Universidade Federal de Santa Catarina

GitOps: uma nova proposta para a infraestrutura

Roberto Rivelino Ventura da Silva

2020/1

Universidade Federal de Santa Catarina
Centro Tecnológico
Departamento de Informática e Estatística
Curso de Sistemas de Informação

GitOps: uma nova proposta para a infraestrutura

Roberto Rivelino Ventura da Silva

Florianópolis - SC
2020/1

Roberto Rivelino Ventura da Silva

GitOps: uma nova proposta para a infraestrutura

Trabalho de Conclusão de Curso apresentado como parte dos requisitos para obtenção do grau de Bacharel em Sistemas de Informação.

Carla Merkle Westphall
Universidade Federal de Santa Catarina

Banca Examinadora:

Alex Sandro Roschildt Pinto
Universidade Federal de Santa Catarina

Lucas Maltempi Monfardine
Universidade Federal de Santa Catarina

Patrícia Della Méa Plentz
Universidade Federal de Santa Catarina

Agradecimentos

Agradeço aos meus pais e minha irmã, que sempre estiveram comigo ao longo dos anos com palavras de afeto e paciência. À minha namorada Júlia Thomé que sempre me incentivou a continuar, mesmo durante momentos conturbados e de desânimo.

Aos meus amigos, em especial ao Lucas Maltempi que esteve presente comigo em cada etapa deste trabalho como co-orientador.

Aos professores que foram tão importantes durante minha estadia na UFSC, em especial à minha orientadora Carla Merkle Westphall que me auxiliou ao longo deste trabalho e me ajudou a manter a calma durante todo o processo.

Sumário

Lista de ilustrações

Resumo

O presente trabalho propõe-se a desenvolver uma metodologia mais ágil em comparação às operações tradicionais de criação da infraestrutura. Tais operações realizam processos manuais, e o tempo gasto na elaboração dos recursos cresce expressivamente diante da falta de procedimentos automatizados que encurtem o intervalo entre a concepção da infraestrutura e a disponibilidade para o cliente. Utilizou-se então a metodologia *GitOps* como forma de automatizar tais processos. Esta metodologia tem como propósito centralizar a declaração da infraestrutura e das aplicações em uma única fonte (*Git*). Sua principal vantagem é encurtar o tempo e reduzir o custo de transformar uma ideia em produto através das etapas de Integração e Entrega Contínua, tornando o processo como um todo mais otimizado. Inicialmente, o trabalho trata de explicar a cultura *DevOps* e seus conceitos e regras, que são fundamentais para a implantação de uma metodologia *GitOps*. No desenvolvimento, demonstra-se o processo adotado desde seu planejamento até a criação de um ambiente de infraestrutura e de uma aplicação exemplo.

Palavras-chave: Infraestrutura como Código, *DevOps*, Integração Contínua, Entrega Contínua, *GitOps*, *AWS*.

Abstract

This present research aims to develop a more agile metodology when compared to the traditional operations of infrastructure creation. Such operations carry out the processes manually, making the hours spent on elaboration of resources grow significantly due to the lack of automated procedures that could shorten the gap between conception of the infrastructure and the customer's availability. The GitOps metodology was selected as a way of automating such processes. This metodology aims to assemble the infrastructure declaration and applications in a single Git source. Its main advantage is to reduce time and cost of turning an idea into a product using the steps Continuous Integration and Continuous Delivery, making the process as a whole more optimized. Inicially, the research elaborates about DevOps culture and its precepts, which are fundamental to the implementation of a GitOps metodology. Later, the research presents the chosen process step by step, since its planning to the creation of an infrastructure environment and a example of application.

Keywords: Infrastructure as Code, DevOps, Continuous Integration, Continuous Delivery, GitOps, AWS.

Lista de abreviaturas e siglas

VCS Version Control Systems

IaC Infrastructure as Code

CI Continuous Integration

CD Continuous Delivery

AWS Amazon Web Services

GCP Google Cloud Platform

Dev Development

Ops Operations

YAML Yet Another Markup Language

PV Persistent Volume

HTTP Hypertext Transfer Protocol

HCL HashiCorp Configuration Language

EC2 Amazon Elastic Compute Cloud

EKS Amazon Elastic Kubernetes Service

VPC Amazon Virtual Private Cloud

EFS Amazon Elastic Files System

RDS Amazon Relational Database Service

ELB Amazon Elastic Load Balancing

S3 Amazon Simples Storage Service

NLB Network Load Balancer

VPN Virtual Private Network

NAT Network Address Translation

IP Internet Protocol

SG Security Groups

CPU Central Process Unit

IA Infrequent Access

SQL Structured Query Language

ALB Application Load Balancers

GWLB Gateway Load Balancers

OSI Open System Interconnection

HTTPS Hyper Text Transfer Protocol Secure

TCP Transmission Control Protocol

1 Introdução

1.1 Motivação

Em meados de 2000, devido aos avanços na tecnologia e na adoção de princípios e práticas ágeis, o tempo necessário para desenvolver novas funcionalidades de *software* foi reduzido para semanas ou meses. Entretanto, o tempo gasto para implementar essas novas funcionalidades em produção continuou o mesmo (KIM et al., 2016).

Esse fenômeno criou um mercado próprio, com demandas que não poderiam ser satisfeitas através de produtos tradicionais. Para prestar um serviço ágil, empresas precisaram melhorar seus processos e adotar práticas de automação que tornassem possível cortar gastos, tempo de desenvolvimento e implementação de um novo recurso (GONZALEZ, 2017).

Com a introdução do *DevOps* (ação conjunta entre Desenvolvedores e Operação) e a comoditização de *hardware*, *software* e nuvem, recursos passaram a ser criados em semanas, sendo rapidamente implantados em produção em apenas algumas horas ou minutos. Para as organizações, a implantação finalmente tornou-se uma rotina de baixo risco (KIM et al., 2016).

Na cultura *DevOps*, há inúmeras ferramentas de código aberto, como por exemplo: *Subversion* para Sistema de Controle de Versão, *Jenkins* para integração contínua, *Sonar* para gerenciamento e qualidade de projetos, *Nexus* para gerenciamento de repositórios, *Puppet*, *Chef* e *Ansible* para automação e gerenciamento de configuração e *Terraform* que utiliza a infraestrutura como código para provisionar infraestrutura, nuvem ou serviço (DAVIS; DANIELS, 2015)..

Junto com o crescimento da utilização de microsserviços e a necessidade do mercado em criar um gerenciador de *cluster* para contêiner, surge a ferramenta *Kubernetes*, um mecanismo de orquestração de contêineres de código aberto utilizado para automatizar a implantação, dimensionamento e gerenciamento de aplicativos em contêiner (KUBERNETES, 2020).

Dentre as organizações que utilizam *Kubernetes* surge também a necessidade de uma solução de entrega contínua que traga segurança, usabilidade e estabilidade nos processos de implementação (EBERMANN, 2019).

A empresa *Weaveworks* cria a metodologia *GitOps*, uma maneira de gerenciar *clusters Kubernetes* e a entrega contínua de suas aplicações. Essa metodologia funciona usando o *Git* como uma única fonte para infraestrutura e aplicações. Com o *GitOps*, o uso de agentes de *software* pode alertar sobre qualquer divergência entre o *Git* e o que está sendo executado em um *cluster* e, se houver uma diferença, os controladores *Kubernetes*

atualizam ou revertem automaticamente a alteração. Com o *Git* no centro de seus *pipelines* de entrega, os desenvolvedores podem utilizar ferramentas com que estão familiarizados para fazer solicitações de *pull*, acelerando e simplificando a implantação de aplicações e tarefas operacionais no *cluster Kubernetes* (WEAVEWORKS, 2020).

1.2 Objetivo Geral

O objetivo geral deste trabalho consiste em criar um ambiente de infraestrutura através de processos automatizados e ferramentas *DevOps*, garantindo também a entrega contínua das aplicações de maneira ágil, utilizando como meio a metodologia *GitOps*.

1.3 Objetivos Específicos

São objetivos específicos deste trabalho:

- Provisionar um ambiente de infraestrutura completo na nuvem (máquinas virtuais, rede, sistema de arquivos, banco de dados, balanceador de carga e *cluster*) através da infraestrutura como código;
- Garantir que este ambiente de infraestrutura e suas aplicações sejam criados de forma automatizada através de ferramentas *DevOps*;
- Garantir que haja um histórico das alterações por intermédio de um Sistema de Controle de Versão - *VCS* como o *GitLab*;
- Utilizar a metodologia *GitOps* para garantir que o estado descrito no *VCS* sempre corresponda com o estado do ambiente de infraestrutura.

1.4 Organização do Trabalho

O trabalho está organizado em cinco capítulos.

O primeiro e atual capítulo trata de apresentar as motivações para o desenvolvimento deste trabalho, assim como seus objetivos gerais e específicos.

No capítulo 2, são descritas as principais metodologias utilizadas para o desenvolvimento da infraestrutura, a cultura *DevOps*, o princípio de Infraestrutura como Código conhecido como *IaC* e a metodologia *GitOps*.

No terceiro capítulo são apresentadas as ferramentas utilizadas para o desenvolvimento, como o orquestrador de código aberto *Kubernetes*, o Sistema de Controle de Versão (*VCS*) chamado *GitLab* e a ferramenta de *IaC Terraform*.

No capítulo 4 o desenvolvimento da metodologia *GitOps* é mostrado, desde a

proposta do trabalho até o provisionamento da aplicação exemplo utilizada no trabalho. Por fim, no capítulo 5 é apresentada a conclusão e trabalhos futuros.

2 Fundamentação Teórica

Neste capítulo são apresentados conceitos relevantes ao desenvolvimento deste trabalho, como Metodologias de desenvolvimento da infraestrutura, *DevOps*, Infraestrutura como Código e o conjunto de práticas *GitOps*.

2.1 Metodologias de desenvolvimento da infraestrutura

2.1.1 Sistema de Controle de Versão (VCS -Version Control Systems)

Os sistemas de controle de versão são uma categoria de ferramentas de *software* que ajudam equipes de *software* a gerenciar alterações no código-fonte com o passar do tempo. O Sistema de controle de versão mantém registro de todas as modificações no código em um tipo especial de banco de dados. Se um erro for cometido, os desenvolvedores podem reverter as alterações e comparar versões anteriores do código para ajudar a corrigir o erro enquanto diminuem interrupções para todos os membros da equipe (ATLASSIAN, 2020).

Ainda de acordo com Atlassian (2020), os principais benefícios adquiridos com a utilização de um *VCS* são:

- Um histórico de alterações completo de todos os arquivos de um projeto. Nesse histórico estão incluídas a exclusão, edição, renomeação e movimentação dos arquivos, assim como a data, notas sobre as modificações e o autor;
- Criação de ramificações dos projetos em desenvolvimento. Tais ramificações permitem manter diversos fluxos de trabalho independentes, além de facilitar a junção de um novo código presente na ramificação ao código principal, verificando se há conflito entre arquivos antes dessa junção. Muitas equipes adotam a prática de criar uma ramificação para cada nova versão que será gerada;

Existe no mercado uma série de ferramentas *VCS*, cada uma com sua respectiva estrutura e topologia.

2.1.2 Integração Contínua

De acordo com Pittet (2020) a Integração Contínua (ou *Continuous Integration - CI*) é a prática de automatizar a integração das alterações de código de vários contribuidores de uma equipe em um único projeto de *software*. Portanto, um Sistema de Controle de Versão é um fator crucial para seu funcionamento.

A Integração Contínua visa melhorar a qualidade do software e reduzir o tempo de entrega, substituindo a prática tradicional de aplicar o controle de qualidade após a conclusão de todo o desenvolvimento (PEPGOTESTING, 2020).

O processo de *CI* é composto de ferramentas automáticas como *Jenkins* ou *Gitlab CI*, que garantem a correção do novo código antes da integração.

Um dos principais benefícios de adotar a Integração Contínua é a economia de tempo durante o ciclo de desenvolvimento da aplicação, tornando possível a identificação e resolução de conflitos em fases iniciais de um projeto. Também é uma ótima maneira de reduzir a quantidade de tempo gasto na correção de problemas, colocando maior ênfase em possuir um bom conjunto de testes (PITTET, 2020). O fluxo detalhado da Integração Contínua pode ser visto na Figura 1.

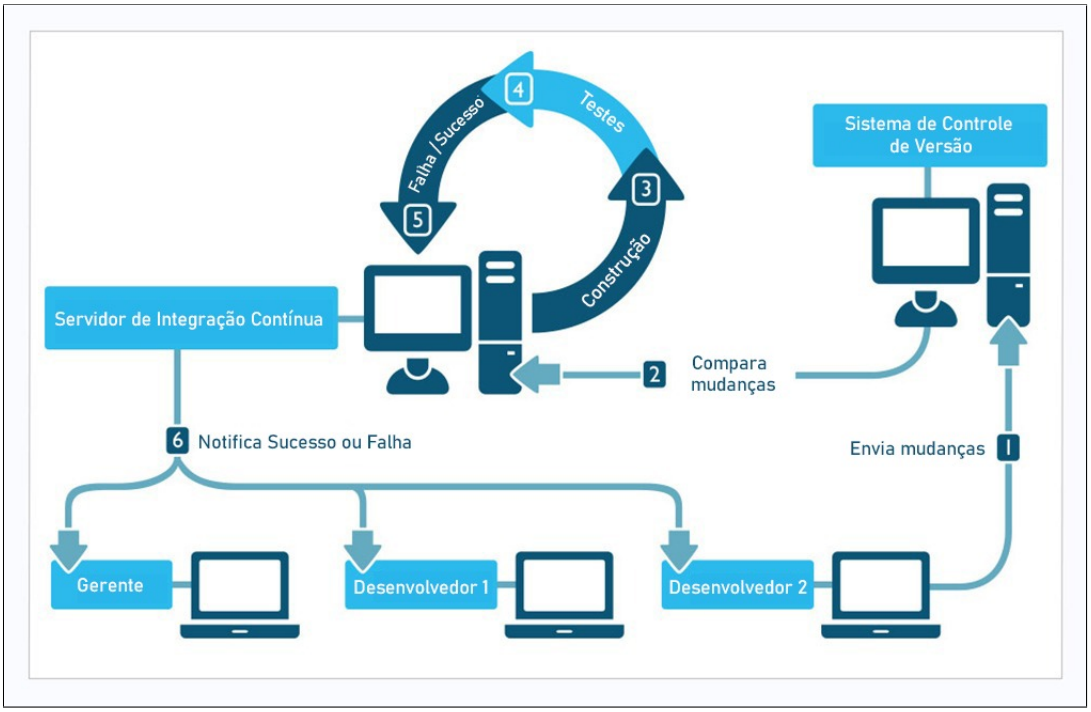

Figura 1 – Fluxo de Integração Contínua (*CI*)

Fonte: Adaptado de PepGoTesting (2020)

2.1.3 Entrega Contínua

Entrega contínua (ou *Continuous Delivery - CD*) é o resultado de uma Integração Contínua bem-sucedida, quando o *software* atualizado pode ser liberado para produção a qualquer momento (PEPGOTESTING, 2020).

Segundo Ebermann (2019), após a criação de uma nova versão de *software*, com qualidade aceitável a partir de um fluxo de Integração Contínua, a próxima etapa é implantar o código em um sistema acessível, para que possa ser usado por outros desenvolvedores. Esta implementação e execução das versões de desenvolvimento a cada alteração de código chama-se Entrega Contínua e deve sempre ser utilizada junto à Integração Contínua, o chamado *CI/CD*. A junção dos dois fluxos automatiza todo o percurso de codificação, construção, criação de versão, testes, implantação e execução do *software* e os aplica em um ambiente de desenvolvimento até estar apto para produção.

Normalmente em uma empresa há diferentes ambientes de desenvolvimento utilizados para testar versões sem impactar o usuário final, como:

- Integração/Desenvolvimento: Ambiente para testes de desenvolvimento, onde toda alteração de código é testada. Nesse ambiente os desenvolvedores podem experimentar novos recursos e a interação com diferentes partes do sistema;
- Homologação: Ambiente utilizado para testar versões completas do *software* pouco antes do seu lançamento, a fim de encontrar possíveis erros. Nesse ambiente as versões precisam ser as mais semelhantes possíveis em relação a produção;
- Produção: Ambiente que será utilizado pelos clientes. Nele deve conter apenas versões estáveis livres de erros do sistema.

Na figura 2 temos um fluxo completo de Integração e Entrega Contínua, comumente chamado de *CI/CD*.

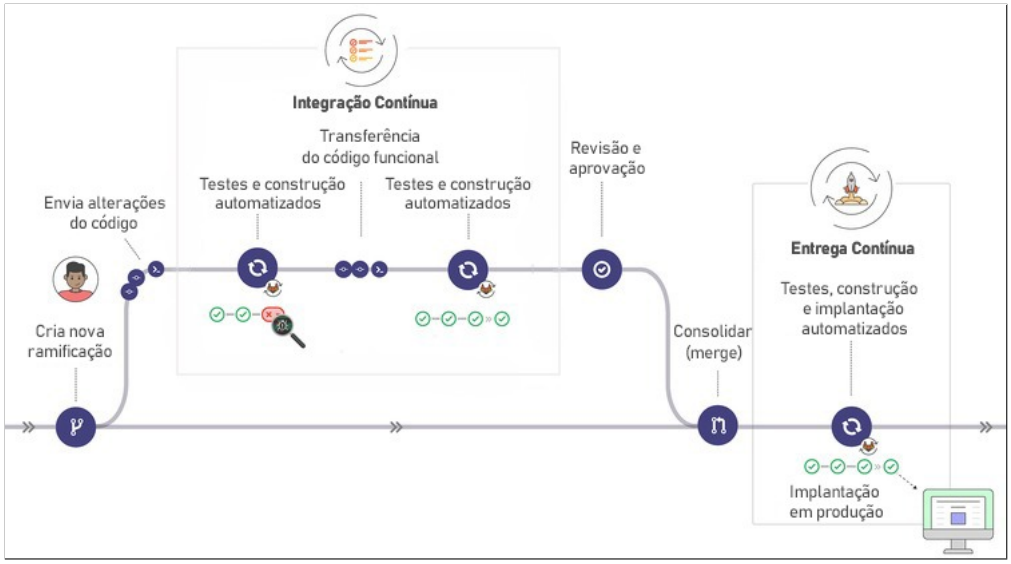

Figura 2 – Fluxo *CI/CD*

Fonte: Adaptado de Kamaruzzaman (2019)

2.1.4 Docker

Docker é uma plataforma para desenvolvedores e administradores de sistemas construírem, executarem e compartilharem suas aplicações em contêineres. Essa prática é chamada de conteinerização, considerada bastante recente no que diz respeito a produtos de *software* (DOCKER, 2020).

Segundo o site da empresa *Docker* (DOCKER, 2020), a conteinerização está em crescente ascensão devido a suas características:

- Flexibilidade: mesmo os aplicativos mais complexos podem ser armazenados em contêineres;
- Leveza: Como os contêineres compartilham o mesmo núcleo operacional do sistema (*Kernel*), torna-se possível executar vários contêineres de forma independente em um único sistema operacional.
- Portabilidade: É possível criar contêineres localmente, na nuvem ou em outros ambientes que possuam um sistema operacional;
- Fraco acoplamento: São autossuficientes e altamente encapsulados, permitindo que sejam substituídos ou atualizados sem interromper o funcionamento de outros contêineres executados no mesmo sistema operacional;
- Escalabilidade: Réplicas podem ser geradas de forma automática ou manual para outros nodos de um cluster, de acordo com a necessidade computacional do ambiente que estão inseridos.

Ainda em conformidade com o site da empresa *Docker* (DOCKER, 2020), um contêiner pode ser caracterizado como um processo isolado em execução. Esse processo tem seus recursos encapsulados dentro de seu próprio sistema de arquivos que é chamado de imagem.

Segundo Ebermann (2019), uma imagem consiste em várias camadas, conforme mostrado na Figura 3.

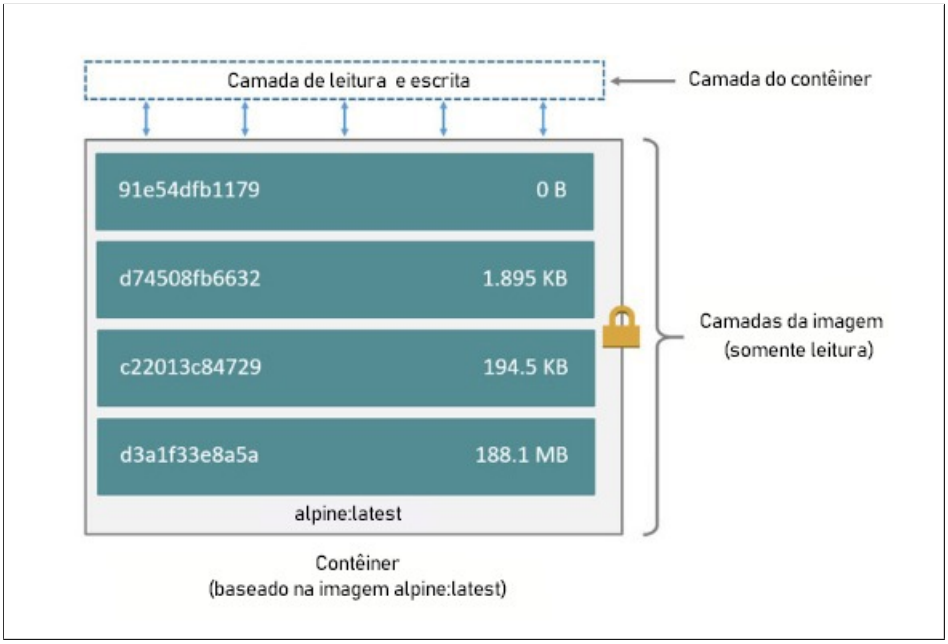

Figura 3 – Imagem *Docker* constituída por múltiplas camadas

Fonte: Adaptado de Ebermann (2019)

2.1.5 Infraestrutura Imutável

Infraestrutura Imutável é um paradigma de infraestrutura no qual os ambientes nunca são modificados depois de implantados. Caso algum componente precise ser atualizado, consertado ou modificado, novos componentes são construídos com as alterações apropriadas, sendo novamente provisionados para substituir os antigos.

Os benefícios de uma infraestrutura imutável incluem maior consistência e confiabilidade em sua infraestrutura, além de um processo de implantação mais simples e previsível. Tal metodologia mitiga problemas que são comuns em infraestruturas mutáveis, como desvio de configuração e servidores legados com bastante alterações manuais (VIRDÓ, 2017).

2.1.6 Computação em Nuvem

A nuvem é um sistema de computação distribuído e paralelo que consiste em uma coleção de computadores interconectados e virtualizados que são provisionados

dinamicamente e apresentados como um ou mais recursos de computação unificados com base em acordos de nível de serviço, estabelecidos por meio de negociação entre o provedor de serviços e os consumidores (BUYYA et al., 2008).

São exemplos de plataformas de Computação em Nuvem: *Amazon Web Services (AWS), Google Cloud Platform (GCP), Azure, DigitalOcean* e *Oracle Cloud*. Para este trabalho foi utilizado a *AWS* pelo fato desta ferramenta já ter sido estudada previamente em projetos anteriores, permitindo maior familiaridade e conhecimento dos recursos disponíveis.

2.2 DevOps

DevOps é um movimento cultural que busca aprimorar o desenvolvimento de *software* e a vida profissional das pessoas envolvidas (DAVIS; DANIELS, 2015).

Pode ser definido como uma abordagem organizacional e cultural que se concentra em colaboração e integração de desenvolvimento e operação para produzir produtos e serviços de *software* com maior rapidez e melhor qualidade (DÍAZ et al., 2019).

Conforme Bass, Weber e Zhu (2015), *DevOps* é um conjunto de práticas com intenção de reduzir o tempo entre uma alteração no sistema e a mudança em produção, garantindo alta qualidade. As práticas de *DevOps* impactam processos, produtos, estruturas organizacionais e práticas de negócio, portanto, a adoção de suas normas não costuma ser implementada de forma suave. A mudança revolucionária de sua natureza introduz uma grande tensão à organização e seus profissionais.

Temos a Figura 4 que retrata a associação entre as práticas de desenvolvimento (*Dev*) e as práticas operacionais (*Ops*).

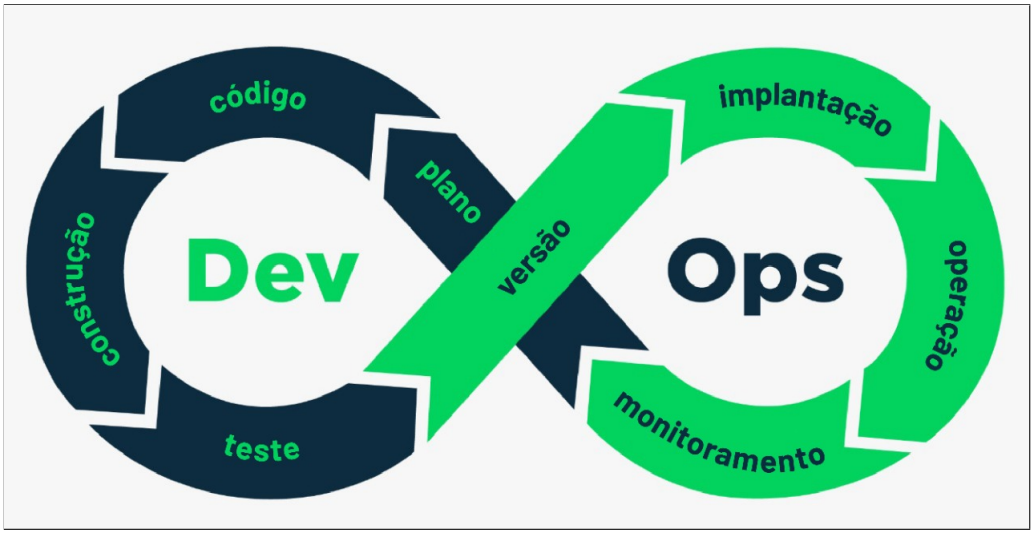

Figura 4 – Ciclo *DevOps*

Fonte: Adaptado de Queiroz (2019)

2.2.1 Práticas Devops

Segundo Bass, Weber e Zhu (2015), há cinco diferentes práticas *DevOps* importantes para implementar a cultura em uma organização:

- O operacional (*Ops*) deve ser visto como parte crucial para o sucesso do ciclo de vida de um *software*. No operacional há o conjunto de requisitos pertencentes ao monitoramento das aplicações, geração de logs e implantação (*deploy*) automatizada, que trazem segurança ao ciclo de um *software*. Envolver o operacional no desenvolvimento permite que as mensagens de *logs* fiquem compreensíveis e úteis, facilitando a análise e correção de possíveis falhas de sistema.

- O desenvolvimento (*Dev*) deve também se responsabilizar pelo tratamento de incidentes da aplicação. Essa prática visa diminuir o tempo entre a observação e o reparo de um erro, tornando a aplicação flexível e menos concentrado a uma única equipe.

- A etapa de implantação do sistema (*deployment*) deve ser realizada por ambas as partes - *Dev* e *Ops*. Isto visa garantir sua alta qualidade, evitando erros de configuração ou desenvolvimento que podem não ser identificados quando há apenas uma equipe. Esta etapa deve também possuir um histórico de alterações e documentação ativa, a fim de compreender cada modificação e seus componentes, bem como seu responsável.

- A entrega contínua (*Continuous Delivery - CD*) é crucial para o sucesso da cultura *DevOps*. As práticas relacionadas à entrega contínua se propõem a reduzir o tempo que o desenvolvedor leva para encaminhar o código gerado para um repositório e a construção de seu artefato em pequenos ciclos. Nesta etapa também está prevista a realização de testes automatizados que visam trazer uma maior qualidade ao código que entrará em produção.

- O desenvolvimento da infraestrutura deve estar em harmonia com as práticas de *IaC (Infrastructure as Code)*. Esta harmonia tem o objetivo de garantir a alta qualidade na entrega, tornando o controle da infraestrutura transparente através de seu estado salvo em código.

2.2.2 Diferença entre Tradicional Ops e DevOps

De acordo com Vehent (2018), o objetivo do *DevOps* é encurtar o tempo e reduzir o custo de transformar uma ideia em produto, para isso utiliza-se intensos processos automatizados que visam acelerar o desenvolvimento e a implantação de suas aplicações.

Em uma operação tradicional (Tradicional *Ops*) o tempo entre a concepção da

infraestrutura e a disponibilidade para o cliente utilizá-la costuma durar 8 dias. Implantar essa infraestrutura consome a maior parte do tempo, pois os profissionais encarregados precisam criar todos os componentes necessários para hospedagem do *software* de forma manual.

Utilizando a abordagem *DevOps*, conseguimos reduzir esse tempo entre concepção da infraestrutura e disponibilidade em até dois dias utilizando apenas processos automatizados que lidam com o provisionamento destes componentes. A Figura 5 ilustra a comparação entre as duas abordagens.

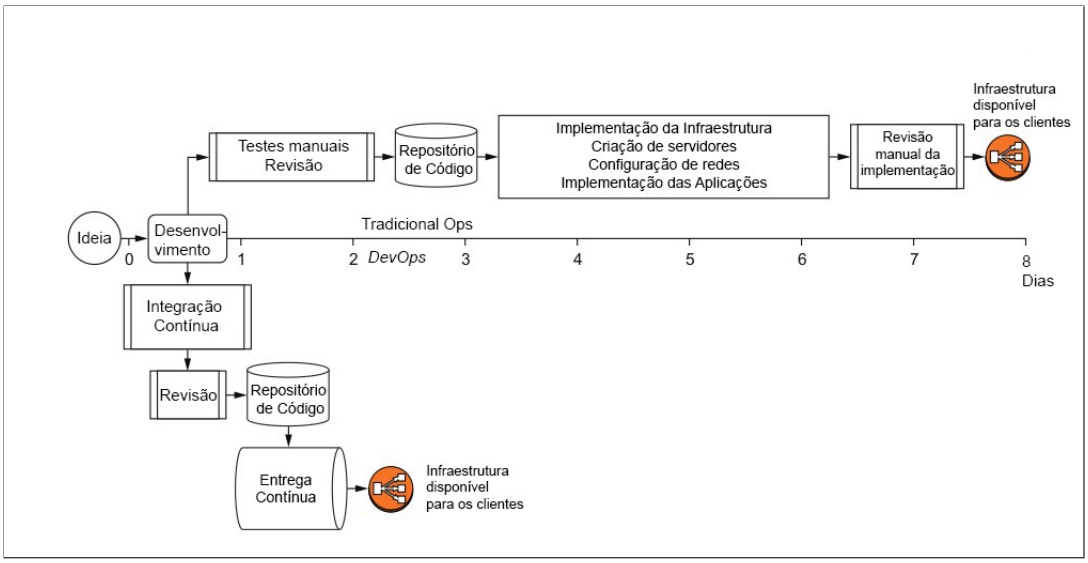

Figura 5 – Tradicional *Ops vs. DevOps*

Fonte: Adaptado de Vehent (2018)

2.3 Infraestrutura como Código

Infraestrutura como código (*Infrastructure as Code - IaC*) consiste no gerenciamento da infraestrutura (redes, máquinas virtuais, balanceadores de carga e topologias de conexão) em um modelo descritivo, utilizando o mesmo controle de versão que a equipe *DevOps* aplica para o código-fonte. Assim como o princípio de que o mesmo código-fonte gera o mesmo binário, um modelo *IaC* gera o mesmo ambiente toda vez que é aplicado. *IaC* é uma prática importante de *DevOps* e é usada em conjunto com Entrega Contínua (GUCKENHEIMER, 2017).

Ainda de acordo com Guckenheimer (2017), à medida que o código de desenvolvimento das aplicações evolui para resolver um problema, aumenta-se também sua complexidade. Dessa forma, os ambientes de infraestrutura manuais tornam-se gradualmente confusos, pois dependem de uma única configuração de difícil replicação.

Como necessidade para solucionar a complexidade nos ambientes de infraestrutura manuais, utiliza-se o *IaC*, que contém como essência a infraestrutura imutável. Este

processo é realizado através de ferramentas que provisionam um ambiente de infraestrutura completo, utilizando como seu principal fundamento linguagens de configuração declarativa. Este método de criação da infraestrutura abre espaço para a utilização e aplicação de ferramentas de desenvolvimento de *software*, como Sistema de Controle de Versão, Integração Contínua e Entrega Contínua.

Na figura 6 temos uma visão global da utilização do *IaC*.

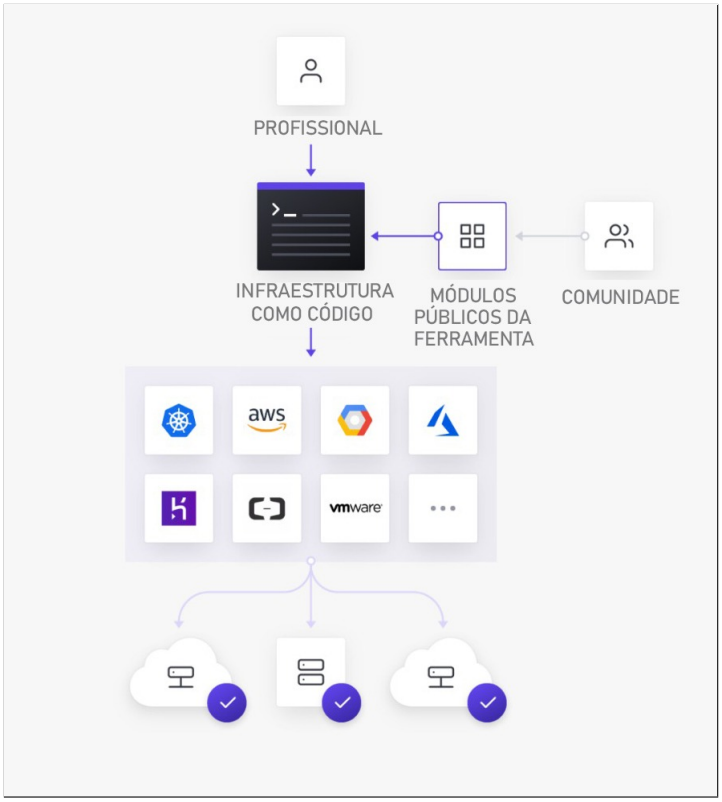

Figura 6 – Visão global do *IaC*

Fonte: Adaptado de Terraform (2020)

2.3.1 Princípios do IaC

De acordo com Morris (2016), há cinco grandes princípios do *IaC* que devem ser conhecidos para se obter êxito na implementação da metodologia:

1. Deve ser possível reconstruir facilmente e de forma confiável qualquer elemento da infraestrutura, tornando a correção de falhas simples e rápida.

2. Os sistemas devem ser descartáveis, podendo ser facilmente criados, destruídos, substituídos, redimensionados ou movidos.

3. Os sistemas devem ser consistentes, possuindo múltiplos elementos de infraestrutura idênticos que facilitem a alteração de seus recursos.

4. Tarefas curtas ou de pequena periodicidade também devem ser automatizadas para garantir que as alterações estejam sempre em conformidade.

5. Realizar modificações em serviços já existentes, como implementação de novos recursos ou correção de falhas, devem ocorrer de forma simples, segura, e consistente. Portanto, a infraestrutura deve ser projetada de forma limpa e devidamente documentada.

2.4 GitOps

Git é um *VCS* de código aberto projetado e desenvolvido por Linus Torvalds, o mesmo criador do sistema operacional *Linux*.

GitOps é uma metodologia utilizada para atender a entrega contínua e o gerenciamento de um *cluster Kubernetes*, que funciona usando o *Git* como única fonte para a infraestrutura declarativa das aplicações.

Com o *GitOps*, é possível identificar e alertar sobre qualquer divergência entre o sistema e as informações inseridas no *Git*. O conceito do *GitOps* concentra-se em ter um repositório *Git* que contenha descrições declarativas da infraestrutura desejada e um processo automatizado que torne possível ao estado do ambiente sempre corresponder ao estado descrito no repositório (WEAVEWORKS, 2020).

2.4.1 Princípios do GitOps

De acordo com a empresa WeaveWorks (2020) criadora da metodologia, para iniciar o gerenciamento do *cluster Kubernetes* utilizando o fluxo de trabalho do *GitOps* os seguintes princípios devem ser seguidos:

1. Todo ambiente deve estar descrito declarativamente. Assim, as aplicações podem ser facilmente implementadas e revertidas no sistema.

2. O estado desejado do sistema deve estar versionado no *Git*. Com a declaração do sistema armazenado em um Sistema de Controle de Versão e servindo como sua única fonte, tem-se um único lugar de onde tudo é derivado e conduzido.

3. Após ter o estado declarado mantido no *Git*, a próxima etapa é permitir que quaisquer mudanças nesse estado sejam aplicadas automaticamente ao seu sistema. Dessa forma, não são necessárias credenciais do sistema para realizar quaisquer alterações.

2.4.2 Benefícios do GitOps

Ainda de acordo com a WeaveWorks (2020), os principais benefícios na utilização dessa metodologia são:

1. Aumento de produtividade: com o *GitOps* as equipes precisam se preocupar menos com a implantação das versões geradas, permitindo despender mais tempo no desenvolvimento das aplicações.

2. Os desenvolvedores podem utilizar ferramentas familiares como o *Git* para gerenciar atualizações e recursos para o *Kubernetes*, sem precisar ter experiência com sistemas *Kubernetes*.

3. Maior confiabilidade: como todo o sistema é descrito no *Git*, torna-se mais simples reverter alterações e solucionar falhas de sistema, diminuindo o tempo de recuperação.

4. Consistência e padronização: com o *GitOps* é possível criar um modelo padronizado para criação e alteração tanto da infraestrutura quanto das aplicações, tornando o fluxo de trabalho consistente.

5. Garantia de maior segurança: com o *Git* é possível ter o rastreamento e gerenciamento de alterações de código, bem como a capacidade de assinar alterações que comprovam sua autoria e origem.

Na figura 7 podemos visualizar o processo de *CI/CD* utilizando a metodologia *GitOps*.

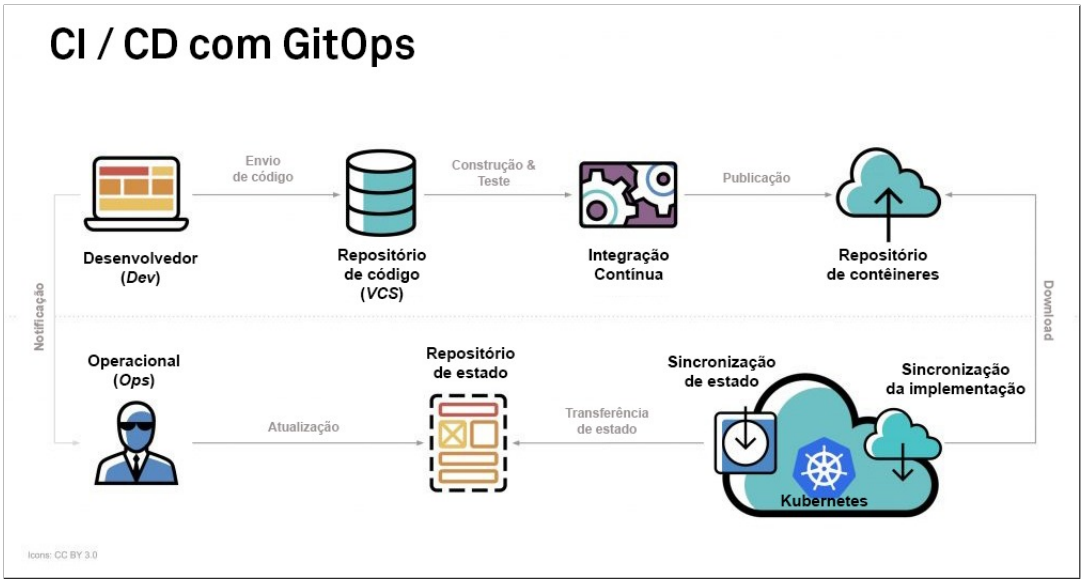

Figura 7 – Processo simplificado do *CI/CD* com *GitOps*

Fonte: Adaptado de WeaveWorks (2020)

3 Ferramentas

Neste capítulo são apresentadas ferramentas utilizadas para o desenvolvimento deste trabalho, como sistema de orquestração de contêineres *Kubernetes*, Sistema de Controle de Versão, Integração e Entrega Contínua *GitLab* e *GitLab CI/CD* e a ferramenta para Infraestrutura como Código chamada *Terraform*.

3.1 Kubernetes

Kubernetes é um orquestrador de código aberto utilizado para implantar e gerenciar contêineres *Docker* em máquinas virtuais ou físicas que façam parte de seu próprio *cluster Kubernetes*.

Foi originalmente desenvolvido pela empresa Google, inspirada por uma década de experiência na implantação de sistemas escalonáveis e confiáveis em contêineres (BURNS; HIGHTOWER; BEDA, 2017).

Os recursos são declarados em arquivos de configuração chamados *manifest* que utilizam uma linguagem de serialização de dados chamada *YAML - Yet Another Markup Language*, que podem ser removidos ou atualizados a partir da ferramenta de linha de comando *kubectl*. Para separar múltiplos projetos em um mesmo *cluster Kubernetes*, geralmente é utilizado um delimitador abstrato chamado espaço de nomes ou *namespace* (EBERMANN, 2019).

Em um *cluster Kubernetes* os contêineres não são executados diretamente, mas envolvidos em uma estrutura de nível superior chamada *pod*. Todos os contêineres pertencentes ao mesmo *pod* compartilham os mesmos recursos e rede local.

Os *pods* são utilizados como unidades de replicação altamente escalonáveis, portanto, caso a aplicação esteja recebendo bastante requisições e uma única instância não seja suficiente, o *cluster Kubernetes* pode se encarregar de gerar novas réplicas para balancear a carga.

Embora os *pods* sejam a unidade básica de computação em um *cluster Kubernetes*, eles também não são implantados diretamente. Para isso, há mais uma camada de abstração chamada *Deployment*. A funcionalidade básica dessa camada é declarar quantas réplicas de um *pod* devem estar em execução por vez, e caso ocorra alguma falha e o *pod* seja removido, o *Deployment* se encarrega de manter o mesmo estado desejado, recriando-o (SANCHE, 2018).

De acordo com a documentação do Kubernetes (2020), há outros recursos necessários para o funcionamento de uma aplicação em *Kubernetes* como *ConfigMaps* e *Secrets* que são utilizados para mapear as variáveis de ambiente que serão utilizadas pelo

pod, instâncias de armazenamento virtuais chamadas *Persistent Volume (PV)*, balanceadores de carga utilizados para direcionar os tráfegos de rede para os *pods*, denominados *Services*, e o *Ingress*, um objeto *Kubernetes* que nos permite externalizar os Services do *cluster*. Alguns desses recursos também são abordados ao longo do desenvolvimento do trabalho.

Na Figura 8 temos o exemplo simples de um *manifest* que gera uma camada de abstração *Deployment*, que por sua vez implementa um *pod* denominado *gitops-whoami* responsável por imprimir informações do Sistema Operacional e as solicitações *HTTP* do ambiente na qual foi implementado.

```
1  apiVersion: apps/v1beta2
2  kind: Deployment
3  metadata:
4    name: gitops-whoami
5    labels:
6      app: gitops-whoami
7      family: gitops
8  spec:
9    replicas: 1
10   selector:
11     matchLabels:
12       app: gitops-whoami
13   template:
14     metadata:
15       labels:
16         app: gitops-whoami
17     spec:
18       containers:
19       - name: gitops-whoami
20         image: containous/whoami
21         ports:
22         - containerPort: 80
23           name: http
24           protocol: TCP
```

Figura 8 – Exemplo de um manifest *deployment.yaml*.

Fonte: Criado pelo autor

3.2 GitLab

GitLab é um sistema utilizado para gerenciar repositórios *Git*. Escrito em *Ruby*, permite a implantação de um controle de versão para o seu código de forma rápida e fácil.

Foi publicado pela primeira vez na plataforma de hospedagem de código-fonte *GitHub* em outubro de 2011 e tornou-se uma ferramenta poderosa desde então. Fundada por Dmitriy Zaporozhets, hoje conta com uma plataforma de hospedagem própria que pode ser utilizada gratuitamente por qualquer profissional ou entusiasta da área (HETHEY, 2013).

De acordo com o site oficial da empresa Gitlab Inc. (2020) há duas alternativas

na utilização da ferramenta:

- *GitLab SaaS*: desta forma é usada a hospedagem própria da GitLab Inc. (gitlab.com), sem necessidade de baixar e instalar a ferramenta em uma máquina física ou virtual. Criando a conta no site, já é possível usufruir da ferramenta.
- *GitLab Self-Managed*: desta forma é implantada uma instância própria do *GitLab* em um servidor local (*on-premise*) ou na nuvem. Assim, o administrador do sistema torna-se responsável pela instalação, administração, gerenciamento, atualização e manutenção do servidor.

Independente da forma de utilização escolhida, há também a necessidade de definir o plano que será utilizado, dentre:

a) Core/Free: Plano gratuito com recursos limitados da ferramenta;

b) *Starter/Bronze*: todos os recursos da versão gratuita, maior controle sobre o código e suporte até o próximo dia útil;

c) *Gold/Premium*: todos os recursos da versão *Starter/Bronze*, configurações avançadas de segurança e funcionalidades utilizadas para garantir a qualidade do código.

3.2.1 GitLab CI/CD

GitLab CI/CD é uma ferramenta incorporada ao *GitLab* para desenvolvimento de *software* por meio de metodologias contínuas como Integração Contínua (*CI*), Entrega Contínua (*CD*) e Implantação Contínua (*CD*).

A Integração Contínua funciona enviando pequenos pedaços de código para o projeto hospedado em um repositório *Git* e, para cada ação de inserção ou alteração de código, executa um fluxo de tarefas (*pipeline*) de roteiros (*scripts*) para construir, testar e validar as alterações antes de consolidá-las (*merge*) na ramificação (*branch*) do código principal.

Já a Entrega Contínua e Implantação Contínua consistem em uma etapa adicional de Integração Contínua que fornece a implantação da aplicação para o ambiente desejado (integração/desenvolvimento, homologação ou produção) a cada inserção de código (GITLAB INC., 2020).

Conforme GitLab Inc. (2020), a configuração do *GitLab CI/CD* é realizada através de um arquivo que utiliza a linguagem *YAML*, chamado *gitlab-ci.yaml* que deve estar presente na raiz do repositório. A partir dele, é possível construir um *pipeline* que será executado toda vez que tiver alterações no código. Estes *pipelines* consistem em um ou mais estágios (*stages*) que são executados em ordem com seus *scripts* pré-estabelecidos

que podem ser executados em paralelo dependendo da necessidade. Na figura 9 temos o exemplo de um arquivo *gitlab-ci.yaml* que configura um estágio de implantação (*deploy*) junto com seus *scripts* de execução.

Contents of .gitlab-ci.yml

```
1  image: ruby:2.7
2
3  workflow:
4    rules:
5      - if: '$CI_COMMIT_BRANCH'
6
7  pages:
8    stage: deploy
9    script:
10     - gem install bundler
11     - bundle install
12     - bundle exec jekyll build -d public
13   artifacts:
14     paths:
15       - public
16   rules:
17     - if: '$CI_COMMIT_BRANCH == "master"'
```

Figura 9 – Exemplo de um arquivo *gitlab-ci.yaml.*

Fonte: Criado pelo autor

Para execução do *GitLab CI/CD* são necessárias máquinas virtuais ou físicas chamadas corredores (*runners*), instâncias utilizadas para distribuir a carga de execução dos *pipelines*. A Figura 10 mostra um *pipeline* completo e seus stages definidos (*Validate, Plan, Apply, Ansible, Destroy*) no *GitLab* utilizando o *GitLab CI/CD*.

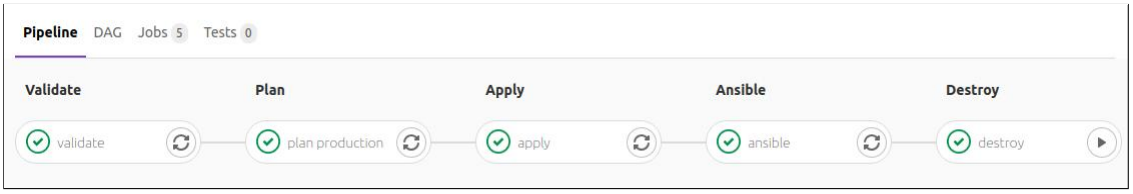

Figura 10 – *Pipeline* do *Gitlab CI/CD*

Fonte: Criado pelo autor

3.3 Terraform

Terraform é uma ferramenta de código aberto de provisionamento de infraestrutura, criada pela HashiCorp, que permite que definamos nossa Infraestrutura como Código, usando uma linguagem simples e declarativa (SOUZA, 2017).

Conforme Brikman (2016), o binário do *Terraform* criado na linguagem *Go* permite que seja criada uma infraestrutura inteira (bancos de dados, balanceadores de carga, topologias de rede, máquinas virtuais, dentre outros recursos) de seu próprio computador ou de um servidor sem precisar executar qualquer outra funcionalidade ou ferramenta. Isso

é possível devido às chamadas de acesso às plataformas de provedores que o binário realiza através dos recursos do *Terraform*. Dentre os provedores disponíveis para integração estão: *AWS, Azure, GCP, DigitalOcean* e *VMware*.

O código do *Terraform* é escrito na Linguagem de Configuração HashiCorp (*HCL*) em arquivos com a extensão *.tf*. É uma linguagem declarativa, portanto seu objetivo é descrever a infraestrutura desejada e o *Terraform* descobrirá como criá-la.

Na Figura 11 temos o exemplo de um arquivo *.tf* escrito em *HCL* que provisiona uma instância *EC2 - Amazon Elastic Compute Cloud* na *AWS*.

```
1  //Criação de uma instância EC2
2
3  provider "aws" {
4    version = "~> 2.61.0"
5    region  = "us-east-2"
6    access_key = "my-access-key"
7    secret_key = "my-secret-key"
8  }
9
10 resource "aws_instance" "ec2" {
11
12   ami                         = "ami-0e01ce4ee18447327"
13   instance_type               = "t2.micro"
14
15   tags = {
16     Name        = "gitops-ec2-instance"
17     terraform   = "true"
18     environment = "Dev"
19   }
20 }
```

Figura 11 – Arquivo *.tf* escrito em *HCL*

Fonte: Criado pelo autor

3.3.1 Estado do Terraform

Cada vez que o *Terraform* é executado, as informações sobre a infraestrutura criada são registradas em um arquivo de estado do *Terraform* chamado *terraform.tfstate* (BRIKMAN,2016).

Segundo a empresa HashiCorp, o objetivo principal deste arquivo é armazenar o vínculo entre os objetos de um sistema remoto e os recursos declarados nos arquivos do *Terraform*. Quando o *Terraform* cria um objeto, é registrada a identidade deste objeto no arquivo de estado, tornando possível atualizar ou remover esse objeto através do vínculo criado. Na Figura 12 é demonstrada a geração de um estado do Terraform.

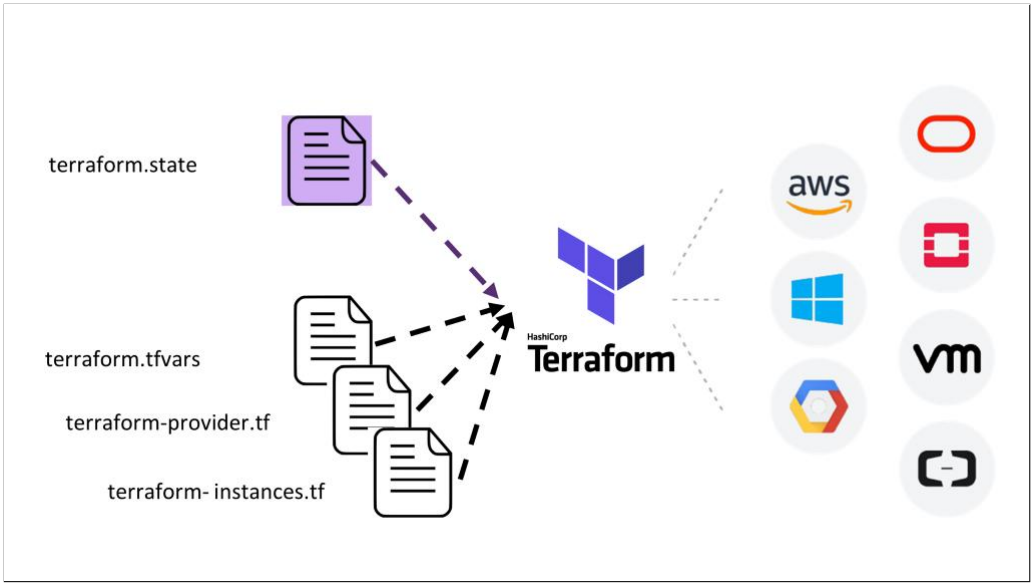

Figura 12 – Geração do estado do *Terraform* através dos arquivos de configuração.

Fonte: Belchior (2018)

4 Desenvolvimento da metodologia GitOps

Neste capítulo é apresentada a proposta do trabalho e o seu desenvolvimento, desde a utilização das ferramentas para gerenciamento e controle do modelo *GitOps*, até o código utilizado para o provisionamento da Infraestrutura como Código - *IaC* na Nuvem da *AWS*.

4.1 Proposta do Trabalho

Nesta seção é apresentado o ambiente de infraestrutura e a proposta do trabalho que visa o provisionamento de uma infraestrutura automatizada, o gerenciamento e a entrega contínua de suas aplicações através da metodologia GitOps.

4.1.1 Ambiente de Infraestrutura

Foi utilizado para o desenvolvimento deste trabalho a plataforma de serviços de computação em nuvem da empresa Amazon, chamada *Amazon Web Services - AWS*.

Os recursos utilizados nesta plataforma foram:

- *Amazon Elastic Compute Cloud (EC2)*: Abstração de uma máquina virtual com capacidade computacional redimensionável de acordo com sua região geográfica.
- *Amazon Elastic Kubernetes Service (EKS)*: Recurso que permite a criação de um *cluster Kubernetes* que permite a abstração de toda a parte de configuração de seus controladores. O *EKS* fornece a criação de *Node Groups*, ou grupos de nós, que permitem provisionar nós *EC2* de forma automática, sem necessidade de configurar um a um manualmente.
- *Amazon Virtual Private Cloud (VPC)*: Recurso que permite a criação de uma rede virtual definida pelo usuário. Nela é possível criar um *gateway* de conversão de endereços de rede (*NAT Gateway*), sub-redes públicas e privadas, tabelas de rotas e grupos de segurança (*security groups*).
- *Amazon Elastic File System (EFS)*: Recurso utilizado para criação de um sistema de arquivos escalável que permite o compartilhamento de dados pelas instâncias *EC2*.
- *Amazon Relational Database Service (RDS)*: Recurso que permite a criação de uma banco de dados escalável e redimensionável sem a necessidade de deixá-lo inacessível.

- *Amazon Elastic Load Balancing (ELB)*: Recurso que permite a criação de um balanceador de carga que distribui automaticamente o tráfego de entrada de instâncias *EC2*, contêineres e endereços *IPs*.
- *Amazon Simple Storage Service (S3)*: Serviço de armazenamento de objetos escalável que comporta qualquer volume de dados. Utilizado também para guardar o estado das aplicações.

Na figura 13 podemos visualizar através de um diagrama como estes recursos se comunicam.

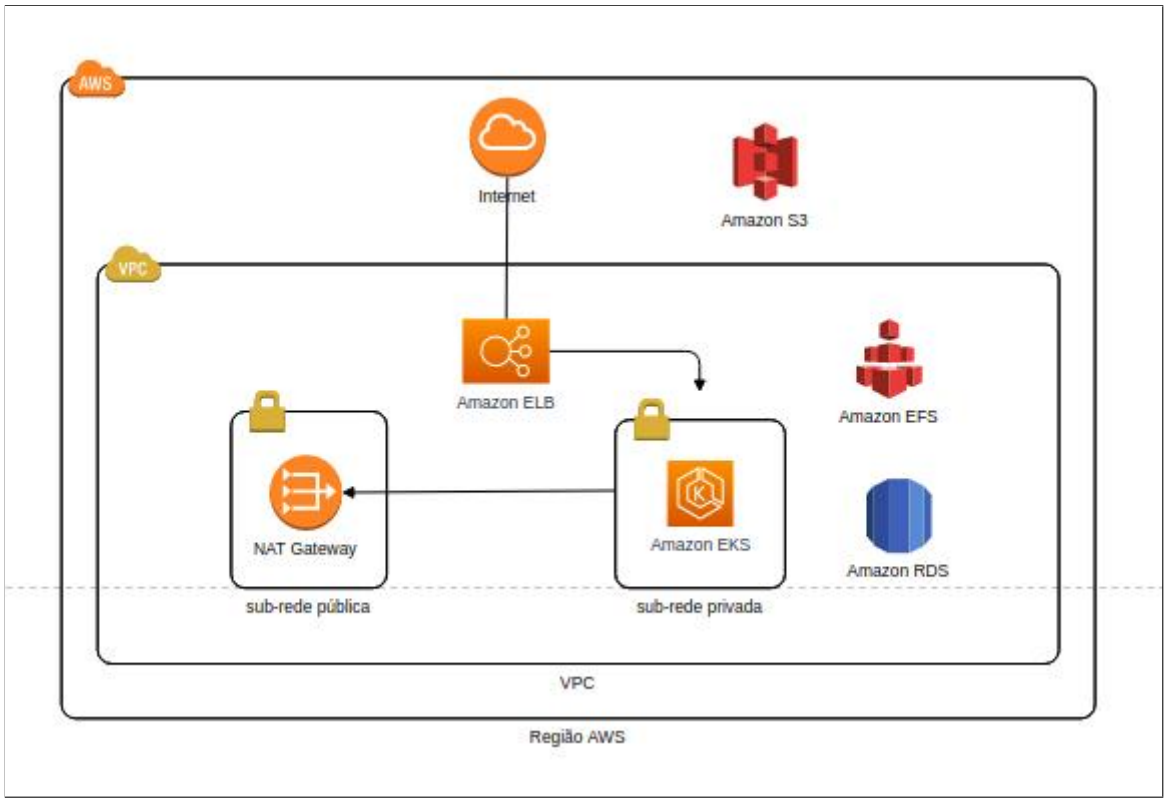

Figura 13 – Diagrama de comunicação entre os recursos da *AWS*.

Fonte: Criado pelo autor

4.1.2 Proposta

A proposta apresentada aborda os seguintes pontos:

- Integrar os processos de desenvolvimento de *software (Dev)* e operacionais *(Ops)* através das práticas *DevOps* tratadas no trabalho.
- Provisionar um ambiente de Infraestrutura Imutável de forma automática com os recursos abordados anteriormente na *AWS*, declarados estritamente através da Infraestrutura como Código, tendo *Kubernetes* como seu gerenciador e o *GitLab* como Sistema de Controle de Versão.

- Tornar possível a Integração e Entrega contínua desta infraestrutura e das aplicações através das ferramentas de *CI/CD* contidas no *GitLab*.
- Garantir que o estado descrito no ambiente de infraestrutura e gerado através da ferramenta de Infraestrutura como Código *Terraform*, esteja em conformidade com os estados dos repositórios de infraestrutura e aplicações presentes no *GitLab* utilizando como preceito a metodologia *GitOps*.

O diagrama presente na Figura 14 ilustra as metodologias, ferramentas e recursos que são utilizados para o desenvolvimento do trabalho e como é realizada a interação entre eles.

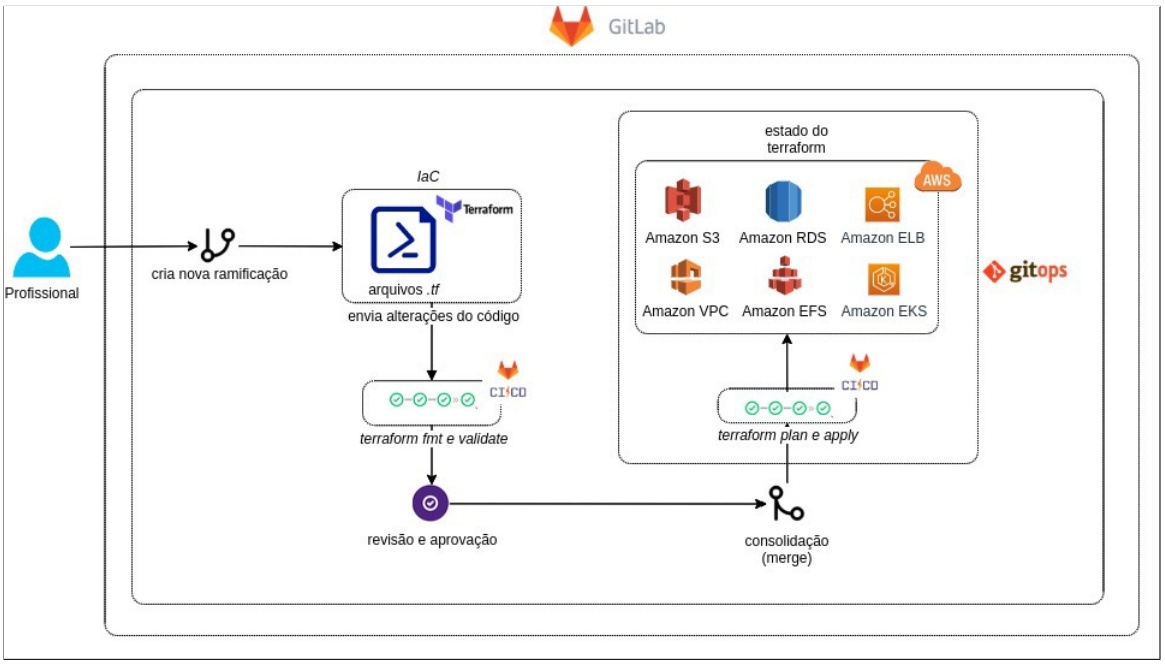

Figura 14 – Diagrama das ferramentas utilizadas

Fonte: Criado pelo autor

4.2 AWS

Nesta seção são descritas as formas de provisionamento da infraestrutura de uma aplicação através dos recursos da *AWS*: criação de máquinas virtuais (*EC2*), criação das redes (*VPC*), definição do sistema de arquivos (*EFS*), criação do banco de dados do ambiente (*RDS*), definição do balanceador de carga para acesso ao ambiente (*NLB*) e criação do serviço de armazenamento do estado do ambiente (*S3*). Os códigos das aplicações são executados em um cluster *Kubernetes (EKS)*. Todos os recursos foram provisionados na região leste dos EUA *(Ohio) - us-east-2*.

4.2.1 Elastic Compute Cloud - EC2

Conforme visto em Ambiente de Infraestrutura, o *EC2* é um recurso disponível para a criação de máquinas virtuais redimensionáveis, que possui tamanhos específicos para cada região da *AWS*.

Os *Node Groups*, ou grupos de nós do *EKS* podem ser criados a partir de instâncias *EC2*, fornecendo o poder computacional necessário para o funcionamento do *cluster Kubernetes*.

4.2.2 Virtual Private Cloud - VPC

Como citado no capítulo 4, *VPC* é um recurso presente na *AWS* que nos permite criar uma rede virtual isolada.

Para o desenvolvimento deste recurso, foi utilizado o módulo *VPC* da comunidade do *Terraform* e mantido por Anton Babenko, disponibilizado no *GitHub* no *link* `https://github.com/terraform-aws-modules/terraform-aws-vpc`.

Este módulo nos permite, através da linguagem original do *Terraform - HCL*, criar *VPCs* e seus sub recursos, como: sub-redes públicas e privadas, *NAT Gateways*, *VPN Gateways*, e outros recursos que não foram abordados neste trabalho.

Para o provisionamento da VPC, foi criado um arquivo HCL vpc.tf seguindo os seguintes atributos:

- *name*: Nome que será atribuído à *VPC* no momento de sua criação na *AWS*;
- *cidr*: Utilizado para repartir os endereços *IP* em blocos;
- *azs*: Zonas de disponibilidade da *AWS*;
- *public_subnets*: Lista de sub-redes públicas que serão utilizadas nesta *VPC*;
- *private_subnets*: Lista de sub-redes privadas que serão utilizadas nesta *VPC*;
- *enable_nat_gateway*: Atributo lógico (*Boolean*) que permite a criação de uma *NAT Gateway* para cada rede privada da *VPC*;
- *single_nat_gateway*: *Boolean* que permite criar apenas uma *NAT Gateway* para todas as redes privadas da *VPC*;
- *external_nat_ip_ids*: Lista de *IPs* estáticos que serão utilizados pela *NAT Gateway*;
- *enable_dns_hostnames*: *Boolean* que permite a utilização de *hostnames* na *VPC*.

A partir destes atributos, torna-se possível provisionar uma rede virtual privada, ou *VPC*, que nos permite configurar um ambiente completo de infraestrutura. Um trecho do código utilizado para este provisionamento pode ser observado na Figura 15.

```
11  module "vpc" {
12    source  = "terraform-aws-modules/vpc/aws"
13    version = "2.64.0"
14
15    name                 = format("gitops-vpc-%s", var.environment)
16    cidr                 = "10.0.0.0/16"
17    azs                  = ["us-east-2a", "us-east-2b", "us-east-2c"]
18    private_subnets      = ["10.0.1.0/24", "10.0.2.0/24", "10.0.3.0/24"]
19    public_subnets       = ["10.0.101.0/24", "10.0.102.0/24", "10.0.103.0/24"]
20    enable_nat_gateway   = true
21    single_nat_gateway   = true
22    reuse_nat_ips        = true
23    external_nat_ip_ids = aws_eip.nat.*.id
24
25    enable_vpn_gateway                = true
26    propagate_public_route_tables_vgw = true
27
28    enable_dns_hostnames = true
29    enable_dns_support   = true
```

Figura 15 – Trecho de código para criação da *VPC* em *HCL*

Fonte: Criado pelo autor

4.2.2.1 Security Groups - SGs

De acordo com a *AWS* (2021), *Security Groups (SGs)* ou Grupos de Segurança são regras que atuam como um *firewall* virtual para suas instâncias *EC2* e que podem ser utilizadas para outros recursos como: *RDS, EFS*, dentre outros da *AWS*.

Nos *SGs* é possível definir apenas regras de permissão de acesso, como por exemplo os tráfegos de entrada e saída baseados em portas.

No desenvolvimento do trabalho, o módulo utilizado foi *terraform-aws-security-group* disponível no *GitHub* no *link* `https://github.com/terraform-aws-modules/terraform-aws-security-group` mantido pela comunidade do *Terraform* e gerenciado por Anton Babenko.

Foram provisionados dois *security groups*, um para o *Relational Database Service* e outro para o *Elastic File System*. Desta forma, podemos garantir que esses dois recursos só poderão ser acessados pelas faixas de *IPs* especificadas nos blocos presentes em *ingress_with_cidr_blocks*.

Na Figura 16 podemos observar o trecho de código de um dos *SGs* e seus atributos.

```
3  module "rds_sg" {
4
5    source = "terraform-aws-modules/security-group/aws"
6
7    name        = format("gitops-rds-sg-%s", var.environment)
8    description = "Permite acesso dos IPs BRy"
9    vpc_id      = module.vpc.vpc_id
10
11   ingress_with_cidr_blocks = [
12     {
13       from_port   = 5432
14       to_port     = 5432
15       protocol    = "tcp"
16       description = "Permitir acesso a VPC"
17       cidr_blocks = "10.0.0.0/16"
18     },
19   ]
```

Figura 16 – Trecho de código do módulo *terraform-aws-security-group*

Fonte: Criado pelo autor

4.2.3 Elastic Kubernetes Service - EKS

Como mencionado no capítulo 4, *EKS* é um recurso disponível na *AWS* que nos possibilita criar e gerenciar um *cluster Kubernetes* abstraindo grande parte de sua arquitetura.

Em seu desenvolvimento, foi utilizado o módulo *terraform-aws-eks*, disponível no *GitHub* no *link* `https://github.com/terraform-aws-modules/terraform-aws-eks` e mantido pela comunidade do *Terraform*.

Para seu provisionamento foi criado um arquivo *eks.tf* escrito na linguagem do *Terraform* e seguindo os principais atributos:

- *cluster_name*: Nome que será atribuído ao *EKS* no momento de sua criação;
- *cluster_version*: Versão do *cluster Kubernetes* disponível no *EKS*;
- *subnets*: Lista de sub redes que serão utilizadas pelo *EKS*;
- *vpc_id*: Identificador da *VPC* que será utilizado para a implantação do *cluster EKS*;
- *enable_irsa*: *Boolean* que permite a criação de políticas e permissões específicas para o *EKS*.

Na figura 17 podemos acompanhar a declaração destes atributos.

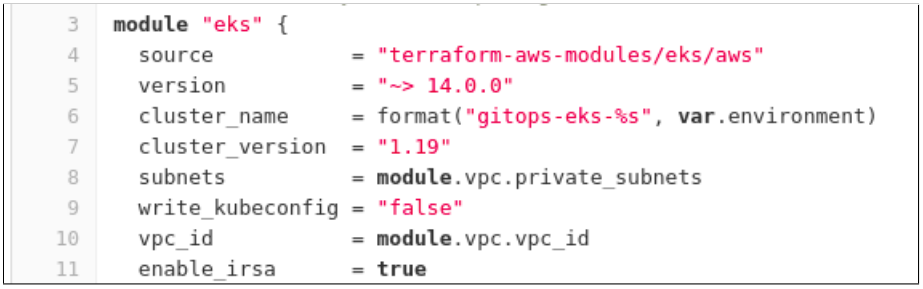

Figura 17 – Trecho de código do módulo *EKS*

Fonte: Criado pelo autor

Para o desenvolvimento do trabalho, foi definida a utilização da versão do *Kubernetes* mais recente disponível para o *EKS*, 1.19.

Além destes atributos, existem dois blocos bastante importantes para a criação do *EKS*. Um nos permite configurar a arquitetura dos processadores dos nós e o tamanho do disco, o outro nos permite delimitar a capacidade mínima, máxima e a desejada de nós no *cluster*.

Na Figura 18, temos o bloco *node_groups_defaults* que mostra as sub-redes que serão utilizadas por estes nós, a arquitetura do processador e o tamanho do disco.

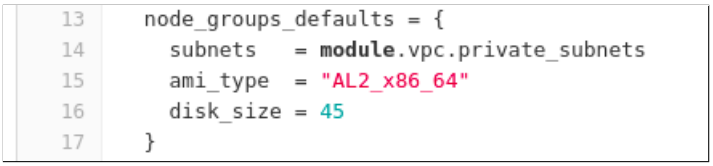

Figura 18 – Declaração do bloco *node_groups_defaults*

Fonte: Criado pelo autor

Já a Figura 19 mostra o bloco *node_groups*, responsável pela configuração dos nós, dispondo dos seguintes atributos:

- *name*: Nome que será atribuído ao grupo de nós do *EKS*;
- *desired_capacity*: Número desejado de nós no cluster *EKS*;
- *max_capacity*: Número máximo de nós que o *cluster EKS* pode atingir caso haja um uso elevado de recursos;
- *min_capacity*: Número mínimo de nós que pode haver no *cluster EKS*;
- *key_name*: Nome da chave de acesso que será utilizada para acessar os nós do *cluster*;

- *instance_type*: Tipo de instância *EC2* que será utilizada para o provisionamento dos nós, de acordo com as especificações necessárias de *CPU*, memória, capacidade de rede e disponibilidade de região na *AWS*.

```
19  node_groups = {
20    gitops_t2large = {
21      name             = format("gitops-eks-node-t2large-%s", var.environment)
22      desired_capacity = 2
23      max_capacity     = 5
24      min_capacity     = 2
25      key_name         = "gitops-ohio"
26      instance_type    = "t2.large"
```

Figura 19 – Declaração do bloco *node_groups*

Fonte: Criado pelo autor

4.2.4 Elastic File System - EFS

No desenvolvimento do *EFS* foi utilizado o módulo *terraform-aws-efs* disponibilizado gratuitamente pela empresa *Cloud Posse*, tendo seu código aberto disponível no *GitHub* no *link* `https://github.com/cloudposse/terraform-aws-efs`.

Para seu provisionamento foi criado o arquivo *efs.tf* com os consecutivos atributos:

- *name*: Nome atribuído ao *EFS*;
- *region*: Região da *AWS* na qual os recursos serão provisionados;
- *vpc_id*: Identificador da *VPC* que será utilizada pelo *EFS*;
- *subnets*: sub rede que será utilizada pelo *EFS*;
- *security_groups*: Identificador do grupo de segurança que dará permissão de acesso e conexão às portas do *EFS* e faixas de *IPs*;
- *encrypted*: *Boolean* que permite criptografar o sistema de arquivos;
- *transition_to_ia*: Delimita o tempo que leva para transferir os arquivos do *EFS* para o armazenamento de pouco acesso (*Infrequent Access - IA*).

Um trecho do código do módulo *terraform-aws-efs* em *HCL* pode ser observado na Figura 20.

```
4  module "efs" {
5    source = "git::https://github.com/cloudposse/terraform-aws-efs.git?ref=tags/0.30.1"
6
7    name             = format("gitops-efs-%s", var.environment)
8    region           = "us-east-2"
9    vpc_id           = module.vpc.vpc_id
10   subnets          = module.vpc.public_subnets
11   security_groups  = [module.efs_sg.this_security_group_id]
12   encrypted        = true
13   transition_to_ia = "AFTER_60_DAYS"
```

Figura 20 – Declaração do módulo *terraform-aws-efs*

Fonte: Criado pelo autor

4.2.5 Relational Database Service - RDS

Também previamente mencionado no capítulo 4, *RDS* é um recurso da *AWS* que permite a criação de um banco relacional altamente escalável oferecendo seis mecanismos de bancos de dados comuns, sendo eles: *PostgreSQL, MySQL, Amazon Aurora, MariaDB, Oracle Database* e *SQL Server*. Para o desenvolvimento do trabalho, o sistema gerenciador de banco de dados (*SGBD*) escolhido foi o *PostgreSQL*.

O módulo *Terraform* escolhido para a criação da instância *RDS* foi o *terraform-aws-rds*, presente também no *GitHub* no *link* `https://github.com/terraform-aws-modules/terraform-aws-rds` e mantido por um grupo de profissionais que participam da comunidade do *Terraform*, gerenciado por Anton Babenko.

Neste módulo há uma série de atributos importantes que devem ser declarados para o bom funcionamento do *RDS*, sendo eles:

- *identifier*: Nome atribuído à instância *RDS*;
- *engine*: O mecanismo de banco de dados a ser utilizado;
- *engine_version*: Versão do mecanismo de banco de dados;
- *instance_class*: Classe, ou tamanho da instância de acordo com sua região na *AWS*;
- *allocated_storage*: Especifica o tamanho de armazenamento do *RDS*;
- *max_allocated_storage*: Especifica o valor máximo que o armazenamento do banco pode expandir;
- *storage_encrypted*: *Boolean* que permite criptografar os dados da instância *RDS* caso verdadeira;
- *multi_az*: *Boolean* que especifica se a instância será *Multi-AZ* (instância principal + instância de espera em outra zona de disponibilidade para contenção de erros);

- *username*: Usuário principal do *RDS*;
- *password*: Senha do Usuário principal;
- *port*: Especifica a porta na qual o *RDS* aceitará conexões;
- *vpc_security_group_ids*: Identificador do grupo de segurança que dará permissão de acesso e conexão às portas do *RDS* e faixas de *IPs*.

Na Figura 21 podemos observar um trecho de código do arquivo *rds.tf* com os atributos citados acima.

```
16  module "rds" {
17    source  = "terraform-aws-modules/rds/aws"
18    version = "~> 2.0"
19
20    identifier = format("gitops-rds-%s", var.environment)
21
22    engine               = "postgres"
23    engine_version       = "12.5"
24    instance_class       = "db.t2.micro"
25    allocated_storage    = 20
26    max_allocated_storage = 100
27    storage_encrypted    = false
28    multi_az             = true
29
30    # NOTE: Do NOT use 'user' as the value for 'username' as it throws:
31    # "Error creating DB Instance: InvalidParameterValue: MasterUsername
32    # user cannot be used as it is a reserved word used by the engine"
33    username = "gitops"
34
35    password = "Ly5ZesYuE8qKfBy2"
36    port     = "5432"
37
38    vpc_security_group_ids = [data.aws_security_group.default.id]
```

Figura 21 – Trecho de código do módulo *terraform-aws-rds*

Fonte: Criado pelo autor

4.2.6 Network Load Balancer - NLB

Como mencionado no capítulo 4, o *Elastic Load Balancing (ELB)* permite a criação de um balanceador de carga que distribui automaticamente o tráfego de entrada de instâncias *EC2*, contêineres e endereços *IPs*.

De acordo com a documentação da *AWS* (2021), os balanceadores de carga aceitos pelo *ELB* são: *Application Load Balancers (ALB), Network Load Balancers (NLB), Gateway Load Balancers (GWLB)*, ou balanceadores de carga clássicos. Para o desenvolvimento do trabalho o balanceador de carga escolhido foi o *NLB*.

Um *NLB* funciona na quarta camada do modelo *OSI*, possibilitando lidar com milhões de solicitações por segundo. Após o *NLB* receber uma solicitação de conexão, ela

é direcionada para o grupo de destino especificado nas regras e políticas do *NLB*. Desta forma, é aberta uma conexão *TCP* com o destino e a porta solicitada.

Neste trabalho, optou-se utilizar o balanceador de carga de alta performance *Nginx*, amplamente aplicado em empresas como: *StarBucks, Bank Of America, American Express* e *Telecom Italia*, segundo a equipe *Nginx* (2021).

Nginx também expõe rotas *HTTP* e *HTTPS* de fora do cluster para serviços dentro do *cluster*. Este tráfego é controlado por regras definidas nos arquivos *ingress* de cada aplicação (KUBERNETES, 2021).

Na Figura 22, adaptada de *Kubernetes* (2021), temos um exemplo de um *ingress* direcionando o tráfego externo para um serviço dentro do *cluster*.

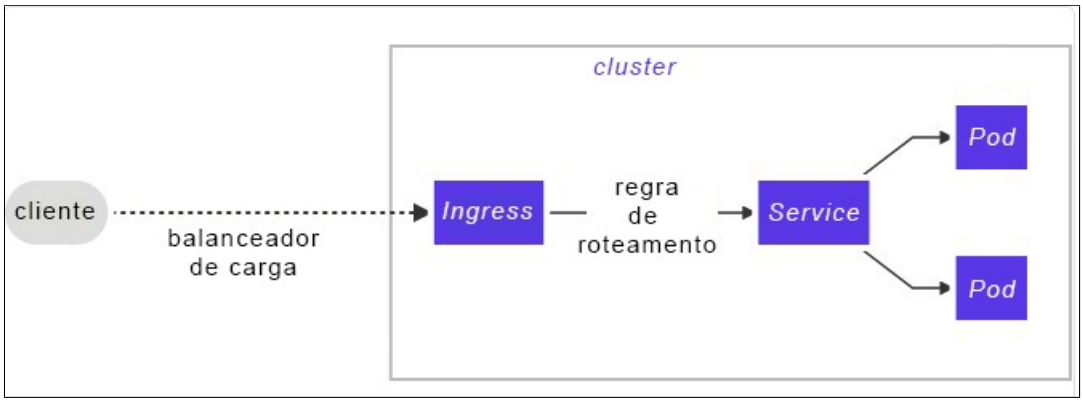

Figura 22 – Tráfego externo até serviço interno no *cluster*

Fonte: Criado pelo autor

Os manifestos necessários para a implantação do *Nginx*, tais como: *Namespace, Daemonset, ConfigMap*, e *Services* foram disponibilizados na página oficial do *Kubernetes* do *GitHub* no *link* `https://github.com/kubernetes/ingress-nginx`. Estes manifestos foram transformados do formato *YAML* para o *HCL* a fim de serem provisionados por *Terraform*.

Na Figura 23, temos um trecho do código responsável por criar o delimitador de projetos *Namespace*.

```
resource kubernetes_namespace "nginx" {
  metadata {
    name = "ingress-nginx"
    labels = {
      "app.kubernetes.io/name"    = "ingress-nginx"
      "app.kubernetes.io/part-of" = "ingress-nginx"
    }
  }
  lifecycle {
    ignore_changes = [
      metadata[0].annotations,
      metadata[0].labels,
    ]
  }
  timeouts {
    delete = "10m"
  }
  depends_on = [null_resource.module_depends_on]
}
```

Figura 23 – Trecho de código do *Namespace Nginx*

Fonte: Criado pelo autor

Já na Figura 24, podemos observar uma fração do código em *HCL* do DaemonSet. Recurso semelhante ao *Deployment*, mas que garante que todos os nós do *cluster* executem uma cópia do *pod*.

```
resource "kubernetes_daemonset" "nginx" {
  metadata {
    name      = "nginx-ingress-controller"
    namespace = kubernetes_namespace.nginx.metadata.0.name
    labels = {
      "app.kubernetes.io/name"    = "ingress-nginx"
      "app.kubernetes.io/part-of" = "ingress-nginx"
    }
  }
...                                   ...
      spec {
        termination_grace_period_seconds = 0
        service_account_name             = kubernetes_service_account.nginx.metadata.0.name
        automount_service_account_token  = "true"
        host_network                     = "true"
        container {
          image = "quay.io/kubernetes-ingress-controller/nginx-ingress-controller:0.32.0"
          name  = "nginx-ingress-controller"
          args = [
            "/nginx-ingress-controller",
            "--configmap=$(POD_NAMESPACE)/nginx-configuration",
            "--tcp-services-configmap=$(POD_NAMESPACE)/tcp-services",
            "--udp-services-configmap=$(POD_NAMESPACE)/udp-services",
            "--publish-service=$(POD_NAMESPACE)/ingress-nginx",
            "--annotations-prefix=nginx.ingress.kubernetes.io",
          ]
```

Figura 24 – Trecho de código do *Daemonset Nginx*

Fonte: Criado pelo autor

Na Figura 25 temos o *ConfigMap* responsável por guardar as variáveis ou dados

não confidenciais da aplicação.

```
resource "kubernetes_config_map" "nginx" {
  metadata {
    name      = "nginx-configuration"
    namespace = kubernetes_namespace.nginx.metadata.0.name
    labels = {
      "app.kubernetes.io/name"    = "ingress-nginx"
      "app.kubernetes.io/part-of" = "ingress-nginx"
    }
  }

  lifecycle {
    ignore_changes = [
      metadata[0].annotations,
      metadata[0].labels,
    ]
  }

  data = {
    enable-underscores-in-headers = "True"
    server-tokens                 = "False"
    proxy-body-size               = "10m"
    ignore-invalid-headers        = "True"
    ssl-ciphers                   = "ECDHE-RSA-AES256-GCM-SHA384:ECDHE-RSA-CHACHA20-
  }
}
```

Figura 25 – Trecho de código do *ConfigMap Nginx*

Fonte: Criado pelo autor

Por fim, na Figura 26 podemos observar uma parte do código do recurso *Service* que define um conjunto lógico de políticas de acesso nas portas do *NLB*.

Figura 26 – Trecho de código do *Service Nginx*

Fonte: Criado pelo autor

4.2.7 Simple Storage Service - S3

Como já abordado em Ambiente de Infraestrutura, o *S3* é um serviço de armazenamento que comporta qualquer tipo de dado.

O *S3* armazena estes dados dentro de *buckets*. Para o desenvolvimento do trabalho foi criado um *bucket* de forma manual com o nome *gitops-tfstate*, encarregado de armazenar os estados da infraestrutura já provisionada e da aplicação utilizada no trabalho.

A utilização do *S3* em *Terraform* é feita a partir da configuração de um *backend S3* e seus atributos são:

- *bucket*: Nome do *bucket* que armazenará os dados;
- *key*: Caminho no *bucket* que será utilizado para o armazenamento deste dado;
- *region*: Região da *AWS* na qual o *bucket* está configurado.

Estes atributos podem ser declarados tanto ndo arquivo *backend.tf*, quanto no comando *terraform init*. Neste caso, os atributos foram declarados no arquivo *.gitlab-ci.yml*, arquivo este que fornece a configuração do *GitLab CI/CD* que veremos mais à frente.

4.3 Comandos do Terraform

Nesta seção os comandos do *Terraform* são descritos: *terraform init, terraform validate, terraform plan, terraform apply* e *terraform destroy*. Estes comandos servem para transformar todas as definições dos recursos da *AWS* declarados nos arquivos de configuração *HCL* em algo real, tranformando-os em instâncias de serviço.

4.3.1 terraform init

De acordo com a documentação do *Terraform* (2021), o comando *terraform init* é utilizado para inicializar o diretório de trabalho do *Terraform* que contém os arquivos de configuração em *HCL* no formato *.tf*.

Este é o primeiro comando a ser executado quando trabalhamos com *Terraform*, pois é o responsável por baixar as dependências necessárias para o provisionamento dos recursos e atualizá-los de acordo com as versões especificadas no arquivo de versões (*versions.tf*). Este comando também serve para acessar o armazenamento no qual o estado do *Terraform* será salvo, através das configurações presentes no arquivo *backend.tf*, ou ler um estado já armazenado com intuito de realizar atualizações nos objetos provisionados por ele.

Na Figura 27, podemos observar a execução do *terraform init* responsável por baixar as dependências utilizadas para o trabalho e acessar o *bucket gitops-tfstate* do *S3* , utilizado para armazenar o estado da infraestrutura e das aplicações.

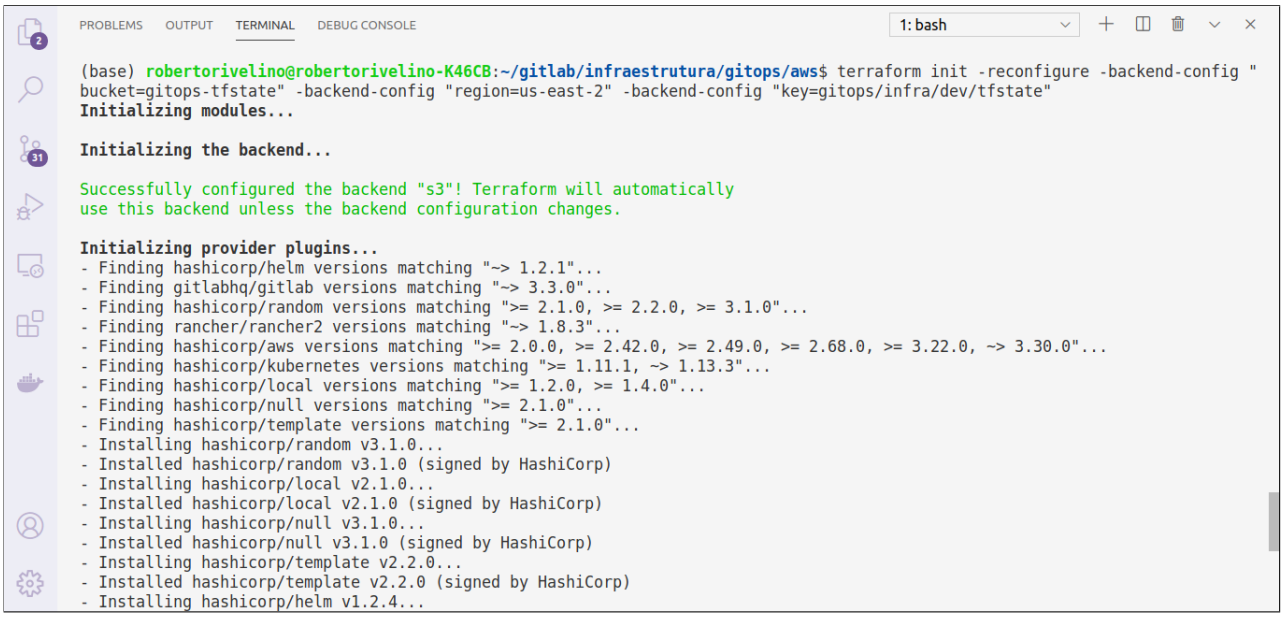

Figura 27 – Comando *terraform init*

Fonte: Criado pelo autor

4.3.2 TERRAFORM VALIDATE

De acordo com a documentação do *Terraform* (2021), o comando *terraform validate* realiza a validação dos arquivos de configuração *.tf* presentes no projeto raiz. Nesta validação é verificado se o código é sintaticamente válido. Caso haja algum problema relacionado à sintaxe, este possível erro é mostrado na saída do comando, bem como sua solução.

Na Figura 28 tems um exemplo demonstrando a execução do comando *terraform validate* com saída bem-sucedida.

Figura 28 – Comando *terraform validate* com saída bem-sucedida

Fonte: Criado pelo autor

Já a Figura 29, mostra a execução do mesmo comando com a saída alertando erros de sintaxe.

Figura 29 – Comando *terraform validate* com erros de sintaxe

Fonte: Criado pelo autor

4.3.3 TERRAFORM PLAN

Conforme a documentação do *Terraform* (2021), o comando *terraform plan* cria um plano de execução. Este plano é utilizado para verificar se já existe um estado atual do ambiente ou aplicação, e caso exista, quais recursos devem ser atualizados. Caso não exista, todos os recursos a serem provisionados pelo *Terraform* são apresentados.

Este comando serve como um facilitador, mostrando o conjunto de mudanças que devem ser realizadas no estado, sem aplicá-las. Assim, é possível garantir que nenhuma alteração será aplicada sem seu devido conhecimento.

Na Figura 30 podemos observar a execução do *terraform plan* e sua saída:

```
(base) robertorivelino@robertorivelino-K46CB:~/gitlab/infraestrutura/gitops/aws$ terraform plan -var="environment=dev" -out=tf
.plan
                                    ...

  # module.rds.module.db_subnet_group.aws_db_subnet_group.this[0] will be created
  + resource "aws_db_subnet_group" "this" {
      + arn         = (known after apply)
      + description = "gitops-rds-dev subnet group"
      + id          = (known after apply)
      + name        = (known after apply)
      + name_prefix = "gitops-rds-dev-"
      + subnet_ids  = (known after apply)
      + tags        = {
          + "Environment" = "dev"
          + "Name"        = "gitops-rds-dev"
          + "terraform"   = "true"
        }
    }

Plan: 91 to add, 0 to change, 0 to destroy.

Changes to Outputs:
  + efs_dns_name       = (known after apply)
  + rancher_cluster_id = (known after apply)
```

Figura 30 – Execução do *terraform plan*

Fonte: Criado pelo autor

4.3.4 TERRAFORM APPLY

Segundo a documentação do *Terraform* (2021), o comando *terraform apply* é utilizado de forma a aplicar as alterações necessárias para atingir o estado desejado da infraestrutura ou aplicação.

Nele também é feita a leitura do plano gerado anteriormente no *terraform plan* e aplicado suas alterações.

Na Figura 31 podemos observar a execução do comando *terraform apply* e os *logs* que mostram a criação dos recursos mencionados no capítulo 4.1.

```
(base) robertorivelino@robertorivelino-K46CB:~/gitlab/infraestrutura/gitops/aws$ terraform apply tf.plan
module.rds.module.db_instance.random_id.snapshot_identifier[0]: Creating...
module.rds.module.db_instance.random_id.snapshot_identifier[0]: Creation complete after 0s [id=QH6fgg]
module.eks.aws_iam_policy.cluster_elb_sl_role_creation[0]: Creating...
module.efs.aws_efs_file_system.default[0]: Creating...
module.rds.module.db_parameter_group.aws_db_parameter_group.this[0]: Creating...
module.vpc.aws_vpc.this[0]: Creating...
module.eks.aws_iam_role.cluster[0]: Creating...
module.eks.aws_iam_policy.cluster_elb_sl_role_creation[0]: Creation complete after 4s [id=arn:aws:iam::535034659307:policy/gitops-eks-dev-elb-sl-role-creation2
021040322325526950000002]
module.eks.aws_iam_role.cluster[0]: Creation complete after 8s [id=gitops-eks-dev-2021040322325527620000004]
module.eks.aws_iam_role_policy_attachment.cluster_AmazonEKSServicePolicy[0]: Creating...
module.eks.aws_iam_role_policy_attachment.cluster_AmazonEKSVPCResourceControllerPolicy[0]: Creating...
module.eks.aws_iam_role_policy_attachment.cluster_elb_sl_role_creation[0]: Creating...
module.eks.aws_iam_role_policy_attachment.cluster_AmazonEKSClusterPolicy[0]: Creating...
module.rds.module.db_parameter_group.aws_db_parameter_group.this[0]: Creation complete after 10s [id=gitops-rds-dev-20210403223255268600000001]
module.eks.aws_iam_role_policy_attachment.cluster_AmazonEKSVPCResourceControllerPolicy[0]: Creation complete after 2s [id=gitops-eks-dev2021040322325527620000
004-20210403223303810200000007]
module.eks.aws_iam_role_policy_attachment.cluster_elb_sl_role_creation[0]: Creation complete after 2s [id=gitops-eks-dev-2021040322325527620000004-202104032233
03802700000005]
module.eks.aws_iam_role_policy_attachment.cluster_AmazonEKSServicePolicy[0]: Creation complete after 2s [id=gitops-eks-dev-2021040322325527620000004-2021040322
3303805300000006]
module.eks.aws_iam_role_policy_attachment.cluster_AmazonEKSClusterPolicy[0]: Creation complete after 2s [id=gitops-eks-dev-2021040322325527620000004-2021040322
3303877800000008]
module.efs.aws_efs_file_system.default[0]: Still creating... [10s elapsed]
module.vpc.aws_vpc.this[0]: Still creating... [10s elapsed]
module.efs.aws_efs_file_system.default[0]: Creation complete after 10s [id=fs-8243a2f9]
gitlab_group_variable.dev_efs_dns[0]: Creating...
gitlab_group_variable.dev_efs_dns[0]: Creation complete after 1s [id=378:dev_efs_dns]
module.vpc.aws_vpc.this[0]: Still creating... [20s elapsed]
module.vpc.aws_vpc.this[0]: Creation complete after 27s [id=vpc-0208eb7625cbc2b05]
data.aws_vpc.selected: Reading...
module.vpc.aws_internet_gateway.this[0]: Creating...
module.vpc.aws_subnet.private[0]: Creating...
module.efs.aws_security_group.efs[0]: Creating...
```

Figura 31 – Execução do comando *terraform apply*

Fonte: Criado pelo autor

Ao final da execução é criado ou atualizado o arquivo de estado do *Terraform*, que tem um papel fundamental no desenvolvimento do trabalho.

4.3.5 TERRAFORM DESTROY

De acordo com a documentação do *Terraform* (2021), o comando *terraform destroy* é utilizado para destruir os recursos provisionados pelo *Terraform*. Na execução deste comando é realizada a leitura completa do estado gerado da infraestrutura ou da aplicação, e a partir das informações obtidas é realizada a destruição dos recursos atrelados a este estado.

Na Figura 32 vemos a execução do comando *terraform destroy*, e os *logs* que demonstram a destruição dos recursos.

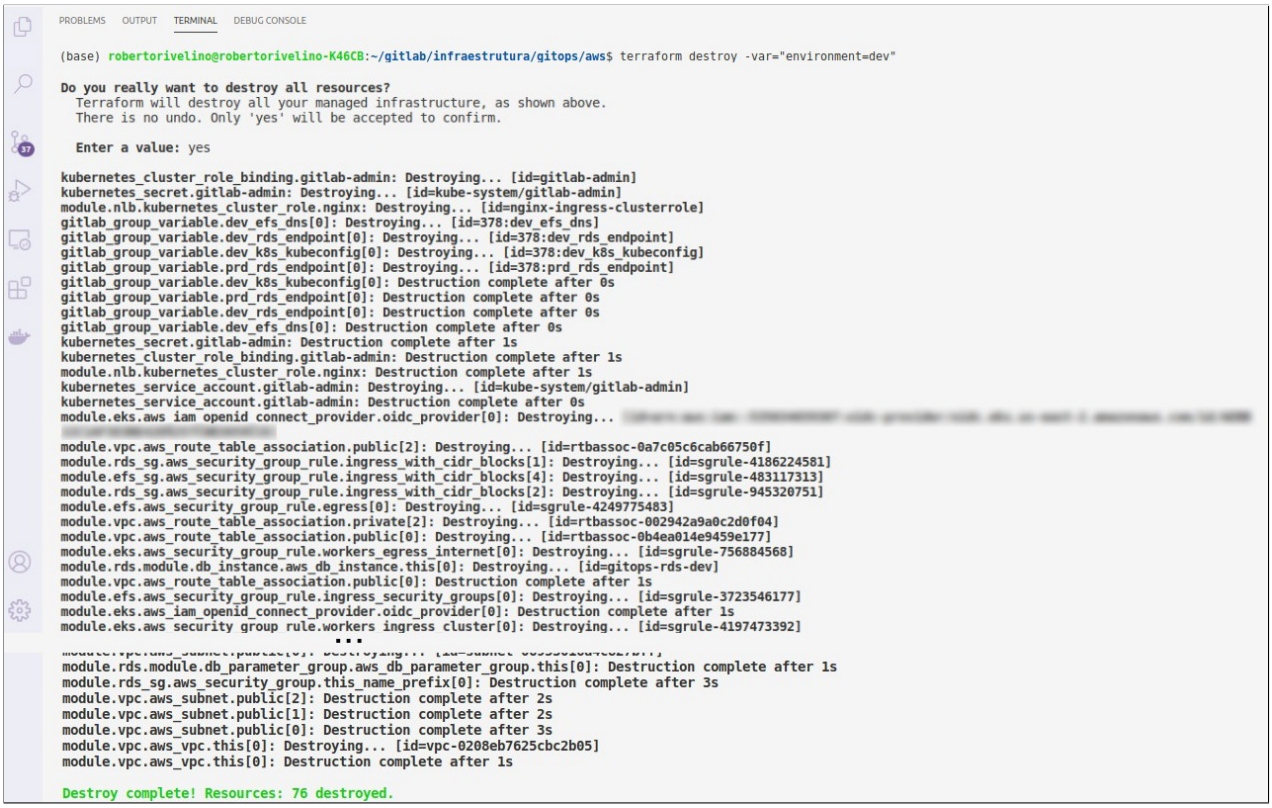

Figura 32 – Execução do comando *terraform apply*

Fonte: Criado pelo autor

4.4 GitLab CI/CD

Nesta seção, é apresentado o arquivo de configuração do *GitLab CI/CD* chamado *.gitlab-ci.yml*. Este arquivo é responsável pelo controle do fluxo de tarefas (*pipeline*) do projeto. Nele, encontramos os comandos executados antes de cada estágio (*before_scritps*), o primeiro estágio que valida a sintaxe dos arquivos (*Validate*), o estágio de planejamento

dos recursos provisionados ou atualizados (*Plan*), o estágio de execução do plano (*Apply*) e o estágio de destruição dos recursos provisionados *(Destroy)*.

4.4.1 BEFORE_SCRIPTS

De acordo com o *GitLab Inc.* (2021), *before_scripts* é uma palavra-chave (*keyword*) que permite escrever um conjunto de comandos que sempre são executados antes da execução de cada estágio pertencente ao *pipeline*.

Para o trabalho, *before_scripts* foi utilizado para executar o comando *terraform init*, necessário em todos os estágios para que a inicialização do *Terraform* seja realizada, efetivando também a leitura do estado no *bucket gitops-tfstate*.

Na Figura 33 é possível ver a parte do código do arquivo *.gitlab-ci.yml* que mostra o *before_scripts*.

```
19  before_script:
20    - terraform --version
21    - terraform init -reconfigure -backend-config "bucket=$S3_BUCKET"
22      -backend-config "region=$S3_REGION"
23      -backend-config "key=gitops/$CI_COMMIT_REF_NAME/tfstate"
```

Figura 33 – Código *before_scripts*

Fonte: Criado pelo autor

4.4.2 VALIDATE

Este estágio serve para executar o comando *terraform validate* descrito anteriormente, tornando possível validar a sintaxe dos arquivos de configuração *.tf* ao rodar o *pipeline* do *GitLab CI/CD*.

Na Figura 34 podemos observar o estágio *validate* em código.

```
29  validate:
30    stage: validate
31    script:
32      - cd ${TF_ROOT}
33      - terraform validate
34      - terraform fmt -check=true
35    only:
36      - branches
37    tags:
38      - dc43
```

Figura 34 – Estágio *validate*

Fonte: Criado pelo autor

Os *logs* do estágio *validate* são mostrados na interface gráfica do *GitLab*, nos permitindo verificar se a validação ocorreu com sucesso. Nas figuras 35 e 36 podemos observar o histórico do *pipeline* e os *logs* do estágio de validação.

Figura 35 – Histórico *Validate* no *GitLab*

Fonte: Criado pelo autor

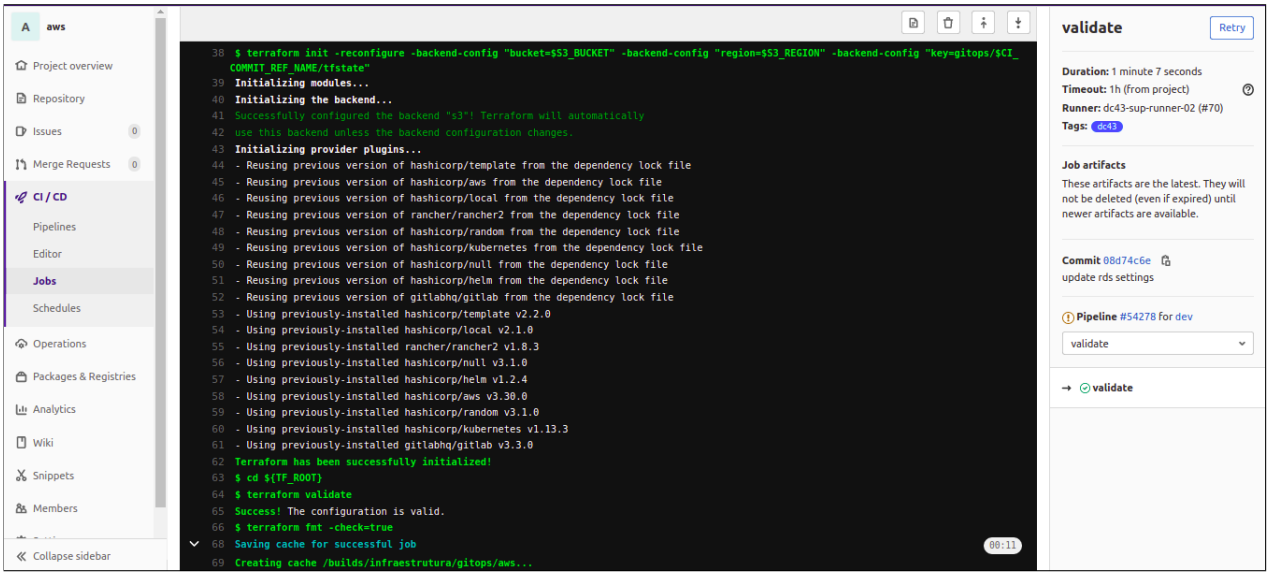

Figura 36 – *Logs* do estágio de validação.

Fonte: Criado pelo autor

4.4.3 Plan

Neste estágio, o comando a ser executado é o *terraform plan* descrito anteriormente, o que torna possível criar um arquivo de plano *tf.plan* com todas as criações ou alterações que devem ser aplicadas na *AWS*.

A Figura 37 mostra um trecho de código do arquivo *.gitlab-ci.yml* referente ao estágio *plan* do *Terraform*.

```
69  plan production:
70    stage: plan
71    script:
72      - cd ${TF_ROOT}
73      - terraform plan -var="environment=$CI_COMMIT_REF_NAME" -out=$PLAN
74    artifacts:
75      name: plan
76      paths:
77        - $PLAN
78    only:
79      - prd
80      - hom
81      - dev
82    tags:
83      - dc43
```

Figura 37 – Estágio *plan*

Fonte: Criado pelo autor

As figuras 38, 39 e 40 representam o histórico do estágio *plan* no *GitLab* e o plano de criação dos recursos declarados em *HCL* que serão provisionados na *AWS*, em forma de *logs*.

Figura 38 – Histórico do estágio *plan*

Fonte: Criado pelo autor

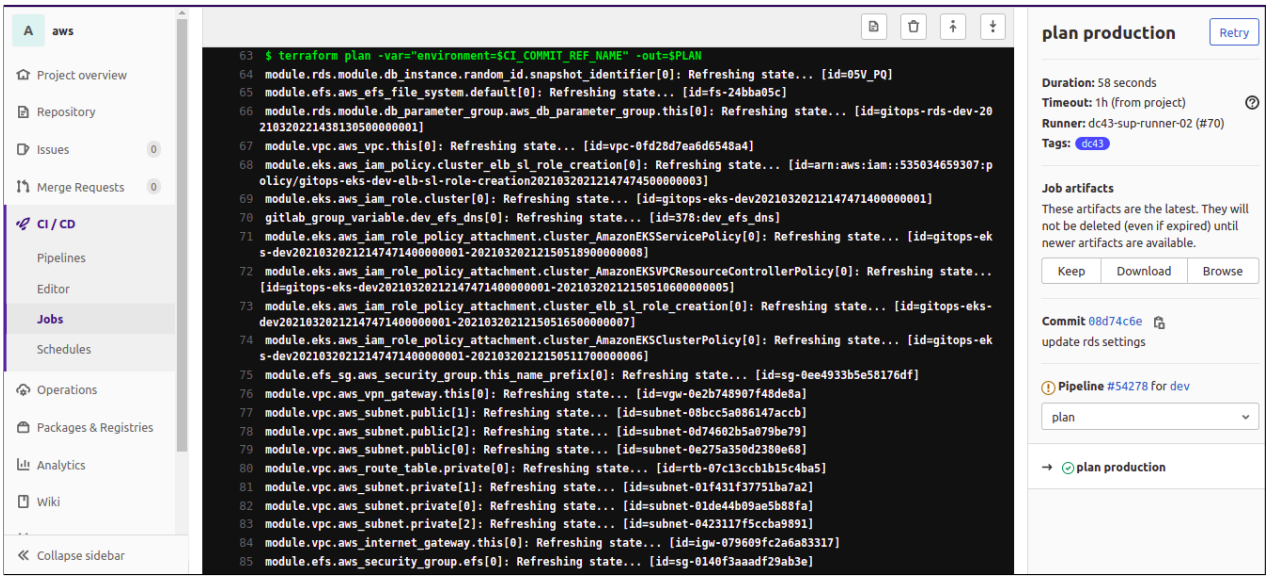

Figura 39 – Verificação do estado atual

Fonte: Criado pelo autor

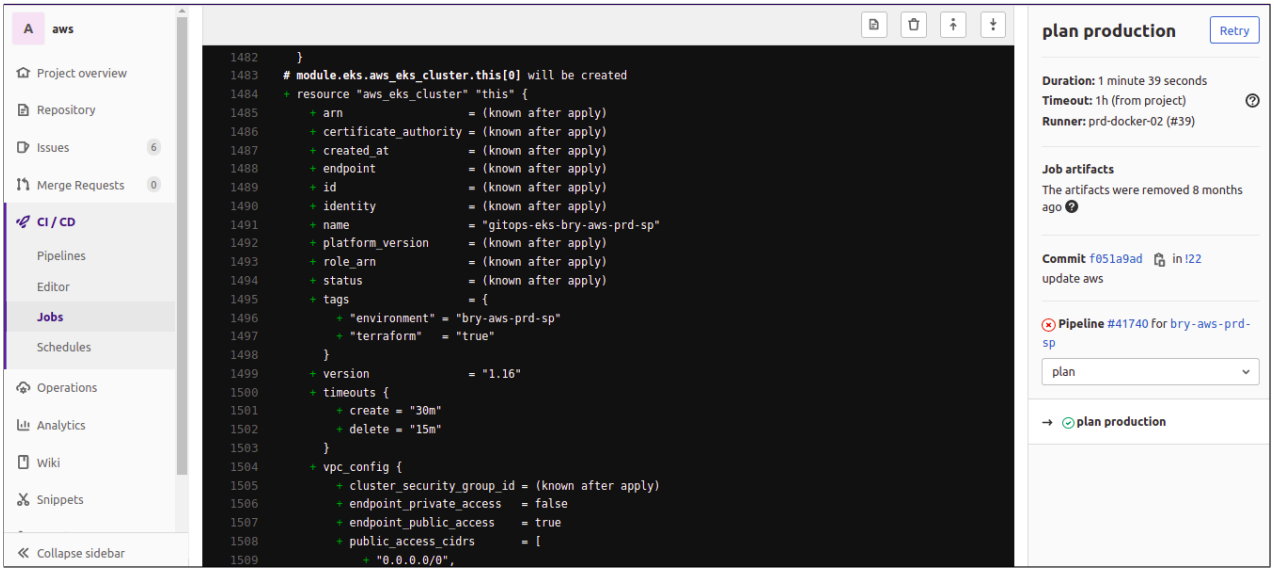

Figura 40 – Plano de criação ou alteração dos recursos

Fonte: Criado pelo autor

4.4.4 Apply

Neste estágio ocorre a execução do *terraform apply*, dando início ao provisionamento dos recursos *VPC, EKS, RDS, EFS, SGs* e *NLB*, levando como argumento o arquivo *tf.plan* gerado no estágio anterior.

Temos a Figura 41 que retrata uma parte do código referente ao estágio *apply*.

```
85  apply:
86    stage: apply
87    script:
88      - cd ${TF_ROOT}
89      - terraform apply -auto-approve -input=false $PLAN
90    dependencies:
91      - plan production
92    artifacts:
93      name: $CI_COMMIT_REF_SLUG
94      untracked: true
95    only:
96      - prd
97      - hom
98      - dev
99    tags:
100     - dc43
```

Figura 41 – Estágio *apply*

Fonte: Criado pelo autor

As figuras 42 e 43 mostram o histórico do estágio *apply* e os *logs* de provisionamento dos recursos.

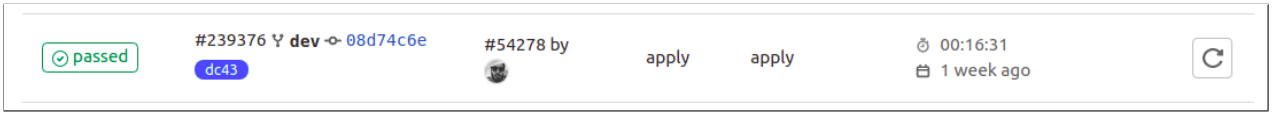

Figura 42 – Histórico do *apply*

Fonte: Criado pelo autor

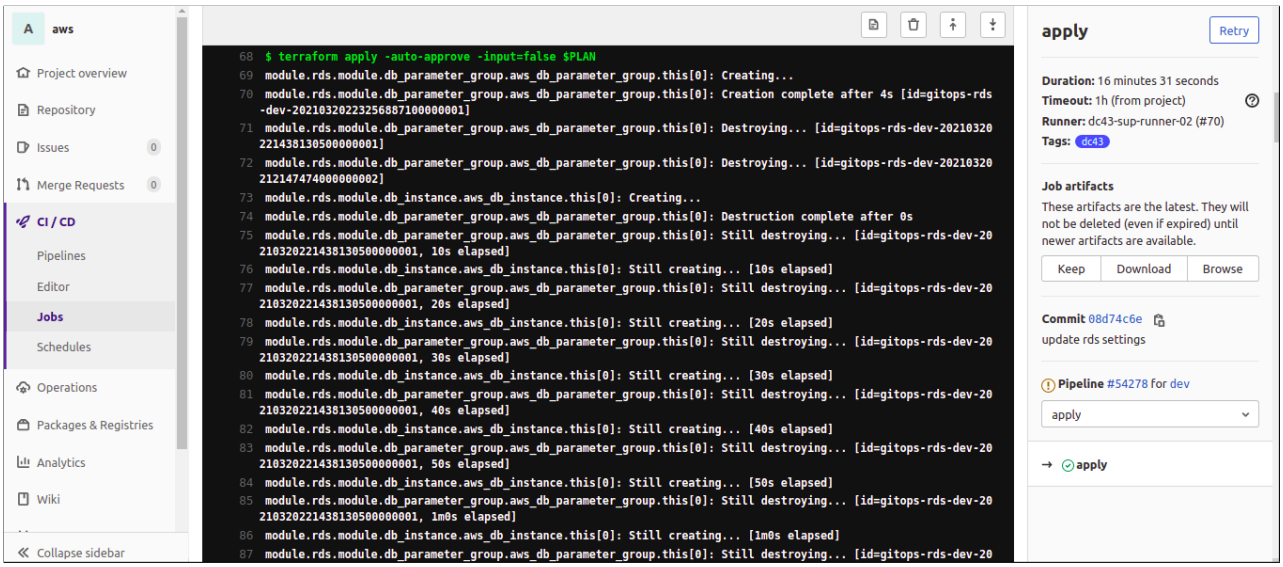

Figura 43 – *Logs* de provisionamento dos recursos

Fonte: Criado pelo autor

4.4.5 Destroy

No estágio *destroy* é executado o comando *terraform destroy* mencionado anteriormente. Na Figura 44 temos um trecho de código referente ao estágio *destroy*.

```
120 destroy:
121   stage: destroy
122   script:
123     - cd ${TF_ROOT}
124     - terraform destroy -auto-approve -var="environment=$CI_COMMIT_REF_NAME"
125   dependencies:
126     - apply
127   when: manual
128   only:
129     - prd
130     - hom
131     - dev
132   tags:
133     - dc43
```

Figura 44 – Estágio *destroy*

Fonte: Criado pelo autor

Já a Figura 45 mostra o histórico do estágio *destroy* no *GitLab*.

Figura 45 – Histórico do estágio *destroy*

Fonte: Criado pelo autor

4.5 GitOps

Este tópico contempla o processo completo da metodologia *GitOps*. Através da adoção desta metodologia, é possível concentrar os arquivos de declaração da infraestrutura e da aplicação exemplo em um único Sistema de Controle de Versão (*VCS*). Esta etapa mantém o estado atual dos projetos em sincronia com o ambiente de infraestrutura proposto, através da Entrega e Integração Contínua (*CI/CD*).

4.5.1 Variáveis de ambiente

Para ter acesso à *AWS* através do *GitLab*, é necessário a criação de variáveis de ambiente que permitam a autenticação ao sistema. Sendo elas:

- *AWS_ACCESS_KEY_ID*: Chave de acesso *AWS* usada para autenticar o usuário;
- *AWS_SECRET_ACCESS_KEY*: Senha de acesso *AWS* usada para autenticar o usuário.

Estas variáveis são utilizadas no *GitLab CI/CD*, permitindo a autenticação e execução do fluxo de tarefas na conta da *AWS*.

Há também a criação da variável de ambiente *dev_k8s_kubeconfig* que permite acesso ao *cluster* através do binário *kubectl*, podendo assim criar, modificar ou remover manifestos que compõem as aplicações.

Este arquivo chamado *kubeconfig* é utilizado para armazenar informações de autenticação do *cluster Kubernetes*. A variável é gerada no estágio de *Apply* assim que o provisionamento do *EKS* é finalizado.

Na Figura 46 podemos observar uma parte do código que gera a variável *dev_k8s_kubeconfig*.

```
17  // APPS - Kubeconfig
18  resource "gitlab_group_variable" "dev_k8s_kubeconfig" {
19    group         = 378
20    key           = "dev_k8s_kubeconfig"
21    value         = module.eks.kubeconfig
22    variable_type = "file"
23    protected     = false
24
25    count = var.environment == "dev" ? 1 : 0
26    # lifecycle {
27    #   prevent_destroy = true
28    # }
29  }
```

Figura 46 – Código que cria a variável *dev_k8s_kubeconfig*

Fonte: Criado pelo autor

Na Figura 47 são mostradas as variáveis citadas acima pela interface gráfica do *GitLab*.

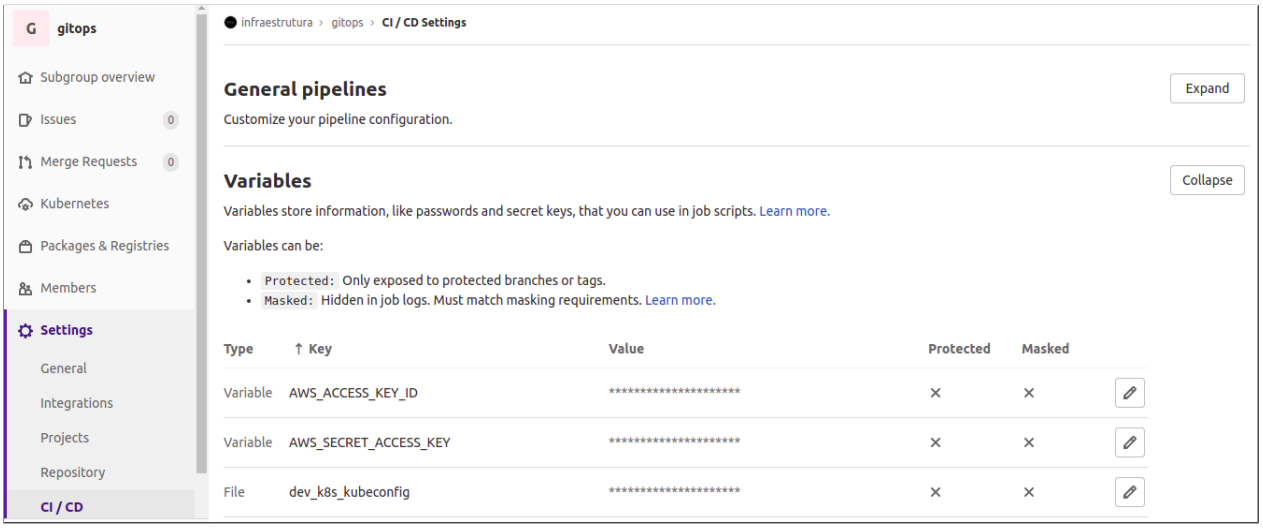

Figura 47 – Variáveis de ambiente

Fonte: Criado pelo autor

4.5.2 Provisionamento da Infraestrutura

Primeiramente, os arquivos de configuração *.tf* do *Terraform* são registrados (*commit*) em um projeto no *GitLab*, possibilitando o disparo automático do fluxo de tarefas presente no arquivo *.gitlab-ci.yml*.

Para este trabalho, foi criado um projeto chamado *aws* com a ramificação *dev*, que remete a um ambiente de desenvolvimento.

Na Figura 48 podemos ver o primeiro registro do projeto, que transfere os arquivos de configuração do *Terraform* para o *GitLab*.

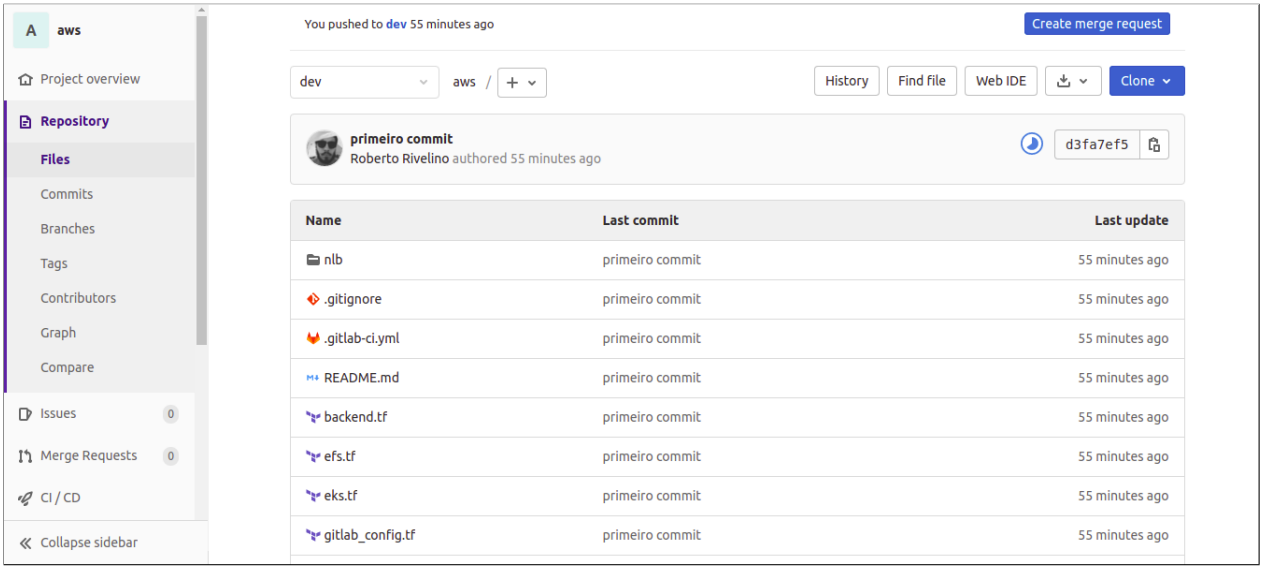

Figura 48 – Primeiro registro no projeto *aws*

Fonte: Criado pelo autor

Quando um registro é gerado, o fluxo de tarefas dispara e damos início ao provisionamento dos recursos na *AWS*.

O primeiro fluxo de tarefas executado é encarregado de gerar o estado do *Terraform* no *bucket*, estado este atualizado toda vez que alguma alteração for feita nos arquivos de configuração do *Terraform* através do disparo de um novo fluxo de tarefas.

Na Figura 49 podemos observar o primeiro fluxo de tarefas e seus estágios *Validate*, *Plan*, *Apply*, e *Destroy* no *VCS*.

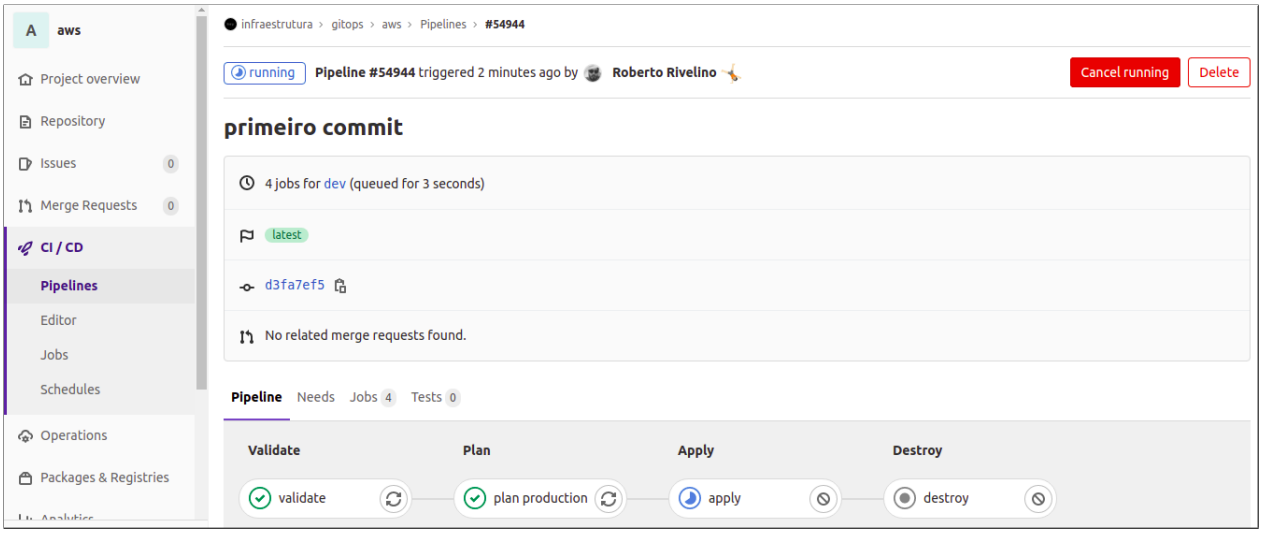

Figura 49 – Primeiro fluxo de tarefas (*pipeline*)

Fonte: Criado pelo autor

A cada novo *commit*, é criado um registro de alterações no *VCS*. Este registro possibilita ter um controle de versionamento dos arquivos modificados. Na Figura 50 temos um exemplo de versionamento do arquivo de configuração *rds.tf*.

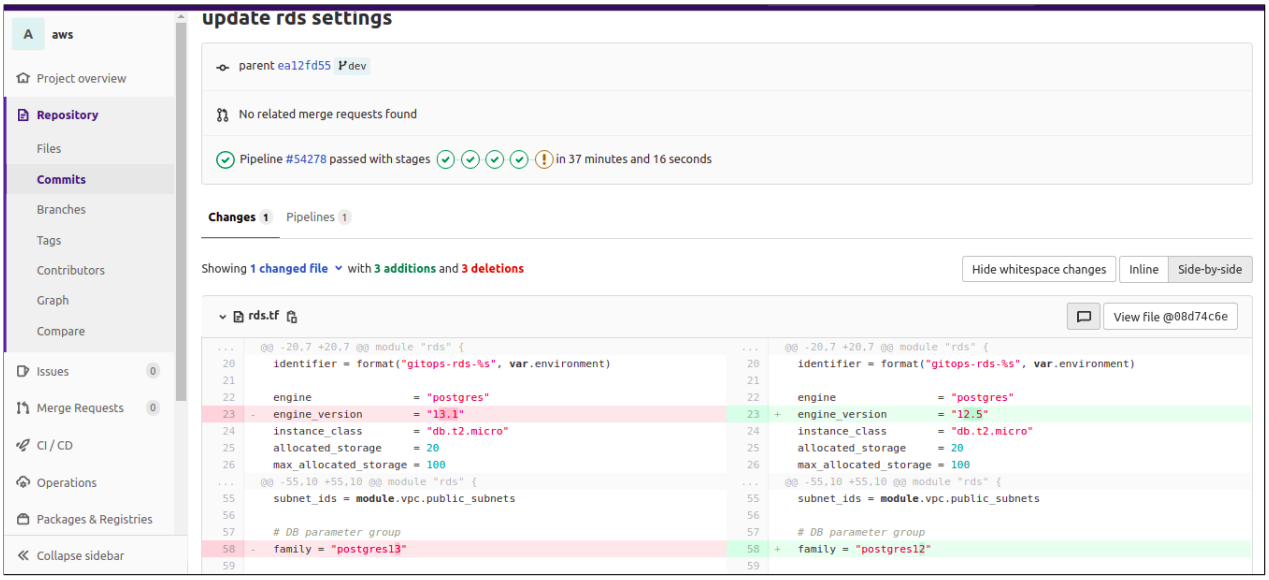

Figura 50 – Controle de versão do arquivo *rds.tf*

Fonte: Criado pelo autor

Durante a execução do *Apply* já é possível visualizar a criação dos recursos na interface gráfica da *AWS*. Nas figuras 51, 52, 53, 54, 55 e 56 podemos observar os recursos *RDS, EKS, EFS, VPC*, EC2 e *NLB*, respectivamente.

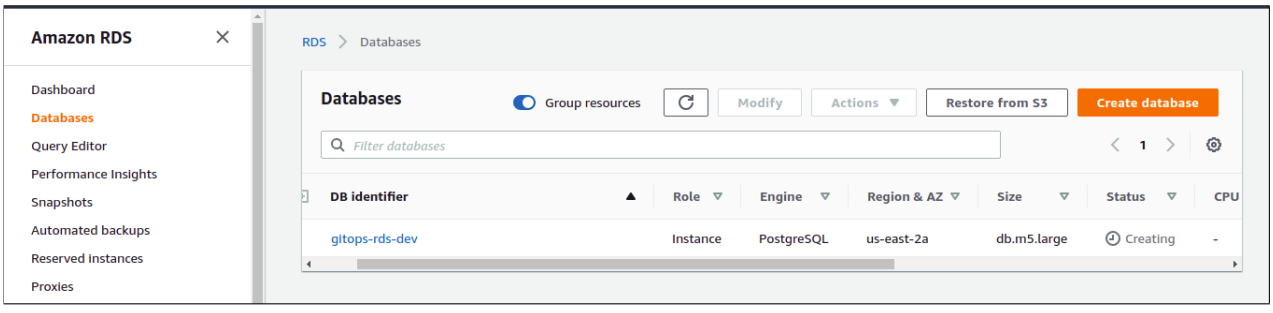

Figura 51 – *Relational Database Service - RDS*

Fonte: Criado pelo autor

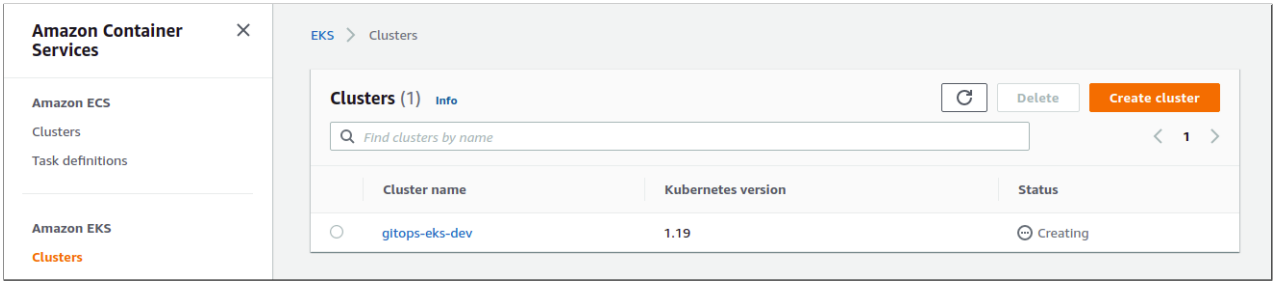

Figura 52 – *Elastic Kubernetes Service - EKS*

Fonte: Criado pelo autor

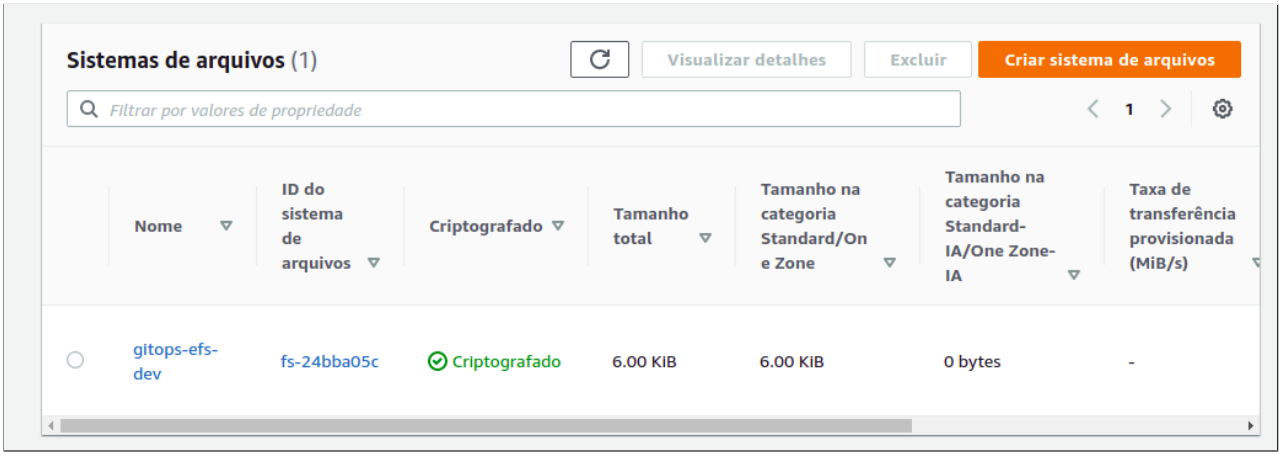

Figura 53 – *Elastic File System - EFS*

Fonte: Criado pelo autor

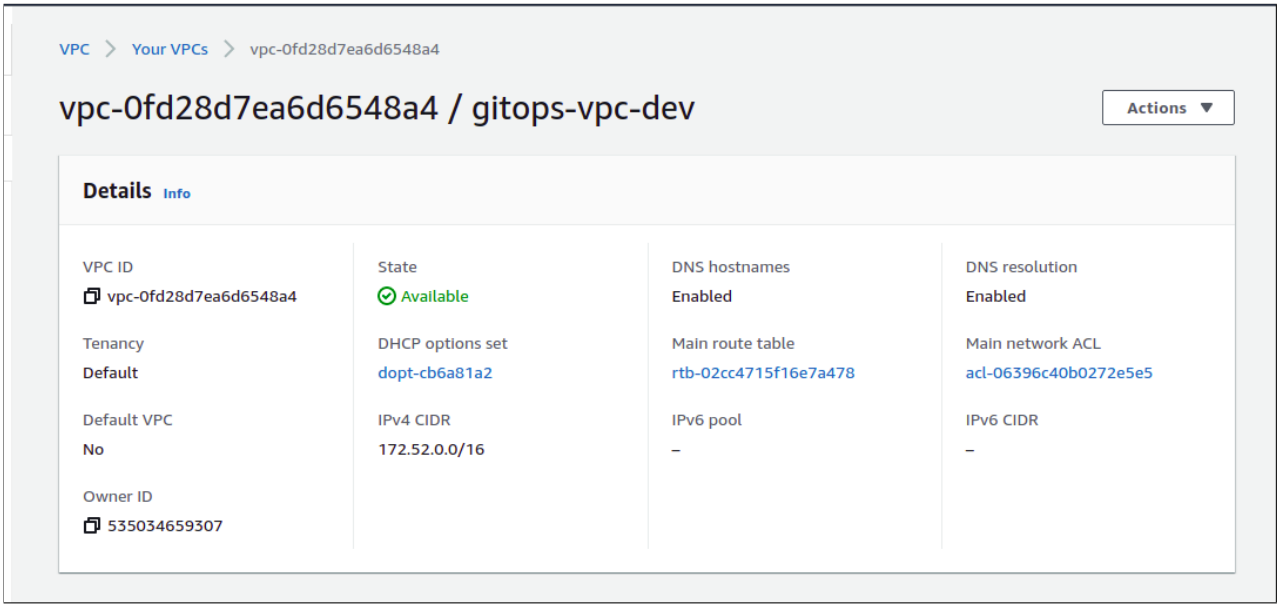

Figura 54 – *Virtual Private Cloud - VPC*

Fonte: Criado pelo autor

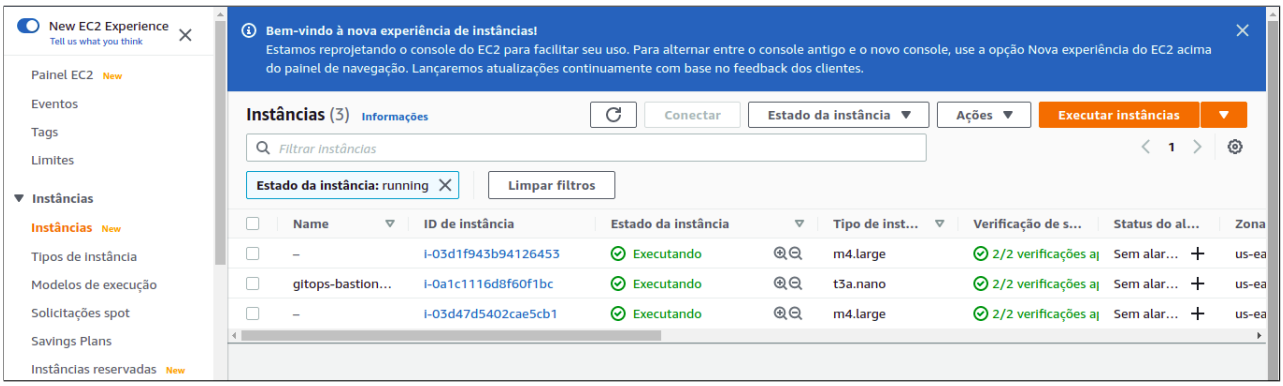

Figura 55 – *Elastic Computing - EC2*

Fonte: Criado pelo autor

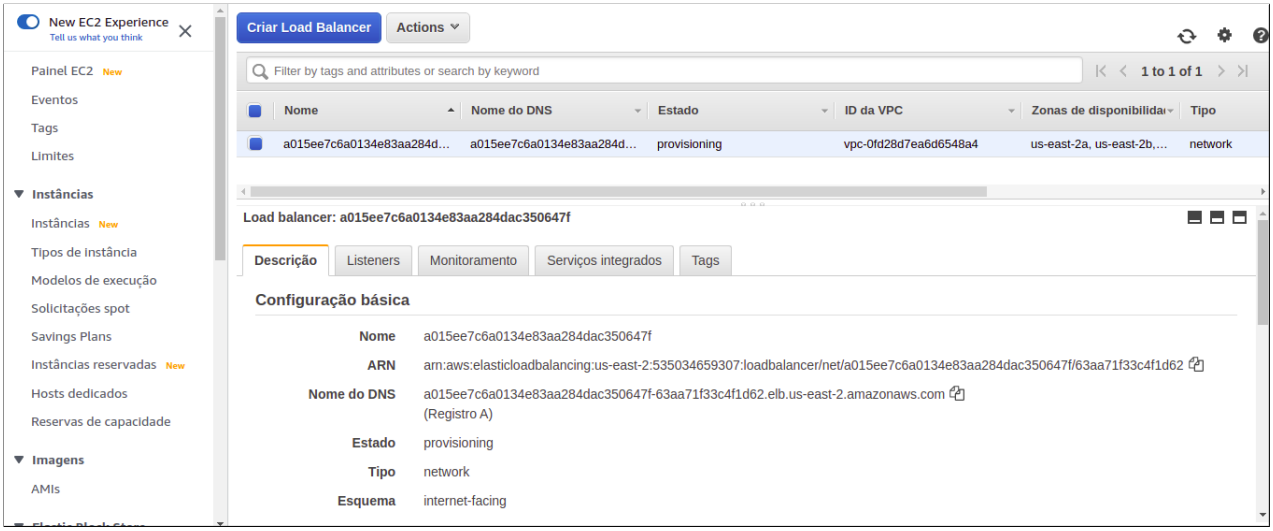

Figura 56 – *Network Load Balancer - NLB*

Fonte: Criado pelo autor

Ao final da execução do *Apply* é gerado o arquivo de estado do *Terraform* que é armazenado no *bucket gitops-tfstate*, como mostra a Figura 57.

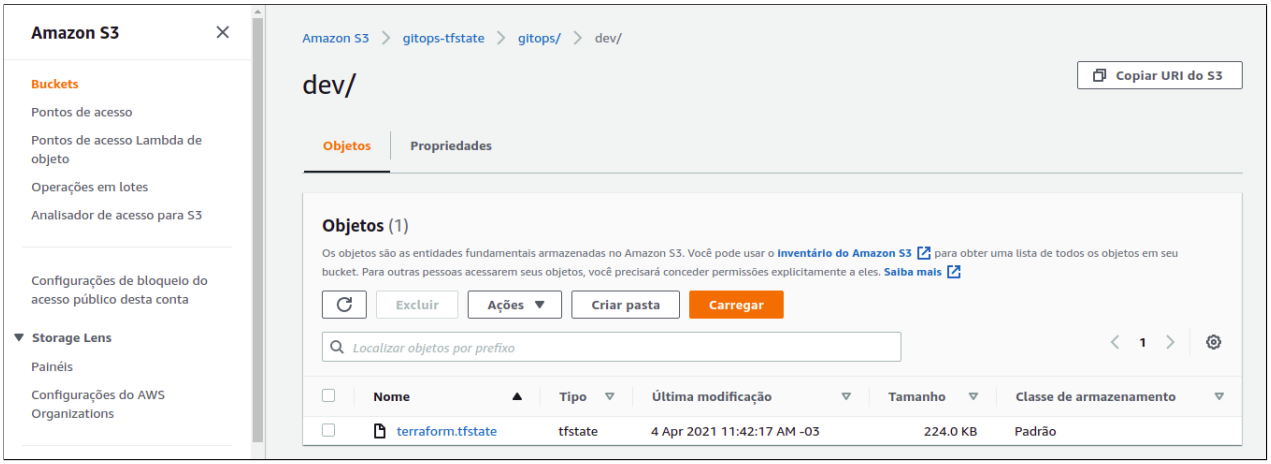

Figura 57 – Estado do *Terraform* salvo no *bucket*

Fonte: Criado pelo autor

Na Figura 58 temos um trecho do código do estado, revelando o vínculo entre os recursos criados e seus identificadores.

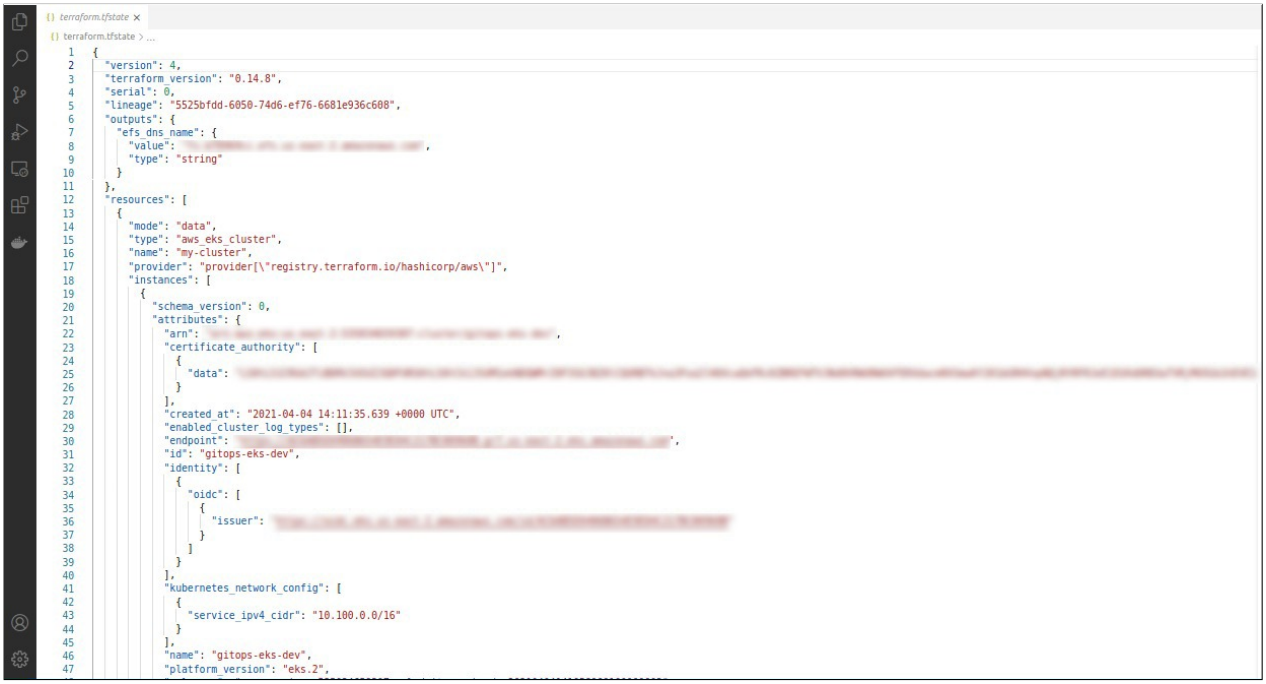

Figura 58 – Trecho de código do *terraform.tfstate*

Fonte: Criado pelo autor

4.6 Aplicação exemplo

Neste tópico é abordada a criação de uma aplicação exemplo no *cluster Kubernetes* através do fluxo de tarefas do *GitLab CI/CD* utilizando a metodologia *GitOps*.

4.6.1 echoserver

A aplicação escolhida para a criação através da Infraestrutura como Código utilizando as diretrizes da metodologia *GitOps* foi o *echoserver*, que mostra as informações de cabeçalho, servidor e cliente de um servidor *web*.

Os manifestos desta aplicação foram transformados de *YAML* para *HCL*. Os arquivos originais podem ser encontrados no *link* `https://gist.github.com/chukaofili/d0a6713734d0953ce1ce667958464edb`.

Para o provisionamento do *echoserver* foram necessários os arquivos de configuração *Namespace, Deployment, Service,* e *Ingress* que estão presentes nas figuras 59, 60, 61 e 62, respectivamente.

```
resource "kubernetes_namespace" "echoserver" {
  metadata {
    annotations = {
    }

    name = "echoserver"
  }
  lifecycle {
    ignore_changes = [
      metadata[0].annotations,
      metadata[0].labels,
    ]
  }

}
```

Figura 59 – Trecho de código do *Namespace* do *echoserver*

Fonte: Criado pelo autor

```
resource "kubernetes_deployment" "echoserver-deploy" {
  metadata {
    name      = "echoserver-deploy"
    namespace = kubernetes_namespace.echoserver.metadata.0.name
    labels = {
      app = "echoserver"
    }
  }

  lifecycle {
    ignore_changes = [
      metadata[0].annotations,
      metadata[0].labels,
    ]
  }

  spec {
    selector {
      match_labels = {
        app = "echoserver"
      }
    }

    strategy {
      type = "RollingUpdate"
    }
```

Figura 60 – Trecho de código do *Deployment* do *echoserver*

Fonte: Criado pelo autor

```
resource "kubernetes_service" "echoserver-service" {
  metadata {
    name      = "echoserver-service"
    namespace = kubernetes_namespace.echoserver.metadata.0.name
    labels = {
      app = "echoserver"
    }
  }

  lifecycle {
    ignore_changes = [
      metadata[0].annotations,
      metadata[0].labels,
    ]
  }

  spec {
    type = "ClusterIP"

    selector = {
      app = "echoserver"
    }

    port {
      port        = "80"
      target_port = "8080"
      name        = "http"
      protocol    = "TCP"
    }
```

Figura 61 – Trecho de código do *Service* do *echoserver*

Fonte: Criado pelo autor

```
resource "kubernetes_ingress" "echoserver-ingress" {
  metadata {
    name      = "echoserver-ingress"
    namespace = kubernetes_namespace.echoserver.metadata.0.name
    labels = {
      app = "echoserver"
    }

    annotations = {
      "kubernetes.io/ingress.class" = "nginx"
    }
  }

  lifecycle {
    ignore_changes = [
      metadata[0].annotations,
      metadata[0].labels,
    ]
  }

  spec {
    rule {
      host = "echoserver.gitops.com.br"
      http {
        path {
          backend {
            service_name = kubernetes_service.echoserver-service.metadata.0.name
            service_port = 80
          }

          path = "/"
```

Figura 62 – Trecho de código do *Ingress* do *echoserver*

Fonte: Criado pelo autor

No arquivo de configuração *.gitlab-ci.yml* do *echoserver*, em *before_scripts*, há a adição da variável *dev_k8s_kubespray* que, conforme mencionada anteriormente, fornece a autenticação ao *cluster EKS*.

Na Figura 63 podemos observar o bloco *before_scripts* e o conteúdo da variável *dev_k8s_kubespray* sendo direcionada para o arquivo *~ /.kube/config*. Este arquivo é lido pelo binário *kubectl* que permite executar comandos *Kubernetes* e aplicar os arquivos de configuração *.tf* do *echoserver*.

```
20  before_script:
21    - cat ${CI_PROJECT_DIR}.tmp/${CI_COMMIT_REF_NAME}_k8s_kubeconfig > ~/.kube/config
22    - efs_dns=$(cat ${CI_PROJECT_DIR}.tmp/${CI_COMMIT_REF_NAME}_efs_dns)
23    - terraform --version
24    - cd ${TF_ROOT}
25    - terraform init -reconfigure -backend-config "bucket=$S3_BUCKET" -backend-config "region=$S3_REGION" -backend-confi
```

Figura 63 – *before_scripts* do projeto *echoserver*

Fonte: Criado pelo autor

Após os arquivos de configuração do *echoserver* estarem declarados, é feito o registro (*commit*) em seu respectivo projeto e executado o *pipeline* de criação dos recursos como podemos ver nas figuras 64 e 65.

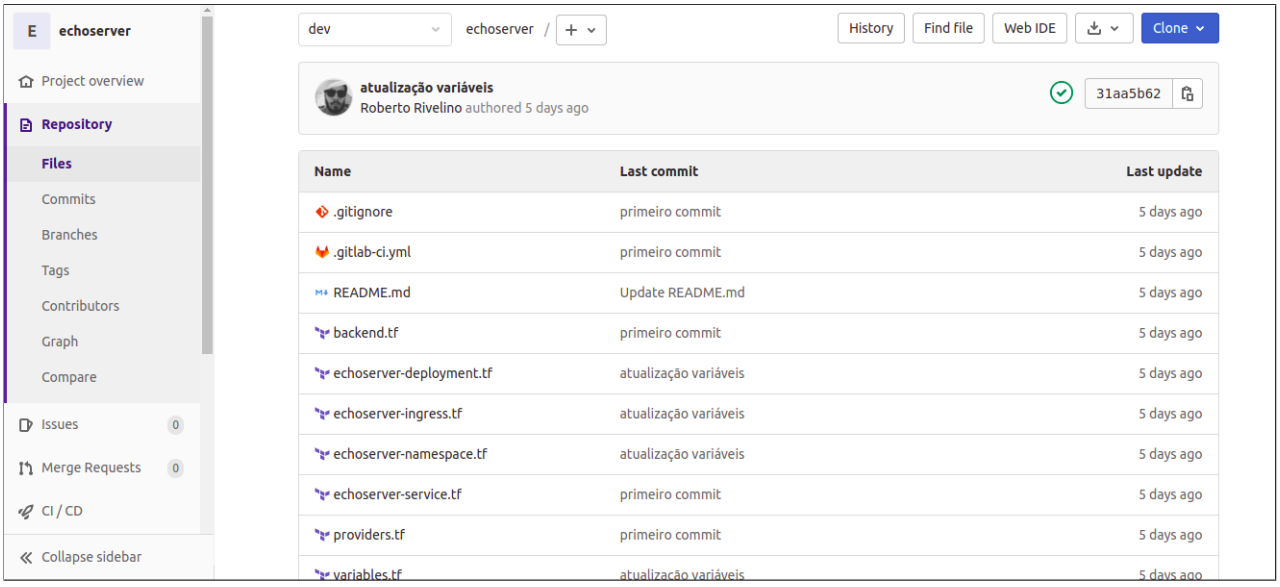

Figura 64 – Últimos registros no projeto *echoserver*.

Fonte: Criado pelo autor

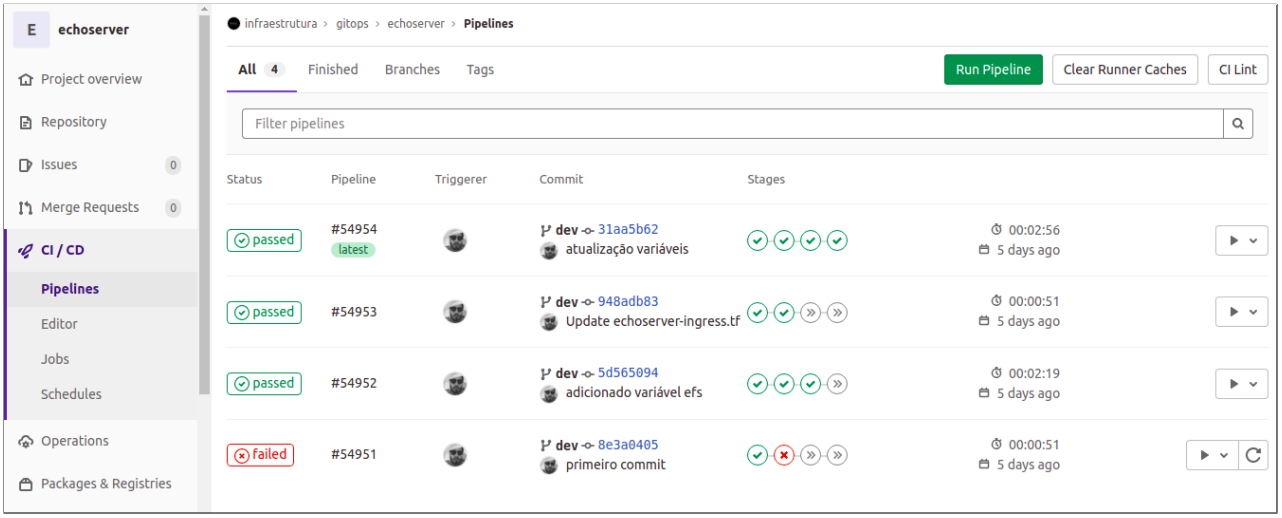

Figura 65 – Histórico de alterações do *echoserver* através dos *pipelines* no *VCS*

Fonte: Criado pelo autor

Depois do *pipeline* executado, é possível acessar a aplicação *echoserver* pelo navegador. Na Figura 66 temos a aplicação *echoserver* em funcionamento:

Figura 66 – Aplicação *echoserver* em funcionamento

Fonte: Criado pelo autor

5 Conclusão e Trabalhos Futuros

O intuito deste trabalho foi estimular a criação, configuração e gerenciamento de um ambiente de infraestrutura utilizando ferramentas de Infraestrutura como Código.

Implantar um ambiente de infraestrutura desta complexidade utilizando os preceitos do *Tradicional Ops* torna-se obsoleto quando são visíveis os benefícios ofertados pelo modelo proposto. Quando os processos manuais de criação da infraestrutura são adotados, o tempo gasto na elaboração dos recursos cresce expressivamente, diante da falta de procedimentos automatizados que encurtem o intervalo entre a concepção da infraestrutura e a disponibilidade para o cliente.

O tempo de suporte nas aplicações também aumenta em contrapartida ao modelo *GitOps*, pois as alterações nestas aplicações demandam processos manuais e não guardam um estado real, pecando em consistência e confiabilidade, ferindo o conceito de Infraestrutura Imutável. Portanto, através das metodologias de desenvolvimento da infraestrutura, práticas *DevOps*, metodologia *GitOps* e demais ferramentas adotadas no trabalho, foi possível atingir o objetivo geral de criar um ambiente de infraestrutura automatizado que garanta a entrega contínua das aplicações de maneira ágil. Os objetivos específicos, que derivavam sobretudo da criação deste ambiente, também foram alcançados.

Por meio da metodologia *GitOps*, foi possível manter um histórico único de alterações no Sistema de Controle de Versão - *VCS* utilizado e garantir que o estado da infraestrutura e das aplicações sempre estarão em conformidade com os registros declarados no *VCS* sem a necessidade de alterações manuais. Este processo foi realizado através da Integração e Entrega Contínua (*CI/CD*), que permitiram dispender menor tempo na resolução de conflitos durante o ciclo de desenvolvimento da infraestrutura e da aplicação, disponibilizando-as para o uso.

Contudo, durante o desenvolvimento deste trabalho, outras ferramentas *GitOps* foram aprimoradas, tornando o processo de sincronização de estado entre aplicação e arquivos declarados no *VCS* ainda mais ágil. Dentre elas, por exemplo, temos: *Argo CD, Flux* e *JenkinsX*. Assim, será possível explorá-las em uma futura extensão do trabalho.

Além disso, será possivel em trabalhos futuros abordar outras áreas da computação como *Machine Learning*, e outros processos como testes de mesa (*testbed*), a fim de mensurar a eficiência do modelo *GitOps* em contraposição aos outros tipos de infraestrutura.

6 Referências

AMBLER, Scott W.; LINES, Mark. **Disciplined Agile Delivery:** a practitioner's guide to agile software delivery in the enterprise. Upper Saddle River, NJ: IBM Press, 2012. 544 p.

ATLASSIAN (Austrália). **O que é controle de versão**. Disponível em: `https://www.atlassian.com/br/git/tutorials/what-is-version-control`. Acesso em: 17 out. 2020.

AWS. **What is a Network Load Balancer?**. Elaborada pela Amazon Web Services, Inc. Disponível em: `https://docs.aws.amazon.com/elasticloadbalancing/latest/network/introduction.html`. Acesso em: 02 abr. 2021.

AWS. **Security groups for your VPC**. Elaborada pela Amazon Web Services, Inc. Disponível em: `https://docs.aws.amazon.com/vpc/latest/userguide/VPC_SecurityGroups.html`. Acesso em: 02 abr. 2021.

BASS, Len; WEBER, Ingo; ZHU, Liming. **DevOps:** a software architect's perspective. Old Tappan Road, Tappan Road - Nj: Pearson Education, Inc., 2015. 352 p.

BELCHIOR, Rafael. **DevOps101 — First Steps on Terraform:** terraform + openstack + ansible. 2018. Elaborada por Bitcoin Insider. Disponível em: `https://www.bitcoininsider.org/article/51729/devops101-first-steps-terraform-terraform-openstac` Acesso em: 08 out. 2020.

BRIKMAN, Yevgeniy. **Terraform Up And Running:** writing infrastructure as code. Sebastopol, Ca: O'Reilly Media, Inc, 2016. (ISBN 978-1-491-97703-3).

BURNS, Brendan; HIGHTOWER, Kelsey; BEDA, Joe. **Kubernetes Up And Running:** dive into the future of infrastructure. Sebastopol, Ca: O'Reilly Media, Inc., 2017. 354 p. Acesso em 31 out. 2020.

BUYYA, Rajkumar; YEOA, Chee Shin; BROBERG, James; BRANDIC, Ivona; VENUGOPAL, Srikumar. **Cloud computing and emerging IT platforms:** Vision, hype, and reality for delivering computing as the 5th utility. Future Generation Com-

puter Systems. 03/12/2008. Elsevier. Disponível em: http://www.buyya.com/papers/Cloud-FGCS2009.pdf. Acesso em 31 out. 2020.

DAVIS, Jennifer; DANIELS, Katherine. **Effective DevOps:** building a culture of collaboration, affinity, and tooling at scale. Gravenstein Highway North, Sebastopol, Ca: O'Reilly Media, Inc., 2015. 378 p.

DOCKER. **Get Started: Quickstart:** Part 1 Orientation and setup. 2020. Disponível em: https://docs.docker.com/get-started/. Acesso em: 26 out. 2020.

EBERMANN, Alwin. **Evaluation of GitOps Security in a CI/CD Environment**. 2019. 59 f. TCC (Graduação) - Curso de Electrical And Computer Engineering, Department Of Electrical And Computer Engineering, Technical University Of Munich, Munich, Germany, 2019.

GITLAB INC. (org.). **Choosing between GitLab.com and self-managed subscriptions**. Elaborada por GitLab Inc.. Disponível em: https://about.gitlab.com/handbook/marketing/strategic-marketing/dot-com-vs-self-managed/. Acesso em: 02 nov. 2020.

GITLAB INC. (org.). **GitLab CI/CD.** Elaborada por GitLab Inc.. Disponível em: https://docs.gitlab.com/ee/ci/. Acesso em: 02 nov. 2020.

GONZALEZ, David. **Implementing Modern DevOps:** enabling it organizations to deliver faster and smarter. Birmingham, Uk: Packt, 2017.

GUCKENHEIMER, Sam. **What is Infrastructure as Code?** 2017. Elaborada por Microsoft. Disponível em: https://docs.microsoft.com/en-us/azure/devops/learn/what-is-infrastructure-as-code. Acesso em: 07 set. 2020.

HETHEY, Jonathan M.. **GitLab Repository Management**. Birmingham, Uk: Packt Publishing Ltd., 2013. (ISBN 978-1-78328-179-4).

J. Díaz, J. E. Pérez, M. A. Lopez-Peña, G. A. Mena and A. Yagüe, **"Self-Service Cybersecurity Monitoring as Enabler for DevSecOps,"** in IEEE Access, vol. 7, pp. 100283-100295, 2019, doi: 10.1109/ACCESS.2019.2930000.

KAMARUZZAMAN, Shahril Bin. **How to set up Gitlab for Continuous**

Integration and Deployment on CentOS. 2019. Elaborado por HowtoForge. Disponível em: https://www.howtoforge.com/how-to-set-up-gitlab-server-for-ci-cd-operatio Acesso em: 22 nov. 2020.

KIM, Gene et al. **The DevOps Handbook:** how to create world-class agility, reliability, and security in technology organizations. 25 Nw 23Rd Pl, Suite 6314 Portland, Or 97210: It Revolution Press, Llc, 2016. 105 p.

KUBERNETES (org.). **Kubernetes**. 2020. Disponível em: https://kubernetes.io/pt/docs Acesso em: 27 set. 2020.

KUBERNETES (org.). **Ingress**. 2021. Disponível em: https://kubernetes.io/docs/concepts/services-networking/ingress/. Acesso em: 01 abr. 2021.

MORRIS, Kief. **Infrastructure as Code:** managing servers in the cloud. Gravenstein Highway North, Sebastopol, Ca: O'Reilly Media, Inc., 2016. 447 p.

PEPGOTESTING (org.). **Automated software testing in Continuous Integration (CI) and Continuous Delivery (CD).** Disponível em: https://pepgotesting.com/continuous-integration/. Acesso em: 10 out. 2020.

PITTET, Sten. **How to get started with Continuous Integration.** 2020. Elaborada por Atlassian. Disponível em: https://www.atlassian.com/continuous-delivery/continuous-integration/how-to-get-to-continuous-integration. Acesso em: 10 out. 10.

QUEIROZ, Renan. **DevOps para Bancos SQL Server.** 2019. Elaborada por Medium. Disponível em: https://medium.com/@renanlq/devops-para-bancos-sql-server-2 Acesso em: 22 nov. 2020.

SANCHE, Daniel. **Kubernetes 101:** Pods, Nodes, Containers, and Clusters. 2018. Elaborado por Medium. Disponível em: https://medium.com/google-cloud/kubernetes-101-pods-nodes-containers-and-clusters-c1509e409e16. Acesso em: 31 out. 2020.

SOUZA, Igor. **Terraform - Uma pequena introdução**. 2017. Elaborada por Medium. Disponível em: https://medium.com/@igordcsouza/terraform-uma-pequena-intro C3%A7%C3%A3o-eae86f22db55. Acesso em: 02 nov. 2020.

TERRAFORM. **How Terraform Works.** 2020. Disponível em: https://www.hashicorp.com/products/terraform. Acesso em: 09 out. 2020.

TERRAFORM. **Command: validate.** 2021. Disponível em: https://www.terraform.io/docs/cli/commands/validate.html. Acesso em: 02 abr. 2021.

TERRAFORM. **Command: plan.** 2021. Disponível em: https://www.terraform.io/docs/cli/commands/plan.html. Acesso em: 03 abr. 2021.

VAN BAARSEN, Jeroen. **GitLab Cookbook:** over 60 hands-on recipes to efficiently self-host your own git repository using gitlab. Birmingham, Uk: Packt Publishing Ltd., 2014. (ISBN 978-1-78398-684-2).

VEHENT, Julien. **Securing DevOps:** security in the cloud. Shelter Island, Ny 11964: Manning Publications Co., 2018. 401 p. (ISBN 9781617294136).

VIRDÓ, Hazel. **What Is Immutable Infrastructure?** 2017. Elaborada por DigitalOcean. Disponível em: https://www.digitalocean.com/community/tutorials/what-is-immutable-infrastructure#differences-between-mutable-and-immutable-infrastr
Acesso em: 12 out. 2020.

WEAVEWORKS. **Guide to GitOps.** 2020. Disponível em: https://www.weave.works/technologies/gitops/. Acesso em: 29 set. 2020.

Apêndices

A Arquivos do Projeto aws

A.1 *.gitignore*

```
# Local .terraform directories
**/.terraform/*

# .tfstate files
*.tfstate
*.tfstate.*
*.terraform.*

# Crash log files
crash.log

# Ignore override files as they are usually used to override resources locally and so
# are not checked in
override.tf
override.tf.json
*_override.tf
*_override.tf.json

# Include override files you do wish to add to version control using negated pattern
#
# !example_override.tf

# Include tfplan files to ignore the plan output of command: terraform plan -out=tfplan
# example: *tfplan*

# Ignore CLI configuration files
.terraformrc

# Terraform plan
tf.plan

#Terraform version dev log
dev.log
```

Código Fonte A.1 – aws/.gitignore

A.2 *.gitlab-ci.yml*

```
1
2  image:
3    name: registry.bry.com.br/infra/bry-iac:1.3
4    entrypoint:
5      - '/usr/bin/env'
6      - 'PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
         '
7
8  # Default output file for Terraform plan
9  variables:
10   PLAN: plan.tfplan
11   TF_IN_AUTOMATION: "true"
12   TF_ROOT: ${CI_PROJECT_DIR}
13
14 cache:
15   key: "${TF_ROOT}"
16   paths:
17     - ${TF_ROOT}/.terraform/
18     - ${TF_ROOT}/.terraform.lock.hcl
19
20 before_script:
21   - terraform --version
22   - terraform init -reconfigure -backend-config "bucket=$S3_BUCKET" -
       backend-config "region=$S3_REGION" -backend-config "key=gitops/
       $CI_COMMIT_REF_NAME/tfstate"
23 stages:
24   - validate
25   - plan
26   - apply
27   - destroy
28
29 validate:
30   stage: validate
31   script:
32     - cd ${TF_ROOT}
33     - terraform validate
34     - terraform fmt -check=true
35   only:
36     - branches
37   tags:
38     - dc43
39
40 merge review:
41   before_script:
42     - terraform --version
43     - terraform init -reconfigure -backend-config "bucket=$S3_BUCKET" -
```

```
      backend-config "region=$S3_REGION" -backend-config "key=gitops/
      $CI_COMMIT_REF_NAME/tfstate"
  stage: plan
  script:
    - terraform init -reconfigure -backend-config "bucket=$S3_BUCKET" -
      backend-config "region=$S3_REGION" -backend-config "key=gitops/
      $CI_COMMIT_REF_NAME/tfstate"
    - terraform plan -var="environment=
      $CI_MERGE_REQUEST_TARGET_BRANCH_NAME" -out=$PLAN
    - echo \`\`\`diff > plan.txt
    - terraform show -no-color ${PLAN} | tee -a plan.txt
    - echo \`\`\` >> plan.txt
    - sed -i -e 's/  +/+/g' plan.txt
    - sed -i -e 's/  ~/~/g' plan.txt
    - sed -i -e 's/  -/-/g' plan.txt
    - MESSAGE=$(cat plan.txt)
    - >-
      curl -X POST -g -H "PRIVATE-TOKEN: ${GITLAB_TOKEN}"
      --data-urlencode "body=${MESSAGE}"
      "${CI_API_V4_URL}/projects/${CI_PROJECT_ID}/merge_requests/${
        CI_MERGE_REQUEST_IID}/discussions"
  dependencies:
    - validate
  artifacts:
    name: plan
    paths:
      - $PLAN
  only:
    - merge_requests

plan production:
  stage: plan
  script:
    - cd ${TF_ROOT}
    - terraform plan -var="environment=$CI_COMMIT_REF_NAME" -out=$PLAN
  artifacts:
    name: plan
    paths:
      - $PLAN
  only:
    - prd
    - hom
    - dev
  tags:
    - dc43

apply:
```

```
85    stage: apply
86    script:
87      - cd ${TF_ROOT}
88      - terraform apply -auto-approve -input=false $PLAN
89    dependencies:
90      - plan production
91    artifacts:
92      name: $CI_COMMIT_REF_SLUG
93      untracked: true
94    only:
95      - prd
96      - hom
97      - dev
98    tags:
99      - dc43
100   when: manual
101
102 destroy:
103   stage: destroy
104   script:
105     - cd ${TF_ROOT}
106     - terraform destroy -auto-approve -var="environment=
          $CI_COMMIT_REF_NAME"
107   dependencies:
108     - apply
109   when: manual
110   only:
111     - prd
112     - hom
113     - dev
114   tags:
115     - dc43
```

Código Fonte A.2 – aws/.gitlab-ci.yml

A.3 *backend.tf*

```
1 terraform {
2   backend "s3" {}
3 }
```

Código Fonte A.3 – aws/backend.tf

A.4 *efs.tf*

```
1 module "efs" {
```

```
  source = "git::https://github.com/cloudposse/terraform-aws-efs.git?ref
     =tags/0.30.1"
  name              = format("gitops-efs-%s", var.environment)
  region            = "us-east-2"
  vpc_id            = module.vpc.vpc_id
  subnets           = module.vpc.public_subnets
  security_groups   = [module.efs_sg.this_security_group_id]
  encrypted         = true
  transition_to_ia = "AFTER_60_DAYS"
  tags = {
    terraform   = "true"
    environment = var.environment
  }
}
resource "aws_security_group_rule" "efs_sg" {
  type              = "ingress"
  from_port         = 2049
  to_port           = 2049
  protocol          = "tcp"
  cidr_blocks       = ["10.0.0.0/16"]
  security_group_id = module.efs.security_group_id
}
```

Código Fonte A.4 – aws/efs.tf

A.5 *eks.tf*

```
module "eks" {
  source            = "terraform-aws-modules/eks/aws"
  version           = "~> 14.0.0"
  cluster_name      = format("gitops-eks-%s", var.environment)
  cluster_version   = "1.19"
  subnets           = module.vpc.private_subnets
  write_kubeconfig = "false"
  vpc_id            = module.vpc.vpc_id
  enable_irsa       = true
  node_groups_defaults = {
    subnets    = module.vpc.private_subnets
    ami_type   = "AL2_x86_64"
    disk_size = 45
  }
  node_groups = {
    gitops_t2large = {
      name             = format("gitops-eks-node-t2large-%s", var.
         environment)
      desired_capacity = 2
      max_capacity     = 5
```

```
      min_capacity     = 2
      key_name         = "gitops-ohio"
      instance_type    = "t2.large"
      k8s_labels = {
        Terraform           = "true"
        propagate_at_launch = true
      }
    }
  }
  tags = {
    Terraform   = "true"
    environment = var.environment
  }
}
```

Código Fonte A.5 – aws/eks.tf

A.6 *gitlab_config.tf*

```
// Configura cluster para ser gerenciado pelo Gitlab
resource "gitlab_project_cluster" "aws_cluster" {
  project                       = 1313
  name                          = "aws-cluster"
  enabled                       = true
  kubernetes_api_url            = module.eks.cluster_endpoint
  kubernetes_token              = data.kubernetes_secret.gitlab-admin-
      token.data.token
  kubernetes_ca_cert            = trimspace(base64decode(module.eks.
      cluster_certificate_authority_data))
  kubernetes_namespace          = "kube-system"
  kubernetes_authorization_type = "rbac"
  environment_scope             = "eks/*"
  count = var.environment == "prd" ? 1 : 0
}
resource "gitlab_group_variable" "dev_k8s_kubeconfig" {
  group         = 378
  key           = "dev_k8s_kubeconfig"
  value         = module.eks.kubeconfig
  variable_type = "file"
  protected     = false
  count = var.environment == "dev" ? 1 : 0
}
resource "gitlab_group_variable" "hom_k8s_kubeconfig" {
  group         = 378
  key           = "hom_k8s_kubeconfig"
  value         = module.eks.kubeconfig
  variable_type = "file"
```

```
  protected     = false
  count = var.environment == "hom" ? 1 : 0
}
resource "gitlab_group_variable" "prd_k8s_kubeconfig" {
  group         = 378
  key           = "prd_k8s_kubeconfig"
  value         = module.eks.kubeconfig
  variable_type = "file"
  protected     = false
  count = var.environment == "prd" ? 1 : 0
}
resource "gitlab_group_variable" "dev_efs_dns" {
  group         = 378
  key           = "dev_efs_dns"
  value         = module.efs.dns_name
  variable_type = "file"
  protected     = false
  count = var.environment == "dev" ? 1 : 0
}
resource "gitlab_group_variable" "hom_efs_dns" {
  group         = 378
  key           = "hom_efs_dns"
  value         = module.efs.dns_name
  variable_type = "file"
  protected     = false
  count = var.environment == "hom" ? 1 : 0
}
resource "gitlab_group_variable" "prd_efs_dns" {
  group         = 378
  key           = "prd_efs_dns"
  value         = module.efs.dns_name
  variable_type = "file"
  protected     = false
  count = var.environment == "prd" ? 1 : 0
}
resource "gitlab_group_variable" "prd_rds_endpoint" {
  group         = 378
  key           = "prd_rds_endpoint"
  value         = module.rds.this_db_instance_endpoint
  variable_type = "file"
  protected     = false
  count = var.environment == "dev" ? 1 : 0
}
resource "gitlab_group_variable" "dev_rds_endpoint" {
  group         = 378
  key           = "dev_rds_endpoint"
  value         = module.rds.this_db_instance_endpoint
```

```
74    variable_type = "file"
75    protected     = false
76    count = var.environment == "dev" ? 1 : 0
77  }
78  resource "gitlab_group_variable" "hom_rds_endpoint" {
79    group         = 378
80    key           = "hom_efs_dns"
81    value         = module.rds.this_db_instance_endpoint
82    variable_type = "file"
83    protected     = false
84    count = var.environment == "hom" ? 1 : 0
85  }
```

Código Fonte A.6 – aws/gitlab_config.tf

A.7 *k8s_gitlab_auth.tf*

```
1   data "aws_eks_cluster" "my-cluster" {
2     name = module.eks.cluster_id
3   }
4   data "aws_eks_cluster_auth" "my-auth" {
5     name = module.eks.cluster_id
6   }
7   resource "kubernetes_service_account" "gitlab-admin" {
8     metadata {
9       name      = "gitlab-admin"
10      namespace = "kube-system"
11    }
12  }
13  resource "kubernetes_secret" "gitlab-admin" {
14    metadata {
15      name      = "gitlab-admin"
16      namespace = "kube-system"
17      annotations = {
18        "kubernetes.io/service-account.name" = kubernetes_service_account.
              gitlab-admin.metadata.0.name
19      }
20    }
21    lifecycle {
22      ignore_changes = [
23        data
24      ]
25    }
26    type = "kubernetes.io/service-account-token"
27  }
28  data "kubernetes_secret" "gitlab-admin-token" {
29    metadata {
```

```
30      name      = kubernetes_service_account.gitlab-admin.
            default_secret_name
31      namespace = "kube-system"
32    }
33  }
34  resource "kubernetes_cluster_role_binding" "gitlab-admin" {
35    metadata {
36      name = "gitlab-admin"
37    }
38    role_ref {
39      api_group = "rbac.authorization.k8s.io"
40      kind      = "ClusterRole"
41      name      = "cluster-admin"
42    }
43    subject {
44      kind      = "ServiceAccount"
45      name      = "gitlab-admin"
46      namespace = "kube-system"
47    }
48  }
```

Código Fonte A.7 – aws/k8s_gitlab_auth.tf

A.8 *nlb.tf*

```
1  module "nlb" {
2    source = "./nlb"
3    module_depends_on = [module.eks]
4  }
```

Código Fonte A.8 – aws/nlb.tf

A.9 *providers.tf*

```
1  provider "aws" {
2    region = "us-east-2"
3  }
4  provider "gitlab" {
5    base_url = "https://git.com.br/api/v4/"
6    token    = "************"
7  }
8  provider "kubernetes" {
9    host                   = data.aws_eks_cluster.my-cluster.endpoint
10   cluster_ca_certificate = base64decode(data.aws_eks_cluster.my-cluster.
         certificate_authority.0.data)
11   token                  = data.aws_eks_cluster_auth.my-auth.token
```

```
  load_config_file       = false
}
```

Código Fonte A.9 – aws/providers.tf

A.10 *rds.tf*

```
data "aws_vpc" "selected" {
  id = module.vpc.vpc_id
}
data "aws_security_group" "default" {
  vpc_id = data.aws_vpc.selected.id
  name   = module.rds_sg.this_security_group_name
}
module "rds" {
  source  = "terraform-aws-modules/rds/aws"
  version = "~> 2.0"
  identifier = format("gitops-rds-%s", var.environment)
  engine                = "postgres"
  engine_version        = "12.5"
  instance_class        = "db.t2.micro"
  allocated_storage     = 20
  max_allocated_storage = 50
  storage_encrypted     = false
  multi_az              = true
  username = "gitops"
  password = "Ly5ZesYuE8qKfBy2"
  port     = "5432"
  vpc_security_group_ids = [data.aws_security_group.default.id]
  maintenance_window = "Sun:03:30-Sun:04:30"
  backup_window      = "01:00-02:00"
  backup_retention_period = 7
  tags = {
    terraform   = "true"
    Environment = var.environment
  }
  enabled_cloudwatch_logs_exports = ["postgresql", "upgrade"]
  subnet_ids = module.vpc.public_subnets
  family = "postgres12"
  major_engine_version = "12"
  deletion_protection = false
  publicly_accessible = true
}
```

Código Fonte A.10 – aws/rds.tf

A.11 *security_group_efs.tf*

```
module "efs_sg" {
  source = "terraform-aws-modules/security-group/aws"
  name        = format("gitops-efs-sg-%s", var.environment)
  description = "Permite acesso a VPC"
  vpc_id      = module.vpc.vpc_id
  ingress_with_cidr_blocks = [
    {
      from_port   = 2049
      to_port     = 2049
      protocol    = "tcp"
      description = "Toda VPC do gitops"
      cidr_blocks = "10.0.0.0/16"
    },
  ]
  tags = {
    Terraform   = "true"
    Environment = var.environment
  }
}
```

Código Fonte A.11 – aws/security_group_efs.tf

A.12 *security_group_rds.tf*

```
module "rds_sg" {
  source = "terraform-aws-modules/security-group/aws"
  name        = format("gitops-rds-sg-%s", var.environment)
  description = "Permite acesso a VPC"
  vpc_id      = module.vpc.vpc_id
  ingress_with_cidr_blocks = [
    {
      from_port   = 5432
      to_port     = 5432
      protocol    = "tcp"
      description = "Permitir acesso a VPC"
      cidr_blocks = "10.0.0.0/16"
    },
  ]
  tags = {
    Terraform   = "true"
    Environment = var.environment
  }
}
```

Código Fonte A.12 – aws/security_group_rds.tf

A.13 *variables.tf*

```
variable "environment" {
  type = string
}
```

Código Fonte A.13 – aws/variables.tf

A.14 *versions.tf*

```
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 3.30.0"
    }
    gitlab = {
      source  = "gitlabhq/gitlab"
      version = "~> 3.3.0"
    }
    kubernetes = {
      source  = "hashicorp/kubernetes"
      version = "~> 1.13.3"
    }
    null = {
      source = "hashicorp/null"
    }
  }
  required_version = ">= 0.13"
}
```

Código Fonte A.14 – aws/versions.tf

A.15 *vpc.tf*

```
resource "aws_eip" "nat" {
  count = 1
  vpc = true
}
module "vpc" {
  source  = "terraform-aws-modules/vpc/aws"
  version = "2.64.0"
  name                 = format("gitops-vpc-%s", var.environment)
  cidr                 = "10.0.0.0/16"
  azs                  = ["us-east-2a", "us-east-2b", "us-east-2c"]
  private_subnets      = ["10.0.1.0/24", "10.0.2.0/24", "10.0.3.0/24"]
```

```
12   public_subnets       = ["10.0.101.0/24", "10.0.102.0/24", "
        10.0.103.0/24"]
13   enable_nat_gateway   = true
14   single_nat_gateway   = true
15   reuse_nat_ips        = true
16   external_nat_ip_ids = aws_eip.nat.*.id
17   enable_vpn_gateway                = true
18   propagate_public_route_tables_vgw = true
19   enable_dns_hostnames = true
20   enable_dns_support   = true
21   tags = {
22     Terraform                                                    = "
        true"
23     Environment                                                  = var
        .environment
24     format("kubernetes.io/cluster/gitops-eks-%s", var.environment) = "
        shared"
25   }
26   public_subnet_tags = {
27     Terraform                                                    = "
        true"
28     Environment                                                  = var
        .environment
29     "kubernetes.io/role/elb"                                     = "1"
30     format("kubernetes.io/cluster/gitops-eks-%s", var.environment) = "
        shared"
31   }
32   private_subnet_tags = {
33     "kubernetes.io/role/internal-elb" = "1"
34   }
35   public_route_table_tags = {
36     "kubernetes.io/role/elb" = "1"
37   }
38   private_route_table_tags = {
39     "kubernetes.io/role/internal-elb" = "1"
40   }
41 }
```

Código Fonte A.15 – aws/vpc.tf

A.16 *module_depends_on.tf*

```
1 variable "module_depends_on" {
2   default = [""]
3 }
4 resource "null_resource" "module_depends_on" {
5   triggers = {
```

```
    value = length(var.module_depends_on)
  }
}
```

Código Fonte A.16 – aws/nlb/module_depends_on.tf

A.17 *namespace.tf*

```
resource kubernetes_namespace "nginx" {
  metadata {
    name = "ingress-nginx"
    labels = {
      "app.kubernetes.io/name"    = "ingress-nginx"
      "app.kubernetes.io/part-of" = "ingress-nginx"
    }
  }
  lifecycle {
    ignore_changes = [
      metadata[0].annotations,
      metadata[0].labels,
    ]
  }
  timeouts {
    delete = "10m"
  }
  depends_on = [null_resource.module_depends_on]
}
```

Código Fonte A.17 – aws/nlb/namespace.tf

A.18 *nginx-configuration-configmap.tf*

```
resource "kubernetes_config_map" "nginx" {
  metadata {
    name      = "nginx-configuration"
    namespace = kubernetes_namespace.nginx.metadata.0.name
    labels = {
      "app.kubernetes.io/name"    = "ingress-nginx"
      "app.kubernetes.io/part-of" = "ingress-nginx"
    }
  }
  lifecycle {
    ignore_changes = [
      metadata[0].annotations,
      metadata[0].labels,
    ]
```

```
  }
  data = {
    enable-underscores-in-headers = "True"
    server-tokens                 = "False"
    proxy-body-size               = "10m"
    ignore-invalid-headers        = "True"
    ssl-ciphers                   = "ECDHE-RSA-AES256-GCM-SHA384:ECDHE-
       RSA-CHACHA20-POLY1305:ECDHE-RSA-AES128-GCM-SHA256:ECDHE-RSA-
       AES256-SHA384:ECDHE-RSA-AES128-SHA256"
  }
}
```

Código Fonte A.18 – aws/nlb/nginx-configuration-configmap.tf

A.19 *nginx-ingress-clusterrole-nisa-binding.tf*

```
resource "kubernetes_cluster_role_binding" "nginx" {
  metadata {
    name = "nginx-ingress-clusterrole-nisa-binding"
    labels = {
      "app.kubernetes.io/name"    = "ingress-nginx"
      "app.kubernetes.io/part-of" = "ingress-nginx"
    }
  }
  lifecycle {
    ignore_changes = [
      metadata[0].annotations,
      metadata[0].labels,
    ]
  }
  role_ref {
    api_group = "rbac.authorization.k8s.io"
    kind      = "ClusterRole"
    name      = "nginx-ingress-clusterrole"
  }
  subject {
    kind      = "ServiceAccount"
    name      = kubernetes_service_account.nginx.metadata.0.name
    namespace = kubernetes_namespace.nginx.metadata.0.name
  }
}
```

Código Fonte A.19 – aws/nlb/nginx-ingress-clusterrole-nisa-binding.tf

A.20 *nginx-ingress-clusterrole-rbac.tf*

```
resource "kubernetes_cluster_role" "nginx" {
  metadata {
    name = "nginx-ingress-clusterrole"
    labels = {
      "app.kubernetes.io/name"    = "ingress-nginx"
      "app.kubernetes.io/part-of" = "ingress-nginx"
    }
  }
  lifecycle {
    ignore_changes = [
      metadata[0].annotations,
      metadata[0].labels,
    ]
  }
  rule {
    api_groups = [""]
    resources  = ["configmaps", "endpoints", "nodes", "pods", "secrets"]
    verbs      = ["list", "watch"]
  }
  rule {
    api_groups = [""]
    resources  = ["nodes"]
    verbs      = ["get"]
  }
  rule {
    api_groups = [""]
    resources  = ["services"]
    verbs      = ["get", "list", "watch"]
  }
  rule {
    api_groups = [""]
    resources  = ["events"]
    verbs      = ["create", "patch"]
  }
  rule {
    api_groups = ["extensions", "networking.k8s.io"]
    resources  = ["ingresses"]
    verbs      = ["get", "list", "watch"]
  }
  rule {
    api_groups = ["extensions", "networking.k8s.io"]
    resources  = ["ingresses/status"]
    verbs      = ["update"]
  }
}
```

Código Fonte A.20 – aws/nlb/nginx-ingress-clusterrole-rbac.tf

A.21 *nginx-ingress-controller-daemonset.tf*

```
resource "kubernetes_daemonset" "nginx" {
  metadata {
    name      = "nginx-ingress-controller"
    namespace = kubernetes_namespace.nginx.metadata.0.name
    labels = {
      "app.kubernetes.io/name"    = "ingress-nginx"
      "app.kubernetes.io/part-of" = "ingress-nginx"
    }
  }
  lifecycle {
    ignore_changes = [
      metadata[0].annotations,
      metadata[0].labels,
    ]
  }
  spec {
    selector {
      match_labels = {
        "app.kubernetes.io/name"    = "ingress-nginx"
        "app.kubernetes.io/part-of" = "ingress-nginx"
      }
    }
    template {
      metadata {
        labels = {
          "app.kubernetes.io/name"    = "ingress-nginx"
          "app.kubernetes.io/part-of" = "ingress-nginx"
        }
        annotations = {
          "prometheus.io/port"   = "10254"
          "prometheus.io/scrape" = "true"
        }
      }
      spec {
        termination_grace_period_seconds = 0
        service_account_name             = kubernetes_service_account.
           nginx.metadata.0.name
        automount_service_account_token  = "true"
        host_network                     = "true"
        container {
          image = "quay.io/kubernetes-ingress-controller/nginx-ingress-
             controller:0.32.0"
          name  = "nginx-ingress-controller"
          args = [
            "/nginx-ingress-controller",
            "--configmap=$(POD_NAMESPACE)/nginx-configuration",
```

```
45            "--tcp-services-configmap=$(POD_NAMESPACE)/tcp-services",
46            "--udp-services-configmap=$(POD_NAMESPACE)/udp-services",
47            "--publish-service=$(POD_NAMESPACE)/ingress-nginx",
48            "--annotations-prefix=nginx.ingress.kubernetes.io",
49          ]
50          security_context {
51            allow_privilege_escalation = "true"
52            capabilities {
53              drop = ["ALL"]
54              add  = ["NET_BIND_SERVICE"]
55            }
56            run_as_user = 101
57          }
58          env {
59            name = "POD_NAME"
60            value_from {
61              field_ref {
62                field_path = "metadata.name"
63              }
64            }
65          }
66          env {
67            name = "POD_NAMESPACE"
68            value_from {
69              field_ref {
70                field_path = "metadata.namespace"
71              }
72            }
73          }
74          port {
75            name           = "http"
76            container_port = 80
77            host_port      = 80
78          }
79          port {
80            name           = "https"
81            container_port = 443
82            host_port      = 443
83          }
84          port {
85            name           = "metrics"
86            container_port = 10254
87            host_port      = 10254
88          }
89        }
90      }
91    }
```

```
92    }
93  }
```

Código Fonte A.21 – aws/nlb/nginx-ingress-controller-daemonset.tf

A.22 *nginx-ingress-metrics-service.tf*

```
1  resource "kubernetes_service" "nginx-ingress-metrics-service" {
2    metadata {
3      name      = "nginx-ingress-metrics-service"
4      namespace = kubernetes_namespace.nginx.metadata.0.name
5      labels = {
6        "app.kubernetes.io/name"    = "ingress-nginx"
7        "app.kubernetes.io/part-of" = "ingress-nginx"
8      }
9    }
10   lifecycle {
11     ignore_changes = [
12       metadata[0].annotations,
13       metadata[0].labels,
14     ]
15   }
16   spec {
17     type = "ClusterIP"
18     selector = {
19       "app.kubernetes.io/name"    = "ingress-nginx"
20       "app.kubernetes.io/part-of" = "ingress-nginx"
21     }
22     port {
23       name        = "metrics"
24       port        = 10254
25       target_port = 10254
26     }
27   }
28 }
```

Código Fonte A.22 – aws/nlb/nginx-ingress-metrics-service.tf

A.23 *nginx-ingress-role-nisa-binding.tf*

```
1  resource "kubernetes_role_binding" "nginx" {
2    metadata {
3      name      = "nginx-ingress-role-nisa-binding"
4      namespace = "ingress-nginx"
5      labels = {
6        "app.kubernetes.io/name"    = "ingress-nginx"
```

```
      "app.kubernetes.io/part-of" = "ingress-nginx"
    }
  }
  lifecycle {
    ignore_changes = [
      metadata[0].annotations,
      metadata[0].labels,
    ]
  }
  role_ref {
    api_group = "rbac.authorization.k8s.io"
    kind      = "Role"
    name      = "nginx-ingress-role"
  }
  subject {
    kind      = "ServiceAccount"
    name      = kubernetes_service_account.nginx.metadata.0.name
    namespace = kubernetes_namespace.nginx.metadata.0.name
  }
}
```

Código Fonte A.23 – aws/nlb/nginx-ingress-role-nisa-binding.tf

A.24 *nginx-ingress-role.tf*

```
resource "kubernetes_role" "nginx" {
  metadata {
    name      = "nginx-ingress-role"
    namespace = kubernetes_namespace.nginx.metadata.0.name
    labels = {
      "app.kubernetes.io/name"    = "ingress-nginx"
      "app.kubernetes.io/part-of" = "ingress-nginx"
    }
  }
  lifecycle {
    ignore_changes = [
      metadata[0].annotations,
      metadata[0].labels,
    ]
  }
  rule {
    api_groups = [""]
    resources  = ["configmaps", "pods", "secrets", "namespaces"]
    verbs      = ["get"]
  }
  rule {
    api_groups     = [""]
```

```
    resources      = ["configmaps"]
    resource_names = ["ingress-controller-leader-nginx"]
    verbs          = ["get", "update"]
  }
  rule {
    api_groups = [""]
    resources  = ["configmaps"]
    verbs      = ["create"]
  }
  rule {
    api_groups = [""]
    resources  = ["endpoints"]
    verbs      = ["get"]
  }
}
```

Código Fonte A.24 – aws/nlb/nginx-ingress-role.tf

A.25 *nginx-ingress-serviceaccount.tf*

```
resource "kubernetes_service_account" "nginx" {
  metadata {
    name      = "nginx-ingress-serviceaccount"
    namespace = kubernetes_namespace.nginx.metadata.0.name
    labels = {
      "app.kubernetes.io/name"    = "ingress-nginx"
      "app.kubernetes.io/part-of" = "ingress-nginx"
    }
  }
  lifecycle {
    ignore_changes = [
      metadata[0].annotations,
      metadata[0].labels,
    ]
  }
  automount_service_account_token = "true"
}
```

Código Fonte A.25 – aws/nlb/nginx-ingress-serviceaccount.tf

A.26 *nlb-service.tf*

```
resource "kubernetes_service" "nlb" {
  metadata {
    name      = "ingress-nginx"
    namespace = kubernetes_namespace.nginx.metadata.0.name
```

```
    labels = {
      "app.kubernetes.io/name"     = "ingress-nginx"
      "app.kubernetes.io/part-of" = "ingress-nginx"
    }
    annotations = {
      "service.beta.kubernetes.io/aws-load-balancer-type" = "nlb"
    }
  }
  lifecycle {
    ignore_changes = [
      metadata[0].annotations,
      metadata[0].labels,
    ]
  }
  spec {
    external_traffic_policy = "Local"
    type                    = "LoadBalancer"
    selector = {
      "app.kubernetes.io/name"     = "ingress-nginx"
      "app.kubernetes.io/part-of" = "ingress-nginx"
    }
    port {
      name        = "http"
      port        = 80
      target_port = "http"
    }
    port {
      name        = "https"
      port        = 443
      target_port = "https"
    }
  }
}
```

Código Fonte A.26 – aws/nlb/nlb-service.tf

A.27 *tcp-services-configmap.tf*

```
resource "kubernetes_config_map" "tcp" {
  metadata {
    name      = "tcp-services"
    namespace = kubernetes_namespace.nginx.metadata.0.name
    labels = {
      "app.kubernetes.io/name"     = "ingress-nginx"
      "app.kubernetes.io/part-of" = "ingress-nginx"
    }
  }
```

```
  lifecycle {
    ignore_changes = [
      metadata[0].annotations,
      metadata[0].labels,
    ]
  }
}
```

Código Fonte A.27 – aws/nlb/tcp-services-configmap.tf

A.28 *udp-services-configmap.tf*

```
resource "kubernetes_config_map" "udp" {
  metadata {
    name      = "udp-services"
    namespace = kubernetes_namespace.nginx.metadata.0.name
    labels = {
      "app.kubernetes.io/name"    = "ingress-nginx"
      "app.kubernetes.io/part-of" = "ingress-nginx"
    }
  }
  lifecycle {
    ignore_changes = [
      metadata[0].annotations,
      metadata[0].labels,
    ]
  }
}
```

Código Fonte A.28 – aws/nlb/udp-services-configmap.tf

B Arquivos do Projeto echoserver

B.1 *.gitignore*

```
# Local .terraform directories
**/.terraform/*

# .tfstate files
*.tfstate
*.tfstate.*
*.terraform.*

# Crash log files
crash.log

# Ignore override files as they are usually used to override resources locally and so
# are not checked in
override.tf
override.tf.json
*_override.tf
*_override.tf.json

# Include override files you do wish to add to version control using negated pattern
#
# !example_override.tf

# Include tfplan files to ignore the plan output of command: terraform plan -out=tfplan
# example: *tfplan*

# Ignore CLI configuration files
.terraformrc

# Terraform plan
tf.plan

#Terraform version dev log
dev.log
```

Código Fonte B.1 – echoserver/.gitignore

B.2 *.gitlab-ci.yml*

```
image:
  name: registry.bry.com.br/infra/bry-iac:1.3
  entrypoint:
    - '/usr/bin/env'
    - 'PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
      '

# Default output file for Terraform plan
variables:
  PLAN: plan.tfplan
  TF_IN_AUTOMATION: "true"
  TF_ROOT: ${CI_PROJECT_DIR}

cache:
  key: "${TF_ROOT}"
  paths:
    - ${TF_ROOT}/.terraform/
    - ${TF_ROOT}/.terraform.lock.hcl

before_script:
  - cat ${CI_PROJECT_DIR}.tmp/${CI_COMMIT_REF_NAME}_k8s_kubeconfig > ~/.
    kube/config
  - efs_dns=$(cat ${CI_PROJECT_DIR}.tmp/${CI_COMMIT_REF_NAME}_efs_dns)
  - terraform --version
  - cd ${TF_ROOT}
  - terraform init -reconfigure -backend-config "bucket=$S3_BUCKET" -
    backend-config "region=$S3_REGION" -backend-config "key=gitops/
    $CI_COMMIT_REF_NAME/$CI_PROJECT_NAME/tfstate"

stages:
  - validate
  - plan
  - apply
  - destroy

validate:
  stage: validate
  script:
    - cd ${TF_ROOT}
    - terraform validate
    - terraform fmt -check=true
  only:
    - branches
  tags:
    - docker

```

```
43 merge review:
44   before_script:
45   - cat ${CI_PROJECT_DIR}.tmp/${CI_MERGE_REQUEST_TARGET_BRANCH_NAME}
       _k8s_kubeconfig > ~/.kube/config
46   - efs_dns=$(cat ${CI_PROJECT_DIR}.tmp/${
       CI_MERGE_REQUEST_TARGET_BRANCH_NAME}_efs_dns)
47   - terraform --version
48   - terraform init -reconfigure -backend-config "bucket=$S3_BUCKET" -
       backend-config "region=$S3_REGION" -backend-config "key=gitops/
       $CI_MERGE_REQUEST_TARGET_BRANCH_NAME/$CI_PROJECT_NAME/tfstate"
49   stage: plan
50   script:
51     - terraform plan -var="efs=$efs_dns" -var="environment=
         $CI_MERGE_REQUEST_TARGET_BRANCH_NAME" -out=$PLAN
52     - echo \`\`\`diff > plan.txt
53     - terraform show -no-color ${PLAN} | tee -a plan.txt
54     - echo \`\`\` >> plan.txt
55     - sed -i -e 's/  +/+/g' plan.txt
56     - sed -i -e 's/  ~/~/g' plan.txt
57     - sed -i -e 's/  -/-/g' plan.txt
58     - >-
59       curl -X POST -g -H "PRIVATE-TOKEN: ${GITLAB_TOKEN}"
60       --data-urlencode "body=@plan.txt"
61       "${CI_API_V4_URL}/projects/${CI_PROJECT_ID}/merge_requests/${
           CI_MERGE_REQUEST_IID}/discussions"
62   dependencies:
63     - validate
64   artifacts:
65     name: plan
66     paths:
67       - $PLAN
68   only:
69     - merge_requests
70   tags:
71     - docker
72
73 plan production:
74   stage: plan
75   script:
76     - cd ${TF_ROOT}
77     - terraform plan -var="efs=$efs_dns" -var="environment=
         $CI_COMMIT_REF_NAME" -out=$PLAN
78   artifacts:
79     name: plan
80     paths:
81       - $PLAN
82   only:
```

```
83      - prd
84      - hom
85      - dev
86    tags:
87      - docker
88
89  apply:
90    stage: apply
91    script:
92      - cd ${TF_ROOT}
93      - terraform apply -input=false $PLAN
94    dependencies:
95      - plan production
96    artifacts:
97      name: $CI_COMMIT_REF_SLUG
98      untracked: true
99    only:
100     - prd
101     - hom
102     - dev
103   when: manual
104   tags:
105     - docker
106
107 destroy:
108   stage: destroy
109   script:
110     - cd ${TF_ROOT}
111     - terraform destroy -auto-approve -var="efs=$efs_dns" -var="
          environment=$CI_COMMIT_REF_NAME"
112   when: manual
113   only:
114     - prd
115     - hom
116     - dev
117   tags:
118     - docker
```

Código Fonte B.2 – echoserver/.gitlab-ci.yml

B.3 *backend.tf*

```
1  terraform {
2    backend "s3" {}
3  }
```

Código Fonte B.3 – echoserver/backend.tf

B.4 *echoserver-deployment.tf*

```
resource "kubernetes_deployment" "echoserver-deploy" {
  metadata {
    name      = "echoserver-deploy"
    namespace = kubernetes_namespace.echoserver.metadata.0.name
    labels = {
      app = "echoserver"
    }
  }
  lifecycle {
    ignore_changes = [
      metadata[0].annotations,
      metadata[0].labels,
    ]
  }
  spec {
    selector {
      match_labels = {
        app = "echoserver"
      }
    }
    strategy {
      type = "RollingUpdate"
    }
    template {
      metadata {
        labels = {
          app = "echoserver"
        }
      }
      spec {
        container {
          name              = "echoserver"
          image             = "gcr.io/google_containers/echoserver:1.0"
          image_pull_policy = "Always"

          port {
            container_port = "8080"
            name           = "http"
          }

        }
      }
    }
  }
  timeouts {
    create = "5m"
```

```
    update = "5m"
    delete = "2m"
  }
  depends_on = [
    kubernetes_namespace.echoserver,
  ]
}
```

Código Fonte B.4 – echoserver/echoserver-deployment.tf

B.5 *echoserver-ingress.tf*

```
resource "kubernetes_ingress" "echoserver-ingress" {
  metadata {
    name      = "echoserver-ingress"
    namespace = kubernetes_namespace.echoserver.metadata.0.name
    labels = {
      app = "echoserver"
    }
    annotations = {
      "kubernetes.io/ingress.class" = "nginx"
    }
  }
  lifecycle {
    ignore_changes = [
      metadata[0].annotations,
      metadata[0].labels,
    ]
  }
  spec {
    rule {
      host = "echoserver.gitops.com.br"
      http {
        path {
          backend {
            service_name = kubernetes_service.echoserver-service.
               metadata.0.name
            service_port = 80
          }
          path = "/"
        }
      }
    }
  }
  depends_on = [
    kubernetes_namespace.echoserver,
  ]
```

```
}
```

Código Fonte B.5 – echoserver/echoserver-ingress.tf

B.6 *echoserver-namespace.tf*

```
resource "kubernetes_namespace" "echoserver" {
  metadata {
    annotations = {
    }
    name = "echoserver"
  }
  lifecycle {
    ignore_changes = [
      metadata[0].annotations,
      metadata[0].labels,
    ]
  }
}
```

Código Fonte B.6 – echoserver/echoserver-namespace.tf

B.7 *echoserver-service.tf*

```
resource "kubernetes_service" "echoserver-service" {
  metadata {
    name      = "echoserver-service"
    namespace = kubernetes_namespace.echoserver.metadata.0.name
    labels = {
      app = "echoserver"
    }
  }
  lifecycle {
    ignore_changes = [
      metadata[0].annotations,
      metadata[0].labels,
    ]
  }
  spec {
    type = "ClusterIP"

    selector = {
      app = "echoserver"
    }
    port {
      port        = "80"
```

```
23        target_port = "8080"
24        name        = "http"
25        protocol    = "TCP"
26      }
27    }
28    depends_on = [
29      kubernetes_namespace.echoserver,
30    ]
31  }
```

Código Fonte B.7 – echoserver/echoserver-service.tf

B.8 *providers.tf*

```
1  provider "kubernetes" {
2  }
```

Código Fonte B.8 – echoserver/providers.tf

B.9 *variables.tf*

```
1  variable "environment" {
2    type = string
3  }
4  variable "efs" {
5    type = string
6  }
```

Código Fonte B.9 – echoserver/variables.tf

B.10 *versions.tf*

```
1  terraform {
2    required_providers {
3      kubernetes = {
4        source  = "hashicorp/kubernetes"
5        version = "1.13.3"
6      }
7    }
8    required_version = ">= 0.13"
9  }
```

Código Fonte B.10 – echoserver/versions.tf

C Artigo

GitOps: Uma nova proposta para a infraestrutura

Roberto Rivelino Ventura da Silva

Curso de Bacharelado em Sistemas de Informação – Departamento de Informática e Estatística– Universidade Federal de Santa Catarina (UFSC) – 88040-900 - Florianópolis – SC – Brasil

robertorivelino3@gmail.com

Abstract. *This present research aims to develop a more agile metodology when compared to the traditional operations of infrastructure creation. Such operations carry out the processes manually, making the hours spent on elaboration of resources grow significantly due to the lack of automated procedures that could shorten the gap between conception of the infrastructure and the customer's availability. The GitOps metodology was selected as a way of automating such processes. This metodology aims to assemble the infrastructure declaration and applications in a single Git source. Its main advantage is to reduce time and cost of turning an idea into a product using the steps Continuous Integration and Continuous Delivery, making the process as a whole more optimized. Inicially, the research elaborates about DevOps culture and its precepts, which are fundamental to the implementation of a GitOps metodology. Later, the research presents the chosen process step by step, since its planning to the creation of an infrastructure environment and a example of application.*

Resumo. *O presente trabalho propõe-se a desenvolver uma metodologia mais ágil em comparação às operações tradicionais de criação da infraestrutura. Tais operações realizam processos manuais, e o tempo gasto na elaboração dos recursos cresce expressivamente diante da falta de procedimentos automatizados que encurtem o intervalo entre a concepção da infraestrutura e a disponibilidade para o cliente. Utilizou-se então a metodologia GitOps como forma de automatizar tais processos. Esta metodologia tem como propósito centralizar a declaração da infraestrutura e das aplicações em uma única fonte Git). Sua principal vantagem é encurtar o tempo e reduzir o custo de transformar uma ideia em produto através das etapas de Integração e Entrega Contínua, tornando o processo como um todo mais otimizado. Inicialmente, o trabalho trata de explicar a cultura DevOps e seus conceitos e regras, que são fundamentais para a implantação de uma metodologia GitOps. No desenvolvimento, demonstra-se o processo adotado desde seu planejamento até a criação de um ambiente de infraestrutura e de uma aplicação exemplo.*

1. Introdução

Em meados de 2000, devido aos avanços na tecnologia e na adoção de princípios e práticas ágeis, o tempo necessário para desenvolver novas funcionalidades de *software* foi reduzido para semanas ou meses. Para prestar um serviço ágil, empresas precisaram melhorar seus processos e adotar práticas de automação que tornassem possível cortar gastos, tempo de desenvolvimento e implementação de um novo recurso.

Junto com o crescimento da utilização de microsserviços e a necessidade do mercado em criar um gerenciador de *cluster* para contêiner, surge a ferramenta *Kubernetes*, um mecanismo de orquestração de contêineres de código aberto utilizado para automatizar a implantação, dimensionamento e gerenciamento de aplicativos em contêiner.

Dentre as organizações que utilizam *Kubernetes* surge também a necessidade de uma solução de entrega contínua que traga segurança, usabilidade e estabilidade nos processos de implementação. Com o *GitOps*, o uso de agentes de *software* pode alertar sobre qualquer divergência entre o *Git* e o que está sendo executado em um *cluster* e, se houver uma diferença, os controladores *Kubernetes* atualizam ou revertem automaticamente a alteração.

O objetivo geral deste trabalho consiste em criar um ambiente de infraestrutura através de processos automatizados e ferramentas *DevOps*, garantindo também a entrega contínua das aplicações de maneira ágil, utilizando como meio a metodologia *GitOps*.

2. Fundamentação Teórica

3.1. Sistema de Controle de Versão

Os sistemas de controle de versão são uma categoria de ferramentas de *software* que ajudam equipes de *software* a gerenciar alterações no código-fonte com o passar do tempo. O Sistema de controle de versão mantém registro de todas as modificações no código em um tipo especial de banco de dados. Se um erro for cometido, os desenvolvedores podem reverter as alterações e comparar versões anteriores do código para ajudar a corrigir o erro enquanto diminuem interrupções para todos os membros da equipe (ATLASSIAN, 2020).

Existe no mercado uma série de ferramentas de Sistema de Controle de Versão, cada uma com sua respectiva estrutura e topologia.

3.2. Integração Contínua

De acordo com Pittet (2020) a Integração Contínua (ou *Continuous Integration* - CI) é a prática de automatizar a integração das alterações de código de vários contribuidores de uma equipe em um único projeto de *software*. Portanto, um Sistema de Controle de Versão é um fator crucial para seu funcionamento.

A Integração Contínua visa melhorar a qualidade do *software* e reduzir o tempo de entrega, substituindo a prática tradicional de aplicar o controle de qualidade após a conclusão de todo o desenvolvimento (PEPGOTESTING, 2020). O processo de *CI* é composto de ferramentas automáticas como *Jenkins* ou *GitLab CI*, que garantem a correção do novo código antes da integração.

Um dos principais benefícios de adotar a Integração Contínua é a economia de tempo durante o ciclo de desenvolvimento da aplicação, tornando possível a identificação e resolução de conflitos em fases iniciais de um projeto. Também é uma ótima maneira de reduzir a quantidade de tempo gasto na correção de problemas, colocando maior ênfase em possuir um bom conjunto de testes (PITTET, 2020).

3.3 Entrega Contínua

Entrega contínua (ou *Continuous Delivery - CD*) é o resultado de uma Integração Contínua bem-sucedida, quando o *software* atualizado pode ser liberado para produção a qualquer momento (PEPGOTESTING, 2020).

Segundo Ebermann (2019), após a criação de uma nova versão de *software*, com qualidade aceitável a partir de um fluxo de Integração Contínua, a próxima etapa é implantar o código em um sistema acessível, para que possa ser usado por outros desenvolvedores. Esta implementação e execução das versões de desenvolvimento a cada alteração de código chama-se Entrega Contínua e deve sempre ser utilizada junto à Integração Contínua, o chamado *CI/CD*. A junção dos dois fluxos automatiza todo o percurso de codificação, construção, criação de versão, testes, implantação e execução do *software* e os aplica em um ambiente de desenvolvimento até estar apto para produção.

3.4. Infraestrutura Imutável

Infraestrutura Imutável é um paradigma de infraestrutura no qual os ambientes nunca são modificados depois de implantados. Caso algum componente precise ser atualizado, consertado ou modificado, novos componentes são construídos com as alterações apropriadas, sendo novamente provisionados para substituir os antigos.

Os benefícios de uma infraestrutura imutável incluem maior consistência e confiabilidade em sua infraestrutura, além de um processo de implantação mais simples e previsível. Tal metodologia mitiga problemas que são comuns em infraestruturas mutáveis, como desvio de configuração e servidores legados com bastante alterações manuais (VIRDÓ, 2017).

3.5. *DevOps*

DevOps é um movimento cultural que busca aprimorar o desenvolvimento de *software* e a vida profissional das pessoas envolvidas (DAVIS; DANIELS, 2015). Pode ser definido como uma abordagem organizacional e cultural que se concentra em colaboração e integração de desenvolvimento e operação para produzir produtos e serviços de *software* com maior rapidez e melhor qualidade (DÍAZ et al., 2019).

Conforme Bass, Weber e Zhu (2015), *DevOps* é um conjunto de práticas com intenção de reduzir o tempo entre uma alteração no sistema e a mudança em produção, garantindo alta qualidade. As práticas de *DevOps* impactam processos, produtos, estruturas organizacionais e práticas de negócio, portanto, a adoção de suas normas não costuma ser implementada de forma suave. A mudança revolucionária de sua natureza introduz uma grande tensão à organização e seus profissionais.

3.6. Diferença entre *Tradicional Ops* e *DevOps*

De acordo com Vehent (2018), o objetivo do *DevOps* é encurtar o tempo e reduzir o custo de transformar uma ideia em produto, para isso utiliza-se intensos processos automatizados que visam acelerar o desenvolvimento e a implantação de suas aplicações.

Em uma operação tradicional (*Tradicional Ops*) o tempo entre a concepção da infraestrutura e a disponibilidade para o cliente utilizá-la costuma durar 8 dias. Implantar essa infraestrutura consome a maior parte do tempo, pois os profissionais encarregados precisam criar todos os componentes necessários para hospedagem do *software* de forma manual.

Utilizando a abordagem *DevOps*, conseguimos reduzir esse tempo entre concepção da infraestrutura e disponibilidade em até dois dias utilizando apenas processos automatizados que lidam com o provisionamento destes componentes. A Figura 1 ilustra a comparação entre as duas abordagens.

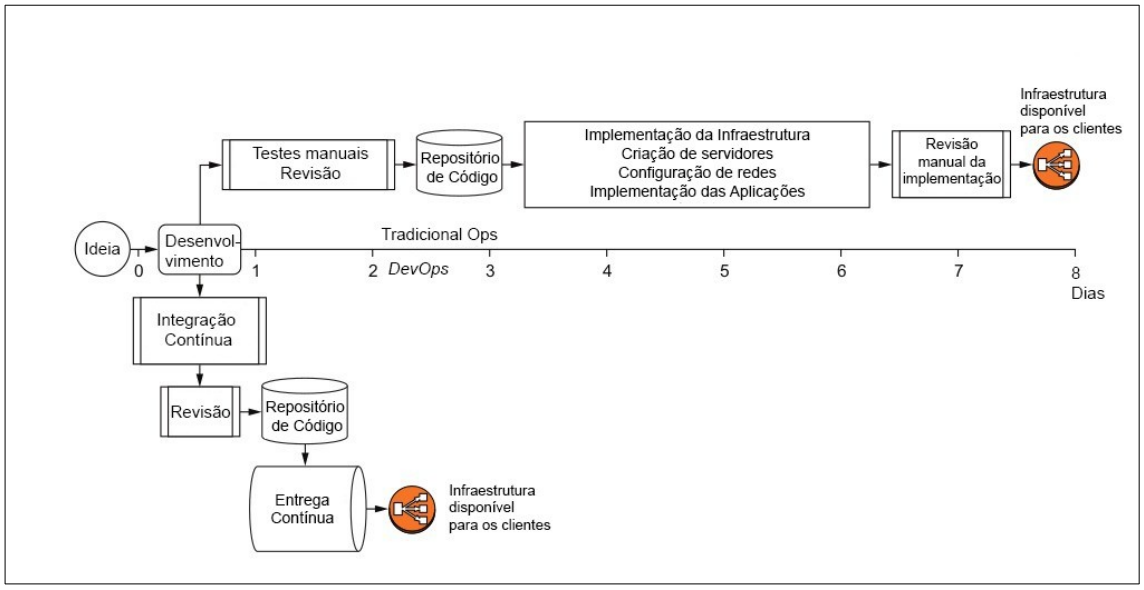

Figura 1. *Tradicional Ops* vs. *DevOps* (Adaptado de Vehent (2018)

3.7. Infraestrutura como Código

Infraestrutura como código (*Infrastructure as Code - IaC*) consiste no gerenciamento da infraestrutura (redes, máquinas virtuais, balanceadores de carga e topologias de conexão) em um modelo descritivo, utilizando o mesmo controle de versão que a equipe *DevOps* aplica para o código-fonte. Assim como o princípio de que o mesmo código-fonte gera o mesmo binário, um modelo *IaC* gera o mesmo ambiente toda vez que é aplicado. *IaC* é uma prática importante de *DevOps* e é usada em conjunto com Entrega Contínua (GUCKENHEIMER, 2017).

Ainda de acordo com Guckenheimer (2017), à medida que o código de desenvolvimento das aplicações evolui para resolver um problema, aumenta-se também sua complexidade. Dessa forma, os ambientes de infraestrutura manuais tornam-se gradualmente confusos, pois dependem de uma única configuração de difícil replicação.

Como necessidade para solucionar a complexidade nos ambientes de infraestrutura manuais, utiliza-se o *IaC*, que contém como essência a infraestrutura imutável. Este processo é realizado através de ferramentas que provisionam um ambiente de infraestrutura completo, utilizando como seu principal fundamento linguagens de configuração declarativa.

Este método de criação da infraestrutura abre espaço para a utilização e aplicação de ferramentas de desenvolvimento de *software*, como Sistema de Controle de Versão, Integração Contínua e Entrega Contínua.

3.8. *GitOps*

Git é um Sistema de Controle de Versão de código aberto projetado e desenvolvido por Linus Torvalds, o mesmo criador do sistema operacional *Linux*. *Git*Op é uma metodologia utilizada para atender a entrega contínua e o gerenciamento de um *cluster Kubernetes*, que funciona usando o *Git* como única fonte para a infraestrutura declarativa das aplicações.

Com o *GitOps*, é possível identificar e alertar sobre qualquer divergência entre o sistema e as informações inseridas no *Git*. O conceito do *GitOps* concentra-se em ter um repositório *Git* que contenha descrições declarativas da infraestrutura desejada e um processo automatizado que torne possível ao estado do ambiente sempre corresponder ao estado descrito no repositório (WEAVEWORKS, 2020).

De acordo com a empresa WeaveWorks (2020) criadora da metodologia, para iniciar o gerenciamento do *cluster Kubernetes* utilizando o fluxo de trabalho do *GitOps* os seguintes princípios devem ser seguidos:

- Todo ambiente deve estar descrito declarativamente;
- O estado desejado do sistema deve estar versionado no *Git*;
- Permitir que quaisquer mudanças nesse estado sejam aplicadas automaticamente ao seu sistema.

Ainda de acordo com a WeaveWorks (2020), os principais benefícios na utilização dessa metodologia são:

- Aumento de produtividade: com o *GitOps* as equipes precisam se preocupar menos com a implantação das versões geradas, permitindo despender mais tempo no desenvolvimento das aplicações;
- Os desenvolvedores podem utilizar ferramentas familiares como o *Git* para gerenciar atualizações e recursos para o *Kubernetes*, sem precisar ter experiência com sistemas *Kubernetes*;
- Maior confiabilidade: como todo o sistema é descrito no *Git*, torna-se mais simples reverter alterações e solucionar falhas de sistema, diminuindo o tempo de recuperação;
- Consistência e padronização: com o *GitOps* é possível criar um modelo padronizado para criação e alteração tanto da infraestrutura quanto das aplicações, tornando o fluxo de trabalho consistente;

- Garantia de maior segurança: com o *Git* é possível ter o rastreamento e gerenciamento de alterações de código, bem como a capacidade de assinar alterações que comprovam sua autoria e origem.

4. Ferramentas

4.1. *Kubernetes*

Kubernetes é um orquestrador de código aberto utilizado para implantar e gerenciar contêineres *Docker* em máquinas virtuais ou físicas que façam parte de seu próprio *cluster Kubernetes*.

Os recursos são declarados em arquivos de configuração chamados manifest que utilizam uma linguagem de serialização de dados chamada *YAML - Yet Another Markup Language*, que podem ser removidos ou atualizados a partir da ferramenta de linha de comando *kubectl*. Para separar múltiplos projetos em um mesmo *cluster Kubernetes*, geralmente é utilizado um delimitador abstrato chamado espaço de nomes ou namespace (EBERMANN, 2019).

4.2. *GitLab* e *GitLab CI/CD*

GitLab é um sistema utilizado para gerenciar repositórios *Git*. Escrito em *Ruby*, permite a implantação de um controle de versão para o seu código de forma rápida e fácil. Foi publicado pela primeira vez na plataforma de hospedagem de código-fonte *GitHub* em outubro de 2011 e tornou-se uma ferramenta poderosa desde então. Fundada por Dmitriy Zaporozhets, hoje conta com uma plataforma de hospedagem própria que pode ser utilizada gratuitamente por qualquer profissional ou entusiasta da área (HETHEY, 2013).

Já o *GitLab CI/CD* é uma ferramenta incorporada ao *GitLab* para desenvolvimento de *software* por meio de metodologias contínuas como Integração Contínua), Entrega Contínua e Implantação Contínua.

A Integração Contínua funciona enviando pequenos pedaços de código para o projeto hospedado em um repositório *Git* e, para cada ação de inserção ou alteração de código, executa um fluxo de tarefas para construir, testar e validar as alterações antes de consolidá-las na ramificação do código principal.

A Entrega Contínua e Implantação Contínua consistem em uma etapa adicional de Integração Contínua que fornece a implantação da aplicação para o ambiente desejado (integração/desenvolvimento, homologação ou produção) a cada inserção de código (*GitLab INC.*, 2020).

4.3. *Terraform*

Terraform é uma ferramenta de código aberto de provisionamento de infraestrutura, criada pela *HashiCorp*, que permite que definamos nossa Infraestrutura como Código, usando uma linguagem simples e declarativa (SOUZA, 2017).

Conforme Brikman (2016), o binário do *Terraform* criado na linguagem *Go* permite que seja criada uma infraestrutura inteira (bancos de dados, balanceadores de carga, topologias de rede, máquinas virtuais, dentre outros recursos) de seu próprio computador ou de um servidor sem precisar executar qualquer outra funcionalidade ou

ferramenta. Isso é possível devido às chamadas de acesso às plataformas de provedores que o binário realiza através dos recursos do *Terraform*.

5. Desenvolvimento

A proposta deste trabalho compreende os seguintes pontos:

- Integrar os processos de desenvolvimento de *software* (*Dev*) e operacionais (*Ops*) através das práticas *DevOps* tratadas no trabalho;
- Provisionar um ambiente de Infraestrutura Imutável de forma automática com os recursos abordados anteriormente na *AWS*, declarados estritamente através da Infraestrutura como Código, tendo *Kubernetes* como seu gerenciador e o *GitLab* como Sistema de Controle de Versão;
- Tornar possível a Integração e Entrega contínua desta infraestrutura e das aplicações através das ferramentas de *CI/CD* contidas no *GitLab*;
- Garantir que o estado descrito no ambiente de infraestrutura e gerado através da ferramenta de Infraestrutura como Código *Terraform*, esteja em conformidade com os estados dos repositórios de infraestrutura e aplicações presentes no *GitLab* utilizando como preceito a metodologia *GitOps*.

O diagrama presente na Figura 2 ilustra as metodologias, ferramentas e recursos que são utilizados para o desenvolvimento do trabalho e como é realizada a interação entre eles.

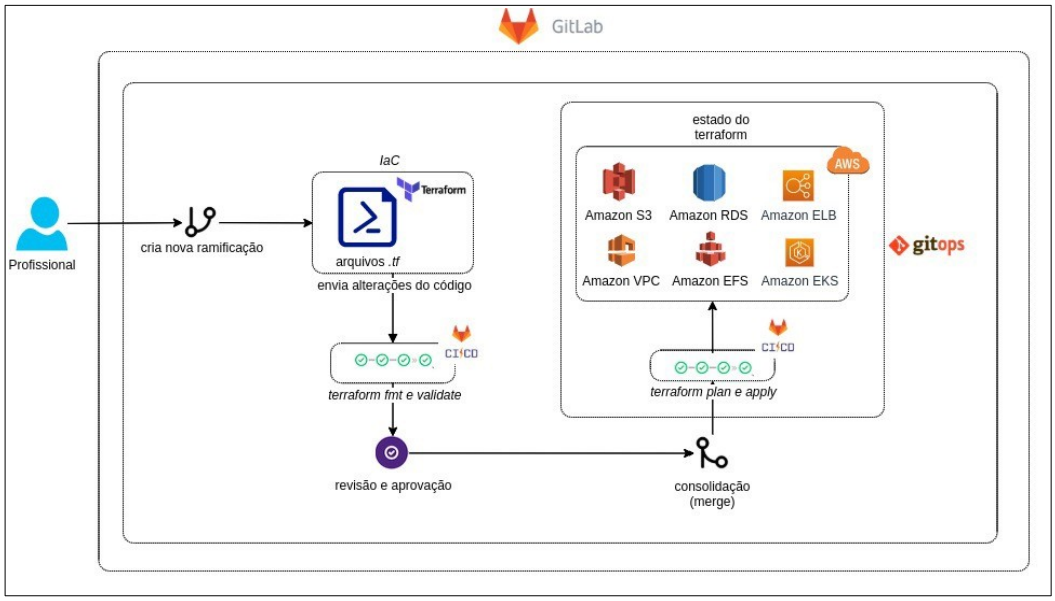

Figura 2. Diagrama das ferramentas utilizadas (Criado pelo autor)

5.1. *AWS*

A plataforma de serviços de computação em nuvem escolhida para o desenvolvimento do trabalho foi a *Amazon Web Services*. Os recursos utilizados para o provisionamento de um ambiente completo de infraestrutura na *AWS* foram: criação de máquinas virtuais (*EC2*), criação das redes (*VPC*), definição do sistema de arquivos (*EFS*), criação do banco de dados do ambiente (*RDS*), definição do balanceador de

carga para acesso ao ambiente (*NLB*) e criação do serviço de armazenamento do estado do ambiente (*S3*). Os códigos das aplicações são executados em um *cluster Kubernetes* (*EKS*). Todos os recursos foram provisionados na região leste dos *EUA* (*Ohio*) – *us-east-2*.

5.2. *GitOps*

Através da adoção da metodologia *GitOps*, é possível concentrar os arquivos de declaração da infraestrutura e da aplicação exemplo em um único Sistema de Controle de Versão . Esta etapa mantém o estado atual dos projetos em sincronia com o ambiente de infraestrutura proposto, através da Entrega e Integração Contínua.

Para ter acesso à *AWS* através do *GitLab*, é necessário a criação de variáveis de ambiente que permitam a autenticação ao sistema. Sendo elas:

- *AWS_ACCESS_KEY_ID*: Chave de acesso *AWS* usada para autenticar o usuário;
- *AWS_SECRET_ACCESS_KEY*: Senha de acesso *AWS* usada para autenticar o usuário.

Estas variáveis são utilizadas no *GitLab CI/CD*, permitindo a autenticação e execução do fluxo de tarefas na conta da *AWS*.

Há também a criação da variável de ambiente *dev_k8s_kubeconfig* que permite acesso ao *cluster* através do binário *kubectl*, podendo assim criar, modificar ou remover manifestos que compõem as aplicações.

Este arquivo chamado *kubeconfig* é utilizado para armazenar informações de autenticação do *cluster Kubernetes*. A variável é gerada no estágio de *Apply* assim que o provisionamento do *EKS* é finalizado.

Na Figura 3 são mostradas as variáveis citadas acima pela interface gráfica do *GitLab*.

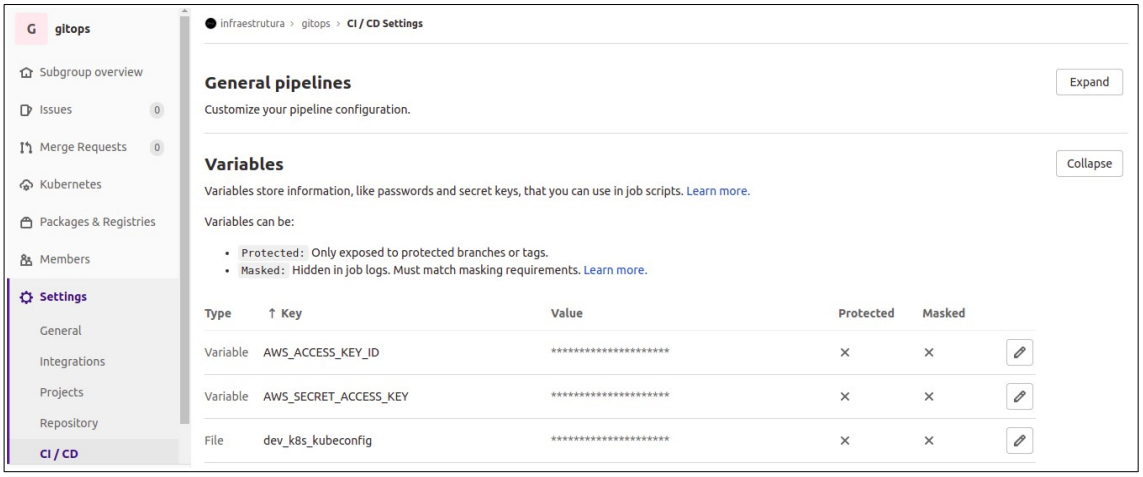

Figura 3. Variáveis de ambiente (Criado pelo autor)

5.2.1. Provisionamento da Infraestrutura

Para o provisionamento da infraestrutura, os arquivos de configuração *.tf* do *Terraform* são registrados (*commit*) em um projeto no *GitLab*, possibilitando o disparo automático do fluxo de tarefas presente no arquivo *.gitlab-ci.yml*.

Quando um registro é gerado, o fluxo de tarefas dispara e damos início ao provisionamento dos recursos na *AWS*.

O primeiro fluxo de tarefas executado é encarregado de gerar o estado do *Terraform* no *bucket*, estado este atualizado toda vez que alguma alteração for feita nos arquivos de configuração do *Terraform* através do disparo de um novo fluxo de tarefas.

Na Figura 4 podemos observar o primeiro fluxo de tarefas e seus estágios *Validate*, *Plan*, *Apply*, *e Destroy* no Sistema de Controle de Versão.

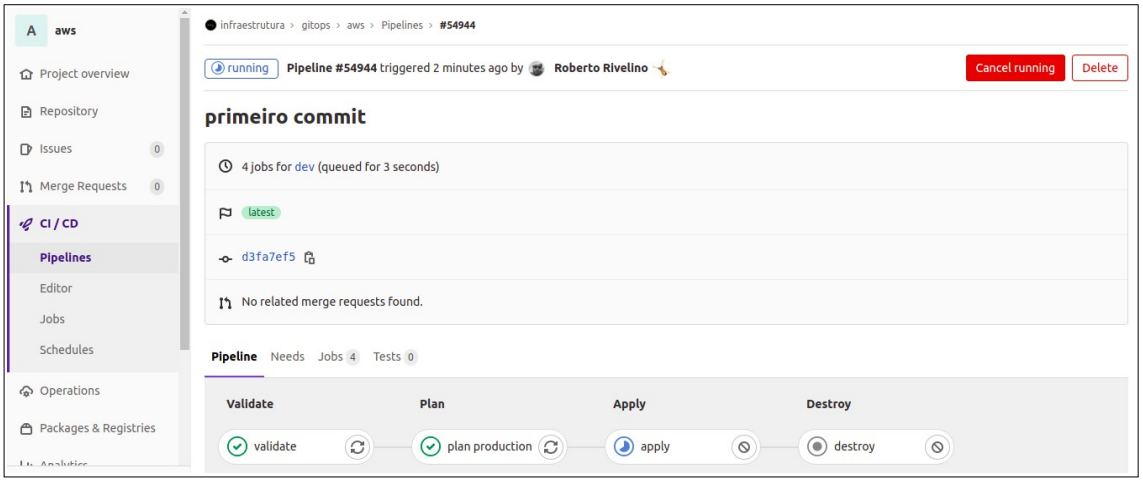

Figura 4. Primeiro fluxo de tarefas (Criado pelo autor)

A cada novo registro, é criado um histórico de alterações no Sistema de Controle de Versão. Este histórico possibilita ter um controle de versionamento dos arquivos modificados. Na Figura 5 temos um exemplo de versionamento do arquivo de configuração *rds.tf*.

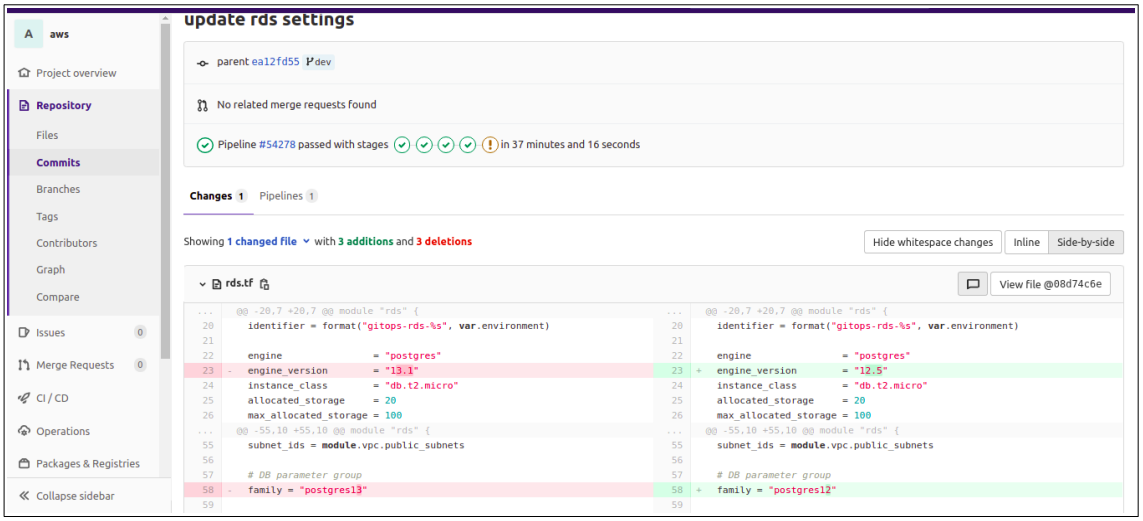

Figura 5. Controle de versão do arquivo *RDS*.tf (Criado pelo autor)

Durante a execução do *Apply* já é possível visualizar a criação dos recursos na interface gráfica da *AWS*. Nas figuras 6 e 7 podemos observar alguns recursos sendo provisionados.

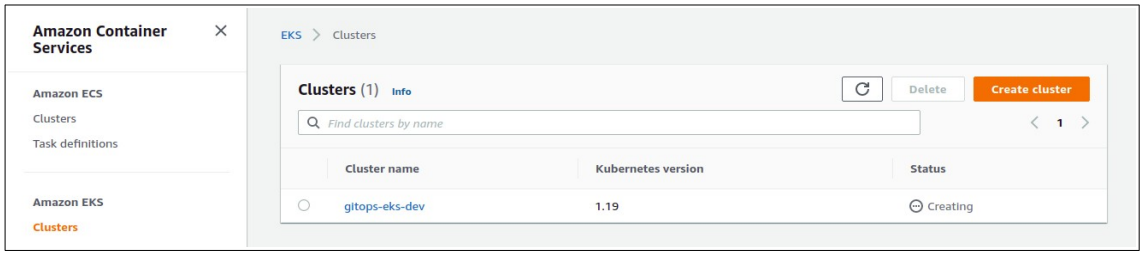

Figura 6. Provisionamento do *cluster Kubernetes* - *EKS* (Criado pelo autor)

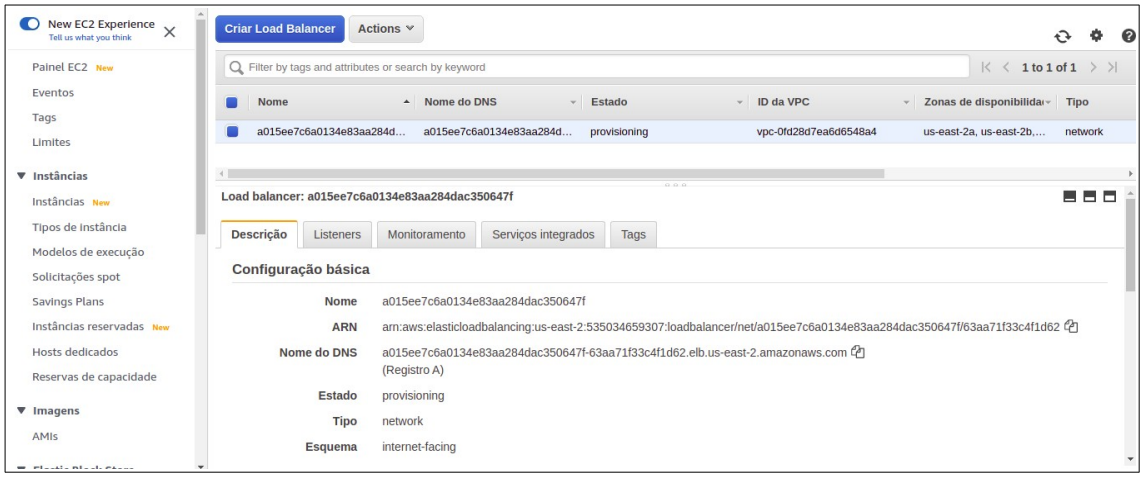

Figura 7. Network Load Balancer – *NLB* (Criado pelo autor)

Ao final da execução do *Apply* é gerado o arquivo de estado do *Terraform* que é armazenado no *bucket gitops-tfstate*, como mostra a Figura 8.

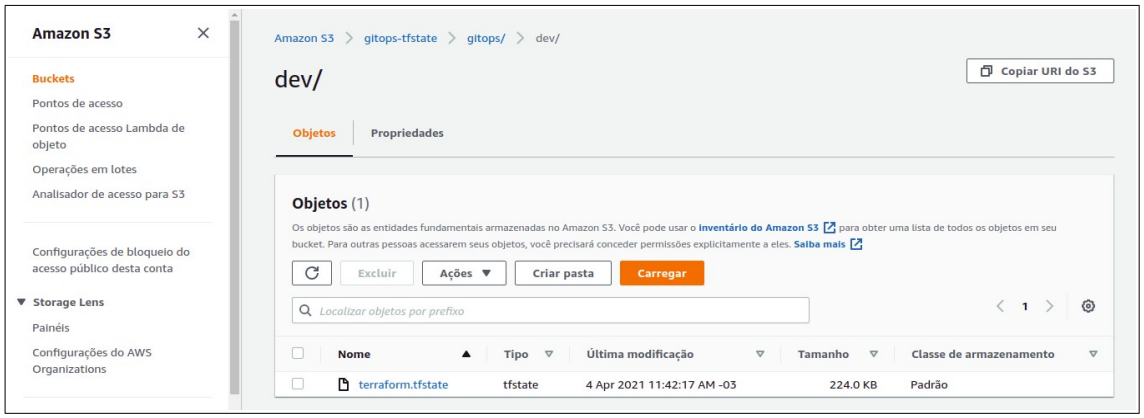

Figura 8. Estado do *Terraform* salvo no *bucket gitops*-tfstate (Criado pelo autor)

5.2.2. Aplicação exemplo

A aplicação escolhida para a criação através da Infraestrutura como Código utilizando as diretrizes da metodologia *GitOps* foi o *echoserver*, que mostra as informações de cabeçalho, servidor e cliente de um servidor *web*.

Os manifestos desta aplicação foram transformados de *YAML* para a linguagem do *Terraform*. Os arquivos originais podem ser encontrados no *link* https://gist.github.com/chukaofili/d0a6713734d0953ce1ce667958464edb. Para o

provisionamento do *echoserver* foram necessários os arquivos de configuração *Namespace*, *Deployment*, *Service*.

No arquivo de configuração *.gitlab-ci.yml* do *echoserver*, em *before_scripts*, há a adição da variável *dev_k8s_kubespray* que, conforme mencionada anteriormente, fornece a autenticação ao *cluster EKS*.

Após os arquivos de configuração do *echoserver* estarem declarados, é feito o registro em seu respectivo projeto e executado o fluxo de tarefas de criação dos recursos como podemos ver nas figuras 9 e 10.

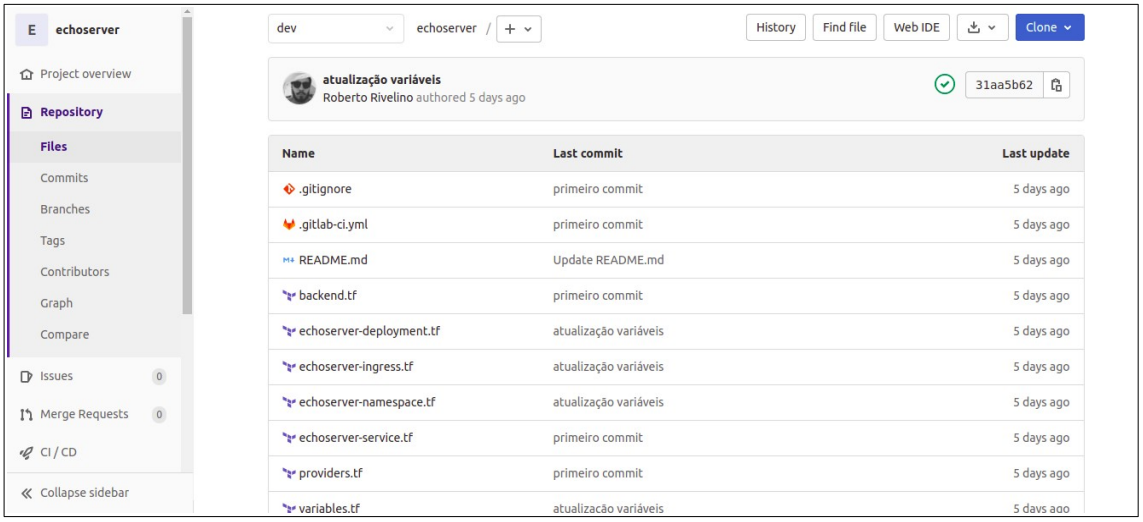

Figura 9. Registros do projeto *echoserver* (Criado pelo autor)

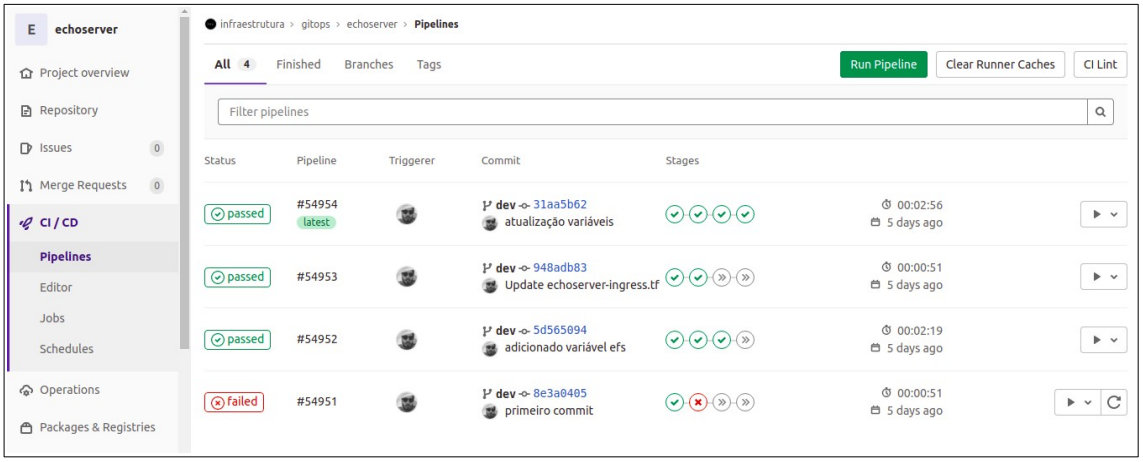

Figura 10. Histórico de alterações através dos fluxos de tarefas (Criado pelo autor)

Depois do fluxo de tarefas executado, é possível acessar a aplicação *echoserver* pelo navegador. Na Figura 11 temos a aplicação *echoserver* em funcionamento:

Figura 11. Aplicação *echoserver* em funcionamento (Criado pelo autor)

6. Conclusão e Trabalhos Futuros

O intuito deste trabalho foi estimular a criação, configuração e gerenciamento de um ambiente de infraestrutura utilizando ferramentas de Infraestrutura como Código.

Implantar um ambiente de infraestrutura desta complexidade utilizando os preceitos do *Tradicional Ops* torna-se obsoleto quando são visíveis os benefícios ofertados pelo modelo proposto. Quando os processos manuais de criação da infraestrutura são adotados, o tempo gasto na elaboração dos recursos cresce expressivamente, diante da falta de procedimentos automatizados que encurtem o intervalo entre a concepção da infraestrutura e a disponibilidade para o cliente.

O tempo de suporte nas aplicações também aumenta em contrapartida ao modelo *GitOps*, pois as alterações nestas aplicações demandam processos manuais e não guardam um estado real, pecando em consistência e confiabilidade, ferindo o conceito de Infraestrutura Imutável. Portanto, através das metodologias de desenvolvimento da infraestrutura, práticas *DevOps*, metodologia *GitOps* e demais ferramentas adotadas no trabalho, foi possível atingir o objetivo geral de criar um ambiente de infraestrutura automatizado que garanta a entrega contínua das aplicações de maneira ágil. Os objetivos específicos, que derivavam sobretudo da criação deste ambiente, também foram alcançados.

Por meio da metodologia *GitOps*, foi possível manter um histórico único de alterações no Sistema de Controle de Versão - *VCS* utilizado e garantir que o estado da infraestrutura e das aplicações sempre estarão em conformidade com os registros declarados no *VCS* sem a necessidade de alterações manuais. Este processo foi realizado através da Integração e Entrega Contínua (*CI/CD*), que permitiram dispender menor tempo na resolução de conflitos durante o ciclo de desenvolvimento da infraestrutura e da aplicação, disponibilizando-as para o uso.

Contudo, durante o desenvolvimento deste trabalho, outras ferramentas *GitOps* foram aprimoradas, tornando o processo de sincronização de estado entre aplicação e arquivos declarados no *VCS* ainda mais ágil. Dentre elas, por exemplo, temos: *Argo CD*, Flux e *Jenkins*X. Assim, será possível explorá-las em uma futura extensão do trabalho.

Referências

AMBLER, Scott W.; LINES, Mark. **Disciplined Agile Delivery**: a practitioner's guide to agile *software* delivery in the enterprise. Upper Saddle River, Nj: Ibm Press, 2012. 544 p.

ATLASSIAN (Austrália). **O que é controle de versão**. Disponível em: <https://www.atlassian.com/br/git/tutorials/what-is-version-control>. Acesso em: 17 out. 2020.

AWS. **What is a Network Load Balancer?**. Elaborada pela Amazon Web Services, Inc. Disponível em: <https://docs.aws.amazon.com/elasticloadbalancing/latest/network/introduction.html>. Acesso em: 02 abr. 2021.

AWS. **Security groups for your VPC**. Elaborada pela Amazon Web Services, Inc. Disponível em: <https://docs.aws.amazon.com/vpc/latest/userguide/VPC_SecurityGroups.html>. Acesso em: 02 abr. 2021.

BASS, Len; WEBER, Ingo; ZHU, Liming. ***DevOps***: a *software* architect's perspective. Old Tappan Road, Tappan Road - Nj: Pearson Education, Inc., 2015. 352 p.

BELCHIOR, Rafael. ***DevOps*101—First Steps on *Terraform***: *Terraform* + openstack + ansible. 2018. Elaborada por Bitcoin Insider. Disponível em: <https://www.bitcoininsider.org/article/51729/*DevOps*101-first-steps-*Terraform*-*Terraform*-openstack-ansible>. Acesso em: 08 out. 2020.

BRIKMAN, Yevgeniy. ***Terraform* Up & Running**: writing *Infrastructure as Code*. Sebastopol, Ca: O'Reilly Media, Inc, 2016. (ISBN 978-1-491-97703-3).

BURNS, Brendan; HIGHTOWER, Kelsey; BEDA, Joe. ***Kubernetes* Up & Running:** dive into the future of infrastructure. Sebastopol, Ca: O'Reilly Media, Inc., 2017. 354 p. Acesso em 31 out. 2020.

BUYYA, Rajkumar; YEOA, Chee Shin; BROBERG, James; BRANDIC, Ivona; VENUGOPAL, Srikumar. **Cloud computing and emerging IT platforms:** Vision, hype, and reality for delivering computing as the 5th utility. Future Generation Computer Systems. 03/12/2008. Elsevier. Disponível em: <http://www.buyya.com/papers/Cloud-FGCS2009.pdf>. Acesso em 31 out. 2020.

DAVIS, Jennifer; DANIELS, Katherine. **Effective *DevOps***: building a culture of collaboration, affinity, and tooling at scale. Gravenstein Highway North, Sebastopol, Ca: O'Reilly Media, Inc., 2015. 378 p.

Docker. **Get Started**: Quickstart: Part 1 Orientation and setup. 2020. Disponível em: <https://docs.*Docker*.com/get-started/>. Acesso em: 26 out. 2020.

EBERMANN, Alwin. **Evaluation of *GitOps* Security in a *CI/CD* Environment**. 2019. 59 f. TCC (Graduação) - Curso de Electrical And Computer Engineering, Department Of Electrical And Computer Engineering, Technical University Of Munich, Munich, Germany, 2019.

GitLab INC. (org.). **Choosing between *GitLab*.com and self-managed subscriptions**. Elaborada por *GitLab* Inc.. Disponível em: <https://about.*GitLab*.com/handbook/marketing/strategic-marketing/dot-com-vs-self-managed/>. Acesso em: 02 nov. 2020.

GitLab INC. (org.). ***GitLab CI/CD***. Elaborada por *GitLab* Inc.. Disponível em: <https://docs.*GitLab*.com/ee/ci/>. Acesso em: 02 nov. 2020.

GONZALEZ, David. **Implementing Modern *DevOps***: enabling it organizations to deliver faster and smarter. Birmingham, Uk: Packt, 2017.

GUCKENHEIMER, Sam. **What is *Infrastructure as Code*?** 2017. Elaborada por Microsoft. Disponível em: <https://docs.microsoft.com/en-us/azure/*DevOps*/learn/what-is-infrastructure-as-code>. Acesso em: 07 set. 2020.

HETHEY, Jonathan M.. ***GitLab* Repository Management**. Birmingham, Uk: Packt Publishing Ltd., 2013. (ISBN 978-1-78328-179-4).

KAMARUZZAMAN, Shahril Bin. **How to set up *GitLab* for *Continuous Integration* and Deployment on CentOS**. 2019. Elaborado por HowtoForge. Disponível em: <https://www.howtoforge.com/how-to-set-up-*GitLab*-server-for-ci-cd-operation-on-centos/>. Acesso em: 22 nov. 2020.

J. Díaz, J. E. Pérez, M. A. Lopez-Peña, G. A. Mena and A. Yagüe, **"Self-Service Cybersecurity Monitoring as Enabler for DevSecOps,**" in *IEEE Access*, vol. 7, pp. 100283-100295, 2019, doi: 10.1109/ACCESS.2019.2930000.

KIM, Gene et al. **The *DevOps* Handbook:** how to create world-class agility, reliability, & security in technology organizations. 25 Nw 23Rd Pl, Suite 6314 Portland, Or 97210: It Revolution Press, Llc, 2016. 105 p.

Kubernetes (org.). ***Kubernetes***. 2020. Disponível em: <https://*Kubernetes*.io/pt/docs/home/>. Acesso em: 27 set. 2020.

Kubernetes (org.). **Ingress**. 2021. Disponível em: <https://*Kubernetes*.io/docs/concepts/services-networking/ingress/>. Acesso em: 01 abr. 2021.

MORRIS, Kief. ***Infrastructure as Code***: managing servers in the cloud. Gravenstein Highway North, Sebastopol, Ca: O'Reilly Media, Inc., 2016. 447 p.

NGINX (org.) **Nginx**. 2021. Disponível em : <https://www.nginx.com/>. Acesso em: 02 abri. 2021.

PEPGOTESTING (org.). **Automated *software* testing in *Continuous Integration* (CI) and *Continuous Delivery* (CD)**. Disponível em: <https://pepgotesting.com/continuous-integration/>. Acesso em: 10 out. 2020.

PITTET, Sten. **How to get started with *Continuous Integration*.** 2020. Elaborada por Atlassian. Disponível em: <https://www.atlassian.com/continuous-delivery/continuous-integration/how-to-get-to-continuous-integration>. Acesso em: 10 out. 10.

QUEIROZ, Renan. ***DevOps* para Bancos SQL Server**. 2019. Elaborada por Medium. Disponível em: <https://medium.com/@renanlq/*DevOps*-para-bancos-sql-server-2eb5bee79d87>. Acesso em: 22 nov. 2020.

SANCHE, Daniel. ***Kubernetes* 101: Pods, Nodes, Containers, and *cluster*s**. 2018. Elaborado por Medium. Disponível em: <https://medium.com/google-cloud/*Kubernetes*-101-pods-nodes-containers-and-*cluster*s-c1509e409e16>. Acesso em: 31 out. 2020.

SOUZA, Igor. ***Terraform* - Uma pequena introdução**. 2017. Elaborada por Medium. Disponível em: <https://medium.com/@igordcsouza/*Terraform*-uma-pequena-introdu%C3%A7%C3%A3o-eae86f22db55>. Acesso em: 02 nov. 2020.

Terraform. **How *Terraform* Works**. 2020. Disponível em: <https://www.*HashiCorp*.com/products/*Terraform*>. Acesso em: 09 out. 2020.

Terraform. **Command: validate**. 2021. Disponível em: <https://www.*Terraform*.io/docs/cli/commands/validate.html>. Acesso em: 02 abr. 2021.

Terraform. **Command: plan**. 2021. Disponível em: <https://www.*Terraform*.io/docs/cli/commands/plan.html>. Acesso em: 03 abr. 2021.

VAN BAARSEN, Jeroen. ***GitLab* Cookbook**: over 60 hands-on recipes to efficiently self-host your own *Git* repository using *GitLab*. Birmingham, Uk: Packt Publishing Ltd., 2014. (ISBN 978-1-78398-684-2).

VEHENT, Julien. **Securing *DevOps***: security in the cloud. Shelter Island, Ny 11964: Manning Publications Co., 2018. 401 p. (ISBN 9781617294136).

VIRDÓ, Hazel. **What Is Immutable Infrastructure?** 2017. Elaborada por Di*Git*alOcean. Disponível em: <https://www.di*Git*alocean.com/community/tutorials/what-is-immutable-infrastructure#differences-between-mutable-and-immutable-infrastructure>. Acesso em: 12 out. 2020.

WEAVEWORKS. **Guide to *GitOps***. 2020. Disponível em: <https://www.weave.works/technologies/*GitOps*/>. Acesso em: 29 set. 2020.