

UNIVERSIDADE FEDERAL DE UBERLÂNDIA
FACULDADE DE GESTÃO E NEGÓCIOS
GRADUAÇÃO EM GESTÃO DA INFORMAÇÃO

BRENDA MENDONÇA SILVA OLIVEIRA

Implementação de Infraestrutura como Código e Integração CI/CD na AWS

ORIENTADOR: PROF. DR. JOSÉ EDUARDO FERREIRA LOPES

UBERLÂNDIA – MG

2024

BRENDA MENDONÇA SILVA OLIVEIRA

Implementação de Infraestrutura como Código e Integração CI/CD na AWS

Monografia apresentada ao Curso de Graduação em Gestão da Informação, da Universidade Federal de Uberlândia, como exigência parcial para a obtenção do título de Bacharel.

Orientador: Prof. Dr. José Eduardo Ferreira Lopes

UBERLÂNDIA – MG

2024

RESUMO

Objetivou-se com este relato tecnológico descrever o processo de implantação de um ambiente automatizado na nuvem, utilizando práticas de DevOps e Infraestrutura como Código para otimizar o desenvolvimento e *deploy* de aplicações. A situação problema abordada foi a necessidade de um ambiente escalável e eficiente para suportar o ciclo de vida de desenvolvimento e operações, reduzindo o trabalho manual e erros humanos. Como solução, foi implementada uma arquitetura que combina Lambda, S3, DynamoDB e API Gateway, provisionados via Terraform e integrados em uma pipeline de CI/CD no GitHub Actions. Os resultados esperados incluem uma estrutura de infraestrutura na nuvem e uma aplicação executável. Foram analisados os tempos necessários para a criação automatizada da estrutura e os benefícios na adoção da Infraestrutura como Código.

Palavras-chave: Arquitetura; Automação; Nuvem.

LISTA DE FIGURAS

Figura 1 - Exemplo	10
Figura 2 - Arquitetura	14
Figura 3 - Estrutura de pastas	15
Figura 4 - Pipeline.....	15
Figura 5 - Jobs.....	16
Figura 6 - Plan.....	16
Figura 7 - Apply.....	17
Figura 8 - Tabela DynamoDB	18
Figura 9 - Lambda.....	18
Figura 10 - API Gateway	19
Figura 11 - Bucket	19

SUMÁRIO

1. INTRODUÇÃO.....	6
2. FUNDAMENTAÇÃO TEÓRICA	7
2.1. DevOps.....	7
2.2. Terraform	9
2.3. Amazon Web Service.....	11
2.4. Automação e deploy	12
3. CONTEXTO INVESTIGADO E SITUAÇÃO PROBLEMA	13
5. RESULTADOS ALCANÇADOS.....	17
6. CONSIDERAÇÕES FINAIS	19
7. REFERÊNCIAS BIBLIOGRÁFICAS	21

1. INTRODUÇÃO

Qualquer ambiente de aplicação necessita de diversos componentes de infraestrutura, como máquinas virtuais, redes, conexões de banco de dados e armazenamento. A Infraestrutura como Código (IaC) representa a capacidade de provisionar e manter essa infraestrutura de computação por meio de código (Red Hat, 2022).

Historicamente, o provisionamento de infraestrutura era um processo manual, dispendioso e demorado. No entanto, a ascensão da computação em nuvem através da criação de camadas virtuais e da padronização de ambientes, eliminou a dependência do hardware físico, trazendo consigo um cenário de infraestrutura exponencialmente mais complexo, com o aumento do número de componentes e exigindo que a infraestrutura fosse flexível para se adaptar às frequentes mudanças e escalonamentos. Sem a adoção da prática de IaC, o gerenciamento da infraestrutura em grande escala se torna um desafio crescente (Red Hat, 2022).

A IaC revoluciona o gerenciamento de TI, dotando as organizações de agilidade, padronização e confiabilidade incomparáveis. Através da codificação da infraestrutura, as empresas podem automatizar a criação e configuração de seus ambientes, eliminando tarefas manuais repetitivas e propensas a erros. Com a IaC, os *templates* de provisionamento definem as configurações exatas de cada ambiente, desde servidores e sistemas operacionais até firewalls e outros componentes. Essa abordagem garante consistência impecável em todos os ambientes, desde o desenvolvimento até a produção, minimizando falhas e otimizando o desempenho (Red Hat, 2022).

A sinergia entre IaC e DevOps é a chave para alcançar um novo patamar de agilidade e confiabilidade no processo de desenvolvimento. Com scripts simples, a infraestrutura é preparada rapidamente, liberando tempo para os desenvolvedores se concentrarem em suas principais funções: criar e inovar. A combinação de IaC e DevOps promove um alinhamento perfeito entre as equipes de desenvolvimento e operações, garantindo a comunicação, a colaboração e a resolução de problemas, promovendo um ambiente de trabalho mais eficiente e produtivo (Red Hat, 2022).

Nesse contexto, objetiva-se com esse relato tecnológico, descrever o processo de implementação de um ambiente de TI utilizando os conceitos e técnicas de Infraestrutura como Código (IaC) e práticas de DevOps, com o objetivo de otimizar a eficiência e agilidade no provisionamento, gerenciamento e deploy de infraestrutura.

Por meio da automação de processos e integração contínua, este estudo pretende demonstrar como a aplicação dessas metodologias pode melhorar a consistência, reduzir erros e acelerar o ciclo de vida do desenvolvimento de software.

2. FUNDAMENTAÇÃO TEÓRICA

Neste capítulo, por meio de uma pesquisa em referências literárias, será construída uma base teórica para embasar o conceito de IaC. Serão abordados os conceitos de DevOps e a sinergia com IaC para entregas contínuas, a apresentação de uma ferramenta tecnológica chamada Terraform que provisiona e gerencia a IaC e uma introdução à Amazon Web Service (AWS), uma plataforma de computação em nuvem abrangente que oferece uma ampla gama de serviços para provisionar, gerenciar e escalar infraestrutura com eficiência e segurança. Além disso, será discutida a automação de processos de *deploy* e a criação de *pipelines* de CI/CD, que são fundamentais para garantir a integração contínua e a eficiência no ciclo de vida do desenvolvimento.

2.1. DevOps

DevOps não é o nome de uma equipe, de um cargo ou de uma tecnologia específica. Em vez disso, é um conjunto de processos, ideias e técnicas. Em outras palavras, DevOps é mais do que uma simples tendência, é uma transformação cultural que redefine a forma como as empresas desenvolvem, entregam e gerenciam seus softwares. Os princípios fundamentais do DevOps incluem a colaboração e comunicação entre equipes, a automação de processos manuais, e a obtenção de feedback contínuo. Essa abordagem busca criar uma mentalidade de responsabilidade compartilhada, onde todos os membros da equipe se comprometem com a qualidade e o sucesso do software (Brikman, 2019).

No cenário da tecnologia da informação, costuma haver uma divisão entre equipes de Desenvolvimento de Software ("Dev"), responsáveis pela criação de softwares, e equipes de Operações ("Ops"), focadas no gerenciamento de infraestrutura. Historicamente, o fluxo de trabalho era marcado pela entrega de aplicações finalizadas pela equipe de Dev para a equipe de Ops, que por sua vez, assumia a responsabilidade manual de implementação e execução (Brikman, 2019).

Parte dessa necessidade de trabalho manual era inevitável, devido à natureza física da infraestrutura (por exemplo, instalação de servidores em rack e cabeamento de rede). No entanto, mesmo tarefas de software, como a instalação de aplicativos e dependências, frequentemente dependiam da execução manual de comandos em servidores (Brikman, 2019).

Embora esse modelo funcione em seus estágios iniciais, ele apresenta desafios à medida que a empresa cresce. Hoje, as organizações que adotam os princípios e práticas de DevOps frequentemente implementam alterações centenas ou até milhares de vezes por dia independente da área em que atua e que está competindo, o modo como adquire clientes e entrega valores a eles é dependente do fluxo de valor da tecnologia (Kim et al., 2018).

“The goal of DevOps is to make software delivery vastly more efficient.” (Brikman, 2019, p. 23)

Como exemplo de como a adoção de práticas de DevOps pode transformar significativamente a capacidade de uma organização de entregar valor rapidamente e com alta qualidade, Kim et al. (2018) apresenta o caso da Salesforce, que começou em 2000 como uma plataforma de CRM, enfrentou grandes desafios na entrega de novas funcionalidades para seus clientes ao longo dos anos. Em 2007, a empresa estava lutando para manter o ritmo com a demanda, realizando apenas um grande lançamento de funcionalidades naquele ano, comparado a quatro em 2006, mesmo após contratar mais engenheiros. O processo de lançamento era cada vez mais demorado e problemático, o que levou a uma crescente insatisfação.

Para resolver esses problemas, a Salesforce iniciou uma transformação significativa em 2009, adotando práticas de DevOps. Essa transformação foi marcada pela mudança de um processo de desenvolvimento em cascata para um modelo de entrega mais incremental e ágil. Como resultado, a Salesforce conseguiu reduzir o tempo necessário para implementações de seis dias para apenas cinco minutos até 2013.

Um aspecto da transformação foi a integração de testes automatizados em todas as fases do desenvolvimento e da operação. A Salesforce criou uma ferramenta open source chamada Rouser para testes funcionais e começou a realizar testes destrutivos, que envolvem colocar os serviços sob cargas extremas até falharem. Isso ajudou a equipe a entender melhor os modos de falha e a melhorar a qualidade dos serviços.

Além disso, a empresa promoveu uma colaboração mais estreita entre equipes de engenharia, operações e segurança da informação, integrando ferramentas de segurança no processo de testes automatizados. Com essas mudanças, a Salesforce conseguiu melhorar a qualidade do serviço e a eficiência operacional, reduzindo a necessidade de aprovação para alterações na infraestrutura e incentivando a automação.

2.2. Terraform

Com o advento do movimento DevOps, desenvolvido pela empresa HashiCorp, o Terraform permite a definição da infraestrutura de forma modular e reutilizável. Com ele, é possível implantar e gerenciar infraestrutura em diversos provedores de nuvem (Brikman, 2019). A ferramenta utiliza sua própria linguagem declarativa de configuração, denominada HCL (HashiCorp Configuration Language), em arquivos com a extensão `.tf`.

Para executar o arquivo Terraform, no diretório onde contém o arquivo, o primeiro comando a ser executado é o *terraform init*. Ele inicializa o ambiente de trabalho, preparando-o para o uso do Terraform. Durante a inicialização, o Terraform baixa os *plugins* necessários para interagir com os provedores de nuvem especificados nos arquivos de configuração (Brikman, 2019).

Em seguida, O comando *terraform plan* gera um plano de execução, mostrando quais mudanças serão feitas na infraestrutura para alcançar o estado desejado. Ele compara a infraestrutura atual (definida no arquivo de estado) com as configurações presentes nos arquivos `.tf`, exibindo uma lista de adições, modificações ou remoções de recursos (Brikman, 2019).

O comando *terraform apply* aplica as mudanças planejadas e definidas pelo *terraform plan*. Ou seja, ele cria, modifica ou remove os recursos declarados. Por exemplo, ao executar *terraform apply* após configurar um servidor EC2 na AWS, o Terraform criará a instância real na nuvem (Brikman, 2019).

Cada vez que o arquivo é executado ele registra informações sobre a infraestrutura criada em um arquivo chamado *state*. Este arquivo contém um formato JSON personalizado que registra um mapeamento dos recursos do Terraform, informações detalhadas, como IDs de instâncias e endereços IPs (Brikman, 2019).

Por fim, existe um comando que permite remover toda a infraestrutura gerenciada pelo Terraform. Ele destrói todos os recursos definidos nos arquivos de configuração, sendo especialmente útil para eliminar ambientes temporários ou de teste. Por exemplo, se um ambiente de desenvolvimento foi provisionado com máquinas virtuais, bancos de dados e redes, o *terraform destroy* removerá todos esses componentes de forma automatizada, liberando os recursos de nuvem e evitando custos desnecessários (Brikman, 2019).

Um exemplo é apresentado na Figura 1. O Terraform é utilizado para criar uma instância EC2 na AWS. O bloco *provider* define a região em que os recursos serão provisionados, que no caso é "us-east-1". O recurso *aws_instance* é o responsável por definir a instância EC2.

Figura 1 - Exemplo

```
provider "aws" {  
  region = "us-east-1"  
}  
resource "aws_instance" "example" {  
  ami = "ami-0c55b159cbfaffe1f0" # ID da imagem AMI  
  instance_type = "t2.micro" # Tipo da instância  
  tags = {  
    Name = "Exemplo-EC2"  
  }  
}
```

Fonte: Autor

Na configuração da instância, o campo “ami” refere-se à Amazon Machine Image (AMI), que é a imagem que define o sistema operacional e as configurações básicas da instância. Neste caso, foi utilizada a ID da imagem AMI para o Amazon Linux 2. O campo “instance_type” define o tipo de instância, neste exemplo, “t2.micro”, que é elegível para o nível gratuito da AWS e é uma opção de baixo custo. Além disso, o bloco “tags” adiciona uma etiqueta para facilitar a identificação da instância, nomeando-a como "Exemplo-EC2" (Hashcorp, 2024).

Esse código pode ser utilizado em combinação com os comandos mencionados anteriormente. Após rodar *terraform init* para inicializar o ambiente e *terraform plan* para

visualizar o plano de execução, o comando *terraform apply* criará a instância EC2 na AWS e gerará ou atualizará o arquivo de estado que contém detalhes sobre os recursos provisionados. Caso a infraestrutura não seja mais necessária, o comando *terraform destroy* removerá a instância e liberará os recursos (Brikman, 2019).

2.3. Amazon Web Service

Quando comparada a outras plataformas de nuvem, a Amazon Web Services (AWS) se destaca por sua ampla gama de serviços e maturidade. Ela é reconhecida por abranger tanto soluções tradicionais de infraestrutura, como computação, armazenamento, redes e bancos de dados, quanto tecnologias inovadoras, como aprendizado de máquina, inteligência artificial, Internet das Coisas (IoT) e *lake* de dados (Amazon Web Services, 2024).

Outro ponto importante é a flexibilidade oferecida pela nuvem, permitindo que os usuários acessem e gerenciem seus recursos de qualquer lugar, desde que conectados à internet, o que facilita o trabalho remoto e a colaboração entre equipes distribuídas geograficamente. Além disso, a segurança é um benefício significativo, uma vez que os provedores de nuvem investem pesadamente em soluções de criptografia, autenticação multifator, backup e recuperação de desastres, além de garantirem conformidade com padrões de segurança e regulamentações (Amazon Web Services, 2024).

Para o desenvolvimento deste projeto, utilizaremos os seguintes serviços da AWS:

- AWS Lambda: permite executar código com alta disponibilidade, para qualquer aplicativo ou serviço de *backend* sem provisionar ou gerenciar servidores. O Lambda escala automaticamente as requisições e administra a quantidade de recursos conforme a demanda (Amazon Web Services, 2024).
- Amazon S3 (Simple Storage Service): é ideal para o armazenamento de dados estáticos e arquivos, como imagens, vídeos e backups. Ele oferece uma solução de armazenamento altamente escalável e durável, garantindo que os dados da aplicação sejam armazenados de forma segura e estejam disponíveis quando necessário. O S3 também é útil para armazenar logs e outros arquivos gerados pela aplicação, facilitando a gestão e análise desses dados (Amazon Web Services, 2024).
- Amazon DynamoDB: é um serviço de banco de dados NoSQL totalmente gerenciado, que oferece desempenho rápido e previsível com escalabilidade automática. Ele é

projetado para lidar com grandes volumes de dados e suportar aplicativos com tráfego de solicitações intensivo. Também fornece suporte a recursos como backup, restauração, encriptação e integrações com outros serviços da AWS para análises de dados e processamento em tempo real (Amazon Web Services, 2024).

- Amazon API Gateway: gerencia o roteamento de chamadas de API, controle de tráfego, autenticação e autorização, além de monitoramento e gerenciamento de versões de APIs. Facilita a criação, publicação, manutenção, monitoramento e segurança de APIs em qualquer escala. Permite desenvolvedores conectarem a serviços de *backend*, como aplicações hospedadas no AWS Lambda, instâncias EC2, ou outros serviços da AWS (Amazon Web Services, 2024).

Combinando o AWS Lambda para a execução da lógica da aplicação sem a necessidade de gerenciar servidores, o Amazon S3 para o armazenamento seguro e escalável de arquivos, o Amazon DynamoDB para a gestão eficiente de dados com baixa latência e o API Gateway para a interface de comunicação entre os usuários e os serviços da aplicação, é possível construir uma solução *serverless* e completamente gerenciada na nuvem.

2.4. Automação e deploy

Para garantir um fluxo de trabalho ágil e confiável de desenvolvimento para operações, é necessário criar ambientes de produção automatizados e sob demanda. Isso minimiza o trabalho manual e permite que os ambientes sejam reproduzidos rapidamente com base no controle de versão. Essa prática de automação do ciclo de desenvolvimento, integração, teste e entrega de software possui um nome que é CI/CD, a combinação de duas importantes estratégias: Continuous Integration (CI) e Continuous Delivery (CD).

A integração contínua envolve a integração frequente do código por diferentes desenvolvedores em um repositório central. Cada vez que uma mudança é feita no código, ela é automaticamente testada e integrada ao projeto principal. O objetivo do CI é identificar e corrigir rapidamente erros e incompatibilidades no código, garantindo que novas adições ao projeto sejam compatíveis com as versões anteriores e que o software permaneça estável (Wolff, 2017).

Já a entrega contínua é a prática de automatizar o processo de entrega de software, permitindo que o código aprovado nos testes de CI seja preparado para implantação em

produção a qualquer momento. No entanto, para que o CD funcione adequadamente, é essencial ter uma pipeline de entrega altamente otimizada (Wolff, 2017).

De acordo com Wolff (2017), a pipeline de automação é o conjunto de etapas que leva o código desde o *commit* até sua liberação em produção, passando por fases como compilação, testes, análise de qualidade e, finalmente, a implantação. No contexto de CI/CD, a pipeline garante que todo o processo seja automatizado e padronizado, proporcionando *feedback* rápido e permitindo que o software seja entregue com mais frequência e confiança.

3. CONTEXTO INVESTIGADO E SITUAÇÃO PROBLEMA

O presente trabalho ocorre em uma empresa de grande porte, no setor de atendimento telefônico. A organização está levando ao mercado uma solução que até então era um produto interno, usado para gerenciamento da empresa. Com o objetivo de novas fontes de receita, além do seu serviço principal que é de telefonia, a empresa está constantemente criando produtos e para isso precisa otimizar a entrega de suas soluções para seus clientes. O departamento em questão é o de inovação e novos produtos.

Atualmente, o setor enfrenta desafios relacionados à gestão e automação do processo de provisionamento e *deploy* de ambientes. O processo vigente ainda utiliza abordagens manuais e pouco automatizadas para a criação e gerenciamento de infraestruturas, o que resulta em inconsistências, aumento no tempo de entrega e uma maior incidência de erros durante o *deploy* de aplicações. Essas dificuldades comprometem o fluxo de trabalho do time, gerando retrabalho e atrasos na entrega de soluções aos clientes internos e externos.

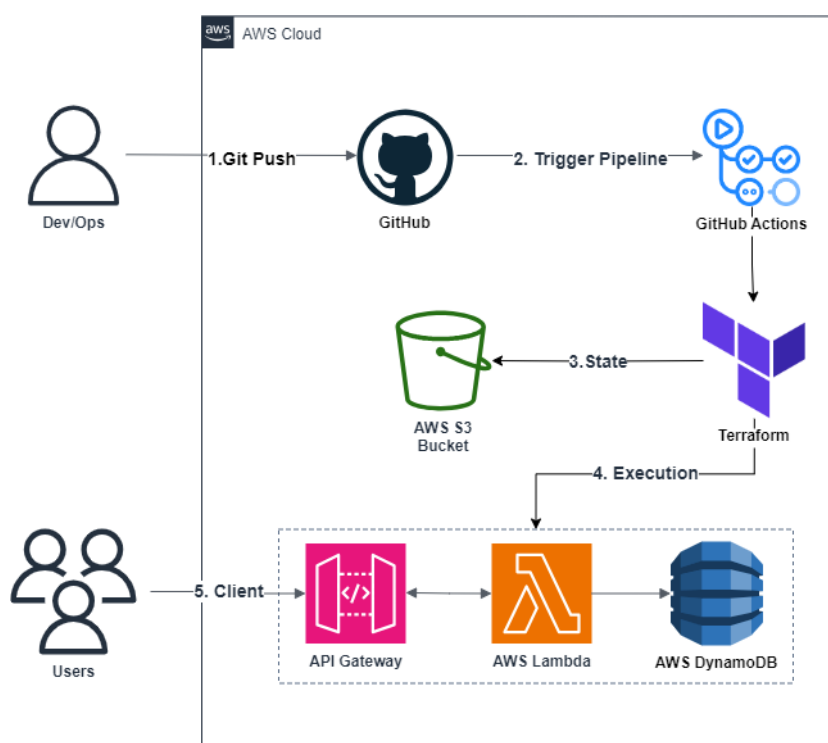
Além disso, a empresa utiliza uma combinação de ferramentas isoladas e scripts manuais, o que dificulta a criação de uma pipeline unificada e automatizada para integração contínua (CI) e entrega contínua (CD). A ausência de uma infraestrutura como código (IaC) bem implementada também aumenta o risco de configurações divergentes entre ambientes de desenvolvimento, homologação e produção. Essa lacuna no processo de automação gera atrasos e reduz a eficiência das equipes, destacando a necessidade de uma solução mais robusta.

Portanto, a situação atual evidencia a necessidade de uma intervenção focada na implementação de práticas mais modernas de DevOps, com a adoção de pipelines automatizadas e soluções de IaC para aumentar a eficiência, reduzir erros e melhorar a entrega contínua de serviços.

4. INTERVENÇÃO ADOTADA

Para concepção da solução foi planejada uma arquitetura (Figura 2) que segmenta a infraestrutura em cinco etapas. O fluxo começa com desenvolvedores (DevOps) realizando um *git push* no repositório do GitHub (etapa 1). Isso aciona uma *pipeline* no GitHub Actions (etapa 2) para automatizar a entrega contínua. O Terraform, então, utiliza o estado armazenado no S3 (etapa 3) para a execução do deploy (etapa 4), que cria e configura os recursos na AWS, como API Gateway, AWS Lambda e DynamoDB, que atendem aos clientes finais (etapa 5).

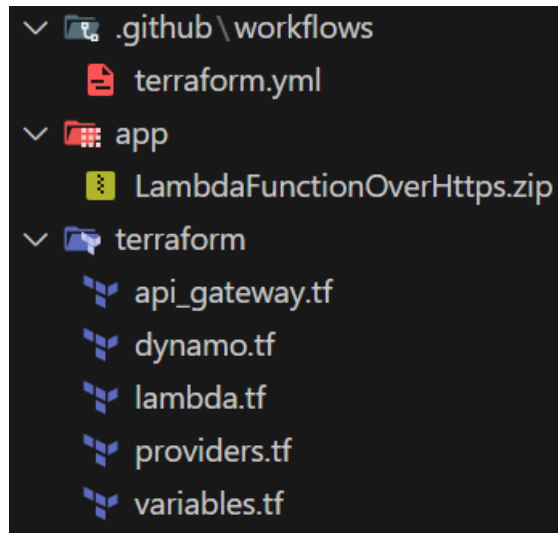
Figura 2 - Arquitetura



Fonte: Autor

A criação do Lambda, DynamoDB e da Api Gateway na plataforma de *cloud* da Amazon são realizadas com a utilização da ferramenta Terraform. O código de cada configuração foi versionado para o repositório Github, tornando-o acessível e executável de qualquer computador com acesso ao repositório nesse link: <https://github.com/brendamendon/TCC>. A estrutura dos arquivos é demonstrada na Figura 3, onde a nomenclatura dos arquivos foi definida pelo autor. O repositório possui uma pasta com os 3 recursos que serão criados com IaC, as variáveis e o provedor. Uma outra pasta contendo a aplicação. E uma pasta contendo a pipeline.

Figura 3 - Estrutura de pastas



Fonte: Autor

Para disparar a criação da estrutura e o deploy da aplicação, o arquivo terraform.yaml será utilizado. Na figura 4, são definidos os *triggers*, ou seja, a pipeline será acionada se houver um *push* ou *pull_request* na *branch* “main” e quaisquer alterações nos paths especificados. Além disso, foi adicionado o gatilho “workflow_dispatch”, que permite disparar manualmente a *pipeline* com um *input action* para escolher entre as ações *apply* ou *destroy*. Estão definidas variáveis de ambiente, como o nome do bucket onde está armazenado o arquivo de estado e as credenciais da conta na plataforma *cloud*.

Figura 4 - Pipeline

```
on:
  workflow_dispatch:
    inputs:
      action:
        description: 'Action to perform (apply or destroy)'
        required: true
        default: 'apply'
  push:
    branches:
      - main
    paths:
      - terraform/**
  pull_request:
    branches:
      - main
    paths:
      - app/**
env:
  TF_LOG: INFO
  AWS_ACCESS_KEY_ID: ${ secrets.AWS_ACCESS_KEY_ID }
  AWS_SECRET_ACCESS_KEY: ${ secrets.AWS_SECRET_ACCESS_KEY }
  BUCKET_TF_STATE: ${ secrets.BUCKET_TF_STATE }
```

Fonte: Autor

Ainda sobre a *pipeline*, na figura 5 é configurado o *checkout* do repositório e a versão do Terraform. Em seguida, há tarefas relacionadas ao Terraform em etapas separadas. Primeiro inicializa o ambiente Terraform de acordo com a configuração do *backend*, verifica a formatação correta dos arquivos e confirma que a configuração é válida.

Figura 5 - Jobs

```
jobs:
  terraform:
    name: "Terraform Infrastructure Change Management"
    runs-on: ubuntu-latest
    defaults:
      run:
        shell: bash
        working-directory: ./terraform

    steps:
      - name: Checkout the repository to the runner
        uses: actions/checkout@v2

      - name: Setup Terraform with specified version on the runner
        uses: hashicorp/setup-terraform@v2
        with:
          terraform_version: 1.3.0

      - name: Terraform init
        id: init
        run: terraform init -backend-config="bucket=$BUCKET_TF_STATE"

      - name: Terraform format
        id: fmt
        run: terraform fmt -check

      - name: Terraform validate
        id: validate
        run: terraform validate
```

Fonte: Autor

Na figura 6, observa-se a configuração da *pipeline* que permite a geração automática de um plano do Terraform. Esse plano é comentado diretamente na *pull request*, oferecendo uma visualização clara das alterações previstas, como criação, modificação ou exclusão de recursos. Esse mecanismo facilita a revisão, pois todos na equipe podem conferir as mudanças sem sair do GitHub.

Figura 6 - Plan

```
name: Terraform plan
id: plan
run: terraform plan -no-color -input=false
continue-on-error: true

uses: actions/github-script@v6
if: github.event_name == 'pull_request'
env:
  PLAN: "${{ steps.plan.outputs.stdout }}"
with:
  script: |
    if (context.issue.number) {
      github.rest.issues.createComment({
        issue_number: context.issue.number,
        owner: context.repo.owner,
        repo: context.repo.repo,
        body: `### Terraform Plan\n<details><summary>Show Plan</summary>\n${process.env.PLAN}\n</details>`
      })
    } else {
      console.log("No issue number, comment not added.");
    }
  }
```

Fonte: Autor

Por fim, na Figura 7, após a validação bem-sucedida do plano, a *pipeline* executa o comando “terraform apply” automaticamente para implementar as alterações na *branch* “main”. Esta etapa é configurada para verificar se o plano foi bem-sucedido antes de avançar, que interrompendo a execução caso haja falha. Além disso, a pipeline permite uma seleção dinâmica de ações para executar a aplicação ou a destruição da infraestrutura. Esse controle é feito pela variável ACTION, que é configurada para receber o valor *apply* ou *destroy*, dependendo da necessidade.

Figura 7 - Apply

```
name: Terraform Plan Status
if: steps.plan.outcome == 'failure'
run: exit 1

name: Terraform Apply or Destroy
if: github.ref == 'refs/heads/main'
run: |
  ACTION="${{ github.event.inputs.action || 'apply' }}"
  echo "Action selected: $ACTION"
  if [ "$ACTION" == "apply" ]; then
    terraform apply -auto-approve -input=false
  elif [ "$ACTION" == "destroy" ]; then
    terraform destroy -auto-approve -input=false
  else
    echo "Invalid action. Use 'apply' or 'destroy'."
    exit 1
  fi
```

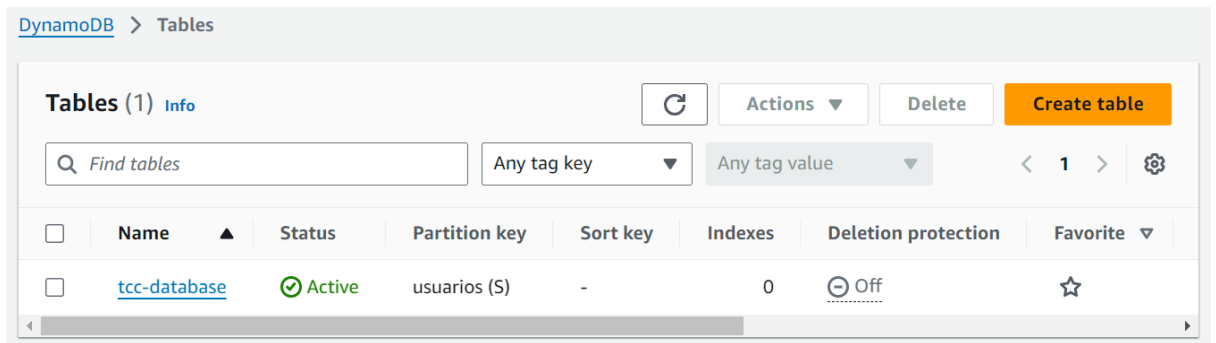
Fonte: Autor

Os arquivos complementares podem ser acessados pelo link fornecido no início deste capítulo. O arquivo localizado na pasta “app” contém o código da aplicação, que está compactado para atender ao formato exigido pelo serviço Lambda. Portanto, qualquer atualização nesse código deve ser seguida de uma nova compactação para manter a compatibilidade. Na pasta “terraform”, encontram-se os arquivos que definem os recursos necessários para o provisionamento dos serviços em *cloud* utilizados neste projeto.

5. RESULTADOS ALCANÇADOS

Conforme esperado, a pipeline executou com sucesso, implantando todos os serviços planejados no Capítulo 4. A Figura 8 apresenta evidências da tabela no DynamoDB, criada com o Terraform.

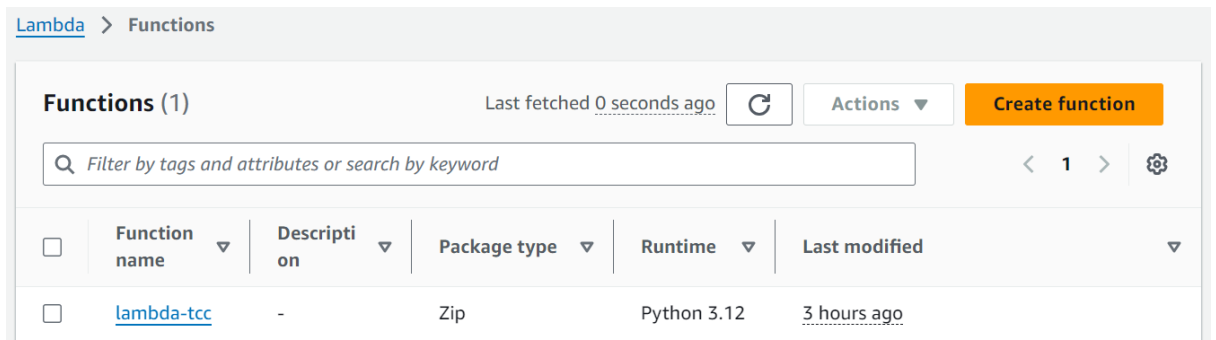
Figura 8 - Tabela DynamoDB



Fonte: Autor

A função Lambda também foi criada com êxito. Para visualizar o código executado e os logs gerados durante sua execução, basta clicar no nome da função no console. Embora o Terraform tenha realizado o deploy do código, alterações diretamente no console não são recomendadas, uma vez que o controle de versionamento é gerenciado exclusivamente pelo GitHub. Portanto, todas as novas funcionalidades desenvolvidas para a aplicação devem ser compactadas novamente e atualizadas através do Terraform, que implantará a nova versão na função Lambda. Para otimizar o processo de atualização do Lambda, uma melhoria na pipeline automatizando o processo de build reduz a sobrecarga de compactação e upload manual, também melhora a eficiência e a consistência do processo de desenvolvimento.

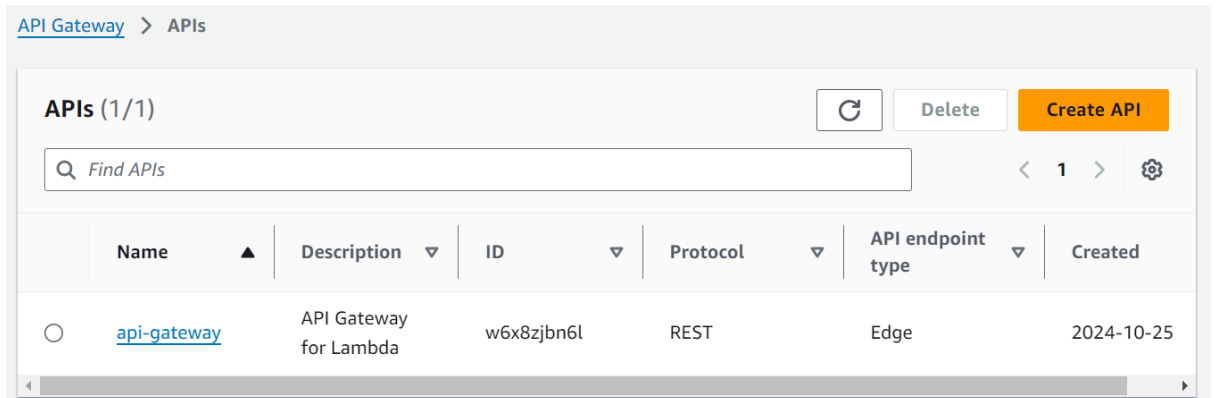
Figura 9 - Lambda



Fonte: Autor

A Figura 10 apresenta a API criada com sucesso. Ela permite que os usuários interajam com a função Lambda através de chamadas HTTP.

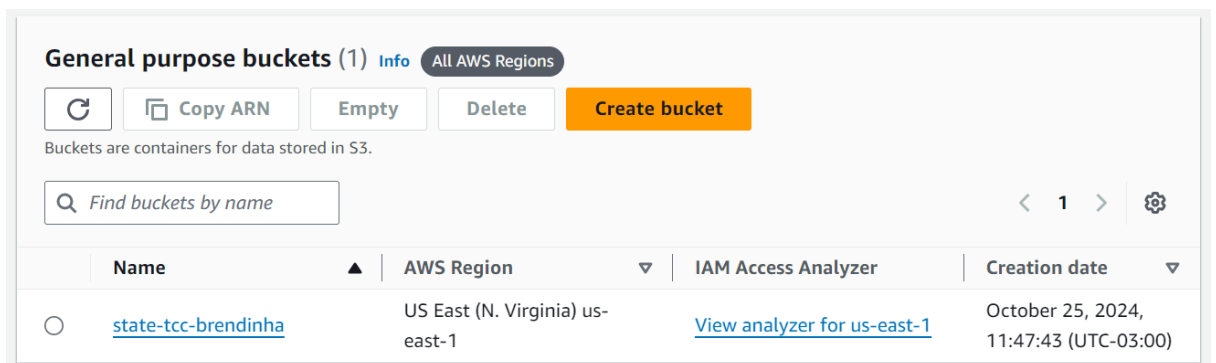
Figura 10 - API Gateway



Fonte: Autor

O bucket foi criado manualmente, porém pode-se notar que o arquivo que o Terraform utiliza para gravar o estado da infraestrutura está presente, como mostra a Figura 11.

Figura 11- Bucket



Fonte: Autor

Por fim, o processo conclui com uma resposta à chamada da função via API, demonstrando a aplicação em pleno funcionamento em uma plataforma cloud, que foi provisionada de forma automatizada em apenas alguns segundos.

Do ponto de vista do negócio, a intervenção trouxe ganhos estratégicos notáveis. A automação dos processos e o uso de pipelines resultaram em uma significativa redução no tempo de entrega e maior previsibilidade nas implantações. Além disso, permite que novos clientes e demandas sejam atendidos de forma rápida e com baixo impacto operacional. Com isso, a empresa ganha uma vantagem competitiva, estando mais preparada para responder rapidamente às mudanças do mercado e aumentar sua capacidade de inovação.

6. CONSIDERAÇÕES FINAIS

Este relato tecnológico teve como objetivo descrever a intervenção focada na implementação de práticas mais modernas de DevOps, com ênfase na adoção de pipelines

automatizadas e soluções de Infraestrutura como Código. O intuito foi otimizar o processo de desenvolvimento e *deployment* de aplicações e serviços.

Durante o processo, houve vários desafios. A complexidade na configuração inicial do Terraform, especialmente na integração com serviços como AWS Lambda e API Gateway, exigiu um aprofundamento técnico significativo. Além disso, criar um código com funcionalidades semelhantes ao produto que a empresa está adotando no mercado exigiu tempo e segurança para que nenhuma propriedade fosse exposta.

Apesar dos desafios, a intervenção revelou várias oportunidades, como a automação dos processos de *build*. A implementação de pipelines automatizadas abre espaço para práticas de Continuous Integration (CI) e Continuous Deployment (CD), que são fundamentais para a evolução contínua da aplicação.

Ficou claro que o uso de ferramentas de automação, como Terraform, demonstrou sua importância na criação de infraestrutura confiável e escalável. Por fim, a importância de uma documentação clara e acessível foi evidenciada, facilitando o entendimento e a adesão às novas práticas por todos que utilizarem.

Em suma, a intervenção focada na implementação de práticas modernas de DevOps, com a adoção de pipelines automatizadas e soluções de IaC, não apenas alcançou os objetivos propostos, mas também proporcionou um ambiente mais colaborativo e eficiente. Os desafios enfrentados foram superados, e as oportunidades identificadas abriram caminho para futuras inovações e melhorias contínuas.

7. REFERÊNCIAS BIBLIOGRÁFICAS

AMAZON WEB SERVICES. **O que é computação em nuvem?** 2024a. Disponível em: <<https://aws.amazon.com/pt/what-is-cloud-computing/>>. Acesso em: 9 set. 2024.

AMAZON WEB SERVICES. **O que é AWS?** 2024b. Disponível em: <https://aws.amazon.com/pt/what-is-aws/>. Acesso em: 9 set. 2024.

AMAZON WEB SERVICES. **Welcome to AWS Documentation.** 2024c. Disponível em: <https://docs.aws.amazon.com/>. Acesso em: 9 set. 2024.

BRIKMAN, Y. **Terraform: Up & Running.** 2. ed. Sebastopol, CA: O'Reilly Media, 2019.

HASHICORP. **Resource: aws_instance.** Disponível em: <https://registry.terraform.io/providers/hashicorp/aws/latest/docs/resources/instance>. Acesso em: 1 set. 2024.

KIM, G.; HUMBLE, J.; DEBOIS, P.; WILLIS, J. **Manual de DevOps: como obter agilidade, confiabilidade e segurança em organizações tecnológicas.** São Paulo: Alta Books, 2018.

RED HAT. **What is infrastructure as code (IaC).** 2024. Disponível em: <https://www.redhat.com/pt-br/topics/automation/what-is-infrastructure-as-code-iac>. Acesso em: 17 jun. 2024.

WOLFF, E. **A Practical Guide to Continuous Delivery.** Addison-Wesley, 2017.