serverless.gcp.yml	*service: gcp-express-example* *provider:* * name: google* * stage: dev* * runtime: nodejs12* * region: us-east1* * project: teak-instrument-281406* * credentials: ~/.gcloud/keyfile.json* *frameworkVersion: '2'* *plugins:* * - serverless-google-cloudfunctions* *package:* * exclude:* * - node_modules/***

	- .gitignore *- .git/*** *functions:* *myHandler:* *handler: myHandler* *events:* *- http: /{proxy+}* *method: ANY*

Tabela 6 - Configurações utilizadas pelas aplicações no experimento

4.3 Resultados e discussão

A Tabela 7 abaixo apresenta os resultados dos experimentos descritos na seção 5.1. Inicialmente, o planejado seria medir o esforço necessário para adaptar essas aplicações Express.js para o meu plugin através do número de refatorações necessárias para tal, porém pode-se observar que praticamente nenhuma aplicação precisou de refatoração, o único esforço exigido foi o de adicionar os arquivos de configuração ao projeto, instalar o plugin e executar os comandos de compilação e implantação.

A segunda e a terceira coluna da Tabela 7, indica com “Sim” ou “Não” se a aplicação foi migrada com sucesso, seguida de uma observação caso necessário, isto para cada provedor(Amazon Web Services e Google Cloud Platform).

Todas as aplicações foram implantadas nas contas das plataformas *Cloud* pertencentes ao autor, ambas sob uso do plano gratuito limitado(*Free Tie*r). Este que oferece uma quantidade razoável de invocações gratuitas de serviços *FaaS*.

Aplicação	**Resultado na AWS**	**Resultado na GCP**	**Número de refatorações**
auth	Sim, porém instável por uso de in memory sessions	Sim, porém instável por uso de in memory sessions	0
content-negotiation	Sim	Sim	0
cookie-sessions	Sim, porém instável	Sim, porém instável	0

	por uso de in memory sessions	por uso de in memory sessions	
cookies	Sim	Sim	0
downloads	Sim	Sim	0
ejs	Sim	Sim	0
error-pages	Sim	Sim	0
error	Sim	Sim	0
hello-world	Sim	Sim	0
markdown	Sim	Sim	0
multi-router	Sim	Sim	0
multipart	Sim	Não, incompatibilidade de biblioteca usada no upload de arquivos	0
mvc	Sim, porém instável por uso de in memory sessions	Sim, porém instável por uso de in memory sessions	0
online	Sim, Alterando endereço do Redis	Sim, Alterando endereço do Redis	1
params	Sim	Sim	0
resource	Sim	Sim	0
route-map	Sim	Sim	0
route-middleware	Sim	Sim	0
route-separation	Sim	Sim	0
search	Sim	Sim	0
session	Sim, porém instável por uso de in memory sessions	Sim, porém instável por uso de in memory sessions	0
static-files	Sim	Sim	0
vhost	Não, por fazer uso de virtual hosts	Não, por fazer uso de virtual hosts	-
view-constructor	Sim	Sim	0
view-locals	Sim	Sim	0
web-service	Sim	Sim	0

Tabela 7 - Resultado do experimento das aplicações Express.js

Pode-se observar que das 26 aplicações utilizadas no experimento, 20 foram migradas usando o provedor sem apresentar nenhum erro ou limitação aparente. Algumas das aplicações faziam uso de in-memory sessions ou server-side sessions, armazenando dados em memória para alguma de suas funcionalidades, porém a arquitetura de Aplicações *Serverless* foram feitas para aplicações Stateless, onde o armazenamento de dados relevantes para a aplicação não pode ser feito em memória e idealmente em um sistema externo como um banco de dados[6]. Este tipo de armazenamento de dados em memória gera uma instabilidade em aplicações Serverless pois em requisições diferentes não há garantia que a mesma instância da aplicação vai responder este evento, além de que após alguns minutos sem receber nenhuma chamada as instâncias da aplicação são eliminadas, escalando para zero e perdendo os dados persistidos em memória.

Para uma aplicação Express.js ser migrada com sucesso para uma arquitetura Serverless ela tem que respeitar algumas propriedades e limitações deste tipo de arquitetura. Pode-se destacar algumas destas propriedades como: ser *Stateless* e ter operações de curta duração(efêmeras).

A única refatoração realizada foi no projeto "online", a refatoração consistiu em alterar o endereço Redis para um cluster existente e não para um endereço local.

Quanto às métricas definidas no GQM, ficou expressivo(levando em consideração a amostra) que não existe um esforço de refatoração significativo para adaptação de aplicações ao formato aceito pelo plugin. E sobre a métrica 2, referente a quais desafios de ambientes multi-cloud ele aborda, pode-se observar que ele oferece uma interface de handler comum para o desenvolvimento aplicações serverless, sendo o desafio de interfaces diferentes abordado pela solução desenvolvida.

Com a realização destes experimentos, foi possível perceber que a solução implementada apresenta indícios de permitir uma simples migração de aplicações Express.js para serviços *Serverless* da Google Cloud Platform e Amazon Web

Services. Claro que com algumas limitações, pois o objetivo da ferramenta não é fazer com que as aplicações façam algo que a arquitetura *Serverless* não permite, mas sim permitir a escrita de aplicações *Serverless* de uma maneira familiar e com configurações simples para ambientes possivelmente multi-cloud.

4.4 Sumário do Capítulo

Começamos este capítulo apresentando a metodologia *Goal Question Metric* (GQM) e como ela foi utilizada neste trabalho, seguimos com a explicação da experimentação realizada e finalizamos com a exposição dos resultados e discutindo os mesmos.

O próximo capítulo finaliza o trabalho, concluindo o raciocínio desenvolvido ao longo deste documento e evidenciando possíveis limitações e trabalhos futuros.

5. Conclusão e trabalhos futuros

Este capítulo apresenta a conclusão deste trabalho, apresentando as considerações finais e apresenta também as perspectivas de trabalhos futuros que possam contribuir para o problema abordado. A primeira seção compõe a conclusão, já a segunda apresenta tópicos para trabalhos futuros.

5.1 Conclusão

Por fim, pode-se concluir que este trabalho atingiu satisfatoriamente a Meta 1 do GQM apresentado no capítulo 3, sendo esta: “Analisar o serverless-agnostic para avaliar o impacto sobre o esforço de desenvolvimento e portabilidade entre provedores em projetos possivelmente multi-cloud do ponto de vista do desenvolvedor”.

Após a fundamentação teórica sobre o tema abordado e a análise do estado da arte, formou-se a base de conhecimento que guiou o processo de desenvolvimento deste trabalho. Foi desenvolvida uma ferramenta capaz de facilitar o desenvolvimento de aplicações *Serverless* em ambientes possivelmente *multi-cloud*, sem que o desenvolvedor tenha que se preocupar com as peculiaridades da interface de cada provedor.

A experimentação realizada mostrou uma fácil adaptação de diversas aplicações Express.js, com diferentes casos de uso, para ambientes Serverless em plataformas de *Cloud* pública usando a ferramenta desenvolvida. Desenvolvedores de software podem usar a solução construída neste trabalho para criar aplicações *Serverless* com fácil portabilidade em dois grandes provedores de *Cloud Computing*.

5.2 Possíveis Limitações

- **Enviesamento dos resultados**
  O experimento foi realizado por apenas um desenvolvedor, o autor deste trabalho, gerando um certo risco para o enviesamento resultados, mesmo que as métricas sejam objetivas. Diferentes desenvolvedores podem ter abordagens distintas ao arquitetar uma solução.

- **Amostra limitada**
  O experimento contou com 26 aplicações de diferentes casos de uso, porém não necessariamente atingiu toda gama de possibilidades para funções Serverless. Podem existir erros desconhecidos em aplicações com casos de uso diferentes dos presentes na amostra.

- **Dependência no Serverless Framework**
  O módulo de implantação das funções da solução faz uso do Serverless Framework, caso esta ferramenta Open Source seja abandonada por seus mantenedores, um novo módulo terá que ser desenvolvido para se manter atualizado com as APIs dos provedores.

- **Funcionalidade de compilação específica por provedor**
  Embora a funcionalidade de compilar código para um provedor específico ajude em situações onde determinada plataforma necessite de comportamentos distintos, ela pode gerar uma maior complexidade no código.

5.3 Trabalhos futuros

- **Expandir os provedores de Cloud aceitos para incluir Microsoft Azure e IBM Cloud**
  Atualmente apenas os provedores Amazon Web Services e Google Cloud Platform são aceitos. A plataforma da IBM Cloud e principalmente da Microsoft Azure são relevantes no mercado de provedores de Cloud Computing e ambos oferecem serviços Serverless relevantes para o ecossistema [37].

- **Desenvolver comandos de monitoramento para o plugin**
  Atualmente o serverless-agnostic contém apenas comandos para compilar(para o formato de cada provedor) e implantar o projeto. Uma possível melhoria seria desenvolver um comando para monitorar o estado da aplicação em cada provedor.

- **Adicionar mais eventos aos *triggers* aceitos.**
  Atualmente o plugin trabalha apenas com o *trigger* do evento HTTP para funções Serverless, seria interessante implementar uma compatibilidade com o formato dos CloudEvents [49], um projeto Open Source mantido pela CNCF(Cloud Native Computing Foundation) que busca facilitar a especificação e integração de eventos em diferentes serviços e plataformas.

- **Investigar possíveis problemas de compatibilidade e avisar ao desenvolvedor.**
  A Tabela 6 apresenta algumas aplicações que não conseguiram ser migradas para as plataformas Serverless, por conta de problemas de compatibilidade com bibliotecas utilizadas pela aplicação. Seria interessante implementar checagens no momento de compilação para verificar se as bibliotecas especificadas nas dependências são compatíveis com os provedores escolhidos.
- **Investigar a possibilidade de abstrair o Express.js.**
  Talvez fosse interessante para o projeto abstrair a especificidade da implementação através de uma DSL(Domain Specific Language) ou alguma forma de *low code*.

6. Referências

[1] Pew Research Centre, Internet/Broadband Fact Sheet - https://www.pewresearch.org/internet/fact-sheet/internet-broadband/ - Acesso em 04/04/2021

[2] JAIN, Nancy; CHOUDHARY, Sakshi. Overview of virtualization in cloud computing. In: 2016 Symposium on Colossal Data Analysis and Networking (CDAN). IEEE, 2016. p. 1-4.

[3] GROSSMAN, Robert L. The case for cloud computing. IT professional, v. 11, n. 2, p. 23-27, 2009.

[4] DRAGONI, Nicola et al. Microservices: yesterday, today, and tomorrow. Present and ulterior software engineering, p. 195-216, 2017.

[5] Serverless Architectures - https://martinfowler.com/articles/serverless.html - Acesso em: 20/03/2020.

[6] CASTRO, Paul et al. The server is dead, long live the server: Rise of Serverless Computing, Overview of Current State and Future Trends in Research and Industry. arXiv preprint arXiv:1906.02888, 2019.

[7] Amazon Web Services, AWS. Acesso em: 04/04/2021. Disponível em: <https://aws.amazon.com/pt/>.

[8] Google Cloud, Google Cloud Platform. Acesso em: 04/04/2021. Disponível em: <https://cloud.google.com/>.

[9] Microsoft Azure, Microsoft Azure. Acesso em: 04/04/2021. Disponível em: <https://azure.microsoft.com/pt-br/>.

[10] IBM Cloud, IBM Cloud. Acesso em: 04/04/2021. Disponível em: <https://www.ibm.com/cloud>.

[11] Amazon Web Services, AWS Lambda. Acesso em: 04/04/2021. Disponível em: <https://aws.amazon.com/pt/lambda/>.

[12] Google Cloud, Cloud Functions. Acesso em: 04/04/2021. Disponível em: <https://cloud.google.com/functions?hl=pt-br>.

[13] Microsoft Azure, Azure Functions. Acesso em: 04/04/2021. Disponível em: <https://azure.microsoft.com/pt-br/services/functions/>.

[14] IBM Cloud, IBM Cloud Functions. Acesso em: 04/04/2021. Disponível em: <https://www.ibm.com/cloud/functions>.

[15] HAYES, Brian. Cloud computing. 2008.

[16] On Premise vs. Serverless - https://customers-love-solutions.com/?p=784 - Acesso em: 05/04/2021

[17] HÖFER, C. N.; KARAGIANNIS, Georgios. Cloud computing services: taxonomy and comparison. Journal of Internet Services and Applications, v. 2, n. 2, p. 81-94, 2011.

[18] Google Cloud, App Engine. Acesso em: 06/04/2021. Disponível em: <https://cloud.google.com/appengine/>.

[19] Amazon Web Services, AWS Elastic Beanstalk. Acesso em: 06/04/2021. Disponível em: <https://aws.amazon.com/elasticbeanstalk/>.

[20] Gmail. Acesso em: 06/04/2021. Disponível em: <http://mail.google.com/mail>.

[21] Facebook. Acesso em: 06/04/2021. Disponível em: <https://www.facebook.com/>.

[22] Google Docs. Acesso em: 06/04/2021. Disponível em: <https://docs.google.com/>.

[23] Amazon Web Services, Amazon Cognito. Acesso em: 07/04/2021. Disponível em: <https://aws.amazon.com/pt/cognito/>.

[25] Firebase, Cloud Firestore. Acesso em: 07/04/2021. Disponível em: <https://firebase.google.com/products/firestore>.

[25] Auht0. Acesso em: 07/04/2021. Disponível em: <https://auth0.com/#!>.

[26] Flexera 2020 State of the Cloud Report. Acesso em 08/04/2021 - Disponível em: <https://info.flexera.com/SLO-CM-REPORT-State-of-the-Cloud-2020>.

[27] PETCU, Dana. Multi-cloud: expectations and current approaches. In: Proceedings of the 2013 international workshop on Multi-cloud applications and federated clouds. 2013. p. 1-6.

[28] HOGAN, Michael et al. Nist cloud computing standards roadmap. NIST Special Publication, v. 35, p. 6-11, 2011.

[29] BALDINI, Ioana et al. Serverless computing: Current trends and open problems. In: Research advances in cloud computing. Springer, Singapore, 2017. p. 1-20.

[30] EIVY, Adam; WEINMAN, Joe. Be wary of the economics of" Serverless" Cloud Computing. IEEE Cloud Computing, v. 4, n. 2, p. 6-12, 2017.

[31] OpenFaaS. Acesso em: 07/08/2021. Disponível em: <https://www.openfaas.com/>.

[32] Kubeless. Acesso em: 07/08/2021. Disponível em: <https://kubeless.io/>.

[33] Fn Project. Acesso em: 07/08/2021. Disponível em: <https://fnproject.io/>.

[34] Apache OpenWhisk. Acesso em: 07/08/2021. Disponível em: <https://openwhisk.apache.org/>.

[35] Kubernetes. Acesso em: 07/08/2021. Disponível em: <https://kubernetes.io/pt-br/>.

[36] The State Of Serverless. Acesso em: 07/08/2021. Disponível em: <https://www.datadoghq.com/state-of-serverless/>.

[37] O'Reilly, O'Reilly serverless survey 2019: Concerns, what works, and what to expect. Acesso em: 07/08/2021. Disponível em: <https://www.oreilly.com/radar/oreilly-serverless-survey-2019-concerns-what-works-and-what-to-expect/>.

[38] Serverless framework. Acesso em: 07/08/2021. Disponível em: <https://www.serverless.com/>.

[39] StackOverflow, StackOverflow 2021 Developer Survey. Acesso em: 07/08/2021. Disponível em: <https://insights.stackoverflow.com/survey/2021#stack-overflow-site-use-newso-sites>.

[40] Trivadis Developer Blog, Performance Evaluation of Different Programming Languages in AWS Lambda. Acesso em: 07/08/2021. Disponível em:

<https://blog.oio.de/2021/03/29/performance-evaluation-of-different-programming-languages-in-aws-lambda/>.

[41] JAMES Thomas. Taming Dragons: Building Multi-Provider Serverless Apps With The Serverless Framework. 22 jun. 2017. Disponível em: <https://www.youtube.com/watch?v=DxVoI5jM9RE>. Acesso em: 7 ago. 2021.

[42] VAN SOLINGEN, Rini et al. Goal question metric (gqm) approach. Encyclopedia of software engineering, 2002.

[43] Azure Functions, Gatilhos e Associações. Acesso em: 10/08/2021. Disponível em: <https://docs.microsoft.com/pt-br/azure/azure-functions/functions-triggers-bindings>.

[44] Oracle Blog, Cloud Computing Definition & Architecture for Cloud Service Brokers. Acesso em: 10/08/2021. Disponível em: <https://blogs.oracle.com/soacommunity/cloud-computing-definition-architecture-for-cloud-service-brokers>.

[45] Github, serverless-http. Acesso em: 10/08/2021. Disponível em: <https://github.com/dougmoscrop/serverless-http>.

[46] Google Cloud Functions, Functions Framework. Acesso em: 10/08/2021. Disponível em: <https://cloud.google.com/functions/docs/functions-framework>.

[47] Github, jscodeshift. Acesso em: 10/08/2021. Disponível em: <https://github.com/facebook/jscodeshift>.

[48] Express.js, examples. Acesso em: 10/08/2021. Disponível em: <https://expressjs.com/en/starter/examples.html>.

[49] Github, CloudEvents. Acesso em: 15/08/2021. Disponível em: <https://github.com/cloudevents/spec>.