PONTIFÍCIA UNIVERSIDADE CATÓLICA DE GOIÁS
ESCOLA POLITÉCNICA E DE ARTES
GRADUAÇÃO EM ENGENHARIA DA COMPUTAÇÃO

DEPLOY DE UMA APLICAÇÃO DE GESTÃO DE RESTAURANTE UTILIZANDO A PRÁTICA DE *DEVOPS*

MATHEUS GUILHERME SOUZA SANTANA

GOIÂNIA
2024

MATHEUS GUILHERME SOUZA SANTANA

DEPLOY DE UMA APLICAÇÃO DE GESTÃO DE RESTAURANTE UTILIZANDO A PRÁTICA DE *DEVOPS*

Trabalho de Conclusão de Curso apresentado à Escola Politécnica e de Artes, da Pontifícia Universidade Católica de Goiás, como parte dos requisitos para a obtenção do título de Bacharel em Engenharia da Computação.

Orientador(a):

Prof.M.E.E. Rafael Leal Martins

Banca examinadora:

Prof. Dr. Solange da Silva

Prof. Me. Olegário Correa da Silva Neto

GOIÂNIA
2024

MATHEUS GUILHERME SOUZA SANTANA

DEPLOY **DE UMA APLICAÇÃO DE GESTÃO DE RESTAURANTE UTILIZANDO A PRÁTICA DE *DEVOPS***

Trabalho de Conclusão de Curso aprovado em sua forma final pela Escola Politécnica e de Artes, da Pontifícia Universidade Católica de Goiás, para obtenção do título de Bacharel em Engenharia da Computação, em _____/_____/_____.

_____
Orientador(a): Prof.M.E.E. Rafael Leal Martins

_____
Prof. Dr. Solange da Silva

_____
Prof. Me. Olegário Correa da Silva Neto

GOIÂNIA
2024

A Deus pela minha vida e oportunidades.

Aos meus familiares.

Ao meu avô Miguel por desde pequenino ter me incentivado nos estudos e a minha avó Maria por minha criação e apoio.

AGRADECIMENTOS

Ao Professor Rafael Leal Martins, orientador acadêmico, pelo apoio e confiança depositada.

À Coordenação da Escola Politécnica e de Artes, da Pontifícia Universidade Católica de Goiás por ajudar de forma inestimável na execução deste projeto.

A todos que direta ou indiretamente colaboraram para a materialização deste trabalho.

“Não importa o quanto você vá devagar, contanto que não pare.”

Confúcio

RESUMO

Apresenta-se um estudo prático de uma implementação de uma infraestrutura utilizando a cultura de *DevOps*. Para que tal estudo fosse possível, foi realizado um estudo sobre os principais conceitos pertinentes na criação de uma infraestrutura com essa característica, conceitos como: Servidores, Arquiteturas de Servidores, Conceitos de *DevOps*, CI/CD, *Deploy* e *Containers*. Além disso também foram realizados estudos sobre o ferramental necessário para o desenvolvimento dessa infraestrutura, sendo essas: Docker, Nginx e GitLab. Para demonstrar o uso dessa arquitetura, foi desenvolvido um sistema *web* para gestão de restaurantes de nome *Baratie*, sendo desenvolvido inteiramente em Java e com SGBD PostgreSQL. O principal objetivo deste trabalho é documentar o processo de *Deploy*, de forma que seja possível posteriormente em trabalhos futuros, verificar a diferença que a cultura *DevOps* pode causar em um projeto de software. Além disso, o objetivo geral do trabalho foi implementar uma infraestrutura que utilize a cultura de DevOps em um software de gestão de restaurante, no qual foi implementado utilizando o GitLab juntamente com tecnologias como AWS, RDS e GitLab Runner. Após testes na implementação foi concluído que é possível implementar uma infraestrutura utilizando a cultura de DevOps, apesar de ter desafios, como limitação de armazenamento oferecido pela AWS e configurações manuais.

Palavras-Chave: *DevOps*, CI/CD, Docker, GitLab, Nginx.

ABSTRACT

A practical study is presented of an implementation of an infrastructure using the DevOps culture. In order to conduct this study, an exploration of key concepts relevant to building such an infrastructure was performed, including: Servers, Server Architectures, DevOps Concepts, CI/CD, Deployment, and Containers. Additionally, studies were conducted on the necessary tools for developing this infrastructure, namely: Docker, Nginx, and GitLab. To demonstrate the use of this architecture, a web system named Baratie was developed for restaurant management, entirely in Java and using PostgreSQL as the DBMS. The main objective of this work is to document the Deployment process, enabling future research to assess the impact of DevOps culture on software projects. Furthermore, the overall goal was to implement an infrastructure utilizing DevOps culture for a restaurant management software, which was achieved using GitLab along with technologies such as AWS, RDS, and GitLab Runner. After testing the implementation, it was concluded that implementing an infrastructure using DevOps culture is feasible, despite challenges such as storage limitations provided by AWS and manual configurations.

***Keywords**:* DevOps, CI/CD, Docker, GitLab, Nginx.

LISTA DE ILUSTRAÇÕES

Figura 1 - Conjuntos de Ferramentas do DevOps.....20
Figura 2 - Pipeline do CI/CD.....23
Figura 3 - Modelo Cascata..... 25
Figura 4 - Modelo em Camadas..... 27
Figura 5 - Arquitetura cliente-servidor..... 28
Figura 6 - Arquitetura do Nginx..... 32
Figura 7 - Estratégia orientada a eventos dos workers.....33
Figura 8 - Diferenças entre virtualização e containers..... 36
Figura 9 - Diferença do Docker para o LXC..... 37
Figura 10 - Pipeline de CI/CD do GitLab..... 40
Figura 11 - Diagrama Arquitetural da pipeline de CI/CD do GitLab.....41
Figura 12 - Rotina de Deploy.....46
Figura 13 - Variáveis de Ambiente do Projeto..... 50
Figura 14 - Job que cria arquivos de configuração..... 51
Figura 15 - Trecho de job que cria arquivos de configuração..... 51
Figura 16 - Build da Imagem da aplicação com versionamento..... 52
Figura 17 - Criação de Instância na AWS..... 53
Figura 18 - Portas abertas da instância.....54
Figura 19 - Comando para se conectar com a instância dev-server.....54
Figura 20 - Instalação do Runner na máquina..... 55
Figura 21 - Registro de um tipo de Runner na máquina..... 55
Figura 22 - Instalação do Java na máquina..... 56
Figura 23 - Parte do job de Deploy que configura instância de servidor.....57
Figura 24 - Criação de um IP elástico..... 58
Figura 25 - Associação de IP Estático a uma instância..... 59
Figura 26 - Opções de Banco.....60
Figura 27 - Modelos de banco de Dados..... 61
Figura 28 - Configuração da Instância..... 62
Figura 29 - Configuração de Conectividade do RDS..... 63
Figura 30 - Atual Retorno do Endpoint /teste..... 64
Figura 31 - Merge-request.....64
Figura 32 - Alterações no Código.....65
Figura 33 - Execução do Stage de Teste..... 65
Figura 34 - Console do Job de Preenchimento de Variáveis..... 67

Figura 35 - Console do job de Compilação do Projeto.....68
Figura 36 - Execução da Pipeline de Deploy..... 69
Figura 37 - Console do Job de Construção da Imagem Docker do Projeto.....70
Figura 38 - Console do job de Upload da Imagem Docker no Registry..... 71
Figura 39 - Console do job de Deploy aguardando acionamento..... 72
Figura 40 - Console do Job de Deploy..... 72
Figura 41 - Retorno do Endpoint com a alteração..... 74
Figura 42 - Retorno do endpoint /cozinheiro..... 74
Figura 43 - Logs de redirecionamento do Nginx..... 76
Figura 44 - Imagens Docker não utilizadas presentes na instância.....77

LISTA DE ABREVIATURAS

Dr	Doutor
Me(a)	Mestre(a)
M.E.E	Mestre em engenharia elétrica
Prof	Professor

LISTA DE SIGLAS

AWS	*Amazon Web Services*
CI/CD	*Continuous Integration/Continuous Delivery*
CPU	*Central Processing Unit*
DBMS	*Database Management System*
DEV	Desenvolvimento
DNS	*Domain Name System*
EC2	*Elastic Compute Cloud*
IP	*Internet Protocol*
ISO	*International Organization for Standardization*
LXC	*Linux Containers*
OPS	Operações
QA	*Quality Assurance*
RDS	*Relational Database Service*
SGBD	Sistema Gerenciador de Banco de Dados
SSH	*Secure Shell*
SO	Sistema Operacional
TCC	Trabalho de Conclusão de Curso
TI	Tecnologia da Informação
VM	*Virtual Machine*
VPC	*Virtual Private Cloud*

SUMÁRIO

1 INTRODUÇÃO..... 15
1.1 OBJETIVOS GERAIS..... 16
1.2 OBJETIVOS ESPECÍFICOS..... 16
1.3 MÉTODO..... 16
1.4 ESTRUTURA DA MONOGRAFIA..... 17
2 REFERENCIAL TEÓRICO..... 19
2.1. DEVOPS..... 19
2.2. CI/CD..... 22
2.3. DEPLOY..... 25
2.4. ARQUITETURA DE UM SISTEMA WEB..... 26
2.5 SERVIDORES..... 28
2.6 NGINX..... 31
2.7. CONTAINERS..... 34
2.8. DOCKER..... 36
2.9. GITLAB..... 38
2.10 GITLAB RUNNER..... 40
2.11 AWS..... 43
2.12 EC2..... 43
2.13 RDS..... 44
3 ESTUDO DE CASO..... 45
3.1 ESQUEMA LÓGICO DA PIPELINE..... 45
3.2 ARQUITETURA DA PIPELINE..... 47
3.3 VARIÁVEIS DE AMBIENTE..... 49
3.4 VERSIONAMENTO DE IMAGENS..... 51
3.5 CONFIGURAÇÃO DAS INSTÂNCIAS AWS..... 52
3.5 CONFIGURAÇÃO DOS RUNNERS..... 54
3.6 CONFIGURAÇÃO DO SERVIDOR DA APLICAÇÃO..... 56
3.6.1 CONFIGURAÇÃO DE IP..... 57
3.6.2 CONEXÃO COM BANCO DE DADOS..... 59
3.7 TESTES..... 63
3.7.1 TESTE DE DEPLOY..... 64
3.7.2 TESTE DE CONEXÃO COM O BANCO DE DADOS..... 74
3.7.3 TESTES COM NGINX..... 76

4 ANÁLISE DOS RESULTADOS OBTIDOS..... 77
5 CONSIDERAÇÕES FINAIS..... 79
5.1 TRABALHOS FUTUROS..... 80
REFERÊNCIAS..... 81

1 INTRODUÇÃO

Com a pandemia do Covid-19, a demanda no mercado de TI cresceu de forma significativa, segundo a CNN (2021), a procura por profissionais de tecnologia cresceu cerca de 671% no mundo e existem projeções de que o Brasil gera mais de 420 mil vagas até 2025 (G1, 2023). Com essa crescente demanda, faz-se necessário que o processo de software seja ágil, pois o mercado fica gradativamente mais dependente da tecnologia da informação. Com essa premissa de desenvolver software de forma ágil, tanto para o lado de quem programa, de quem testa e até para o usuário final, nasce a cultura *DevOps*, que vem sendo utilizada como cultura central em empresas que desenvolvem software de forma ágil. Quando se fala de adotar a cultura *DevOps* em uma empresa, na realidade significa implementar uma série de procedimentos para que torne o processo mais veloz e constante, e entre esses procedimentos a prática de CI/CD pode ser argumentada como a mais fundamental de todas, pois ela cobre todas as partes envolvidas no processo de software e em praticamente todos os passos do processos.

Justifica-se estudar esse tema, pois o processo de software têm se tornado cada vez mais ágil e rápido, e isso se dá em parte pelo difusão da cultura *DevOps*, pregando sempre agilidade nos processos, colaboração entre equipes e inovação contínua. Outro ponto válido como justificativa é demonstrar que é possível construir um processo de software maduro e ágil, utilizando a cultura *DevOps* como base principal.

O presente trabalho de conclusão de curso (TCC), tem como objetivo principal explorar a cultura *DevOps* aplicada no desenvolvimento de software, desde a parte de codificação até a entrega do produto para o cliente final.

Diante desse contexto, o presente trabalho busca responder a seguinte questão de pesquisa:

- É possível implementar a cultura *DevOps* em um projeto de software de ponta a ponta?

1.1 OBJETIVOS GERAIS

O trabalho tem como objetivo implementar uma infraestrutura que utilize a cultura de *DevOps* em um software de gestão de restaurante.

1.2 OBJETIVOS ESPECÍFICOS

- Configurar um ambiente com testes automatizados via GitLab.
- Implementar um sistema de *Deploy* utilizando imagens Docker.
- Permitir que o código seja revisado e aprovado por terceiros.
- Configurar um servidor Nginx para servir como host da aplicação.
- implementar uma *pipeline* de CI/CD utilizando as tecnologias supracitadas.
- Simular o processo de software, desde a codificação, até a entrega final para o cliente.

1.3 MÉTODO

Para alcançar os objetivos do presente trabalho, foi efetuada a pesquisa bibliográfica, que se refere ao estudo de artigos, livros, teses e outras publicações que são comumente disponibilizadas por editoras e indexadas (WAZLAWICK, 2014).

Quanto à natureza deste trabalho, trata-se de um resumo de assunto, pois apenas trabalha e expande assuntos de trabalhos feitos e publicados, que se tornaram a motivação para a criação deste trabalho (WAZLAWICK, 2014).

Este resumo de assunto visa sistematizar uma área de conhecimento específico, mostrando sua evolução e importância.

Quanto aos seus objetivos, trata-se de uma pesquisa descritiva, pois com este trabalho busca-se obter dados mais consistentes sobre a implementação de uma cultura *DevOps* em um projeto de software, descrevendo os fatos como são. A pesquisa é feita através de um estudo de levantamento de dados para análise.

Quanto aos procedimentos técnicos, trata-se de uma pesquisa experimental, caracterizada pela manipulação e obtenção de variáveis de um ambiente, sendo no caso deste trabalho o ambiente virtual em que o software irá operar.

Nesta pesquisa experimental, as variáveis das ferramentas utilizadas no processo de *Deploy*, como o Docker e GitLab são observados, analisadas e controladas através de testes pelo pesquisador, cuja medição de controle, monitoramento leva a uma determinada conclusão através da análise dos resultados estatísticos e generalizáveis.

1.4 ESTRUTURA DA MONOGRAFIA

A estrutura do presente trabalho consiste em 4 Capítulos que tratam os seguintes assuntos:

No capítulo 1 é apresentado a introdução sobre os assuntos abordados, juntamente da justificativa, objetivos gerais, específicos e metodologia.

No capítulo 2 é feito o referencial teórico necessário para o desenvolvimento da solução do problema do trabalho. Neste capítulo serão apresentados conceitos essenciais, como *DevOps*, CI/CD, *Deploy*, Arquitetura de um sistema *web*, servidores, Nginx, *containers*, Docker e GitLab.

No Capítulo 3 é apresentado o projeto de software que será utilizado como estudo de caso para a implementação da cultura de *DevOps*, este que será um sistema de gestão de restaurantes de nome fictício *Baratie.*

No Capítulo 4 são apresentados problemas encontrados durante o desenvolvimento do trabalho e possíveis soluções para os mesmos.

No Capítulo 5 são apresentadas as conclusões obtidas através do desenvolvimento do trabalho prático e teórico, e as sugestões de trabalhos futuros para quem for dar continuidade para o desenvolvimento do trabalho.

2 REFERENCIAL TEÓRICO

A fim de fornecer um embasamento adequado para a compreensão dos objetivos desta monografia, é fundamental abordar alguns conceitos básicos que envolvem as temáticas centrais deste trabalho, sendo estes: o que é *DevOps*; CI/CD; *Deploy*; Arquitetura de um sistema *Web*; Servidores; Nginx; *Containers*; Docker e GitLab. Os próximos tópicos tem como objetivo detalhar e esclarecer a estrutura e o funcionamento desse conceito, culminando no objetivo principal a ser alcançado neste trabalho.

2.1. *DEVOPS*

DevOps não têm uma descrição definida, porém uma boa parte da comunidade responsável pela difusão a define como sendo:

> [...] Uma filosofia, que prioriza pessoas no lugar de processos, e processos no lugar de ferramentas. *DevOps* constrói uma cultura de confiança, colaboração e melhora contínua. Como uma cultura, ele enxerga o processo de desenvolvimento de uma forma holística, levando em conta todos os envolvidos, sejam eles: desenvolvedores, *tester*, equipe de operação, segurança, e engenheiros de infraestrutura. *DevOps* não coloca nenhum desses grupos acima de outros, nem ranqueia a importância de seu trabalho. Ao invés disso, uma companhia *DevOps* trata o time de desenvolvimento com a mesma urgência que se trata a garantia de que o usuário tenha a melhor experiência possível (Freeman, 2019, p. 8).

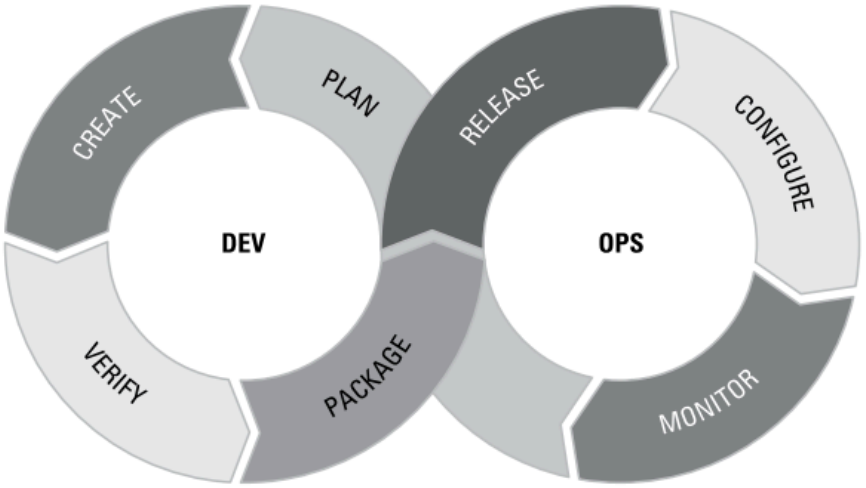
Uma das principais características do *DevOps* é a automatização. As ferramentas de automatização são utilizadas para automatizar tarefas repetitivas e manuais, como a compilação do código, testes, implantação e monitoramento. Essas ferramentas auxiliam na redução de erros, aumentando a eficiência e

aceleração do tempo de entrega (VALENTE, 2023). Essa característica se dá pela adoção da prática de CI/CD em *DevOps*.

DevOps também promove a escalabilidade e resiliência de software. As equipes de desenvolvimento e segurança precisam trabalhar juntas para garantir que o software possa lidar com um grande volume de tráfego e se recuperar rapidamente de falhas (VALENTE, 2023).

A Figura 1 ilustra o ciclo de vida de desenvolvimento que é associado em *DevOps.*

Figura 1 - Conjuntos de Ferramentas do *DevOps*.



Fonte: (FREEMAN, 2019).

Segundo Freeman (2019), as principais etapas de um ciclo de *DevOps* são:

1. Planejar

   É onde as melhorias, mudanças ou correções de bugs são preparadas para o restante do processo. Em um ambiente ágil, é nessa fase que os sprints são configurados e as tarefas são atribuídas aos desenvolvedores para trabalhar na liberação de um novo conjunto de alterações no aplicativo.

2. Criar/Codificar

Depois que o novo conjunto de mudanças for acordado e o trabalho atribuído, é hora de os desenvolvedores começarem a escrever seu código. À medida que os recursos são concluídos e configurados para teste, os cartões no quadro de trabalho movem-se lentamente do estado "A Fazer" para "Concluído".

3. Verificar

Após o Código ser desenvolvido, é necessário que ele seja minimamente verificado. Ele deve passar por uma bateria de testes automatizados, para garantir que o que feito não tenha nenhum impacto indesejado, além disso ele também deve ser verificado por algum membro da equipe, para garantir que ele siga os melhores padrões.

4. Empacotar

Uma vez que o código foi revisado pela equipe e aprovado, a fase de compilação começa utilizando ferramentas de automação para construir os artefatos de lançamento. Os artefatos são uma cópia isolada do código compilado ou de outra forma "configurado" que pode ser implantado nos ambientes necessários para teste ou produção. Um pequeno subconjunto de testes unitários de execução rápida pode ser executado aqui para validar rapidamente se a compilação será aprovada ou falhará antes de passar para testes mais rigorosos.

5. Lançar

A fase de lançamento atua como um marco em todo o processo *DevOps* e é onde a parte "Ops" do processo começa. Neste ponto, todos os testes devem ter sido aprovados e a compilação pode ser configurada com segurança para implantação em sistemas de produção.

6. Configurar

> A fase de operação/configurar determina que a equipe de operações garanta que os componentes de infraestrutura permaneçam online e cresçam de acordo com a demanda, conforme necessário. Isso pode significar aumentar ou diminuir os recursos conforme necessário. Os ambientes em nuvem simplificam isso fornecendo métodos para monitorar a carga no seu sistema e girar automaticamente novos nós para equilibrar a carga adequadamente.

7. Monitorar

> O monitoramento é crucial para qualquer sistema para garantir que as coisas estejam se movendo na direção certa. A coleta de dados analíticos pode ajudar a fornecer uma visão mais holística do que é necessário, além do feedback fornecido diretamente pelos usuários, e a realização de retrospectivas sobre os processos seguidos pela equipe como um todo pode ajudar a simplificar os procedimentos no futuro, levando a um fluxo mais suave em geral.

2.2. CI/CD

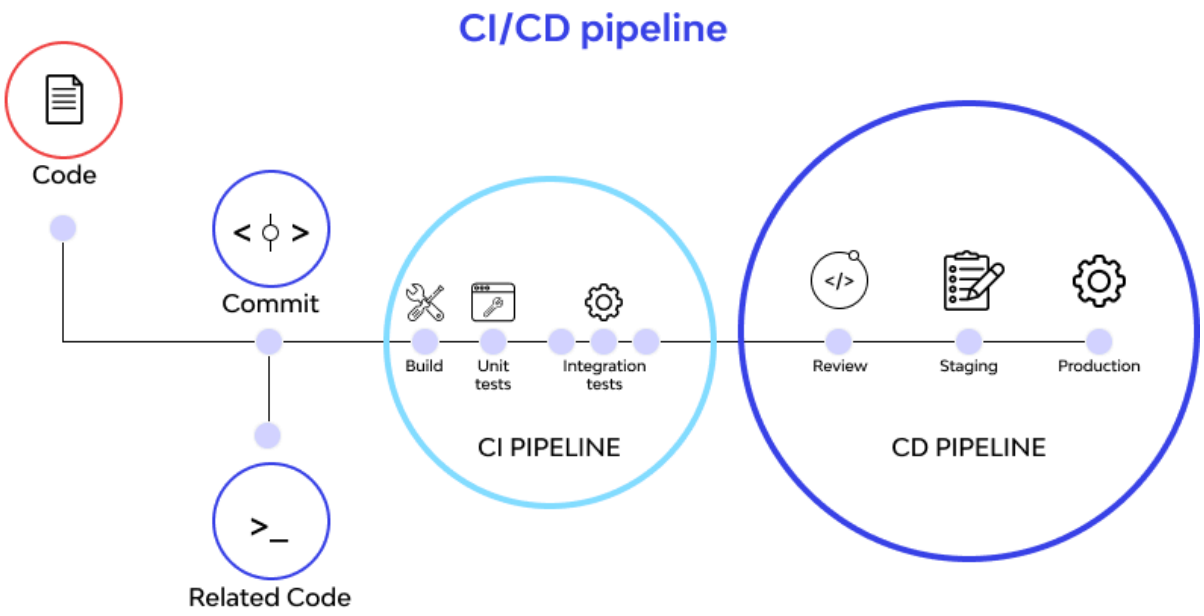
Antes da metodologia ágil emergir, as equipes de desenvolvimento implementavam alguma funcionalidade atribuída, e depois se esqueciam sobre o que foi feito. O código era jogado para o time de QA, que jogava novamente para o desenvolvimento por questões de bugs ou moviam a frente para o time de operação. Essa por sua vez era responsável por implantar e dar manutenção no código em produção (FREEMAN, 2019).

Esse processo era grosseiro, e causava bastante conflito, pois os times não conversavam e tinham pouco ou nenhum conhecimento sobre como os demais times operam, incluindo seus processos e motivações (FREEMAN, 2019).

CI/CD, que se traduz para algo como integração contínua e entrega contínua, mira em quebrar as paredes que historicamente dividem os times envolvidos em um projeto de software, buscando instituir um processo de software mais suave (FREEMAN, 2019).

O CI/CD oferece muitos benefícios, entretanto, o processo de construir uma *pipeline* (série de etapas interconectadas que permitem que o software seja desenvolvido a partir do código-fonte até um executável em funcionamento) pode ser demorado, além de exigir o apoio da equipe e da liderança executiva (FREEMAN, 2019). Alguns benefícios ao se utilizar CI/CD são a adoção de testes automatizados completos, um ciclo veloz de feedback, conflitos interpessoais são diminuídos e um processo de *Deploy* seguro (FREEMAN, 2019). A Figura 2 descreve visualmente como é uma *pipeline* CI/CD.

Figura 2 - *Pipeline* do CI/CD



Fonte: https://blog.xpeducacao.com.br/pipeline-ci-cd/

A prática de CI (*Continuous Integration*) têm haver com a forma de gerência da base de código, uma equipe que pratica CI faz merge das alterações de código

para a *branch master* ou de desenvolvimento assim que possível. CI normalmente usa ferramentas de integração para validar a *build* e executar testes automatizados no código novo (FREEMAN, 2019). O processo de CI permite que os desenvolvedores de um time trabalhem com a mesma base de código com a garantia que as mudanças sejam mínimas e que grandes conflitos de *merge* sejam evitados.

CD (*Continuous Delivery/Continuous Deploy*) é um termo que tem como base duas práticas, a entrega contínua e o *Deploy* contínuo. Se tratando de entrega contínua, é um passo seguinte do CI, no qual os desenvolvedores tratam cada mudança no código como entregável, diferente do *Deploy* contínuo, deve ser acionado por um humano, e as mudanças não devem ser entregues imediatamente para o usuário final. O processo de entrega contínua é feito da seguinte forma:

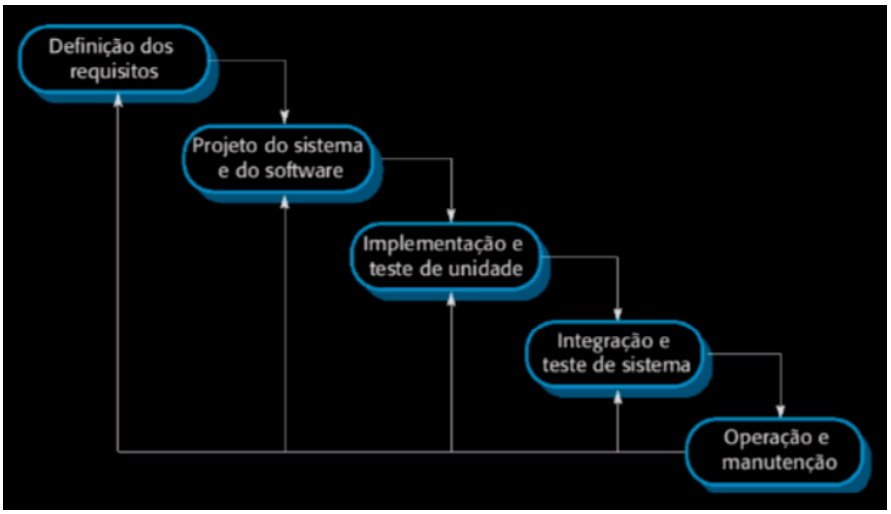
1. Desenvolvedores fazem merge e *Deploy* de seu código através de um único botão. Ao fazer frequentes pequenas iterações, a equipe garante que possíveis problemas sejam facilmente resolvidos.
2. Após o código passar dos testes automatizados e é “construído”, o time pode fazer *Deploy* em qualquer ambiente em específico, como QA, ou testes. Geralmente um par de equipe manualmente revisa o código antes que algum engenheiro responsável faça o merge do código em uma branch de release (FREEMAN, 2019).

Deploy contínuo é um avanço ainda maior da integração contínua. Cada mudança que passa pela *pipeline* de release é implantada para o cliente, ou seja, o código é implantado diretamente em produção. *Deploy* contínuo elimina a intervenção humana do processo de *Deploy* e requer um conjunto de testes completamente automatizados (FREEMAN, 2019). Para implementar o processo de *Deploy* contínuo é preciso ter a manter uma forte cultura de testes, documentar novas funcionalidades e coordenação com outros departamentos.

2.3. *DEPLOY*

No atual ano de confecção deste TCC, os principais modelos de processo de software lidam com o código em diversas etapas, do desenvolvimento até a parte em que o cliente de fato utiliza o software. No clássico modelo cascata, existem duas etapas após a implementação que lidam com o código, sendo estes a integração e teste de sistema, seguidos de uma última etapa que é a operação e manutenção (SOMMERVILLE, 2019). Nessas etapas o software é comumente utilizado em outras máquinas, para que o processo possa ser mais ágil e de certa forma até mais seguro. O software que o desenvolvedor fez precisa ir para a equipe de análise de testes, depois para a equipe de garantia de qualidade, ou QA (*Quality Assurance*), e então para o cliente final, ou seja, o código precisa ser **implantado** em outros ambientes. A Figura 3 abaixo ilustra como é o modelo cascata.

Figura 3 - Modelo Cascata.



Fonte: (SOMMERVILLE, 2019)

Deploy, que é a tradução literal de implantar, é o ato de mover o código de um ambiente para outro, ambiente esse que pode ser de testes de aceitação de usuário; garantia de qualidade; homologação e produção. um *Deploy* para produção, especialmente para cliente é a forma mais pura da palavra liberar, ou *release*, quando se trata de processo de software (FREEMAN, 2019).

2.4. ARQUITETURA DE UM SISTEMA WEB

Antes de falar efetivamente sobre *Deploy*, é preciso ter uma noção de como são as arquiteturas utilizadas em sistemas *web*, pois isso afeta diretamente o processo de *Deploy*. Primeiramente deve ser definido o que é uma arquitetura de um sistema.

Em computação, arquitetura é uma palavra difusa, que pode significar mais de uma coisa, depende muito do que o autor está tratando. Para esse TCC se encaixa melhor a definição da INTERNATIONAL ORGANIZATION FOR STANDARDIZATION (2017) como sendo: “Conceitos ou propriedades fundamentais de um sistema em seu ambiente, incorporados em seus elementos, relações e nos princípios de seu design e evolução”. Com essa definição em mente é possível explorar melhor arquiteturas aplicadas à *web*.

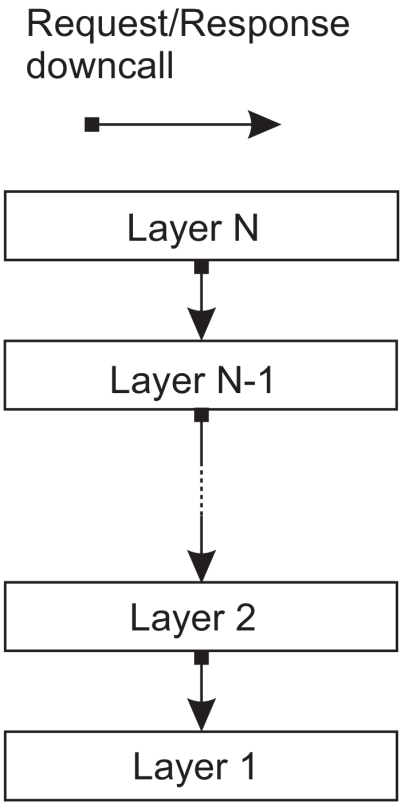
A arquitetura de sistemas *web* é por natureza distribuída, ou seja, ela é um sistema de computadores em rede, no qual os processos e recursos estão suficientemente distribuídos em vários computadores (TANENBAUM e VAN STEEN, 2023). O que permite que esse tipo de sistema exista é a troca de mensagens (ou protocolo de comunicação), que para o caso de sistemas *web* é utilizado o protocolo TCP (TANENBAUM e VAN STEEN, 2023).

Seguindo o conceito de arquitetura, agora aplicado em sistemas distribuídos, significa a forma organizacional de todos os componentes que regem o sistema. Sistemas distribuídos podem ter vários estilos arquiteturais, podem ser em camadas (que terá ênfase nesse trabalho), orientadas a serviços ou *publish-subscribe* (TANENBAUM e VAN STEEN, 2023).

Tanenbaum e Van Steen (2023) dizem que o estilo de modelo em camadas é baseado no modelo cascata de desenvolvimento de software, ou seja, os componentes são organizados em camadas, onde componente na camada N faz

uma chamada para a camada abaixo N-1 e geralmente é esperado uma resposta, assim ilustra a Figura 4.

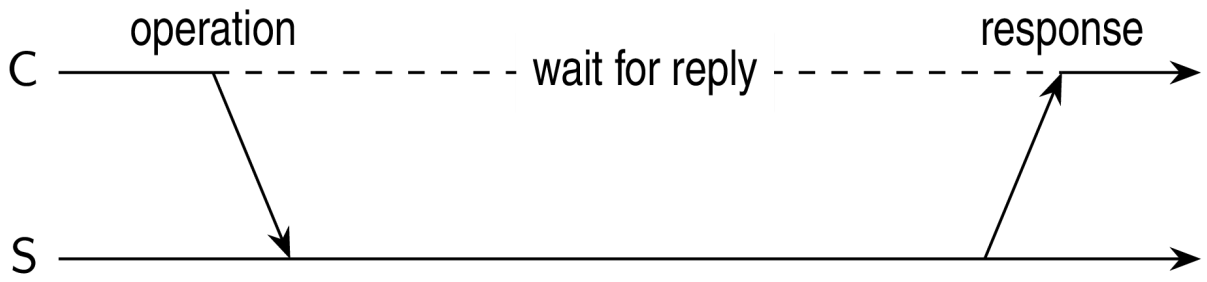
Figura 4 - Modelo em Camadas.



Fonte: (TANENBAUM e VAN STEEN, 2023).

Dentre as arquiteturas que utilizam o estilo em camadas existe a arquitetura cliente-servidor que divide a aplicação em duas partes. Uma parte (na maioria dos casos, é o cliente) se comunica com a outra parte através de um protocolo, geralmente requisitando algum tipo de dado. A outra parte verifica a requisição e se tudo estiver bem, responde com o conteúdo que foi pedido, a Figura 5 ilustra bem como essa arquitetura funciona (TANENBAUM e VAN STEEN, 2023). Segundo o Viana (2023), a arquitetura cliente-servidor é utilizada na maioria dos sistemas *web*.

Figura 5 - Arquitetura cliente-servidor.



Fonte: (TANENBAUM e VAN STEEN, 2023).

2.5 SERVIDORES

De acordo com Posey (2021), um servidor é um software ou um dispositivo que provém um serviço para outro programa de computador, também conhecido como cliente. Em um *data center* o computador físico que hospeda o software também é referenciado como um servidor, sendo que a máquina pode ser utilizada como um servidor dedicado ou pode ser utilizada para outros propósitos. Alguns servidores podem tanto fornecer um serviço enquanto consomem de outra máquina, ser cliente ou servidor não algo mutuamente excludente.

Existe uma grande variedade de tipos de servidores, segundo Paessler (2016) os principais são:

- Servidores de arquivo
    - Servidores de arquivo armazenam e distribuem arquivos. Múltiplos clientes ou usuários podem compartilhar arquivos armazenados nesse servidor. Em adição, armazenar arquivos centralmente em um servidor oferece um *backup* ou soluções de tolerância a falhas mais fáceis do que tentar prover segurança e integridade para arquivos em todos os dispositivos em uma organização.
- Servidores de aplicação

- Servidores de aplicação executam software localmente, ao invés de executar diretamente na máquina do cliente. Esse tipo de servidor comumente roda aplicações intensas que são compartilhadas por um vasto número de usuários. Utilizar esse tipo de servidor remove a necessidade de instalar e fazer manutenção em diversas máquinas que teriam que rodar uma aplicação, reduzindo apenas para uma.
- Servidores de aplicação possuem alguns subtipos, sendo estes:
  - Servidores DNS
    - Os servidores DNS (Sistema de Nomes de Domínio) são servidores de aplicativos que fornecem resolução de nomes para computadores clientes, convertendo nomes facilmente compreendidos por humanos em endereços IP legíveis por máquinas. O sistema DNS é um banco de dados amplamente distribuído de nomes e outros servidores DNS, cada um dos quais pode ser usado para solicitar um nome de computador desconhecido. Quando um cliente precisa do endereço de um sistema, ele envia uma solicitação DNS com o nome do recurso desejado para um servidor DNS. O servidor DNS responde com o endereço IP necessário a partir de sua tabela de nomes.
  - Servidores de *e-mail*
    - Servidores de *e-mail* são um tipo muito comum de servidor de aplicação. Os servidores de *e-mail* recebem *e-mails* enviados a um usuário e os armazenam até que sejam solicitados por um cliente em nome deste usuário.
    - Ter um servidor de *e-mail* permite que uma única máquina seja devidamente configurada e conectada à rede o tempo todo. Está pronta para enviar mensagens, em vez de exigir que cada máquina cliente tenha seu próprio subsistema de *e-mail* em execução continuamente.

- Servidores *Web*
  - Um servidor *web* é um tipo especial de servidor de aplicativos que hospeda programas e dados solicitados por usuários pela Internet ou por uma intranet. Servidores *web* respondem a solicitações de navegadores em execução em computadores clientes para páginas da *web* ou outros serviços baseados na *web*. Servidores *web* comuns incluem servidores Apache, servidores Microsoft Internet Information Services (IIS) e servidores Nginx.
- Servidores de Bancos de Dados
  - Bancos de dados precisam estar acessíveis para vários clientes a qualquer momento e podem exigir quantidades extraordinárias de espaço em disco. Ambas essas necessidades são bem atendidas ao localizar tais bancos de dados em servidores. Servidores de banco de dados executam aplicativos de banco de dados e respondem a inúmeras solicitações de clientes. Aplicações comuns de servidores de banco de dados incluem Oracle, Microsoft SQL Server, DB2 e Informix.
- Servidor *Proxy*
  - Um servidor *proxy* atua como intermediário entre um cliente e um servidor. Frequentemente usado para isolar clientes ou servidores por motivos de segurança, um servidor *proxy* recebe a solicitação do cliente. Em vez de responder diretamente ao cliente, ele encaminha a solicitação para outro servidor ou processo. O servidor *proxy* recebe a resposta do segundo servidor e, em seguida, responde ao cliente original como se estivesse respondendo por conta própria. Dessa forma, nem o

cliente nem o servidor de resposta precisam se conectar diretamente um ao outro.

2.6 NGINX

O Nginx, como foi citado no tópico anterior, é uma opção de servidor *web*. Ele é um servidor *web* e *proxy* reverso de alta performance, gratuito e de código aberto. Ele é conhecido por sua alta performance, estabilidade, vastos recursos, configuração simples e baixo consumo de recursos (SYSOEV, 2023).

A motivação para sua criação foi resolver o problema C10K, este que, brevemente falando, é um problema de otimização de *sockets* para lidar com um alto número de clientes simultaneamente. O nome C10K faz referência a 10 mil conexões concorrentes, que era o número de conexões que o servidor não conseguia lidar, na época que o termo foi criado por Dan Kegel, em 1999. Muito do causa desse problema era relacionado ao hardware da época e a arquitetura utilizada nos servidores, onde cada conexão era controlada por uma thread ou processo próprio (C10K PROBLEM…,2023). Então conforme o número de conexões crescia, a sobrecarga com cada thread/processo também escalava. Até o corrente ano de confecção deste trabalho, os processadores utilizados em servidores podem facilmente passar de 64 núcleos e 128 threads, o que torna esse problema mais maleável, porém em 2001, os servidores utilizados em sistemas *web* eram *single core*, limitando a execução em apenas uma *thread* por vez com isso a troca de contexto entre as *threads* acarretava em sobrecarga. Quanto mais threads eram adicionadas, a CPU gasta uma quantidade crescente de tempo apenas para fazer a troca de threads do que fazendo trabalho de fato. (C10K PROBLEM…, 2023). Os resultados são uma performance e escalabilidade fracas, causa do problema C10K.

O Nginx é um servidor desenvolvido utilizando arquitetura orientada a eventos, ou seja, ele mantém uma conexão aberta e “espera” um evento (Ex: uma

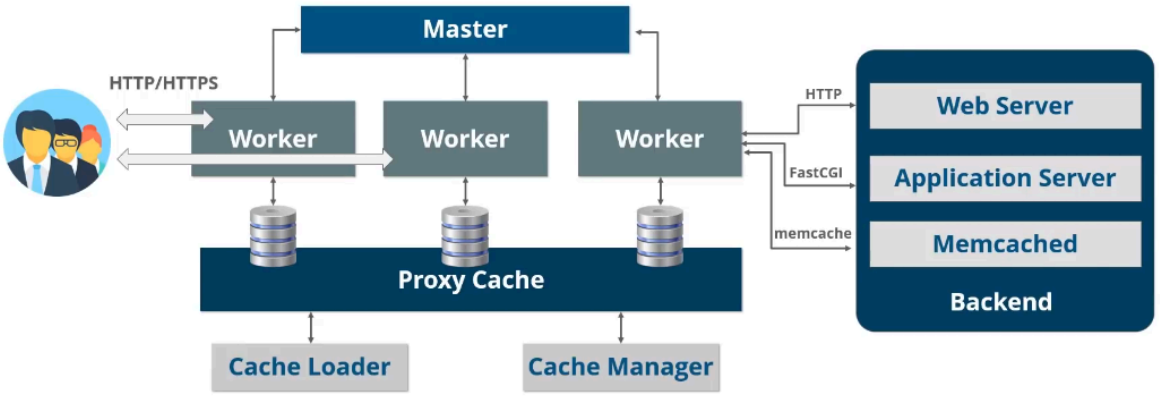
nova conexão é solicitada) para reagir, ao invés criar nova thread para cada conexão, como outros servidores fazem (C10K PROBLEM…, 2023).

Segundo Garrett (2015), a arquitetura do Nginx é composta de quatro partes principais, sendo elas:

- O processo *master* é responsável por fazer operações de privilégio, como ler configurações, fazer o *binding* de portas, criar um número pequeno de processos filhos (citados abaixo), e responder às requisições recebidas.
- O processo de *cache loader* é executado na inicialização para carregar o cache baseado em disco na memória, e em seguida encerra. Ele é agendado de forma conservadora, de modo que suas demandas de recursos são baixas.
- O processo de *cache manager* é executado periodicamente e remove entradas dos caches em disco para mantê-los dentro do tamanho configurado.
- Os processos *worker* fazem todo o trabalho, eles lidam com as conexões de rede, a leitura e escrita do conteúdo em disco e a comunicação com os servidores *upstream*. Quando um *worker* termina de processar uma requisição, ele envia o resultado para o processo *master*, que por sua vez redireciona para o cliente.

A figura 6 ilustra como funciona a arquitetura do Nginx.

Figura 6 - Arquitetura do Nginx



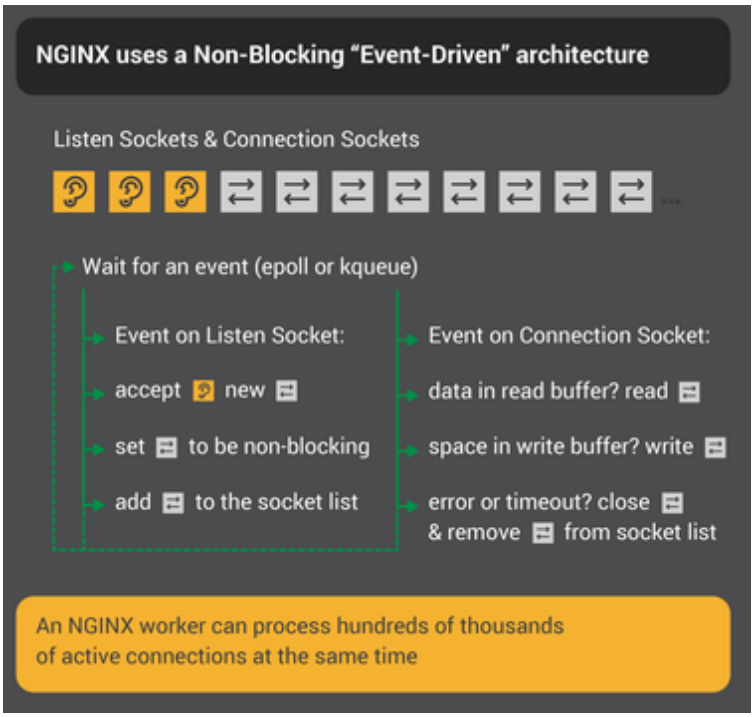
Fonte: (PREMKUMAR, 2019).

De todos esses processos, apenas os *workers* ficam ocupados, cada *worker* lida com múltiplas conexões de maneira não bloqueante, reduzindo o número de trocas de contexto. Cada *worker* é executado em uma única *thread* independente, “pegando” novas conexões e as executando. Os processos podem se comunicar usando memória compartilhada para dados de *cache* compartilhados, persistência de dados de sessão e outros recursos compartilhados (GARRETT, 2015). Cada *worker* é iniciado com as configurações do Nginx e recebe um conjunto de *sockets* de escuta do processo mestre. Eles conseguem lidar com mais de 1000 requisições por vez em uma única *thread*, essa abordagem economiza o tamanho do RAM e ROM (GARRETT, 2015). Os *workers* conseguem responder essa quantidade de requisições por conta da estratégia que é implementada neles. Essa estratégia trata os *workers* como grão-mestres de xadrez que jogam centenas de oponentes simultaneamente. Ela funciona da seguinte forma:

1. O *worker* espera eventos vindos dos *sockets* de escuta e de conexão.
2. Um evento ocorre nos *sockets* e o *worker* lida com eles:
    a. Se ocorrer um evento no *socket* de escuta o *worker* cria uma nova conexão via *socket*.
    b. Se ocorrer um evento no *socket* de conexão o *worker* prontamente responde a requisição.

Um *worker* nunca bloqueia o tráfego de rede, esperando o cliente responder, quando ele executa sua “jogada”, o *worker* parte para outra conexão que está aguardando resposta, ou cria uma nova conexão caso um cliente novo faça uma solicitação. Essa abordagem ilustrada na Figura 7, é a grande responsável por fazer o Nginx ser uma ferramenta que aguenta tanto processamento simultâneo (GARRETT, 2015).

Figura 7 - Estratégia orientada a eventos dos *workers*



Fonte: (GARRETT, 2015).

2.7. CONTAINERS

Segundo Google (2023), *containers* são pacotes de software que contém todos os elementos necessários para serem executados em qualquer ambiente. Dessa forma, os *containers* virtualizam o sistema operacional e são executados em qualquer lugar, desde um data center privado até a nuvem pública ou até mesmo no laptop de um desenvolvedor (GOOGLE, 2023). Alguns benefícios de se utilizar *containers* são:

- Separação de Responsabilidade
    - A conteinerização oferece uma separação clara de responsabilidades, porque os desenvolvedores focam na lógica e nas dependências da aplicação, enquanto as equipes de operações de TI podem focar na implantação e no gerenciamento em vez de nos detalhes da aplicação

e no gerenciamento em vez de nos detalhes do aplicativo, como versões de software e configurações específicas.

- Portabilidade de cargas do trabalho
    - Os *containers* podem ser executados em praticamente qualquer lugar, facilitando muito o desenvolvimento e a implantação nos sistemas operacionais Linux, Windows e Mac, em máquinas virtuais ou em servidores físicos, na máquina de um desenvolvedor ou em data centers no local e, obviamente, na nuvem pública.
- Isolamento de aplicativos
    - Os *containers* virtualizam recursos de CPU, memória, armazenamento e rede no nível do sistema operacional, oferecendo aos desenvolvedores uma visualização do SO isolado logicamente de outros aplicativos (GOOGLE, 2023).

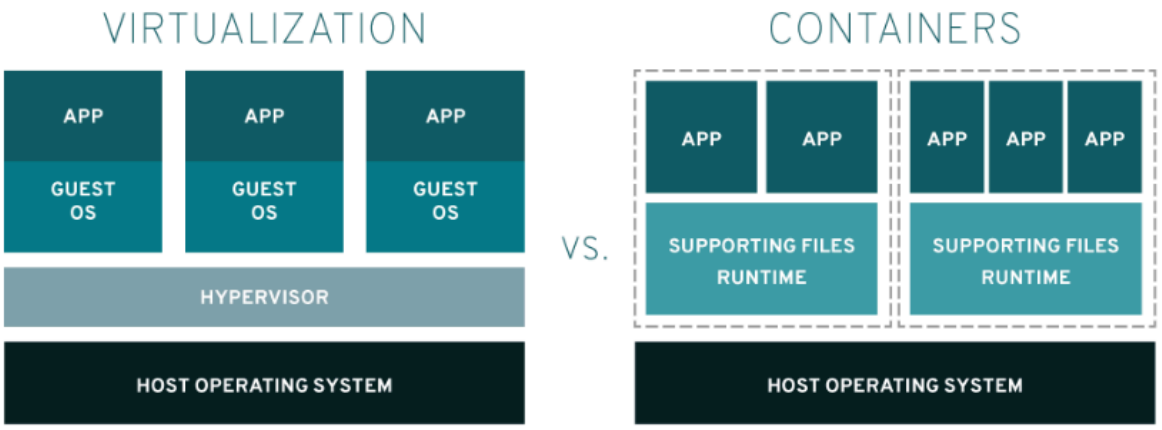
O papel que os *containers* têm em projetos de software é parecido com o papel que as VMs (Virtual Machines). Mas um *container* têm diferenças substanciais de uma VM, algumas dessas diferenças são:

- Virtualização faz com que os sistemas operacionais virtualizados executem simultaneamente em um único sistema de hardware.
- *Containers* compartilham o mesmo sistema *kernel* e isola os processos de aplicação do resto do sistema: Exemplo: Sistemas Linux ARM executam *containers* Linux ARM, Sistemas Linux x86 executam *containers* Linux x86. *Containers* Linux são extremamente portáteis, porém, eles devem ser compatíveis com o sistema base (REDHAT, 2022).

Por não se tratar de uma virtualização, *containers* não utilizam *Hypervisor* para emular o *hardware*, que é algo que permite que múltiplos sistemas operacionais sejam executados lado a lado. Esse tipo de operação não é tão leve quanto usar *containers* (REDHAT, 2022). Quando se têm recursos limitados em um projeto de software, é preciso ter aplicações leves que possam ser implantadas intensamente. *Containers* Linux executam nativamente no sistema operacional, compartilhando-o

entre todos os *containers* do SO, assim as aplicações e serviços continuam leves e executando rapidamente em paralelo (REDHAT, 2022). A Figura 8 ilustra as diferenças entre virtualização e *containers*.

Figura 8 - Diferenças entre virtualização e *containers*



Fonte: https://www.redhat.com/en/topics/containers/whats-a-linux-container

2.8. DOCKER

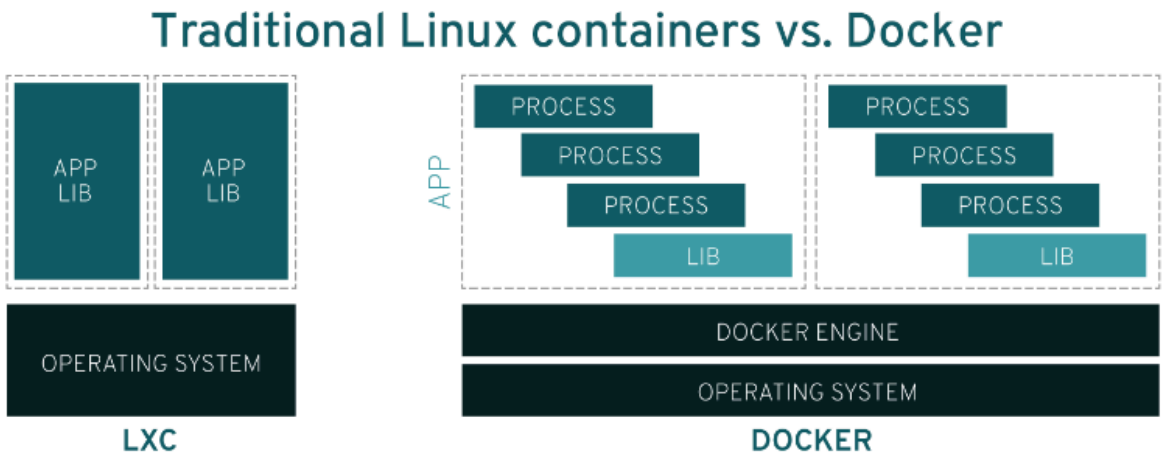
O *Docker* é uma tecnologia de *container* que tem se tornado essencial em projetos de software. Ela permite que os desenvolvedores criem, empacotem e distribuam aplicativos de maneira mais rápida e eficiente do que com as estruturas tradicionais de infraestrutura (VALENTE, 2023).

Uma das principais vantagens de se utilizar Docker é a capacidade de criar *containers* independentes e isolados que podem ser executados em qualquer ambiente, independentemente do sistema operacional ou das configurações da infraestrutura. isso significa que os desenvolvedores podem criar aplicativos uma vez e depois implantá-los em qualquer lugar, sem precisar se preocupar com incompatibilidades ou erros de configuração (VALENTE, 2023).

O Docker utiliza o kernel do Linux em seu desenvolvimento, além de outras funcionalidades como *cGroups* e *namespaces*, conseguindo assim fazer o

isolamento de processos de um *container*. Isso faz com que o Docker por padrão seja uma ferramenta exclusiva da plataforma Linux. Ele funciona de uma forma parecida com o LXC (*Linux Container*), que por sua vez também é uma tecnologia originária do Linux para “resolver” o problema de virtualização. Porém, diferente do LXC, o Docker possui algumas particularidades de implementação que faz com que ele seja uma ferramenta mais fácil de se manusear. O LXC era utilizado como uma virtualização leve, mas não oferecia uma boa experiência para os usuários e desenvolvedores, isso se dava pois, diferente do Docker, o LXC não facilita o processo de construção de *containers* e não possui o envio de versão de imagens. Outra diferença na implementação das duas tecnologias é que o LXC utiliza um sistema init que é capaz de gerenciar vários processos, ou seja, aplicações inteiras são executadas como uma, já o Docker incentiva a segregação de aplicações em processos separados, ou seja, os cada processo possui suas próprias bibliotecas isoladas e o *Docker Engine* faz a gerência dos *containers* segregados, poupando assim a kernel (REDHAT, 2022). A Figura 9 ilustra essas diferenças.

Figura 9 - Diferença do Docker para o LXC



Fonte: https://www.redhat.com/en/topics/containers/what-is-docker

Um dos principais pontos de uso de Docker é a facilidade de gerenciamento. Os desenvolvedores podem criar, empacotar e distribuir aplicações de forma veloz e fácil, utilizando ferramentas que podem estender o seu uso, como Git e o Docker Hub. Além disso, o Docker oferece recursos de gerenciamento e monitoramento

integrados, permitindo que os desenvolvedores monitorem e gerenciem seus aplicativos de maneira eficiente (VALENTE, 2023).

Outro ponto de diferencial no Docker é a forma que ele lida com modularização, se concentrando na habilidade de “desligar” um pedaço da aplicação, seja para reparo, atualização, e etc., sem interrompê-la totalmente (REDHAT, 2022).

Docker é uma tecnologia que se encaixa perfeitamente em um ecossistema *DevOps*. A ferramenta foi criada justamente para atender empresas que tinham dificuldade em mudar com agilidade alguma tecnologia, requisito do cliente ou lógica. A razão pela qual o Docker ser tão crucial para o *DevOps* são seus benefícios e cases de conteinerização de aplicações que apoiam o desenvolvimento e a “liberação” rápida. O Docker como ferramenta, permite que a equipe consiga implantar ambientes de desenvolvimento, testes e produção de modo a prover controle contínuo sobre todas as mudanças. O Docker garante que se uma uma funcionalidade funciona em um ambiente de desenvolvimento, também irá funcionar em um ambiente de produção (KODEKLOUD, 2020).

2.9. GITLAB

O GitLab além de ser uma empresa é uma ferramenta baseada em *web* para construção de software (COWELL; LOTZ; TIMBERLAKE, 2023). Ela foi criada em 2011 com um simples objetivo, facilitar e melhorar o uso de Git, sendo basicamente um repositório online para projetos de software, porém com o passar do tempo o escopo da ferramenta escalou e passou de uma ferramenta que auxilia no uso de GIt para dar suporte em todo o ciclo de desenvolvimento de software. Segundo Cowell; Lotz; Timberlake (2023), a ferramenta é dividida em 10 estágios, sendo estes:

1. Gerenciar

Criar relatórios de auditoria e de conformidade e acesso restrito aos seus recursos.

2. Planejar

   Dividir o trabalho em “nacos” trabalháveis que pode ser priorizado, e atribuir para membros do time.

3. Criar

   Commitar, revisar e aprovar edições de arquivos, sendo esses arquivos de código, configuração, informações ou afins.

4. Verificar

   Executar testes automatizados, para garantir que o software faça o que lhe é planejado.

5. Empacotar

   Empacotar o código em um formato implantável.

6. Assegurar

   Achar quaisquer vulnerabilidades de segurança nas dependências do software.

7. Liberar

   Liberar o software (fazer o *Deploy*), opcionalmente utilizando ferramentas sofisticadas, como *feature flags* e *canary Deployment*.

8. Configurar

   Configurar os ambientes em que o software será implantado.

9. Monitorar

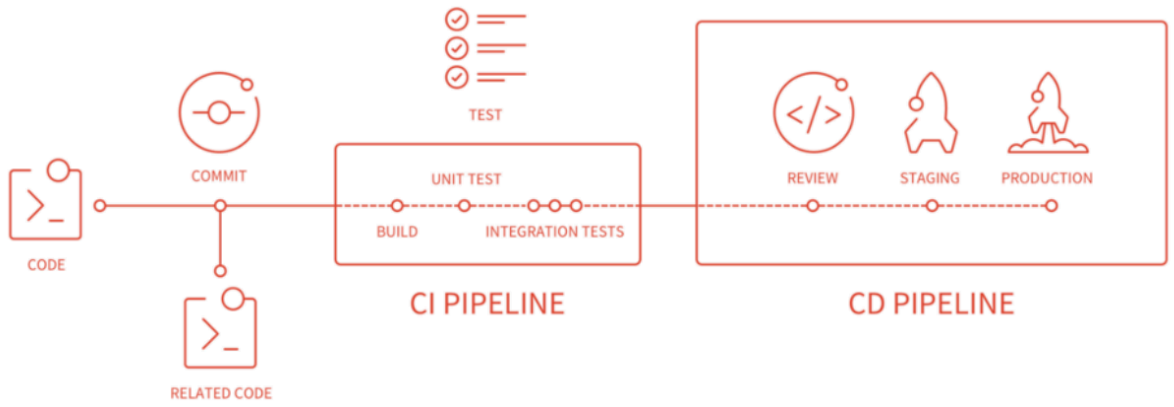
   Emitir relatórios de métricas de performance, incidentes ou erros.

10. Proteger

Detectar potenciais problemas de segurança em ambientes de implementação, como clusters de Kubernetes.

Pode-se ser notado que esses estágios batem com muitas das características de uma *pipeline* de CI/CD, exemplo disso são os estágios 3, 4 e 5 claramente fazem uma alusão a uma *pipeline* de CI envolvendo toda a parte de construção, testes unitários e de integração, por conta dessas características o GitLab acaba se tornando uma opção assertiva de ferramenta quando se precisa de uma tecnologia unificada que implanta a cultura de *DevOps*. A Figura 10 ilustra a semelhança entre o conceito de *pipeline* de CI/CD com a implementação do GitLab.

Figura 10 - *Pipeline* de CI/CD do GitLab



Fonte: https://techpelomundo.com.br/gitlab-plataforma-colaborativa

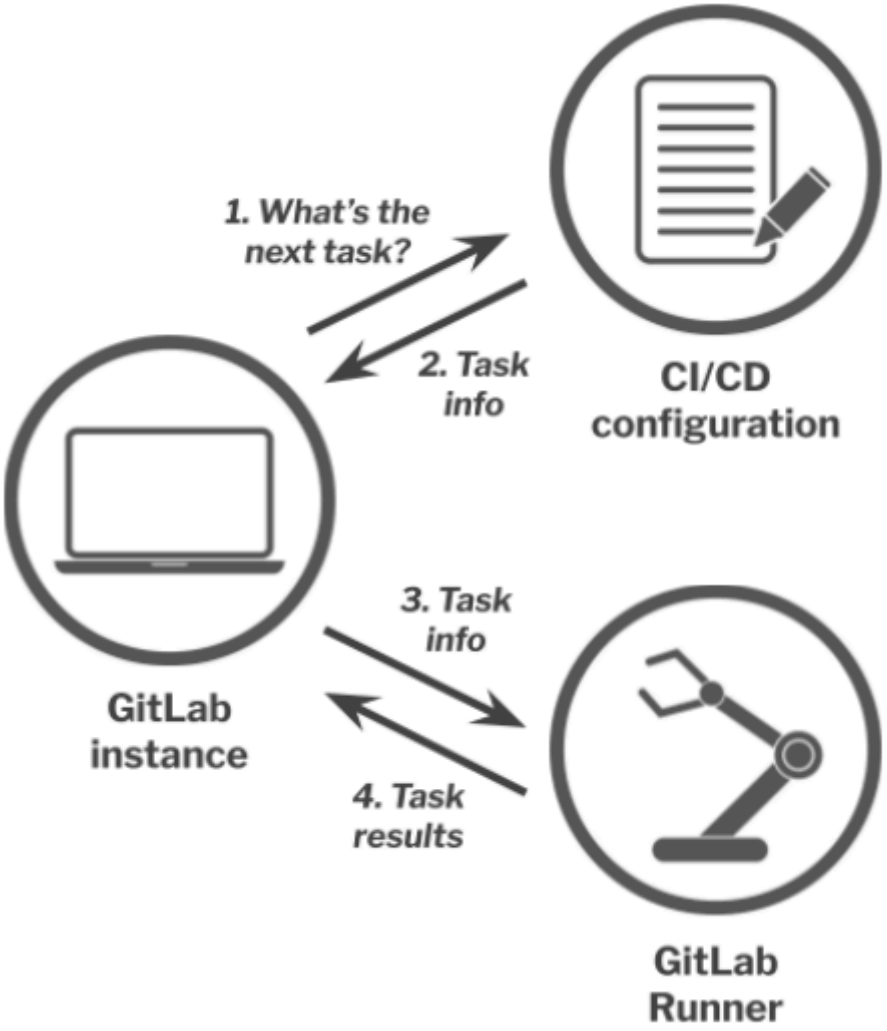
2.10 GITLAB RUNNER

Segundo Cowell; Lotz; Timberlake (2023), uma *pipeline* CI/CD é uma série de comandos que são executados por um robô, onde os comandos executam tarefas relacionadas a construção, verificação e entrega de software. GitLab Runners são os robôs que executam tais comandos.

Tecnicamente falando, o GitLab Runner é um pequeno programa que recebe comandos para executar vindos do servidor GitLab, eles são responsáveis por

converter um código de configuração de CI/CD em procedimentos reais de construção, verificação, assegurar e entregar o código. A Figura 11 abaixo ilustra esse processo.

Figura 11 - Diagrama Arquitetural da *pipeline* de CI/CD do GitLab



Fonte: Cowell; Lotz; Timberlake (2023).

Antes de Saber como o *Runner* é feito, é necessário ter em mente alguns conceitos sobre as partes de uma *pipeline*, estes são:

- *Stages*
    - Um *Stage* é uma coleção de tarefas de uma *pipeline* que estão tematicamente relacionadas. Exemplos:
        - Stage de *Build*
            - Nesse Stage estão agregadas todas as tarefas que compilam o código-fonte em um formato que seja implantável.
        - Staged de Test
            - Nesse Stage estão agregadas todas as tarefas que executam os testes automatizados, scans de qualidade de código e etc.
- *Jobs*
    - É a descrição formal das tarefas citadas no começo desta seção, cada *job* precisa ter um nome que descreve qual tarefa ele faz.
- Comandos
    - São os mesmos comandos que um humano digitaria em um terminal. Cada *job* é composto por um ou mais comandos.

Esses conceitos são de extrema importância, pois toda a configuração de uma *pipeline* é feita utilizando-os.

No GitLab a configuração CI/CD é feita através de um arquivo de extensão .yml, tipicamente nomeado como .gitlab-ci.yaml, que deve ficar salvo na raiz do projeto. Todo arquivo .gitlab-ci.yaml utiliza uma linguagem de domínio específica que possui palavras-chaves, valores e colas sintáticas. Algumas palavras-chaves definem os stages e quais *jobs* fazem parte dessas stages (COWELL; LOTZ; TIMBERLAKE, 2023).

Como dito anteriormente, o GitLab *Runner* é um programa externo ao GitLab, e seu propósito é receber os *jobs* publicados pelo GitLab e seguir as instruções definidas no arquivo .gitlab-ci.yml. Os *Runners* podem ser instalados e configurados

para trabalhar em uma grande variedade e tipos de infraestrutura, incluindo servidores *standalone*, máquinas virtuais, containers entre outros.

Segundo Cowell; Lotz; Timberlake (2023), os *Runners* se comunicam com o servidor do GitLab através de requisições HTTPS, que são assinadas por um token privado que é gerado pelo próprio GitLab. Os *Runners* também possuem um componente essencial para o seu funcionamento, chamados de Executores. Executores se referem a qual ambiente o processo *Runner* irá ser executado pelo *job*, sendo os possíveis ambientes: Docker; Shell; VirtualBox; Parallels; Kubernetes; Docker Machine SSH ou Custom. O Executor é definido no momento da criação do *Runner* e para que ele funcione em sua totalidade a máquina hospedeira deve ter o serviço do executor instalado, exemplo: Um *Runner* que utiliza um executor de Kubernetes deve ter o Kubernetes instalado na máquina hospedeira.

2.11 AWS

A AWS *(Amazon Web Services*) é um dos chamados IaaS *(Infrastructure as a Service*), traduzido literalmente como infraestrutura como serviço, é um serviço de computação em nuvem desenvolvido pela Amazon. Alguns dos serviços, projetos e recursos oferecidos são focados em tecnologias de infraestrutura como: Computação; Armazenamento e banco de dados; Machine Learning; Inteligência artificial; Data lakes; Análises; Internet das coisas, entre outros (SYOZI, 2022).

2.12 EC2

O Amazon EC2 (Elastic Compute Cloud), é o serviço de máquinas virtuais oferecido pela Amazon. Ele têm como principal característica oferecer capacidade de computação por demanda na nuvem da AWS. Uma de suas principais vantagens é a redução de custos de forma que o desenvolvimento e a implantação de aplicações seja feito com rapidez através de instâncias (AMAZON, 2024). Instâncias são VMs criadas dentro da infraestrutura da AWS, a principal característica delas é a

autonomia, onde é possível realizar qualquer tarefa administrativa dentro do sistema operacional das próprias instâncias, desde coisas simples até elementos mais avançados como instalação de programas, serviços e alterações de configurações (FONSECA, 2019).

O EC2 faz uso de outro serviço da AWS para fazer o provisionamento de rede, sendo assim, de suma importância, seu nome é VPC (Virtual Private Cloud). O VPC estabelece uma seção isolada logicamente na nuvem da AWS para a gerência de rede. Por definição de design, a nuvem da AWS é distribuída entre regiões e zonas de disponibilidade espalhadas por todo o mundo, o acesso a essa infraestrutura, de forma isolada, é possível graças às VPCs e suas subnets (MACHADO, 2021).

2.13 RDS

O RDS (Amazon Relational Database Service) é um serviço da AWS que tem como principal característica a facilidade de configuração, operação e escalabilidade de um banco de dados relacional na Nuvem AWS. Ele oferece uma capacidade econômica e redimensionável para um banco de dados relacional padrão do setor e gerencia tarefas comuns de administração de banco de dados (AMAZON, 2024).

O RDS é composto por uma instância de EC2, com o diferencial da AWS fazer a gerência das camadas abaixo do sistema operacional. Dessa maneira, o EC2 elimina parte do peso de gerenciar um servidor de banco de dados. Como o RDS é basicamente uma modificação EC2, isso faz com que ele opere através de instâncias. Uma instância RDS pode conter um ou mais bancos de dados criados pelo usuário (AMAZON, 2024).

Outra vantagem da utilização do RDS é a ressonância que ele possui com os outros serviços da AWS como o próprio EC2. É possível fazer a conexão de um banco de dados que esteja sendo executado em uma instância RDS com uma

aplicação paralela que esteja sendo executada em uma instância EC2, utilizando o VPC (AMAZON, 2024).

3 ESTUDO DE CASO

A seguir será descrito um caso de uso de uma implementação de uma infraestrutura de *Deploy* que utiliza a cultura de *DevOps*. O objetivo é demonstrar os benefícios de uma linha de produção de software que utiliza a cultura de *DevOps*.

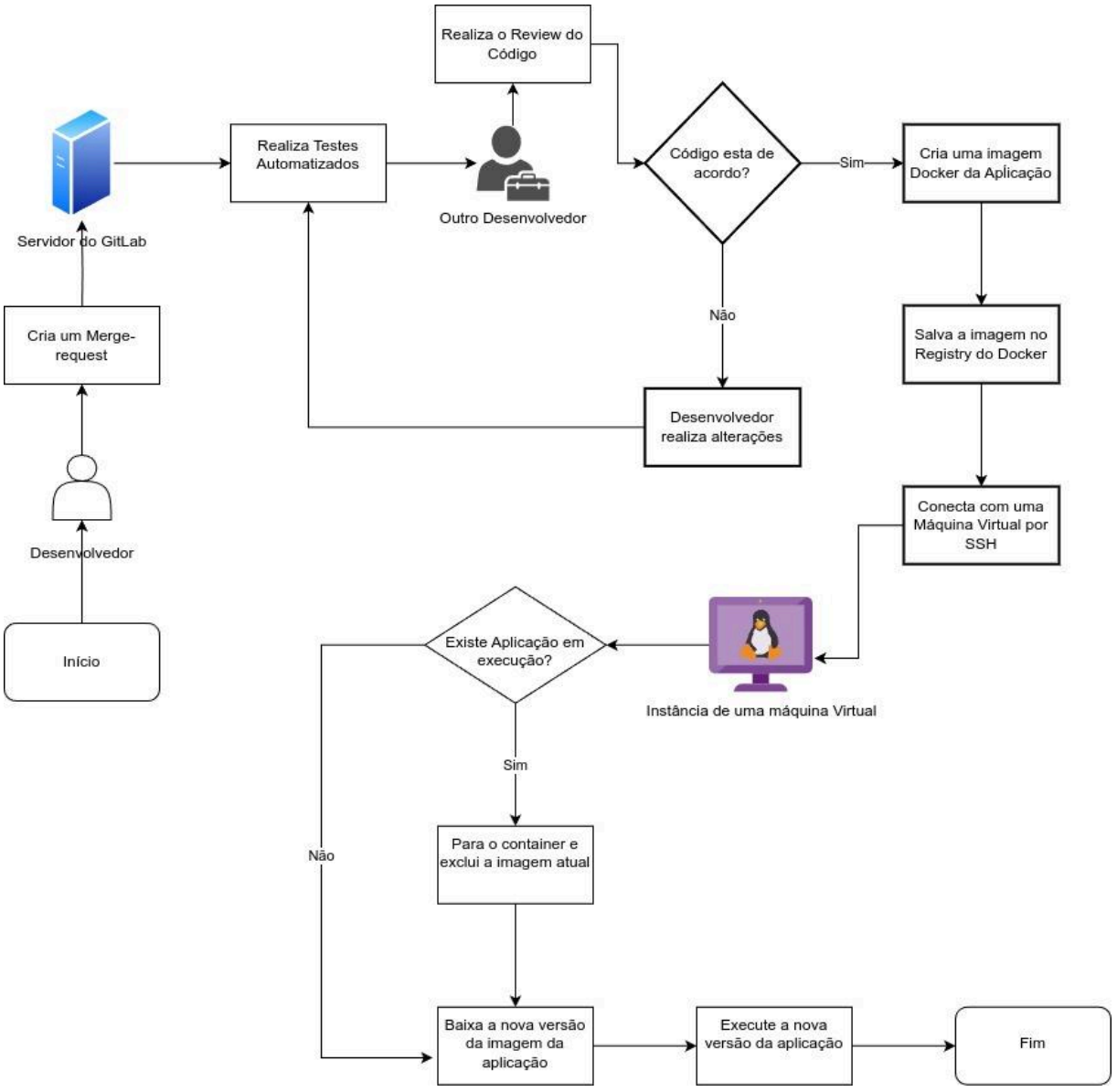
Para implementar um caso de uso, será necessário um projeto de software, tendo isso em mente foi decidido que esse projeto de software será uma aplicação *web* para gestão de restaurantes, de nome *Baratie*. Essa aplicação tem como objetivo automatizar a gestão de um restaurante em todos os seus aspectos, desde a gerência de cozinheiros, até os menus de chefs.

Sendo uma aplicação *web*, o restaurante irá seguir uma arquitetura de Cliente-Servidor. A princípio essa aplicação terá apenas o *backend*, ou seja, só irá receber requisições e retornar informações. O sistema será escrito inteiramente em Java, utilizando o *framework* Spring, e contará com um servidor Nginx para fazer o controle de carga nas requisições. Para persistência e operações com os dados será utilizado o SGBD *PostgreSQL*. O projeto será hospedado no GitLab, onde será responsável fazer a gestão de versões e hospedagem do código. O GitLab também será responsável por fazer os *pipelines* de CI/CD onde irá gerar um *container* Docker completo da aplicação, pronto para ser implantado.

3.1 ESQUEMA LÓGICO DA *PIPELINE*

A *pipeline* deverá seguir uma lógica bem definida para que o processo de Deploy funcione de forma correta, para que isso ocorra deve-se pensar no processo de software como um todo, desde a parte em que o desenvolvedor codifica, até a parte que o usuário utiliza o software. Olhando por essa ótica, foi-se desenhando o seguinte esquema lógico, que pode ser visto na Figura 12.

Figura 12 - Rotina de *Deploy*



Fonte: Elaborada pelo Autor

Tudo começa pelo desenvolvedor, ao fazer alguma alteração em código ele irá commitar o que foi feito e integrar ao código da branch master através de um merge-request. com o merge-request feito o GitLab automaticamente irá rodar uma rotina de testes automatizados para verificar se as alterações não comprometem o funcionamento do software. Após esta etapa outro desenvolvedor irá fazer o review do código, caso o código esteja de acordo com algum padrão definido no projeto e não haja alguma falha identificada por ele, o merge-request é aprovado e as alterações são integradas na branch master, caso contrário o autor dos commits é notificado para realizar alterações no código.

Após as alterações serem integradas no código, a rotina de *Deploy* é iniciada, começando novamente com os testes automatizados para verificar se o software não foi comprometido. Tendo essa parte sido concluída com o êxito a rotina de build é acionada, nesta rotina é feita uma imagem Docker da aplicação, através de um arquivo Dockerfile presente no projeto. Caso a imagem seja construída com sucesso a rotina de upload é acionada, nesta rotina a imagem recém construída é hospedada no Container Registry do GitLab.

Com as rotinas de testes, verificação de qualidade e construção de imagem do software concluídas, é iniciada a última parte da *pipeline*, o *Deploy*. Para que o *Deploy* seja efetivo e o software com as novas alterações seja acessível para o cliente, é feito uma conexão com uma instância de uma máquina virtual da AWS, isso é feito através de uma conexão SSH que é feita pelo próprio GitLab. Com a conexão estabelecida, caso a máquina virtual não esteja com as configurações (arquivo docker-compose e arquivo de configuração do Nginx) para executar a aplicação, elas são criadas e caso já tenha um container com a aplicação sendo executado, o container é removido e substituído com a nova versão do software.

3.2 ARQUITETURA DA *PIPELINE*

A *pipeline* do projeto será definida da seguinte forma: Serão 3 principais estágios, cada um deles será composto por um ou mais *jobs* que executarão uma

determinada operação necessária para o processo de *Deploy*, os ditos estágios serão:

- **Test**
    - Nesse estágio serão executados os *jobs* que irão verificar se o projeto está executável, o Gradle é o responsável por fazer a compilação de código e testes.
- **Build**
    - Nesse estágio será criada uma imagem Docker do Projeto e adicionada no *Container Registry* do GitLab.
- ***Deploy***
    - Nesse estágio os *jobs* irão se conectar com uma instância EC2 e irão baixar a imagem Docker da nova versão da aplicação e executá-la em um novo container.

Além disso foram definidas algumas regras de como a *pipeline* irá se comportar, sendo essas:

- **Execução da *Pipeline***
    - A *pipeline* só será executada quando for feito um commit diretamente na branch master ou quando ocorrer um evento de *merge-request*.
    - Essa medida se faz necessária pois a execução (pelo menos o começo) deve ser feita de forma automática.
- **Exclusão de Estágios**
    - Quando a *pipeline* for acionada por um *merge-request*, apenas o estágio de *test* será executado.
    - Isso se faz necessário pois não é viável fazer um *Deploy* a cada vez que um *merge-request* for feito, pois o código ainda precisará passar por um *review* e talvez nem seja de fato implantado no projeto de imediato.

- ***Deploy* manual**
    - O estágio de *Deploy* será executado apenas quando alguém responsável pelo projeto permitir.
    - Essa regra é necessária, pois não é visto como uma boa prática no meio *DevOps* fazer *Deploy* automaticamente, visto que sendo um procedimento delicado, se faz necessário ter uma atenção maior nesse procedimento.
- **Dependência de *Jobs***
    - Para que alguns *jobs* sejam executados é necessário que outros tenham sido executados com sucesso.
    - Isso é necessário, pois arquivos essenciais para o processo de *Deploy* são gerados em um *job* e devem ser mantidos para serem utilizados em outro posterior.

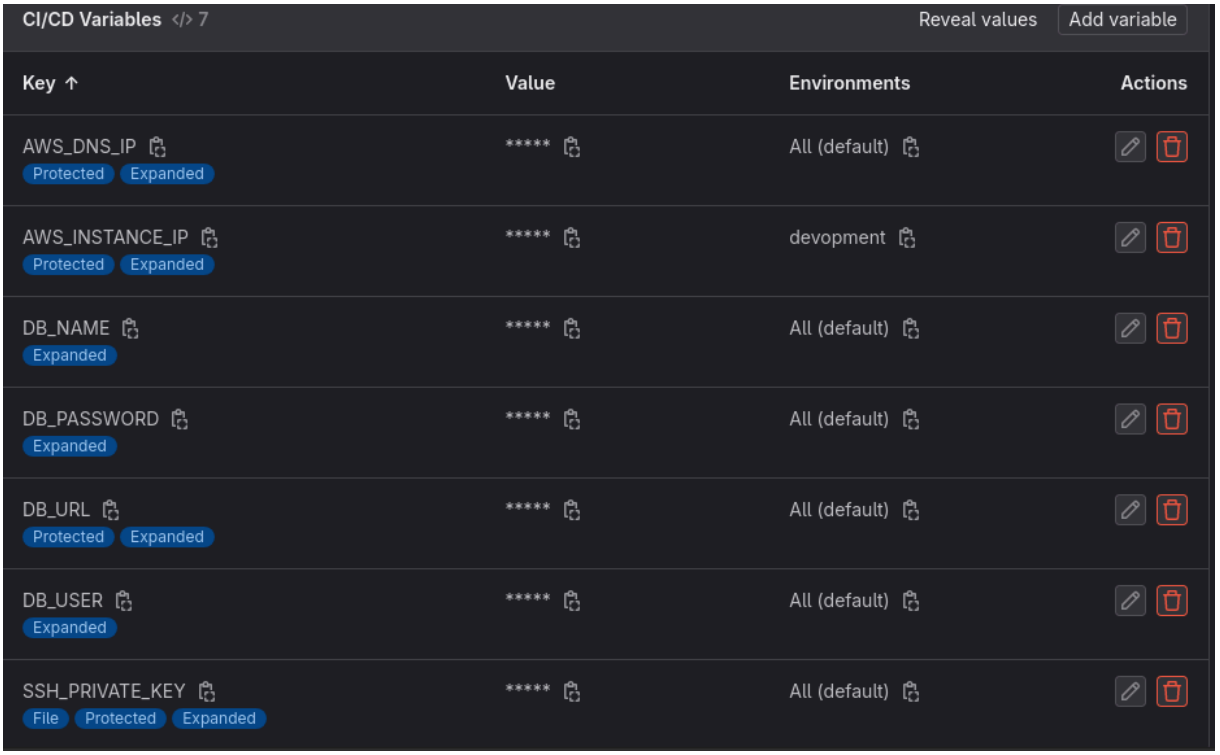
3.3 VARIÁVEIS DE AMBIENTE

O processo de *Deploy* é algo dinâmico, ou seja, sempre algo novo será executado, seja um nome de dependência, uma versão de imagem ou endereço para conexão, logo faz-se necessário que a *pipeline* esteja ajustada para trabalhar de forma dinâmica. A forma mais simples de alcançar essa característica é implementando variáveis de ambientes no processo de software. Uma variável de ambiente é um valor que pode ser diferente de acordo com qual ambiente ela está inserida, no processo de software sua utilização é vista como uma boa prática, pois ela auxilia na automação de processos, manutenção de um projeto e sua segurança. Para que o projeto consiga ser executado ele precisa de arquivos de configuração, nesses arquivos, estão definidas coisas como: versão do projeto e url para acesso ao banco, além de usuário e senha de acesso. Essas configurações são importantes em etapas seguintes do *Deploy*, logo é pertinente a presença deles nas etapas seguintes porém de uma forma dinâmica, para que o processo seja mais volátil. Além dos arquivos de configuração existem variáveis com valores sensíveis que são

imprescindíveis para o processo de *Deploy* que são externas ao projeto, como o ip para a conexão das instâncias de AWS e a chave para acesso via SSH.

O GitLab por padrão possui uma seção nas configurações apenas para gerenciamento de variáveis de ambiente, lá é possível adicionar ou remover variáveis que podem ser utilizadas na implementação da *pipeline*, a Figura 13 ilustra essa seção.

Figura 13 - Variáveis de Ambiente do Projeto



Fonte: Elaborada pelo Autor

Para que essas variáveis sejam utilizadas nos arquivos de configuração do projeto, é necessário que eles sejam preenchidos nos arquivos de forma dinâmica. Para solucionar essa questão no projeto é introduzida a utilização de arquivos template de configuração. Esses arquivos de template são uma duplicata de um arquivo de configuração comum, porém com uma extensão diferente para que o projeto não o entenda como um arquivo de configuração real. Nele todos os valores que seriam

sensíveis ou dinâmicos são alterados para uma referência a uma variável de ambiente, desta forma é possível ter uma configuração dinâmica no projeto. Com essa alteração é possível gerar os arquivos de configurações reais em tempo de execução da *pipeline*, através de rotinas na implementação da *pipeline*. Para o projeto foi utilizado o comando *envsubst*, nativo de Linux, que automaticamente cria um arquivo definitivo e com os valores definitivos daquela iteração da *pipeline*. As Figuras 14 e 15 a seguir ilustram como esse processo é feito.

Figura 14 - *Job* que cria arquivos de configuração

```
fill_env_variables:
  stage: test
  tags:
    - shell
    - ec2
    - remote
  script:
    - envsubst '$DB_URL $DB_NAME $DB_USER $DB_PASSWORD' < ./src/main/resources/application.yaml.template > ./
src/main/resources/application.yaml
  artifacts:
    paths:
      - ./src/main/resources/application.yaml
```

Fonte: Elaborada pelo Autor

Figura 15 - Trecho de *job* que cria arquivos de configuração

```
- export DC_IMAGE_NAME=$CI_REGISTRY_IMAGE
- export DC_IMAGE_TAG=$VERSION
- export NGX_PROXY_URL=$DEV_ENDPOINT
- envsubst '$DC_IMAGE_NAME $DC_IMAGE_TAG' < ./docker-compose.yml.template > ./docker-compose.yml
- envsubst '$NGX_PROXY_URL' < ./nginx/nginx.conf.template > ./nginx/nginx.conf
```

Fonte: Elaborada pelo Autor

3.4 VERSIONAMENTO DE IMAGENS

Um processo muito importante no desenvolvimento de software que acaba se estendendo para implementação de um ambiente de *Deploy* é o versionamento de software. Para o projeto o versionamento do *Deploy* é feito através das versões da imagem Docker, onde cada imagem/versão do projeto têm um número único. Para

que isso seja possível na rotina de build da imagem Docker é utilizado uma variável de ambiente customizada, que é composta pela versão da aplicação fornecida pelo Gradle concatenado com o ID da *pipeline* (valor fornecido pelo próprio GitLab através da variável *CI_PIPELINE_IID*). Dessa maneira é possível ter um controle direto das versões da aplicação, facilitando assim um possível processo de *rollback* de versão, caso a versão em execução seja problemática. A Figura 16 ilustra esse job.

Figura 16 - Build da Imagem da aplicação com versionamento

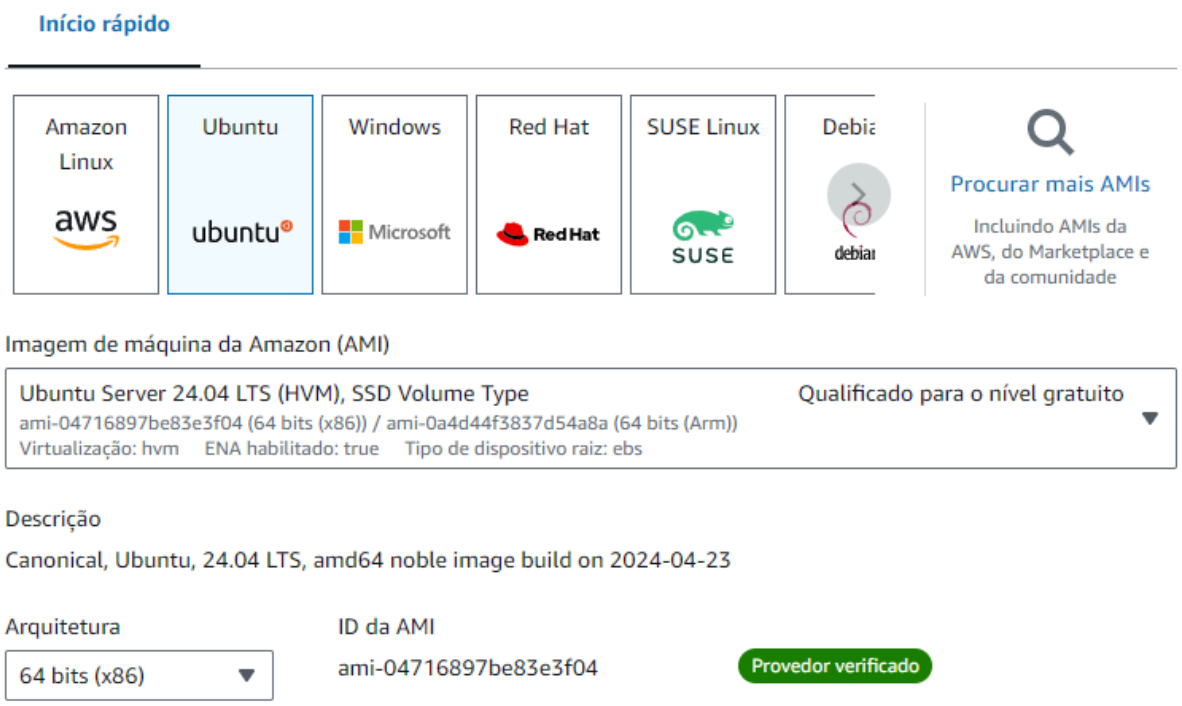
```
build_docker_image:
  stage: build
  tags:
    - ec2
    - remote
    - shell
  needs:
    - compile_project
  before_script:
    - export GRADLE_BUILD_VERSION=$(./gradlew properties --no-daemon --console=plain -q | grep "^version:" | awk '{printf $2}')
    - envsubst '$GRADLE_BUILD_VERSION' < ./Dockerfile.template > ./Dockerfile
    - export VERSION=$GRADLE_BUILD_VERSION.$CI_PIPELINE_IID
    - echo $VERSION > version-file.txt
  script:
    - docker build -t $CI_REGISTRY_IMAGE:$VERSION .
  artifacts:
    paths:
      - build/libs/*.jar
      - Dockerfile
      - version-file.txt
  rules:
  - if: $CI_COMMIT_BRANCH == "master"
    when: always
  - when: never
```

Fonte: Elaborada pelo Autor

3.5 CONFIGURAÇÃO DAS INSTÂNCIAS AWS

Foram definidas duas instâncias para o ambiente de Deploy da aplicação. Uma instância ficará responsável por executar os *runners* e o outro ficará responsável por hospedar a aplicação em si. Para que isso seja possível, é necessário primeiro criar as instâncias no console da AWS, a Figura 17 ilustra este processo.

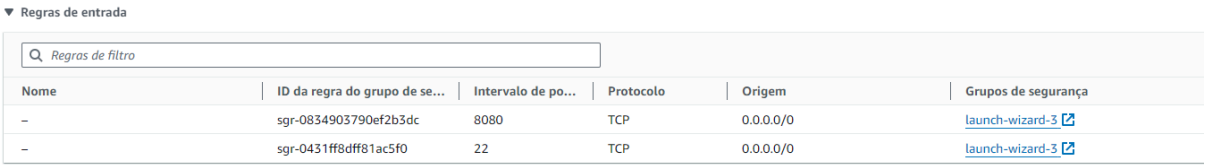
Figura 17 - Criação de Instância na AWS



Fonte: Elaborada pelo Autor

No console da AWS é possível escolher o sistema operacional que a instância irá utilizar, para o projeto foi definido que o SO das instâncias será o Ubuntu, pois ele possui muitos pacotes essenciais para o processo de *Deploy* (como o *openssh* e o *envsubst*). Com isso, serão criadas as instâncias de nome dev-server e gitlab-runner. Também deve ser definido quais serão as portas abertas para acesso externo, no caso da aplicação será a porta 8080 para instância dev-server, a Figura 18 ilustra as portas de acesso disponíveis.

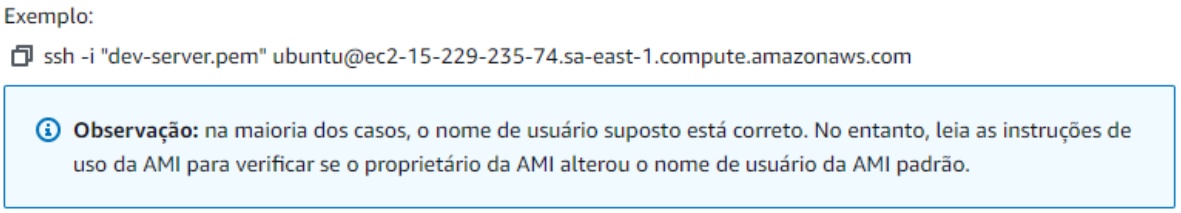
Figura 18 - Portas abertas da instância



Fonte: Elaborada pelo Autor

Na criação das instâncias também são criados arquivos .pem que servem como chaves para acessá-las via SSH, a AWS também fornece o endereço de ip da instância para fazer a conexão, conforme ilustrado na Figura 19:

Figura 19 - Comando para se conectar com a instância dev-server



Fonte: Elaborada pelo Autor

3.5 CONFIGURAÇÃO DOS *RUNNERS*

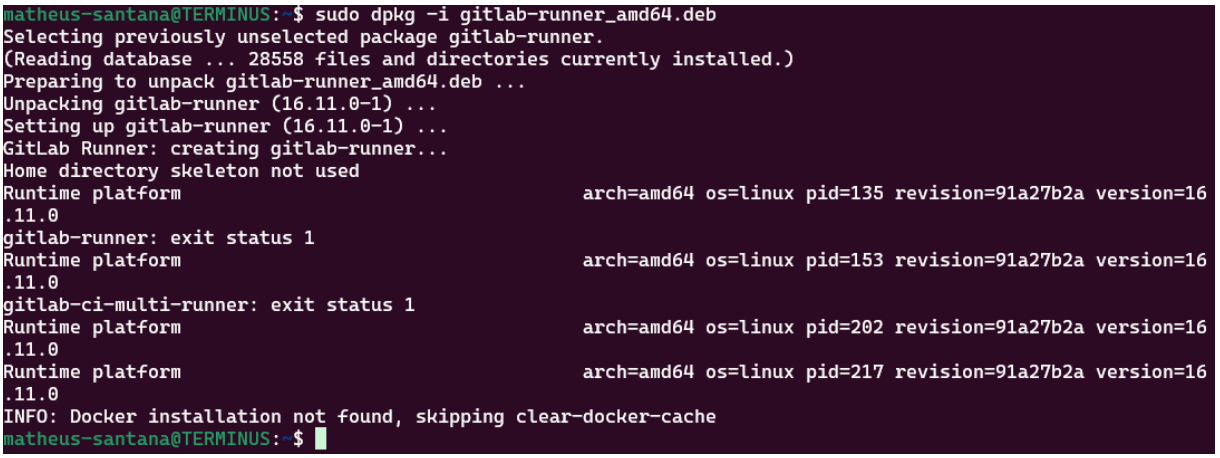
Para o projeto foi decidido que seria necessário dois *Runners*, um utilizando um executor Docker para compilar o projeto e testar o código, além de gerar os executáveis .jar, que serão utilizados na criação do Dockerfile do projeto, e outro utilizando o executor shell, que será responsável pelo restante dos *jobs* dentre eles o de preenchimento de variáveis de ambiente até o de *Deploy* em si.

Ambos os *Runners* serão executados em uma instância EC2 da AWS. Para que isso seja possível a instância deve ter o GitLab Runner e as dependências configuradas. Como o projeto é relativamente simples, as únicas dependências necessárias são o Docker e o Java, pois é necessário ter o Docker para utilizar um o

executor Docker e o executor shell irá executar alguns comandos de Java para fins de versionamento.

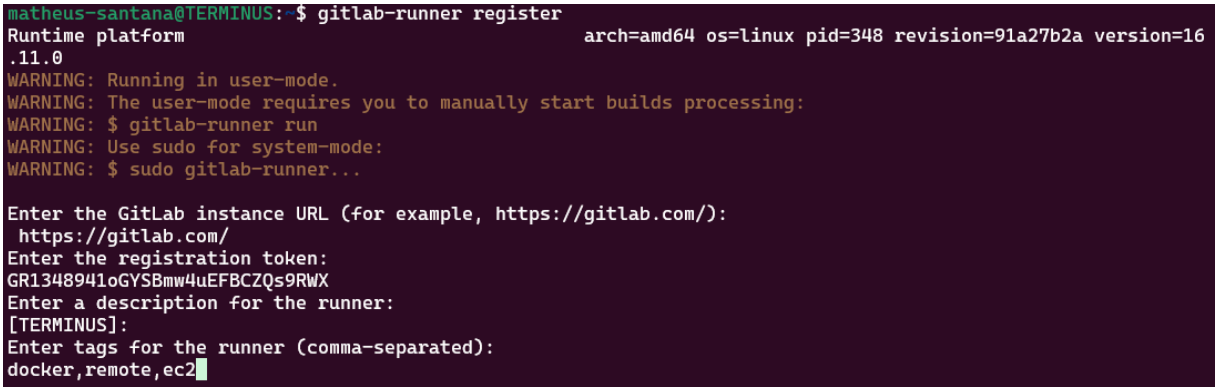
Cada *Runner* deve ter um conjunto de tags individuais, que serão utilizadas para identificar qual *Runner* específico será utilizado na execução de um *job*. Essa configuração pode ser feita pelo prompt no momento em que o *Runner* está sendo registrado, ou no próprio GitLab. As Figuras 20 a 22 ilustram esse processo de configuração de *Runners*.

Figura 20 - Instalação do *Runner* na máquina



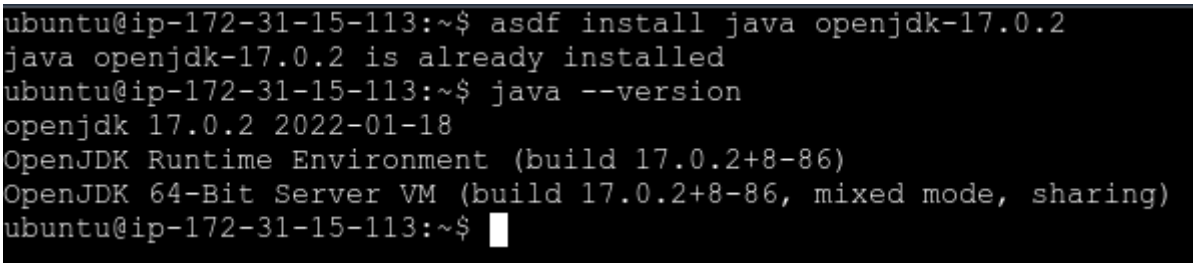
Fonte: Elaborada pelo Autor

Figura 21 - Registro de um tipo de *Runner* na máquina



Fonte: Elaborada pelo Autor

Figura 22 - Instalação do Java na máquina



Fonte: Elaborada pelo Autor

3.6 CONFIGURAÇÃO DO SERVIDOR DA APLICAÇÃO

Para que a instância de servidor funcione da maneira correta é preciso que alguns arquivos de configuração estejam presentes. Essas arquivos são:

- docker-compose.yml
    - Para que o Docker consiga executar a aplicação corretamente é necessário que tenha o arquivo de docker compose, esse arquivo possui todas as especificações necessárias para que o software seja executado em containers. Esse arquivo contém: A versão da imagem do projeto; Especificação dos volumes que serão utilizados; Versão do Nginx utilizado.
- nginx.conf
    - O software irá utilizar o Nginx como um servidor *proxy*, desta maneira, fica mais simples fazer configurações de redirecionamento, domínio e url disponível. Para que essas configurações tenham efeito, o Nginx precisa que um arquivo de configuração, onde todas essas questões são definidas, este arquivo é o nginx.conf. Como o Nginx irá ser executado como um container (para facilitar o seu gerenciamento), no arquivo de compose do Docker deve ser especificado em qual diretório da instância (volume) ficará tal arquivo de configuração.

Esses arquivos estão presentes em formato de template no repositório do GitLab, porém eles precisam estar presentes como arquivos funcionais dentro da instância da AWS, para que os containers consigam ser executados corretamente. A solução encontrada para este problema foi criar uma seção no *job* de *Deploy* que cria os arquivos definitivos com as variáveis corretas e as copia para a instância da AWS via comando scp (comando faz uma conexão via SSH e copia arquivos de uma máquina para a outra), a Figura 23 ilustra esse processo.

Figura 23 - Parte do *job* de *Deploy* que configura instância de servidor

```
script:
  - export DC_IMAGE_NAME=$CI_REGISTRY_IMAGE
  - export DC_IMAGE_TAG=$VERSION
  - export NGX_PROXY_URL=$DEV_ENDPOINT
  - envsubst '$DC_IMAGE_NAME $DC_IMAGE_TAG' < ./docker-compose.yml.template > ./docker-compose.yml
  - envsubst '$NGX_PROXY_URL' < ./nginx/nginx.conf.template > ./nginx/nginx.conf
  - scp -o StrictHostKeyChecking=no -i $SSH_PRIVATE_KEY ./docker-compose.yml  ubuntu@$DEV_SERVER_HOST:/home/ubuntu
  - ssh -o StrictHostKeyChecking=no -i $SSH_PRIVATE_KEY ubuntu@$DEV_SERVER_HOST "/bin/mkdir -p nginx"
  - scp -o StrictHostKeyChecking=no -i $SSH_PRIVATE_KEY -r ./nginx/nginx.conf ubuntu@$DEV_SERVER_HOST:/home/ubuntu/nginx/nginx.conf
```

Fonte: Elaborada pelo Autor

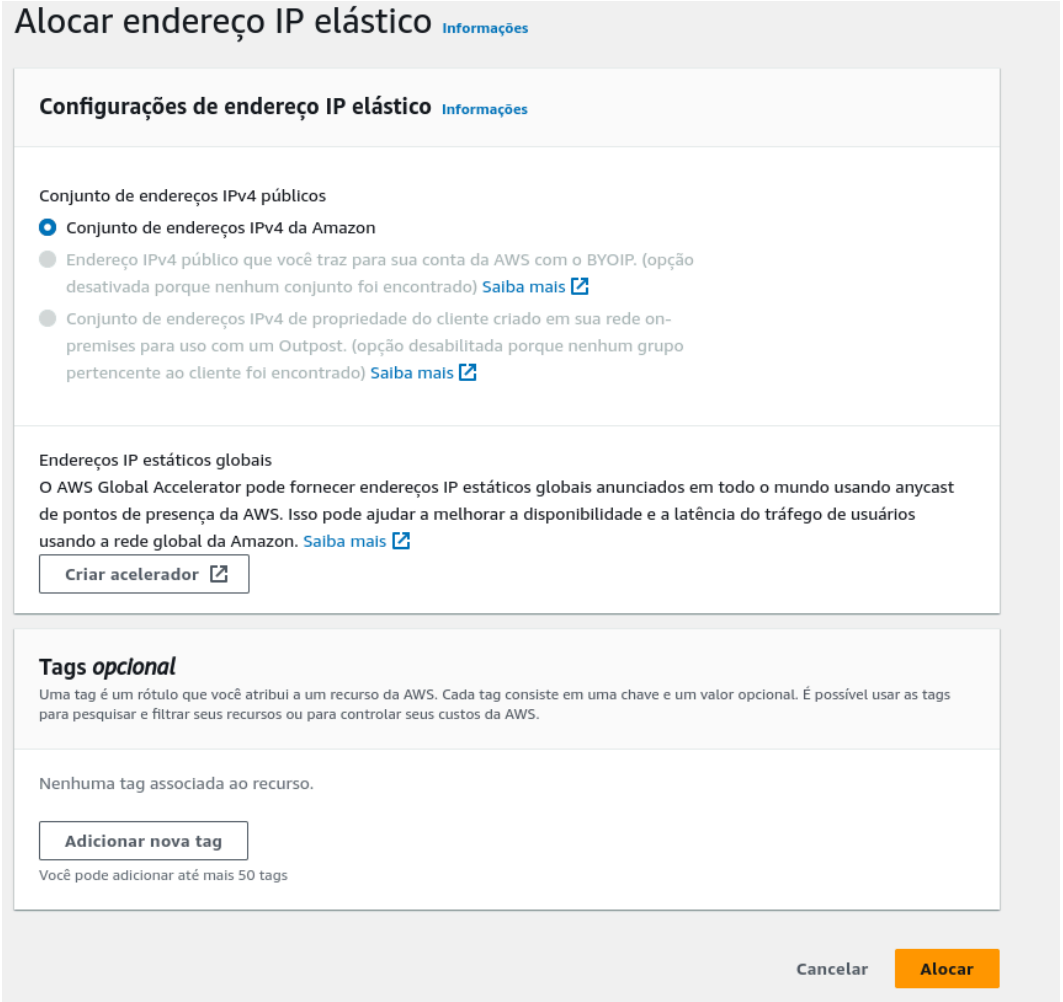
3.6.1 CONFIGURAÇÃO DE IP

Surge um problema quando se utiliza instâncias da AWS, mas especificamente na parte de conexão SSH, este problema é a mudança de IP da instância a cada vez que a mesma é reiniciada. Isso afeta diretamente o processo de *Deploy*, pois a rotina de *Deploy* utiliza o endereço IP da instância para fazer a conexão via SSH. Com esse tipo de comportamento é necessário atualizar o endereço de IP na implementação da *pipeline* todas as vezes que a instância for iniciada. Esse comportamento é prejudicial apenas para a instância de servidor, uma vez que a instância que executa os *Runners* se conecta com o GitLab diretamente pelo software do *Runner*.

Para solucionar essa situação e reduzir a dependência de ajustes manuais, a AWS oferece uma funcionalidade de IP elástico, que funciona como um IP fixo. Esse

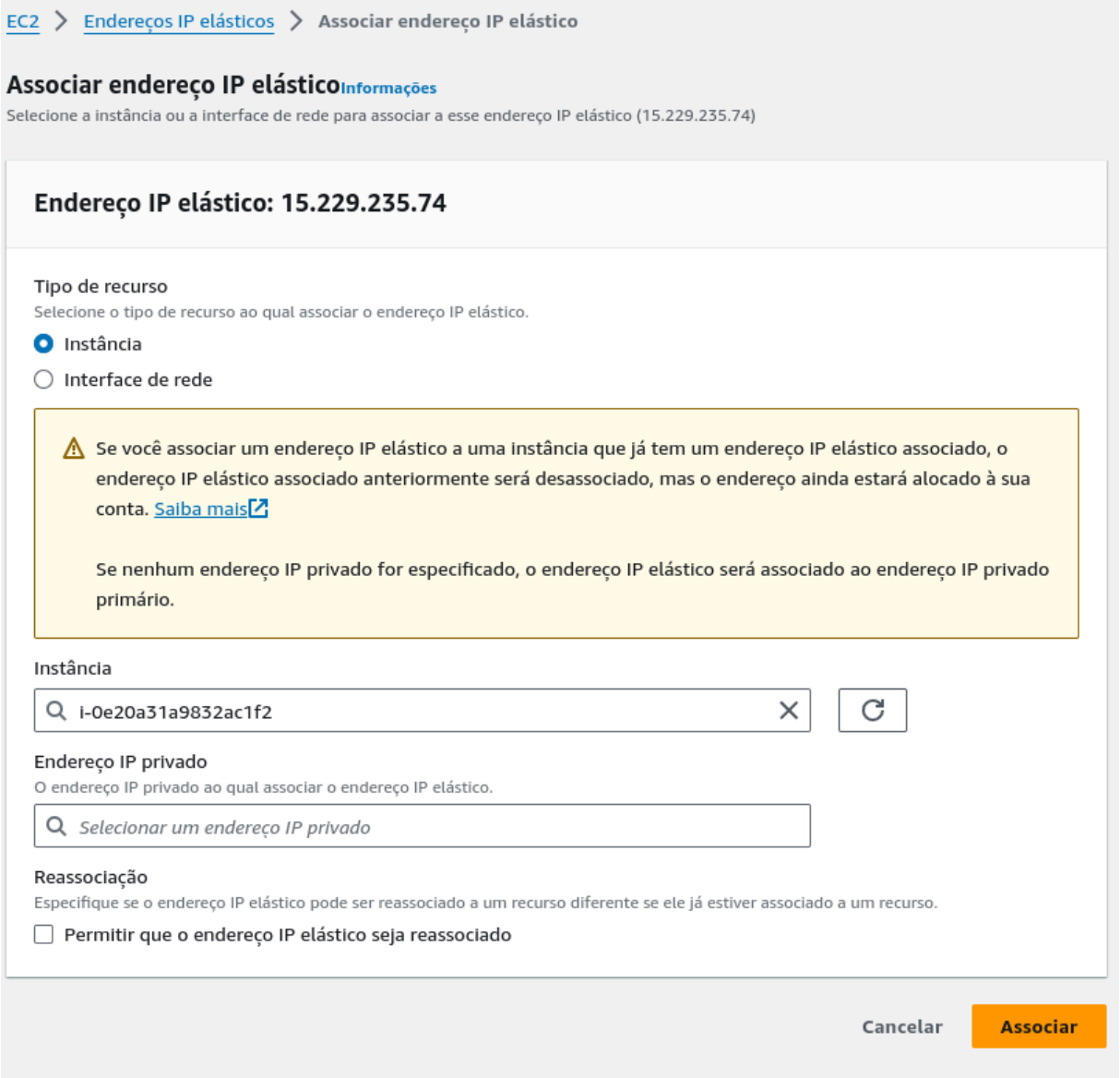
IP pode ser armazenado como uma variável de ambiente no GitLab e referenciado no arquivo de CI/CD, tornando ajustes manuais desnecessários, uma vez que este endereço não será mais alterado. As Figuras 24 e 25 ilustram como a criação e alocação de IP elástico é feito na AWS.

Figura 24 - Criação de um IP elástico



Fonte: Elaborada pelo Autor

Figura 25 - Associação de IP Estático a uma instância



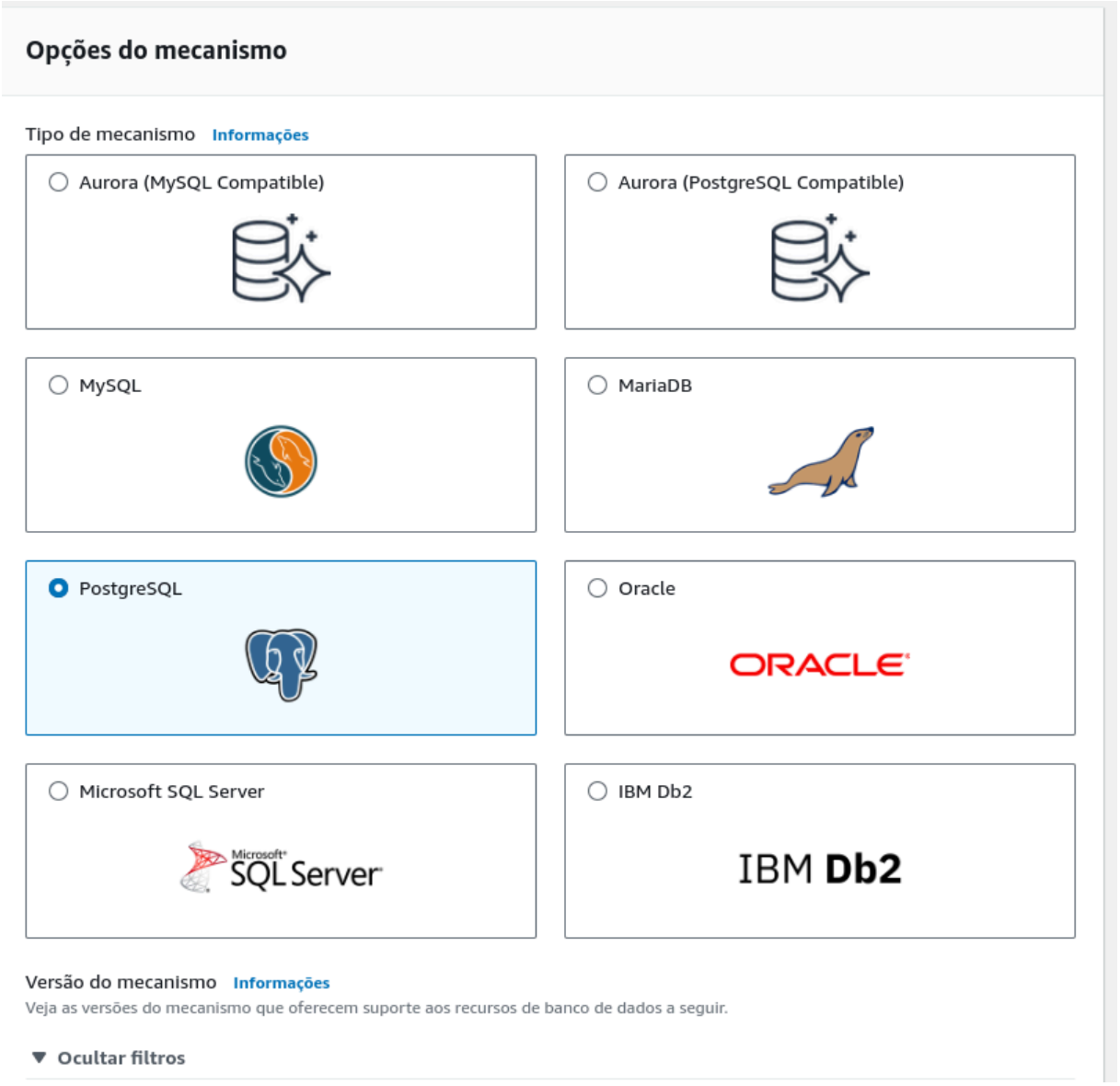
Fonte: Elaborada pelo Autor

3.6.2 CONEXÃO COM BANCO DE DADOS

A conexão com o banco de dados é feita através de outro produto da AWS, o RDS. Foi decidido utilizá-lo pois ele possui uma gerência de banco mais ampla e simplificada pelo d*ashboard* da Amazon do que um container Docker teria, apesar do Docker Compose facilitar muitas configurações. Para utilizar o banco é

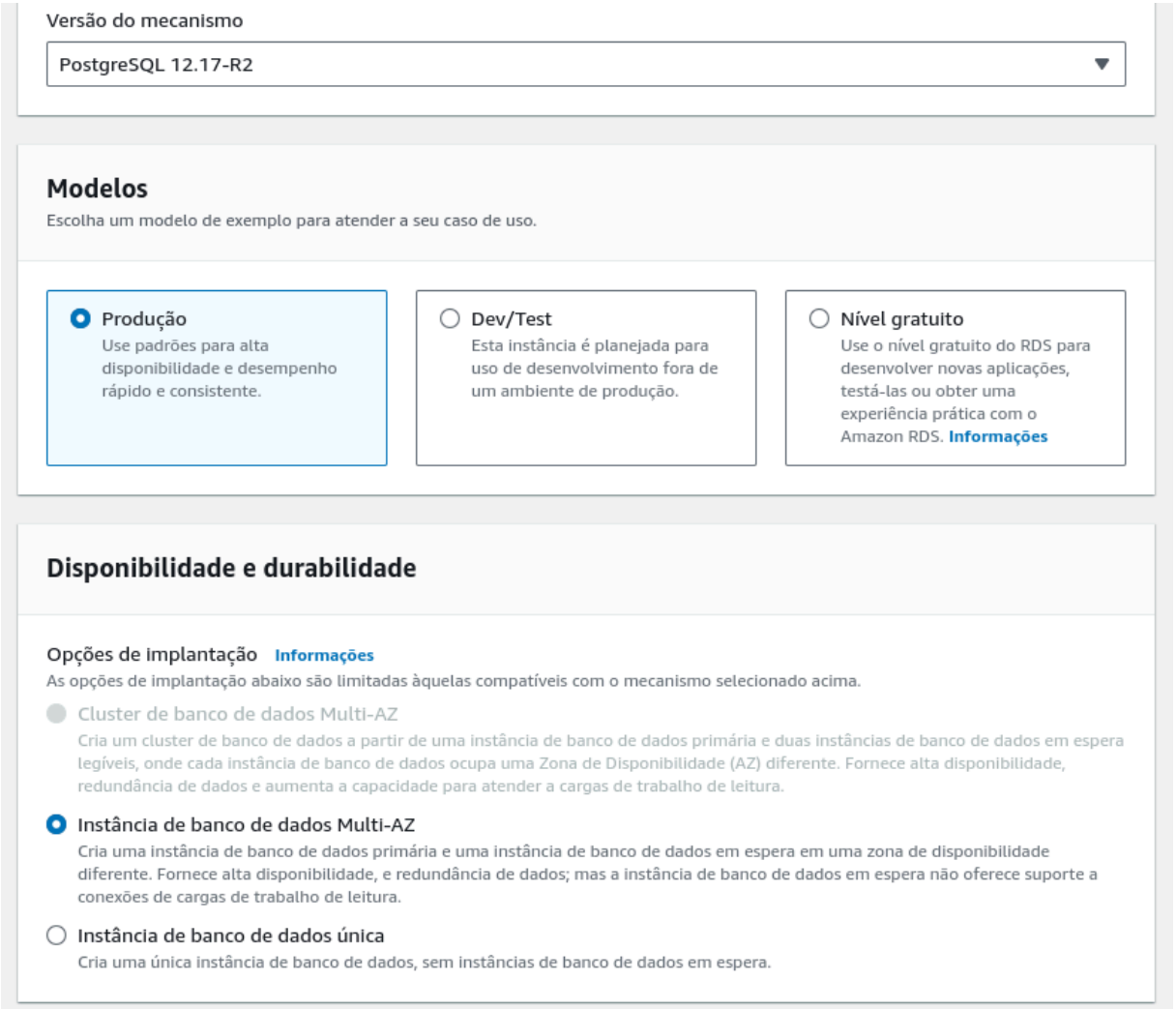
necessário primeiro criar uma instância de banco de dados, as Figuras 26 a 28 a seguir ilustram como é feita a criação através do *dashboard* da AWS.

Figura 26 - Opções de Banco



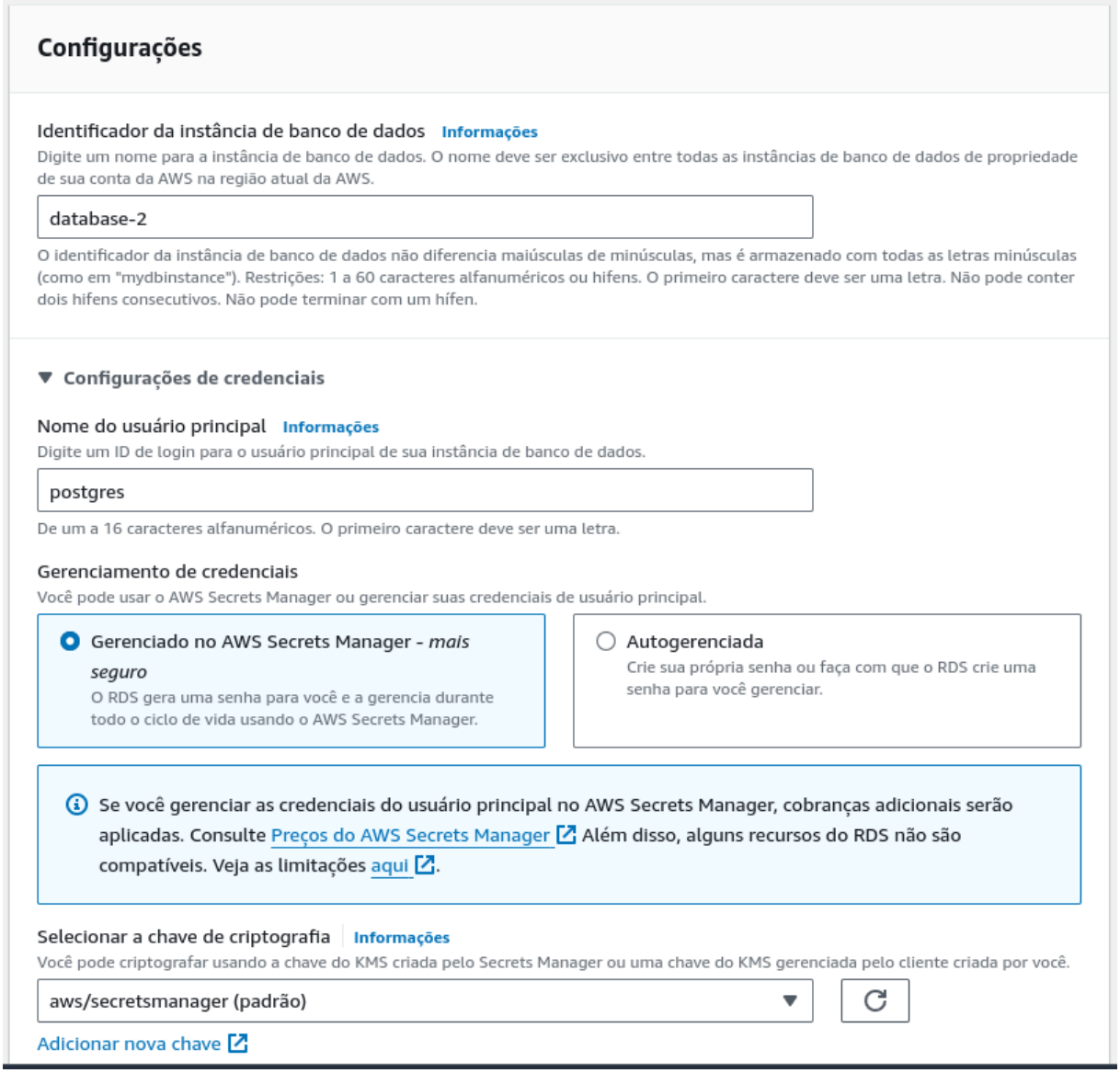
Fonte: Elaborada pelo Autor

Figura 27 - Modelos de banco de Dados



Fonte: Elaborada pelo Autor

Figura 28 - Configuração da Instância

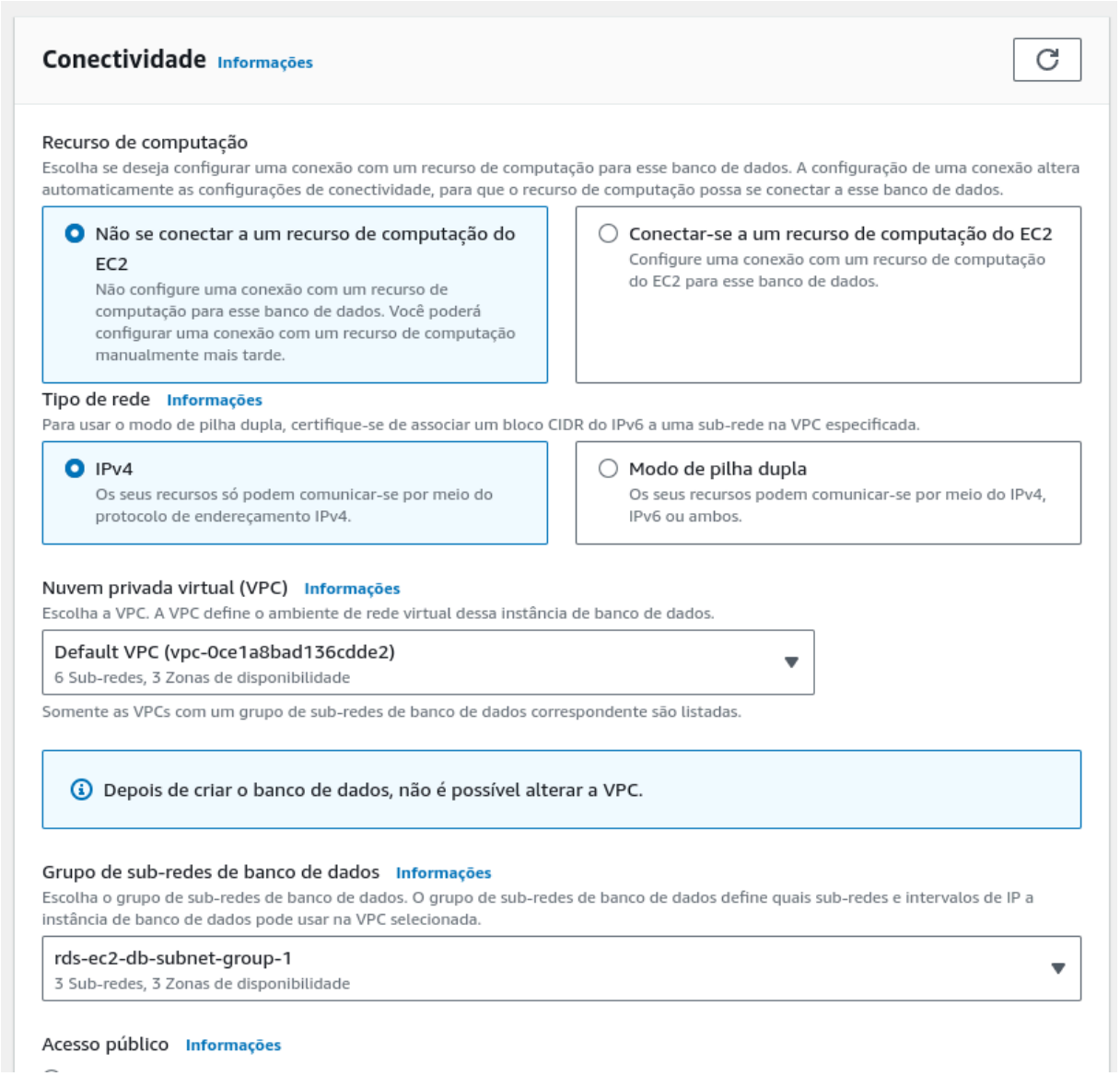


Fonte: Elaborada pelo Autor

No console é possível definir coisas como o tipo de SGBD, versão do SGBD, memória disponível e nome da instância e conectividades disponíveis para o banco. Também é necessário configurar uma VPC (*Virtual Private Cloud*), para que aplicações consigam se conectar com o banco via internet. Desta forma serviços

como instâncias de EC2 conseguem utilizar o banco de dados. A Figura 29 ilustra esse processo.

Figura 29 - Configuração de Conectividade do RDS



Fonte: Elaborada pelo Autor

3.7 TESTES

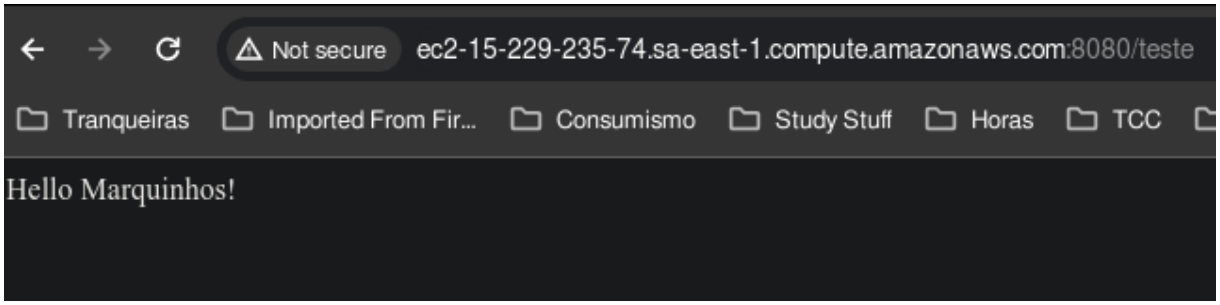
Nesta seção serão executados testes para verificar se o projeto está executando de forma automatizada a *pipeline* de *Deploy*. Também será verificado se

o software está conseguindo se conectar com o banco de dados e se Nginx está fazendo a *proxy* reverso na instância, pois a porta disponível para acesso na instância é a 8080 (padrão para web), porém a aplicação será executada na porta 9096, logo é necessário que o Nginx faça esse redirecionamento a cada requisição.

3.7.1 TESTE DE *DEPLOY*

Foi criada uma branch diretamente no GitLab com o objetivo de testar o funcionamento da *pipeline* automatizada. Nesta branch foi feita uma alteração na mensagem de retorno do *endpoint* /teste de “Hello Marquinhos” para “olá nãna!”, logo se o *endpoint* for acessado antes da *pipeline* ser executada, ele terá o seguinte retorno conforme a Figura 30 abaixo.

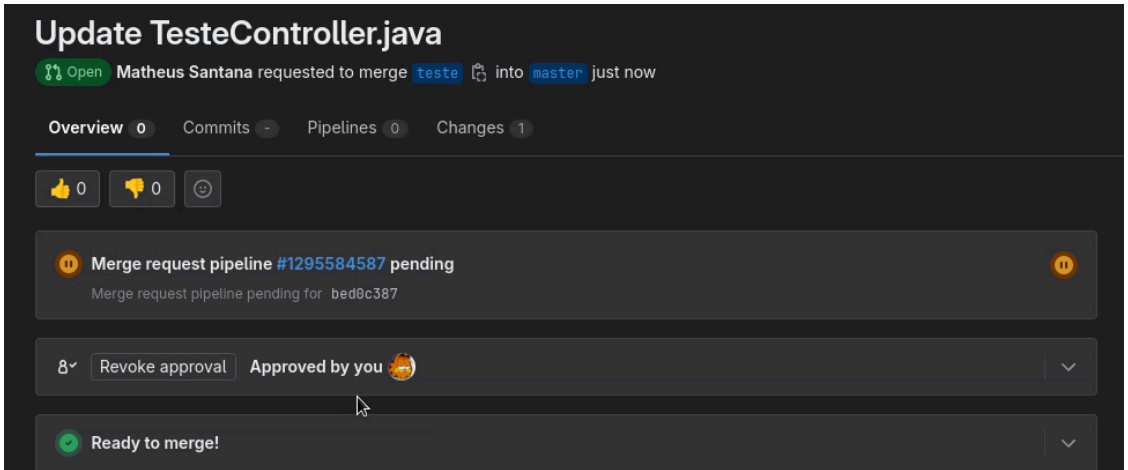
Figura 30 - Atual Retorno do Endpoint /teste



Fonte: Elaborada Pelo Autor

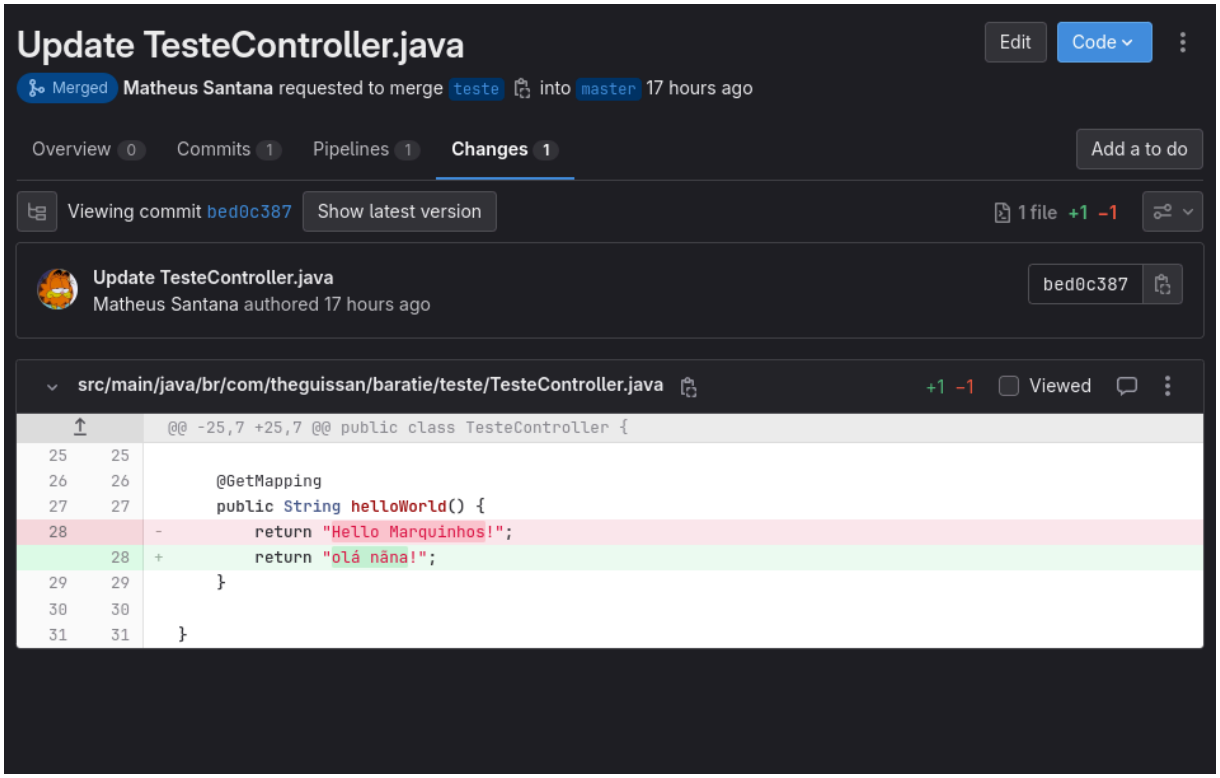
Após a alteração no código ser feita, foi criado um merge-request para integrar as alterações na branch master. No momento em que o merge-request foi feito, o GitLab disparou a execução da *pipeline* automaticamente, executando apenas o *stage* de teste, as Figuras 31 a 35 abaixo ilustram esse processo.

Figura 31 - Merge-request



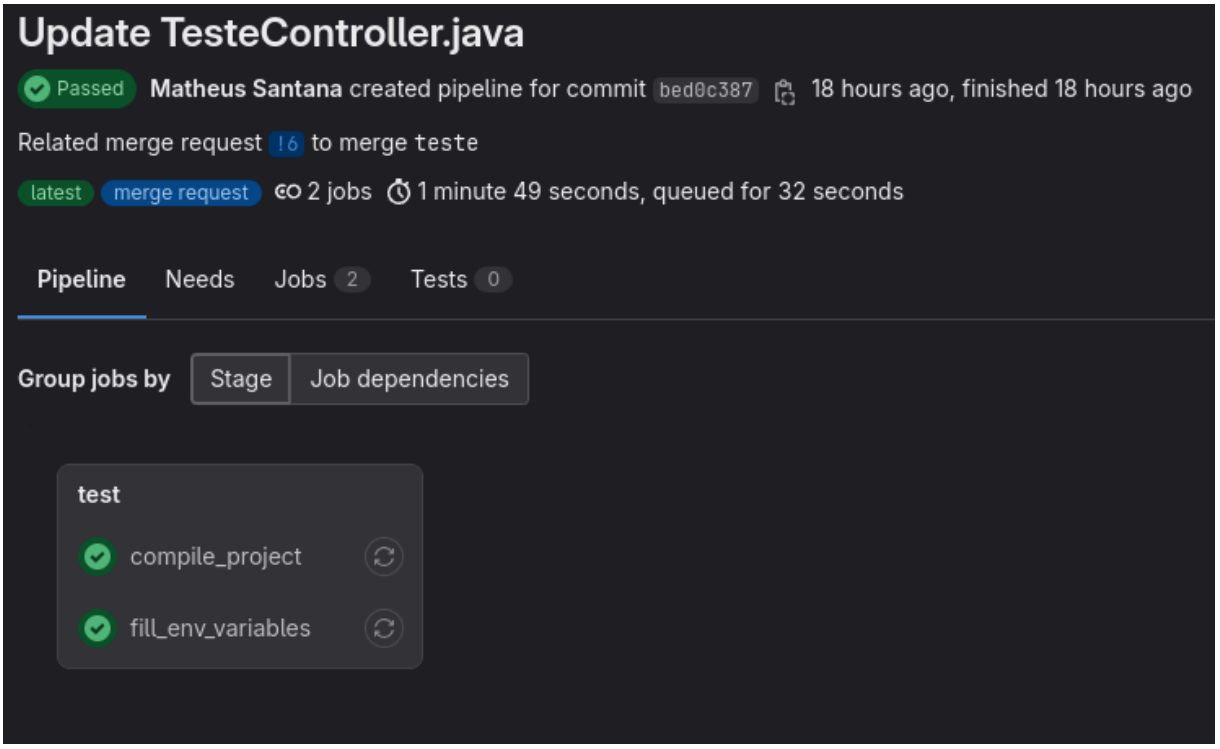
Fonte: Elaborada pelo Autor

Figura 32 - Alterações no Código



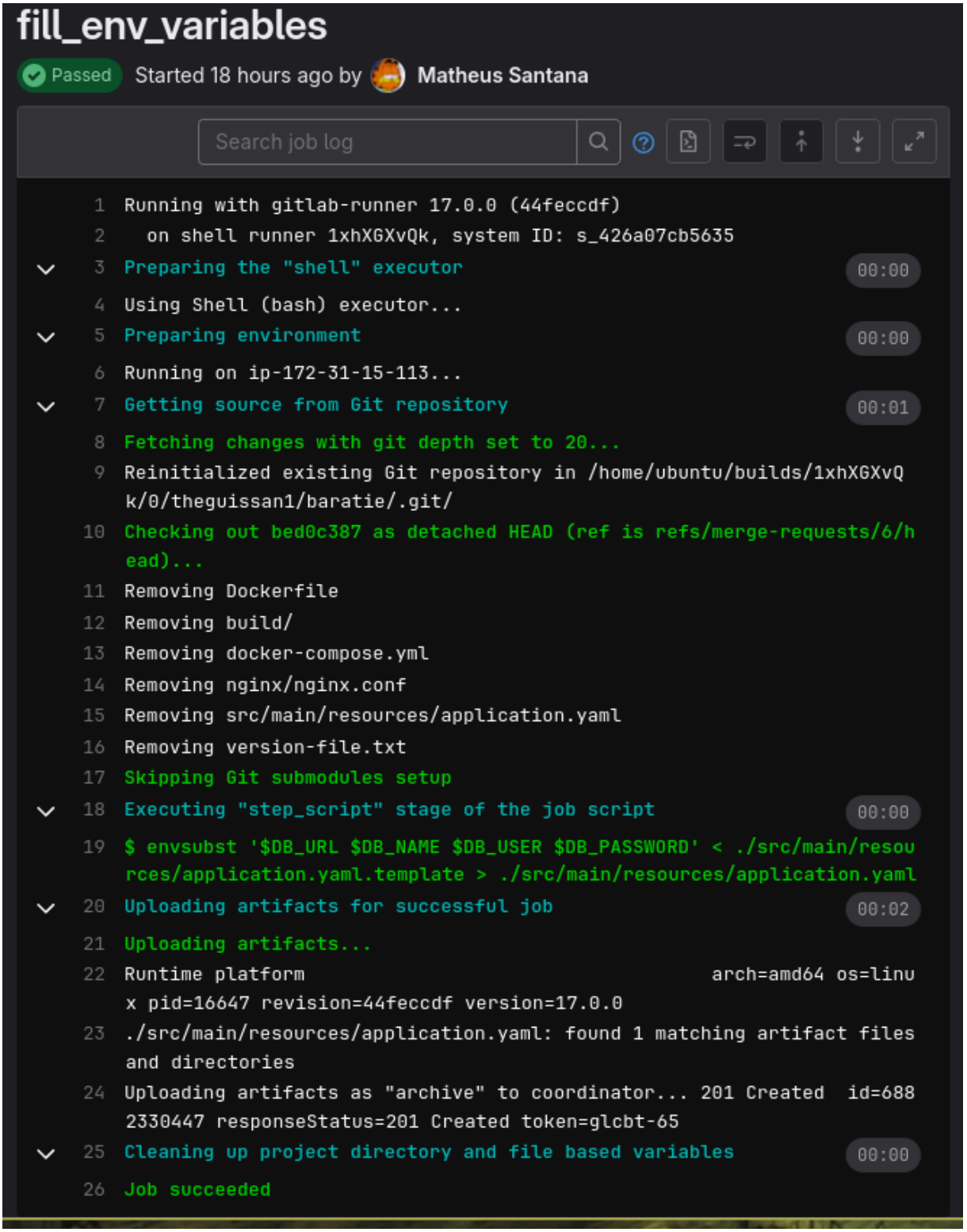
Fonte: Elaborada pelo Autor

Figura 33 - Execução do Stage de Teste



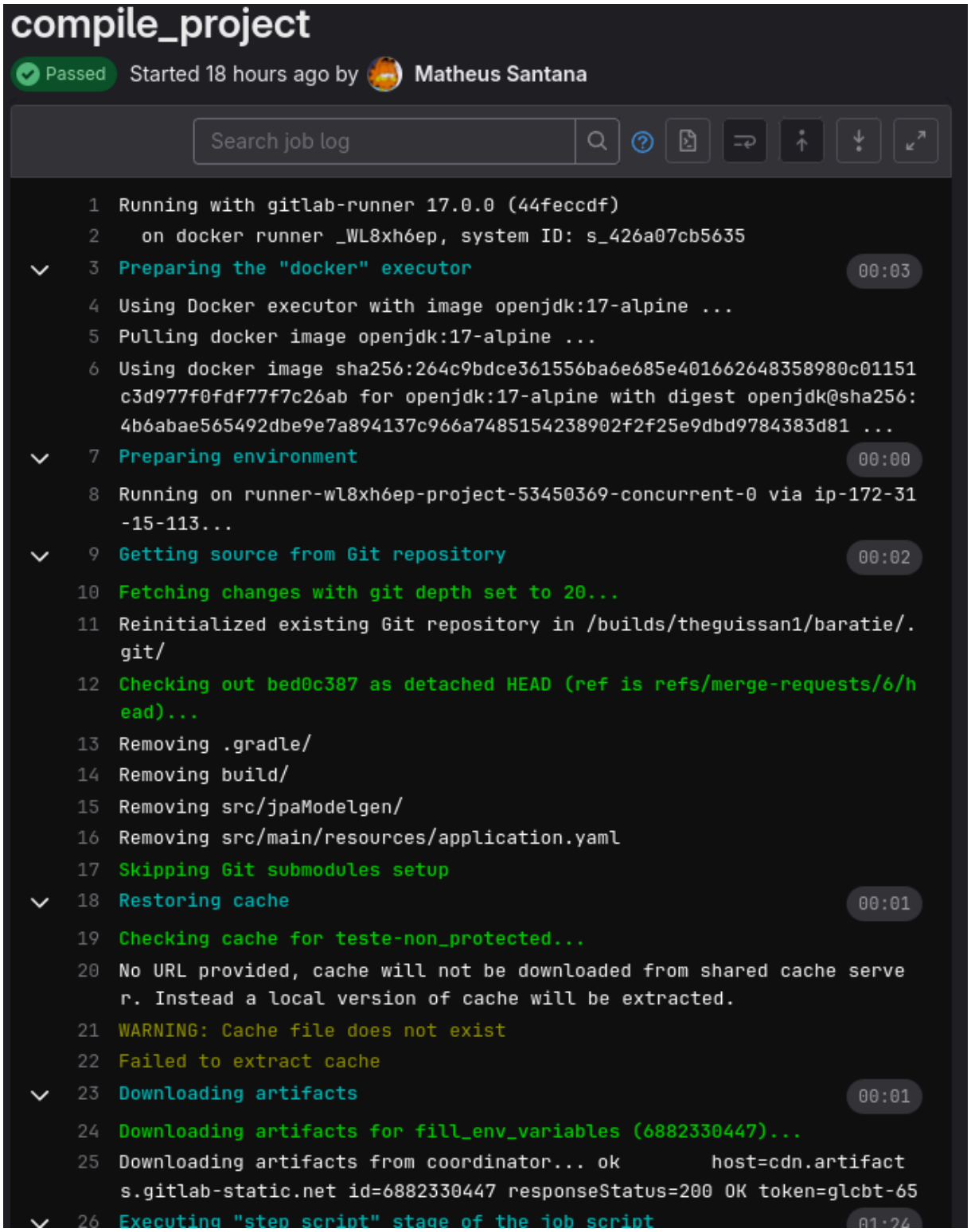
Fonte: Elaborada pelo Autor

Figura 34 - Console do Job de Preenchimento de Variáveis



Fonte: Elaborada pelo Autor

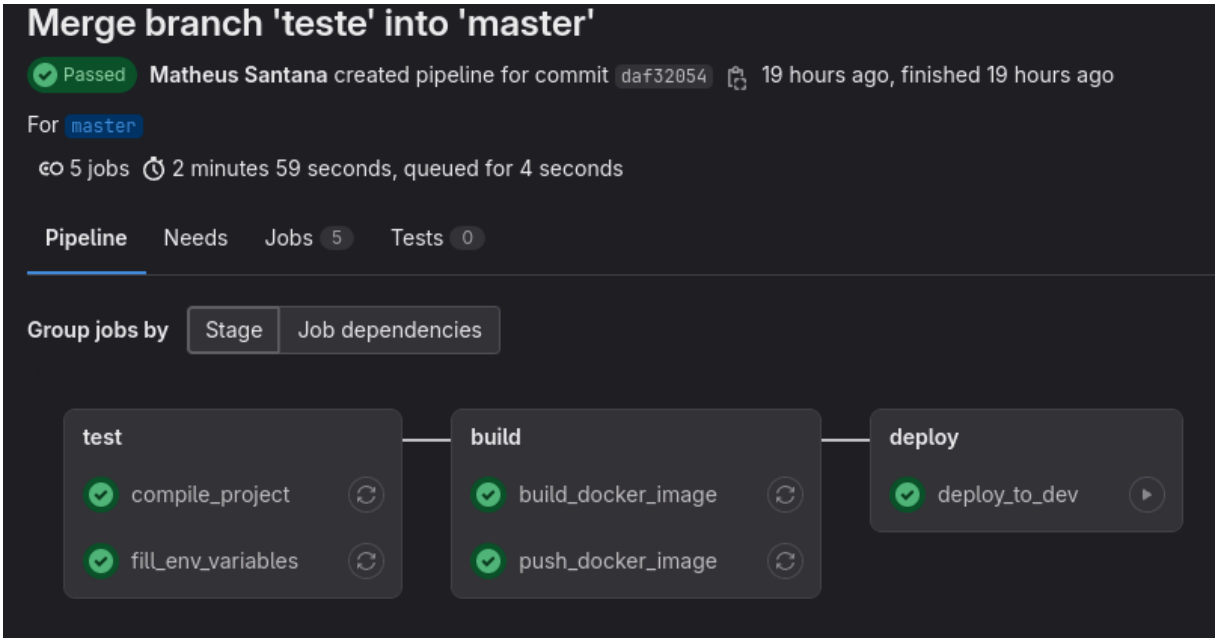
Figura 35 - Console do *job* de Compilação do Projeto



Fonte: Elaborada pelo Autor

Após a *pipeline* ser executada e o código aprovado, foi executada a integração das alterações na branch master, então outra *pipeline* é acionada, porém desta vez com o objetivo de realizar o *Deploy* das alterações na instância da AWS. As Figuras 36 a 38 abaixo demonstram esse processo.

Figura 36 - Execução da Pipeline de *Deploy*

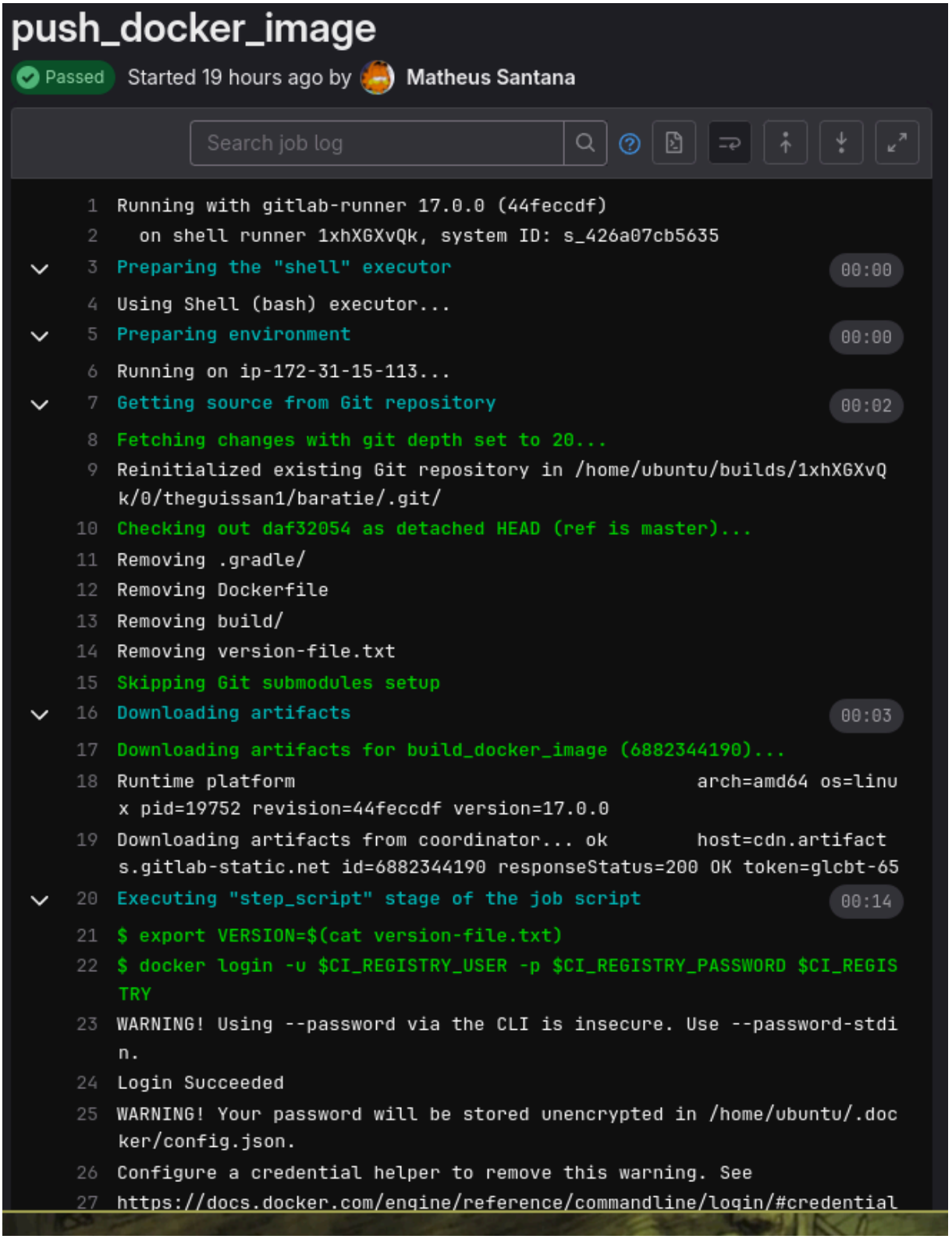


Fonte: Elaborada pelo Autor

Figura 37 - Console do Job de Construção da Imagem Docker do Projeto

Fonte: Elaborada pelo Autor

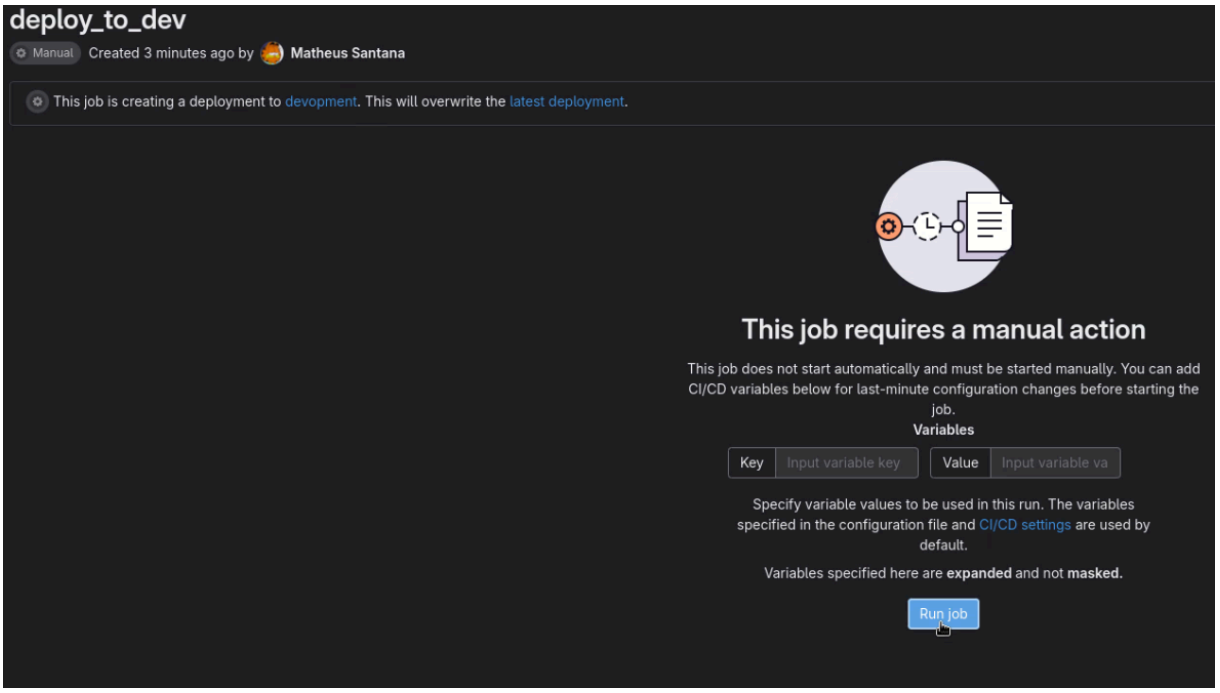
Figura 38 - Console do *job* de Upload da Imagem Docker no Registry



Fonte: Elaborada pelo Autor

Após a execução dos dois primeiros stages, o GitLab bloqueia a execução do *job* de de *Deploy* para que ele só ocorra caso alguém responsável pelo projeto aprove o *Deploy*, a Figura 39 ilustra esse caso.

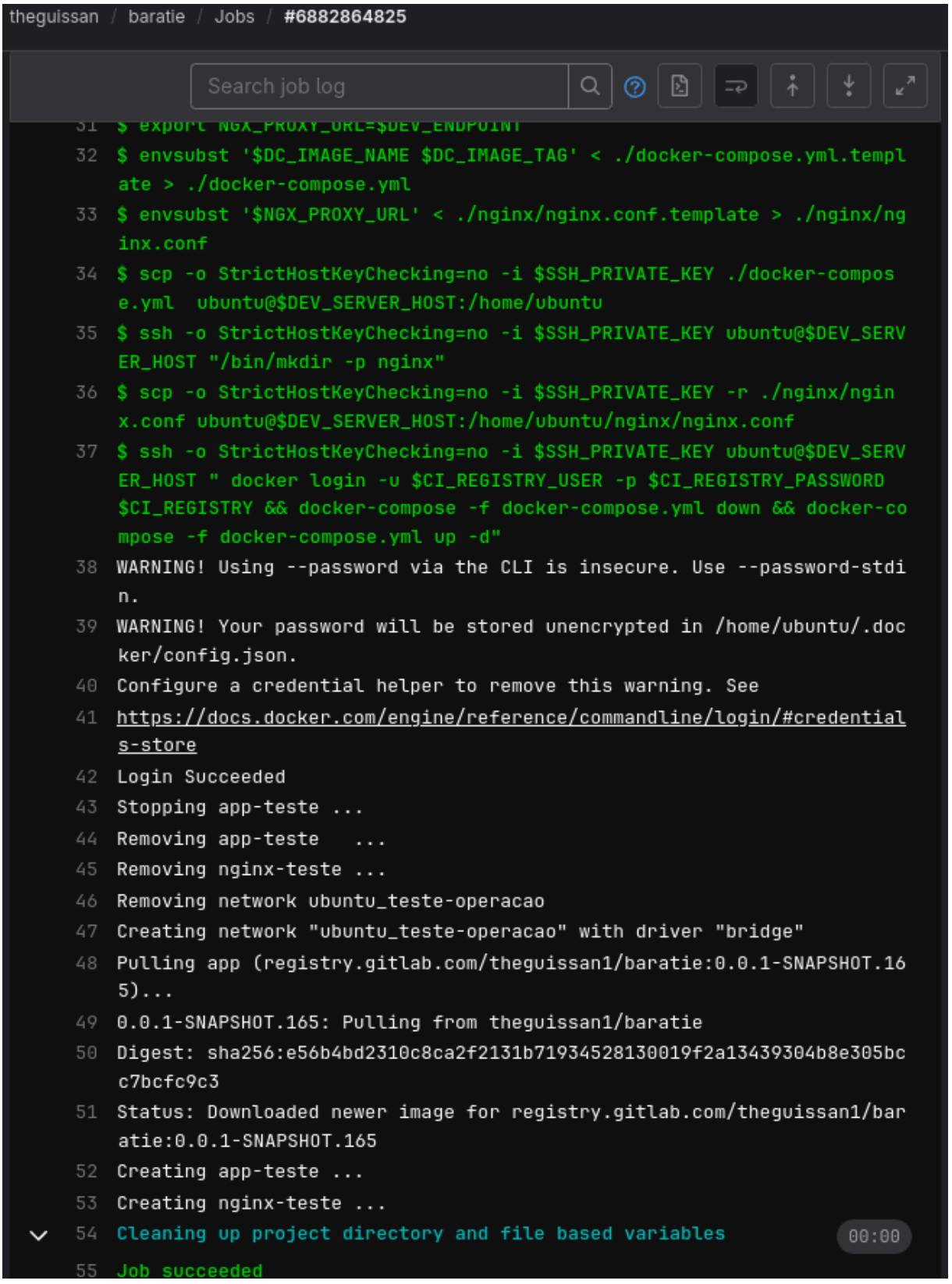
Figura 39 - Console do *job* de *Deploy* aguardando acionamento



Fonte: Elaborada pelo Autor.

Após o *job* ser acionado manualmente o processo de *Deploy* é de fato iniciado, com o *job* acessando a instância da AWS e alterando a versão da imagem e reiniciando a aplicação para que o software com a alteração seja efetivamente implantado. A Figura 40 abaixo ilustra esse processo.

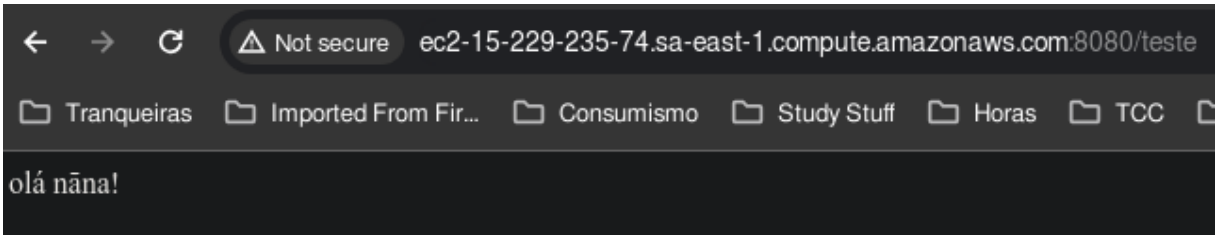
Figura 40 - Console do Job de *Deploy*


theguissan / baratie / Jobs / #6882864825
Search job log
31 $ export NGX_PROXY_URL=$DEV_ENDPOINT
32 $ envsubst '$DC_IMAGE_NAME $DC_IMAGE_TAG' < ./docker-compose.yml.template > ./docker-compose.yml
33 $ envsubst '$NGX_PROXY_URL' < ./nginx/nginx.conf.template > ./nginx/nginx.conf
34 $ scp -o StrictHostKeyChecking=no -i $SSH_PRIVATE_KEY ./docker-compose.yml ubuntu@$DEV_SERVER_HOST:/home/ubuntu
35 $ ssh -o StrictHostKeyChecking=no -i $SSH_PRIVATE_KEY ubuntu@$DEV_SERVER_HOST "/bin/mkdir -p nginx"
36 $ scp -o StrictHostKeyChecking=no -i $SSH_PRIVATE_KEY -r ./nginx/nginx.conf ubuntu@$DEV_SERVER_HOST:/home/ubuntu/nginx/nginx.conf
37 $ ssh -o StrictHostKeyChecking=no -i $SSH_PRIVATE_KEY ubuntu@$DEV_SERVER_HOST " docker login -u $CI_REGISTRY_USER -p $CI_REGISTRY_PASSWORD $CI_REGISTRY && docker-compose -f docker-compose.yml down && docker-compose -f docker-compose.yml up -d"
38 WARNING! Using --password via the CLI is insecure. Use --password-stdin.
39 WARNING! Your password will be stored unencrypted in /home/ubuntu/.docker/config.json.
40 Configure a credential helper to remove this warning. See
41 https://docs.docker.com/engine/reference/commandline/login/#credentials-store
42 Login Succeeded
43 Stopping app-teste ...
44 Removing app-teste ...
45 Removing nginx-teste ...
46 Removing network ubuntu_teste-operacao
47 Creating network "ubuntu_teste-operacao" with driver "bridge"
48 Pulling app (registry.gitlab.com/theguissan1/baratie:0.0.1-SNAPSHOT.165)...
49 0.0.1-SNAPSHOT.165: Pulling from theguissan1/baratie
50 Digest: sha256:e56b4bd2310c8ca2f2131b71934528130019f2a13439304b8e305bcc7bcfc9c3
51 Status: Downloaded newer image for registry.gitlab.com/theguissan1/baratie:0.0.1-SNAPSHOT.165
52 Creating app-teste ...
53 Creating nginx-teste ...
54 Cleaning up project directory and file based variables
00:00
55 Job succeeded

Fonte: Elaborada pelo Autor

Após a execução do *job*, a instância da AWS executa o software com as alterações feitas, logo se o *endpoint* /teste for acessado novamente, o retorno será “olá nãna!”, ao invés de “Hello Marquinhos!”, a Figura 41 abaixo ilustra essa mudança.

Figura 41 - Retorno do Endpoint com a alteração

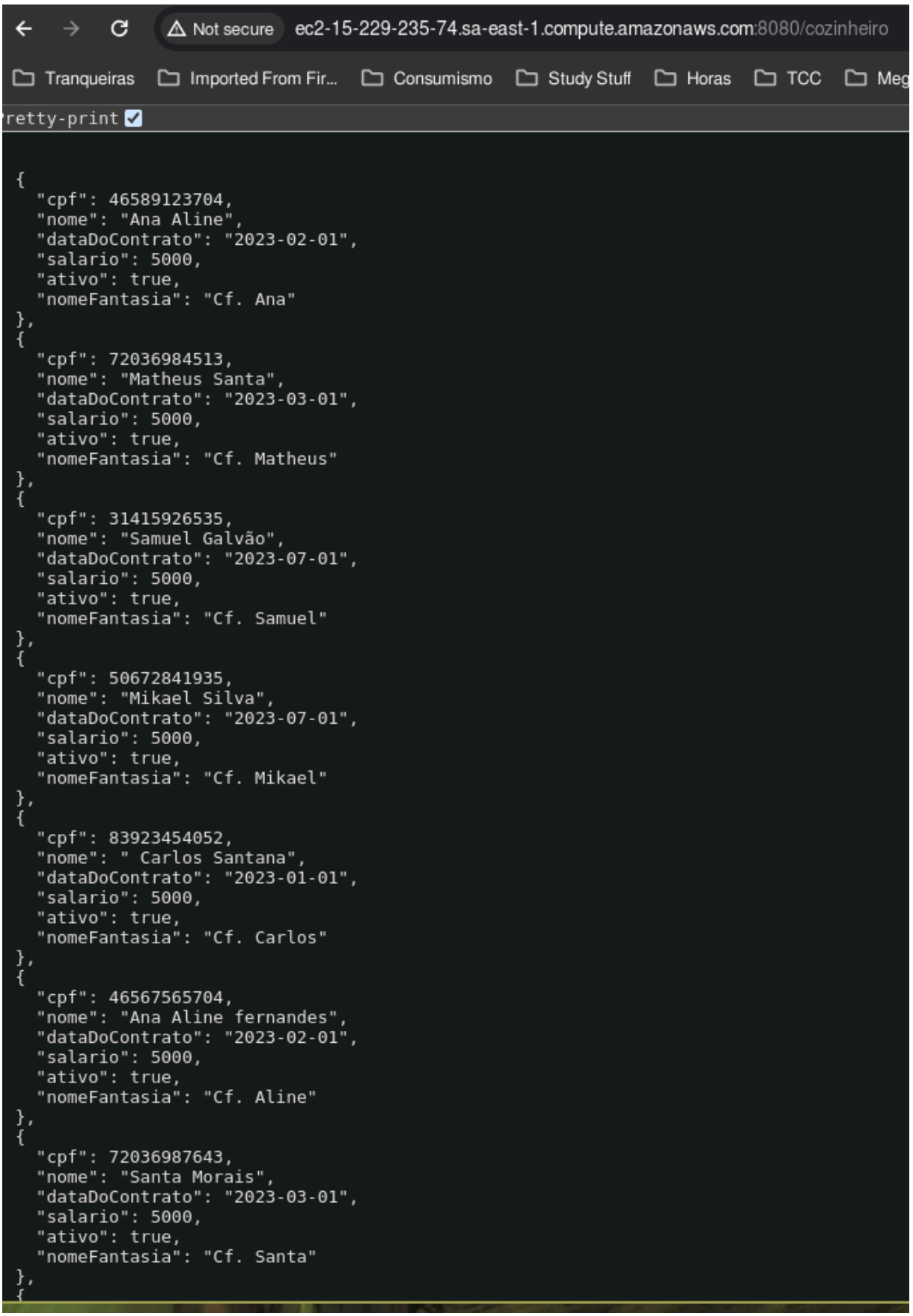


Fonte: Elaborada pelo Autor

3.7.2 TESTE DE CONEXÃO COM O BANCO DE DADOS

Para verificar se a aplicação conseguiu fazer conexão com o banco de dados corretamente, foi feito o acesso ao *endpoint* /cozinheiro, que busca todos os registros da tabela cozinheiro do banco de dados. Ao fazer o acesso nota-se que o *endpoint* retornou um json com a lista de cozinheiros conforme esperado, logo a conexão com o banco de dados está sendo feita com sucesso. A Figura 42 ilustra esse processo.

Figura 42 - Retorno do endpoint /cozinheiro

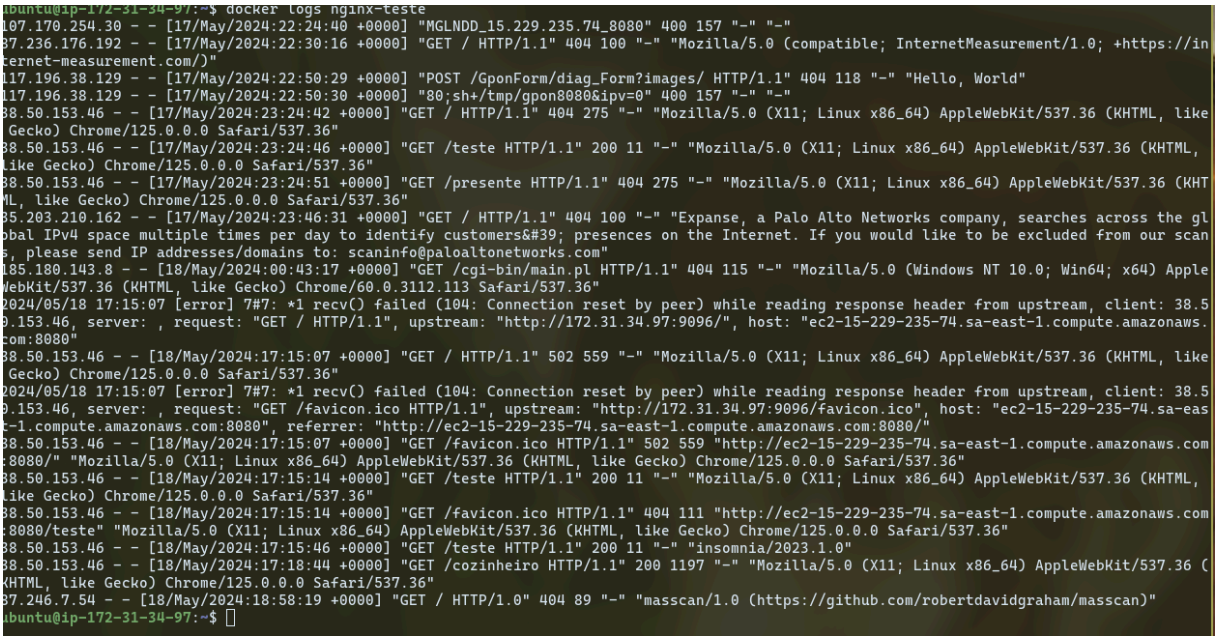

Not secure
ec2-15-229-235-74.sa-east-1.compute.amazonaws.com:8080/cozinheiro
Tranqueiras
Imported From Fir...
Consumismo
Study Stuff
Horas
TCC
retty-print
{
"cpf": 46589123704,
"nome": "Ana Aline",
"dataDoContrato": "2023-02-01",
"salario": 5000,
"ativo": true,
"nomeFantasia": "Cf. Ana"
},
{
"cpf": 72036984513,
"nome": "Matheus Santa",
"dataDoContrato": "2023-03-01",
"salario": 5000,
"ativo": true,
"nomeFantasia": "Cf. Matheus"
},
{
"cpf": 31415926535,
"nome": "Samuel Galvão",
"dataDoContrato": "2023-07-01",
"salario": 5000,
"ativo": true,
"nomeFantasia": "Cf. Samuel"
},
{
"cpf": 50672841935,
"nome": "Mikael Silva",
"dataDoContrato": "2023-07-01",
"salario": 5000,
"ativo": true,
"nomeFantasia": "Cf. Mikael"
},
{
"cpf": 83923454052,
"nome": " Carlos Santana",
"dataDoContrato": "2023-01-01",
"salario": 5000,
"ativo": true,
"nomeFantasia": "Cf. Carlos"
},
{
"cpf": 46567565704,
"nome": "Ana Aline fernandes",
"dataDoContrato": "2023-02-01",
"salario": 5000,
"ativo": true,
"nomeFantasia": "Cf. Aline"
},
{
"cpf": 72036987643,
"nome": "Santa Morais",
"dataDoContrato": "2023-03-01",
"salario": 5000,
"ativo": true,
"nomeFantasia": "Cf. Santa"
},

Fonte: Elaborada pelo Autor

3.7.3 TESTES COM NGINX

Ao fazer um acesso aos *endpoints* na porta 8080, o Nginx redirecionou as requisições para a porta 9096, isso pode ser confirmado verificando o log do container conforme a Figura 43 ilustra.

Figura 43 - Logs de redirecionamento do Nginx

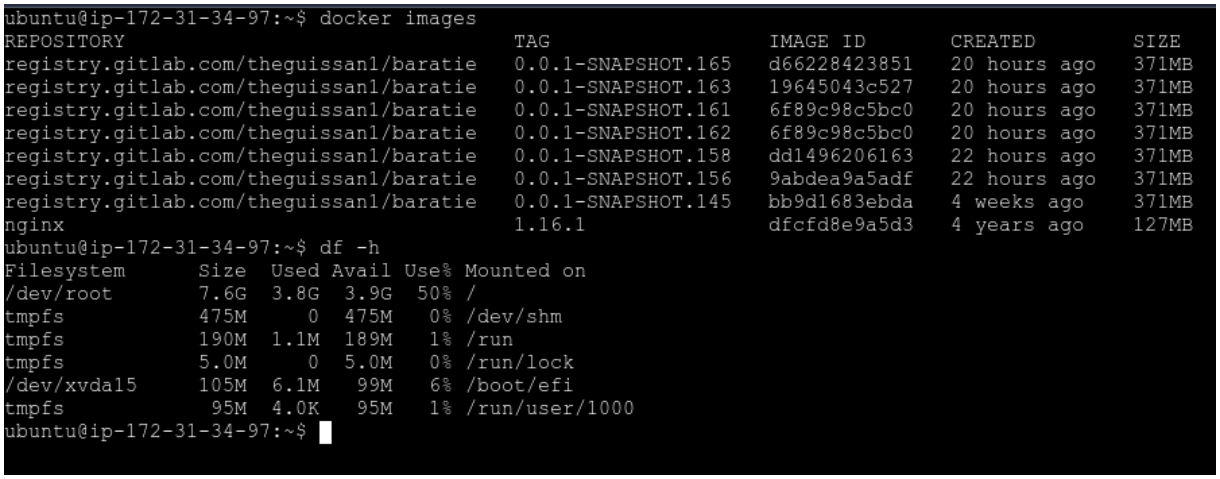


Fonte: Elaborada pelo Autor

4 ANÁLISE DOS RESULTADOS OBTIDOS

Como a cada iteração da *pipeline* uma versão nova da imagem da aplicação era baixada na instância, foi notado que o Docker acaba acumulando muitas versões que não estavam sendo utilizadas, e isso acabou se tornando um problema, pois cada imagem possui em média 371 MiB de tamanho, e as instâncias por terem armazenamento limitado de 8 GB por conta da licença gratuita da AWS ficaram sem espaço em disco para baixar novas versões, sendo assim necessário o acesso à instância remotamente e a remoção das versões não utilizadas. Esse problema pode ser verificado na Figura 44.

Figura 44 - Imagens Docker não utilizadas presentes na instância



Fonte: Elaborada pelo Autor

Logo é recomendado que caso um ambiente parecido com a que foi feito seja implementado na realidade, seja feita uma a otimização das imagens Docker ou a seja feita alguma rotina na instância que remova as imagens não utilizadas de tempos em tempos.

Outro problema encontrado foi a configuração manual dos ambientes, pois apesar de *Deploy* ser quase inteiramente automático, o ambiente em que eles são executados também deveria ser de fácil configuração, pois essa etapa acaba sendo burocrática e esse problema é uma dos pontos que a cultura *DevOps* tenta sanar. E

esse problema é altamente escalável pois se houvesse mais de uma instância do projeto para ser configurada, levaria ainda mais tempo para finalizar essa etapa. Recomenda-se então que seja aplicado algum software que automatize esse processo caso um ambiente parecido seja implementado na realidade.

5 CONSIDERAÇÕES FINAIS

O presente trabalho propôs-se a demonstrar se era possível implementar um ambiente de *Deploy* de ponto a ponta em um projeto de software fazendo uso da cultura de *DevOps*.

O estudo realizado neste trabalho permitiu concluir que a implantação de software é um desafio tão grande quanto o desenvolvimento de um software em si. Mostra-se necessário ter experiência tanto do campo de desenvolvimento quanto ao de operação, e que a cultura *DevOps* auxilia de forma significante na facilidade de integração entre Engenheiros de Software e Analistas de Infraestrutura.

Também concluiu-se que o GitLab mostrou–se uma ferramenta que atende essa demanda de equilibrar o lado do Engenheiro e do Analista de forma simples e concisa. Da mesma forma a AWS se mostrou um serviço que combina muito bem com o ambiente de *DevOps*, apesar das restrições com o ferramental gratuito oferecido.

Em certo momento durante o desenvolvimento deste trabalho, foi notada que a limitação de hardware pode ser um problema a ser enfrentado ao se trabalhar com automatização de processos, logo é pertinente que ao se trabalhar no desenvolvimento de um ambiente real de *Deploy*, os envolvidos levem essa questão em conta, tomando decisões que sanem esse problema.

Por fim, este trabalho contribuiu para a disseminação do conhecimento sobre a cultura de *DevOps*, e espera-se que este estudo seja útil para profissionais de TI, pesquisadores e organizações interessadas em utilizar a cultura de *DevOps* em seus trabalhos. Esse trabalho também deve serventia em demonstrar que a automatização de processos é algo que pode refletir significativamente na diminuição de erros humanos e na economia de tempos. Os objetivos propostos neste trabalho foram todos cumpridos.

5.1 TRABALHOS FUTUROS

Como trabalhos futuros sugere-se:

- Desenvolver uma aplicação Frontend, que consuma a API desenvolvida neste trabalho.
- Realizar a automatização da configuração das instâncias da AWS, utilizando ferramentas como Ansible e Terraform.
- Fazer uma separação mais resiliente entre ambientes de desenvolvimento, homologação e produção.
- Desenvolver um sistema de monitoramento que auxilie na gestão de recursos das instâncias, com ferramentas como o Grafana e o Prometheus.

REFERÊNCIAS

AMAZON. **O que é o Amazon EC2? - Amazon Elastic Compute Cloud.** 2024. Disponível em: <https://docs.aws.amazon.com/pt_br/AWSEC2/latest/UserGuide/concepts.html>. Acesso em: 17 maio 2024.

AMAZON. **O que é o Amazon Relational Database Service (Amazon RDS)? - Amazon Relational Database Service.** 2024. Disponível em: <https://docs.aws.amazon.com/pt_br/AmazonRDS/latest/UserGuide/Welcome.html>. Acesso em: 17 maio 2024.

C10K Problem: Understanding and Overcoming the 10,000 Concurrent Connections Challenge. **Web Hosting Geeks**. 26 fev. 2023. Blog. Disponível em: <https://webhostinggeeks.com/blog/c10k-problem-understanding-and-overcoming-the-10000-concurrent-connections-challenge/>. Acesso em: 12 nov. 2023

CNN. **Procura por profissionais de tecnologia cresce 671% durante a pandemia.** 27 out. 2021. Disponível em: <https://www.cnnbrasil.com.br/economia/procura-por-profissionais-de-tecnologia-cresce-671-durante-a-pandemia/>. Acesso em: 23 nov. 2023.

COWELL, C.; LOTZ, N.; TIMBERLAKE, C. Automating DevOps with GitLab CI/CD Pipelines. [s.l.] Packt Publishing, 2023.

FONSECA, Elton. Criando instâncias no Amazon EC2. **Treinaweb**, 2019. Disponível em: <https://www.treinaweb.com.br/blog/criando-instancias-no-amazon-ec2>. Acesso em: 17 maio 2024.

FREEMAN, E. **DevOps For Dummies**. EUA: John Wiley & Sons, 2019. Acesso em: 26 set. 2023.

G1. **A área de TI deve gerar quase 420 mil vagas até 2025, mas faltam profissionais.** 13 mar. 2023. Disponível em: <https://g1.globo.com/jornal-hoje/noticia/2023/03/13/area-de-ti-deve-gerar-quase-420-mil-vagas-ate-2025-mas-faltam-profissionais.ghtml>. Acesso em: 23 nov. 2023.

GARRETT, O. Inside NGINX: Designed for Performance & Scalability. **Nginx.** 10 jun. 2015. Blog. Disponível em: <https://www.nginx.com/blog/inside-nginx-how-we-designed-for-performance-scale/>. Acesso em: 12 nov. 2023.

GOOGLE. **O que são contêineres?** 2023**.** Disponível em: <https://cloud.google.com/learn/what-are-containers?hl=pt-br>. Acesso em: 15 nov. 2023.

INTERNATIONAL ORGANIZATION FOR STANDARDIZATION. **ISO/IEC/IEEE 12207:2017:** Systems and software engineering — Software life cycle processes. ISO, 2017. Acesso em 10 out. 2023.

KODEKLOUD. The Role of Docker in DevOps. **DEV Community**, 22 fev. 2020. Disponível em: <https://dev.to/kodekloud/the-role-of-docker-in-devops-1con>. Acesso em: 16 nov. 2023.

MACHADO, Gabriel. O que é VPC e como ela funciona? **Treinaweb**, 2021. Disponível em: <https://www.treinaweb.com.br/blog/o-que-e-vpc-e-como-ela-funciona>. Acesso em: 17 maio 2024.

PAESSLER. **Server - Definition and details.** 2016. Disponível em: <https://www.paessler.com/it-explained/server>. Acesso em: 9 nov. 2023.

POSEY, Brien. What is a Server? **Techtarget**, 2021. Disponível em: <https://www.techtarget.com/whatis/definition/server>. Acesso em: 30 out. 2023.

PREMKUMAR M. NGINX — Architecture. **Medium**, 2019. Disponível em: <https://medium.com/@premsuryamj/nginx-architecture-9f97cf7887e2>. Acesso em: 12 nov. 2023.

REDHAT, **What´ s Docker?**. 2022. Disponível em: <https://www.redhat.com/pt-br/topics/containers/what-is-docker>. Acesso em: 16 nov. 2023.

REDHAT, **What's a Linux container?**. 2022. Disponível em: <https://www.redhat.com/en/topics/containers/whats-a-linux-container>. Acesso em: 16 nov. 2023.

SOMMERVILLE, I. **Engenharia de Software**. 10. ed. São Paulo: Pearson, 2019. Acesso em: 26 set. 2023.

SYOZI, Ricardo. O que é a AWS? [Amazon Web Services] – Tecnoblog. **Tecnoblog**. 2022. Disponível em: <https://tecnoblog.net/responde/o-que-e-a-aws-amazon-web-services/>. Acesso em: 17 maio 2024.

SYSOEV, I. **Nginx**. Versão 1.25.1. F5 Inc. 2023. Disponível em: <https://www.nginx.com/resources/wiki/>. Acesso em: 12 nov. 2023. [software].

TANENBAUM, A. S., VAN STEEN, M. **Distributed Systems**. 4. ed. USA: Createspace Independent Publishing Platform, 2023. Acesso em: 10 out. 2023.

VALENTE, J. **Virtualização de servidores utilizando proxmox aplicado ao ambiente devops.** Trabalho de Conclusão de Curso (Graduação em Engenharia da Computação) - Escola Politécnica e de Artes, Pontifícia Universidade Católica de Goiás. Goiás. Acesso em: 14 nov. 2023.

VIANA, Jesiel. Arquitetura de aplicações web. **Guia Web**, 4 ago. 2023. Disponível em: <https://www.guiaweb.dev.br/apendices/i.-arquitetura-de-aplicacoes-web>. Acesso em: 16 out. 2023.

WAZLAWICK, Raul Sidnei. **Metodologia de Pesquisa para Ciência da Computação.** 2. ed. Rio de Janeiro: Elsevier Editora Ltda, 2014.