UNIVERSIDADE FEDERAL DO PAMPA

Gustavo Bittencourt Satheler

Alta Disponibilidade de Funções como Serviço em Ambiente de Múltiplas Nuvens de Computação

Alegrete - RS
2021

Gustavo Bittencourt Satheler

Alta Disponibilidade de Funções como Serviço em Ambiente de Múltiplas Nuvens de Computação

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Engenharia de Software da Universidade Federal do Pampa como requisito parcial para a obtenção do título de Bacharel em Engenharia de Software.

Orientador: Prof. Dr. Diego Kreutz

Alegrete - RS
2021

Ficha catalográfica elaborada automaticamente com os dados fornecidos
pelo(a) autor(a) através do Módulo de Biblioteca do
Sistema GURI (Gestão Unificada de Recursos Institucionais) .

S253a Satheler, Gustavo Bittencourt
Alta Disponibilidade de Funções como Serviço em Ambiente de Múltiplas Nuvens de Computação / Gustavo Bittencourt Satheler.
66 p.

Trabalho de Conclusão de Curso(Graduação)-- Universidade Federal do Pampa, ENGENHARIA DE SOFTWARE, 2021.
"Orientação: Diego Luis Kreutz".

1. Computação em Nuvem. 2. Função como serviço. 3. Alta disponibilidade. 4. Redundância. I. Título.

SERVIÇO PÚBLICO FEDERAL

MINISTÉRIO DA EDUCAÇÃO

Universidade Federal do Pampa

GUSTAVO BITTENCOURT SATHELER

ALTA DISPONIBILIDADE DE FUNÇÕES COMO SERVIÇO EM AMBIENTE DE MÚLTIPLAS NUVENS DE COMPUTAÇÃO

Monografia apresentada ao Programa de Engenharia de Software da Universidade Federal do Pampa, como requisito parcial para obtenção do Título de Bacharel em Engenharia de Software.

Monografia defendida e aprovada em: 09, de junho de 2021.

Banca examinadora:

Prof. Dr. Diego Luis Kreutz

Orientador

Unipampa

Prof. Dr. Claudio Schepke
Unipampa

Prof. Dr. Vinicius Vielmo Cogo
Universidade de Lisboa

Dr. Vitor Chaves de Oliveira
Campasso UOL

Assinado eletronicamente por **VITOR CHAVES DE OLIVEIRA**, **Usuário Externo**, em 09/06/2021, às 14:19, conforme horário oficial de Brasília, de acordo com as normativas legais aplicáveis.

Assinado eletronicamente por **Vinícius Vielmo Cogo**, **Usuário Externo**, em 09/06/2021, às 14:19, conforme horário oficial de Brasília, de acordo com as normativas legais aplicáveis.

Assinado eletronicamente por **CLAUDIO SCHEPKE**, **PROFESSOR DO MAGISTERIO SUPERIOR**, em 09/06/2021, às 14:40, conforme horário oficial de Brasília, de acordo com as normativas legais aplicáveis.

Assinado eletronicamente por **DIEGO LUIS KREUTZ**, **PROFESSOR DO MAGISTERIO SUPERIOR**, em 14/06/2021, às 17:21, conforme horário oficial de Brasília, de acordo com as normativas legais aplicáveis.

A autenticidade deste documento pode ser conferida no site https://sei.unipampa.edu.br/sei/controlador_externo.php?acao=documento_conferir&id_orgao_acesso_externo=0, informando o código verificador **0522083** e o código CRC **1F0A2273**.

Este trabalho é dedicado à todos que fizeram parte da minha trajetória,
me ajudando tanto com conhecimento quanto com apoio nos momentos difíceis.

AGRADECIMENTOS

Em primeiro lugar, quero agradecer aos meus pais Alcendino e Sanya e ao meu irmão João Pedro, que apoiaram a minha decisão de sair da minha cidade natal para dedicar-me aos estudos longe deles. Como também, agradeço por compreenderem a minha ausência enquanto eu me dedicava.

Aos meus incríveis amigos Michael Martins e Judson Henrique que tive a sorte e felicidade de conhecer durante essa jornada, me apoiaram nos mais diversos “perrengues” e nas mais adversas situações. Não posso deixar de citar os meus amigos Jon Snow (Tadeu Jenuario) e Brandow Buenos que também foram essenciais para a minha formação. E um agradecimento especial à Melissa, em que principalmente na fase de conclusão de curso, me ajudou nos momentos difíceis de cansaço e de fraqueza com palavras motivação, de afeto e carinho.

Quero também agradecer ao meu professor, orientador e amigo Diego Kreutz que aceitou esta difícil missão de me orientar neste trabalho. Durante toda a minha jornada me auxiliou, foi paciente (principalmente quando fui teimoso) e se fez a disposição de diversas maneiras. E por fim, mas não menos importante, agradeço a todos os professores, alunos e amigos envolvidos durante esta jornada. Todos tiveram a sua importância e influência para o meu desenvolvimento pessoal e profissional.

"Alguns homens vêem as coisas como são, e dizem 'Por quê?' Eu sonho com as coisas que nunca foram e digo 'Por que não?'

(Geroge Bernard Shaw)

RESUMO

A computação sem servidor está se tornando cada vez mais atraente para arquitetos e desenvolvedores de soluções em nuvem. Esse novo paradigma de computação deu origem a plataformas de Funções como Serviço (FaaS) que permitem a implantação de funções sem se preocupar com a infraestrutura. Um desafio importante no projeto de plataformas FaaS é garantir a disponibilidade das funções implantadas. Neste trabalho, propomos uma abordagem de alta disponibilidade baseada em redundância ativo-passivo. Os resultados mostram que é possível balancear as aplicações FaaS entre os provedores de nuvem (AWS e Microsoft Azure) e que o tempo de disponibilidade pode ultrapassar 99,999%.

Palavras-chave: Computação em Nuvem. Função como serviço. Alta disponibilidade. Redundância.

ABSTRACT

Serverless computing is becoming more and more attractive for cloud solution architects and developers. This new computing paradigm has given rise to Functions as a Service (FaaS) platforms that allow you to deploy roles without worrying about infrastructure. An important challenge in designing FaaS platforms is ensuring the availability of deployed functions. In this work, we propose an active-passive redundancy-based high availability approach. The results of our evaluation on a use case show that it is possible. The results show that it is possible to balance FaaS applications between cloud providers (AWS and Microsoft Azure) and that uptime can exceed 99.999%.

Key-words: Cloud Computing. Function-as-a-Service. High Availability. Redundancy.

LISTA DE FIGURAS

Figura 1 – Arquitetura em múltiplos provedores de nuvem com microsserviços. . . 27
Figura 2 – Proposta da arquitetura entre nuvens com FaaS. 31
Figura 3 – Implementação da arquitetura 35
Figura 4 – Visão geral da integridade dos provedores 41
Figura 5 – Visão geral das tabelas do banco de dados em ambos provedores. . . . 42
Figura 6 – Quantidade de conexões ativas no banco de dados 42
Figura 7 – Tempo de indisponibilidade da aplicação 47
Figura 8 – Tempo de resposta por solicitações por segundo 47

LISTA DE TABELAS

Tabela 1 – Resumo das características dos trabalhos relacionados 29
Tabela 2 – Tabela comparativa dos provedores/plataforma que oferecem balanceamento de cargas. 36
Tabela 3 – Sequência de ações 43
Tabela 4 – Custos dos provedores de nuvem 45

LISTA DE ABREVIATURAS

Azure Microsoft Azure

RESTful *Representational State Transfer*

S12R Serverlessizer

LISTA DE SIGLAS

API *Application Programming Interface*

AWS Amazon Web Services

DNS *Domain Name System*

FaaS *Function-as-a-Service*

gRPC *General-purpose Remote Procedure Calls*

HTTP *Hypertext Transfer Protocol*

IaaS *Infrastructure-as-a-Service*

MMR *Multi-Master Replication*

PaaS *Platform-as-a-Service*

SaaS *Software-as-a-Service*

WAL *Write-Ahead Logging*

SUMÁRIO

1 INTRODUÇÃO 25
1.1 Problema 26
1.2 Proposta 27
1.3 Estrutura do Trabalho 28

2 ESTADO DA ARTE 29
2.1 Trabalhos relacionados 29
2.2 Modelos de replicação 29

3 ALTA DISPONIBILIDADE COM FAAS 31
3.1 Balanceador de cargas 31
3.2 Monitor de disponibilidade 32
3.3 *Gateway* de API 32
3.4 Funções 32
3.5 Banco de dados replicado 33

4 IMPLEMENTAÇÃO 35
4.1 Seleção dos provedores 35
4.2 Balanceador de cargas 36
4.3 Monitor de disponibilidade 36
4.4 *Gateway* de API e Funções 37
4.5 Banco de dados 39
4.6 Modelo de sistema 39
4.7 Implantação 40

5 AVALIAÇÃO 41
5.1 Cenários de testes 41
5.2 Aplicação 43
5.3 Teste de carga 44
5.4 Ambiente de testes 44
5.5 Custo 45
5.6 Resultados 46

6 CONSIDERAÇÕES FINAIS 49
6.1 Contribuições 49
6.2 Trabalhos Futuros 49

REFERÊNCIAS 51

ANEXOS **55**

ANEXO A – CONFIGURAÇÃO NA AMAZON WEB SERVICES (AWS) 57

ANEXO B – CONFIGURAÇÃO NA MICROSOFT AZURE 59

ANEXO C – CONFIGURAÇÃO NA CLOUDFLARE 61

Índice 63

1 INTRODUÇÃO

Os provedores de computação em nuvem vêm provocando uma revolução na forma de desenvolver e disponibilizar sistemas. Diferentemente das infraestruturas tradicionais, como um *data center in-house*, os provedores permitem alocar e reconfigurar recursos sob demanda, conforme a capacidade e quantidade definida pelo cliente. O cliente é cobrado somente pelos recursos utilizados (VAQUERO et al., 2008).

As nuvens computacionais oferecem diversos modelos de nuvem, os quais manifestam como diferentes tipos de recursos e que são oferecidos como serviços pelos provedores. Atualmente, conforme (BUYYA et al., 2018) os modelos mais conhecidos são: *Infrastructure-as-a-Service* (IaaS), *Platform-as-a-Service* (PaaS) e *Software-as-a-Service* (SaaS). O modelo IaaS oferece acesso a recursos computacionais, como processamento (máquinas virtuais e/ou contêineres), rede e armazenamento, que são configurados e gerenciados pelo cliente. Enquanto isso, o modelo PaaS é direcionado para usuários que demandam menor controle sobre os recursos computacionais utilizados comparado ao IaaS e oferece a capacidade de criação e implantação de aplicações na nuvem. Já no modelo SaaS, o cliente tem acesso a componentes predefinidos ou aplicações completas dentro do acervo do provedor. Neste modelo, o cliente é um usuário final, sem a necessidade de conhecimentos em programação.

Além dos três modelos tradicionais (IaaS, PaaS e SaaS), uma tendência emergente em nuvens computacionais tem sido a adoção do modelo *Function-as-a-Service* (FaaS). As aplicações são construídas como conjunto de funções que executam em instâncias[1] sob demanda em resposta a eventos (ROBERTS, 2018). Essas instâncias são temporárias e não guardam estado, podendo permanecer ativas durante a execução de uma única requisição, e gerenciadas inteiramente pelo provedor, que fica responsável por garantir sua escalabilidade e disponibilidade. O cliente é cobrado por invocação, sem pagar por recursos ociosos. Dessa forma, o modelo FaaS pode propiciar uma economia significativa em relação aos modelos mais tradicionais IaaS e PaaS, ao mesmo tempo em que dispensa o desenvolvedor de preocupações com o gerenciamento da infraestrutura (ADZIC; CHATLEY, 2017).

Em suma, a tarifação do FaaS é baseada na quantidade de invocações, na memória alocada e tempo de execução da função. Portanto, o custo do FaaS para alguns tipos de aplicações tende a ser mais variável comparados aos modelos IaaS e PaaS, na qual a tarifação deriva dos recursos alocados (EIVY; WEINMAN, 2017).

A arquitetura em microsserviços vem ganhando impulso nos últimos anos nas empresas de tecnologia, já sendo empregada nas principais soluções de grandes empresas (*e.g.* Amazon, Netflix, Spotify e Twitter) (THÖNES, 2015). Isso ocorre porque os microsserviços (se configurados corretamente) podem ajudar a atingir os requisitos de qualidade, por

[1] O termo instância será utilizado para denotar o contêiner ou máquina virtual que uma função é executada, conforme (WANG et al., 2018).

exemplo, disponibilidade, portabilidade, escalabilidade (SOLDANI; TAMBURRI; HEUVEL, 2018). Para melhorar a sua disponibilidade, o sistema de orquestração de contêineres fornece tolerância a falha para seus microsserviços gerenciados, ou seja, reiniciando ou substituindo os contêineres com falha, como é o caso do (The Linux Foundation, 2021). Embora essas ações melhorem naturalmente a disponibilidade dos microsserviços implantados, a redundância continua sendo o recurso mais importante para obter a alta disponibilidade (VAYGHAN et al., 2019).

1.1 Problema

Alta disponibilidade é uma qualidade de infraestrutura de computação que permite que uma aplicação continue funcionando, mesmo quando alguns de seus componentes falham. Isso é um ponto crucial principalmente para sistemas críticos que não podem tolerar a interrupção do serviço, em que qualquer tempo de inatividade pode causar prejuízos, danos ambientais e até perda da vida humana.

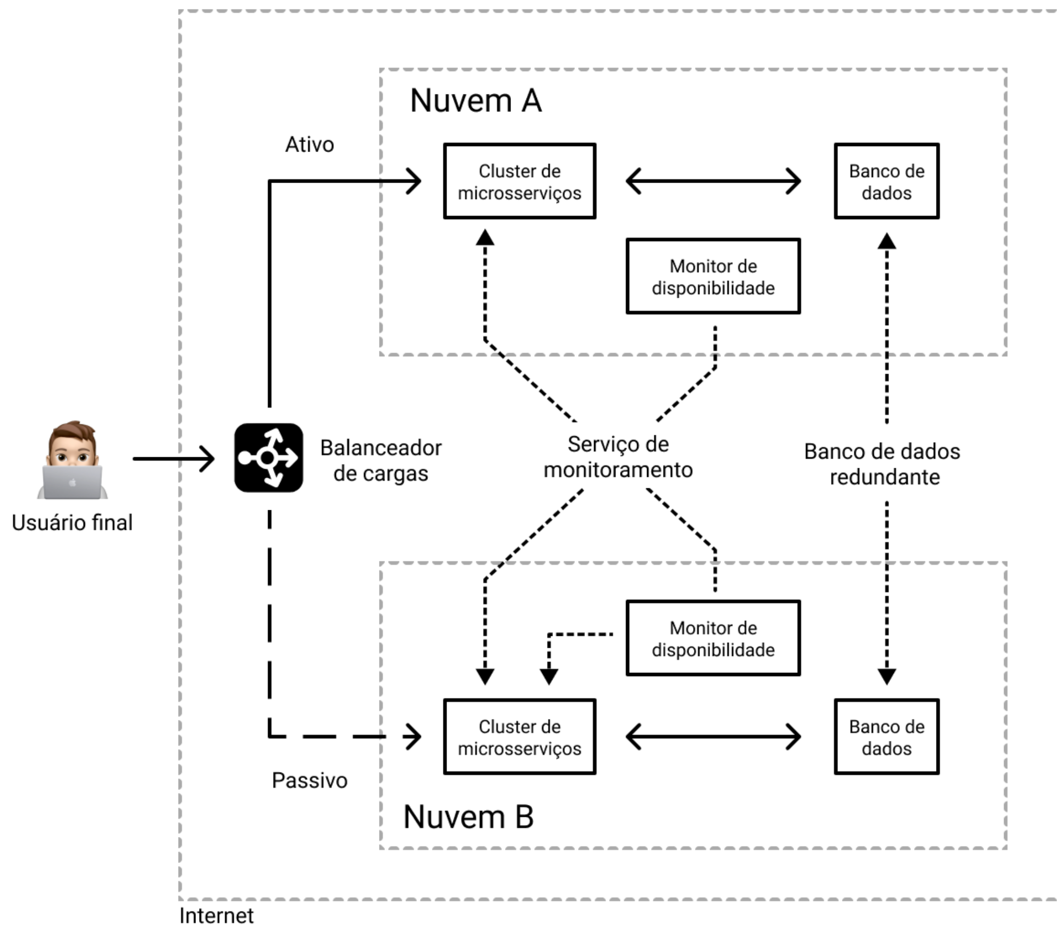
A disponibilidade é um requisito não-funcional importante que precisa ser atendido para alcançar um alto grau de confiabilidade de uma aplicação. Os sistemas considerados altamente disponíveis garantem uma certa porcentagem de tempo de atividade. A quantidade de “noves” é comumente utilizado para indicar o grau de disponibilidade. Por exemplo, sistemas considerados altamente disponíveis devem conter 5 “noves”, ou seja, 99,999% do tempo, com não mais de 5 minutos fora do ar no período de 1 ano (NABI; TOEROE; KHENDEK, 2016).

A Figura 1 mostra como é implementada na prática a alta disponibilidade com microsserviços em múltiplos provedores de nuvem. O modelo apresentado é baseado no padrão de redundância ativo-passivo (AHLUWALIA; JAIN, 2006). Enquanto na nuvem A, o *cluster* de microsserviços mantém escalabilidade da aplicação conforme a demanda, isto é, gerenciando as instâncias de acordo com o tráfego, a nuvem B mantém a quantidade mínima de contêineres ativos. A fim de verificar a disponibilidade da aplicação nos provedores, o serviço de monitoramento realiza requisições periódicas nos provedores. Por exemplo, ao detectar falhas na nuvem A, o monitor pode indicar ao balanceador de cargas que o tráfego deve ser redirecionado para a nuvem B, diminuindo o tempo de inatividade da aplicação.

Embora existam algumas soluções arquiteturais de referência para melhorar a alta disponibilidade e recuperação de desastres em provedores de nuvem, os mais comuns tem o foco nos modelos IaaS e PaaS. Não temos conhecimento de qualquer literatura que verifique os desafios e as características envolvidas na implementação de um modelo de alta disponibilidade em múltiplos provedores de nuvem utilizando o modelo FaaS.

Adicionalmente, de acordo com os termos de compromisso de serviço da Amazon Web Services (AWS) e Microsoft Azure (Azure), cada uma garante uma porcentagem de tempo de atividade mensal dos seus serviços de pelo menos 99,95% (SERVICES, 2021)

Figura 1 – Arquitetura em múltiplos provedores de nuvem com microsserviços.



Fonte: Modificado de (ADDO; AHAMED; CHU, 2014).

(AZURE, 2021), ou seja, com apenas 3 noves e meio.

Q1. Como manter a alta disponibilidade de aplicações baseadas em FaaS no ambiente de múltiplos provedores de nuvem?

Q2. Quais os desafios de Engenharia de Software e técnicos para criar aplicações FaaS em múltiplos provedores de nuvem?

Q3. O balanceamento de carga funciona da mesma maneira entre aplicações baseadas em microsserviços e FaaS?

1.2 Proposta

Neste trabalho, temos como o objetivo geral propor uma arquitetura de alta disponibilidade de FaaS em ambiente de múltiplas nuvens de computação. A nossa proposta é embasada em uma arquitetura de alta disponibilidade de aplicações baseadas em microsserviços. Para alcançar o objetivo principal são definidos alguns objetivos específicos conforme a seguir:

- Definir e implementar uma aplicação FaaS para avalição de alta disponibilidade em ambiente de múltiplos provedores de nuvem.

- Avaliar questões técnicas e operacionais de alta disponibilidade FaaS em ambiente de múltiplos provedores de nuvem.

1.3 Estrutura do Trabalho

O trabalho está organizado da seguinte forma:

Capítulo 1: Introdução do problema e esboço da nossa proposta para a alta disponibilidade com FaaS.

Capítulo 2: Contextualização e trabalhos relacionados.

Capítulo 3: Apresentação da proposta, que visa orientar as soluções de alta disponibilidade com FaaS.

Capítulo 4: Detalhamento da implementação da aplicação em múltiplos provedores de nuvem e serviços utilizados.

Capítulo 5: Apresentação e discussão dos resultados da avaliação da proposta.

Capítulo 6: Considerações finais, destacando os resultados, contribuições e trabalhos futuros.

2 ESTADO DA ARTE

Analisamos os trabalhos que abordam FaaS com relação ao custo, desempenho e alta disponibilidade (Seção 2.1). Similarmente, agrupamos e discutimos soluções arquiteturais de aplicações em múltiplos provedores de nuvem observando modelos de replicação (Seção 2.2).

2.1 Trabalhos relacionados

A Tabela 1 resume três características de trabalhos: custos, desempenho, alta disponibilidade. Como pode ser obervado na tabela, a maioria dos trabalhos tem como intuito investigar sobre os custos e desempenho.

Tabela 1 – Resumo das características dos trabalhos relacionados

Trabalho	Custo	Desempenho	Alta disponibilidade
(BORTOLINI; OBELHEIRO, 2019)	✓	✓	×
(JACKSON; CLYNCH, 2018)	✓	✓	×
(BOUIZEM et al., 2020)	×	✓	✓
(ADZIC; CHATLEY, 2017)	✓	×	×
(EIVY; WEINMAN, 2017)	✓	×	×

✓ O trabalho tem a característica. × O trabalho não tem a característica.

Um dos principais desafios para as plataformas FaaS é garantir alta disponibilidade para as funções implantadas (BOUIZEM et al., 2020). Todas as plataformas existentes suportam uma forma básica de tolerância a falhas por meio de novas tentativas de execuções de funções (BOUIZEM et al., 2020). A proposta utiliza a abordagem de comutação ativo-passivo, destacando o uso de um único provedor ou plataforma. Em seus resultados, demonstram um crescimento na disponibilidade e na performance da aplicação em comparação a abordagem de novas tentativas.

2.2 Modelos de replicação

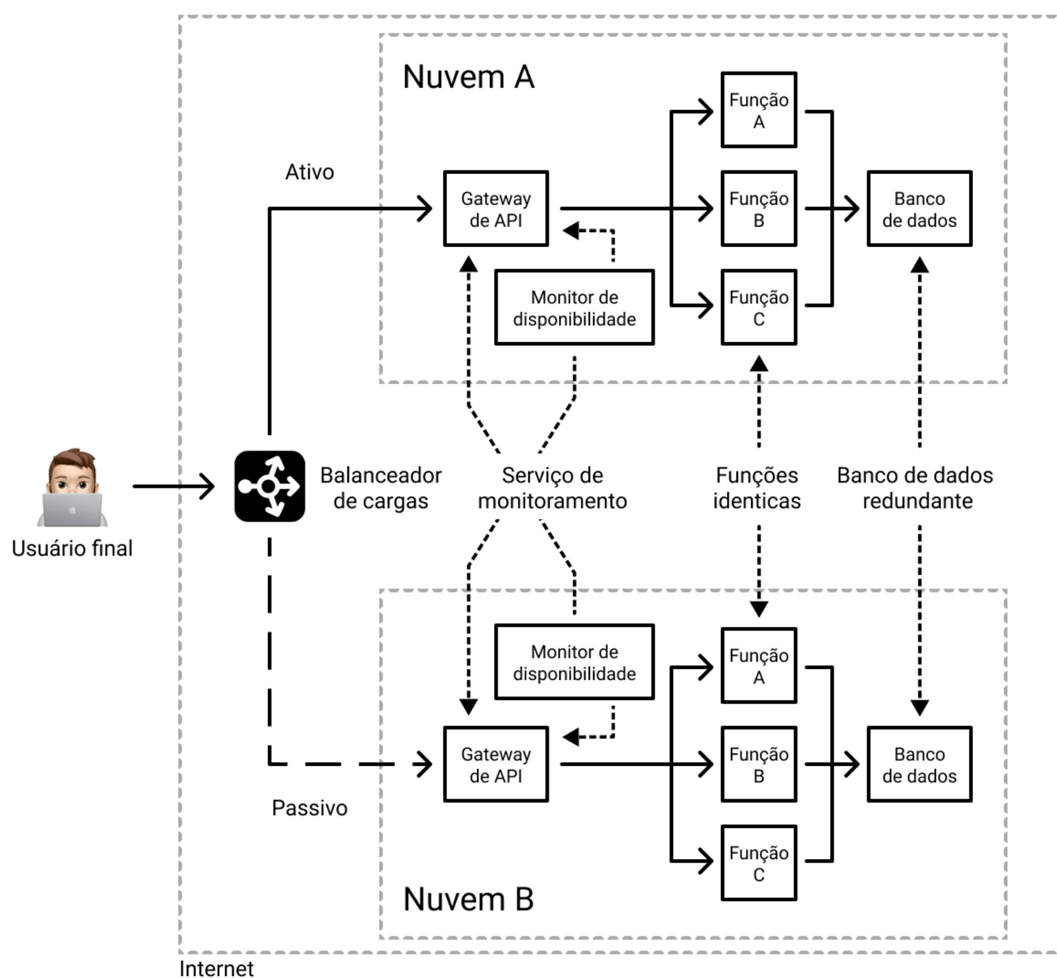
Replicação é a chave para prover alta disponibilidade e tolerância a falhas em sistemas distribuídos (COULOURIS et al., 2013). Quando se trata de alta disponibilidade há dois tipos de replicações principais:

- ativo-ativo: as solicitações passam por um balanceador de carga que distribui o trafégo em vários servidores ativos.
- ativo-passivo: as solicitações são tratadas por um servidor principal que recebe todo o tráfego. Enquanto outro servidor permanece em espera, recebendo o tráfego apenas em caso de falha no servidor principal.

3 ALTA DISPONIBILIDADE COM FAAS

A Figura 2 resume a arquitetura proposta. Como pode ser observado, há uma similaridade com a arquitetura de microsserviços, conforme apresentada na Figura 1. Há um balanceador de cargas, um monitor de disponibilidade, a aplicação (representada na Figura como função A, B e C) e o banco de dados. Entretanto, existem duas diferenças essenciais entre as propostas: (a) o *Gateway* de API que age como a "porta de entrada" para as funções; (b) e as funções com um nível menor de granularidade em comparação aos microsserviços.

Figura 2 – Proposta da arquitetura entre nuvens com FaaS.



O modelo apresentado possui apenas dois provedores de nuvem no intuito de simplificar a explicação. Porém, este modelo pode ser estendido para a quantidade de provedores de nuvem desejado.

3.1 Balanceador de cargas

O balanceamento de carga se refere à distribuição eficiente do tráfego de rede de entrada em um grupo de servidores (ALI, 2015). Um balanceador de carga atua como

o "guarda de tráfego" posicionado à frente dos servidores e encaminhando as requisições do cliente em todos os servidores disponíveis, de maneira a maximizar a velocidade e a utilização da capacidade e garantir que nenhum servidor fique sobrecarregado, o que pode prejudicar o desempenho. Se um único servidor ficar inativo, o balanceador de carga redirecionará o tráfego para os servidores disponíveis restantes (NGINX, 2021).

3.2 Monitor de disponibilidade

Um monitor de disponibilidade é um recurso responsável por monitorar o desempenho e interrupções de serviços nos provedores e serviços utilizados para maximizar o tempo de atividade do serviço. Além disso, ele é capaz de notificar ou executar operações quando uma interrupção do serviço é detectada e, quando o serviço é restaurado, respectivamente.

3.3 *Gateway* de API

Um *gateway* de API recebe todas as chamadas dos usuários e as encaminha para o serviço apropriado com roteamento de solicitação, composição e conversão de protocolos (REDHAT, 2021). Ao receber uma solicitação, o *gateway* de API consulta um mapa de roteamento que específica qual serviço deverá atender a solicitação. Um mapa de roteamento pode, por exemplo, mapear um método (*e.g.* GET) e caminho (*e.g. /health*) para determinar o serviço.

Ele também pode implementar algumas operações usando composição de API. Isto é, agrega vários serviços para atender uma solicitação e retornar o resultado apropriado. Além disso, o *gateway* de API pode fornecer uma API RESTful para os clientes, embora os serviços usem uma mistura de protocolos, por exemplo, RESTful e *General-purpose Remote Procedure Calls* (gRPC). Desta forma, as operações são convertidas entre a API externa RESTful e as APIs internas baseadas em gRPC.

Em nossa proposta, o *gateway* de API tem a atribuição de receber uma solicitação e converter para o serviço de FaaS na forma de eventos.

3.4 Funções

Uma função é o menor componente, em termos de granularidade, que executa a lógica de negócios e de aplicação. As aplicações podem ser compostas de muitas funções.

As funções são executadas em resposta a eventos. Na sequência, as instâncias são iniciadas e processam solicitações individuais. Caso haja várias solicitações simultâneas a serem processadas por uma função, o sistema criará um número de cópias suficientes para atender à demanda. Quando a demanda cai, o provedor reduz a escala automatica-

mente. A escalabilidade dinâmica é um benefício do FaaS, além de ser econômico, pois os provedores cobram apenas pelos recursos efetivamente usados, e não pelo tempo ocioso.

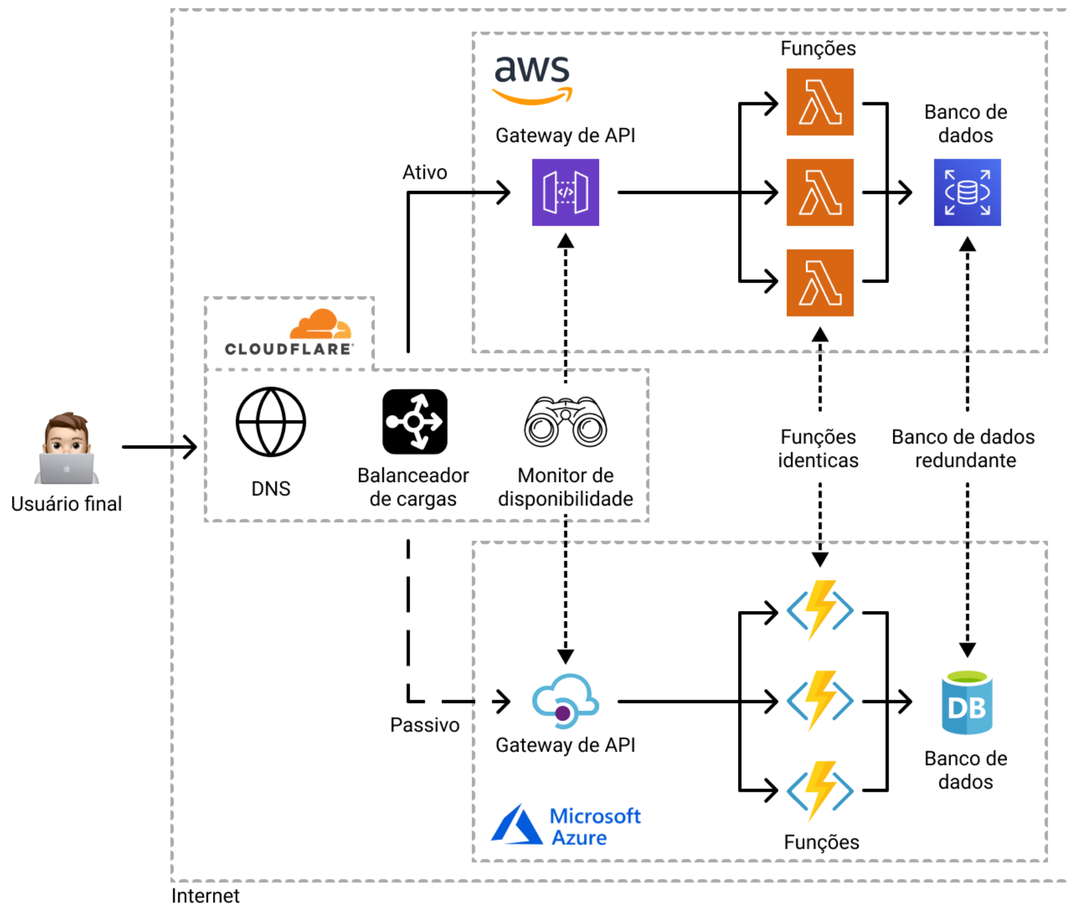
3.5 Banco de dados replicado

A replicação de banco de dados é amplamente usada para tolerância a falhas, escalabilidade e desempenho (KEMME; JIMÉNEZ-PERIS; PATIÑO-MARTÍNEZ, 2010). É bastante complexo alternar automaticamente entre nuvens sem perda de dados (SAHA; MUKHOPADHYAY; BANERJEE, 2006). Por exemplo, o uso de arquivamento contínuo pode ser usado para criar uma configuração de um *cluster* de alta disponibilidade com um ou mais servidores em espera prontos para assumir as operações se o servidor primário falhar. Esse recurso é amplamente conhecido como espera passiva (em inglês, *warm standby*) ou envio de registros (em inglês, *log shipping*). De maneira resumida, o servidor primário e o servidor em espera trabalham juntos para fornecer esse recurso, embora os servidores estejam apenas fracamente acoplados. O servidor principal opera em modo de arquivamento contínuo, enquanto cada servidor em espera opera em modo de recuperação contínua, lendo os arquivos de registros do principal.

4 IMPLEMENTAÇÃO

A Figura 3 mostra a implementação da arquitetura, além da visão geral dos provedores e seus serviços. Percebe-se que foram utilizados três provedores de nuvem: Cloudflare, Amazon Web Services e Microsoft Azure. Na Cloudflare foram utilizados os serviços de balanceador de cargas, DNS e o monitor de disponibilidade. Já na AWS e na Azure foram utilizados os recursos de *gateway* de API, plataforma FaaS e o banco de dados.

Figura 3 – Implementação da arquitetura



Fonte: Autoral.

4.1 Seleção dos provedores

Inicialmente foi realizada uma pesquisa de mercado para levantar os provedores de nuvem existentes e a sua adoção no mercado. De acordo com as últimas pesquisas da (Canalys, 2021) e (Synergy Research Group, 2021), AWS e Azure controlam mais de 50% dos gastos mundiais com serviços de infraestrutura nas nuvens. Adicionalmente, os dois provedores oferecem todos os serviços essenciais para o nosso estudo de caso e, portanto, foram as duas plataformas escolhidas.

4.2 Balanceador de cargas

Para a seleção do serviço de balanceamento de cargas consideramos as seguintes premissas: (a) não ser um serviço oferecido pelos provedores de nuvem selecionados; (b) possuir a verificação de disponibilidade. Resumidamente, decidimos não usar o serviço de balanceador de cargas nos provedores de nuvem selecionados seguindo a hipótese de uma suposta indisponibilidade do provedor durante os nossos testes. Em seguida foi considerado o custo dos serviços conforme a Tabela 2. Nos provedores que os custos são baseados no uso mensal, ou seja, na quantidade de servidores de apontamento e dados processados, os valores foram definidos como segue:

- Servidores de apontamento: 2
- Dados processados: 1 *gigabyte*

Como os valores dos serviço nos provedores são precificados em dólar americano, de forma a simplificar os valores, definimos a equivalência de 5 reais para cada 1 dólar.

Tabela 2 – Tabela comparativa dos provedores/plataforma que oferecem balanceamento de cargas.

Provedor/plataforma	Monitoramento	Preço (mensal)
Cloudflare	✓	R$25,00
DigitalOcean	✓	R$50,00
IBM Cloud	✓	R$90,90
Google Cloud	✓	R$138,75

Por possuir o valor mais acessível, este trabalho utilizou o serviço oferecido pela Cloudflare. Na Cloudflare, o balanceador de cargas é identificado através de um DNS (por exemplo, www.example.com). Adicionalmente, configuramos para que o conjunto de servidores opere de forma ativo-passivo. Com essa configuração, o balanceador de carga direcionará todo o tráfego para o primário até que o limite de falhas (configurável) seja atingido. E só então o balanceador de carga direcionará o tráfego para o próximo conjunto de servidores disponível de acordo com a ordem definida. No caso de todos os conjuntos de servidores estarem inacessíveis, a Cloudflare usa o conjunto de servidores de reserva, que é a opção de último recurso para enviar tráfego com sucesso para um servidor. Como o servidor de reserva é o último recurso, a verificação de disponibilidade não é realizada.

4.3 Monitor de disponibilidade

As verificações foram configuradas por meio de monitores na Cloudflare, que definem que tipo de verificação deve ser executada e com que frequência. Além disso é possivel configurar o monitoramento de endereços específicos através de requisições HTTP[1] perió-

[1] HTTP é um protocolo de comunicação entre sistemas de informação.

dicas, determinar intervalos personalizáveis, tempo limite de resposta e códigos de estado. As falhas são definidas por parada, ou seja, quando o provedor de nuvem não está apto a lidar com as solicitações. O monitoramento é realizado por cada um dos servidores da Cloudflare, na qual possui servidores em 13 regiões geográficas que abrangem o mundo. As verificações de disponibilidade resultam em uma mudança de estado que são registradas como eventos nos registros do balanceador de carga. O intervalo entre as verificações é definido em segundos e o seu valor mínimo é determinado conforme o plano contratado. No plano que contratamos, o valor mínimo permitido foi de 60 segundos. Porém, há planos (de maior custo) que permitem definir o intervalo de verificação em 5 segundos.

4.4 *Gateway* de API e Funções

Para abstrair e adaptar os eventos criados pelos *gateways* de API em cada um dos provedores foi criado a biblioteca Serverlessizer (S12R)[2]. O S12R recebe o evento do *gateway* e, através de um algoritmo do tipo *Strategy Pattern* formata a requisição e resposta para a aplicação. Por exemplo, ao realizar uma solicitação no seguinte endereço: “https://www.example.com/path/resource/?id=123”, o *gateway* de API faz a conversão de forma que o serviço responsável possa interpretar adequadamente. Os Códigos 4.1 e 4.2 apresentam um exemplo da conversão realizada pelos *gateways* da AWS e Azure, respectivamente.

Código 4.1 – Exemplo do evento originado pelo *gateway* de API da AWS

```
1  {
2    "resource": "/",
3    "path": "/path/resource",
4    "httpMethod": "GET",
5    "headers": {
6      "accept": "*/*",
7      "Host": "example.com",
8    },
9    "queryStringParameters": {
10     "id": "123"
11   },
12   "pathParameters": null,
13   "stageVariables": null,
14   "requestContext": {
15     "domainName": "www.exemple.com"
16   },
17   "body": null,
```

[2] Disponível em <https://github.com/satheler/s12r>.

```
  "isBase64Encoded": false
}
```

Código 4.2 – Exemplo do evento no originado pelo *gateway* de API da Azure

```
{
  "method": "GET",
  "url": "https://www.example.com/",
  "originalUrl": "https://www.example.com/path/resource/?id=1
     23",
  "headers": {
    accept: "*/*",
    host: "example.com",
  },
  "query": {
    "id": "123"
  },
  "params": {},
  "body": null,
  "rawBody": null
}
```

Adicionalmente, o *gateway* de API espera uma resposta padronizada do serviço que processou a solicitação. Os Códigos 4.3 e 4.4 apresentam um exemplo da resposta esperada pelos *gateways* da AWS e Azure, respectivamente.

Código 4.3 – Exemplo da resposta esperada pelo *gateway* de API da AWS

```
{
  "headers": {},
  "statusCode": 200
  "body": "Sucesso!",
  "isBase64Encoded": false,
}
```

Código 4.4 – Exemplo da resposta esperada pelo *gateway* de API da Azure

```
{
  "status": 200,
  "body": "Sucesso!",
  "headers": {}
}
```

4.5 Banco de dados

Em nossa implementação utilizamos o PostgreSQL, uma vez que os dois provedores de nuvem o suportam, além de possuir o recurso de replicação. Uma vez que a replicação tenha sido definida e configurada, o processo de recuperação pode ocorrer se o servidor primário para o banco de dados falhar. A detecção pode levar algum tempo, em particular, o PostgreSQL não fornece ferramentas integradas de detecção de falhas no servidor. Em contrapartida, existem ferramentas disponíveis que ajudam a detectar falhas e alternar automaticamente para o conjunto de servidores em modo de espera, minimizando o tempo de inatividade do banco de dados.

Para a replicação, foi adotada a estratégia de *Multi-Master Replication* (MMR), que consiste em definir mais de um banco de dados primário. Desta forma, toda operação que ocorre nos bancos de dados primários são replicados para as tabelas correspondentes nos outros bancos de dados, sendo eles primários ou em espera.

Todas as alterações efetuadas por uma transação são salvas primeiro em um arquivo em forma de registro e, em seguida, o resultado da transação é enviado aos demais bancos. Os próprios arquivos de dados não são alterados em todas as transações. Este é um mecanismo padrão para evitar a perda de dados em caso de circunstâncias como travamento do sistema operacional, falha de hardware ou travamento do banco. Esse mecanismo é denominado *Write-Ahead Logging* (WAL).

Cada operação realizada pela transação (*INSERT*, *UPDATE*, *DELETE*, *COMMIT*) é gravado como um registro no WAL. Os registros WAL são gravados primeiramente em memória. Quando a transação é confirmada, os registros são gravados em um arquivo de segmento WAL no disco.

4.6 Modelo de sistema

Para o nosso modelo de sistema assumimos algumas premissas para determinar alguns dos comportamentos da arquitetura proposta. Os múltiplos provedores de nuvem tem como finalidade receber uma solicitação, processar os dados e montar uma resposta para a solicitação. Esses provedores denominamos de provedores de nuvem de execução. Um provedor é definido como falho, quando algum de seus serviços não estão em pleno funcionamento, ou seja, o provedor é definido como falho por completo. Também, definimos que o balanceamento de cargas deve ser realizado por um provedor ou plataforma sem vínculo com os provedores de nuvem de execução.

Principalmente, definimos uma premissa que o balanceador de cargas, o serviço de DNS e o monitor de disponibilidade não fiquem indisponíveis e que nenhuma falha ocorra. Bem como, nosso modelo é baseado na suposição de que é muito raro que dois grandes provedores de nuvem experimentem uma condição de inatividade simultânea.

Para tolerar a falha de DNS da Cloudflare, poderiamos definir um servidor de

DNS secundário nos provedores de nuvem. Bem como, o mesmo poderia ser feito para o serviço de balancemento de cargas e o monitor de disponibilidade.

4.7 Implantação

Para garantir que a versão da aplicação nos provedores sejam sempre a mesma, foi criado um fluxo de entrega contínua com o *Serverless Framework* e *Github Actions*. O fluxo é acionado no repositório quando a ramificação principal sofre alterações. O arquivo de configuração do *Serverless Framework* descreve todos os recursos que devem ser gerenciados no provedor alvo. Alguns dos exemplos de configurações neste arquivo são: provedor alvo, região, ambiente de execução e variavéis de ambiente, arquivo para executar a função e entre outros.

Ao iniciar o fluxo de trabalho, são instaladas as dependências da aplicação assim como algumas bibliotecas do sistema operacional (*e.g.* OpenSSL). Nota-se que ao utilizar algumas bibliotecas do sistema operacional, é possivel encontrar alguns problemas de compatibilidade de ambientes. Por exemplo, o serviço de FaaS da Azure (Azure Functions), por padrão, utiliza o Windows, enquanto o serviço de FaaS da AWS (AWS Lambda) utiliza Linux para execução das funções. Dessa forma, para a implantação nos provedores foi necessário executar em fluxos de trabalhos distintos, cada fluxo com o sistema operacional compatível.

5 AVALIAÇÃO

Para a avaliação definimos cenários de teste (Seção 5.1), desenvolvemos uma aplicação para o FaaS (Seção 5.2), executamos testes de carga na aplicação (Seção 5.3) para análisar aspectos como disponibilidade e desempenho, apresentamos o ambiente em que os testes foram executados (Seção 5.4), os custos envolvidos (Seção 5.5) e por fim os resultados (Seção 5.6).

5.1 Cenários de testes

Os testes foram definidos em cinco cenários, os quais correspondem a quantidade de usuários ativos simultâneos: 10, 20, 30, 40 e 50. Esta quantidade foi definida de acordo com a menor quantidade de conexões ativas simultâneas que os bancos de dados suportavam. A AWS suportava um limite de 85 conexões, enquanto a Azure suportava 54 conexões. Além disso, para cada cenário foi definida a duração de 10 minutos.

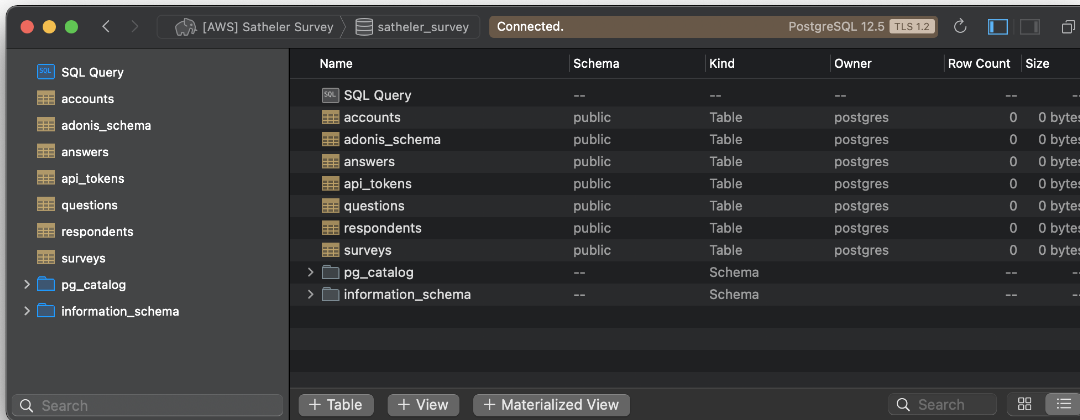
Com o intuito de garantir a consistência entre os testes, isto é, que o teste seja executado sempre nas mesmas condições, algumas ações foram definidas antes do início de cada de teste: (a) certificar-se que os provedores estavam em pleno funcionamento (Figura 4); (b) certificar-se que o banco de dados não continha nenhum registro nas tabelas (Figura 5) e que as conexões ativas do banco sejam mínimas, ou seja, somente as conexões das instâncias invocadas pelos monitores de disponibilidade (Figura 6).

Figura 4 – Visão geral da integridade dos provedores

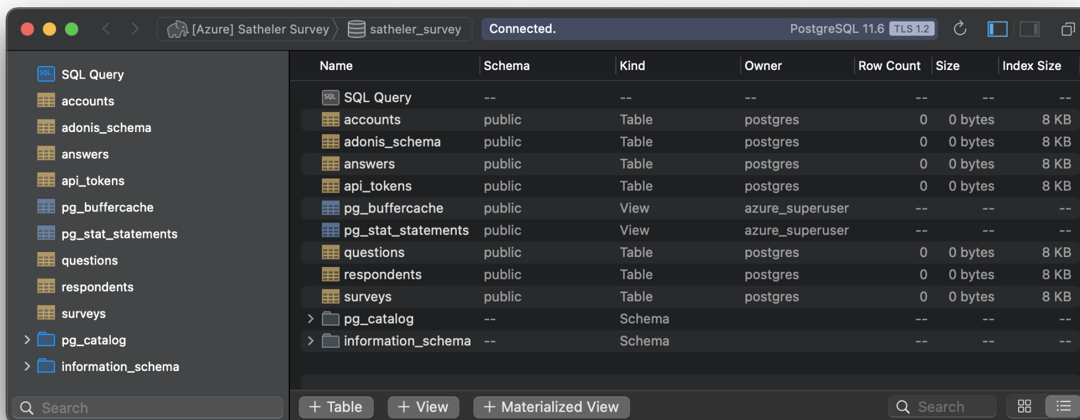
Fonte: Autoral.

Para testar a disponibilidade da aplicação em múltiplos provedores de forma a entender como o balanceador de cargas se comportaria, foi simulada a indisponibilidade no provedor ativo. Para isso, foi criado uma função a parte no AWS Lambda que altera o limite máximo de instâncias ativas simultâneas da aplicação. Por exemplo, ao alterar este valor para zero, determina que nenhuma instância pode estar ativa. Em contrapartida,

Figura 5 – Visão geral das tabelas do banco de dados em ambos provedores.



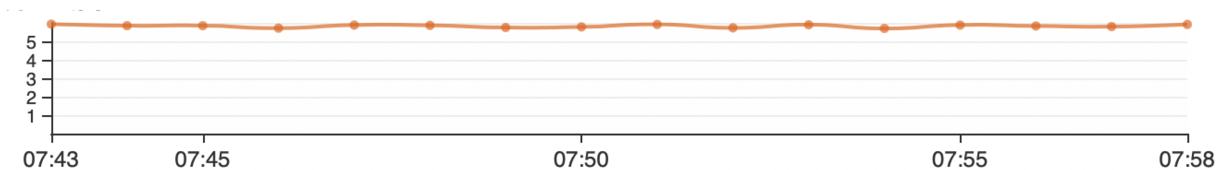
(a) AWS



(b) Azure

Fonte: Autoral.

Figura 6 – Quantidade de conexões ativas no banco de dados



Fonte: Autoral.

qualquer valor acima de zero, determina a quantidade máxima de instâncias ativas. Para executar essa ação, a função foi disponibilizada na forma de API RESTful pelo serviço de *gateway* de API da AWS. O tempo determinado para a indisponibilidade ficou definido da seguinte forma:

- 02 minutos: a nuvem ativa é inativada;
- 04 minutos: a nuvem ativa é restaurada;
- 06 minutos: a nuvem ativa é inativada;
- 08 minutos: a nuvem ativa é restaurada;

Com o propósito de simular o uso da aplicação como um usuário, foi elaborada uma ordem sequencial de ações. A ações são definidas como um par de solicitação e resposta. Também foi considerado que cada usuário deveria usar as principais funcionalidades da aplicação. Em suma, a Tabela 3 mostra como foi definida a sequência das ações e o tamanho aproximado das solicitações e respostas em *bytes*.

Tabela 3 – Sequência de ações

Ordem	Ação	Tamanho (bytes)	
		Solicitação	Resposta
1	Registrar-se	124	300
2	Autenticar-se	62	108
3	Criar uma pesquisa	137	347
4	Incluir perguntas na pesquisa criada	115	239
5	Listar pesquisas em aberto	0	3500
6	Selecionar uma pesquisa de forma aleatória	*	
7	Consultar em detalhes pesquisa selecionada	0	**
8	Responder as perguntas da pesquisa selecionada	**	238

* Ação realizada localmente
** Há variação de acordo com a quantidade de perguntas

5.2 Aplicação

Pesquisas são úteis para obter *feedback* sobre uma variedade de tópicos. Para desenvolvedores esta solução pode ser utilizada para coletar satisfação de aplicativos, levantar requisitos, necessidades futuras, problemas, prioridades, entre outros. Satheler Survey é uma aplicação de pesquisa *online* semelhantemente ao Formulários Google (Google, 2021). A aplicação permite que os usuários criem, administrem e visualizem suas pesquisas. As solicitações são realizadas na forma de uma API RESTful, composto de 18 rotas, cada rota com funcionalidades distintas.

Para o desenvolvimento foi utilizado o *framework* Adonis escrito em *Node.js*. Ele foi escolhido no intuito de seguir uma arquitetura convencional de desenvolvimento, na mesma linha que o Ruby on Rails (Ruby), Laravel (PHP), Spring Boot (Java) entre outros.

Para que a aplicação funcionasse corretamente nos provedores de nuvem sem precisar alterá-la de acordo com o provedor implantado, foi desenvolvido a biblioteca Serverlessizer (S12R). Ao receber o evento do *gateway* de API, a biblioteca faz a conversão

da requisição e a resposta para a aplicação, respectivamente. Assim, a aplicação têm o mesmo comportamento fora do ambiente de FaaS.

5.3 Teste de carga

O teste de carga é realizado para verificar o volume de operações, acessos simultâneos ou usuários que uma aplicação suporta. Dessa forma é possível (a) analisar a estabilidade em um período de grande acesso de modo a estabelecer um limite de operação, encontrar itens do sistema que podem vir a falhar ou implicar em erros durante momentos de grande carga; (b) verificar como o sistema se comporta enquanto a carga de informações vai aumentando, de modo a verificar lentidão, falhas, bugs e entre outros. Diferentemente de testes funcionais ou de regressão, nos quais resultados de aprovação ou reprovação são definidos, em um teste de carga os resultados são muito menos nítidos e dependem da interpretação para identificar se são satisfatórios. Desta forma avaliamos nossa solução usando as seguintes métricas:

- Disponibilidade: A disponibilidade é medida usando o tempo de recuperação, que é o tempo entre a primeira reação à falha e o momento em que o serviço está disponível novamente. Também capturamos o código de estado do HTTP retornado na resposta, sendo o código 500 que determina a indisponibilidade.
- Desempenho: O desempenho é medido usando a taxa de transferência e tempo de resposta. A taxa de transferência é o número de solicitações atendidas por segundo e o tempo de resposta é o tempo entre uma solicitação do usuário e a resposta do sistema.

Existem ferramentas que permitem aplicar testes de carga, como Tsung e o Apache JMeter™. Decidimos usar o Apache JMeter™ já que é uma ferramenta de código aberto (Apache Sofware Foundation, 1999), com atualizações recentes e que permite fazer comunicação com a API.

Os testes foram reproduzidos 10 vezes para cada cenário de teste e por fim realizado uma média dos valores.

5.4 Ambiente de testes

Os testes foram executados em um MacBook Air, com o processador Apple Silicon M1 com 8 núcleos e 8 *gigabytes* de memória RAM. O equipamento estava localizado na cidade de Porto Alegre, Rio Grande do Sul. Os provedores de nuvem (AWS e Azure) foram configurados para a região da América Latina, especificamente no estado de São Paulo.

Durante os testes, foi analisado também a latência média dos provedores. A AWS teve uma média de 20,74 milissegundos. Já com a Cloudflare teve uma média de 19,13

milissegundos. Infelizmente, não foi possível realizar a análise na Azure devido as algumas restrições da plataforma, além do tempo disponível para a realização deste trabalho e pela falta de conhecimento do autor.

5.5 Custo

A Tabela 4 detalha o custo mensal de cada um dos recursos utilizados dos provedores de nuvem. Os custos são baseados em algumas váriaveis. A precificação do *gateway* de API se dá pela quantidade de solicitações mensais. Já a precificação das funções se dá pela quantidade de memória alocada, tempo de execução máximo e a quantidade de execuções por mês. Por fim, a precificação do banco de dados se dá pelo tipo de processador da instância alocada, tempo de funcionamento mensal (em horas) e quantidade de armazenamento (em *gigabyte*).

Para o nosso cálculo levamos em consideração a quantidade de solicitações mensais que o monitor de disponibilidade faz nos provedores de nuvem, como mostrada na Equação 5.1. Na qual, $n_{servidores}$ é a quantidade de servidores que realizam a verificação e o $t_{intervalo}$ é o intervalo de tempo (em segundos) entre as verificações do monitor de disponibilidade. Já o valor de 43.200 representa a quantidade total de segundos em um mês.

$$solicitações_{mensais} = n_{servidores}(43.200(\frac{60}{t_{intervalo}})) \tag{5.1}$$

Desta forma, como a Cloudflare utiliza 13 servidores para realizar a verificação e, o intervalo de verificação é de 60 segundos, então são realizadas 561.600 solicitações em cada um dos provedores. Consideramos também que a AWS por ser o provedor ativo, recebe a quantidade de solicitações mensais somado à quantidade de solicitações mensais do monitor de disponibilidade.

Como os valores dos serviço nos provedores são precificados em dólar americano, de forma a simplificar os valores, definimos a equivalência de 5 reais para cada 1 dólar.

Tabela 4 – Custos dos provedores de nuvem

	AWS	Azure
Gateway de API	12,40	15,50
Funções	164,20	24,15
Banco de dados	310,15	269,75
Total	486,75	309,40

Portanto, para manter a arquitetura redundante na Azure têm-se um custo adicional de 63,56%. Percebe-se também que o banco de dados é o serviço que possui o maior custo em ambos os provedores.

5.6 Resultados

A Equação 5.2 mostra o valor em porcentagem do tempo de indisponibilidade da aplicação no pior caso de acordo com a configuração do monitor de disponibilidade e o cenário de teste. Onde n_{falhas} é a quantidade de vezes que o provedor ativo fica indisponível durante os testes, $t_{intervalo}$ é o intervalo de tempo (em segundos) entre as verificações do monitor de disponibilidade, t_{limite} é o tempo limite (em segundos) antes de definir a verificação de disponibilidade como falha, $n_{tentativas}$ é a quantidade de tentativas realizadas pelo monitor e o t_{total} representa a duração total do teste (em segundos).

$$\frac{indisponibilidade}{100} = \frac{n_{falhas}(t_{intervalo} + t_{limite}(n_{tentativas} + 1))}{t_{total}} \tag{5.2}$$

A Figura 7 mostra o resultado esperado conforme a configuração do monitor de disponibilidade e o cenário de teste. Os testes tiveram uma duração de 600 segundos (10 minutos) e foram configurados da seguinte forma:

- duas simulações de falha no provedor ativo;
- intervalo de verificação de 60 segundos;
- tempo limite de 5 segundos;
- duas tentativas de verificação;

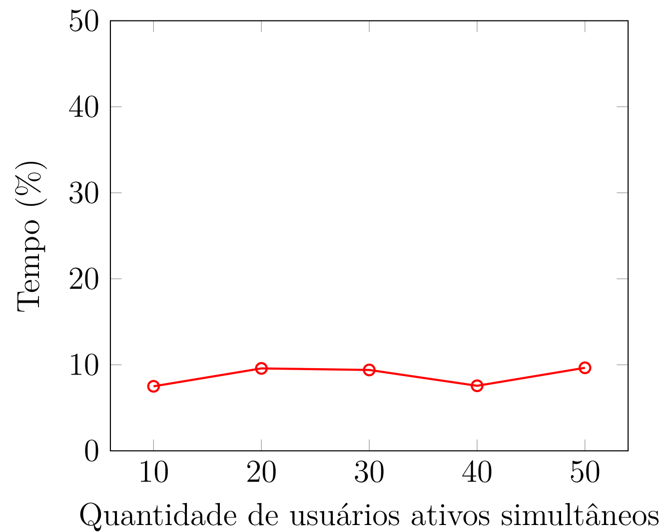
Desta forma, utilizando a Equação 5.2, a aplicação manteve-se abaixo dos 25% do tempo de inatividade, sendo 19,41% o pior caso dentre todos os testes. A média de indisponibilidade da aplicação por cada cenário ficou entre 7,50% (equivalente a 45 segundos) e 9,65% (equivalente a 58 segundos).

Da mesma maneira, mantendo os cenários de testes e utilizando as melhores configurações oferecidas pelo monitor de disponibilidade têm-se:

- intervalo de verificação de 5 segundos;
- tempo limite de 1 segundo;
- sem tentativas de verificação;

Desta forma, é possível obter apenas 2% de indisponibilidade da aplicação no pior caso, equivalente a 12 segundos. Similarmente, se aplicarmos em um cenário na qual ocorre 4 falhas graves no provedor de nuvem ativo no período de 1 ano, ou seja, que o balanceador de cargas precise alternar o tráfego 4 vezes para o provedor de nuvem passivo, teriamos o total de 24 segundos de indisponibilidade por ano, o equivalente à 99,999999% (8 noves) de disponibilidade. Eventualmente, um dos desafios para atingir a alta disponibilidade pode ser o monitor de disponibilidade e o intervalo para detecção das falhas.

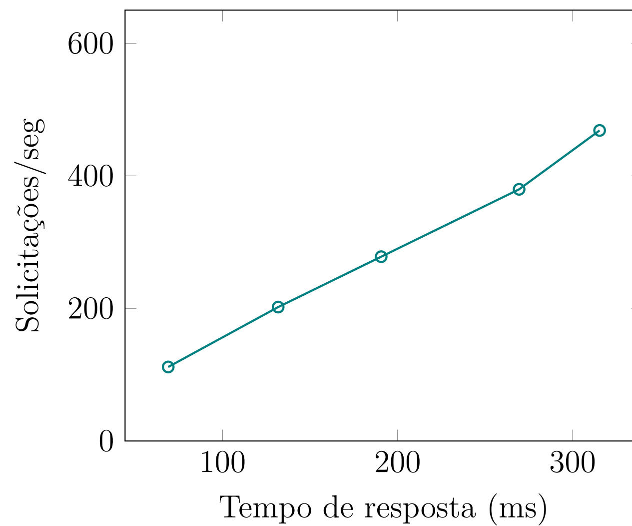
Figura 7 – Tempo de indisponibilidade da aplicação



Fonte: Autoral.

A Figura 8 mostra o desempenho da aplicação. O tempo médio de resposta manteve-se linear a medida que a quantidade de solicitação por segundo aumentava. Usando a Tabela 3 como referência, cada usuário no teste enviava aproximadamente 438 bytes e recebia aproximadamente 4.822 bytes.

Figura 8 – Tempo de resposta por solicitações por segundo



Fonte: Autoral.

6 CONSIDERAÇÕES FINAIS

Neste capítulo, iniciamos com um resumo das principais contribuições do trabalho. Na sequência, apresentamos alguns exemplos de trabalhos futuros.

6.1 Contribuições

As principais contribuições deste trabalho são: (a) proposta da arquitetura de alta disponibilidade com FaaS; (b) avaliação da arquitetura; (c) implementação do S12R. A proposta de arquitetura de alta disponibilidade de funções como serviço em ambiente de múltiplas nuvens de computação utilizou-se da estratégia de redundância ativo-passivo. A proposta foi avaliada através de testes de cargas com a ferramenta Apache JMeter™. A implementação funcional, incluindo o arquivo de testes, a configuração do Serverless Framework e implantação contínua está disponível em <https://github.com/satheler/satheler-survey>. As configurações no âmbito dos provedores (AWS e Azure) através da plataforma *web* ou via linha de comando (terminal) podem ser encontrados nos anexos deste trabalho.

Os resultados mostram que é possível fazer o balanceamento entre provedores nuvens AWS e Azure com FaaS usando o *gateway* de API como a "porta de entrada". Os testes mostram que a indisponibilidade da aplicação ficou abaixo do máximo esperado, incluindo o monitor de disponibilidade.

A implementação do Serverlessizer (S12R) permitiu executar aplicações que não foram projetadas para o ambiente FaaS em provedores de nuvem (AWS e Azure) sem a necessidade de alterar a aplicação. A biblioteca é de código aberto e está disponível em <https://github.com/satheler/s12r>.

6.2 Trabalhos Futuros

Para trabalhos futuros, podem ser estudadas arquiteturas baseadas em outros tipos de redundância (*e.g.* ativo-ativo). Também, uma comparação de custo e desempenho da mesma aplicação em microsserviços e FaaS no âmbito da alta disponibilidade em múltiplos provedores de nuvem. Como ainda, incluir outros provedores de nuvem com intuito de comparar custos de recursos e serviços.

REFERÊNCIAS

ADDO, I. D.; AHAMED, S. I.; CHU, W. C. A reference architecture for high-availability automatic failover between paas cloud providers. In: IEEE. **2014 International Conference on Trustworthy Systems and their Applications**. [S.l.], 2014. p. 14–21. Citado na página 27.

ADZIC, G.; CHATLEY, R. Serverless computing: economic and architectural impact. In: **Proceedings of the 2017 11th joint meeting on foundations of software engineering**. [S.l.: s.n.], 2017. p. 884–889. Citado 2 vezes nas páginas 25 e 29.

AHLUWALIA, K. S.; JAIN, A. High availability design patterns. In: **Proceedings of the 2006 conference on Pattern languages of programs**. [S.l.: s.n.], 2006. p. 1–9. Citado na página 26.

ALI, M. F. **Study of load balancing strategies for the distributed computing system**. Tese (Doutorado) — Aligarh Muslim University, 2015. Citado na página 31.

Apache Sofware Foundation. **Apache JMeter™**. 1999. Online. Acessado em 04 de Abril 2021. Disponível em: <http://jmeter.apache.org/>. Citado na página 44.

AZURE, M. **Azure Service-level agreements**. 2021. Disponível em: <https://azure.microsoft.com/en-us/support/legal/sla/>. Citado na página 27.

BORTOLINI, D.; OBELHEIRO, R. R. Investigating performance and cost in function-as-a-service platforms. In: SPRINGER. **International Conference on P2P, Parallel, Grid, Cloud and Internet Computing**. [S.l.], 2019. p. 174–185. Citado na página 29.

BOUIZEM, Y. et al. Active-standby for high-availability in faas. In: **Proceedings of the 2020 Sixth International Workshop on Serverless Computing**. [S.l.: s.n.], 2020. p. 31–36. Citado na página 29.

BUYYA, R. et al. A manifesto for future generation cloud computing: Research directions for the next decade. **ACM Comput. Surv.**, Association for Computing Machinery, New York, NY, USA, v. 51, n. 5, nov. 2018. ISSN 0360-0300. Disponível em: <https://doi.org/10.1145/3241737>. Citado na página 25.

Canalys. **Global cloud infrastructure market Q3 2020**. 2021. Online. Acessado em 20 de Fevereiro 2021. Disponível em: <https://www.canalys.com/newsroom/worldwide-cloud-market-q320>. Citado na página 35.

COULOURIS, G. et al. **Sistemas Distribuídos-: Conceitos e Projeto**. [S.l.]: Bookman Editora, 2013. Citado na página 29.

EIVY, A.; WEINMAN, J. Be wary of the economics of "serverless"cloud computing. **IEEE Cloud Computing**, IEEE, v. 4, n. 2, p. 6–12, 2017. Citado 2 vezes nas páginas 25 e 29.

Google. **Google Forms**. 2021. Online. acessado em 29 Maio 2021. Disponível em: <https://www.google.com/intl/pt-BR/forms/about/>. Citado na página 43.

JACKSON, D.; CLYNCH, G. An investigation of the impact of language runtime on the performance and cost of serverless functions. In: IEEE. **2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)**. [S.l.], 2018. p. 154–160. Citado na página 29.

KEMME, B.; JIMÉNEZ-PERIS, R.; PATIÑO-MARTÍNEZ, M. Database replication. **Synthesis Lectures on Data Management**, Morgan & Claypool Publishers, v. 5, n. 1, p. 1–153, 2010. Citado na página 33.

NABI, M.; TOEROE, M.; KHENDEK, F. Availability in the cloud: State of the art. **Journal of Network and Computer Applications**, Elsevier, v. 60, p. 54–67, 2016. Citado na página 26.

NGINX. **What Is Load Balancing?** 2021. Disponível em: <https://www.nginx.com/resources/glossary/load-balancing/>. Citado na página 32.

REDHAT. **What does an API gateway do?** 2021. Disponível em: <https://www.redhat.com/en/topics/api/what-does-an-api-gateway-do>. Citado na página 32.

ROBERTS, M. **Serverless Architectures**. 2018. Online. acessado em 23 de Maio de 2021. Disponível em: <https://martinfowler.com/articles/serverless.html>. Citado na página 25.

SAHA, I.; MUKHOPADHYAY, D.; BANERJEE, S. Designing reliable architecture for stateful fault tolerance. In: IEEE. **2006 Seventh International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT'06)**. [S.l.], 2006. p. 545–551. Citado na página 33.

SERVICES, A. W. **AWS Service Level Agreements (SLAs)**. 2021. Disponível em: <https://aws.amazon.com/legal/service-level-agreements/>. Citado na página 26.

SOLDANI, J.; TAMBURRI, D. A.; HEUVEL, W.-J. V. D. The pains and gains of microservices: A systematic grey literature review. **Journal of Systems and Software**, Elsevier, v. 146, p. 215–232, 2018. Citado na página 26.

Synergy Research Group. **Cloud Market Growth Rate Nudges Up as Amazon and Microsoft Solidify Leadership**. 2021. Online. acessado em 20 de Fevereiro 2021. Disponível em: <https://www.srgresearch.com/articles/cloud-market-growth-rate-nudges-amazon-and-microsoft-solidify-leadership>. Citado na página 35.

The Linux Foundation. **Kubernetes**. 2021. Online. acessado em 31 Março 2021. Disponível em: <https://kubernetes.io/>. Citado na página 26.

THÖNES, J. Microservices. **IEEE software**, IEEE, v. 32, n. 1, p. 116–116, 2015. Citado na página 25.

VAQUERO, L. M. et al. **A break in the clouds: towards a cloud definition**. [S.l.]: ACM New York, NY, USA, 2008. Citado na página 25.

VAYGHAN, L. A. et al. Microservice based architecture: Towards high-availability for stateful applications with kubernetes. In: IEEE. **2019 IEEE 19th International Conference on Software Quality, Reliability and Security (QRS)**. [S.l.], 2019. p. 176–185. Citado na página 26.

WANG, L. et al. Peeking behind the curtains of serverless platforms. In: **2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)**. [S.l.: s.n.], 2018. p. 133–146. Citado na página 25.

Anexos

ANEXO A – CONFIGURAÇÃO NA AMAZON WEB SERVICES (AWS)

Após o cadastro na plataforma é necessário configurar as credenciais para o *Serverless Framework*. Para criar e gerenciar recursos do provedor, basta acessar *Identity and Access Management* (IAM) dentro da plataforma e criar um usuário com uma política de acesso adequada para conceder acesso aos recursos.

Após o registro - Criar um usuário no IAM com as devidas permissões. - Salvar as credenciais, serão utilizadas no Serverless Framework para fazer o deploy. AWS Criar uma instância de banco de dados no AWS RDS - Basta selecionar o banco o tipo do banco de dados (no nosso caso foi utilizado o PostgreSQL) e a senha para conexão. - Identificador - E permitir o acesso externo

O banco então é criado em uma instância com um endereço atrelado.

Para acessar ter acesso ao banco externamente, ainda é necessário configurar o grupos de segurança para permitir o tráfego de entrada e saída da instância nas portas selecionadas. Por exemplo, para permitir o acesso a instância do PostgresSQL, é necessário informar que é permitido o tráfego na porta 5432 e quais de quais endereços tem a permissão de acesso.

Agora basta, colocar todas as informações nas variáveis de ambiente e rodar o comando que faz a implantação através do Serverless Framework.

Com a aplicação implantando, queremos usar um domínio para atender as chamadas.

Adicionar o domínio no site. - Criar um certificado SSL para o domínio através do AWS ACM. - É necessário realizar algumas validações nas configurações de DNS conforme pede, para verificar se o domínio pertence a você. - No API Gateway, adicionar nome de domínio personalizado. - Nome do domínio. - Nas configurações do endpoint selecionar o certificado ACM anteriormente adicionado. - Criar nome de domínio - Na aba Mapeamento de API, adicionamos a nossa API implantada. - Na aba de configurações temos o nome de domínio gerado, e usamos este endereço no balanceador de cargas.

Agora já dá para adicionar no balanceador de cargas a API da AWS. O cabeçalho de host deve ser um subdomínio de uma zona associada a essa conta, deve corresponder ao endereço da origem ou deve ser resolvido publicamente para o endereço de origem.

ANEXO B – CONFIGURAÇÃO NA MICROSOFT AZURE

Após o registro - É necessário baixar e instalar o “Azure CLI”. Diferentemente da AWS que as credenciais são gerenciadas na plataforma web, a Azure faz o gerenciamento pelo terminal. - Para gerar as credencias é necessário rodar o comando ‘az login‘ - Com as credencias geradas, precisamos do código da assinatura (tenantId) e o identificador do usuário (id). - A resposta vai ser algo como aparece no Código B.1.

Código B.1 – Resposta esperada após autenticação na Microsoft Azure

```
[
  {
    "cloudName": "AzureCloud",
    "homeTenantId": "457fede9-ad4c-4dc5-9f87-a303a87f1e0c",
    "id": "cd35715b-b709-43c1-b97b-dc0650425d60",
    "isDefault": true,
    "managedByTenants": [],
    "name": "Free Trial",
    "state": "Enabled",
    "tenantId": "457fede9-ad4c-4dc5-9f87-a303a87f1e0c",
    "user": {
      "name": "your_email@outlook.com",
      "type": "user"
    }
  }
]
```

- Para criar o serviço na qual as funções serão implantadas é necessário rodar o comando ‘az ad sp create-for-rbac‘ - A resposta será próxima ao Código B.2

Código B.2 – Resposta esperada após autenticação na Microsoft Azure

```
{
  "appId": "216c4cfb-4952-4e6e-81ca-2d0e19779eaf",
  "displayName": "azure-cli-2021-04-28-17-38-19",
  "name": "http://azure-cli-2021-04-28-17-38-19",
  "password": "dCLZ5hQKpXCKWLZ7B9Um_bR1b686Qn.6q-",
  "tenant": "f5523bb2-7065-44c0-9ea0-5e56d39e82a8"
}
```

As duas respostas já são suficientes para configurar nossas variáveis de ambiente. Para isso será necessário

Código B.3 – Resposta esperada após autenticação na Microsoft Azure

```
1 export AZURE_SUBSCRIPTION_ID='<id>' # azure-login-response
2 export AZURE_TENANT_ID='<tenant>' # azure-service-create-
    response
3 export AZURE_CLIENT_ID='<name>' # azure-service-create-
    response
4 export AZURE_CLIENT_SECRET='<password>' # azure-service-
    create-response
```

Com o serviço criado, podemos criar um banco de dados na plataforma.

Aplicativo de funções - Adicionar um “Domínio personalizado” e configurar no seu DNS - Na seção de “Configurações de TLS/SSL” - Criar um certificado de chave privada - Criar Certificado Gerenciado do Serviço de Aplicativo - Selecionar o domínio - Clicar no botão “Criar” - Em “Domínios personalizados” - Clicar em “Adicionar associação” - Adicionar o certificado gerado

ANEXO C – CONFIGURAÇÃO NA CLOUDFLARE

Após o cadastro na plataforma, o domínio adquirido foi apontado para o serviço de gerenciamento de DNS da Cloudflare. Embora a plataforma ofereça diversos serviços gratuitos (*e.g.* gerenciamento de DNS, certificados SSL, *Content Delivery Network*, etc), o serviço de balanceamento de cargas é oferecido a parte, ou seja, é cobrado por seu uso. Antes de criar o balanceador de cargas é necessário criar os monitores e as *pools*. Para criar o monitor, basta informar o nome para identificá-lo na plataforma, protocolo de comunicação (*e.g. HTTPS*) e o caminho relativo que realizará a verificação de integridade da aplicação. Por exemplo, se o caminho for definido como "*/health*", então a verificação nos provedores será realizada da seguinte forma:

- Na AWS, https://api-id.execute-api.region.amazonaws.com**/health**
- Na Azure, https://sls-api-id.azurewebsites.net**/health**

Também é possível configurar os intervalos entre cada verificação, o tempo de duração máximo da requisição, quantidade de tentativas, padrão de caracteres da resposta, entre outros. Caso o tempo de duração e a quantidade de tentativas sejam extrapolados ou o padrão da resposta não corresponda ao esperado a *pool* é considerada como degradada. As *pools* são as responsáveis por apontar para os endereços associados aos provedores em nuvem. Basta Para criar um balanceador de cargas na plataforma, é necessário informar o domínio que será utilizado para receber as requisições (*e.g. api.satheler.survey.dev*).

ÍNDICE

API, 31, 32, 35, 37, 38, 42–45, 49
AWS, 26, 35, 37, 38, 40–42, 44, 45, 49, 57, 59, 61
Azure, 26, 35, 37, 38, 40–42, 44, 45, 49, 59, 61

DNS, 35, 36

FaaS, 25–29, 32, 33, 35, 40, 41, 44, 49

gRPC, 32

HTTP, 36, 44

IaaS, 25, 26

MMR, 39

PaaS, 25, 26

RESTful, 32, 42, 43

S12R, 37, 43, 49
SaaS, 25

WAL, 39