UNIVERSIDADE ESTADUAL DO SUDOESTE DA BAHIA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

PABLO BITTENCOURT PEREIRA GOBIRA

UMA ANÁLISE COMPARATIVA ENTRE A ARQUITETURA CLIENTE-SERVIDOR E ARQUITETURA SERVERLESS PARA APLICAÇÕES HTTP

VITÓRIA DA CONQUISTA - BA
2019

PABLO BITTENCOURT PEREIRA GOBIRA

UMA ANÁLISE COMPARATIVA ENTRE A ARQUITETURA CLIENTE-SERVIDOR E ARQUITETURA SERVERLESS PARA APLICAÇÕES HTTP

Trabalho de Conclusão de Curso apresentado ao colegiado do Curso de Ciência da Computação da Universidade Estadual do Sudoeste da Bahia, como requisito parcial para obtenção do grau de Bacharel em Ciência da Computação.

Orientador: Prof. Dr. Roque Mendes Prado Trindade

VITÓRIA DA CONQUISTA - BA

2019

PABLO BITTENCOURT PEREIRA GOBIRA

UMA ANÁLISE COMPARATIVA ENTRE A ARQUITETURA CLIENTE-SERVIDOR E ARQUITETURA SERVERLESS PARA APLICAÇÕES HTTP

Trabalho de Conclusão de Curso apresentado ao colegiado do Curso de Ciência da Computação da Universidade Estadual do Sudoeste da Bahia, como requisito parcial para obtenção do grau de Bacharel em Ciência da Computação.

BANCA EXAMINADORA

_____

Prof. Dr. Roque Mendes Prado Trindade

Universidade Estadual do Sudoeste da Bahia

_____

Profª. Drª. Maísa Soares dos Santos Lopes

Universidade Estadual do Sudoeste da Bahia

_____

Prof. Dr. Hélio Lopes dos Santos

Universidade Estadual do Sudoeste da Bahia

Vitória da Conquista – BA, 25 de março de 2019

AGRADECIMENTOS

À minha mãe, Claudia, que me criou com tanto zelo, carinho, atenção e amor. Por sempre me ensinar o que é certo e me guiar pelo caminho da corretude e honestidade, passando a mim as grandes virtudes a ela ensinadas por seus pais. Por ter paciência e compreensão para suportar minha personalidade forte. Eu me sinto privilegiado por ter alguém tão iluminado ao meu lado, cuidando de mim e me incentivando a ser melhor a cada dia que passa.

À minha avó, Amália, que foi uma das pessoas mais importantes da minha criação, sempre me amou de forma tão pura e verdadeira, sempre me apoiou e quis o meu melhor. Muito obrigado por me dar tanto carinho, que me preencheu e continuará a preencher por toda a minha vida. A senhora permanece constantemente em meus pensamentos.

Ao meu avô, pessoa tão integra, honesta, trabalhadora e alegre, que passou aos seus filhos e netos todos esses valores, que por sua vez foram passados a mim por minha mãe e que certamente passarei aos meus filhos quando for o momento. O senhor é minha maior figura paterna e tem de mim a maior admiração possível.

Ao meu padrasto, Paulo, que tantas oportunidades me deu nessa vida, me permitindo trilhar o caminho que trilho hoje. Você sempre foi um exemplo de integridade, dedicação ao trabalho e cuidado com aqueles que estão ao seu redor. Sem você jamais estaria aqui hoje.

Ao meu pai, Verissimo Gobira Neto, por ter me dado a vida e mesmo que distante desejado meu melhor e torcido pelo meu sucesso.

À minha namorada, Penellopy, com quem compartilho sonhos, planos, objetivos, frustrações e sucessos. Você está sempre do meu lado me dando seu carinho, me apoiando e me ajudando a me reerguer quando caio.

Aos meus tios e tias, em especial minha tia Rosangela, sempre tão preocupada comigo e presente em minha vida.

Aos meus primos e amigos, que sempre me acolheram, incentivaram e estiveram do meu lado quando precisei. Tenho um carinho enorme por vocês.

Aos meus colegas de turma e amigos que fiz na UESB, pessoas que trilharam essa etapa tão importante da minha vida comigo, e que tornaram essa experiência muito mais leve e agradável.

A secretária do colegiado, Celina, por fazer seu trabalho com tamanha atenção, carinho e acolher os alunos sempre que precisam.

Ao meu orientador, Professor Dr. Roque Mendes Prado Trindade, que me proporcionou uma excelente tutoria, assim como ministrou ótimas aulas durante o curso, sempre auxiliando seus alunos a crescer e instigando a curiosidade e vontade de buscar conhecimento. Tenho grande admiração por sua forma tão leve de conduzir os trabalhos e pela sua grande humildade, virtude que tanto prezo.

Aos professores do curso de Ciência da Computação da UESB como um todo, por tudo que me ensinaram durante essa jornada de aprendizado.

À UESB, instituição que me acolhe desde 2011, me proporcionando o aprendizado necessário para que eu pudesse me tornar a pessoa que sou hoje.

“Se você não é teimoso, abandonará experimentos cedo demais. E se você não é flexível, baterá a cabeça contra a parede e não verá uma solução diferente para o problema que está tentando resolver”.

Jeff Bezos

RESUMO

Este trabalho tem como objetivo criar uma aplicação *web*, desenvolver duas arquiteturas para executá-la, - sendo uma a arquitetura cliente-servidor convencional e outra a arquitetura sem servidor – e, por fim, investigar os pontos positivos e negativos de cada arquitetura, através de uma série de testes e análises comparativas. A primeira etapa consistiu em desenvolver a aplicação web, utilizando o framework Laravel, que permite criar um projeto mais estruturado e organizado e dá acesso a uma série de úteis extensões feitas pela comunidade. Para a segunda etapa, que consistiu em desenvolver as duas arquiteturas a serem estudadas e executar a aplicação através delas, utilizamos a Amazon Web Services, plataforma que disponibiliza uma série de serviços e ferramentas para desenvolvedores web, peças-chave que permitiram a criação da infraestrutura necessária para colocar em prática a ideia desse trabalho. Para a implementação da arquitetura cliente-servidor, foi configurada uma instância EC2 capaz de servir a aplicação. Na arquitetura serverless, foi utilizado o plugin Bref, responsável por facilitar a configuração da arquitetura em conjunto com as ferramentas da Amazon, assim como a implantação da aplicação, uma vez concluído o seu desenvolvimento. Por fim, foram feitos testes e análises em cima da aplicação em cada arquitetura, levando em conta os parâmetros performance, custo, compatibilidade e gerenciamento. Através dos resultados obtidos, chegamos à conclusão de que a arquitetura *serverless* ainda não possui um fator diferencial necessário para substituir a arquitetura cliente-servidor.

Palavras-Chaves: Sem Servidor. AWS. Lambda. Servidor. Aplicação Web.

ABSTACT

This work aims to create a web application, to develop two architectures to execute it, - one being the conventional client-server architecture and the other the serverless architecture - and, finally, investigating the positives and negatives of each of them, through a series of tests and comparative analyzes. The first step was to develop the web application, using the Laravel framework, that allows to create a more structured and organized project, and gives access to a series of useful extensions constructed by the community. For the second stage, which consisted of developing the two architectures to be studied and running the application through them, we used Amazon Web Services, a platform that provides a series of services and tools for web developers, key pieces that allowed the creation of the necessary infrastructure to put into practice the idea of this work. For the implementation of the client-server architecture, an EC2 instance capable of serving the application was configured. In the serverless architecture, the Bref plugin was used, responsible for facilitating the architecture's configuration in conjunction with the Amazon tools, as well as the implementation of the application, once its development was completed. Finally, tests and analyzes were done on the application in each architecture, taking into account the parameters performance, cost, compatibility and management. Through the obtained results, we concluded that the serverless architecture does not yet have a differential factor necessary to replace the client-server architecture.

Keywords: Serverless. AWS. Lambda. Server. Web Application.

LISTA DE SIGLAS

AWS: Amazon Web Services;

S3: Simple Storage Service;

RDS: Relational Database Service;

API: Application Programming Interface;

PHP: Hypertext Preprocessor;

EC2: Elastic Compute Cloud;

IP: Internet Protocol;

FAAS: Function as a Service;

DBAAS: Database as a Service;

PAAS: Platform as a Service;

MVC: Model-View-Control;

IAM: Identity and Access Management;

FTP: File Transfer Protocol;

VSFTPD: Very Secure FTP Daemon;

URL: Uniform Resource Locator;

RAM: Random Access Memory;

SCP: Secure Copy Protocol;

SSH: Secure Shell;

RSA: Rivest-Shamir-Adleman;

DNS: Domain Name System;

IAM: Identity and Access Management;

NAT: Network Address Translation;

ENI: Elastic Network Interface;

JMX: Java Management Extensions;

EUA: Estados Unidos da América;

MB: Megabyte(s);

GB: Gigabyte(s);

MS: Milissegundo(s);

AMI: Amazon Machine Image;

VPC: Virtual Private Cloud.

LISTA DE FIGURAS

SUMÁRIO

1. INTRODUÇÃO

A internet já não é mais um luxo como era alguns anos após sua criação. Nos tempos modernos ela se configura mais como um bem comum e, sem dúvida, é o maior veículo de informação hoje, uma vez que dá ao usuário a liberdade de buscar pelo que lhe interessa. Através dela, são propagadas informações de todo o tipo, que chegam a uma gama enorme de usuários todos os dias. Devido à sua relevância, tornou-se praticamente indispensável para a maioria das pessoas possuir alguma representatividade na rede.

Essa representação pode ser feita através de páginas em redes sociais como Facebook e Twitter, ou perfis em aplicativos como o Instagram. No entanto, muitos procuram representatividade não só pessoal, mas também para seu negócio, empresa ou organização. Nesse caso, as empresas normalmente optam por utilizar maneiras mais formais e confiáveis para apresentar se serviço na rede para sua clientela, e para alcançar tal propósito, os sites são a melhor opção. No entanto, criar e pôr no ar um site é mais complexo e custoso do que criar um perfil em uma rede social. Para garantir que o site esteja no ar e em boas condições de funcionamento, é necessário encontrar uma forma de hospedagem e gerenciamento que seja eficiente e tenha um bom custo benefício.

> Nas configurações iniciais, todos os aplicativos eram executados em sua própria máquina física. Os altos custos de comprar e manter um grande número de máquinas, e o fato de que cada uma delas era frequentemente subutilizada, levou a um grande avanço: a virtualização. (HENDRICKSON et al., 2016, p. 1, tradução nossa)[1].

A forma de hospedagem mais utilizada desde os primórdios das aplicações web (aplicações que funcionam na internet, como *sites*) e até os dias atuais, consiste em utilizar uma máquina física (servidor) para armazenar e servir o conteúdo dos sites. Como Hendrickson explica, tal arquitetura apresenta uma série de problemas e limitações, que em certos casos podem ser obstáculos críticos para alcançar os objetivos previamente mencionados.

[1] “In early settings, every application ran on its own physical machine. The high costs of buying and maintaining large numbers of machines, and the fact that each was often underutilized, led to a great leap forward: virtualization”.

Então, por que não tentar alcançar esses objetivos utilizando uma tecnologia recente dentro da área de infraestrutura de servidores em vez de usar soluções convencionais? Para adereçar esta questão, iremos utilizar a recente tecnologia *serverless*, também conhecida como Função como Serviço (*Function as a Service*), e fazer um paralelo entre suas qualidades e defeitos em relação às técnicas comumente utilizadas.

O intuito desse trabalho é investigar a viabilidade da criação e utilização da arquitetura sem servidor (*serverless*), assim como seus pontos positivos e negativos em comparação com a arquitetura convencional, também chamada de cliente-servidor. Serão utilizadas como ferramentas o framework Laravel, para a criação de uma aplicação PHP, e a Amazon Web Services (AWS) para construir a infraestrutura necessária para servir a aplicação nas duas arquiteturas.

1.1. OBJETIVOS E JUSTIFICATIVA

Quando se encontra uma boa solução para um problema, a tendência é que a sua utilização se torne um padrão. Ou seja, aquela solução tende a ser utilizada sempre que aquele problema aparecer. Embora possuir uma solução satisfatória para um certo contratempo seja algo positivo, uma vez que ela se torna muito difundida, tende a levar as pessoas ao comodismo em não buscar formas ainda melhores para solucionar o problema.

A busca por novos conhecimentos é indispensável para o progresso intelectual, portanto é preciso estudar com atenção as novas tecnologias, mesmo que aquelas já estabelecidas supram as necessidades existentes no mercado ou no dia a dia. Sabemos que hoje já existem práticas amplamente utilizadas de hospedagem de sites e aplicações web. A ideia de pesquisar sobre o tema deste trabalho vem da curiosidade em investigar se a infraestrutura de hospedagem sem servidor pode ser mais vantajosa que as formas mais convencionais de hospedagem.

1.1.1. Objetivo Geral

O objetivo deste trabalho é descobrir os benefícios e as desvantagens em utilizar a arquitetura de hospedagem *serverless* para aplicações web, fazendo um paralelo com o método hospedagem mais utilizadas no mercado, a arquitetura cliente-servidor.

1.1.2. Objetivos Específicos

1. Criar uma aplicação PHP, utilizando Laravel, com as funcionalidades de vitrine, newsletter e painel administrativo;

2. Construir um banco de dados para a aplicação;

3. Criar e configurar um servidor remoto para hospedar a aplicação do item 1;

4. Configurar as ferramentas disponibilizadas pela Amazon Web Services a fim de implementar a aplicação do item 1 na arquitetura *serverless*;

5. Fazer uma série de testes para elucidar as qualidades e defeitos de cada arquitetura em comparação com a outra.

Ao completar os quatro primeiros objetivos específicos, teremos uma aplicação PHP rodando em ambas as arquiteturas, o que consiste na plataforma de trabalho ideal, uma vez que o propósito final do trabalho é criar uma análise comparativa completa. Assim é possível fazer uma gama de testes capazes de trazer informações precisas sobre o desempenho de cada uma dessas arquiteturas e, por fim, gerar as conclusões desejadas.

Optamos por integrar as funcionalidades escolhidas (vitrine, newsletter e painel administrativo) à aplicação para que ela seja coerente com a realidade e se assemelhe a aplicação de um empreendimento real.

2. REFERÊNCIAL TEÓRICO

Esta seção é destinada a contar brevemente a história do surgimento da tecnologia *serverless*, mostrando, desde a origem, a idealização do conceito de computação em nuvem que é a tecnologia responsável por possibilitar a existência das arquiteturas sem servidor nos dias de hoje.

2.1. O SURGIMENTO DA TECNOLOGIA SERVERLESS

A tecnologia *serverless* não surgiu do dia para a noite. Ela é o resultado de uma série de avanços tecnológicos relacionados à rede de computadores (internet), virtualização e formas de armazenamento de informação que vem se desenvolvendo há mais de 50 anos. Baldini et al. (2017, p. 4, tradução nossa)[2] confirma isso ao afirmar: "[...] a abordagem sem servidor para computação não é completamente nova. Surgiu após recentes avanços e adoção de máquinas virtuais (VM) e, em seguida, tecnologias de contêiner". O conceito da computação em nuvem (*cloud computing*) é a base para a existência da arquitetura sem servidor. Neto (2011) fala sobre a origem deste conceito:

> O conceito fundamental de computação em nuvem não é novo. John McCarthy, em 1961, foi o primeiro a sugerir publicamente (em um discurso dado para celebrar Centenário do MIT) que a tecnologia de compartilhamento de tempo dos computadores poderia levar a um futuro em que poder de computação e até mesmo aplicativos específicos poderiam ser vendidos através do modelo de negócios de serviços públicos (como água ou eletricidade). (NETO, 2011, p. 2, tradução nossa)[3].

McCarthy deu a esse conceito o nome de *Utility Computing*, ou seja, computação como um serviço de utilidade pública, onde o usuário só paga por aquilo que consome ou utiliza. De fato a visão de McCarthy estava correta, no

[2] "[...] the serverless approach to computing is not completely new. It has emerged following recent advancements and adoption of virtual machine (VM) and then container technologies".

[3] "The underlying concept of cloud computing is not a new one. John McCarthy in 1961, was the first to publicly suggest (in a speech given to celebrate MITs centennial) that computer time-sharing technology might lead to a future in which computing power and even specific applications could be sold through the utility business model (like water or electricity)".

entanto, ela iria se concretizar apenas décadas depois, uma vez que as tecnologias disponíveis até então constituíam uma barreira de implementação para a sua ideia. O conceito ficaria adormecido até o ano de 1997, quando Ramnath Chellappa adereçaria o assunto novamente durante uma palestra em Dalas nos Estados Unidos, já se referindo ao assunto através do termo “computação em nuvem”, que iria se popularizar no futuro como termo definitivo para adereçar o assunto. Mell e Grance (2011) definem o conceito de *cloud computing* da seguinte forma:

> Computação em nuvem é um modelo para permitir acesso de rede onipresente, conveniente e sob demanda a uma rede compartilhada de recursos computacionais configuráveis (por exemplo, redes, servidores, armazenamento, aplicativos e serviços) que podem ser rapidamente provisionados e liberados com esforço mínimo de gerenciamento ou interação com o provedor de serviços. (MELL; GRANCE, 2011, p. 7, tradução nossa)[4].

A gigante transnacional Amazon, percebendo o potencial da nova tecnologia, decidiu investir na nova área, e apenas cinco anos após a palestra de Chellappa, anunciou o lançamento de uma série de serviços que alavancaria a computação em nuvem em escala mundial.

> Em 2002, a Amazon Web Services lançou um conjunto de serviços baseados em nuvem, incluindo armazenamento, computação e até mesmo humano inteligência através da *Amazon Mechanical Turk*. Ela continou essa conquista em 2006 com sua Elastic Compute Cloud (E2C), que fornece um serviço comercial através do qual os usuários podem alugar computadores e executar suas próprias aplicações. (KAUFMAN, 2009, p. 61-62, tradução nossa)[5].

A *Amazon Web Services*, doravante AWS, lançada em 2002, foi apenas uma versão inicial. O grande impacto ocorreu com o a disponibilização do serviço EC2 *(Elastic Compute Cloud*) em 2006, fato que faz com que muitos creditem a esse ano o lançamento da AWS. Com este novo serviço, os usuários agora podiam “alugar” máquinas com diferentes configurações em diferentes partes do mundo para atender

---

[4] “Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction”.

[5] “In 2002, Amazon Web Services launched a suite of cloud-based services, including storage, computation, and even human intelligence through the Amazon Mechanical Turk. It followed up this accomplishment in 2006 with its Elastic Compute Cloud (E2C) service, which provides a commercial service through which users can rent computers and run their own applications”.

às suas necessidades. Além disso, outro ponto positivo dos novos serviços disponibilizados pela Amazon era a compatibilidade. Praticamente todas as novas funcionalidades estavam interligadas, o que contribuía para facilitar a vida do usuário, uma vez que ele tinha ao seu dispor todas as ferramentas necessárias para criar sua aplicação em um único ambiente de trabalho, e a maioria delas possuía compatibilidade umas com as outras.

Um fator relevante para a popularização das novas tecnologias da AWS foi o fato de sua utilização não ter preço fixo. O modelo de monetização utilizado para os serviços desse conjunto de serviços consiste em cobrar do usuário apenas por aquilo que ele utilizar, como foi previsto por John McCarthy em 1961. Dessa forma foram supridas as necessidades de inúmeras pessoas ao redor do mundo, como explica Kaufman (2009)

> Agora que a computação em nuvem emergiu como uma plataforma viável e prontamente disponível, muitos usuários com diferentes origens (por exemplo, instituições financeiras, educadores ou cibercriminosos) estão compartilhando máquinas virtuais para executar suas atividades diárias. (KAUFMAN, 2009, p. 62, tradução nossa)[6].

Com o passar de alguns anos, outras empresas começaram a investir na área da computação em nuvem, e assim surgiram o Google Cloud Functions, Microsoft Azure Functions e o IBM OpenWhisk. Como resultado dessa concorrência, a computação em nuvem vem evoluindo cada vez mais e os maiores beneficiados acabam sendo os usuários, pois isso resulta na diminuição do preço dos serviços relacionados à *cloud computing* e impulsiona o desenvolvimento de novas tecnologias.

Em novembro de 2014 a AWS lança um novo serviço: o AWS Lambda, uma plataforma de computação sem servidor baseada em eventos. Ele permite que os usuários criem funções em certas linguagens (como Python e NodeJS) e que essas funções sejam executadas através de gatilhos definidos pelo seu criador. Este serviço também é comumente referido como FaaS (*Function as a Service*). O motivo dessa nomenclatura vem do fato de que cada função, ao ser acionada, é executada

[6] “Now that cloud computing has emerged as a viable and readily available platform, many users from disparate backgrounds (for example, financial institutions, educators, or cybercriminals) are sharing virtual machines to perform their daily activities”.

individualmente, provendo um serviço ao cliente através da sua execução, e depois é encerrada. Podemos definir então que "[...] AWS Lambda é executado em um ambiente computacional isolado dedicado a uma única solicitação, nas outras arquiteturas, cada servidor web executa várias solicitações concorrentes." (VILLAMIZAR et al., p. 182, 2008, tradução nossa)[7].

2.2. FERRAMENTAS UTILIZADAS

Nessa seção iremos falar sobre as ferramentas mais importantes que foram utilizadas para construir a aplicação proposta pelo trabalho, assim como as ferramentas responsáveis por possibilitar a criação da infraestrutura necessária para servir essa aplicação ao usuário. Aqui estão listados recursos cruciais para a criação tanto da arquitetura serverless (como a AWS Lambda), quanto da arquitetura convencional de servidores (como a EC2). É importante notar que a maioria dessas ferramentas se encaixa no termo IaaS (*Infrastructure as a Service*). Neto (2011) explica o conceito por trás desse termo:

> "Serviços em nuvem que fornecem acesso a recursos de computação, como processamento ou armazenamento que pode ser obtido como um serviço. Os exemplos mais populares de IaaS são a Amazon Web Services (AWS) com seu Elastic Compute Cloud (EC2) para processamento, Simple Storage Service (S3) para armazenamento e Rackspace". (NETO, 2011, p. 3, tradução nossa)[8].

Portanto, a infraestrutura é o serviço que nos foi provido pela AWS, sendo a peça principal do desenvolvimento deste trabalho. Sem este serviço seria possível desenvolver a arquitetura cliente-servidor, mas não a arquitetura *serverless*.

---

[7] "[...] AWS Lambda is executed in an isolated computing environment dedicated to a single request, while in the other architectures each web server executes several concurrent requests".

[8] "Cloud services that provide access to computing resources such as processing or storage which can be obtained as a service. The most popular examples of IaaS are Amazon Web Services (AWS) with its Elastic Compute Cloud (EC2) for processing and Simple Storage Service (S3) for storage, and Rackspace".

2.2.1. AWS Lambda

A ferramenta AWS Lambda foi lançada em dezembro de 2014. Baldini et al. (2017, p. 6, tradução nossa)[9] diz que ela “foi a primeira plataforma sem servidor e definiu vários dimensões-chave, incluindo custo, modelo de programação, implantação, limites de recursos, segurança e monitoramento". O Guia do Desenvolvedor da AWS Lambda define o serviço da seguinte forma:

> O AWS Lambda é um serviço de computação que permite que você execute o código sem provisionar ou gerenciar servidores. O AWS Lambda executa seu código somente quando necessário e dimensiona automaticamente, desde algumas solicitações por dia a milhares por segundo. Você paga somente pelo tempo de computação utilizado; não haverá cobrança quando seu código não estiver em execução. Com o AWS Lambda, você pode executar códigos para praticamente qualquer tipo de aplicativo ou serviço de backend, sem necessidade de administração. (AWS, 2019).

No momento, a AWS Lambda aceita a criação de código nas linguagens: Node.js, Java, C#, Go e Python. No entanto é possível ler outras linguagens através do processo de compilação de um arquivo binário da linguagem desejada em um ambiente Amazon Linux compatível com o ambiente onde será executada a função Lambda. Uma vez finalizado o processo de compilação basta fazer o upload do arquivo binário para a função e fazer com que alguma das linguagens citadas anteriormente interprete o arquivo binário e retorne a sua saída. Esse processo é descrito por Bennet (2018) em seu tutorial sobre como hospedar uma aplicação em Laravel (framework da linguagem PHP, que não figura entre as linguagens aceitas por padrão, listadas acima) na AWS Lambda.

Villamizar et al (2016) fez uma comparação de custo infraestrutural do desenvolvimento de uma arquitetura monolítica e uma arquitetura de micro serviços. Para tal tarefa foi utilizada a AWS Lambda e foi concluído que a utilização da arquitetura de micro serviços operados pela Amazon pode ajudar a reduzir em até 13,42% do custo com infraestrutura. Pérez (2018) descreve a arquitetura do framework SCAR (*Serverless Container-aware Architectures*) em conjunto com o serviço Lambda, demonstrando em suas conclusões que a utilização deste

[9] “was the first serverless platform and it defined several key dimensions including cost, programming model, deployment, resource limits, security, and monitoring”.

framework pode fazer com que a Lambda possa se tornar uma plataforma conveniente para computação de alto rendimento. Em seu trabalho, Pérez fala sobre as limitações existentes na AWS Lambda, como por exemplo a compatibilidade natural com um grupo seleto de linguagens de programação e a dificuldade de instalar e utilizar bibliotecas externas.

2.2.2. EC2

A Elastic Compute Cloud é um serviço disponibilizado em 2006 pela Amazon Web Services cuja função é fornecer capacidade computacional segura e escalável. A EC2 dispõe aos seus clientes instâncias com uma grande variedade de configurações e preços, o que torna possível atender às necessidades de muitas pessoas. Ostermann et al (2009) conduziu um estudo que analisou a performance de algumas instâncias da plataforma EC2 utilizando *micro-benchmarks* e *kernels*, chegando à conclusão de que ainda era necessário um aumento de performance nas instâncias, para que elas pudessem ser utilizadas pela comunidade cientifica. No entanto, essa análise foi feita há uma década, e como foi frisado pelos autores, a tecnologia de computação em nuvem está em constante evolução e avançou muito desde então.

Outro ponto positivo desse serviço, é o modelo de monetização, que se assemelha ao da AWS Lambda, onde o usuário paga apenas por aquilo que utilizar. Ou seja, o cliente paga pelas horas de utilização de cada instância, podendo encerrá-las ou transferir o conteúdo da instância para um tipo de instância mais barata a qualquer momento.

Além da conveniência em relação a custo, quando em conjunto com a ferramenta Elastic Load Balancing e CloudWatch (que fazem parte da AWS), a EC2 torna-se conveniente também em relação à escalabilidade. Através da utilização de alarmes definidos no CloudWatch, é possível definir limites de uso de processamento ou transferência de dados que, ao serem ultrapassados, podem fazer com que o Elastic Load Balancing escale instâncias em tempo real de forma automática e balanceie a carga de processamento e tráfego de informação entre

elas. Chaczko et al (2011) explica em seu trabalho a relevância da utilização dessa técnica

> "As técnicas de balanceamento de carga, na área de computação em nuvem, reduzem os custos associados ao documento sistemas de gestão e maximiza a disponibilidade de recursos, reduzindo o tempo de inatividade que afeta empresas durante interrupções". (CHACZKO et al., 2011, p. 134, tradução nossa)[10].

O usuário das instâncias virtuais da Amazon contam com outros recursos interessantes como IPs elasticos que podem ser associados às instâncias, zonas de disponibilidade e as imagens de instâncias, ferramenta muito útil para que funciona tanto como um backup, quanto como uma ferramenta de escalabilidade para que as instâncias possam ser replicadas.

2.2.3. RDS

O Relational Database Service, ou simplesmente RDS, é o serviço da AWS que disponibiliza bancos de dados relacionais na nuvem, tipo de serviço este que também é conhecido através da sigla DbaaS (*database as a service*), que quer dizer "banco de dados como um serviço". Curino et al (2011) fala sobre os benefícios desse tipo de serviço

> O paradigma "banco de dados como serviço" (DBaaS) é atraente por dois motivos. Primeiro, devido às economias de escala, os custos de hardware e energia incorridos pelos usuários provavelmente serão muito menores uma vez que estão pagando pelo compartilhamento de um serviço em vez de executarem tudo por conta própria. Em segundo lugar, os custos incorridos em um DBaaS bem projetado serão proporcionais ao uso real ("pagamento por uso") - isso se aplica tanto ao licenciamento de software, quanto aos custos administrativos. (CURINO et al., 2011, p. 235, tradução nossa)[11].

---

[10] "Load balancing techniques, in the area of cloud computing, reduces costs associated with document management systems and maximizes availability of resources reducing the amount of downtime that affect businesses during outages".

[11] "Such a database-as-a-service (DBaaS) is attractive for two reasons. First, due to economies of scale, the hardware and energy costs incurred by users are likely to be much lower when they are paying for a share of a service rather than running everything themselves. Second, the costs incurred in a well-designed DBaaS will be proportional to actual usage ("pay-per-use") - this applies to both software licensing and administrative costs".

No entanto, o paradigma de “banco de dados como um serviço” apresenta alguns obstáculos a serem superados, como apontam os estudos conduzidos por Hacigumus, Iyer e Mehrotra (2002, p. 1, tradução nossa)[12] que relatam que, entre os desafios primários desse paradigma “estão a sobrecarga adicional de acesso a dados, uma infraestrutura para garantir a privacidade dos dados, e o design de interface de usuário para tal serviço.” Eles buscam em seu trabalho alternativas para solucionar os desafios introduzidos, e frisam a importância de adereçar, em especial, o problema da privacidade de dados, que foi resolvido através da utilização de um algoritmo de criptografia adequado.

Além de possibilitar a contratação de bancos de dados, o RDS facilita a vida do usuário em uma série de aspectos, como por exemplo a configuração, operação, backup e escalabilidade desses bancos. Essa funcionalidade é custeada da mesma forma que os serviços EC2 e Lambda, ou seja, por hora de utilização de cada banco. Assim como a EC2, é possível contratar instâncias com diversas configurações, desde às mais simples e com menor preço por hora, até as mais robustas e mais custosas.

2.2.4. S3

Um dos maiores desafios das últimas décadas, em relação à evolução da tecnologia, tem sido o armazenamento de dados. Com o passar dos anos, os *hardwares* de armazenamento de dados têm evoluído, ganhando cada vez mais capacidade. No entanto, o tamanho dos arquivos também tem aumentado em um nível semelhante, ou até maior, o que torna o armazenamento local de dados uma prática custosa e difícil de gerenciar em muitos casos.

Por conta deste fenômeno, passou-se a buscar uma solução alternativa ao armazenamento local e essa solução surgiu com o desenvolvimento da computação em nuvem, que também estava se tornando a resposta para os problemas de várias outras áreas. “Um aspecto fundamental desta mudança de paradigma é que os

[12] “the additional overhead of remote access to data (service delivery penalty), an infrastructure to guarantee data privacy, and user interface design for such a service”.

dados estão sendo centralizados ou terceirizados na nuvem" (Wang et al., 2010, p. 1, tradução nossa)[13].

Um dos maiores serviços de *cloud storage* (armazenamento na nuvem) fornecido hoje é o *Amazon Simple Storage Service*, ou S3. Este serviço segue o princípio monetário da maior dos serviços da AWS, onde o usuário paga mensalmente por aquilo que utiliza (nesse caso, pela quantidade de armazenamento e tráfego de dados utilizados), e está conectado a vários outros serviços da mesma plataforma. Dessa forma, o S3 se tornou a ideal para muitas pessoas, tendo uma base de usuários que vai desde usuários caseiros até pequenas e grandes empresas. Iamnitchi, Ripeanu e Garfinkel (2008) fizeram um estudo sobre o serviço em questão, que buscou elucidar se sua utilização era viável e rentável para o armazenamento de dados de projetos científicos de larga escala. De acordo com as conclusões alcançadas pelos autores:

> "Acreditamos que a solução para reduzir os custos de armazenamento é entender e responder aos requisitos da aplicação. Mais especificamente, S3 agrega, em uma única faixa de preço, três características de armazenamento do sistema: durabilidade de dados infinita, alta disponibilidade, e acesso rápido aos dados. Muitas aplicações, no entanto, não precisam destas três características agrupadas - portanto, desagregando-as e oferecendo múltiplas classes de serviço direcionadas para necessidades de aplicação pode levar a custos reduzidos e, portanto, menor preços". (IAMNITCHI; RIPEANU; GARFINKEL, 2008, p. 63, tradução nossa)[14].

Um interessante trabalho foi conduzido por Brantner et al (2008), que buscava demonstrar os benefícios e limitações da utilização da ferramenta S3 como um banco de dados de propósito geral. Ao fim do trabalho, especialmente por conta do fator custo, os autores chegaram à conclusão de que "Até então, a computação utilitária não é atraente para processamento de transações de alto desempenho; tais cenários de aplicação são melhor suportados por sistemas de banco de dados

---

[13] "One fundamental aspect of this paradigm shifting is that data is being centralized or outsourced into the Cloud".

[14] "We believe that the solution to reduce storage costs is to understand and respond to application requirements. More specifically, S3 bundles at a single price point three storage system characteristics: infinite data durability, high availability, and fast data access. Many applications, however, do not need all these three characteristics bundled together – thus 'unbundling' by offering multiple classes of service targeted for specific application needs may lead to reduced costs and thus lower utility prices".

convencionais" (BRANTNER et al, 2008, p. 262, tradução nossa)[15]. No entanto os autores explicam que isso é algo que pode mudar no futuro.

2.2.5. Laravel

Para construção da aplicação web deste trabalho, utilizamos o Laravel, framework PHP livre e *open-source* (modelo de desenvolvimento onde todo o código da ferramenta é disponibilizada ao público para livre consulta, examinação e modificação). A versão mais recente do framework na data de desenvolvimento deste trabalho é a 5.8, lançada em 26 de janeiro de 2019, no entanto utilizamos a versão 5.7 por ser a mais recente no início do desenvolvimento do projeto. Em março de 2015 foi considerado como o framework PHP mais popular por uma pesquisa da SitePoint.

O Laravel utiliza o padrão de arquitetura MVC, que separa o projeto em três partes interconectadas: *Model* (Modelo), *View* (Visão) e *Control* (Controle). Esse desacoplamento traz uma série de benefícios para seus utilizadores e é feito ao separar a representação interna da informação das formas que ela é apresentada ao usuário.

> "O MVC demonstrou seus benefícios para aplicações interativas ao permitir múltiplas representações da mesma informação, promovendo a reutilização de código, e ajudando os desenvolvedores a se concentrarem em um único aspecto da aplicação". (SELFA; CARILLO; BOONE, 2006, p. 48, tradução nossa)[16].

Em seu trabalho, Selfa, Carillo e Boone (p.49 , 2006) também discorrem de forma mais aprofundada sobre os benefícios da utilização do padrão MVC , enumerando-os em seis itens: menos acoplamento, maior coesão, as telas fornecem maior agilidade e flexibilidade pois possibilitam uma série de operações como aninhar telas ou fazer com que a mesma tela tenha comportamentos diferentes de

---

[15] "As of today, utility computing is not attractive for high-performance transaction processing; such application scenarios are best supported by conventional database systems".

[16] "MVC has demonstrated its benefits for interactive applications allowing multiple representations of the same information, promoting the code reutilization, and helping developers to concentrate on a single application aspect".

acordo com o dispositivo de visualização, mais clareza de design, manutenção facilitada e maior escalabilidade.

No trabalho de Yu (2014), foi criado um ambiente virtual de operação em Laravel e outro em CI, ou CodeIgniter, um framework para criação de aplicações em PHP mais antigo e tradicional. De acordo com os experimentos de Yu, os resultados alcançados mostraram que "a eficiência de desenvolvimento do método de *web design* baseado no framework Laravel é maior do que comparado ao método tradicional de web design baseado na estrutura CI" (Yu, p. 303, 2014, tradução nossa)[17].

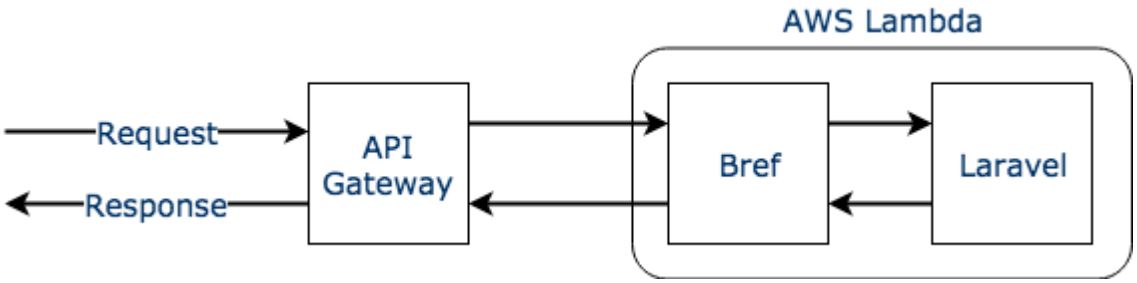
2.2.6. Bref

Como foi dito na seção destinada a AWS Lambda, ela não é capaz de compreender a linguagem PHP. Esse foi um dos principais motivos que levaram Matthieu Napoli, desenvolvedor de aplicações serverless em PHP, a criar o Bref. Essa ferramenta gratuita lançada em 2018, tem o objetivo de facilitar o processo de desenvolvimento de aplicações PHP em plataformas fornecedoras de arquiteturas sem servidor.

Com a sua utilização não é necessário, por exemplo, compilar um arquivo binário PHP para poder fazer com que as funções AWS Lambda consigam entender a linguagem. Além disso, com apenas alguns comandos de prompt é possível fazer o processo de empacotamento da aplicação e criação automática das funções praticamente configuradas por completo. Por outro lado, a ferramenta possui alguns pontos negativos, como por exemplo a integração com uma quantidade limitada de extensões, pouca documentação e comunidade pequena de utilizadores. No entanto, os desenvolvedores estão diariamente trabalhando para evoluir seu produto, da mesma forma que a tecnologia *serverless* vem fazendo avanços a cada dia.

[17] "development efficiency of Web design method based on Laravel framework is higher compared to traditional Web design method based on CI framework".

Figura 1 - Diagrama de Comunicação Serverless



Fonte: Blog de Matthieu Napoli[18]

Não foi possível encontrar nenhuma referência de trabalhos científicos na área que fazem menção a ferramenta Bref por conta do seu recente lançamento e da baixa popularidade da arquitetura sem servidor.

3. METODOLOGIA

O desenvolvimento deste trabalho foi iniciado em meados do mês de setembro de 2018, no entanto, os conhecimentos necessários para a sua criação vêm sendo adquiridos ao longo dos últimos dois anos, datando do início de 2017. O nosso objetivo foi verificar a viabilidade da utilização da arquitetura sem servidor ao servir aplicações HTTP em comparação com um servidor convencional. Para chegar à uma conclusão sobre essa questão, buscamos evidenciar os resultados da avaliação de execução de uma mesma aplicação *web* em duas arquiteturas diferentes: a arquitetura cliente-servidor, mais amplamente utilizada, que exige um *hardware* (geralmente uma máquina dedicada à hospedagem de aplicações) como servidor; e a arquitetura sem servidor, onde esse *hardware* é abstraído, dispensando a necessidade de administração de *hardware* por parte do gerente da aplicação. A forma de estudo baseada na comparação entre dois elementos é bastante comum na área da computação. Para exemplificar tal fato, podemos levar em conta os

[18] Disponível em: <https://mnapoli.fr/serverless-laravel>. Acesso em: 01 jan. 2019.

estudos de Bolzan e Giraffa (2002), assim como o de Maron, Griebler e Schepke (2014) e de Villamizar et al (2016), este último, previamente citado neste trabalho. O ponto em comum entre essas obras é a natureza comparativa de seus estudos, o que as tornou leituras relevantes para a fundamentação da nossa metodologia de pesquisa.

De acordo com o conceito de pesquisa descritiva definido por Raupp e Beuren (2009):

> "[...] a pesquisa descritiva configura-se como um estudo intermediário entre a pesquisa exploratória e a explicativa, ou seja, não é tão preliminar como a primeira nem tão aprofundada como a segunda. Nesse contexto, descrever significa identificar, relatar, comparar, entre outros aspectos". (RAUPP; BEUREN; 2009, p. 81).

Levando em conta esse conceito, chegamos à conclusão de que o nosso trabalho não se enquadra na categoria de pesquisa exploratória, uma vez que o tema principal da pesquisa (a tecnologia sem servidor), embora recente, não é tão inovador, já que vem sendo estudado há cerca de duas décadas. Os objetivos deste trabalho se encaixam melhor na definição de Raupp e Beuren (2009) de pesquisa descritiva, quando defendem que descrever significa identificar, relatar ou comparar, uma vez que a natureza desse estudo é investigativa e comparativa. De acordo com Gil (2007, apud GERHARDT e SILVEIRA, 2009, p. 35), a pesquisa explicativa "preocupa-se em identificar os fatores que determinam ou que contribuem para a ocorrência dos fenômenos", ou seja, busca esclarecer o motivo de algo através dos resultados oferecidos. Esta definição não se encaixa em nosso estudo, contribuindo para concluir que este trabalho é melhor categorizado como uma pesquisa descritiva.

Nos próximos tópicos do capítulo 3, iremos descrever de forma mais aprofundada os seguintes assuntos: as ferramentas e etapas utilizadas que possibilitaram chegar aos resultados finais dessa pesquisa, os fatores de observação que nortearam a análise dos dados coletados, os métodos escolhidos para obtenção dos dados em cada um dos fatores de observação definidos e, por fim, uma listagem de todos os passos percorridos, em ordem de execução, durante o trabalho.

3.1. INSTRUMENTOS DE PESQUISA

A implementação de todos os elementos utilizados neste trabalho, assim como a avaliação de resultados, aconteceu através do meio virtual. Os instrumentos utilizados em nossa pesquisa consistem no levantamento bibliográfico que foi feito para embasar e enriquecer o conteúdo deste trabalho, o desenvolvimento de uma aplicação web construída através do framework Laravel, a criação e configuração de uma instância virtual e outros recursos necessários para hospedar e servir aplicações HTTP, a criação e configuração da infraestrutura responsável por disponibilizar uma arquitetura sem servidor capaz de servir aplicações HTTP e, por fim, um conjunto de ferramentas e painéis das quais coletamos informações que foram usadas para fazer uma análise comparativa das características de ambas arquiteturas observadas.

Para que a análise das arquiteturas seja feita de forma coordenada e imparcial, foi necessário que os quesitos de avaliação fossem definidos previamente à execução dos testes. Na próxima seção iremos elencar e descrever os quesitos de avaliação definidos neste trabalho.

3.2. FATORES OBSERVADOS NA PESQUISA

Em seu trabalho, Ostermann et al (2009) faz uma análise comparativa entre os serviços de computação em nuvem da AWS EC2. Para fazer essa análise ele define o seguinte método de avaliação:

> "Nós projetamos um método de avaliação de desempenho, que permite uma avaliação das tecnologias em nuvem. Para este fim, dividimos o processo de avaliação em duas partes, a primeira específica da nuvem, a segunda como um diagnóstico da infraestrutura". (OSTERMANN, 2009, p. 118, tradução nossa)[19].

[19] "We design a performance evaluation method, that allows an assessment of clouds. To this end, we divide the evaluation procedure into two parts, the first cloudspecific, the second infrastructure-agnostic".

Foster et al (2008) fazem um paralelo entre computação em nuvem e computação em grid. De forma análoga, na seção “Comparando grids e nuvem lado a lado”, eles definem os campos de observação do seu trabalho

> “Esta seção tem como objetivo comparar Grades e Nuvens em uma ampla variedade de perspectivas, desde arquitetura, modelo de segurança, modelo de negócio, modelo de programação, virtualização, modelo de dados, modelo de computação, para proveniência e aplicações”. (Foster et al., 2008, p. 2, tradução nossa)[20].

Ostermann, Foster et al e uma série de outros autores deixam clara a importância de definir previamente os fatores que se deseja observar no estudo. Tendo isso em mente e levando em conta que esse é um trabalho de cunho comparativo, definimos os fatores que irão norteá-lo.

Durante o andamento da pesquisa, foram analisados uma série de fatores, que dividimos em duas categorias: fatores primários e fatores secundários. Os fatores primários consistem em pontos de maior relevância para definir a viabilidade de hospedagem de uma aplicação web, enquanto os secundários, embora relevantes, são considerados menos cruciais como quesito avaliativo.

Os fatores primários avaliados são: performance e custo. O fator performance está relacionado ao tempo de resposta da aplicação a partir do recebimento de uma requisição. A análise do custo diz respeito ao valor mensal gasto para possibilitar a criação e manutenção de cada uma das arquiteturas estudadas.

Os fatores secundários são: compatibilidade com recursos externos e gerenciamento. A compatibilidade com recursos externos diz respeito a quais tipos de bibliotecas e extensões podem ser utilizadas pela aplicação de acordo com cada arquitetura. Por fim, a análise do gerenciamento irá mostrar as dificuldades e facilidades de gerenciamento que o administrador de cada uma das arquiteturas pode enfrentar.

---

[20] “This section aims to compare Grids and Clouds across a wide variety of perspectives, from architecture, security model, business model, programming model, virtualization, data model, compute model, to provenance and applications”.

3.3. MÉTODOS DE COLETA DE DADOS

Para a coleta dos dados que embasaram os resultados finais alcançados por esse trabalho foram utilizadas diversas fontes de informação. A ferramenta utilizada para extrair informações relativas ao quesito performance, fator primário de estudo, foi a BlazeMeter, que funciona como PaaS (*Platform as a Service*). Ela nos permitiu fazer uma série de testes de carga em nossa aplicação através de requisições programadas, afim de verificar o tempo de resposta de cada requisição e como ela se sai em situações de estresse (Ex: muitas requisições em curto período de tempo). Durante a fase de teste desse trabalho, foi utilizada tanto a extensão de navegador do BlazeMeter, quanto a plataforma de testes existente no *website* da aplicação.

O outro fator primário de observação da nossa pesquisa foi o custo relacionado ao desenvolvimento e manutenção das arquiteturas. Para fazer essa análise, observamos os dados disponibilizados pelo painel de faturamento da AWS, que detalha a origem dos gastos mensais de toda a infraestrutura utilizada em nossa conta. É importante destacar que o valor destinado à contratação do domínio não foi levado em conta em nossas avaliações, afinal esse valor não diz respeito às arquiteturas.

O processo de coleta de informação acerca dos fatores secundários ocorreu por meio de estudo e observação, ao decorrer do trabalho. Ao longo do desenvolvimento deste, recolhemos conhecimento que nos ajudou a chegar a conclusões acerca dos dois fatores (compatibilidade com recursos externos e gerenciamento), a fim de chegar nas conclusões que foram discutidas ao final desse estudo.

3.4. OS PASSOS PERCORRIDOS

• Na PRIMEIRA etapa desse trabalho, foi desenvolvida uma aplicação PHP para uma loja fictícia de artigos de informática chamada Power Tech. Para a criação desta aplicação utilizamos o framework Laravel, pelo emprego do padrão de arquitetura de software MVC, que proporciona maior organização

ao projeto, por permitir a integração da aplicação com extensões criadas pela comunidade do framework e uma série de outras vantagens. A aplicação conta com uma vitrine de produtos, onde usuário pode ver informações sobre os produtos disponíveis na nossa loja fictícia, além de poder pesquisar por produtos através do nome ou filtra-los por categorias. O usuário pode também se cadastrar no *newsletter* (boletim informativo) do site para receber e-mails da empresa com informações sobre promoções, produtos novos e coisas do tipo. Além disso, existe um painel administrativo que pode ser acessado pelo administrador para cadastrar novas categorias, novos produtos, gerenciar os usuários cadastrados no *newsletter* e disparar e-mails através dela;

• A SEGUNDA etapa foi dedicada à geração e configuração de uma instância EC2 do tipo t2.micro com o sistema operacional Ubuntu 18.04 LTS. O propósito principal dessa instância foi hospedar a aplicação criada na primeira etapa, no entanto ela também foi utilizada para concluir as configurações realizadas na quarta etapa. Para fazer o upload dos arquivos da aplicação para a instância foi utilizado o VSFTPD. Para servir a aplicação foi escolhido o Nginx, pois ele demonstra um melhor desempenho ao receber um alto volume de requisições quando lidando com pacotes pequenos, o que condiz perfeitamente com a realidade aplicação.

• Na TERCEIRA fase do trabalho, configuramos a infraestrutura para criação da arquitetura sem servidor através da plataforma de serviço AWS. Essa configuração contou com a utilização de várias ferramentas disponíveis na plataforma da Amazon como: VPC, API Gateway, EC2, Cloudwatch, S3, IAM e RDS. É importante apontar que a função Lambda não foi criada nessa etapa.

• Na QUARTA etapa, a aplicação criada na primeira etapa foi lançada na arquitetura *serverless*. Para realizar essa tarefa nos conectamos via SSH à instância criada da segunda fase e então, através dos comandos disponíveis na ferramenta Bref, foi gerada a função Lambda responsável por executar a aplicação. Nesta fase também foram feitas as configurações finais da infraestrutura e os apontamentos necessários no site onde o domínio utilizado foi cadastrado, possibilitando que a aplicação fosse acessada através do

domínio personalizado em vez do *endpoint* padrão definido pela função Lambda.

• A QUINTA e última etapa do trabalho dedicou-se execução de testes, coleta de dados relacionados a cada uma das arquiteturas e análise desses dados. O processo de coleta e análise dos dados foi feito tendo como base os fatores definidos na seção 3.2, para poder extrair as conclusões do trabalho.

4. IMPLEMENTAÇÃO DA APLICAÇÃO WEB

Este capítulo é destinado a demonstrar, de forma detalhada, o processo de construção da aplicação que foi utilizada como base para nosso estudo. A análise comparativa de qualidade de ambas as arquiteturas avaliadas, feita durante o capítulo 6, tem como objeto de estudo essa aplicação. Aqui serão relatadas todas as extensões utilizadas pela aplicação assim como todas as funcionalidades nela implementadas. Para cuidar da hospedagem do código e versionamento da aplicação utilizamos a versão gratuita da plataforma Bitbucket, serviço semelhante ao disponibilizado pelo Github.

4.1. FUNCIONALIDADES

Para definir quais as funcionalidades que deveriam constar em nosso *software*, levamos em conta o propósito do trabalho como um todo. Esse propósito é examinar a viabilidade da arquitetura sem servidor, no entanto, o termo **viabilidade** pode ser bastante relativo. Enquanto uma determinada faixa de custo pode ser viável para algumas pessoas, talvez não seja para outras. Esse é o caso de pessoas que pretendem ter um *site* na internet mas não estão dispostos a investir uma grande quantia para alcançar esse objetivo. O mesmo vale para outros fatores como performance e segurança.

Tendo isso em mente, buscamos desenvolver uma aplicação que fosse relativamente simples, mas não a ponto de ser estática (aplicação que só permite

mudanças relevantes em seu conteúdo através de alterações no código fonte). Um site com funcionalidades que permitem a interação tanto por parte do cliente quanto por parte do administrador do sistema.

Com a expansão do acesso à internet e o aumento da confiabilidade de seus usuários em relação à ferramenta, um dos tipos de aplicação que vem aparecendo cada vez mais é o *e-commerce*, ou varejo eletrônico. Srinivasan, Anderson e Ponnavolu (2002), falam sobre esse fenômeno:

> "Este rápido crescimento do varejo eletrônico reflete as vantagens que ele oferece sobre as lojas convencionais de tijolo e argamassa, incluindo maior flexibilidade, maior alcance do mercado, menor custo estrutural, transações mais rápidas, linhas de produtos mais abrangentes, maior conveniência e personalização". (SRINIVASAN; ANDERSON; PONNAVOLU, 2002, p. 41, tradução nossa)[21].

As vantagens da utilização desse artifício no mundo dos negócios também são apontadas por Schafer, Konstan, Riedl (2001):

> "Enquanto o comércio eletrônico não permitiu necessariamente que as empresas produzissem mais produtos, ele permitiu que elas fornecessem mais opções aos consumidores. Em vez de dezenas de milhares de livros em uma superloja, os consumidores podem escolher entre milhões de livros em uma loja online". (SCHAFER, KONSTAN, RIEDL, 2001, p. 116)[22].

Tendo em vista o crescimento do *e-commerce*, a ideia da temática por trás da aplicação foi criar o *website* de uma empresa fictícia especializada na venda de artigos de informática que intitulamos de *PowerTech*. No entanto, a implementação de um sistema de *e-commerce* completamente funcional, com carrinho de compras e formas de pagamento, seria demasiadamente trabalhoso e não contribuiria para alcançar o objetivo do trabalho. Então optamos por desenvolver o *site* tendo como principal funcionalidade uma vitrine de produtos.

---

[21] "This rapid growth of e-retailing reflects the compelling advantages that it offers over conventional brick-and-mortar stores, including greater flexibility, enhanced market outreach, lower cost structures, faster transactions, broader product lines, greater convenience, and customization".

[22] "While E-commerce hasn't necessarily allowed businesses to produce more products, it has allowed them to provide consumers with more choices. Instead of tens of thousands of books in a superstore, consumers may choose among millions of books in an online store".

4.1.2. Vitrine de produtos

Tendo em vista que iríamos criar um *website* de uma empresa voltada à comercialização de produtos, a primeira funcionalidade que desenvolvemos e integramos ao projeto foi a vitrine de produtos. Trata-se de uma ferramenta que permite aos usuários do *site* visualizar os produtos disponíveis para a empresa, mas não possibilita a compra desses produtos, funcionando apenas de maneira informativa. Tendo em vista que essa funcionalidade seria a mais importante da aplicação, dada a natureza de negócio da *PowerTech*, fizemos dela a *homepage* do site e criamos três formas com quais o cliente pudesse navegar e buscar por produtos.

A primeira foi a **paginação**, técnica utilizada para separar os produtos cadastrados em diferentes seções numeradas, onde é possível navegar entre as páginas através de um menu com seus respectivos números. Uma vez que a quantidade de produtos cadastrados não influenciaria nos testes que seriam feitos, nós definimos como seis a quantidade de produtos que seriam mostrados por página. A maior vantagem em utilizar esse recurso é diminuir a quantidade de informação a ser carregada em cada página, afinal, se todos os produtos fossem listados em apenas uma página, o tempo de carregamento dela poderia aumentar drasticamente, uma vez que as imagens e descrições de cada produto seriam carregadas de uma só vez. É importante destacar que a paginação funciona em conjunto com as outras duas formas de navegação que serão apresentadas a seguir.

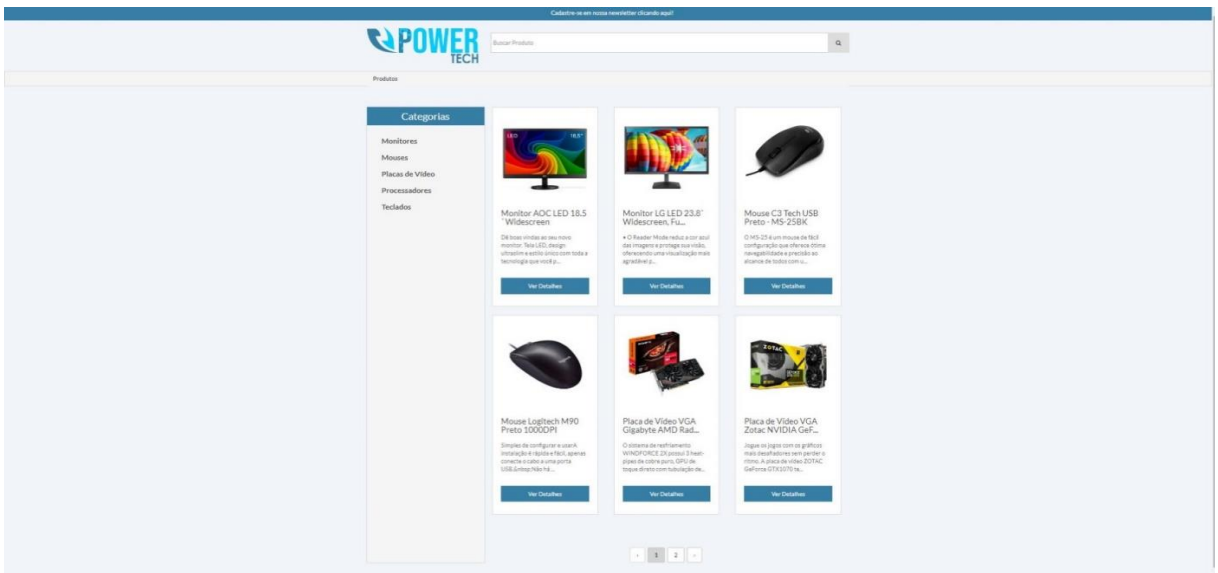
A segunda forma de navegação pela vitrine foi a **barra de pesquisa** localizada no topo da página, ferramenta existente na maioria dos *websites*, que permite ao usuário digitar o nome do produto que procura. A busca é feita no banco de dados não só pelos produtos com nome igual ao que foi digitado, mas por todos os produtos com nome semelhante. Dessa forma a pesquisa torna-se mais ampla e tem mais chances de trazer ao usuário o que ele procura.

Por fim, a terceira e última forma criada foi a **pesquisa por categorias**. Desenvolvemos um sistema de cadastro de categorias de produtos, de forma que cada produto cadastrado deve pertencer a uma categoria, associando-o a ela.

Entraremos em mais detalhes sobre como é feito esse cadastro na seção 4.1.4, cujo tema é o painel administrativo. Em nossa vitrine, além da listagem de produtos, existe uma barra de categorias localizada no canto esquerdo da tela, que exibe todas as categorias cadastradas no sistema, onde cada uma delas funciona como *link* de acesso a uma busca específica aos produtos daquela categoria (como mostra a Figura 1). Então, ao clicar em uma categoria, o usuário será servido com uma página que mostra apenas produtos associados àquela categoria.

Um fator importante que contribuiu para chegar ao visual escolhido para a nossa página inicial foi a utilização da técnica de desconstrução de interface, descrita no trabalho de Cerutti (2010). Nesta técnica, Cerutti propõe que seja analisada a interface de uma aplicação que tenha propósito similar à que se pretende desenvolver. De acordo com ele, “o objetivo geral da desconstrução de interface é usar estas diretivas como lista de verificação durante o projeto da página inicial e avaliar os impactos na usabilidade do produto ao se utilizar diferentes abordagens no design” (Cerruti, 2010, p. 2). A página escolhida para que fosse feita a desconstrução foi da loja virtual Kabum[23], também especializada na venda de produtos eletrônicos (Figura 2).

Figura 1 - Página Inicial da Aplicação



Fonte: Pablo Bittencourt Pereira Gobira (2019)

[23] Disponível em <https://www.kabum.com.br>. Acesso em: 4 mai. 2019

Figura 2 - Página inicial da Kabum

Fonte: https://www.kabum.com.br/

Excluindo o fato de que o site da Kabum possui muito mais botões, como os de carrinho, subcategorias e publicidade (elementos que não existem no site da *PowerTech*), podemos ver que existe uma grande semelhança no design das duas páginas ao fazermos uma comparação entre o *layout* de ambas. A principal diferença que pode ser notada é o posicionamento da barra de categorias, que no site da *PowerTech* fica disposta do lado esquerdo da tela como menu vertical, enquanto no site da Kabum aparece horizontalmente no topo da página. O site da Kabum ainda apresenta uma barra lateral esquerda, que exibe as subcategorias. Em nossa aplicação, como não existem subcategorias, tivemos que escolher entre posicionar as categorias ao lado esquerdo ou no topo da página. Por fim, escolhendo a primeira opção por acreditar que esse posicionamento oferece maior visibilidade às categorias durante a navegação.

Outra diferença entre *layouts* é a presença do cadastro de newsletter na *homepage* do site da Kabum, enquanto em nossa aplicação é necessário acessar um link, encontrado no topo da página, para chegar ao formulário de cadastro. Essa separação foi feita para reduzir ao máximo a quantidade de informação apresentada durante a navegação na vitrine e assim diminuir a poluição visual.

Em seu livro, cujo tema principal é a utilização e a importância da IHC, Rocha e Baranauskas (2001, p. 37) afirmam que “no design para a Web existem basicamente duas abordagens: uma artística onde o designer se expressa e outra dirigida a resolver o problema do usuário”. No nosso trabalho, seguimos pelo segundo caminho, dando preferência à um layout simples que beneficia a usabilidade do usuário.

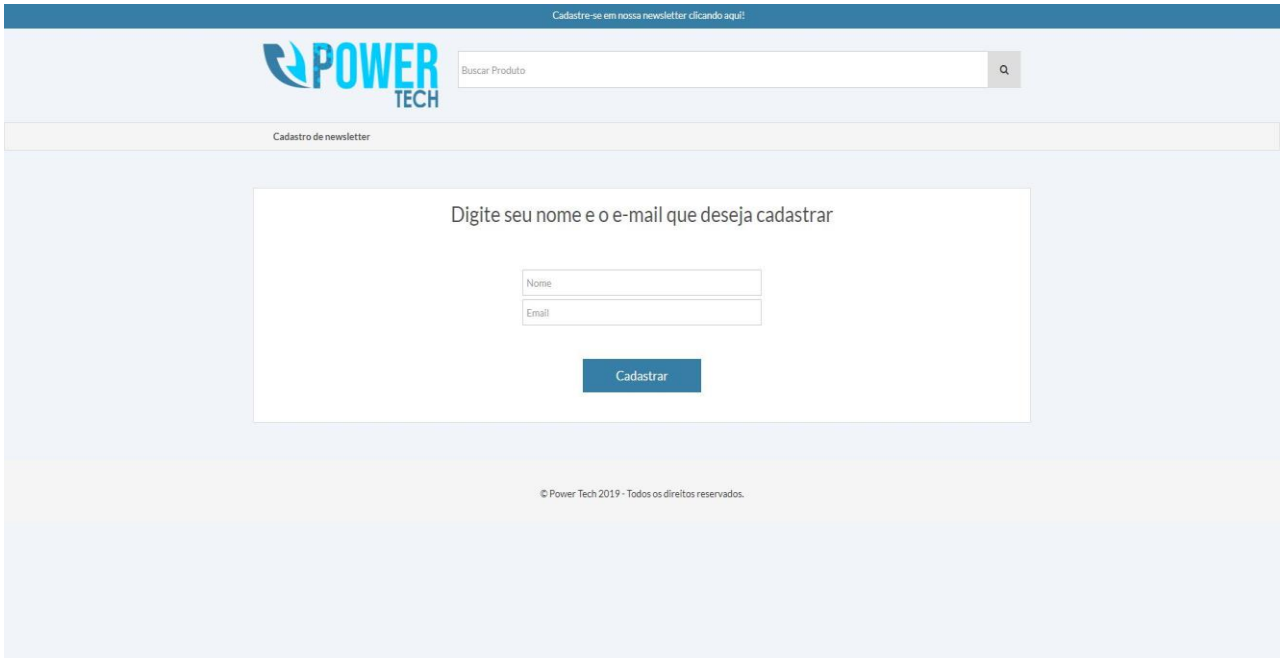
Todas as três formas de navegação podem ser visualizadas na Figura 1, onde é possível ver a logomarca da *PowerTech* (ao lado esquerdo da barra de pesquisa), além da barra azul no topo da página com o *link* escrito “Cadastre-se em nosso *newsletter* clicando aqui!”. Esse link é responsável por direcionar o usuário a uma página onde ele pode interagir com a segunda ferramenta da aplicação, o cadastro de *newsletter*.

4.1.3. Newsletter

De acordo com Petterson, Balasubramanian e Bronnenberg (1997, p. 332) “[...] não é surpresa que a maioria das empresas que atualmente buscar por ‘presença’ na internet tendem a estar preocupadas com as comunicações e potencial de propaganda da internet”. Com o passar dos anos, essa frase tem se provado cada vez mais correta, uma vez que o público da internet só tem aumentado e o marketing através da rede se torna cada vez mais eficiente. A *PowerTech* não poderia deixar de se beneficiar desse tipo de marketing, por tanto implementamos em seu *website* a ferramenta de boletim informativo, mais conhecida como *newsletter*.

Newsletter é um método de distribuição de informação via *e-mail*. Através dele, o usuário se cadastra por meio de um formulário em um site ou página de internet da sua escolha. Uma vez cadastrado, esse usuário torna-se um assinante e, dessa forma, se declara interessado em receber e-mails enviados por aquela página ou site. O formulário de cadastro que foi desenvolvido para a aplicação consiste em apenas dois campos: o nome do usuário e seu *e-mail*.

Figura 3 - Formulário de cadastro no newsletter



Fonte: Pablo Bittencourt Pereira Gobira (2019)

Após se cadastrar no formulário ilustrado na Figura 3, é necessário que o usuário acesse o link de confirmação, que é enviado automaticamente para seu *e-mail*, a fim de efetivar o seu cadastro. Incluímos esse passo adicional como medida de segurança para que uma pessoa mal intencionada não consiga cadastrar o *e-mail* de terceiros no *newsletter*, sem permissão. Ao acessar o link, é mostrada ao usuário uma página cuja mensagem informa que o cadastro foi efetuado com sucesso. Caso seja feita a tentativa de cadastrar um *e-mail* que já esteja inscrito no sistema, o usuário receberá uma mensagem de erro informando que aquele endereço de correio eletrônico já está cadastrado e não lhe será enviado o *e-mail* de confirmação.

Ao concluir as ações descritas acima, pode-se dizer que o usuário já está cadastrado no *newsletter* do *site*. Então, o administrador da página, ao acessar o painel administrativo da mesma, tem o poder de enviar *e-mails* em massa para todas as pessoas que estiverem cadastradas no sistema. Falaremos mais sobre as ferramentas do administrador no próximo tópico.

A maior utilidade do *newsletter* é permitir que o gestor do sistema dissemine informações relativas a sua página ou empreendimento em larga escala, ou seja,

para uma grande quantidade de pessoas. Dessa forma, o *newsletter* funciona como uma ferramenta de *marketing* virtual eficiente e gratuita. No entanto, a motivação de implementar essa ferramenta em nossa aplicação não foi apenas oferecer comodidade ao administrador e aos usuários. Sabíamos que a implantação de um sistema automatizado para o envio de e-mails era prática comum em servidores HTTP convencionais e que para colocá-la em execução, não seria necessário nenhum custo adicional. Porém, não sabíamos se iríamos ter a mesma facilidade executando o envio de *e-mails* através da arquitetura *serverless*. Essa dúvida foi o maior motivo pelo qual incluímos a funcionalidade na aplicação. Os resultados dessa investigação de fato se mostraram bastante relevantes para esse estudo e estão explicitados no capítulo 6.

No tópico a seguir adereçamos a última funcionalidade que foi integrada ao sistema.

4.1.4. Painel administrativo

A última e mais robusta ferramenta da aplicação é o painel administrativo, idealizado para que o administrador possa gerenciar a lista de cadastro do *newsletter* e o conteúdo exibido na vitrine do site.

Para desenvolver o estilo do painel recorremos à mesma técnica utilizada para desenvolver a vitrine de produtos: a desconstrução de interface estudada por Cerutti (2010). Fazer uso dessa técnica nos levou buscar por telas de painéis administrativos utilizados em outras aplicações, o que nos fez encontrar uma série de *templates* de painel administrativo pela internet. Um desses templates foi o AdminLTE 2[24], o qual escolhemos como base para construir o nosso próprio painel.

No site do AdminLTE existe um painel administrativo feito como exemplo para apresentar as funcionalidades do *template*. Ao aplicarmos a desconstrução nas telas desse painel foi possível observar três coisas. A primeira observação foi feita sobre a barra lateral. Percebemos que construí-la comportando todos os menus e

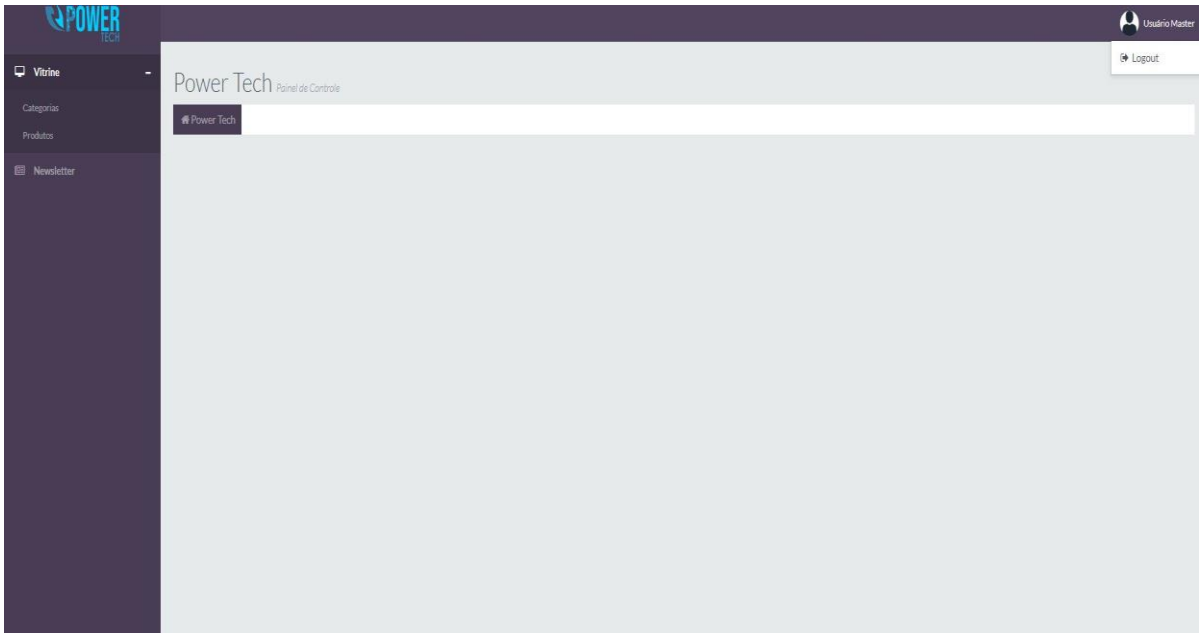
[24] https://adminlte.io/

submenus de funcionalidades do sistema era uma forma elegante e organizada de administrar o conteúdo do painel. Não é à toa que vimos essa mesma forma de dispor as funcionalidades em vários outros templates, por tanto, foi a primeira característica que incorporamos ao painel. O segundo detalhe que observamos foi a presença dos ícones ao lado de cada funcionalidade na barra lateral. Estes ícones permitem achar com muito mais rapidez a ferramenta que se está procurando, logo também decidimos integrar essa característica a nossa tela. O terceiro ponto observado foi o posicionamento do botão que leva à opção de deslogar do sistema. Ele é posicionado no canto superior direito da tela, lugar bastante intuitivo para muitos usuários, uma vez que aplicações amplamente utilizadas, como serviços de e-mail, costumam deixar a opção de *deslogar* posicionada no mesmo lugar. Esse detalhe torna o processo de familiarização com a tela muito mais fácil para os usuários que estão acostumados com esse tipo de organização de tela, portanto seguimos o exemplo e colocamos a opção de *deslogar* no mesmo lugar em nosso painel.

Para acessar o painel administrativo do site é necessário que o administrador adicione “/admin” ao final da *url* base do site, resultando na url “www.powertechserverless.com.br/admin”. Essa forma de acesso foi escolhida como uma medida de segurança, para que a existência do painel seja desconhecida pelos clientes do *site*. Ao acessar o endereço, o usuário será servido com uma tela de *login*, onde deverá digitar suas credenciais (e-mail e senha) e, em seguida, clicar no botão **Entrar**. O sistema não permite ao administrador alterar suas credenciais nem criar outras contas de administrador, portanto as únicas maneiras de criar administradores ou mudar suas credenciais são; alterar o código dos *seeds* (arquivos responsáveis pelo povoamento do banco de dados no Laravel) e repovoar o banco de dados ou inserir o novo administrador na tabela “users”, diretamente no banco de dados.

Ao informar as credenciais corretas na tela de *login*, o administrador irá se deparar com a tela principal do painel administrativo (Figura 4):

Figura 4 - Painel administrativo



Fonte: Pablo Bittencourt Pereira Gobira (2019)

No canto superior direito existe um botão que mostra o nome do administrador que está acessando o painel. Ao clicar nele, um botão com o título **Logout** é revelado. Através dele o administrador pode se desconectar da sua conta, saindo do painel e sendo automaticamente redirecionado para a página inicial da aplicação.

Na parte lateral esquerda da tela está disposta a barra lateral que mostra um botão com o logo da empresa e os módulos do sistema com os quais o administrador pode interagir. Doravante, iremos nos referir a essa barra lateral como “menu de módulos”. O botão com o logo da empresa redireciona o usuário para a mesma tela da Figura 4, ou seja, a página inicial do painel administrativo. Ao clicar no botão com o título **Vitrine**, são revelados dois outros botões: **Categorias** e **Produtos**.

Ao clicar na opção **Categorias** no menu de módulos, o administrador é direcionado à tela de listagem das categorias que são associadas aos produtos. Para cada uma das categorias cadastradas existe um botão de edição, através do qual é possível editar os dados da mesma, e um botão de deletar aquela categoria. Quando o utilizador do sistema clica no botão para deletar uma categoria, ele recebe

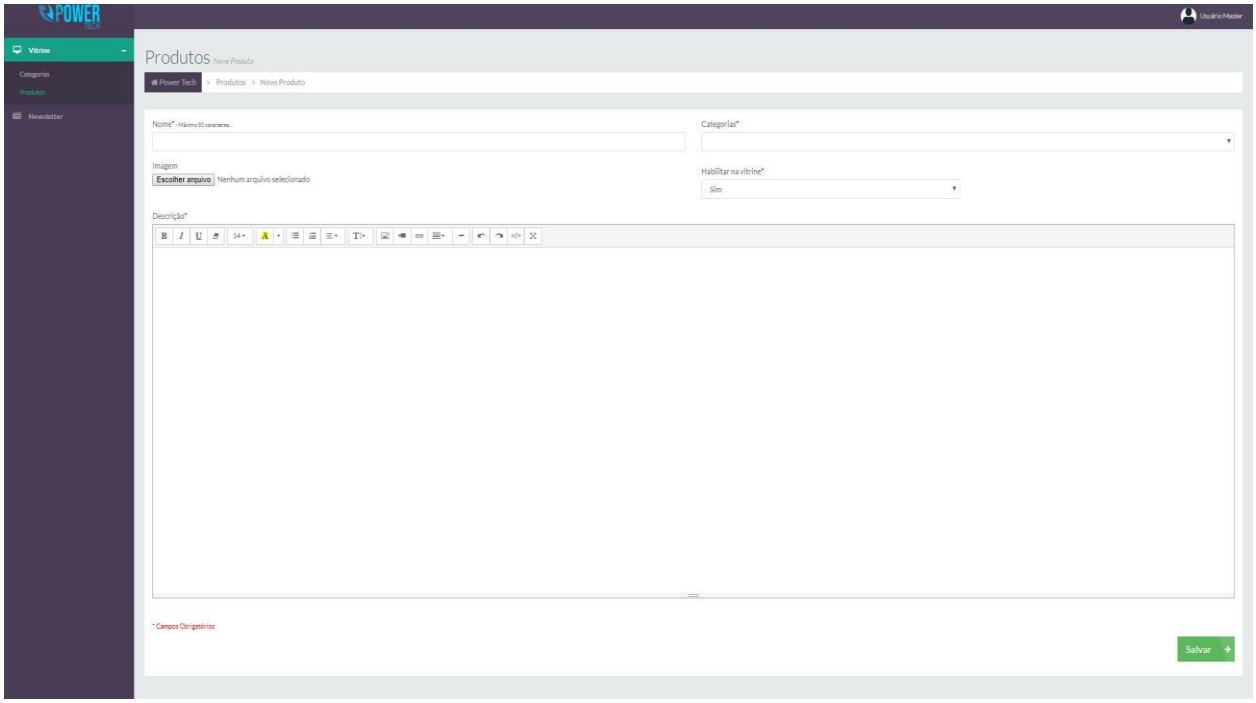
uma mensagem de confirmação. Caso ele confirme que quer excluí-la, todos os produtos associados a ela também serão automaticamente excluídos também. Seria possível dar um tratamento menos radical aos produtos pertencentes às categorias deletadas, no entanto esse detalhe não iria alterar o resultado dos estudos desse trabalho, então escolhemos não o fazer, para investir o tempo em questões de maior importância dentro do projeto da aplicação. É possível também buscar uma categoria através da barra de pesquisa que se encontra na parte superior da tela, digitando o seu nome e apertando a tecla ***Enter***.

Na parte superior direita da tela existe o botão **Nova Categoria**, que ao ser clicado tem como destino a tela que contém o formulário de cadastro de nova categoria. O campo **nome** é o único campo desse formulário, então após preenche-lo e clicar em salvar o administrador terá criado uma nova categoria e será redirecionado para a tela de listagem.

Ao acessar a opção **Produtos** no menu de módulos, o usuário chega a uma página bem similar à de listagem de categorias, mas que possui algumas opções de busca adicionais. Essa página exibe a listagem dos produtos cadastrados no sistema. Suas opções de busca consistem nos filtros de categoria (permite pesquisar por produtos de uma categoria específica) e de habilitação do produto (permite buscar por produtos que estão habilitados ou não na vitrine). Assim como na listagem de categorias, é possível pesquisar um produto pelo seu nome, busca que funciona em conjunto com os dois filtros citados anteriormente, dando mais opções de seleção ao usuário.

Ao lado superior direito da tela de listagem está o botão **Novo Produto**, que ao ser clicado leva até a tela de cadastro de produto. Nessa tela o administrador do sistema entra com os seguintes dados para o cadastro dos produtos: nome, categoria do produto, imagem, opção de se o produto estará habilitado ou não na vitrine e a descrição do produto.

Figura 5 - Formulário de cadastro de produto



Fonte: Pablo Bittencourt Pereira Gobira (2019)

Os campos marcados com o asterisco vermelho ao lado do nome são de preenchimento obrigatório, portanto apenas o cadastro da imagem do produto é dispensável. Caso nenhuma imagem seja cadastrada, o produto será exibido na vitrine com uma imagem padrão, que denota que não foi cadastrada nenhuma imagem para aquele produto. No campo Categoria serão listadas todas as categorias cadastradas no sistema, onde é obrigatório escolher uma delas para finalizar o cadastro do produto. No campo **Habilitar na vitrine**, o administrador escolhe se quer que o produto apareça ou não na listagem do *front-end*. Dessa forma, não é obrigatório excluir um produto caso ele precise ser retirado da vitrine.

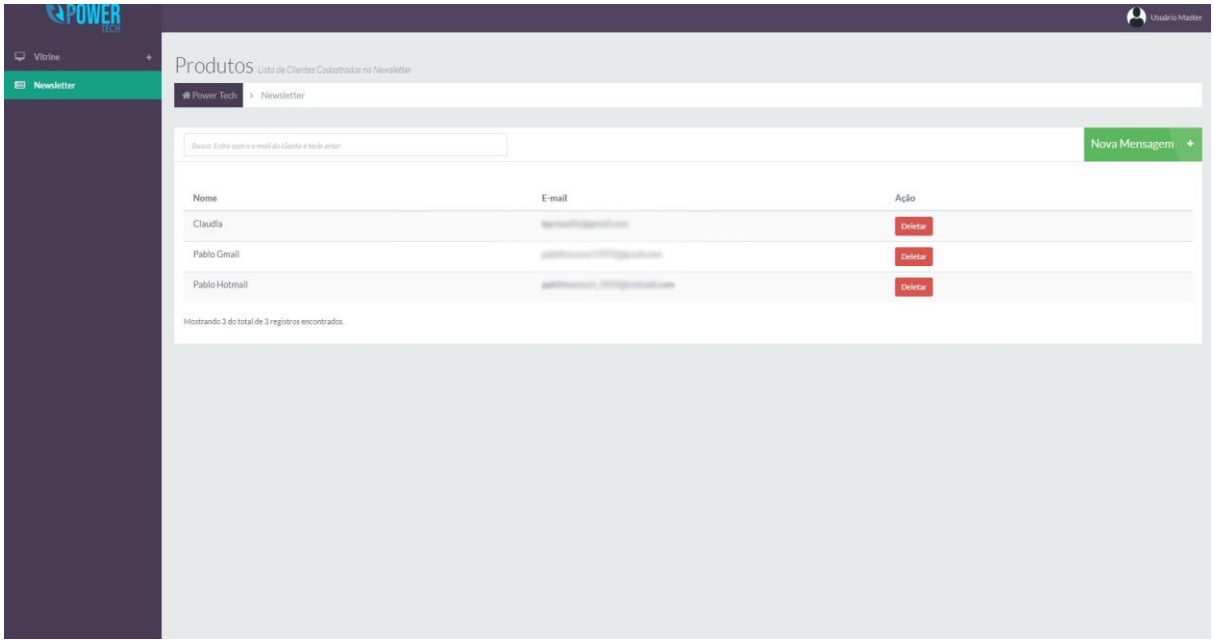
Por fim, o campo **Descrição** possui algumas opções de formatação de texto para que o administrador possa personalizar a descrição do produto. Após preencher todos os campos obrigatório e clicar no botão **Salvar**, encontrado no canto inferior direito da tela, o produto terá sido cadastrado com sucesso e o sistema irá redirecionar o *site* para a página de listagem de produtos.

Assim como na listagem de categorias, na listagem de produtos cada item exibe ao lado direito dois botões, o primeiro é destinado à edição, que ao ser clicado

leva a um formulário igual ao de cadastrado mas com os campos já preenchidos com os dados do produto em questão. Alterando os valores do formulário e clicando no botão **Salvar**, o produto será atualizado com os novos dados. O segundo botão tem a função de remover o produto. Uma vez acionado, é aberta uma *modal* (janela secundária que sobrepõe a atual e geralmente solicita alguma informação do usuário) de confirmação de exclusão, onde, por questão de segurança, o administrador irá escolher se deseja efetuar a ação ou cancelá-la.

Newsletter é a última opção do menu de módulos e através da qual é possível, tanto gerenciar os usuários cadastrados no *newsletter* do *site*, quanto disparar mensagens através da ferramenta.

Figura 6 - Listagem do newsletter



Fonte: Pablo Bittencourt Pereira Gobira (2019)

Após selecionar a opção no menu de módulos, o sistema irá carregar a tela de listagem de usuários que se cadastraram no *newsletter* através do formulário no *front-end*. Acima da listagem existe um campo de busca onde é possível pesquisar por um cadastro em específico, fornecendo o *e-mail* relacionado a ele.

Ao lado de cada cadastro existe um botão de exclusão, que ao ser acionado, assim como na exclusão do produto, abre uma *modal* para que seja feita a confirmação de tal ação.

Seguindo o padrão das telas analisadas anteriormente, no canto superior direito se encontra o botão de **Nova Mensagem**, que leva o usuário à tela de criação e disparo de *e-mail* para as pessoas cadastradas. Essa tela consiste em um formulário que possui dois campos: o assunto do e-mail e seu conteúdo. Uma vez que ambos os campos tenham sido preenchidos, basta clicar no botão **Disparar Mensagem** no canto inferior direito da tela e o e-mail será enviado a todas as pessoas cadastradas.

5. DESENVOLVIMENTO DAS ARQUITETURAS NA AWS

Neste capítulo iremos falar sobre a forma como foram criadas as arquiteturas propostas no capítulo 3 deste trabalho, utilizando a plataforma *AWS*. Começaremos falando sobre a criação dos elementos mais genéricos da infraestrutura, que são necessários para o desenvolvimento de ambas arquiteturas. Em seguida iremos nos aprofundar nos elementos de cada arquitetura separadamente. O primeiro passo dessa etapa foi criar nossa conta na AWS e, após acessar a plataforma, escolher a região de trabalho. A região define onde é alocada a maioria dos componentes físicos que são utilizados. A região mais próxima é a **sa-east-1** situada em São Paulo, porém alocar recursos nela é bem mais custoso. É importante lembrar tentamos aliar gastos com performance para que as soluções apresentadas tenham um bom custo-benefício. Por tanto escolhemos a região padrão sugerida pela plataforma, a **us-east-1** localizada no Norte da Virgínia. Em relação à **sa-east-1**, o preço da maioria das funcionalidades cai pela metade e sua localização não é muito distante, fator que garante uma navegação rápida aos usuários do sistema.

5.1. CONFIGURAÇÃO DA VPC

Após a criação da conta no AWS e definição da região utilizada, foi necessário fazer a configuração de uma *Virtual Private Cloud*, ou VPC. De acordo com Varia e Mathew (2014):

> "O *Amazon Virtual Private Cloud* lhe permite provisionar uma seção logicamente isolada da nuvem do *Amazon Web Services* (AWS), onde você pode iniciar recursos da AWS em uma rede virtual que tenha definido. Você tem controle total sobre seu ambiente de rede virtual, incluindo a seleção da sua própria faixa de endereços IP, a criação de sub-redes e a configuração de tabelas de roteamento e *gateways* de rede". (VARIA; MATHEW, 2014, p. 11, tradução nossa)[25].

Uma vez criada a VPC, é necessário criar sub-redes dentro dela, pois alguns recursos utilizados, como instâncias EC2, devem estar associados a uma ou mais sub-redes. Então foram criadas quatro sub-redes as quais denominamos de Public A, Public B, Private A e Private B. As sub-redes com o nome "Public" possuem tráfego de saída com a internet, enquanto as sub-redes "Private" não possuem. Além de diferenciar o nome das sub-redes entre Public e Private, também os diferimos utilizando as letras A e B. Elas estão relacionadas à zona de disponibilidade as quais cada rede pertence. Cada região da AWS está localizada em uma área geográfica diferente. Em cada uma dessas regiões existem vários locais isolados conhecidos como zonas de disponibilidade. A vantagem de se ter múltiplas sub-redes em diferentes zonas de disponibilidade é perceptível quando ocorre alguma falha relacionada a uma das zonas, impedindo o funcionamento dos componentes associados a ela. Nesse caso, as outras zonas podem continuar a prover normalmente os serviços, caso eles estejam associados múltiplas zonas.

Após a criação das sub-redes, passamos para a configuração das tabelas de roteamento. As tabelas de roteamento são responsáveis por definir o encaminhamento de pacotes de dados, portanto, cada sub-rede deve obrigatoriamente estar associada a uma tabela de roteamento. Para este trabalho

---

[25] "Amazon Virtual Private Cloud lets you provision a logically isolated section of the Amazon Web Services (AWS) Cloud where you can launch AWS resources in a virtual network that you define. You have complete control over your virtual networking environment, including selection of your own IP address range, creation of subnets, and configuration of route tables and network gateways".

criamos duas tabelas de roteamento, uma para a sub-rede pública (doravante tabela de roteamento pública) e outra para as sub-redes privadas (doravante tabela de roteamento privada). A diferença entre elas é que a tabela de roteamento pública está associada a um *internet gateway*, componente da VPC que permite a comunicação entre instâncias e a internet.

Ao finalizar as configurações de sub-rede e tabelas de roteamento, a infraestrutura estava pronta e já era possível gerar a instância que atuaria como servidor da aplicação. Na próxima seção iremos falar sobre as configurações utilizadas para a criação da primeira arquitetura analisada neste trabalho: a arquitetura cliente-servidor.

5.2 ARQUITETURA CLIENTE-SERVIDOR

Sabemos que a arquitetura cliente-servidor é a mais utilizada nos dias de hoje, no entanto, Arantes (2001) fala acerca dessa arquitetura, confirmando que no início do século XXI ela já era a arquitetura mais adotada pelos desenvolvedores:

> "Os modelos de gerência mais adotados atualmente são baseados na arquitetura cliente/servidor ou gerente/agente. Nesta arquitetura uma estação gerenciadora solicita operações aos elementos de rede e é responsável por todo o controle e processamento da atividade de gerência". (ARANTES, 2001, p. 18).

5.2.1 Criação da instância EC2

Nessa etapa nós construímos a primeira das duas arquiteturas estudadas nesse trabalho, utilizando as configurações da VPC feitas anteriormente e uma instância EC2. É importante ressaltar que durante a criação de uma instância EC2 existe uma série de escolhas que permitem ao usuário personalizá-la. Durante esse capítulo, comentamos apenas sobre as escolhas de maior importância. Em todas as outras mantivemos o valor padrão selecionado.

Ao criar nossa conta na AWS, estamos aptos a utilizar o nível gratuito em alguns serviços da plataforma durante o período de um ano. O EC2 é um desses serviços, no entanto só é possível utilizar o nível gratuito se selecionarmos certas configurações para a instância.

Ao iniciarmos o processo de criação da instância, devemos escolher a versão do seu sistema operacional. Dentre as opções disponíveis para o nível gratuito estavam o Windows Server 2019 Base, SUSE Linux Enterprise Server 12 SP4, Red Hat Enterprise Linux 8, entre outros. Dentre as opções disponíveis, escolhemos utilizar o Ubuntu Server 18.04 LTS, por conta da grande popularidade do sistema operacional Ubuntu, fator importante na hora de buscar por fontes de conhecimento na internet, como tutoriais e postagens em fóruns. Além disso, já possuir familiaridade com o sistema operacional foi outro ponto relevante para sua escolha, pois facilitou a configuração da instância via SSH.

Como já foi explicado durante o referencial teórico, as instâncias EC2 são disponibilizadas em vários modelos, com diferentes configurações de poder computacional. Para que pudéssemos utilizar o plano gratuito, foi obrigatório selecionar uma instância do tipo t2.micro. Esse tipo de instância utiliza apenas um núcleo Intel Xeon ou um núcleo AMD EPYC para a computação, além disso possui apenas 1GB de memória RAM.

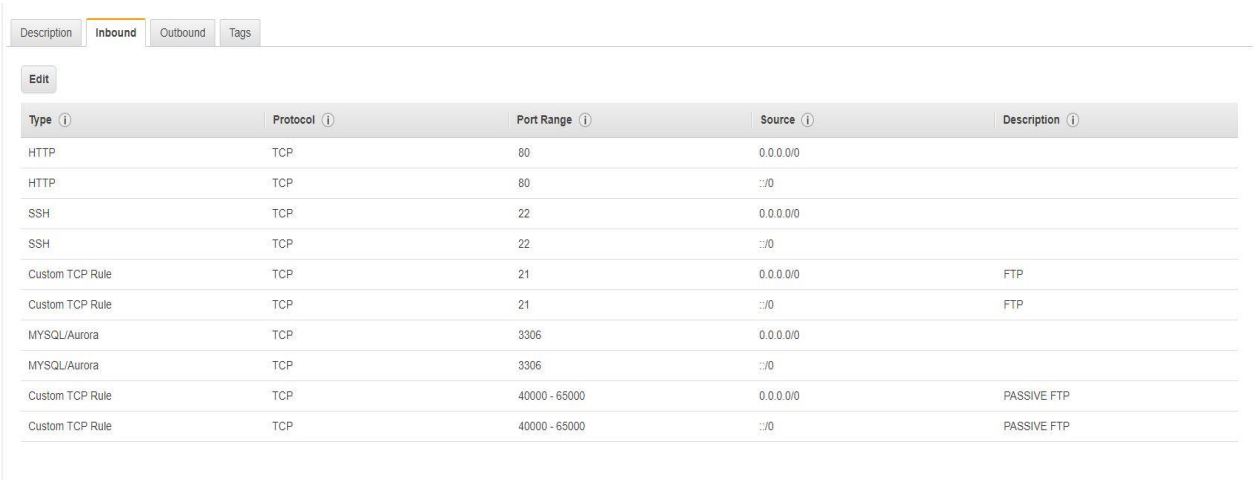
Sem dúvida as instâncias t2.micro são simples e não possuem grande capacidade computacional, mas sua configuração é mais do que suficiente para servir nossa aplicação. Por essa razão, optamos por utilizá-la em detrimento das outras opções. Assim foi possível tirar proveito do nível gratuito de serviço, diminuindo os custos de utilização dessa arquitetura.

Continuando com as configurações necessárias para o lançamento da instância, foi preciso definir a qual sub-rede ela participaria, então escolhemos a sub-rede Public A por estar associada a tabela de roteamento pública, que dá acesso externo à internet. Deixamos o tamanho do HD como 30GB, o máximo possível para uma instância do nível gratuito.

Em seguida foi necessário definir um grupo de segurança para a nossa instância. Os grupos de segurança funcionam como um firewall virtual na AWS. Eles são responsáveis por controlar o tráfego que chega às instâncias. Nós criamos um

grupo de segurança chamado Serverless-SG Padrão, e associamos a ele as permissões mostradas na Figura 7.

Figura 7 - Permissões do grupo de segurança



Fonte: Pablo Bittencourt Pereira Gobira (2019)

Ao definir o IP 0.0.0.0 no campo *Source* (fonte) em uma determinada porta, fica liberado o acesso daquela porta por qualquer IPv4. O mesmo ocorre nas regras cujo valor do campo "*Source"* é ::/0, mas nesse caso, o acesso é liberado para qualquer IPv6. As regras onde a *Port Range* (faixa de portas) é especificada como 40000 – 65000, especificam que todas as portas nesse intervalo estão abertas para acessos através do IP fonte. Os serviços liberados pelas regras do grupo de segurança apresentado na imagem são: HTTP, SSH, FTP ativo, MYSQL e FTP passivo.

Por fim criamos um par de chaves, uma chave pública guardada pela AWS e uma chave privada que é guardada pelo usuário. Essas chaves são utilizadas parada acessar a instância remotamente de forma segura. Após criar o par de chaves e salvar a chave privada em um local seguro, a instância finalmente foi lançada com sucesso.

5.2.2 Configuração da instância EC2

Após colocar a instância em funcionamento, basta acessá-la e fazer as configurações necessárias para que ela sirva à nossa aplicação. Para acessar e configurar a instância, foi usado o software de código aberto PuTTY, emulador de terminal que trabalha com os protocolos SCP, SSH, Telnet e rlogin. Neste trabalho utilizamos o protocolo SSH para acessar a instância; mas, antes disso, foi necessário obter um arquivo chave com a extensão “ppk”, uma vez que, por padrão, a chave privada gerada pela AWS no momento da criação da instância possui a extensão “pem”. Para fazer tal conversão, fizemos o uso da ferramenta PuTTYgen, programa auxiliar do PuTTy, responsável pela geração e conversão de chaves de acesso. Durante o processo de conversão, foi utilizado o algoritmo de criptografia RSA e 2048 como a quantidade de bits da nova chave.

Com a chave privada no formato “ppk” em mãos, com o IP público da instância gerada e o nome de usuário da instância (“ubuntu” é o nome de usuário padrão definido pela AWS para o sistema operacional escolhido), fizemos a conexão via SSH e começamos as configurações internas do servidor. Primeiramente foi preciso escolher qual software iriamos utilizar para servir a aplicação. Entre as possíveis opções estavam o Apache, Lighttpd, Nginx e Cherokee, mas acabamos optando pelo Nginx. O livro de Nedelcu, intitulado *Nginx HTTP Server* (2013), foi um fator determinante para que essa escolha fosse feita, pois ele mostra as qualidades do software em questão.

> “Há muitos aspectos em que o Nginx é mais eficiente que seus concorrentes. Em primeiro lugar, velocidade. Fazendo uso dos soquetes assíncronos, o Nginx não é inicializado tantas vezes quanto recebe solicitações. Um processo por núcleo é suficiente para lidar com milhares de conexões, permitindo uma carga de CPU e um consumo de memória muito mais leves”. (NEDELCU, 2013, p. 21-22, tradução nossa)[26].

O fato do Nginx proporcionar uma redução no uso de memória e poder de processamento, em conjunto com a velocidade de resposta, fez com que a escolha

[26] “There are many aspects in which Nginx is more efficient than its competitors. First and foremost, speed. Making use of the asynchronous sockets, Nginx does not spawn as many times as receives requests. One process per core suffices to handle thousands of connections, allowing for a much lighter CPU load and memory consumption.”

do software parecesse bastante atrativa. A eficiência do servidor se torna ainda mais relevante no contexto deste trabalho, afinal de contas estamos lidando com uma instância de baixa capacidade computacional, portanto optamos pela utilização do Nginx.

Neste ponto já havíamos concluído o desenvolvimento da aplicação e agora possuíamos o software para servi-la. No entanto, ainda era necessária uma forma de subir os arquivos da aplicação para a instância. Precisávamos utilizar um servidor FTP e para esse propósito escolhemos o software VSFTPD. Após o processo de instalação do software, o configuramos de forma que o acesso fosse feito através do usuário “ubuntu”, tendo a pasta onde se encontrava a aplicação como pasta raiz do acesso FTP. É importante também relatar que utilizamos o modo de operação passivo do FTP, fazendo uso do intervalo de portas 40000 até 65000, cujo acesso já havia sido liberado no grupo de segurança associado a instância.

Com o servidor FTP funcionando, fizemos o *upload* dos arquivos da aplicação para a instância e em seguida instalamos o software Composer, um gerenciador de dependências utilizado em aplicações PHP. Com apenas uma linha de comando no terminal, o Composer instalou todas bibliotecas e extensões necessárias para executar a aplicação, definidas no arquivo “composer.json”. Restava configurar apenas três elementos para que a arquitetura cliente-servidor estivesse completa e servindo a aplicação: o banco de dados, o repositório de armazenamento de arquivos e os apontamentos de domínio.

5.2.3 Repositório de Arquivos

O serviço de armazenamento de arquivos da AWS chama-se S3 (*Simple Storage Service*) e nele é possível criar repositórios públicos e privados chamados de *buckets*. Esse serviço é de grande importância para o funcionamento da aplicação na arquitetura cliente-servidor, pois as imagens dos produtos que são cadastrados no sistema precisam ser armazenadas e acessadas de algum lugar seguro. Para tal propósito criamos um *bucket*, o qual denominamos de “bucket-tcc-pablo”, e lhe demos permissão pública apenas para leitura, de forma que as

imagens dos produtos cadastrados pudessem ser mostradas na vitrine. Após criado o *bucket,* adicionamos suas credenciais ao arquivo “filesystems.php”, reponsável por definir qual é o repositório de arquivos utilizado pela aplicação.

5.2.4 Banco de Dados

Entre os sistemas de banco de dados disponíveis, selecionamos o MySQL para este trabalho. Em seu livro, Converse, Park e Morgan (2004) discorrem com grande detalhamento sobre a utilização do PHP e MySQL em conjunto, demonstrando as vantagens dessa combinação. Os autores citam uma série de benefícios advindos da utilização do MySQL, como velocidade, estabilidade e compatibilidade com múltiplas plataformas. No entanto, foi a existência de uma ampla e prestativa comunidade que pendeu a balança para que fosse feita a escolha em favor do MySQL.

> “MySQL, embora seja licenciado como tendo código-fonte para usos não-redistributivos, é um pouco menos impulsionado pela comunidade em termos de seu desenvolvimento. No entanto, se beneficia de uma comunidade crescente de usuários que são ouvidos ativamente pela equipe de desenvolvimento. Raramente um projeto de software respondeu tão vigorosamente à demanda da comunidade. E a comunidade de usuários pode ser extremamente responsiva a outros usuários que precisam de ajuda”. (CONVERSE, PARK, MORGAN, 2004, p. 17, tradução nossa)[27].

O serviço de banco de dados da AWS segue o mesmo padrão da instância EC2 em relação a seu modelo de negócio. São disponibilizadas várias configurações de bancos diferentes, com variações de performance e de espaço de armazenamento. Além disso, o serviço de banco de dados da AWS, o RDS (*Relational Database Service*), também possui um nível gratuito, que permite selecionar apenas à configuração mais básica disponível no momento de criação do banco (db.t2.micro). No entanto, assim como a instância EC2, que possui uma configuração bem modesta por conta do nível gratuito, um banco de dados com a

[27] “MySQL, while open-source licensed for nonredistributive uses, is somewhat less community driven in terms of its development. Nevertheless, it benefits from a growing community of users who are actively listened to by the development team. Rarely has a software project responded so vigorously to community demand. And the community of users can be extremely responsive to other users who need help”.

configuração mais simples da AWS é mais do que suficiente para guardar e servir os dados da aplicação. No entanto, cada tipo de instância de banco de dados na Amazon possui um número limite de acessos simultâneos. No caso da instância db.t2.micro, esse limite é de 66 usuários. Analisamos mais a fundo como esse fator afetou a aplicação no capítulo 6.1, onde discorremos sobre os testes de performance.

Durante a criação do banco, associamos a ele um grupo de segurança próprio chamado “SecurityGroup-DB”, cuja única permissão é de acesso irrestrito à porta 3306, responsável pelo contato com o sistema MySQL. Foram definidas também as credenciais do usuário *master*, então, finalmente fizemos a atualização dos dados do arquivo “database.php” do Laravel, que define o banco de dados que será utilizado pela aplicação. Com a estrutura do banco de dados já em estabelecida, fizemos o acesso à mesma através do software PhpMyAdmin e criamos a base de dados que seria utilizada. Em seguida, fomos até a pasta onde se encontrava nossa aplicação e rodamos um comando do Laravel para criar todas as tabelas e semear o banco com os dados essenciais para o funcionamento do sistema.

5.2.5 DNS

Após serem feitos todos os processos acima, a aplicação estava praticamente funcionando, bastava configurar o Nginx para servi-la através de um domínio. Contratamos o domínio “powertechserverless.com.br”, através da plataforma brasileira “registro.br”. No entanto, como o nome do nosso domínio remetia mais a arquitetura *serverless*, decidimos apontar um subdomínio para a arquitetura cliente servidor. Dessa forma, criamos o apontamento do subdomínio “clienteservidor.powertechserverless.com.br” para o IP do servidor, através de uma entrada do tipo A, também conhecida como *hostname*. Após fazer o apontamento, bastou fazer as configurações de *virtual host* no Nginx do servidor, para associar o subdomínio em questão à pasta onde se encontrava a aplicação a ser servida.

Finalmente, reiniciamos o serviço do Nginx e a aplicação estava sendo servida por nossa instância, pronta para que fossem feitos os testes. Com primeira arquitetura finalizada, partimos para o desenvolvimento da arquitetura sem servidor.

5.3 ARQUITETURA *SERVERLESS*

Com a arquitetura cliente-servidor configurada e funcionando apropriadamente, iniciamos o desenvolvimento da arquitetura *serverless*. A arquitetura cliente-servidor não foi desenvolvida primeiro por acaso. Embora na arquitetura *serverless* não exista um servidor propriamente dito, para desenvolve-la da maneira utilizada por nós (com a ajuda do software Bref), foi necessária uma máquina para acionar alguns comandos via *prompt*. Portanto, a instância responsável por servir a aplicação em nossa primeira arquitetura desempenhou um papel no processo de criação da arquitetura sem servidor.

A VPC já estava configurada previamente, possuindo sub-redes públicas e privadas. Essa configuração é ideal, já que a instância criada nos capítulos anteriores foi associada à uma sub-rede pública (pois necessita de acesso a internet para servir a aplicação), enquanto função Lambda, responsável por receber e responder as requisições da arquitetura sem servidor, foi associada à sub-rede privada.

Na seção a seguir, relatamos as experiências acerca da utilização do *plugin* Bref e da criação e configuração da função Lambda, o coração da arquitetura sem servidor.

5.3.1 Bref

Conforme o que foi explicitado no Referencial Teórico deste trabalho, as funções Lambda não possuem suporte para qualquer linguagem de programação, incluindo a linguagem utilizada pela aplicação criada neste trabalho, o PHP. No entanto existe uma forma de superar esse obstáculo. Essa solução se baseia em

compilar um arquivo binário do PHP em um ambiente Linux compatível com o ambiente utilizado para executar as funções Lambda. Através da utilização desse arquivo binário compilado em conjunto com as camadas (*layers*) da função Lambda, é possível interpretar a linguagem PHP dentro da função Lambda quando ela for acionada (nesse caso, ao receber uma requisição HTTP), permitindo que as requisições sejam respondidas. Esse processo pode ser acompanhado detalhadamente na matéria intitulada *AWS Lambda Custom Runtime for PHP: A Practical Example*, escrita por Michael Moussa em 14 de dezembro de 2018, veiculada no blog da AWS (conf. referencia bibliográfica).

Mas existia uma alternativa mais prática do que realizar o processo de compilação do arquivo binário e em seguida configurar e lançar manualmente a função Lambda. Essa alternativa envolve a utilização do Bref, *plugin* criado para facilitar o desenvolvimento de arquiteturas sem servidor de vários tipos, incluindo aquelas dedicadas à servir aplicações HTTP, como a que nós criamos. Além disso, o Bref possui uma seção específica em sua documentação destinada a auxiliar na implantação de uma aplicação em Laravel. Nessa seção são disponibilizadas até mesmo uma série de configurações otimizadas para o framework PHP. Por conta dos motivos citados acima, decidimos utilizar o *plugin* para assistir no desenvolvimento da arquitetura.

Acessamos a instância servidor da arquitetura cliente-servidor via SSH e então fomos até a pasta onde se encontrava o projeto. Após acessar a pasta em questão, executamos o comando “composer require mnapoli/bref”, responsável por instalar os pacotes do Bref. Com os pacotes instalados, executamos também o comando “vendor/bin/bref init”, que inicializa o projeto após ao criar os arquivos “index.php”, onde fica o código da função Lambda, e o “template.yaml”, que contém a configuração do AWS SAM (*Serverless Application Model*) necessária para lançar a infraestrutura da arquitetura.

O Bref inclui algumas das extensões mais utilizadas do PHP em sua biblioteca, mas uma importante extensão para o funcionamento da nossa aplicação, embora incluída na lista de extensões disponibilizadas pelo Bref, não estava habilitada por padrão. Felizmente, a documentação do Bref possuía as instruções para habilitá-la, as quais consistiam em criar o arquivo “php/conf.d/php.ini” e dentro

dele especificar o nome da extensão que queríamos que fosse habilitada. Essa extensão era o MySQL PDO Driver; logo, incluímos a linha “extension=pdo_mysql” no arquivo e o problema foi resolvido.

Esse foi o momento de utilizar as configurações de lançamento otimizadas para aplicações criadas em Laravel, disponibilizadas pelo Bref. Para isso, na documentação do Bref[28], é possível encontrar um código que deve ser substituído no arquivo “template.yaml”. Além de substituir o código do arquivo em questão, duas outras alterações no código de arquivos do Laravel foram feitas através das instruções da documentação. Os dois arquivos alterados foram “bootstrap/app.php” e “app/Providers/AppServiceProvider.php”.

Para finalizar os preparativos do lançamento da nossa função, executamos um último comando de terminal, o comando “php artisan config:clear”, que deleta o arquivo “bootstrap/cache/config.php”. Isso é necessário porque lançar o Laravel junto com seus arquivos de cache pode resultar em erros ao servir a aplicação via AWS Lambda, por conta da diferença no caminho dos arquivos. Essas foram todas as configurações relacionadas ao Bref, nesse momento já era possível lançar a função Lambda, bastando apenas executar dois comandos no terminal. No entanto, ainda era necessário criar um *bucket* no S3 para receber o código da função Lambda e configurar o API Gateway para que fosse utilizado o domínio que contratamos em vez de um *endpoint* aleatório.

5.3.2 S3

Para lançar a aplicação através do AWS SAM precisamos de um *bucket* no S3 que comporte o código da nossa aplicação. Na documentação do Bref, os desenvolvedores do plugin determinam que “o conteúdo desse *bucket* será gerenciado pelo AWS SAM. Não o use em seu aplicativo para armazenar itens como ativos ou arquivos enviados”[29]. De acordo com essas especificações, não podemos

[28] https://bref.sh/docs/frameworks/laravel.html

[29] “the content of this bucket will be managed by AWS SAM. Do not use it in your application to store things like assets or uploaded files”.

utilizar o mesmo *bucket* que criamos para a aplicação (bucket-tcc-pablo), então criamos um novo bucket, dessa vez com acessibilidade privada, e o denominamos de “bucket-serverless-pablo”. Em seguida, configuramos o API Gateway de forma que a função Lambda *service* a aplicação através do domínio que contratamos.

5.3.3 API Gateway

Mais uma vez seguimos as instruções encontradas na documentação do Bref para concluir essa etapa. Antes de configurar o API Gateway, foi preciso registrar o domínio da aplicação no ACM (AWS Certificate Manager), com o intuito de conseguir um certificado HTTPS para o mesmo. Esse passo é obrigatório para que seja possível servir uma aplicação através de um domínio personalizado na AWS Lambda. Uma vez criado o certificado, é gerado um registro do tipo CNAME que deve ser adicionado no provedor de serviços DNS do domínio. O intuito desse procedimento é verificar se de fato somos os detentores do domínio, portanto realizamos a operação no painel administrativo do site “registro.br” e o certificado foi validado.

Continuando o processo para habilitar os domínios personalizados no API Gateway, ao acessar a aba “Nomes de domínio” selecionamos o botão “Criar nome de domínio personalizado”. Então, na escolha do protocolo utilizado selecionamos o HTTP, no campo “Nome de domínio destino” entramos com o domínio que contratamos previamente, em “Configuração do endpoint” selecionamos a opção “Otimizado para fronteiras” e em “Certificado do ACM” escolhemos o certificado que foi gerado no procedimento anterior.

Figura 8 - Nome de domínio personalizado após criação

Fonte: Pablo Bittencourt Pereira Gobira (2019)

As configurações do API Gateway não estavam completas ainda, mas nesse ponto foi necessário fazer uma operação intermediária antes de continuar com as configurações que estavam em andamento. Era preciso lançar a arquitetura, necessidade que foi suprida após executar apenas dois comandos no terminal da instância que servia nossa aplicação.

O primeiro comando utilizado foi “sam package --output-template-file .stack.yaml --s3-bucket bucket-serverless-pablo”. Esse comando é responsável por duas ações. A primeira é criar um arquivo chamado “stack.yaml” com as instruções de inicialização de uma *stack* (pilha) no AWS *CloudFormation*, baseando-se nos dados encontrados no arquivo “template.yaml”. A segunda ação foi enviar o arquivo criado para o bucket S3 “bucket-serverless-pablo”, que foi criado especificamente para receber esses arquivos.

O segundo comando executado foi o “sam deploy --template-file stack.yaml --capabilities CAPABILITY_IAM --stack-name stack-serverless-tcc”. Esse comando inicializou de fato a arquitetura, pois criou a *stack* e lançou a função Lambda de acordo com os dados do arquivo “stack.yaml”.

A *stack* que acabou de ser criada é um contêiner que engloba um conjunto de elementos da AWS que estão integrados ao projeto. Esse recurso é gerenciado pela ferramenta AWS *CloudFormation*. O intuito de utiliza-la é poder provisionar recursos

de infraestrutura de forma segura, padronizada e repetível. Além disso, ao deletar uma *stack*, todos os elementos contidos nela também são automaticamente excluídos, diminuindo o trabalho manual e a chance de erro humano. No caso desse projeto, os recursos que foram alocados em nossa *stack* consistem em um conjunto de permissões e papéis (estruturas de permissão provenientes no AWS *Identity and Access Management*), estruturas do API Gateway e nossa função Lambda.

Após a criação da função Lambda, a arquitetura *serverless* estava lançada, então retornamos ao API Gateway para finalizar as configurações que restavam. Após selecionar a opção de editar o domínio personalizado que criamos previamente, adicionamos um novo mapeamento com os dados ilustrados na imagem abaixo.

Figura 9 - Configurações do nome do domínio personalizado

Fonte: Pablo Bittencourt Pereira Gobira (2019)

Para finalizar as alterações no API Gateway, acessamos as configurações da estrutura da API que foi criada através do terminal e em “Tipos de mídia binários”

adicionamos a nova mídia “multipart/form-data”. Isso determina a maneira como os dados dos formulários da aplicação serão codificados ao serem passados para a aplicação.

Os procedimentos necessários para servir a aplicação estavam chegando ao fim, bastava fazer algumas configurações na função Lambda e a construção da arquitetura sem servidor estaria finalizada.

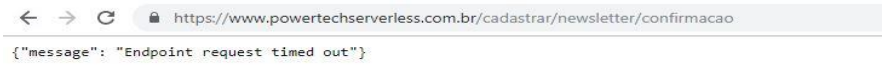
5.3.4 AWS Lambda

Ao acessar a função Lambda nomeada “laravel-tcc”, responsável por receber e responder as requisições da aplicação, foi preciso efetuar algumas modificações. Foi necessário associá-la à mesma VPC que o banco de dados da aplicação. Esse procedimento foi feito para que a função Lambda pudesse se comunicar com a instância do banco de dados. Essa comunicação é crucial para a execução da aplicação, uma vez que, para servi-la ao usuário, é necessário receber e enviar informações ao banco de dados.

Uma vez associada a uma VPC, as funções Lambda exigem que sejam associadas também uma ou mais sub-redes (de preferência duas sub-redes que estejam em zonas de disponibilidade distintas, por questões de segurança) e a um ou mais grupos de segurança. Quanto às sub-redes, associamos a função às duas sub-redes privadas (Private A e Private B) que havíamos criado ao configurar a VPC. O grupo de segurança associado foi o mesmo criado para a instância cliente-servidor (Serverless-SG Padrão).

Com essas configurações, tecnicamente estava tudo pronto para acessar a aplicação através da função Lambda, então fizemos o acesso das páginas e utilizamos todas as funcionalidades do site para garantir que tudo estava funcionando como deveria. Para a nossa surpresa, o sistema de *newsletter* não funcionou corretamente. Após algum tempo de análise, percebemos que sempre que a ação do sistema consistia em enviar um e-mail (tanto os de confirmação do cadastro quanto os disparados aos clientes do *newsletter*), um erro era apresentado pelo sistema.

Figura 10 - Erro ao enviar e-mails



Fonte: Pablo Bittencourt Pereira Gobira (2019)

Através da mensagem de erro "*Endpoint request timed out*" (requisição de ponto de acesso expirada), descobrimos que a aplicação estava tentando executar a tarefa sem sucesso até atingir o tempo máximo de execução, comportamento que nos levou a cogitar a possibilidade de que fosse algo relacionado à comunicação com a internet. Após pesquisar sobre o assunto, chegamos à conclusão de que nossa suspeita estava correta. As funções Lambda não possuem acesso externo à internet por padrão quando associadas a uma VPC. Por esse motivo, a nossa função Lambda conseguia apenas receber e servir requisições de. No entanto, para utilizar um serviço de envio de e-mail, como o Gmail no caso da nossa aplicação, é preciso fazer contato com o servidor do serviço desejado na internet.

Dada a situação, precisávamos encontrar uma forma de conceder acesso externo à internet a nossa função Lambda, para que o serviço de *newsletter* fizesse parte da aplicação. De acordo com a documentação do AWS Lambda:

> "Se sua função Lambda precisar de acesso à Internet, não a conecte a uma sub-rede pública ou a uma sub-rede privada sem acesso à Internet. Em vez disso, anexe-o apenas a sub-redes privadas com acesso à Internet por meio de uma instância do NAT ou de um gateway NAT do Amazon VPC". (AWS LAMBDA DEVELOPER GUIDE, 2019, tradução nossa)[30]

[30] "If your Lambda function needs Internet access, do not attach it to a public subnet or to a private subnet without Internet access. Instead, attach it only to private subnets with Internet access through a NAT instance or an Amazon VPC NAT gateway."

Através desse trecho, pudemos perceber que conceder acesso à internet para uma função Lambda não era tão simples quanto fazer o mesmo para uma instância EC2, onde bastava associar a instância a uma sub-rede pública, cujos acessos são destinados a um Internet Gateway pela tabela de roteamento. Era necessário utilizar um NAT Gateway ou uma instância NAT. Ambas as opções são responsáveis por fazer a mesma coisa: permitir que recursos associados a uma sub-rede privada se conectem à internet ou a outros serviços da AWS, mas impedir que esses recursos recebam tráfego de saída iniciado por outra pessoa na internet.

Nesse ponto do desenvolvimento do trabalho, embora tivéssemos descoberto a solução do problema, outra complicação surgiu, relacionada ao custo mensal da infraestrutura. Como já foi dito em capítulos anteriores, o propósito deste trabalho é verificar a viabilidade da arquitetura sem servidor, logo, um dos fatores mais importantes para determinar isso é o custo mensal da infraestrutura. O uso de um NAT Gateway possui um custo de $0,045 por hora na região que escolhemos para trabalhar, a Virginia do Norte - EUA (região com os custos mais baixos disponíveis pela AWS). Em 24 horas de uso desse recurso seriam gastos $1,08 dólares, com a cotação do dólar em R$3,83 no dia 10 de abril de 2019 seria o equivalente a R$4,14 de gasto por dia. No período de 30 dias, o custo somente desse recurso seria de aproximadamente R$124,20, um valor inviável. Além disso, o Nat Gateway ainda cobra $0,045 adicionais para cada gigabyte de dados processados, o que no caso da nossa aplicação não faz muita diferença, mesmo com uma grande quantidade de requisições, já que a o tráfego de dados é extremamente baixo.

Os valores decorrentes de criar e manter uma instância NAT, que desempenha o mesmo papel de um NAT Gateway (mas na prática é uma instância EC2), se mostraram muito mais viáveis. O valor da contratação de uma instância t2.nano sob demanda, a mais barata disponível, é de $0.0058 por hora de utilização, o que em um mês dá um total aproximado de $4,18 ou R$16,00 reais, valor bem mais econômico que a opção anterior.

Uma vez que a conta da AWS utilizada para conduzir este trabalho possuía menos de um ano de existência, foi possível se beneficiar do plano gratuito para economizar ainda mais; logo, a instância NAT criada foi do tipo t2.micro, única opção disponível no plano gratuito.

Após criar a instância NAT, voltamos rapidamente às configurações da VPC e criamos uma nova entrada na tabela de roteamento das sub-redes privadas. Essa nova entrada teve como destino a entrada 0.0.0.0/0, que representa todos os endereços IPv4 não especificados na tabela, e como alvo a Interface de Rede Elástica (Elastic Network Interface ou ENI) criada pela instância NAT. Dessa forma, as sub-redes privadas ganharam acesso externo à internet, o que concedeu o mesmo acesso a nossa função Lambda, uma vez que ela estava associada a essas sub-redes.

Após fazer as alterações necessárias, finalmente o problema estava resolvido e a aplicação estava sendo executada pela arquitetura *serverless* e acessada através do domínio “powertechserverless.com.br”.

6. ANÁLISE DAS ARQUITETURAS

Esse capítulo foi destinado a analisar as arquiteturas criadas e determinar, dentro das categorias de avaliação definidas no capítulo 3.3, as qualidades e defeitos de cada uma delas, com o intuito de chegar a uma conclusão concreta sobre a viabilidade da arquitetura sem servidor nos dias de hoje.

Primeiramente fizemos a análise dos fatores primários de avaliação (performance e custo) e em seguida dos fatores secundários (compatibilidade com recursos externos e gerenciamento).

6.1 PERFORMANCE

De acordo com a definição de Erinle (2015, p. 3, tradução nossa) em seu livro, cujo tema é a condução de testes de performance em websites através da ferramenta JMeter, “O teste de desempenho é um tipo de teste destinado a determinar a capacidade de resposta, confiabilidade, taxa de transferência,

interoperabilidade e escalabilidade de um sistema e/ou aplicativo sob uma determinada carga de trabalho”[31].

Nesse capitulo, nós conduzimos testes que buscaram demonstrar se as capacidades citadas acima estão ou não presentes nas duas arquiteturas criadas durante o trabalho. Para que os testes fossem feitos de forma realista, foi preciso simular o acesso simultâneo de uma série de usuários, em cada arquitetura separadamente. Para alcançar esse objetivo, utilizamos a extensão BlazeMeter e o *website*[32] da mesma aplicação, por meio do navegador Google Chrome.

A função da extensão de navegador, programa que auxilia na condução de testes de carga em aplicações web, foi gravar o caminho de páginas que o usuário fictício realizou durante seu acesso à aplicação em um arquivo do tipo JMX. A utilidade desse arquivo é permitir que sejam feitos testes de simulação de acesso mais realistas, uma vez que os usuários fictícios estarão seguindo os passos de um humano para acessar a aplicação. Os testes foram realizados em ambas arquiteturas e é importante destacar que o caminho de acesso simulado foi o mesmo para elas, de forma que os resultados gerassem uma comparação justa.

A finalidade do *website* foi utilizar os scripts de acesso criados pela extensão para enviar uma série requisições para os domínios das arquiteturas, simulando acesso de vários usuários simultaneamente e em seguida exibir os dados relativos às respostas dessas requisições.

É importante destacar previamente que durante a série de testes executada, percebemos que algo estava causando uma limitação no fluxo de acesso, pois a partir de um certo ponto após aumentar a carga de acesso na simulação, todas as requisições passavam a não ser respondidas.

Após pesquisar a respeito, descobrimos o motivo do problema. As instância de banco de dados provisionadas pelo RDS possuem um limite máximo de conexões simultâneas, de acordo com o seu tipo. A instância criada para essa aplicação é uma t2.micro, a opção mais simples e com menor capacidade

---

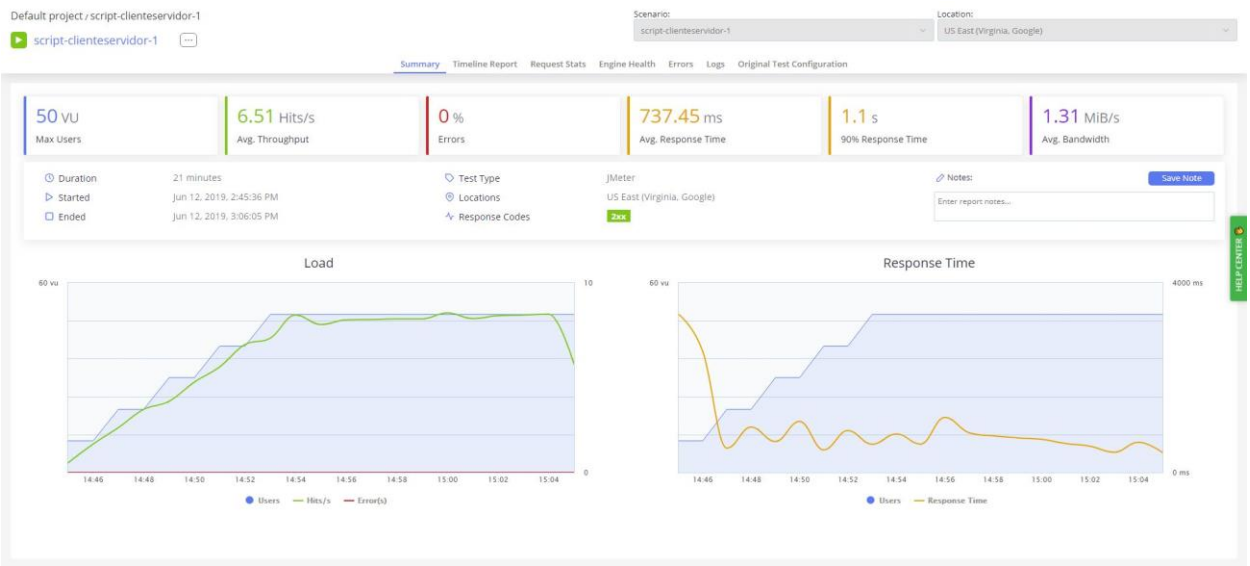
[31] “Performance testing is a type of testing intended to determine the responsiveness, reliability, throughput, interoperability, and scalability of a system and/or application under a given workload”.

[32] https://blazemeter.com/

disponível, que foi escolhida por ser a única disponível no plano gratuito. Esse tipo de instância permite no máximo 66 conexões simultâneas, portanto todas as requisições de acesso que necessitarem de informações do banco de dados não serão atendidas uma vez que esse limite tenha sido alcançado. Dessa forma, para aumentar a capacidade de acessos simultâneos à aplicação seria necessário aprimorar o tipo de instância do banco de dados, ação que incorreria em um aumento de custo considerável. Por essa razão não fizemos esse procedimento e nos ativemos a executar os testes dentro do limite de usuários simultâneos estabelecido pelo banco de dados.

Para a configuração dos testes escolhemos um número máximo de 50 usuários acessando o site ao longo de um período de 20 minutos. Ao início dos testes, apenas 10 usuários estavam ativos. A cada 2 minutos, 10 novos usuários passavam a acessar o site em conjunto com os usuários previamente ativos. Esse processo se repetiu até que os 50 usuários estivessem ativos e eles se mantiveram assim pelo resto dos 20 minutos, efetuando requisições durante todo o tempo. Os acessos foram feitos a partir da região da Virginia nos EUA. Após finalizados os testes, a plataforma do BlazeMeter entrou em ação, executando as análises em cima dos resultados obtidos e apresentando essas informações no painel ilustrado na Figura 11:

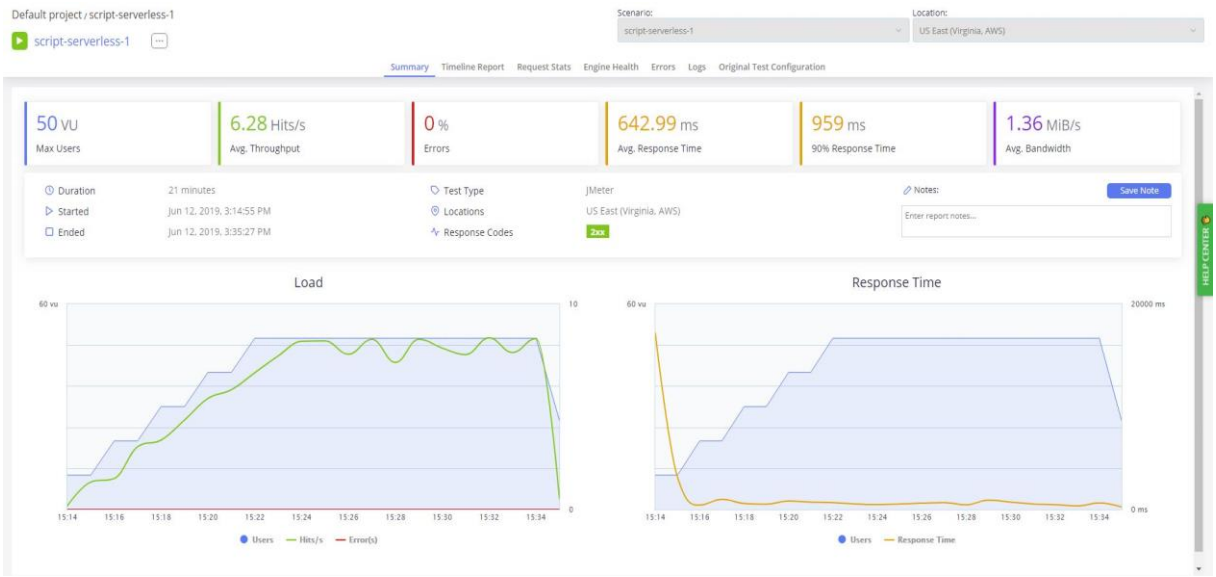
Figura 11 - Resultado dos testes na arquitetura cliente-servidor



Fonte: Pablo Bittencourt Pereira Gobira (2019)

O painel mostra uma série de informações, que foram analisadas de forma comparativa, após a realização dos testes na outra arquitetura. Dentre os dados apresentados, o mais importante é o tempo médio de resposta (*Avg. Response Time*), pois o tempo de resposta das requisições é um dos fatores mais relevantes para determinar a eficiência da aplicação. Outro fator muito importante é a quantidade de requisições não respondidas, representadas no painel pela porcentagem de erros. Como pode ser visto através da imagem, não houve nenhum erro durante os 20 minutos de teste. Em seguida executamos o teste na arquitetura *serverless*, com as mesmas configurações e utilizando um script de acesso com passos idênticos aos feitos em cima da arquitetura cliente-servidor (Figura 12).

Figura 12 - Resultado dos testes na arquitetura sem servidor



Fonte: Pablo Bittencourt Pereira Gobira (2019)

A partir dos resultados que obtivemos ao executar os testes na arquitetura sem servidor, pudemos notar alguns pontos interessantes. O tempo médio de resposta, embora próximo ao da arquitetura cliente-servidor, foi aproximadamente 94 milissegundos mais baixo. No entanto, a arquitetura sem servidor apresentou uma quantidade média de acessos por segundo (*Avg. Throughtput*) um pouco menor, além de possuir a largura de banda levemente mais alta, em comparação à arquitetura cliente-servidor. Esses fatores influenciam o tempo de resposta médio da

aplicação, portanto, por conta dessas condições que favorecem um pouco à arquitetura serverless, podemos considerar um empate no quesito tempo médio de resposta. Em relação a quantidade de erros, as duas arquiteturas também ficaram equiparadas, não apresentando nenhuma falha de requisição durante os testes executados.

As duas arquiteturas apresentaram um tempo de resposta alto durante a interação dos primeiros usuários com a aplicação. No entanto, esses tempos diferem bastante de uma arquitetura para outra, sendo de 3333 milissegundos na arquitetura cliente-servidor e de 17154 milissegundos na arquitetura sem servidor. Essa grande diferença se dá por conta de um fenômeno chamado de *cold starts* (inicios frios), problema proveniente da forma como as funções Lambda funcionam ao receber uma requisição. Byrro (2019) explica de forma mais detalhada o que são os *cold starts*:

> “Inicio frio refere-se ao estado em que nossa função estava ao atender a uma solicitação de chamada específica. Uma função sem servidor é servida por um ou vários microcontêineres. Quando uma solicitação chega, nossa função verificará se já existe um contêiner em execução para veicular a invocação. Quando um contêiner ocioso já está disponível, chamamos de contêiner "quente". Se não houver um contêiner prontamente disponível, a função gerará um novo e isso é o que chamamos de "início a frio"”. (BYRRO, 2019, tradução nossa)[33].

De acordo com a explicação de Byrro, ao invocar a função Lambda é necessário criar um contêiner, caso não exista um já criado para aquela função, o que faz com que a invocação da função, e consequentemente a resposta da requisição, leve mais tempo. Os tempos de resposta dos *cold starts* variam bastante, o que pode ser um problema em alguns casos, como no teste ilustrado pela figura 12, onde as requisições iniciais levaram cerca de 17 segundos para serem respondidas.

Existem alguns métodos para tentar remediar o problema, como aumentar o espaço de memória que pode ser utilizado pela função. Por esse motivo, aumentamos a memória máxima utilizada pela função de 128 MB (megabytes) para

[33] “Cold start refers to the state our function was when serving a particular invocation request. A serverless function is served by one or multiple micro-containers. When a request comes in, our function will check whether there is a container already running to serve the invocation. When an idle container is already available, we call it a “warm” container. If there isn’t a container readily available, the function will spin up a new one and this is what we call a “cold start””.

1024 MB. Outra forma de diminuir o tempo dos *cold starts* é executar a função Lambda fora de uma VPC, o que é impossível no caso desse trabalho, já que a única maneira de manter o contato entre função Lambda e RDS é associar ambas a mesma VPC. Uma forma de contornar o problema é tentar manter a função "pré-aquecida", ou seja, manter seu contêiner em existência em tempo integral. Para fazer isso é necessário conseguir uma forma de invocar a função continuamente após um determinado intervalo de tempo. Esse feito pode ser alcançado utilizando o AWS Cloudwatch, mas pode resultar em aumento de custo, dependendo da frequência com que a função é invocada e o custo de execução da mesma.

É importante ressaltar que o resultado dos testes de performance poderia ter sido mais favoráveis para a arquitetura *serverless*, caso as simulações fossem feitas com mais usuários. O motivo disso está no fato de que a arquitetura *serverless* possui escalabilidade automática, enquanto a arquitetura cliente-servidor não.

6.2 CUSTOS

Para fazer a análise de custo das arquiteturas, segundo fator primário de comparação, usamos a calculadora de custos da AWS[34] em conjunto com as informações de configuração de cada ferramenta utilizada. Primeiramente definimos quais são as funcionalidades que geram gasto e são indispensáveis para ambas arquiteturas. Uma vez que essas funcionalidades geram custo de maneira igual para as duas arquiteturas, não as incluímos nessa análise. Nesse quesito se encaixam o RDS e o S3, pois independente da arquitetura é necessário que a aplicação acesse o banco de dados e disponha de um serviço de armazenamento de arquivos, necessário para que sejam salvas as imagens dos produtos cadastrados. Dessa forma, a análise de custo de cada arquitetura foi feita com base nas ferramentas que compõem a sua infraestrutura e não são utilizadas pela arquitetura concorrente.

[34] https://calculator.s3.amazonaws.com/index.html

6.2.1 Custos da Arquitetura Cliente-Servidor

Quando pensamos em arquitetura cliente-servidor do ponto de vista infraestrutural, a primeira coisa que vem em mente é a máquina que irá servir a aplicação, que no nosso caso é uma instância EC2. De fato o único gasto infraestrutural específico dessa arquitetura será proveniente da instância EC2 que utilizamos como servidor. Para comportar a maioria dos *websites*, incluindo nossa aplicação, uma instância t2.micro é o suficiente, permitindo que a maioria das pessoas possa se beneficiar do plano gratuito. No entanto, só é possível usufruir desse plano por um ano a partir da criação da conta AWS, logo calculamos os gastos como se não estivéssemos no plano gratuito, para definir a viabilidade da arquitetura a longo prazo. O valor por hora de utilização de uma instância t2.micro localizada na região da Virgínia do Norte (us-east-1) é de $0,0116, que em um mês de uso totalizaria $8,35, ou aproximadamente R$32,00.

Embora seja economicamente aceitável continuar utilizando a instância t2.micro, o ideal é passar a utilizar uma instância t2.nano após o fim do período de plano gratuito. Para fazer isso basta criar uma AMI (Amazon Machine Image) da instância t2.micro já configurada e em seguida criar a nova instância t2.nano a partir da AMI. Nesse caso o valor de utilização por hora na mesma região é de $0,0058, o que resulta em um valor de $4,17 ao mês, que traduzindo para reais é o equivalente a aproximadamente R$16,15, praticamente metade do valor da instância t2.micro.

6.2.2 Custos da Arquitetura Serverless

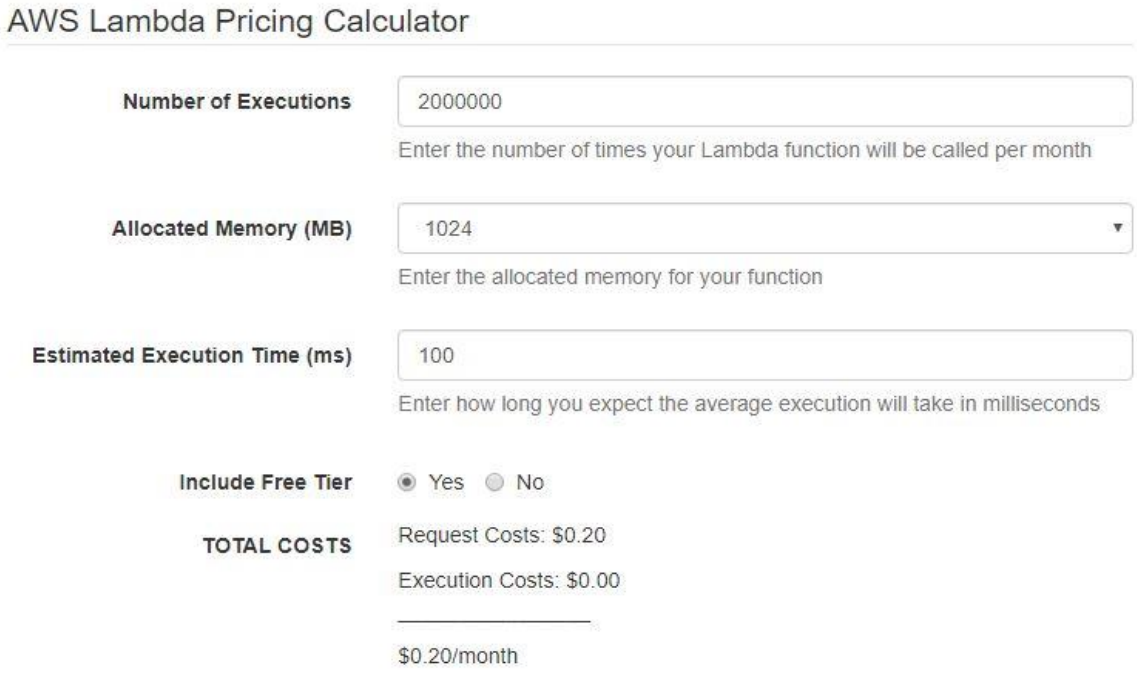
O principal elemento da arquitetura *serverless* é a função Lambda, que têm o mesmo propósito da instância EC2 para a arquitetura cliente-servidor, pois é responsável por responder as requisições. Seus valores de utilização variam de acordo com a quantidade de memória destinada à função e por quanto tempo ela é executada. As funções Lambda também contam com o nível gratuito, que fornece, por mês, 1 milhão de solicitações gratuitas e 400.000 GB (gigabytes) por segundo de tempo de computação. No entanto, diferente das outras funcionalidades, o nível

gratuito da AWS Lambda permanece, mesmo após os 12 primeiros meses da conta, o que ajuda a baratear ainda mais os custos da arquitetura *serverless*.

Optamos por utilizar 1024 MB de memória em nossa função em vez dos 128 MB iniciais, já que essa medida ajuda a diminuir os *cold starts*, em troca de um aumento insignificante no custo mensal da arquitetura. Em seguida observamos os registros de execução da função Lambda no Cloudwatch, para descobrir quanto tempo em média levava a execução da função. Tivemos uma boa surpresa ao perceber que, exceto as execuções com *cold start*, que representavam esmagadora minoria, todas as execuções da função foram feitas em 100 ms (milissegundos), o menor tempo possível.

Utilizando a calculadora de custo da AWS verificamos que, com as configurações definidas para nossa aplicação, o primeiro 1 milhão de solicitações não teria custo relacionado por conta do plano gratuito. Com 2 milhões de solicitações, utilizando as configurações da nossa função, o custo seria de $0,20 em um mês, ou aproximadamente R$0,80, como pode ser visto na Figura 13.

Figura 13 - Custo de execução de 2 milhões de funções Lambda



Fonte: Pablo Bittencourt Pereira Gobira (2019)

Tecnicamente, pode-se dizer que o custo de execução da função Lambda é nulo, pois mesmo em um cenário onde a aplicação recebe muitos acessos num mês, o valor a pagar ainda seria nulo ou muito baixo para ser levado em conta. Isso mostra uma grande vantagem econômica em relação à arquitetura cliente-servidor.

No entanto, houve um custo extra pelo qual não esperávamos ao início desse trabalho, resultante da instância NAT, que foi necessária para garantir conexão de saída à internet para a função Lambda. Em termos de custo, a instância NAT é igual a uma instância EC2 normal, cujo preço varia de acordo com o tempo de utilização e o tipo da instância. Dessa forma é possível selecionar a instância mais barata disponível (t2.nano), fazendo com que a arquitetura *serverless* tenha o mesmo custo mensal da arquitetura cliente-servidor.

Não é obrigatório garantir acesso à internet a função Lambda para que a aplicação servida por ela funcione. Mas caso a aplicação possua algum tipo de funcionalidade que demanda esse tipo de acesso, como o *newsletter* em nosso caso, será necessário utilizar um NAT Gateway ou uma instância NAT. Caso não exista a necessidade de utilizar essas ferramentas ou se sua utilização fosse gratuita, a arquitetura serverless não incorreria em nenhum custo. Como as ferramentas não são gratuitas e precisamos utiliza-las, ambas as arquiteturas acabaram resultando em gastos iguais, de aproximadamente R$16,00 por mês, excluindo os custos do S3 e RDS.

6.3 COMPATIBILIDADE COM RECURSOS EXTERNOS

Nessa seção falaremos sobre a compatibilidade com recursos externos de cada arquitetura, como extensões e bibliotecas que podemos integrar à nossa aplicação. Na arquitetura cliente-servidor não existem barreiras de compatibilidade, afinal de contas, por padrão, os recursos hoje existentes foram desenvolvidos com o intuito de serem executados nesse tipo de arquitetura. Já a arquitetura *serverless* possui alguns empecilhos relacionados ao uso desse tipo de recurso.

O Bref, *plugin* utilizado para automatizar boa parte da configuração e lançamento da arquitetura *serverless*, disponibiliza, por padrão, uma série de importantes extensões PHP que podem ser utilizadas pela aplicação. No entanto, embora as extensões disponibilizadas sejam muitas, várias outras ficam de fora, inviabilizando a utilização do *plugin* para algumas aplicações. Deve ser levado em conta o fato de que o Bref foi criado recentemente, logo é possível que no futuro ele passe a integrar mais extensões. De qualquer forma, a arquitetura cliente-servidor apresenta uma vantagem no quesito compatibilidade.

6.4 GERENCIAMENTO

O último fator de análise desse trabalho está relacionado à demanda e complexidade do gerenciamento de cada arquitetura. Para desenvolver a primeira arquitetura (cliente-servidor), foi necessário contratar e configurar uma instância com os programas necessários para servir a aplicação. Da mesma forma, a arquitetura serverless necessitou de uma série de configurações para funcionar corretamente. De fato as configurações levaram mais tempo para serem concluídas na arquitetura *serverless*, no entanto, acreditamos que isso esteja relacionado à baixa quantidade de conteúdo disponível na internet sobre o desenvolvimento da nova tecnologia e a falta de familiaridade com a mesma. Por outro lado, é muito mais fácil encontrar informação referente à configuração de servidores, através de uma vasta quantidade de guias e postagens de fóruns na internet.

A diferença na necessidade de gerenciamento entre as duas arquiteturas passa a ser mais relevante quando se passa a levar em conta o escalonamento das arquiteturas. Para escalar uma aplicação na arquitetura cliente-servidor, é necessário aumentar a capacidade da instância utilizada como servidor ou configurar um balanceador de carga (*load balancer*) em conjunto com alarmes do Cloudwatch para alcançar o objetivo esperado. Enquanto isso, na arquitetura serverless, o escalonamento da aplicação fica por conta da execução da própria função. Cada requisição vai ativar a função individualmente, independente das outras, de forma que um acumulo de requisições não vá causar uma pane no sistema, coisa que pode acontecer com um servidor.

Embora não tenha sido mencionado anteriormente, a função Lambda possui limitações. São permitidas somente 1000 execuções de função simultaneamente em cada região. As execuções excedentes podem tanto ser rejeitadas quanto colocadas na fila para serem executadas posteriormente. No entanto essa limitação não é preocupante no caso da nossa aplicação por dois motivos. O primeiro motivo é que o tempo médio de execução de cada função, como foi visto no capítulo 6.2.2, gira em torno de 100 milissegundos na maioria dos casos. Isso faz com que mais funções possam ser executadas em menor tempo, o que torna mais difícil alcançar o limite de 1000 execuções simultâneas. Caso o limite tenha sido alcançado, esse fator também permite que as requisições excedentes passem menos tempo em estado de espera antes de serem respondidas. O segundo motivo está relacionado ao gargalo de acessos simultâneos ao banco de dados, que permite o acesso de apenas 66 usuários ao mesmo tempo. Logo seria muito difícil atingir o limite de execução das funções Lambda antes de atingir o limite de acesso do RDS, o que torna o gargalo do banco de dados muito mais preocupante.

Levando em conta os pontos desse capítulo, pudemos concluir que a arquitetura serverless leva vantagem no quesito gerenciamento, especialmente por conta da escalabilidade que ela naturalmente proporciona à aplicação.

7. CONSIDERAÇÕES FINAIS

A partir dos testes feitos percebemos que a arquitetura sem servidor teve resultados semelhantes à arquitetura cliente-servidor. Em questão de performance, os servidores convencionais levam vantagem por conta dos *cold starts* existentes na tecnologia serverless, no entanto, como já foi dito no capítulo dedicado aos testes de performance, existem medidas que podem ser tomadas para reduzir a ocorrência e duração destes eventos. É importante destacar que o excesso de requisições pode gerar falhas nas duas arquiteturas por conta do baixo limite de acessos simultâneos ao banco de dados. Esse limite pode se tornar um gargalo perigoso em algumas aplicações e a única solução para esse problema é aumentar a capacidade da instância do banco, o que resulta em um aumento nos custos mensais da infraestrutura.

Quanto aos custos, esperávamos que a arquitetura sem servidor apresentasse uma grande vantagem em comparação a sua rival. No entanto, devido a necessidade de uma instância NAT para permitir acesso de saída a internet para a função Lambda, ambas tiveram custos exatamente iguais. Se a aplicação desenvolvida nesse trabalho não apresentasse nenhuma funcionalidade que se comunica com um serviço externo a ela (como o serviço de *newsletter*), não seria necessário utilizar a instância NAT e assim o custo da arquitetura sem servidor seria muito mais vantajoso que a da cliente-servidor. Em termos de compatibilidade a arquitetura cliente-servidor possui clara vantagem, no entanto, o oposto ocorre no quesito gerenciamento.

Analisando todos os fatores em conjunto podemos perceber que ambas tecnologias são viáveis, no entanto, a tecnologia *serverless* não apresentou um diferencial relevante o suficiente para que possa substituir o servidor convencional. Por esse motivo, hoje ainda é muito mais comum que os desenvolvedores utilizem a solução padrão na hora de hospedar sua aplicação. No entanto, a cada ano a tecnologia sem servidor vem se desenvolvendo e aprimorando suas funcionalidades, portanto, nossa previsão é de que no futuro ela se torne a solução de hospedagem mais utilizada.

Na Figura 14 é possível ver em quais fatores de avaliação cada arquitetura se saiu melhor. O simbolo "✓" demonstra qual arquitetura se saiu melhor em um determinado fator avaliativo. O símbolo "-" denota que arquiteturas se igualaram.

Figura 14 – Tabela de resultados

Arquiteturas / Fatores Avaliativos	Cliente-Servidor	Serverless
Performance	✓	
Custo	—	—
Compatibilidade com Recursos Externos	✓	
Gerenciamento		✓

Fonte: Pablo Bittencourt Pereira Gobira (2019)

8. TRABALHOS FUTUROS

Hoje existem algumas plataformas de desenvolvimento em nuvem além da Amazon Web Services, como por exemplo a Microsoft Azure e Google Cloud. Em trabalhos futuros sugerimos o estudo de algumas dessas plataformas, para que possa ser feita uma análise sobre as vantagens e desvantagens de cada uma delas. Esse seria um estudo relevante, pois a arquitetura *serverless* pode se tornar ainda mais viável ao ser construída através de outra plataforma.

9. REFERÊNCIAS BIBLIOGRÁFICAS

HENDRICKSON, S. et al. ***Serverless Computation with OpenLambda***. In 8th USENIX Workshop on Hot Topics in Cloud Computing, HotCloud 2016, Denver, CO, USA. 2016. https://www.usenix.org/conference/hotcloud16/workshop-program/presentation/hendrickson.

KRATZKE, N. ***A Brief History of Cloud Application Architectures***. Appl. Sci., vol. 8, no. 8, p. 1368, 2018.

NETO, P. ***Demystifying cloud computing***. *Proceedings of the 6th Doctoral Symposium on Informatics Engineering*. DSIE'11. p. 2971–2978. Faculdade de Engenharia da Universidade do Porto, 2011.

KAUFMAN, L. M. ***Data Security in the World of Cloud Computing***, IEEE Secur Priv. 7. p. 61–64, 2009.

MELL, P.; GRANCE, T. ***The NIST Definition of Cloud Computing***. Gaithersburg: National Institute of Standards and Technology. p. 7, 2011.

VILLAMIZAR, M. et al. ***Infrastructure Cost Comparison of Running Web Applications in the Cloud Using AWS Lambda and Monolithic and Microservice Architectures***. 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing. p. 179–182, 2016

BALDINI, L. et al. ***Serverless computing: Current trends and open problems***. arXiv preprint arXiv:1706.03178. 2017.

AWS Lambda: Guia do Desenvolverdor. Disponível em: <docs.aws.amazon.com/pt_br/lambda/latest/dg/lambda-dg.pdf>. Acesso em: 01 abr. 2019.

BENNET, S. ***Hosting a Laravel Application on AWS Lambda (Full Guide)***. Disponível em: <medium.com/artisanhost/hosting-a-laravel-application-on-aws-lamdba-90b7133c8578>. Acesso em: 08 jan. 2019.

PÉREZ, A. et al. ***Serverless computing for container-based architectures***. *Future Generation Computer Systems*. vol. 83, p. 50-59, 2018.

CHACZKO, Z. Et al. ***Availability and Load Balancing In Cloud Computing***. In Proc. ICCSM. p.134-140, 2011.

OSTERMANN, S. et al. ***A Performance Analysis of EC2 Cloud Computing Services for Scientific Computing***. In Cloudcomp, 2009.

CURINO, C. et al. ***Relational Cloud***: *A Database-as-a-Service for the Cloud*. 5th Biennial Conference on Innovative Data Systems Research, CIDR 2011, Asilomar, California. 2011.

HACIGUMUS, H.; IYER, B.; MEHROTRA, S. ***Providing database as a service***. In Proc. of the Int'l Conf. on Data Engineering. San Jose, California. 2002.

WANG, C.; WANG, Q.; REN, K.; LOU, W. ***Privacy-preserving public auditing for data storage security in cloud computing***. in InfoCom2010, IEEE, 2010.

IAMNITCHI, M. P. A.; RIPEANU, M.; GARFINKEL, S. ***Amazon S3 for science grids***: a viable solution? in DADC '08: Proceedings of the 2008 international workshop on Data-aware distributed computing. ACM. p. 55–64, 2008.

BRANTNER, M. et al. ***Building a database on S3***. In Proceedings of the ACM SIGMOD International Conference on Management of Data. p. 251–264, 2008.

SELFA, D. M.; CARILLO, M.; BOONE, M. D. R. ***A Database and Web Application Based on MVC Architecture***. 16th International Conference on Electronics, Communications and Computers, IEEE Computer Society. p. 48-53, 2006.

YU, H. R. ***Design and implementation of web based on Laravel framework***. Atl. Press, no. Iccet 2014, p. 301-304. 2015.

NAPOLI, M. **Serverless Laravel**. 25/05/2018. Disponível em: <https://mnapoli.fr/serverless-laravel/>. Acesso em: 02 abr. 2019. (Figura 1)

RAUPP, F. M; BEUREN, I. M. **Metodologia da pesquisa aplicável às Ciências Sociais**. In: BEUREN, Ilse Maria (Org.). Como elaborar trabalhos monográficos em São Paulo: Atlas, p. 76-97, 2009.

BOLZAN, W.; GIRAFFA, L. M. M. **Estudo comparativo sobre Sistemas Tutores Inteligentes Multiagentes Web**. Thecnical Report Series. Porto Alegre, n.024, 2002.

MARON, C. A. F.; GRIEBLER, D.; SCHEPKE, C. **Comparacão das Ferramentas OpenNebula e OpenStack em Nuvem Composta de Estacões de Trabalho**. In 14ª Escola Regional de Alto Desempenho do Estado do Rio Grande do Sul - ERAD/RS, Alegrete, RS, Brazil. Sociedade Brasileira de Computacão – SBC. p. 173-176, 2014.

GIL, A. C. Métodos e Técnicas de Pesquisa Social. São Paulo: Atlas, 2007. In: GERHARDT, T. E.; SILVEIRA, D. T. **Métodos de Pesquisa**. Série Educação a Distância. Coordenado pela Universidade Aberta do Brasil – UAB/UFRGS e pelo Curso de Graduação Tecnológica – Planejamento e Gestão para o Desenvolvimento Rural da SEAD/UFRGS. Porto Alegre. 2009.

FOSTER, I.; ZHAO, Yong.; RAICU, I.; LU, S. ***Cloud Computing and Grid Computing 360-Degree Compared***. in: Grid Computing Environments Workshop, 2008, GCE'08. p. 1-10, 2008.

SRINIVASAN, S. S.; ANDERSON, R.; PONNAVOLU, K. ***Customer Loyalty in E-Commerce***: *An Exploration of its Antecedents and Consequences*. Journal of Retailing (78:1). p. 41-50, 2002.

SCHAFER, J. B.; KONSTAN, J. A.; RIEDL, J. ***E-commerce recommendation applications***. Data Mining and Knowledge Discovery, 5(1/2). p. 115-153, 2001.

Rocha, H. V.; BARANAUSKAS, M. C. C. **Design e Avaliação de Interfaces Humano-Computador**. Campinas: Nied-Unicamp. p. 37, 2003.

PETERSON, R. A.; BALASUBRAMANIAN, S. BRONNENBERG, B. J. J. A. M. ***Exploring the implications of the Internet for consumer marketing***. J. Acad. Marketing Sci. 25, p. 329–346, 1997.

CERUTTI, D. M. L. **Ensino de IHC**: Desconstruindo interfaces em sala de aula. WEIHC – Workshop sobre Ensino de IHC. Belo Horizonte, MG, 2010.

VARIA, J.; MATHEW, S. ***Overview of Amazon Web Services***. Disponível em: <http://cabibbo.dia.uniroma3.it/asw-2014-2015/altrui/AWS_Overview.pdf>. Acesso em: 12 abr. 2019.

ARANTES, J. A. **Modelo Analítico para Avaliar Plataformas Clientes/Servidor e Agentes Móveis Aplicado à Gerência de Redes**. Florianópolis, 2001. Dissertação (Mestrado em Ciência da Computação) – Programa de Pós-Graduação em Ciência da Computação, Universidade Federa de Santa Catarina, 2001.

NEDELCU, C. ***Nginx HTTP Server***. Birmingham, UK: Packt Publishing Ltd., 2013.

CONVERSE, T.; PARK, J.; MORGAN, C. ***PHP5 and MySQL Bible***. Indianapolis, Indiana: Wiley Publishing, 2004.

MOUSSA, M. ***AWS Lambda Custom Runtime for PHP: A Practical Example***. Disponível em: <https://aws.amazon.com/pt/blogs/apn/aws-lambda-custom-runtime-for-php-a-practical-example/>. Acesso em: 12 abr. 2019.

AWS Lambda Developer Guide. ***Configuring a Lambda Function to Access Resources in an Amazon VPC***. Disponível em: <https://docs.aws.amazon.com/lambda/latest/dg/vpc.html>. Acesso em: 26 abr. 2019.

ERINLE. B. ***Performance Testing with JMeter***. Second Edition. Packt Publishing, 2015.

BYRRO, R. ***Can We Solve Serverless Cold Starts?*** Disponível em: <https://dashbird.io/blog/can-we-solve-serverless-cold-starts/>. Acesso em: 12 jun. 2019.

AWS Lambda Princing Calculator. Disponível em: <https://s3.amazonaws.com/lambda-tools/pricing-calculator.html>. Acesso em: 14 jun. 2019.