UNIVERSIDADE FEDERAL DO CEARÁ

CAMPUS QUIXADÁ

CURSO DE GRADUAÇÃO EM REDES DE COMPUTADORES

HELTER LUCIO RIBEIRO PITANGA

REPOSITÓRIO VIRTUAL UTILIZANDO *SERVERLESS COMPUTING*

QUIXADÁ

2022

HELTER LUCIO RIBEIRO PITANGA

REPOSITÓRIO VIRTUAL UTILIZANDO *SERVERLESS COMPUTING*

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Redes de Computadores do Campus Quixadá da Universidade Federal do Ceará, como requisito parcial à obtenção do grau de tecnólogo em Redes de Computadores.

Orientador: Prof. Dr. João Marcelo Uchôa de Alencar

QUIXADÁ

2022

Dados Internacionais de Catalogação na Publicação
Universidade Federal do Ceará
Biblioteca Universitária
Gerada automaticamente pelo módulo Catalog, mediante os dados fornecidos pelo(a) autor(a)

P758r Pitanga, Helter Lucio Ribeiro.
Repositório virtual utilizando serverless computing / Helter Lucio Ribeiro Pitanga. – 2022.
67 f. : il. color.

Trabalho de Conclusão de Curso (graduação) – Universidade Federal do Ceará, Campus de Quixadá, Curso de Redes de Computadores, Quixadá, 2022.
Orientação: Prof. Dr. João Marcelo Uchôa de Alencar.

1. Computação em nuvem. 2. Computação sem servidor. 3. Função como serviço. I. Título.
CDD 004.6

HELTER LUCIO RIBEIRO PITANGA

REPOSITÓRIO VIRTUAL UTILIZANDO *SERVERLESS COMPUTING*

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Redes de Computadores do Campus Quixadá da Universidade Federal do Ceará, como requisito parcial à obtenção do grau de tecnólogo em Redes de Computadores.

Aprovada em: ____/_____/_____.

BANCA EXAMINADORA

Prof. Dr. João Marcelo Uchôa de Alencar (Orientador)
Universidade Federal do Ceará (UFC)

Prof. Dr. Wagner Guimarães Al-Alam
Universidade Federal do Ceará (UFC)

Prof. Dr. Jefferson de Carvalho Silva
Universidade Federal do Ceará (UFC)

Dedico este trabalho aos meus pais e irmãos, por sempre estarem ao meu lado nessa jornada e que me deram força nos momentos difíceis e por fazer meu sonho torna-se realidade. Essa vitória é de vocês.

AGRADECIMENTOS

Agradeço a Deus, primeiramente, que me deu força para concluir esta etapa da minha vida.

Aos professores, que com paciência e dedicação, ensinaram-me não somente conteúdos da grade curricular, mas também o sentido da amizade, respeito e trabalho em equipe.

Em especial, ao professor e orientador Dr. João Marcelo Uchôa de Alencar pelo apoio, paciência e compromisso que sempre teve comigo na elaboração deste trabalho, serei sempre grato.

A minha família, meu pai Raimundo Pitanga, minha mãe Francisca Lucia Ribeiro Pitanga e meus irmãos Heitor, Hudson, Hedson e Marcelo, por todo apoio e carinho ao longo dessa jornada.

Aos amigos e colegas que fizeram parte da caminhada ao longo do curso. Em especial, ao meu amigo José Wanderley Gomes da Silva, pelo apoio e ajuda mútua ao longo do curso, que fez parte da minha formação e que vai continuar ao longo da vida.

A esta universidade aos docentes, diretores, coordenadores e administração que proporcionaram o melhor dos ambientes para que esse trabalho fosse realizado.

“Aqui no entanto nós não olhamos para trás por muito tempo. Nós continuamos seguindo em frente, abrindo novas portas e fazendo coisas novas, porque somos curiosos... e a curiosidade continua nos conduzindo por novos caminhos. Siga em frente. ”

(Walt Disney)

RESUMO

A computação em nuvem é um modelo para permitir acesso sob demanda sobre recursos de sistema computacional configuráveis (por exemplo, redes, servidores, armazenamento, aplicativos e serviços). Um serviço que é convenientemente usado é a *Infrastructure as a Service* (IaaS), que permite aos usuários escalar servidores em nuvem de forma prática e rápida, mas, nesse serviço fica a encargo dos usuários a configuração e o controle das aplicações, dados, sistema operacional e o ambiente de execução. Pensando em abstrair toda a configuração do ambiente em torno do usuário, o presente trabalho utiliza-se da tecnologia *Serverless Computing* (SC), para a construção de um repositório virtual de documentos, utilizando os serviços da nuvem pública *Amazon Web Services* (AWS). Após a implantação da aplicação, o usuário fica apenas responsável por prover o *Back-end as a Service* (BaaS), nisso fica por parte do provedor da nuvem controlar e manter todos os serviços criados, podendo assim escalonar quando for necessário. Por fim, comparamos a implementação entre as arquiteturas monolítica e sem servidor em nuvem, relatando uma estimativa de custos de ambas.

Palavras-chave: Computação em nuvem. Computação sem servidor. Função como serviço

ABSTRACT

Cloud computing is a model for enabling on-demand access to configurable computing system resources (e.g. networks, servers, storage, applications, and services). One service that is conveniently used is Infrastructure as a Service (IaaS), which allows users to scale servers in the cloud in a practical and fast way, but in this service, users are responsible for configuring and controlling applications, data, system operating, and execution environment. Thinking about abstracting all the configuration of the environment around the user, the present work uses Serverless Computing (SC) technology to the construction of a virtual repository, using the services of the Amazon Web public cloud Services (AWS), the deployment of the services used in the work is done at the time of deployment of the cloud service. After deployment, the user is only responsible for providing the Back-end as a Service (BaaS) for the application, it is up to the cloud provider to control and keep all the services created, thus being able to scale when necessary. Lastly, compare the implementation between monolithic and serverless cloud architectures, reporting a cost estimate of both.

Keywords: Cloud computing. Serverless computing. Function as a service

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

AIS	*Active Ionospheric Sounding*
API	*Application Programming Interface*
ARPANET	*Advanced Research Projects Agency Network*
AWS	*Amazon Web Services*
BaaS	*Back-end as a Service*
CDN	*Content Delivery Network*
CLI	*Comand Line Interface*
DaaS	*Data as a Service*
DBMS	*Data Base Management System*
EC2	*Elastic Compute Cloud*
ELB	*Elastic Load Balancing*
FaaS	*Functions as a Service*
HTTP	*Hypertext Transfer Protocol*
IaaS	*Infrastructure as a Service*
IAM	*Identity and Access Management*
MAU	*Monthly Active Users*
OTP	*One Time Password*
PaaS	*Platform as a Service*
PDF	*Portable Document Format*
RDS	*Relational Database Service*
REST	*Representational State Transfer*
S3	*Simple Storage Service*
SaaS	*Software as a Service*
SC	*Serverless Computing*
SDK	*Software Development Kit*
SMS	*Short Message Service*
SNS	*Simple Notification Service*
TCO	*Total Cost of Ownership*
TI	*Tecnologia da Informação*
WEB	*World Wide Web*

SUMÁRIO

1 INTRODUÇÃO

Com o avanço da sociedade humana, serviços básicos são fornecidos para que todos possam obter acesso facilmente. Por exemplo, serviços de utilidade pública, como água, eletricidade e telefonia são considerados necessários para suprir a rotina da vida cotidiana. Esses serviços utilitários são utilizados frequentemente e precisam estar disponíveis sempre que o consumidor exigir. Os usuários são capazes de pagar aos provedores de serviços com base no uso desses serviços públicos (BUYYA *et al.*, 2008).

Em 1969, Leonard Kleinrock (KLEINROCK, 2005), um dos principais cientistas do Projeto *Advanced Research Projects Agency Network* (ARPANET), que semeou a Internet, disse: “A partir de agora, as redes de computadores ainda estão na infância, mas à medida que crescem e se sofisticam, provavelmente veremos a expansão de ’utilitários de computadores’ que, como a atual eletricidade e serviços telefônicos, atenderá residências individuais e escritórios em todo o país ". Essa visão dos utilitários de computação baseada no modelo de fornecimento de serviços antecipa a transformação maciça de toda a indústria de computação no século 21, em que serviços de computação estarão prontamente disponíveis sob demanda, como outros serviços utilitários disponíveis no sociedade.(BUYYA *et al.*, 2008)

Da mesma forma que os serviços utilitários tradicionais, usuários de serviços de computação precisam pagar aos fornecedores somente quando acessar serviços de computação. Além disso, os consumidores não precisam mais investir pesadamente ou enfrentar dificuldades na construção e manutenção de infraestrutura de *Tecnologia da Informação* (TI) complexa (BUYYA *et al.*, 2008).

A computação em nuvem pode ser definida como um estilo de computação, no qual recursos dinamicamente escaláveis e muitas vezes virtualizados são fornecidos como serviços através da Internet. A computação em nuvem se tornou uma tendência tecnológica significativa, e muitos os especialistas esperam que a computação em nuvem reformule os processos de tecnologia da informação (TI) e o mercado de TI (FURHT, 2010).

Com a tecnologia de computação em nuvem, os usuários usam uma variedade de dispositivos, incluindo computadores pessoais, *laptops* e *smartphones* para acessar programas, armazenamento e plataformas de desenvolvimento de aplicativos pela Internet, através de serviços oferecidos por provedores de computação em nuvem (FURHT, 2010).

Em 2009, Armbrust *et al.* (2009) identificou seis vantagens em potencial da computação em nuvem, no qual este trabalho tem ênfase nas duas últimas:

- Simplificação da operação e aumento da utilização via virtualização de recursos.
- Maior utilização de *hardware*, multiplexando cargas de trabalho de diferentes organizações.

Nos últimos anos, tais vantagens foram amplamente percebidas, mas os usuários da nuvem continuam tendo uma carga de operações complexas. A computação em nuvem aliviou os usuários do gerenciamento da infraestrutura física, mas em contrapartida os deixou com um aumento de recursos virtuais para gerenciar.

O reconhecimento dessas necessidades levou a criação de uma nova opção de serviço da AWS em 2015, chamada serviço AWS Lambda. O Lambda oferece funções na nuvem e chamou a atenção para computação sem servidor. Embora a computação sem servidor seja indiscutivelmente um oximoro, uma vez que ainda está usando servidores para calcular (JONAS *et al.*, 2019), a gestão dos servidores virtuais é abstraída.

O nome computação sem servidor ou *serverless computing* indica que o usuário da nuvem pode fornecer o código e deixar todas as tarefas de provisionamento e administração do servidor para o provedor da nuvem. O código é representado em funções na nuvem chamadas de *Functions as a Service* (FaaS), que representam o núcleo da computação sem servidor. As plataformas em nuvem também fornecem estruturas especializadas sem servidor que atendem a requisitos de aplicativos específicos, como o *Back End as a Service* BaaS (AMAZON, 2020a). Simplificando, computação sem servidor é a união dos paradigmas FaaS e BaaS. (JONAS *et al.*, 2019)

Neste contexto, considerando as possibilidades observadas, este trabalho tem como objetivo implementar um *site web* desenvolvido no paradigma *Serverless Computing* ou computação sem servidor. Utilizando Serviços do provedor em nuvem AWS (AMAZON, 2008). A aplicação é um repositório de documentos no formato PDF armazenados na nuvem, com total gerenciamento da infraestrutura física e virtual por parte do provedor da nuvem. Com o desenvolvimento da aplicação, os principais conceitos de *serverless computing* são abordados, servindo como base para uma comparação com arquiteturas monolíticas tradicionais.

1.1 Objetivos

A seguir, serão apresentados os objetivos deste trabalho.

1.1.1 Objetivo Geral

Criação de um repositório virtual de documentos em nuvem com interface *web* utilizando o paradigma *Serverless Computing*, visando demonstrar o potencial dessa nova tecnologia quanto ao custo, escalabilidade e disponibilidade, além de realizar uma comparação de custos entre as arquiteturas sem servidor e monolítica.

1.1.2 Objetivos específicos

1. Implementar autenticação na aplicação.
2. Desenvolver funções no AWS Lambda para *upload*, *download* e visualização dos metadados dos documentos via navegador.
3. Armazenar os metadados referentes aos documentos em uma base de dados.
4. Mostrar benefícios da tecnologia como disponibilidade, escalabilidade e flexibilidade.
5. Comparar arquitetura sem servidor do trabalho proposto com arquitetura monolítica.

Este trabalho está organizado da seguinte forma: no capítulo 2, são apresentados os termos e conceitos necessários para entender o trabalho. No capítulo 3, são apresentados os trabalhos relacionados com o presente trabalho. O capítulo 4 é apresentado como a aplicação foi desenvolvida e quais serviços foram utilizados. O capítulo 5 mostra a comparação do trabalho proposto com a arquitetura monolítica. Por fim, capítulo 6 apresenta as conclusões finais do trabalho.

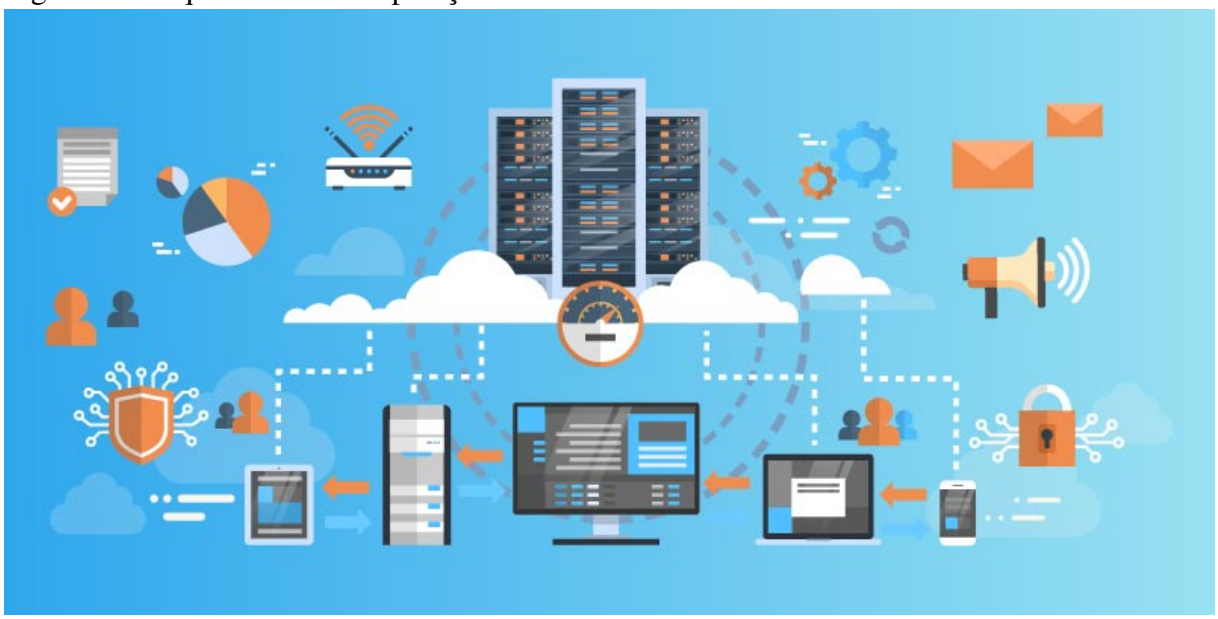
2 FUNDAMENTAÇÃO TEÓRICA

Neste Capítulo, será apresentada uma visão geral sobre os conceitos abordados neste trabalho para um melhor entendimento da problemática tratada.

2.1 Computação em Nuvem

Segundo Mell *et al.* (2011) a computação em nuvem é um modelo para permitir o acesso onipresente, conveniente e sob demanda da rede a um conjunto de recursos de computação configuráveis (por exemplo, redes, servidores, armazenamento, aplicativos e serviços) que pode ser rapidamente provisionado e liberado com o mínimo esforço de gerenciamento ou interação do provedor de serviços. Portanto, é um ambiente com ampla distribuição e várias formas de acesso, como podemos observar na Figura 1.

Figura 1 – Arquitetura da computação em nuvem.



Fonte: (IDEALMARKETING, 2018)

2.1.1 Características Essenciais

O Mell *et al.* (2011) define a computação em nuvem como é conhecida hoje por cinco características. O **atendimento sob demanda** determina que um consumidor pode provisionar unilateralmente recursos de computação, como tempo do servidor e armazenamento de rede, conforme necessário, sem exigir interação com cada provedor de serviços. No **amplo acesso**

à rede, os recursos estão disponíveis na rede e acessados de forma padrão, mecanismos que promovem o uso por plataformas heterogêneas de clientes com pouco poder computacional ou com maior poder computacional (por exemplo, telefones celulares, *tablets, laptops* e estações de trabalho).

O **agrupamento de recursos** permite atender a vários consumidores usando um modelo de multi inquilinos, com diferentes recursos físicos e virtuais dinamicamente atribuídos e reatribuídos de acordo com a demanda do consumidor. Existe uma sensação de localização independente em que o cliente geralmente não tem controle ou conhecimento sobre a exata localização dos recursos fornecidos, mas pode ser capaz de especificar a localização em um nível de abstração (por exemplo, país, estado ou *datacenter*). Exemplos de recursos incluem armazenamento, processamento, memória e largura de banda de rede.

A **elasticidade rápida** garante que os recursos podem ser provisionados e liberados elasticamente, em alguns casos automaticamente, para dimensionar rapidamente para fora e para dentro, proporcional à demanda. Ao consumidor, os recursos disponíveis para provisionamento geralmente parecem ilimitados e ser apropriado em qualquer quantidade e a qualquer momento.

Por último, a **medição do serviço** prevê que os sistemas em nuvem controlem e aperfeiçoem automaticamente o uso de recursos aproveitando uma capacidade de medição em algum nível de abstração apropriado ao tipo de serviço (por exemplo, armazenamento, processamento, largura de banda e contas de usuário ativas). O uso de recursos pode ser monitorado, controlado e relatado, proporcionando transparência tanto para o fornecedor e consumidor do serviço utilizado.

2.1.2 Modelos de Serviços

Segundo Sousa *et al.* (2009) o ambiente de computação em nuvem é composto de três modelos de serviços. Estes modelos são importantes, pois eles definem um padrão arquitetural para soluções de computação em nuvem. Além desses três modelos, foi adicionado um novo modelo de serviço.

2.1.2.1 Software as a Service (SaaS)

O modelo de *Software as a Service* (SaaS) proporciona sistemas de *software* com propósitos específicos que estão disponíveis para os usuários através da *internet*. Os sistemas de *software* são acessíveis a partir de vários dispositivos do usuário por meio de uma interface *thin*

client (cliente magro) como um navegador *web*.

No SaaS, o usuário não administra ou controla a infraestrutura subjacente, incluindo rede, servidores, sistemas operacionais, armazenamento ou mesmo as características individuais da aplicação, exceto configurações específicas. Com isso, os desenvolvedores se concentram em inovação e não na infraestrutura, levando ao desenvolvimento rápido de sistemas de *software*.

2.1.2.2 Platform as a Service (PaaS)

A *Platform as a Service* (PaaS) oferece uma infraestrutura de alto nível de integração para implementar e testar aplicações na nuvem. O usuário não administra ou controla a infraestrutura subjacente, incluindo rede, servidores, sistemas operacionais ou armazenamento, mas tem controle sobre as aplicações implantadas e, possivelmente, as configurações das aplicações hospedadas nesta infraestrutura.

A PaaS fornece um sistema operacional, linguagens de programação e ambientes de desenvolvimento para as aplicações, auxiliando a implementação de sistemas de *software*, já que contém ferramentas de desenvolvimento e colaboração entre desenvolvedores.

2.1.2.3 Infrastructure as a Service (IaaS)

A IaaS é a parte responsável por prover toda a infraestrutura necessária para a PaaS e a SaaS. O principal objetivo da IaaS é tornar mais fácil e acessível o fornecimento de recursos, tais como servidores, rede, armazenamento e outros recursos de computação fundamentais para construir um ambiente sob demanda, que podem incluir sistemas operacionais e aplicativos.

A IaaS possui algumas características, tais como uma interface única para administração da infraestrutura, *Application Programming Interface* (API) para interação com *hosts*, *switches*, balanceadores de carga, roteadores e o suporte para a adição de novos equipamentos de forma simples e transparente. Em geral, o usuário não administra ou controla a infraestrutura da nuvem, mas tem controle sobre os sistemas operacionais, armazenamento e aplicativos implantados, e, eventualmente, seleciona componentes de rede, tais como *firewalls*.

2.1.2.4 Data as a Service (DaaS)

Data as a Service (DaaS) é uma estratégia de gerenciamento de dados e modelo de serviço que se concentra na nuvem para fornecer uma variedade de serviços relacionados a dados, como armazenamento, processamento e análise. O DaaS aproveita o popular paradigma de *Software as a Service* (SaaS), por meio do qual os clientes podem usar aplicativos de *software* baseados em nuvem em vez de implantar servidores dedicados para um conjunto específico de tarefas em um conjunto específico de dados(ACADEMY, 2022).

O DaaS é uma construção arquitetônica, e não uma tecnologia de um único fornecedor. Portanto, oferece uma variedade de fornecer, coletar e processar dados de várias fontes em diferentes formatos. As tecnologias incluídas na categoria DaaS são:

- Soluções de gerenciamento do ciclo de vida da informação.
- Modelagem, qualidade, replicação, transformação de dados.
- Gerenciamento de conteúdo.

2.1.3 Modelos de Implantação

Segundo Ruschel *et al.* (2010) nos modelos de implantação, dependemos das necessidades das aplicações que serão implementadas. A restrição ou abertura de acesso depende do processo de negócios, do tipo de informação e do nível de visão desejado. Percebemos que certas organizações não desejam que todos os usuários possam acessar e utilizar determinados recursos no seu ambiente de computação em nuvem.

Surge assim, a necessidade de ambientes mais restritos, nos quais somente alguns usuários devidamente autorizados possam utilizar os serviços providos. Conforme o Mell *et al.* (2011), os modelos de implantação da computação em nuvem podem ser divididos em: privada, pública, comunitária e híbrida. A seguir, esses tipos de nuvem serão descritos.

2.1.3.1 Nuvem Privada

A infraestrutura de nuvem é utilizada exclusivamente por uma organização. Pode ser geridos pela organização ou de um terceiro e pode existir no local ou remota.

As nuvens privadas são aquelas construídas exclusivamente para um único usuário (uma empresa, por exemplo). Diferentemente de um *data center* privado virtual, a infraestrutura utilizada pertence ao usuário, e, portanto, ele possui total controle sobre como as aplicações são implementadas na nuvem. Uma nuvem privada é, em geral, construída sobre um *data center* privado. (CHIRIGATI, 2009)

Para esse modelo de implantação são empregados políticas de acesso aos serviços. Gerenciamento de redes, configurações dos provedores de serviços e a utilização de tecnologias de autenticação e autorização são as principais características deste modelo. (RUSCHEL *et al.*, 2010)

2.1.3.2 Nuvem Pública

A infraestrutura de nuvem é disponibilizada ao público em geral ou a um grande grupo industrial e é propriedade de uma organização de venda de serviços em nuvem. As nuvens públicas são aquelas executadas por terceiros. As aplicações de diversos usuários ficam misturadas nos sistemas de armazenamento, o que pode parecer ineficiente a princípio. Porém, a implementação de uma nuvem pública considera questões fundamentais, como desempenho e segurança, a existência de outras aplicações sendo executadas na mesma nuvem permanece transparente tanto para os prestadores de serviços como para os usuários.

Para este modelo de implantação as restrições de acessos não podem ser aplicadas. Quanto ao gerenciamento de redes, a aplicação de técnicas de autenticação e autorização também não será possível. Na nuvem pública, a infraestrutura é disponibilizada para o público em geral, sendo acessado por qualquer usuário que conheça a localização do serviço.

2.1.3.3 Nuvem Comunitária

A infraestrutura de nuvem é compartilhada por diversas organizações e suporta uma comunidade específica que compartilha as obrigações (por exemplo, a missão, os requisitos de segurança, política e considerações sobre o acordo). Pode ser administrado por organizações ou de um terceiro e pode existir no local ou remota.

Neste modelo, várias organizações utilizam a mesma nuvem, que poderá ser administrada por uma empresa desta nuvem ou mais, ou até mesmo por uma terceira.

2.1.3.4 Nuvem Híbrida

A infraestrutura de nuvem é uma composição de duas ou mais nuvens (privada, pública ou comunitária) que permanecem entidades únicas, mas são unidas por proprietárias de tecnologia padronizada que permite a portabilidade dos dados e aplicações (por exemplo, balanceamento de carga entre as nuvens).

As nuvens híbridas combinam os modelos das nuvens públicas e privadas. Elas permitem que uma nuvem privada possa ter seus recursos ampliados a partir de uma reserva de recursos em uma nuvem pública. Essa característica possui a vantagem de manter os níveis de serviço mesmo que haja flutuações rápidas na necessidade dos recursos.

A conexão entre as nuvens pública e privada pode ser usada até mesmo em tarefas periódicas que são mais facilmente implementadas nas nuvens públicas, por exemplo. O termo "computação em ondas" (em inglês, *surge computing*) é, em geral, utilizado quando se refere às nuvens híbridas.

2.2 Computação Sem Servidor ou *Serverless Computing*

Computação sem servidor é a arquitetura nativa da nuvem que permite transferir as responsabilidades operacionais ao provedor da nuvem, aumentando a agilidade e a inovação. A arquitetura sem servidor permite criar e executar aplicativos e serviços sem preocupações com servidores. Ela elimina as tarefas de gerenciamento de infraestrutura, como provisionamento de servidores ou de *clusters, patches*, manutenção do sistema operacional e provisionamento de capacidade (AMAZON, 2020a).

Segundo (FLARE, 2020) Computação sem Servidor ou *Serverless Computing* é um modelo de fornecer serviços de *back-end* conforme o uso. Um provedor de computação sem servidor permite que os usuários escrevam e implantem código sem o incômodo de se preocupar com a infraestrutura subjacente.

Um usuário que obtém serviços de *back-end* de um fornecedor sem servidor é cobrado com base em seu uso e não precisa reservar e pagar por uma quantidade fixa de largura de banda ou número de servidores, pois o serviço é dimensionado automaticamente. Observe que, embora sejam chamados sem servidor, servidores físicos ainda são usados, mas os usuários não precisam estar cientes deles (FLARE, 2020).

A arquitetura *serverless* representa um modelo de hospedagem para funções que não necessita de configuração do servidor, ou seja, todas as dependências para que sua aplicação execute já estão instaladas de forma nativa. Essas funções podem ser "disparadas"de duas maneiras: Rotas HTTP assim como em serviços REST comuns ou eventos disparados por outros serviços existentes (FERNANDES, 2019).

2.3 *Functions as a Service* (FaaS)

A FaaS é uma forma de computação sem servidor, na qual o provedor de nuvem gerencia os recursos, o ciclo de vida e a execução orientada a eventos de funções fornecidas pelo usuário (VAN EYK *et al.*, 2018).

Seguindo a liderança da AWS Lambda (AMAZON, 2014), serviços como Microsoft Azure *functions* (MICROSOFT, 2020) e Google Cloud *functions* (GOOGLE, 2020), criaram serviços de computação sem servidor, como também, foram criadas plataformas de código aberto para criação de funções para computação sem servidor como Apache OpenWhisk (OPENWHISK, 2020), Iron.io IronFunctions (IRON.IO, 2020) e OpenLambda (OPENLAMBDA, 2020).

Em uma nuvem que oferece um serviço no FaaS, a lógica do aplicativo é dividida em funções e executado em resposta a eventos. Esses eventos podem ser acionados de fontes externas à plataforma em nuvem, mas também são emitidos entre os serviços da plataforma em nuvem, permitindo que os desenvolvedores componham aplicativos distribuídos compostos por vários serviços em uma nuvem (MCGRATH; Brenner, 2017). Este trabalho utilizá o serviço AWS Lambda para criação de suas funções como serviço. A seguir uma visão geral do serviço AWS lambda.

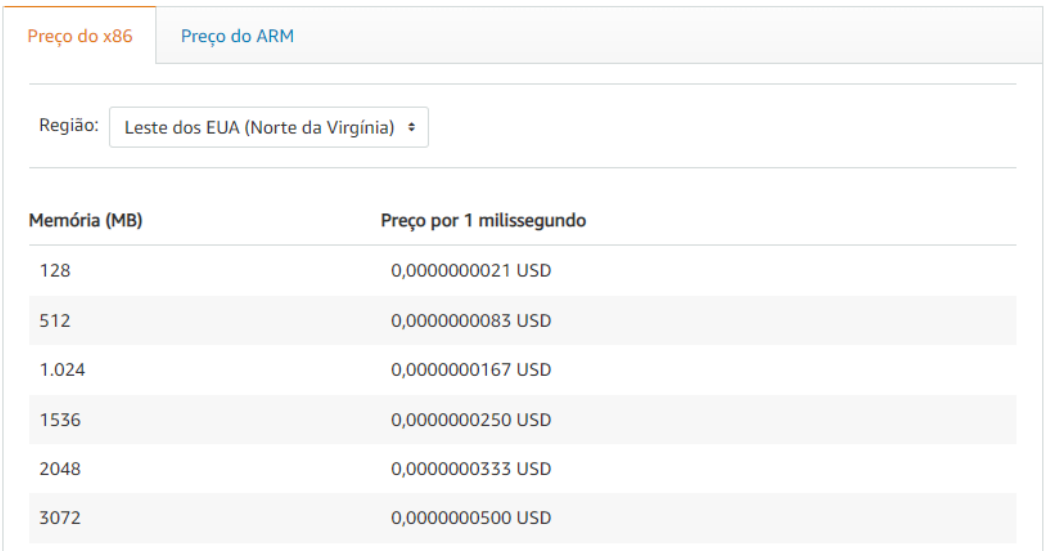
O AWS Lambda é uma estrutura de microsserviço projetada para executar funções Lambda fornecidas pelo usuário em resposta a eventos assíncronos, por exemplo, chegadas de mensagens, *uploads* de arquivos, ou chamadas de API feitas por meio de solicitações HTTP (AMAZON, 2020b). Ao receber um evento, o AWS Lambda invoca um executável que executa em um contêiner Linux com recursos configuráveis de CPUs virtuais de 2,8 GHz, 1.536 MiB de RAM e aproximadamente 500 MB de espaço em disco. O AWS Lambda fornece contêineres adicionais conforme necessário em resposta à demanda (FOULADI *et al.*, 2017).

No AWS Lambda o usuário é cobrado pelo número de solicitações de suas funções e pela duração, o tempo que leva para que seu código seja executado. O Lambda conta uma solicitação cada vez que começa a executar em resposta a uma notificação de evento ou chamada de invocação, incluindo invocações de teste do console (AMAZON, 2014).

A duração é calculada a partir do momento em que seu código começa a ser executado até ele retornar ou encerrar, arredondando para os 1 ms mais próximos. O preço depende da quantidade de memória que você alocar para sua função. No modelo de recursos do AWS Lambda, você seleciona a quantidade de memória que quer para sua função, capacidade de CPU e outros recursos são alocados de forma proporcional (AMAZON, 2014).

Como dito anteriormente, o preço no AWS Lambda é cobrado conforme a quantidade de memória e o número de solicitações da função, como também no tempo de execução da função a cada 1 ms. Na figura 2 é mostrado os valores cobrados no AWS Lambda.

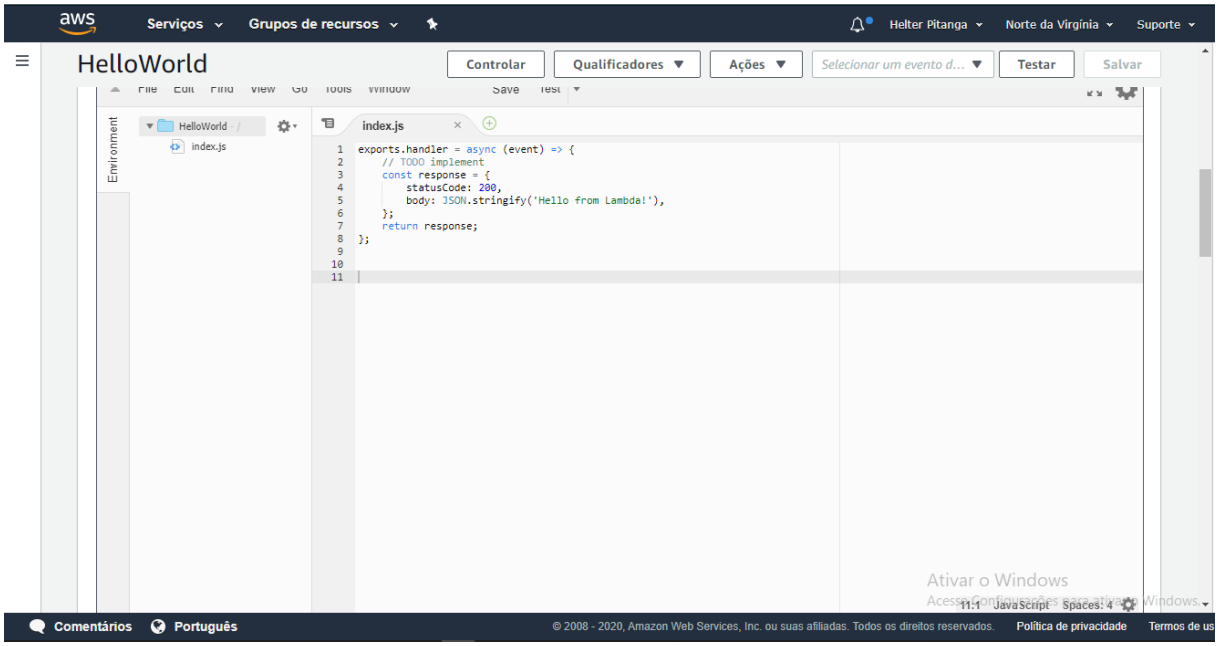
Figura 2 – Definição de preço do AWS Lambda.



Fonte: (AMAZON, 2022a)

Na figura 3 é demonstrado um exemplo simples de uma função feita no serviço AWS Lambda da nuvem Amazon *Web Services.*

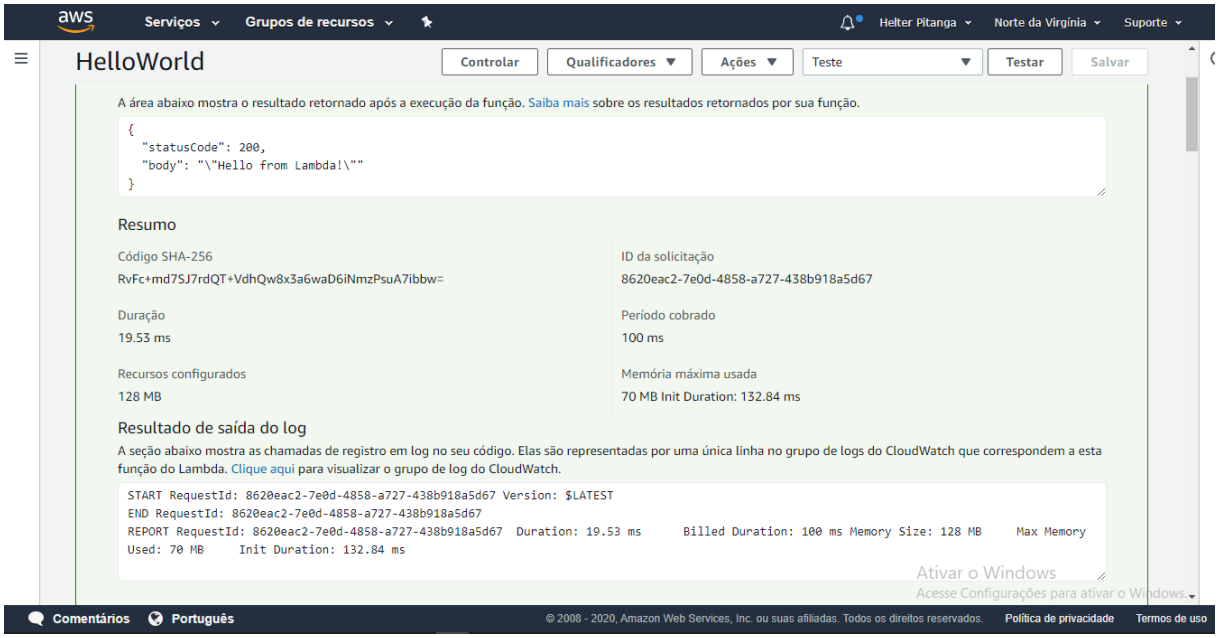
Figura 3 – 'Hello World' em Node.js no console da AWS Lambda.



Fonte: Elaborada pelo próprio autor (2020)

Na figura 4 a saída da função executada no AWS Lambda.

Figura 4 – Resultado da execução 'Hello World' em Node.js no console da AWS Lambda.



Fonte: Elaborada pelo próprio autor (2020)

O AWS Lambda oferece suporte nativamente aos códigos Java, Go, PowerShell, Node.js, C#, Python e Ruby, bem como uma API de tempo de execução que permite usar qualquer linguagem de programação adicional para criar suas funções. (AMAZON, 2014)

2.3.1 Funções de Gatilho

As funções de gatilhos são trechos de códigos que são acionados automaticamente após um evento configurado pelo usuário ser disparado. Esses eventos podem ser combinados com outros serviços da nuvem, como DynamoDB e Amazon S3, ambos serviços oferecidos pela AWS.

Para que uma função lambda seja executada, um evento deve ocorrer. O Lambda não pode acionar outro Lambda diretamente. Para conectar dois Lambdas, é necessário que o primeiro Lambda gere um evento que o segundo Lambda entenda para ser acionado (NOVKOVIC, 2018). Para acionar uma função Lambda, você pode escolher entre várias maneiras diferentes. Nas próximas seções descrevemos as duas maneiras mais utilizadas.

2.3.1.1 Eventos API Gateway

API Gateway é um serviço disponibilizado pela Amazon AWS para criação de APIs HTTPs que podem ser integradas as funções lambda. O uso do API Gateway é uma maneira de acionar o Lambda (AMAZON, 2020b). Esses eventos são considerados eventos clássicos. Simplificando, significa que quando alguém está chamando um API Gateway, ele acionará sua função lambda. Para o Lambda saber que tipo de evento o acionará, primeiro você precisa defini-lo na configuração da função no painel de controle da AWS ou através de arquivos de configuração como o *serverless.yml*, utilizado no *framework Serverless* (NOVKOVIC, 2018).

2.3.1.2 Eventos do Amazon S3

Amazon *Simple Storage Service* (S3) é um serviço de armazenamento de objetos em nuvem disponibilizado pela Amazon AWS (AMAZON, 2020c). Os eventos originados nesse serviço podem ocorrer quando alguém (ou algo) modifica o conteúdo de um *bucket* S3, como é chamado o repositório de arquivos na nuvem. É possível alterar o conteúdo criando, removendo ou atualizando um arquivo. Enquanto você define um evento, é possível especificar que tipo de ação acionará a função lambda, seja criando, removendo ou atualizando um arquivo

(NOVKOVIC, 2018).

2.3.2 Arquiteturas de Aplicações

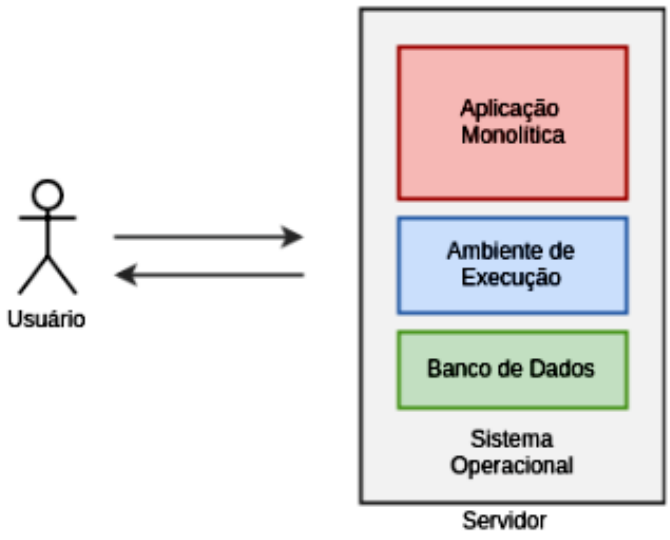
Uma arquitetura compreende-se por um conjunto de módulos agrupados, originando componentes ou pacotes com funcionalidades definidas em um sistema, ressaltando a importância da sua escolha para uma determinada necessidade (WAZLAWICK, 2019). Nesta seção, será explicado brevemente as arquiteturas a serem comparadas nesse trabalho.

2.3.2.1 Arquitetura Monolítica

A arquitetura monolítica é considerada uma forma tradicional de construção de aplicações. Uma aplicação monolítica é construída como uma unidade única e indivisível. Normalmente, essa solução compreende uma interface de usuário do lado do cliente, um aplicativo do lado do servidor e um banco de dados. Ele é unificado e todas as funções são gerenciadas e atendidas em um só lugar.

Normalmente, os aplicativos monolíticos têm uma grande base de código e não possuem modularidade. Se os desenvolvedores desejam atualizar ou alterar algo, eles acessam a mesma base de código. Portanto, eles fazem alterações em toda a pilha de uma vez (ALPHACODES, 2021). Na figura 5 é seguir é mostrado a arquitetura monolítica.

Figura 5 – Arquitetura monolítica



Fonte: (KOLLER, 2022)

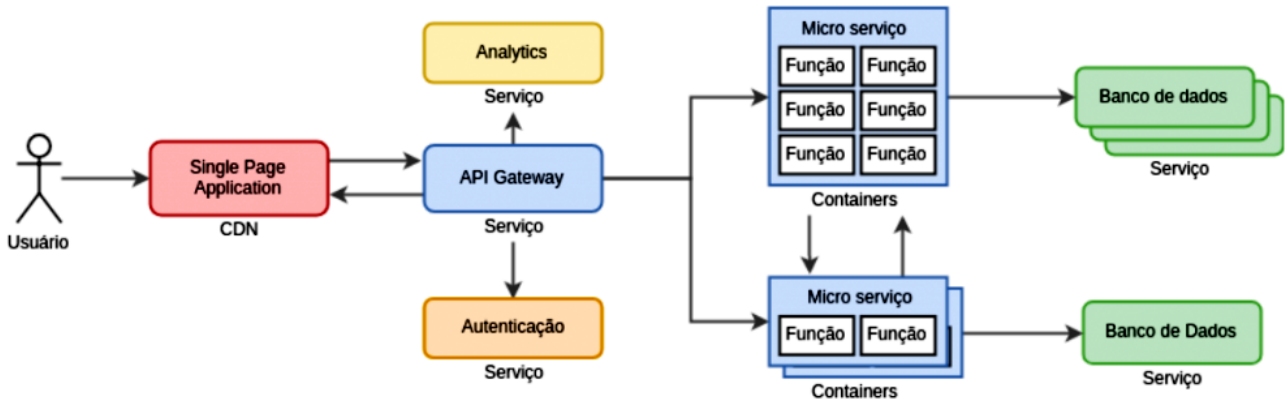
2.3.2.2 Arquitetura Serverless

A arquitetura *serverless* (arquitetura sem servidor) desafia design de *software* e os fundamentos de implementação por alcançar um ótimo nível de desenvolvimento, operação e gerenciamento. Embora herde conceitos elementares da arquitetura de microsserviços, é dotada de padrões arquitetônicos de ponta para atingir o nível mínimo possível de *hardware* ocioso(GUNARATNE, 2022).

Apesar do termo "*serverless computing*" (computação sem servidor) ser um oximoro, ainda ser utilizado servidores, seu significado real é baseado na capacidade de implementar *software* sem ter qualquer envolvimento com a infraestrutura. As plataformas sem servidor automatizam todo o processo de criação, implementação e início de serviços sob demanda. Os usuários só precisam registrar as funções requeridas e suas necessidades de recursos a serem utilizados(GUNARATNE, 2022).

Essas funções podem ser classificadas em dois tipos principais: funções que são acionadas por solicitações de usuários e funções que precisam ser executadas em segundo plano por *triggers* ou disparos de tempo ou eventos(GUNARATNE, 2022). Na figura 6 é mostrado a arquitetura sem servidor.

Figura 6 – Arquitetura sem servidor ou serverless



Fonte: (KOLLER, 2022)

3 TRABALHOS RELACIONADOS

A criação de aplicações utilizando a tecnologia sem servidor ainda não está totalmente disseminada, visto que a tecnologia tem poucos mais de sete anos. Os trabalhos a seguir visam a melhoria através de *Serveless Computing* integrada em suas aplicações quanto a extensibilidade, custo e integração com serviços externos.

3.1 Criando um chatbot com *Serverless Computing*

Na implementação de um *ChatBot* de Yan *et al.* (2016) com *Serverless Computing*, utilizando a plataforma de computação sem servidor distribuída de código aberto OpenWhisk (OPENWHISK, 2020), foram realizadas as criações de funções como serviços (FaaS) que se integrassem com serviços externos da IBM (2020), os principais serviços integrados ao *ChatBot* foram: Função que invoca o noticiário da IBM para obter os três principais títulos de artigos de notícias do dia; Função *Representational State Transfer* (REST) que chama o serviço de piada; Função que retorna a data atual para o usuário. As entradas de dados pelos usuários eram feitas de forma textual ou de áudio, sendo que existia conversão de texto para áudio e vice-versa.

Como neste trabalho, o *ChatBot* armazena arquivos de mídia e metadados relacionados na nuvem através de funções *serverless*. Entretanto, ao contrário de mensagens de áudio e som, este visa armazenar documentos *Portable Document Format* (PDF).

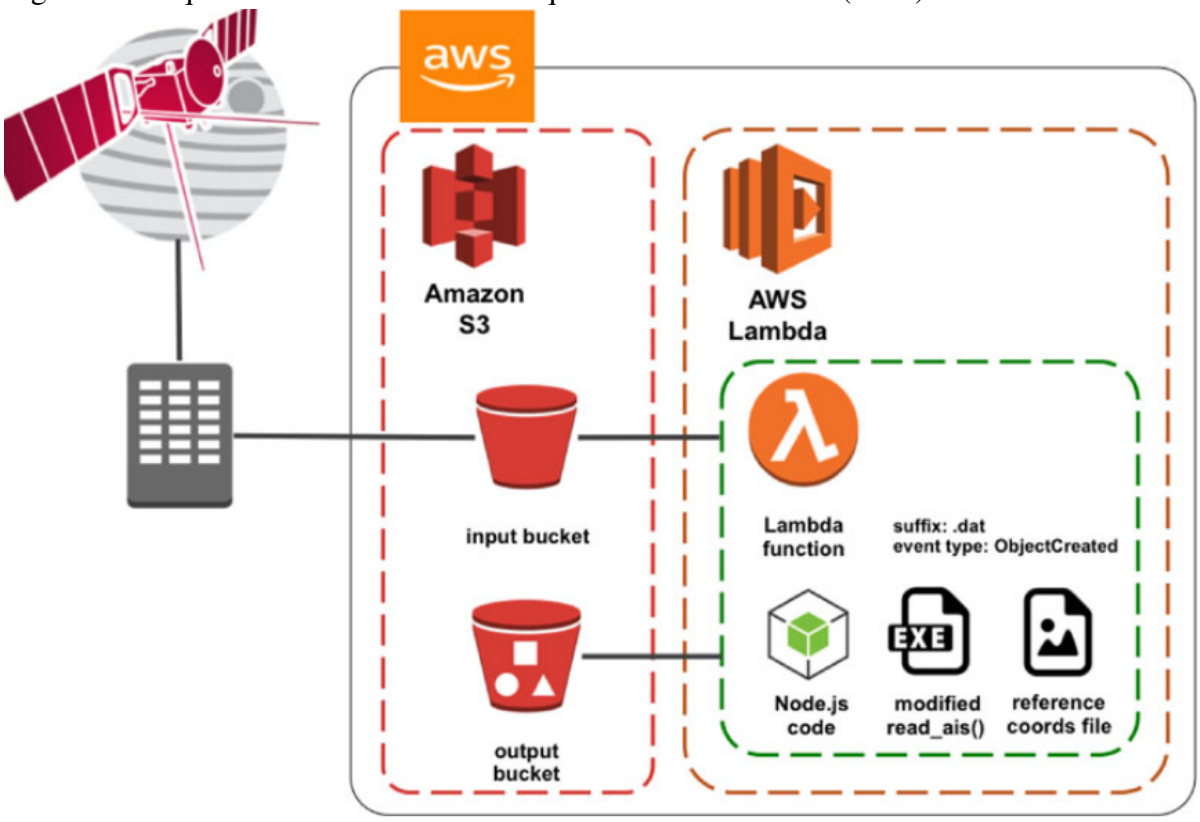
3.2 *Serverless Computing*: do planeta Marte à nuvem

No trabalho de Vázquez-Poletti e Llorente (2018), hospedada na AWS Lambda, foi criada uma aplicação para processamento dos dados vindas do aparelho MARSIS, um sonar e altímetro de radar e baixa frequência instalado no módulo da sonda *Mars Express da European Space*. A aplicação processa os dados do MARSIS a partir da experiência da *Active Ionospheric Sounding* (AIS) e exibe-o graficamente para identificar campos magnéticos.

No Lambda foi criada uma função que executa as principais operações no arquivo de dados especificado e outra que manipula o arquivo de dados, chamando o executável e copiando o arquivo de saída para um repositório especificado. Uma outra função para lidar com a copia de arquivos de saída foi então configurada para ser disparada toda vez que um novo arquivo com o sufixo .dat aparece no *bucket* do serviço S3.

Após a função ser disparada o arquivo analisado era armazenado em outro *bucket*. A figura 7 a seguir mostra a arquitetura do trabalho citado.

Figura 7 – Arquitetura do trabalho de Vázquez-Poletti e Llorente (2018)



Fonte: (Vázquez-Poletti; Llorente, 2018)

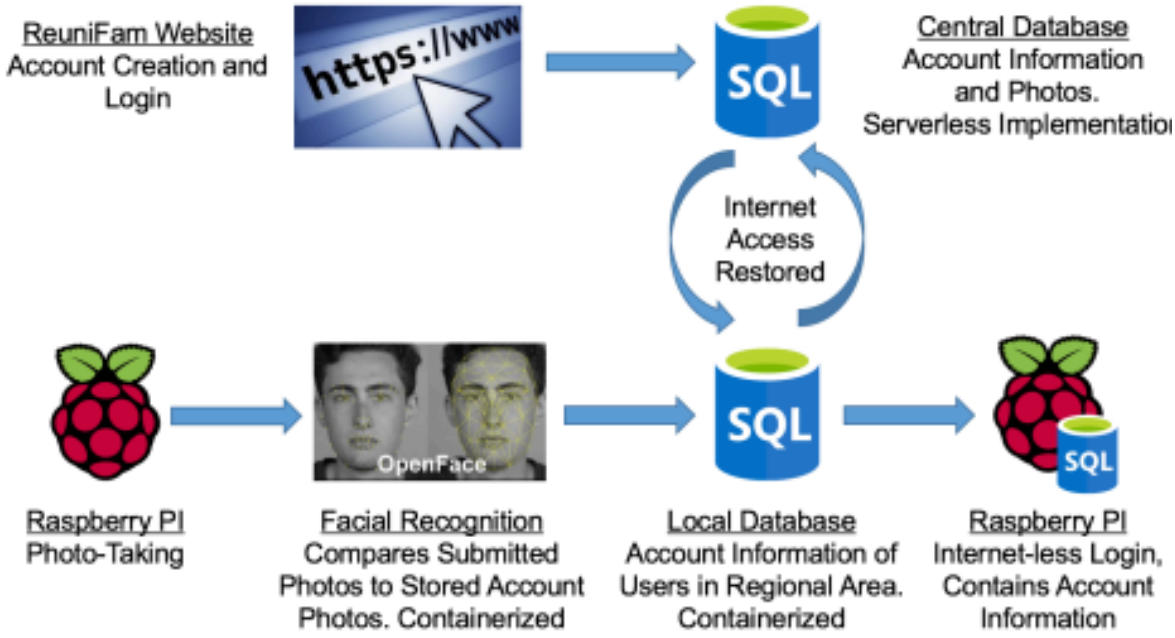
Como no trabalho feito por Vázquez, este trabalho pretende analisar documentos armazenados em um *bucket* para extração dos dados referentes aos documentos armazenados.

3.3 Reunificando famílias após um desastre via *Serverles Computing* e *Raspberry Pi*

No trabalho de FRANZ *et al.* (2018) foi desenvolvido uma aplicação para ajudar na identificação de crianças desaparecidas em tragédias naturais, como o furacão Katrina. O sistema utiliza o serviço de banco de dados da AWS, o *Relational Database Service* (RDS) para armazenar os dados da aplicação. Um veículo aéreo não tripulado equipado com uma câmera coletava imagens e vídeos do lugar ocorrido, como também da possível localização das vitimas da tragédia. Nas imagens obtidas pelo *drone*, eram comparadas com as imagens armazenadas no banco de dados do sistema no qual estava armazenadas imagens das vitimas providas pelos seus parentes.

O reconhecimento das vitimas foi feita através de um sistema de código aberto o OpenFace (OPENFACE, 2020), que fornece reconhecimento facial, assim as vitimas eram reconhecidas e entregues a seus responsáveis o mais rápido possível. A Figura 8 a seguir mostra a arquitetura do trabalho citado.

Figura 8 – Arquitetura do sistema para a reunificação de crianças perdidas com os pais depois de um desastre



Fonte: (FRANZ *et al.*, 2018)

Assim como o trabalho apresentado por Franz, este trabalho previu o acesso a uma base de dados relacional com informações sobre os documentos armazenados.

3.4 Codificação, rápida e lenta: processamento de vídeo de baixa latência utilizando milhares de *threads*

No trabalho de Fouladi *et al.* (2017), há a descrição de uma ferramenta *web*, chamada de *ExCamera* para processamento de vídeo massivamente paralela e baseada em nuvem que cria o *back-end* para aplicações interativas de processamento de vídeo. O *ExCamera* codifica pequenos trechos do vídeo em *threads* independentes (fazendo a maior parte do trabalho "lento"e em paralelo) o *ExCamera* invoca milhares de funções Lambda implementadas em C++ em segundos, que são executadas de forma paralela e, em seguida, junta esses trechos em uma passagem serial "rápida", usando um codificador escrito no estilo de passagem de estado explícito com estados intermediários nomeados. O usuário pode renderizar vídeos de 4K de forma interativa e acessível via *web*, o objetivo do trabalho consiste em executar a tarefa rapidamente.

Assim como o trabalho apresentado por Fouladi, este trabalho prever invocações de funções Lambda via *web*, mas ao contrário do trabalho do citado, este não invocará funções lambdas de forma paralela. As invocações no presente trabalho serão disparadas após algum evento ou submissão de dados do usuário e as funções não serão disparadas de forma paralela, mas, em sequência.

3.5 *On-premises Serverless Computing* Processamento de dados para aplicativos orientados a eventos

No trabalho de Pérez *et al.* (2019) consiste em facilitar o uso de computação orientada a eventos para aplicativos científicos que requerem processamento de arquivos. Nesse contexto, é fornecido aos usuários uma aplicação que possa auto-implantar uma plataforma integrada e escalável, e que possa ser acessada via *web*, na ferramenta os usuários podem definir e gerenciar o ciclo de vida completo das funções, que serão disparadas quando os usuários carregam arquivos para pastas especificadas.

O objetivo do trabalho de (PÉREZ *et al.*, 2019) é abstrair à definição de tarefas a serem executadas, as fontes de eventos, o gerenciamento da contenção de recursos, o gerenciamento das saídas do trabalho e, especialmente, a implantação de toda a plataforma em todas as nuvens, incluindo o gerenciamento de nuvem em plataformas locais, como OpenNebula (OPENNEBULA, 2020) e OpenStack (OPENSTACK, 2020) e provedores públicos de nuvem também. Com isso, os usuários podem fazer *upload* de arquivos através do navegador da *web*, o

que dispara a execução das funções para processar os arquivos. O resultado do arquivo de saída após ser processado é disponibilizado ao usuário para ser feito o *download* do arquivo usando o navegador da *web*.

Assim como o trabalho apresentado, este trabalho utilizará de computação orientada a eventos, após o usuário fazer o *upload* do arquivo para a Amazon S3 via navegador da *web*, será disparado uma função para processamento do arquivo.

Na Tabela 1 é mostrado a comparação em geral dos trabalhos relacionados.

Tabela 1 – Tabela de trabalhos relacionados

	Provedor de Serviços	Orientado a Eventos	Invocação Direta	Dados Processados
Yan et al. (2016)	OpenWhisk	-	X	Aúdio e Texto
Vázquez-Poletti e Llorente (2018)	AWS	X	-	Arquivo de Dados da Sonda MARSIS
Franz et al. (2018)	AWS	-	X	Imagens e Vídeos
Fouladi et al. (2017)	AWS	-	X	Vídeos
Pérez et al. (2019)	OpenFaaS	X	-	Vídeos
Este trabalho	AWS	X	X	Arquivo PDF

Fonte: Elaborado pelo autor.

4 REPOSITÓRIO VIRTUAL DE DOCUMENTOS

Este capítulo descreve a estrutura e desenvolvimento da aplicação, de uma forma mais detalhada tanto *front-end* como *back-end*, mostrando os métodos utilizados para a criação da aplicação.

4.1 Serviços e Ferramentas de Desenvolvimento Utilizados

A Amazon *Web Services* ou AWS é a plataforma de nuvem mais adotada e mais abrangente do mundo (AMAZON, 2021a), oferecendo mais de 200 serviços completos de *datacenters* em todo o mundo. A AWS oferece uma gama de serviços em nuvem: de tecnologias de infraestrutura, como computação, armazenamento e bancos de dados, a tecnologias emergentes como *machine learning* e inteligência artificial, *data lakes*, análises e Internet das Coisas. Portanto, trata-se uma alternativa representativa das possibilidades das nuvens modernas e foi escolhida como base para este trabalho. A seguir detalhamos os serviços da AWS que foram utilizados na construção deste trabalho, além do *framework* e linguagem utilizada.

4.1.1 AWS Lambda

Já apresentamos o AWS Lambda no Capítulo 2, aqui enfatizamos suas características mais importantes para nossa aplicação. O AWS Lambda é um serviço de computação sem servidor e orientado a eventos que permite executar código para praticamente qualquer tipo de aplicação ou serviço de *back-end* sem provisionar ou gerenciar servidores. Você pode acionar o *Lambda* a partir de mais de 200 serviços da AWS e aplicações de *software* como serviço SaaS e pagar apenas pelo que usar (AMAZON, 2014).

O serviço oferece alta disponibilidade e realiza manutenção automática das tarefas relacionadas ao servidor, como provisionamento da capacidade de processamento e ajuste automático do uso da memória para a execução da função. Ele também registra os *logs* das funções após a execução, além disso, é compatível nativamente com várias linguagens de programação, que são as seguintes: Java, Go, PowerShell, Node.js, C#, F#, Python e Ruby.

Os códigos utilizados no Lambda são chamados de funções Lambdas e podem ser chamados em resposta a eventos disparados por outros serviços da AWS, por requisições *Hypertext Transfer Protocol* (HTTP) intermediadas pelo API *Gateway*. Como também, podem ser chamadas por meio da interface de usuário, utilizando *Software Development Kit* (SDK)

oferecido pela AWS, com suporte as linguagens mencionadas anteriormente (AMAZON, 2014).

No presente trabalho, as funções *Lambdas* serão executadas por meio de requisições HTTP, feitas pelo usuário por meio da interface *World Wide Web* (WEB) disponibilizada, como também, executadas por meio de disparos acionados por outro serviço. Serão gerenciadas pelo API *Gateway* a fim de padronizar as chamadas das APIs.

4.1.2 API Gateway

O Amazon API *Gateway* é um serviço gerenciado que permite que desenvolvedores criem, publiquem, mantenham, monitorem e protejam APIs em qualquer escala com facilidade. APIs agem como a "porta de entrada" para aplicativos acessarem dados, lógica de negócios ou funcionalidade de seus serviços de *back-end*. Com o API *Gateway*, é possível criar APIs REST e APIs *WebSocket* que habilitam aplicativos de comunicação bidirecionais em tempo real. O API *Gateway* dá suporte a cargas de trabalho em contêineres e sem servidor, além de aplicativos da *web*. Como também permite a criação de APIs REST que possam interagir com outros serviços da amazon ou serviços externos da *web* (AMAZON, 2020b). Nesse trabalho o modelo de API escolhido foi o REST.

O REST significa *Representational State Transfer* (em português, Transferência de Estado Representacional). Trata-se de uma abstração da arquitetura da *web*. O REST consiste em regras ou princípios que quando seguidas permitem a comunicação e transferência de informações pela rede. Utilizando o modelo REST uma API pode ser consumida por aplicações em diferentes plataformas por meio de metódos HTTP (GET, POST, PUT, PATCH e DELETE) (PIRES, 2021).

4.1.3 Amazon SNS

O Amazon *Simple Notification Service* (SNS) é um serviço de envio de mensagens de publicação/assinatura totalmente gerenciado, altamente disponível, seguro e durável que permite o desacoplamento de microsserviços, sistemas distribuídos e aplicativos sem servidor (AMAZON, 2020d).

O serviço no presente trabalho, é utilizado para enviar um *Short Message Service* (SMS) ao usuário após o cadastro na interface WEB, com um curta mensagem informando-o para que verifique o *e-mail* para concluir o cadastro.

4.1.4 Amazon Cognito

O Amazon *Cognito* permite adicionar cadastramento, *login* e controle de acesso de usuários a aplicativos *web* e móveis com rapidez e facilidade. O Amazon *Cognito* escala até milhões de usuários e oferece suporte a *login* com provedores de identidade social como Facebook, Google e Amazon, além de provedores de identidade empresariais por meio de SAML 2.0 (AMAZON, 2020e).

Os grupos de usuários do Amazon *Cognito* fornecem um armazenamento de identidades seguro que escala até milhões de usuários. Os grupos de usuários do *cognito* podem ser configurados mais facilmente sem provisionar infraestruturas, e todos os membros do grupo de usuários têm um perfil de diretório que pode ser gerenciado por meio de um Kit de Desenvolvimento de Software (SDK).

O Amazon *cognito* oferece soluções para controlar o acesso de um aplicativo a recursos da AWS. Você pode definir funções e mapear usuários a funções diferentes para que o aplicativo possa acessar apenas os recursos autorizados para cada usuário. Opcionalmente, é possível usar atributos de provedores de identidade em políticas de permissão do AWS *Identity and Access Management* (IAM) para controlar o acesso a recursos por usuários que atendem a condições de atributos específicas. Como também, oferece suporte à autenticação multifator.

Após o cadastro na plataforma, o usuário é adicionado ao *pool* de usuários do Amazon *Cognito* que fica aguardando a confirmação no *e-mail* para que o cadastro seja concluído com sucesso. O *pool* de usuários é o diretório que armazena com segurança os atributos de perfil dos usuários criados.

4.1.5 Amazon S3

O Amazon *Simple Storage Service* (Amazon S3) é um serviço de armazenamento de objetos que oferece escalabilidade, disponibilidade de dados, segurança e performance. O Amazon S3 oferece uma grande variedade de casos de uso, como *sites*, aplicações para dispositivos móveis, *backup* e restauração, arquivamento, aplicações empresariais, dispositivos *IoT* e análises de *big data* (AMAZON, 2020c).

O serviço é utilizado para armazenar de forma segura e totalmente escalável os arquivos PDFs, providos pelos usuários. O serviço S3 oferece vários tipos ou classes de armazenamentos na sua utilização, a classe utilizada nesse trabalho foi a *Standard*. Além, de

armazenar os códigos *Lambdas* após o *deploy*.

4.1.6 Amazon RDS

O Amazon *Relational Database Service* (Amazon RDS) facilita a configuração, a operação e a escalabilidade de bancos de dados relacionais na nuvem. O serviço oferece capacidade econômica e redimensionável e automatiza tarefas demoradas de administração, como provisionamento de *hardware*, configuração de bancos de dados, aplicação de *patches* e *backups* (AMAZON, 2020f).

O Amazon RDS oferece suporte a vários bancos de dados relacionais como: Aurora, PostgreSQL, MySQL, MariaDB, Oracle e SQL Server. Que podem ser rapidamente criados e configurados pelos usuários, o banco de dados utilizado no presente trabalho foi o MySQL.

O serviço é utilizado para armazenar os metadados dos arquivos PDFs importados pelos usuários através da interface disponibilizada, após a inserção do arquivo no S3 é disparada uma função que envia os dados do arquivo do S3 ao RDS.

4.1.7 IAM

O AWS IAM fornece controle de acesso refinado em toda a AWS. Com o IAM, é possível especificar quem pode acessar quais serviços e recursos e em que condições. Com as políticas do IAM, é possível gerenciar permissões para seu quadro de funcionários e sistemas para garantir permissões com privilégios mínimos (AMAZON, 2021b).

Outros recursos interessantes do IAM é a criação de funções ou *roles* e políticas de acessos aos serviços fornecidos pela plataforma. Uma função do IAM são os serviços que podem ser atribuidos aos usuários para que eles possam acessar determinado serviço, já as políticas são as ações que o usuário poderá executar naquele serviço, exemplo, criar *buckets* no S3, porém, a politíca aplicada foi que apenas pode visualizar os arquivos que estão nos *buckets*, não podendo fazer a exclusão dos arquivos.

4.1.8 Serverless Framework

O *Serverless Framework* abstrai os principais conceitos em torno da computação sem servidor, ajudando a desenvolver e implantar funções do AWS Lambda, junto com os recursos de infraestrutura da AWS de que eles precisam dos outros serviços. É uma *Comand Line Interface*

(CLI) que oferece estrutura, automação e melhores práticas *out-of-the-box*, permitindo que você se concentre na construção de arquiteturas sofisticadas, orientadas a eventos e sem servidor, compostas por funções e eventos (SERVERLESS, 2021).

O *Serverless Framework* utiliza o arquivo YAML para configuração e *deploy* das funções e recursos a serem utilizados pelas funções. Permitindo aos desenvolvedores concentrarem a maior parte do tempo na codificação, o que aumenta a eficiência e agilidade no momento da implementação e implantação. Com isso, fica bem mais fácil, adicionar uma nova função ou remover um recurso a ser criado, visto que após a configuração do arquivo .YML basta um comando para que se inicie o processo de *deploy* em algum provedor de nuvem utilizado pelo desenvolvedor.

O *framework* utiliza controle de versionamento, o que possibilita atualizar uma implantação já existente ou reverter um processo em caso de erro na implantação. Além disso, ele pode ser utilizado com diferentes linguagens de programação tais como: C#, F#, Go, Node.js, Python e Ruby, como também possui suporte para diversas provedoras em nuvem, tais como *AWS Cloud, Google Cloud, Azure,* dentre outros (SERVERLESS, 2021). A linguagem utilizada em conjunto com o *serverless framework* foi o Nodejs.

4.1.9 NodeJS

O Node.js pode ser definido como um ambiente de execução JavaScript *server-side*. Isso significa que com o Node.js é possível criar aplicações JavaScript para rodar como uma aplicação *standalone* em uma máquina, não dependendo de um *browser* para a execução, como estamos acostumados.

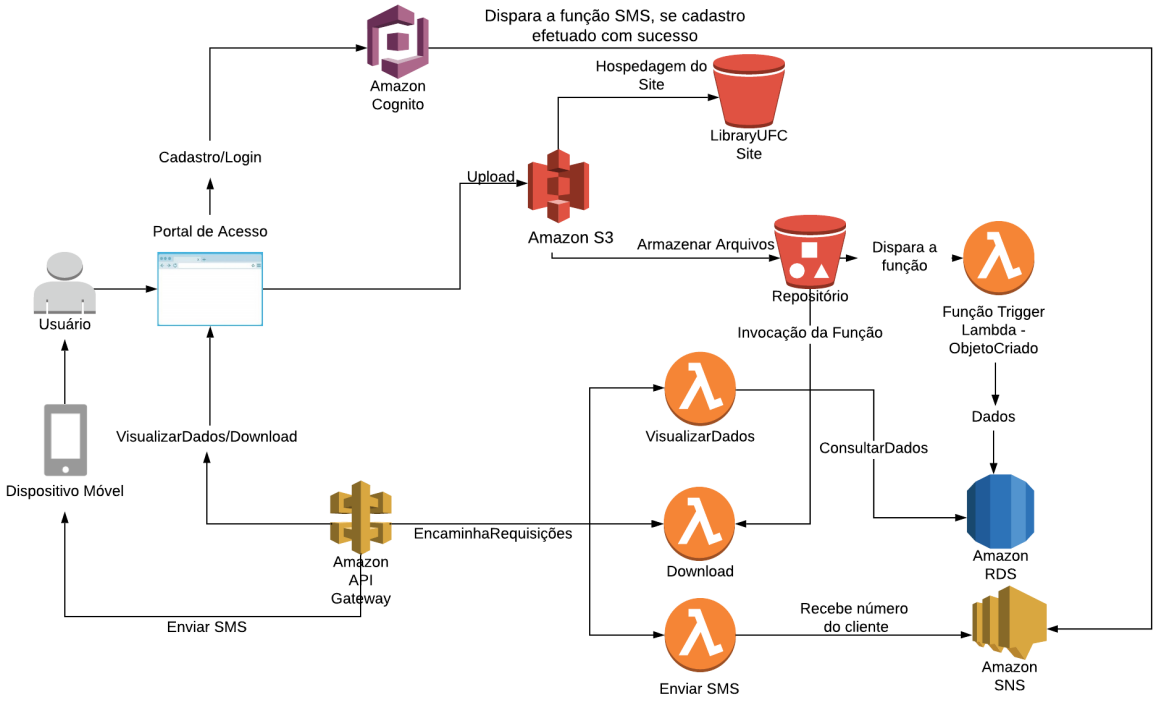
O principal motivo de sua adoção é a sua alta capacidade de escala. Além disso, sua arquitetura, flexibilidade e baixo custo, o tornam uma boa escolha para implementação de microsserviços e componentes da arquitetura *Serverless.* Inclusive, os principais fornecedores de produtos e serviços *Cloud* já têm suporte para desenvolvimento de soluções escaláveis utilizando o Node.js (LENON, 2021).

O *NodeJS* foi a linguagem escolhida para a implementação das funções *Lambdas*, por conta da familiaridade com a linguagem *JavaScript* e pelo *Serverless Framework* dar suporte a linguagem nativamente. Além, de oferecer suporte a vários módulos que são trechos de códigos já escritos que podem executar uma ação de forma rápida e fácil.

4.2 Arquitetura da Aplicação

A Figura 9 mostra a arquitetura utilizada no desenvolvimento da aplicação. Como já discutido, trata-se de um repositório de arquivos na nuvem. Após a autenticação no Amazon *Cognito* o usuário pode fazer *upload*, *download* e armazenamento dos metadados dos arquivos PDF em uma base de dados. Após o usuário ao fazer *upload* de um arquivo na plataforma, o documento é armazenado em um *bucket* do S3, sendo disparada uma função para que o PDF seja processado e seus metadados extraídos. Esses metadados são armazenados em uma base de dados no serviço RDS, com isso é utilizado uma função *Lambda* para retornar os dados do serviço RDS para a tela do usuário, mostrando os metadados do arquivo que foi feito *upload* anteriormente. Como também, a opção de *download* daquele arquivo.

Figura 9 – Arquitetura do Repositório de Documentos



Fonte: Elaborado pelo próprio autor.

4.2.1 Autenticação

Ao se cadastrar na plataforma, é enviado um SMS para o número fornecido por ele, informando para que verifique o *e-mail* informado. O usuário criado é adicionado ao *pool* de usuários do Amazon *Cognito* e fica aguardando a confirmação do usuário que foi enviado para

o *e-mail* fornecido, para que assim seja concluído o cadastro. Somente após a confirmação no *e-mail*, o usuário poderá fazer *login* no sistema, como também fazer o *reset* da senha, nessa opção é enviado uma numeração *One Time Password* (OTP) para o *e-mail* cadastrado e o usuário precisa fornecer essa numeração para fazer a alteração de senha. Após esse processo, o usuário poderá utilizar todos os recursos da aplicação.

4.2.2 Comunicação com as APIs

A comunicação com as APIs se dá por meio de requisições HTTP utilizando o *API Gateway*. As chamadas são realizadas a medida que o usuário vai interagindo com aplicação, ao enviar um formulário ou clicar em botões para *submit* de dados. Todas as requisições são feitas ao serviço *API Gateway*, que irá manipular as funções e designar qual serviço precisa ser invocado, assim retornando o resultado da função caso precise.

Antes de utilizar o serviço *API Gateway* para invocar as funções *Lambdas*, foi dado permissão ao mesmo por meio do IAM para ter total acesso sobre as funções de *back-end Lambdas*. Para que assim seja possível receber chamadas e enviar de volta o resultado.

4.2.3 Processamento dos dados

Após o *upload* do arquivo através da plataforma, uma função *Lambda* é disparada por meio do *API Gateway*. A função *Lambda* recebe os metadados do arquivo no S3 e então os envia ao serviço RDS, que irá armazenaá-los em um banco de dados. Em seguida, ao atualizar a página é disparada uma nova função *Lambda* para invocar os dados que estão no banco de dados e exibi-los na tela. As chamadas são feitas por meio do SDK NodeJS da Amazon, no qual é possível realizar as operações de inserção, atualização, exclusão e consulta, respectivamente dos termos em inglês: *insert, update, delete e select*.

4.2.4 Armazenamento

Os serviços utilizados para o armazenamentos foram os serviços S3 e *RDS*. O S3 foi utilizado para armazenar os arquivos, após o *upload* na plataforma, já o RDS, foi utilizado para armazenar os metadados dos arquivos. Dentro do banco foi criado apenas uma tabela contendo os campos: Id, Titulo, Autor e NumPages, que corresponde ao número de páginas do arquivo.

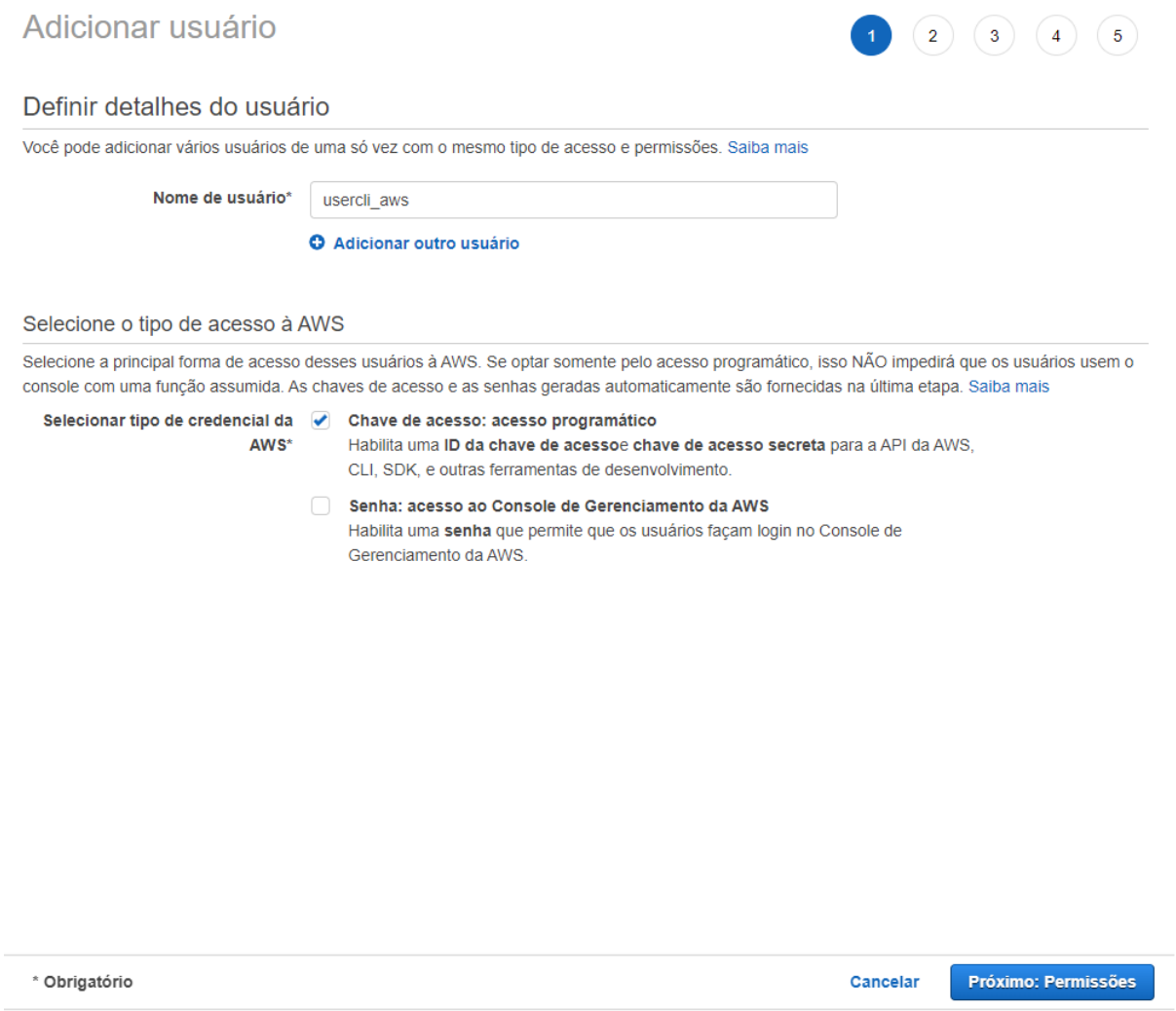
4.3 Desenvolvimento do *Back-end*

Nessa seção, mostramos a implementação do *back-end* da aplicação na nuvem. A seguir as etapas subsequentes feitas durante o processo.

4.3.1 Usuário no IAM

Para a utilização dos serviços da AWS de forma externa, é preciso criar um usuário e atribuir funções e políticas a ele. O processo é bem simples e pode ser feito inteiramente através da *dashboard* da AWS. No painel da AWS basta pesquisar por IAM e clicar no serviço para começar a utilizá-lo. O IAM permite a criação de dois tipos de usuários: acesso programático ou de acesso ao console de gerenciamento da AWS. A Figura 10 mostrar as 2 opções.

Figura 10 – Usuário no IAM



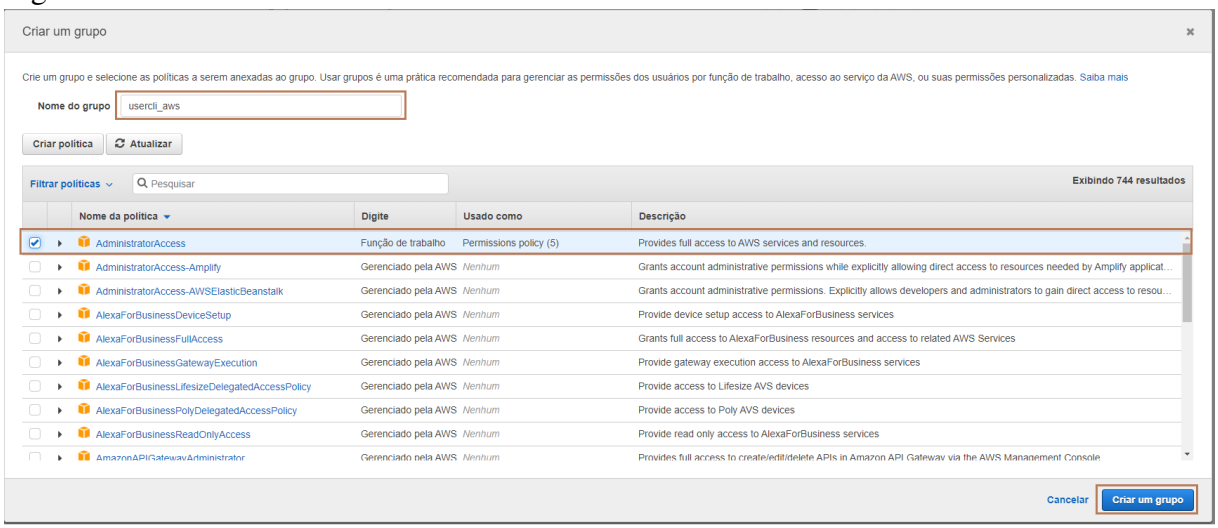
Fonte: Elaborado pelo próprio autor.

- **Acesso programático** - são geradas 2 chaves de acesso para os usuários uma chave de ID de acesso *(Access key ID)* e outra chave secreta *(Access secret key)*. Com essas chaves é possível acessar de forma externa serviços da AWS, através do AWS-SDK ou o AWS-CLI.

- **Acesso ao console** - são criados um nome de usuário e gerada uma senha, o usuário tem acesso ao console da AWS, podendo assim fazer o uso dos serviços que lhe foi permitido.

Na aplicação foi escolhido o acesso programático, a próxima etapa é adicionar as funções ou serviços que o mesmo vai ter e quais políticas de acesso vão lhe ser atribuídas. Foi criado um grupo de usuários e adicionado o usuário criado dentro desse grupo. A permissão que foi dada ao usuário utilizar foi a permissão *AdministratorAccess* que prover acesso completo a todos os serviços da AWS, lembrando, esse tipo de permissão não é indicada, visto que dá todo o poder ao usuário. A Figura 11 mostra as permissões.

Figura 11 – Permissão do usuário no IAM



Fonte: Elaborado pelo próprio autor.

4.3.2 Utilização das chaves de acesso

Existem 2 formas de utilizar as chaves de acessos, a primeira é a de adicionar as chaves dentro do código a ser escrito, o que não é recomendado visto que as chaves de acesso ficam expostas, sendo assim possível que terceiros de alguma forma as tenham. A segunda e a mais recomendada e a que foi utilizada na aplicação é a configuração dentro do AWS-CLI, com apenas um comando dentro do prompt é possível fazer a configuração.

```
$ aws configure
```

Lembrando, para utilizar o comando acima é preciso fazer a instalação do CLI da amazon o AWS-CLI, é fácil e rápido a instalação. Após a configuração das chaves dentro do comando, as chaves serão automaticamente utilizadas pelas aplicações, podendo assim invocar as funções ou serviços através do SDK da Amazon.

4.3.3 Usuário no Amazon Cognito

A adição do usuário ao *pool* de usuários do Amazon *Cognito* é feita por meio da interface do sistema. A criação e configuração do serviço *Cognito* foi feita utilizando o arquivo *serverless.yml*, que é utilizado para o *deploy* da aplicação. Na configuração é preciso adicionar um nome para o *pool*, e outras propriedades necessárias para a configuração, como atributos para a verificação, política de senha, tipo de verificação e qual meio o usuário vai fazer a verificação, na aplicação o atributo para a verificação utilizado foi o *e-mail*, na configuração também é possível personalizar mensagens no envio de *e-mails* automáticos.

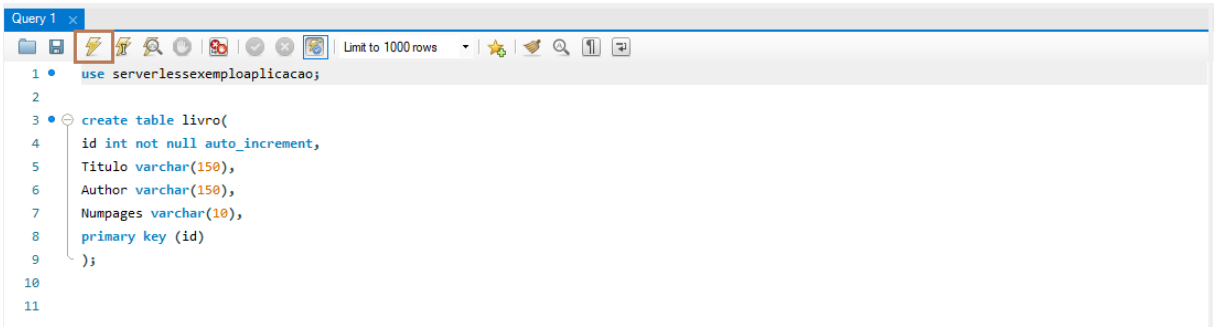
Após a criação do serviço, são disponibilizados duas informações importantes para o usuário, que são: ID do *pool* de usuários e ID de clientes do aplicativo do Amazon Cognito. A primeira é a identificação do recurso localmente dentro do AWS e a segunda é a identificação do provedor de identidade do Amazon Cognito, essas duas informações como também a região que foi implantado o serviço são necessárias para a utilização do mesmo de forma externa.

4.3.4 Banco de Dados no RDS

A criação e configuração do banco de dados também foi feita através do arquivo *serverless.yml*. Na criação do banco de dados é preciso configurar algumas propriedades, entre elas estão: nome da instância, qual o banco a ser utilizado, tamanho da instância, versão do *software*, nome do banco de dados, classe da instância, nome de usuário e senha para acessar o banco, como também permitir que o banco seja público, permitindo que ele seja acessado de forma externa. O banco de dados escolhido foi o MySQL, com o banco criado é possível fazer a conexão com o banco na nuvem utilizando o *software* MySQL *Workbench*, que foi utilizado nesse projeto.

A tabela utilizada na aplicação é constituída dos seguintes atributos: ID, Titulo, Autor e número de páginas. O atributo ID foi definido como *auto increment*, logo qualquer arquivo adicionado ao banco, vai ser incrementando mais um na contagem do ID. Na Figura 12, mostra a tabela criada dentro do *software* MySQL *Workbench*.

Figura 12 – Tabela no banco de dados MySQL



Fonte: Elaborado pelo próprio autor.

4.3.5 Criação e Codificação das APIs

A criação dos serviços e codificação das *APIs* são feitas através do *framework serverless*, após a instalação para criar um projeto *serverless* se dar por meio do seguinte comando:

```
$ serverless create --template aws-nodejs --path serverlessexemplo
```

Após a criação do projeto, dentro da pasta existem 2 arquivos que são bem importantes, o primeiro é o *serverless.yml* e o segundo é o *handler.js*.

- ***serverless.yml*** esse arquivo é onde se encontra toda a infraestrutura da aplicação, nele se encontram os serviços e funções a serem criados na AWS.

- ***handler.js*** nesse arquivo é onde se encontra a codificação das funções de fato e que serão utilizadas no *lambda* através do arquivo *serverless.yml*, que fará uso dessa informação.

Para cada função criada no arquivo *handler.js* é preciso fazer a referência das mesmas dentro do arquivo *serverless.yml*, para que assim possa ser criado dentro do *Lambda*. Com isso, após o *deploy* da aplicação são gerados *endpoints* das funções configuradas, para que as funções possam ser utilizadas pelo usuário. A configuração do arquivo *serverless.yml* e do *handler.js* encontram-se no apêndice.

Com os arquivos já configurados, para fazer o *deploy* da aplicação, é necessário o seguinte comando:

```
$ serverless deploy
```

O arquivo utilizado para configuração dos serviços através do *serverless.yml* se encontram no apêndice, como também um exemplo de função utilizada nesse projeto.

4.4 Interface Gráfica

Após a finalização do *back-end* foi criado a interface gráfica para a interação do usuário com os serviços em nuvem. A interface possui 6 telas, dos quais 4 possuem interação com o usuário, as imagens das telas se encontram no apêndice, juntamente com os códigos utilizados. A seguir são listadas as telas e quais serviços são utilizados nos mesmos.

Na **Tela de *Home*** temos uma tela inicial simples com um menu superior com as opções de navegação que o usuário pode utilizar. Já a **Tela Sobre** apresenta do que se trata a aplicação e como a mesma funciona e uma breve descrição do que é *serverless*. Para acesso ao sistema, **A Tela de *Login***, contendo dois campos com *e-mail* e senha que precisam ser preenchidos pelo usuário, necessários para a autenticação. Também na tela de *login* é utilizado o Amazon Cognito para autenticação do usuário na plataforma. Os usuários fazem o registro na **Tela de Cadastro**, contendo 5 campos com nome, *e-mail*, senha, confirmação da senha e fone, os campos de senha precisam ser preenchidos corretamente pelo usuário, visto que foi configurado no *Cognito* políticas de senha que precisam ser seguidas. Nessa tela é utilizado o serviço *Cognito*, para cadastro do usuário na aplicação.

Caso esqueça a senha, na **Tela de Recuperação de Senha** há um campo *e-mail* que precisa ser preenchido pelo usuário. Após o usuário disponibilizar o *e-mail*, é enviado uma numeração OTP, para o *e-mail* especificado, sendo necessário que o usuário utilize essa numeração na plataforma para efetuar a recuperação da senha. Por fim, na **Tela de *Upload* e Visualização de Dados** tela possui um campo onde o usuário possa escolher o arquivo para ser feito o *upload* para o *bucket* da Amazon. Após o *Upload* do arquivo são desencadeados funções *triggers* ou disparos, para enviar os dados do arquivo no *bucket* para o serviço RDS, como também adicionar os dados do arquivo na tela e o botão de *download* do arquivo na tela. Os serviços utilizados são S3, RDS.

No apêndice, temos as imagens das telas utilizadas nessa aplicação.

5 COMPARAÇÃO ENTRE ARQUITETURAS

Nesta seção, fazemos uma comparação entre a arquitetura proposta pelo trabalho e a arquitetura consagrada utilizada na construção de aplicações *web*, que são as arquiteturas sem servidor e monolítica respectivamente. Para efeito de comparação, consideramos os serviços que seriam utilizados para implementar o cenário criado na arquitetura sem servidor em uma arquitetura monolítica na nuvem. Será utilizado a calculadora *Total Cost of Ownership* (TCO) da Amazon, para calcular o custo de ambas as arquiteturas e apresentar os resultados no decorrer do capítulo.

5.1 Arquitetura Monolítica

Nesta seção, descrevemos um ambiente monolítico mínimo capaz de oferecer as mesmas características do ambiente sem servidor da AWS Lambda, como balanceamento de carga e tolerância a falhas.

5.1.1 Ambiente de Computação

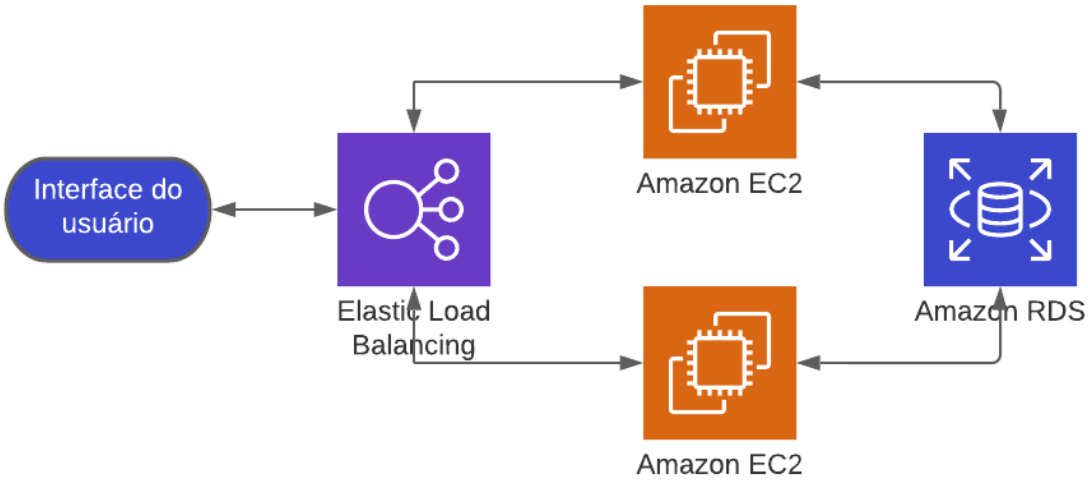
Para desenvolver o ambiente de execução em nuvem na arquitetura monolítica é preciso utilizar um servidor para agrupar os serviços como uma pilha. Na Amazon AWS o serviço existente é o *Elastic Compute Cloud* (EC2) ou o *Lightsail*, mas, para nossa comparação será utilizado o EC2, ele permite a criação e configuração de instâncias em nuvem, podendo ser escolhido pelo usuário qual sistema operacional vai ser iniciada na instância. Dentre eles estão os mais utilizados que são *Windows* e várias distribuições de Linux, além de possuir também para *MacOS*. Supondo que o sistema operacional escolhido foi o Ubuntu 20.4, e o tamanho da instância escolhida foi *t2.micro* com apenas 1GiB de memória *ram* e 1$vCPUs$.

Para armazenamento foi alocado um SSD de uso geral (GP2) com 50GB de capacidade e que a transferência de dados de entrada e saída foi configurada para ser 1TB de dados por mês. Para nosso cenário foi preciso criar 2 instâncias com as mesmas configurações para ter um balanceamento de carga em nossa aplicação. Com as instâncias criadas é necessário instalar dependências como linguagem de programação a ser utilizada e um servidor WEB. Na nossa arquitetura monolítica foi adicionado o serviço Amazon *Elastic Load Balancing* (ELB), para fazer a distribuição do tráfego de rede da aplicação entre os servidores criados. O serviço foi configurado para processar 1TB de dados por mês com destino ao servidores EC2.

5.1.2 Banco de dados

Na arquitetura monolítica a utilização de um banco de dados se dá por meio da instalação e configuração do mesmo dentro do servidor. Nesse tipo de arquitetura é muito comum a utilização de um *Data Base Management System* (DBMS), que é um sistema de gerenciamento de banco de dados, onde é possível encontrar todas as tabelas utilizadas e o compartilhamento das mesmas entre as funcionalidades do sistema. Para efeito de comparação será utilizado o serviço Amazon RDS em ambas arquiteturas, com as seguintes configurações. Uma instância com tamanho *db.t2.micro* com apenas 1 *GiB* de memória *RAM* e 1 vCPUs. Para o armazenamento será utilizado um SSD de uso geral(gp2) com 50GB de armazenamento. A figura 13 a seguir mostra o exemplo da arquitetura monolítica a ser comparada.

Figura 13 – Exemplo arquitetura monolítica em nuvem



Fonte: Elaborado pelo próprio autor.

5.1.3 APIs

Na arquitetura monolítica toda a lógica de negócios do sistema se encontra em um único ponto, logo toda a comunicação da aplicação que são as requisições e respostas são enviadas ao um único servidor. Geralmente as funções são interdependentes e entrelaçadas, de forma que a inclusão de novas funcionalidades ou manutenção do sistema possa acarretar algum tipo de inconsistências ou comportamentos inesperados. Em outras palavras, um servidor WEB é executado em cada instância EC2 expondo a API REST da aplicação.

5.1.4 Envio de Mensagens

O serviço para envio de mensagem SMS usado para compor a arquitetura foi o Amazon *Pinpoint*. Ele é um serviço de fácil configuração e pode ser usado utilizando o SDK da amazon. O serviço foi configurado para enviar 1.000 SMS mensais, na calculadora TCO da amazon não é possível fazer o cálculo de envio de SMS, visto que o valor cobrado é diferente para cada país. É disponibilizado pela amazon o valor do SMS cobrado para o brasil, que é de $0.02297 na classe de SMS promocional.

5.1.5 Escalabilidade e Manutenibilidade

Na arquitetura monolítica é comum a utilização de mais de um servidor, como também a utilização de um balanceador de carga, para que as requisições possam ser mandadas para o servidor mais próximo a fim de evitar possíveis gargalos. Mesmo que para lançar um nova instância seja necessária apenas uma parte do código, na arquitetura monolítica exige que todo o sistema seja replicado, o que pode gerar custos desnecessários. A medida que o código da aplicação aumenta, dificulta a manutenção dos desenvolvedores visto que precisam navegar em gigantescas linhas de códigos para solucionar o problema, o que pode ser mais demorado e ter possíveis perdas de custos durante o período de manutenção. Além, de que qualquer mudança no sistema se faz necessário a reinicialização do sistema, nisso faz que a manutenção seja feita em horários estratégicos.

5.1.6 Custo com a Arquitetura Monolítica

A tabela 2 a seguir detalham os serviços utilizados para manter os serviços em execução na AWS por um mês.

Tabela 2 – Tabela de custo arquitetura monolítica em nuvem

Serviço	Tamanho	Serverless	Processamento de dados/Requisições	Período	Estimativa Amazon AWS
Amazon EC2	t2.micro	Não	1TB de processamento de dados	Mensal	119,10 USD
Amazon ELB	Não possui	Não	1TB de processamento de dados	Mensal	24,62 USD
Amazon RDS	db.t2.micro	Não	Não possui	Mensal	36,32 USD
Amazon Pinpoint	Não possui	Não	1 mil	Mensal	22,97 USD
				Total	203,01 USD

Fonte: Elaborado pelo autor.

5.2 Arquitetura Sem Servidor

5.2.1 Ambiente de Computação

Na arquitetura sem servidor, o ambiente de computação é a própria máquina do usuário precisando somente ser necessário a instalação da CLI da Amazon, e a instalação de uma linguagem de programação de sua preferência para começar a utilizar os serviços. Com isso a arquitetura abstrai qualquer tipo de criação e instalação de infraestrutura necessária para abrigar o código, tanto *front-end* quanto *back-end*.

Para que seja possível a utilização do *site* pela *Internet* será adicionado o serviço Amazon *Cloudfront* para hospedar a aplicação após concluída. O serviço foi configurado para ter 1 *TB* de dados de saída para a *Internet* e 2 milhões de solicitações ou requisição *HTTPs* mensais.

5.2.2 Autenticação

A autenticação da aplicação foi feita utilizando o serviço Amazon *Cognito*, o mesmo foi configurado para se ter 1.000 usuários ativos mensais, que são os chamados *Monthly Active Users* (MAU). A configuração só levou em consideração usuários que utilizam o provedor de identidade do amazon *cognito*, assim não levam em conta outros provedores de identidade que podem ser utilizados para autenticação tais, como *Facebook*, *Google*, ou *Apple*. Na arquitetura monolítica, um sistema de SSO próprio (*Single-Sign-On*) precisaria ser implementando utilizando os padrões da indústria. Porém, uma vez pronto, a solução poderia residir nas próprias máquinas virtuais EC2. Entretanto, como mensurar o custo de desenvolvimento de *software* não é trivial, optamos por apresentar o custo do *Cognito* para que o leitor tenha noção do que o serviço oferece em um custo acessível.

5.2.3 Banco de Dados

A utilização do banco de dados na arquitetura sem servidor utilizada nesse trabalho, foi feita usando o serviço Amazon RDS, a configuração e criação do banco foi feita através do arquivo *serverless.yml*, que após o *deploy* o serviço fica disponível. Após a criação do banco na nuvem, basta o usuário instalar a aplicação cliente do banco escolhido anteriormente, que nesse trabalho foi escolhido o *MySQL*. Sendo assim o banco não roda nativamente em sua máquina, mas sim em um servidor em nuvem. A configuração utilizada na criação do banco foi a mesma

citada acima, na arquitetura monolítica. Portanto, é um aspecto monolítico que ainda persiste na arquitetura sem servidor, mas visto que é um serviço gerenciável da AWS, apresenta os mesmos benefícios da computação sem servidor.

5.2.4 APIs

Na arquitetura sem servidor as funções são independentes e executam uma funcionalidade única ou especifica. Assim, nenhum serviço impede a execução de outro e como são separados em módulos fica mais fácil identificar o problema e fazer a manutenção, visto que somente a função com problema é afetada, assim não afetando o resto da aplicação. Para utilização das *APIs* foi utilizado o serviço API *Gateway* com 2 milhões de solicitações mensais e memória *cache* de 0,5GB.

5.2.5 Funções Lambdas

O serviço foi configurado para ter 2 milhões de solicitações, onde cada solicitação terá uma média de 2000*ms* ou 2(dois) segundos de execução da função e a quantidade de memória alocada foi de 512*MB*.

5.2.6 Armazenamento

Para armazenamento foi utilizado o serviço Amazon S3, na classe *standard* o mesmo foi configurado para ter um armazenamento mensal de 1*TB* por mês. Como também, 2 milhões de solicitações *PUT,COPY,POST,LIST* e *GET,SELECT* mensais.

5.2.7 Envio de Mensagens

O serviço para notificação ao usuário por meio de SMS foi o Amazon *Simple Notification Service* (SNS). Ele foi configurado para enviar 1.000 SMS mensais aos usuários utilizando a forma de SMS promocional. A estimativa desse serviço não se encontra na calculadora TCO, pois para cada região ou país os preços se alteram, no brasil o envio para um SMS de saída equivale a $0.02297 dólares.

5.2.8 Escalabilidade e Manutenibilidade

Na arquitetura sem servidor os *deploys* e as replicações são feitas por servidores, máquinas virtuais ou contêineres que se organizam de forma independente. Sendo assim, ele se replica caso seja necessário, devido a alta demanda. Nessa arquitetura as funções são únicas e exercem somente uma determinada função, assim mesmo que a aplicação continue em crescimento em relação as linhas de códigos, fica bem mais fácil encontrar o erro, visto que as funções são individuais e independentes.

Com isso, caso alguma função venha a ter problema, ela não causa problema ao restante da aplicação, o que fica bem mais rápido e seguro a manutenção e correção. Além, de não ser necessário a reinicialização do sistema como um todo, apenas na função com problema caso necessário.

5.2.9 Custo com A Arquitetura Sem Servidor

A tabela 3 a seguir detalham os serviços utilizados para manter os serviços em execução na AWS por um mês.

Tabela 3 – Tabela de custo arquitetura sem servidor em nuvem

Serviço	Tamanho	Serverless	Processamento de dados/Requisições	Período	Estimativa Amazon AWS
Amazon Cloudfront	Não possui	Sim	2 milhões de solicitações	Mensal	87,04 USD
Amazon Cognito	1.000 usuários	Sim	Não possui	Mensal	50,00 USD
Amazon RDS	db.t2.micro	Não	Não possui	Mensal	36,32 USD
Amazon API Gateway	Não possui	Sim	2 milhões de requisições	Mensal	21,60 USD
Amazon S3	1TB	Sim	2 milhões de requisições	Mensal	34,35 USD
Amazon SNS	Não possui	Sim	1 mil	Mensal	22,97 USD
Amazon Lambda	Não possui	Sim	2 milhões de requisições	Mensal	33,73 USD
				Total	286,01 USD

Fonte: Elaborado pelo autor.

6 CONCLUSÕES E TRABALHOS FUTUROS

Analisando os resultados podemos perceber a diferença entre as duas arquiteturas e observar que a diminuição de custo na opção monolítica não é grande. A proposta por esse trabalho tem mais benefícios e uma vida útil a mais em relação a arquitetura comparada, quando se trata em adição de novos serviços, configuração rápida e menos esforço na administração. Além disso, a opção sem servidor é altamente escalável e resiliente, portanto caso o sistema cresça em demanda, o esforço de reconfiguração é irrisório.

A AWS ainda mantém serviços que executam nativamente em servidores e que precisam de configuração manual como o *Lightsail* e o *Pinpoint* por exemplo, o primeiro trata-se de um serviço para prover um servidor em nuvem configurado com wordpress, Nginx, dentre outros. O segundo trata-se de um serviço para envio de SMS, que pode ser configurado para enviar tanto SMS quanto *e-mails*.

Os últimos lançamentos de serviços da AWS, tem sido com foco em arquitetura *serverless*, podemos citar o AWS *Amplify Studio* e o AWS EFS. O *Amplify Studio* é um ambiente de desenvolvimento que permite aos desenvolvedores criar componentes de interface de usuário e o *back-end* com o mínimo de codificação possível. O segundo AWS EFS é um sistema de arquivos sem servidor, que pode ser utilizado junto ao EC2 ou outros serviços. Esses lançamentos indicam uma tendência pro futuro, onde a grande parte dos serviços vai ter o mínimo de esforço na configuração por parte do usuário.

A construção da aplicação, atingiu os objetivos impostos, mas não foi construída utilizando todos os serviços que poderiam melhorar a gerência e controle dos recursos criados e utilizados. Assim, deixando margem para melhorias futuras.

6.1 Vantagens

A arquitetura sem servidor permite que usuários de uma mesma organização que tenham acesso a plataforma da AWS possam compartilhar a criação de novos recursos e a criação das funções *lambdas*. O que fica bem mais fácil o *deploy* ou exclusão de um recurso ou função. Como também, a manutenção de algum serviço ou adição de um novo fica bem mais prático, visto que são utilizados arquivos separados para a construção das funções e adição de um novo serviço. Sendo possível delegar aos desenvolvedores criarem funções diferentes entre si, e que não impactam na construção da outra.

Além destas vantagens, podemos ressaltar a escabilidade automática dos serviços de processamento de dados e de armazenamento, com isso, tira a responsabilidade do desenvolvedor de se preocupar em aumentar a infraestrutura criada quando for necessário. Outro ponto importante é o método de cobrança que é utilizado que se chama *pay per use*, com isso só será cobrado o tempo que o sistema estiver em execução, ou seja esteve processando algum dado, nisso, quando o sistema fica ocioso não é feita nenhuma cobrança.

6.2 Desvantagens

Uma das desvantagens dessa arquitetura, consiste em antes de começar a construção da aplicação, ter conhecimentos sobre os serviços que vão ser utilizados. O que pode gerar um tempo considerável de estudo da plataforma e seus serviços, além do que é aconselhável fazer um *roadmap* e desenhar como vão ser feitas as comunicações entre os serviços. Como também, ter conhecimentos prévios básicos ou avançados de banco de dados seja ele relacional ou não relacional.

6.3 Trabalhos futuros

Após a finalização da aplicação e durante a escrita desse texto, foram encontrados pontos que poderiam ser melhorados, com o intuíto de melhorar o desenvolvimento e controle da aplicação, abaixo foram listados os pontos.

6.3.1 AWS Amplify

O AWS Amplify é um serviço que reúne um conjunto de ferramentas e recursos que auxiliam os desenvolvedores no momento da construção da aplicação. Com o Amplify é possível monitorar as funções *lambdas* e as APIS Rest que foram criadas, além de outros recursos como armazenamento(AMAZON, 2022b). O serviço não foi utilizado, visto que só foi descoberto quando a aplicação já estava sendo finalizada.

6.3.2 SAM

O AWS *Serverless Application Model (SAM)* ou modelo de aplicativo sem servidor da AWS é uma estrutura de código aberto para criar aplicativos sem servidor. Ele fornece sintaxe abreviada para expressar funções, APIs, bancos de dados e mapeamentos de origem de

eventos. Sendo possível com poucas linhas por recurso definir o aplicativo desejado e modelá-lo usando YAML. Durante a implantação, o SAM transforma a sintaxe do SAM em sintaxe do AWS CloudFormation, permitindo a criação de aplicativos serm servidor mais rapidamente (AMAZON, 2022c).

A implantação dos serviços e funções utilizadas na construção da aplicação foi o YAML, que é o mesmo utilizado pelo SAM, porém, não foi seguindo as melhores práticas como é visto utilizando o SAM.

6.3.3 AWS Cloudfront

O AWS Cloudfront é um serviço de rede de entrega de conteúdo *Content Delivery Network* (CDN) criado para se ter alta performance, segurança e conveniência para o desenvolvedor (AMAZON, 2022d). Com o Cloudfront é possível ter um *site* no ar por toda a *internet*, com uma alta taxa de disponibilidade e sem precisar fazer configurações complexas, o que é bem conveniente para se ter nesse trabalho.

6.3.4 React JS

O *React JS* é uma biblioteca *JavaScript* para criar interfaces de usuário (REACT, 2022). O *React* tem sido a nova tendência na criação de interfaces de usuários, porém, por falta de conhecimento na utilização da linguagem, o presente trabalho não foi construído utilizando a mesma.

REFERÊNCIAS

ACADEMY, D. **Data as a Service (DaaS) – Benefícios e Tendências**.[S.I:s.n]. 2022. Disponível em: https://blog.dsacademy.com.br/data-as-a-service-daas-beneficios-e-tendencias/. Acesso em: 21 fev. 2022.

ALPHACODES. **Complete Guide to Monolithic vs. Microservices Architecture**.[S.I:s.n]. 2021. Disponível em: https://faun.pub/complete-guide-to-monolithic-vs-microservices-architecture-fe1858c2cfef. Acesso em: 16 dez. 2021.

AMAZON. **Amazon AWS**.[S.I:s.n]. 2008. Disponível em: https://aws.amazon.com/pt/. Acesso em: 19 mar. 2020.

AMAZON. **AWS Lambda**.[S.I:s.n]. 2014. Disponível em: https://aws.amazon.com/pt/lambda/. Acesso em: 19 mar. 2020.

AMAZON. **Serverless computing and applications**.[S.I:s.n]. 2020. Disponível em: https://aws.amazon.com/pt/serverless. Acesso em: 15 mar. 2020.

AMAZON. **API Gateway**.[S.I:s.n]. 2020. Disponível em: https://aws.amazon.com/pt/api-gateway/. Acesso em: 19 mar. 2020.

AMAZON. **S3 Simple Storage Service**.[S.I:s.n]. 2020. Disponível em: https://aws.amazon.com/pt/s3/. Acesso em: 19 mar. 2020.

AMAZON. **Amazon Simple Notification Service (SNS)**.[S.I:s.n]. 2020. Disponível em: https://aws.amazon.com/pt/sns/. Acesso em: 5 mai. 2020.

AMAZON. **Amazon Cognito**.[S.I:s.n]. 2020. Disponível em: https://aws.amazon.com/pt/cognito/. Acesso em: 30 mar. 2020.

AMAZON. **Amazon Relational Database Service (RDS)**.[S.I:s.n]. 2020. Disponível em: https://aws.amazon.com/pt/rds/. Acesso em: 4 mai. 2020.

AMAZON. **Computação em nuvem com a AWS**.[S.I:s.n]. 2021. Disponível em: https://aws.amazon.com/pt/what-is-aws/. Acesso em: 17 dez. 2021.

AMAZON. **AWS Identity and Access Management (IAM)**.[S.I:s.n]. 2021. Disponível em: https://aws.amazon.com/pt/iam/. Acesso em: 19 dez. 2021.

AMAZON. **Definição de preço do AWS Lambda**.[S.I:s.n]. 2022. Disponível em: https://aws.amazon.com/pt/lambda/pricing/. Acesso em: 9 fev. 2022.

AMAZON. **AWS Amplify**.[S.I:s.n]. 2022. Disponível em: https://aws.amazon.com/pt/amplify/. Acesso em: 04 fev. 2022.

AMAZON. **AWS Serverless Application Model**.[S.I:s.n]. 2022. Disponível em: https://aws.amazon.com/pt/serverless/sam/. Acesso em: 04 fev. 2022.

AMAZON. **AWS CloudFront**.[S.I:s.n]. 2022. Disponível em: https://aws.amazon.com/pt/cloudfront/. Acesso em: 05 fev. 2022.

ARMBRUST, M.; FOX, A.; GRIFFITH, R.; JOSEPH, A. D.; KATZ, R. H.; KONWINSKI, A.; LEE, G.; PATTERSON, D. A.; RABKIN, A.; STOICA, I.; ZAHARIA, M. **Above the Clouds**: A Berkeley View of Cloud Computing.[S.I:s.n]. 2009. Disponível em: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-28.html. Acesso em: 15 mar. 2020.

BUYYA, R.; YEO, C. S.; VENUGOPAL, S. Market-oriented cloud computing: Vision, hype, and reality for delivering it services as computing utilities. In: IEEE. **2008 10th IEEE international conference on high performance computing and communications**. [S.l.], 2008. p. 7–15.

CHIRIGATI, F. S. **Computação em Nuvem.**[S.I:s.n]. 2009. Disponível em: https://www.gta.ufrj.br/ensino/eel879/trabalhos_vf_2009_2/seabra/arquitetura.html. Acesso em: 17 mar. 2020.

FERNANDES, D. **Serverless:** Quando utilizar e aplicações com NodeJS.[S.I:s.n]. 2019. Disponível em: https://blog.rocketseat.com.br/serverless-nodejs-lambda/. Acesso em: 18 mar. 2020.

FLARE, C. **What Is Serverless Computing? |** Serverless Definition.[S.I:s.n]. 2020. Disponível em: https://www.cloudflare.com/learning/serverless/what-is-serverless/. Acesso em: 18 mar. 2020.

FOULADI, S.; WAHBY, R. S.; SHACKLETT, B.; BALASUBRAMANIAM, K. V.; ZENG, W.; BHALERAO, R.; SIVARAMAN, A.; PORTER, G.; WINSTEIN, K. Encoding, fast and slow:low-latency video processing using thousands of tiny threads. In: **14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)**. [S.l.: s.n.], 2017. p. 363–376.

FRANZ, J.; Nagasuri, T.; Wartman, A.; Ventrella, A. V.; Esposito, F. **Reunifying Families after a Disaster via Serverless Computing and Raspberry Pis**. In: . [S.l.: s.n.], 2018.

FURHT, B. Cloud computing fundamentals. In: **Handbook of cloud computing**. [S.l.]: Springer, 2010.

GOOGLE. **Google Cloud Functions**.[S.I:s.n]. 2020. Disponível em: https://cloud.google.com/functions?hl=pt-br. Acesso em: 19 mar. 2020.

GUNARATNE, I. **Adapting Serverless Architecture**.[S.I:s.n]. 2022. Disponível em: https://dzone.com/articles/adapting-serverless-architecture. Acesso em: 10 fev. 2022.

IBM. **IBM Cloud**.[S.I:s.n]. 2020. Disponível em: https://www.ibm.com/watson/developercloud/. Acesso em: 19 mar. 2020.

IDEALMARKETING. **O QUE É COMPUTAÇÃO EM NUVEM?**.[S.I:s.n]. 2018. Disponível em: https://www.idealmarketing.com.br/blog/o-que-e-computacao-em-nuvem/. Acesso em: 20 mar. 2020.

IRON.IO. **Iron.io IronFunctions**. [S.I:s.n]. 2020. Disponível em: http://open.iron.io/. Acesso em: 19 mar. 2020.

JONAS, E.; SCHLEIER-SMITH, J.; SREEKANTI, V.; TSAI, C.-C.; KHANDELWAL, A.; PU, Q.; SHANKAR, V.; CARREIRA, J. M.; KRAUTH, K.; YADWADKAR, N.; GONZALEZ, J.; POPA, R. A.; STOICA, I.; PATTERSON, D. A. **Cloud Programming Simplified:** A Berkeley View on Serverless Computing.[S.I:s.n]. 2019. Disponível em: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2019/EECS-2019-3.html. Acesso em: 20 nov. 2021.

KLEINROCK, L. **A Vision for the Internet**. v. 2, n. 1, November 2005.

KOLLER, E. **Serverless:** você conhece os benefícios dessa arquitetura?.[S.I:s.n]. 2022. Disponível em: https://blog.bossabox.com/serverless-beneficios-arquitetura/. Acesso em: 10 fev. 2022.

LENON. **Node.js – O que é, como funciona e quais as vantagens**.[S.I:s.n]. 2021. Disponível em: https://www.opus-software.com.br/node-js/. Acesso em: 19 dez. 2021.

MCGRATH, G.; Brenner, P. R. **Serverless Computing:** design, implementation, and performance. In: . [S.l.: s.n.], 2017.

MELL, P.; GRANCE, T. *et al.* **The NIST definition of cloud computing**. Computer Security Division, Information Technology Laboratory, National Institute of Standards and Technology Gaithersburg, 2011.

MICROSOFT. **Azure Functions**.[S.I:s.n]. 2020. Disponível em: https://azure.microsoft.com/en-us/services/functions/. Acesso em: 19 mar. 2020.

NOVKOVIC, N. **What Are AWS Lambda Triggers?** (Explained for Dummies).[S.I:s.n]. 2018. Disponível em: https://dashbird.io/blog/what-are-aws-lambda-triggers/. Acesso em: 19 mar. 2020.

OPENFACE. **OpenFace**.[S.I:s.n]. 2020. Disponível em: https://cmusatyalab.github.io/openface/. Acesso em: 19 mar. 2020.

OPENLAMBDA. **OpenLambda**.[S.I:s.n]. 2020. Disponível em: https://open-lambda.org/. Acesso em: 19 mar. 2020.

OPENNEBULA. **OpenNebula**.[S.I:s.n]. 2020. Disponível em: https://opennebula.io/. Acesso em: 24 mar. 2020.

OPENSTACK. **OpenStack**.[S.I:s.n]. 2020. Disponível em: https://www.openstack.org/. Acesso em: 24 mar. 2020.

OPENWHISK, A. **Apache OpenWhisk**.[S.I:s.n]. 2020. Disponível em: http://openwhisk.apache.org/. Acesso em: 19 mar. 2020.

PÉREZ, A.; RISCO, S.; NARANJO, D. M.; CABALLER, M.; MOLTÓ, G. On-premises serverless computing for event-driven data processing applications. In: IEEE. **2019 IEEE 12th International Conference on Cloud Computing (CLOUD)**. [S.l.], 2019. p. 414–421.

PIRES, J. **O que é API? REST e RESTful?** Conheça as definições e diferenças!.[S.I:s.n]. 2021. Disponível em: https://becode.com.br/o-que-e-api-rest-e-restful/. Acesso em: 20 dez. 2021.

REACT. **React**.[S.I:s.n]. 2022. Disponível em: https://pt-br.reactjs.org/. Acesso em: 05 fev. 2022.

RUSCHEL, H.; ZANOTTO, M. S.; MOTA, W. d. Computação em nuvem. **Pontifícia Universidade Católica do Paraná, Curitiba, Brazil**, 2010.

SERVERLESS. **Do more with less. Serverless.**.[S.I:s.n]. 2021. Disponível em: https://www.serverless.com/. Acesso em: 19 dez. 2021.

SOUSA, F. R.; MOREIRA, L. O.; MACHADO, J. C. Computação em nuvem: Conceitos, tecnologias, aplicações e desafios. **II Escola Regional de Computação Ceará, Maranhão e Piauí (ERCEMAPI)**, p. 150–175, 2009.

VAN EYK, E.; Toader, L.; Talluri, S.; Versluis, L.; Uță, A.; Iosup, A. Serverless is more: From paas to present cloud computing. **IEEE Internet Computing**, v. 22, n. 5, p. 8–17, Sep. 2018.

Vázquez-Poletti, J. L.; Llorente, I. M. Serverless computing: From planet mars to the cloud. **Computing in Science Engineering**, v. 20, n. 6, p. 73–79, Nov 2018. ISSN 1558-366X.

WAZLAWICK, R. **Engenharia de software: conceitos e práticas**. [S.l.]: Elsevier Editora Ltda., 2019. Acesso em: 5 mai. 2020.

YAN, M.; CASTRO, P.; CHENG, P.; ISHAKIAN, V. Building a chatbot with serverless computing. In: **Proceedings of the 1st International Workshop on Mashups of Things and APIs**. [S.l.: s.n.], 2016.

APÊNDICE A – LISTA DAS FUNÇÕES MICROSSERVIÇOS E A FORMA DE UTILIZAÇÃO

Tabela 4 – Tabela de funções

Funções	Método	Descrição
Login	POST	Login na aplicação web
Cadastro	POST	Cadastro na aplicação web
SMS	GET	Envia para o usuário um SMS informando-o para verificar o e-mail
Objeto Criado	GET	Após o arquivo ser armazenado no repositório enviar os dados para o banco de dados
Upload	PUT	Upload do arquivo para o repositório
Download	GET	Download do arquivo armazenado no repositório
Visualizar dados	GET	Retornar para o usuário as informações referentes aos arquivos armazenados no repositório

Fonte: Elaborado pelo autor.

Tabela 5 – Tabela de invocações das funções

Função	Invocação direta	Orientado a eventos
Upload	X	-
Download	X	-
Objeto Criado	-	X
Visualizar dados		X
Cadastro	X	-
Login	X	-
SMS	-	X

Fonte: Elaborado pelo autor.

APÊNDICE B – TRECHO DE CÓDIGO DA CONFIGURAÇÃO SERVERLESS.YML

```
service: bibliotecaserverless
plugins:
  - serverless-dotenv-plugin
  - serverless-stack-output
  - serverless-s3-remover
useDotenv: true
configValidationMode: off

frameworkVersion: "2"

custom:
  bucket: ${env:BUCKET_NAME}
  DBNameRDS: ${env:DB_NAME}
  MasteruserRDS: ${env:USER_NAME_DB}
  PassworduserRDS: ${env:USER_PASSWORD_DB}
  defaultStage: dev
  currentStage: ${opt:stage, self:custom.defaultStage}
  userPoolName: test-user-pool-${self:custom.currentStage}
  userPoolClientName: test-user-pool-client-${self:custom.currentStage}
  remover:
    buckets:
      - ${self:custom.bucket}
  output:
    file: outputs.toml
provider:
  environment:
    Bucket: ${self:custom.bucket}
    DatabaseName: ${self:custom.DBNameRDS}
    DatabaseUser: ${self:custom.MasteruserRDS}
    DatabasePassword: ${self:custom.PassworduserRDS}
    RDSEndpoint:
      Fn::GetAtt: [ DBinstanceRDSMysql, Endpoint.Address ]
  name: aws
  endpointType: REGIONAL
  runtime: nodejs14.x
  stage: dev
  region: us-east-1
```

```
38    memorySize: 512
39    lambdaHashingVersion: 20201221
40    iamRoleStatements:
41      - Effect: Allow
42        Action:
43          - s3:GetObject
44          - s3:PutObject
45          - s3:ListBucket
46          - s3:ListObject
47        Resource:
48          - arn:aws:s3:::${self:custom.bucket}/*
49      - Effect: Allow
50        Action:
51          - sns:Publish
52          - sns:Subscribe
53        Resource: "*"
54  functions:
55    objetocriado:
56      handler: handler.objetocriado
57      events:
58        - s3:
59            bucket: ${self:custom.bucket}
60            event: s3:ObjectCreated:*
61            private: false
62            cors: true
63            rules:
64              - prefix: Artigos/
65              - suffix: .pdf
66    listar:
67        handler: handler.listar
68        events:
69          - http:
70              path: api_list
71              method: get
72              integration: lambda
73              private: false
74              cors: true
75  resources:
76    Resources:
```

```
77      DBinstanceRDSMysql:
78        Type: AWS::RDS::DBInstance
79        Properties:
80          DBInstanceIdentifier: BibliotecaServerless
81          DBName: ${self:custom.DBNameRDS}
82          DBInstanceClass: db.t2.micro
83          AllocatedStorage: 20
84          StorageType: gp2
85          Engine: MySQL
86          EngineVersion: 8.0.23
87          MasterUsername: ${self:custom.MasteruserRDS}
88          MasterUserPassword: ${self:custom.PassworduserRDS}
89          PubliclyAccessible: true
90          AvailabilityZone: us-east-1a
91          CACertificateIdentifier: rds-ca-2019
92          EnableCloudwatchLogsExports:
93            - general
94          Tags:
95            - Key: "DBMysql"
96              Value: "Serverless"
97          LicenseModel: general-public-license
98          MaxAllocatedStorage: 30
99          UseDefaultProcessorFeatures: True
100   Outputs:
101     UserPoolId:
102       Description: ID do pool cognito
103       Value:
104         Ref: CognitoUserPool
105     UserPoolClientId:
106       Description: ID do cliente de aplicativo
107       Value:
108         Ref: CognitoUserPoolClient
```

1

[1] https://github.com/HelterL/Repositorio-de-arquivos-Serverless-AWS

APÊNDICE C – TRECHO DE CÓDIGO DO ARQUIVO HANDLER.JS

```
module.exports.consultardb = (event, context, callback) => {
  var mysql = require('mysql');
  var con = mysql.createConnection({
    host     : process.env.RDSEndpoint,
    user     : process.env.DatabaseUser,
    password : process.env.DatabasePassword,
    database : process.env.DatabaseName
   });

    con.connect(function(err) {
      if (err) throw err;
      con.query("SELECT * FROM livro", function (err, result, fields) {
        if (err) throw err;
        callback(null,JSON.parse(JSON.stringify(result)));
        con.end();
      });
    });
};
```

1

[1] https://github.com/HelterL/Repositorio-de-arquivos-Serverless-AWS

APÊNDICE D – TELAS DA APLICAÇÃO

Figura 14 – Tela de Home

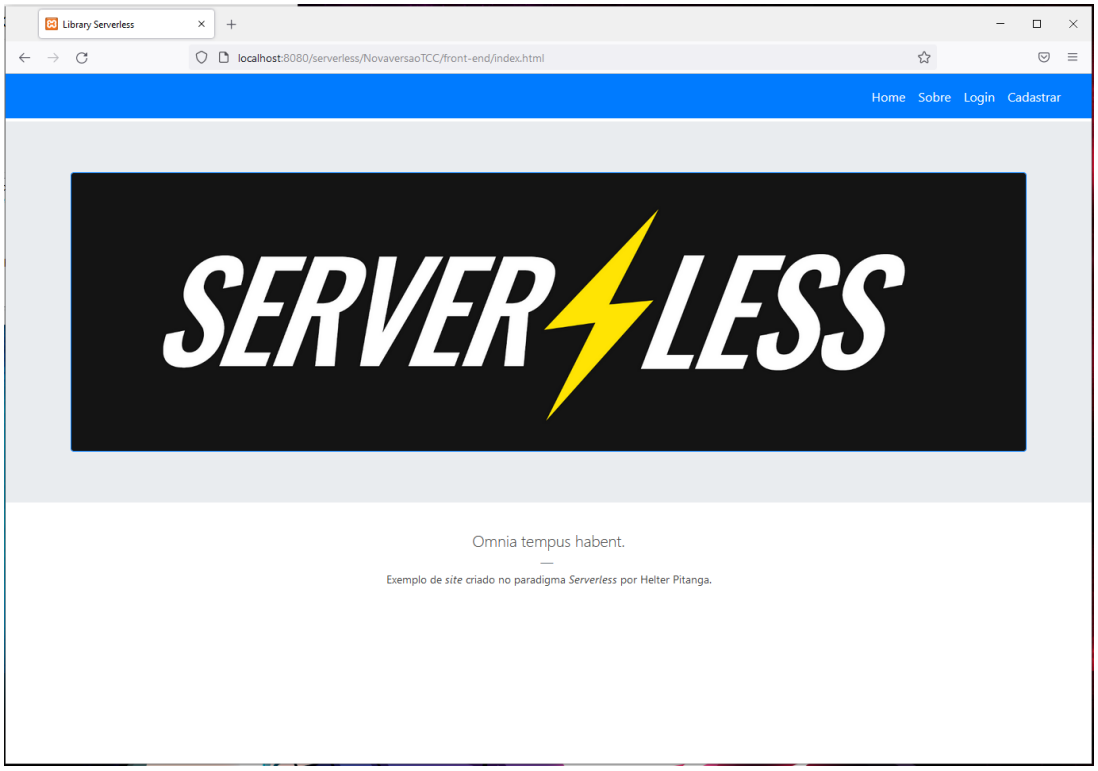


Figura 15 – Tela sobre a aplicação

Figura 16 – Tela de cadastro

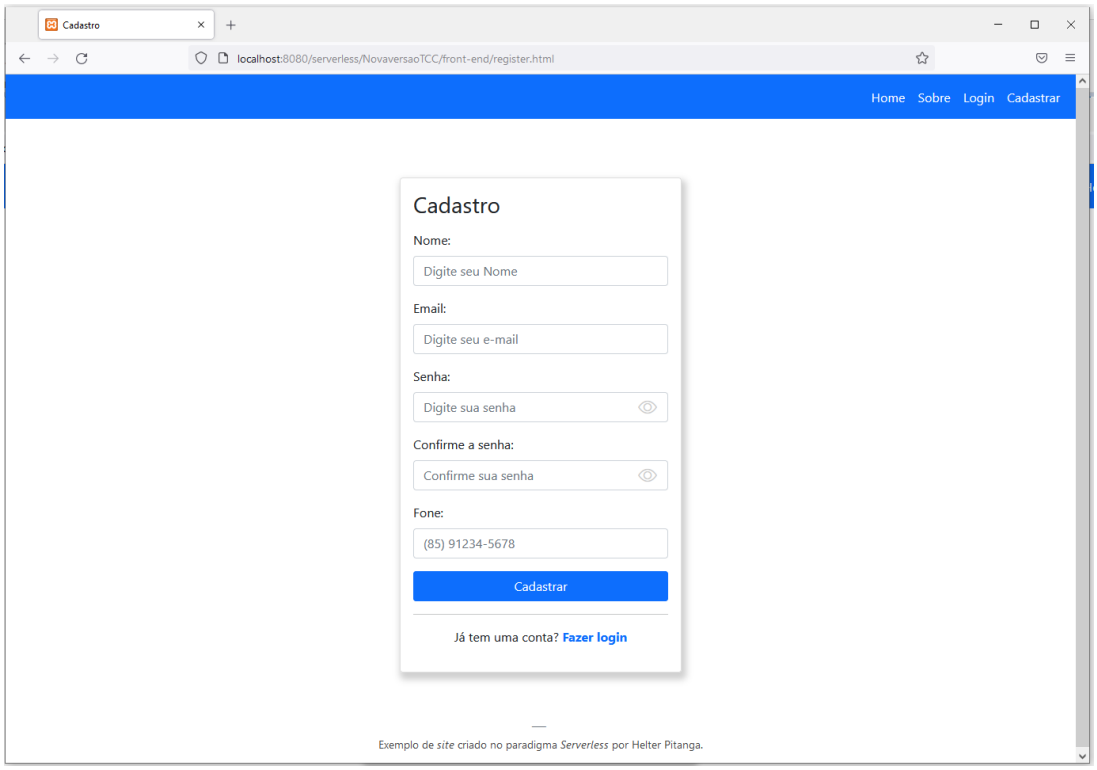


Figura 17 – Tela de login

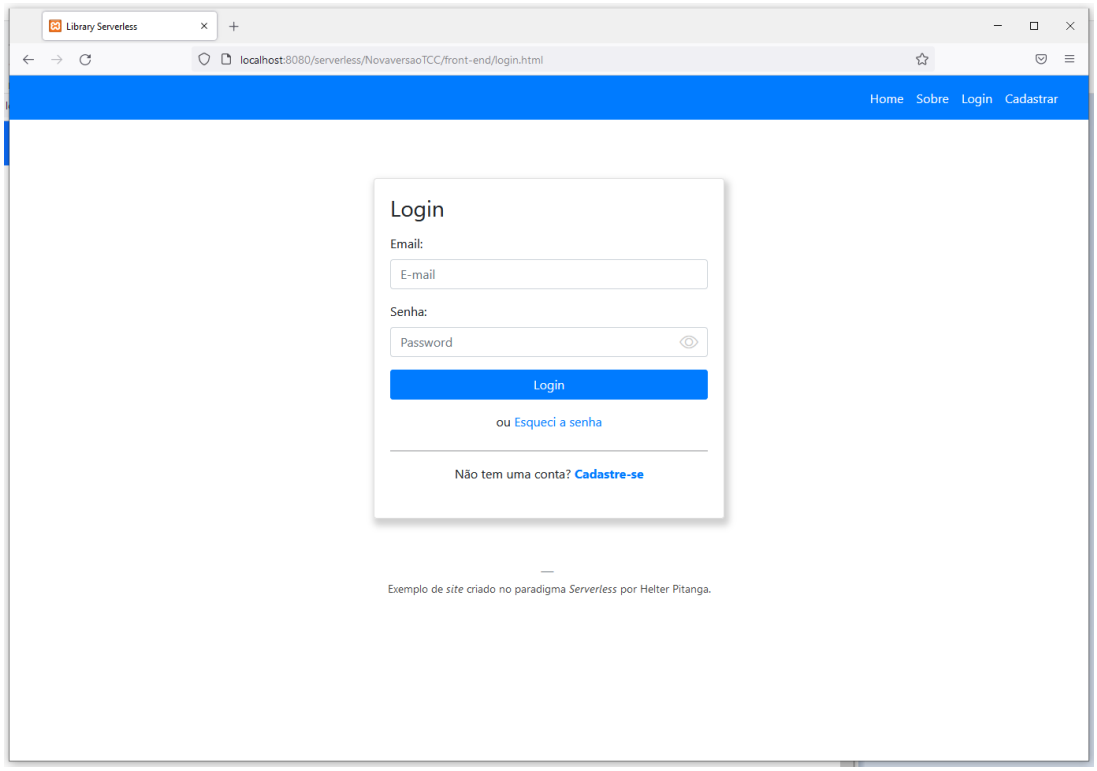


Figura 18 – Tela de upload e visualizaçao de dados