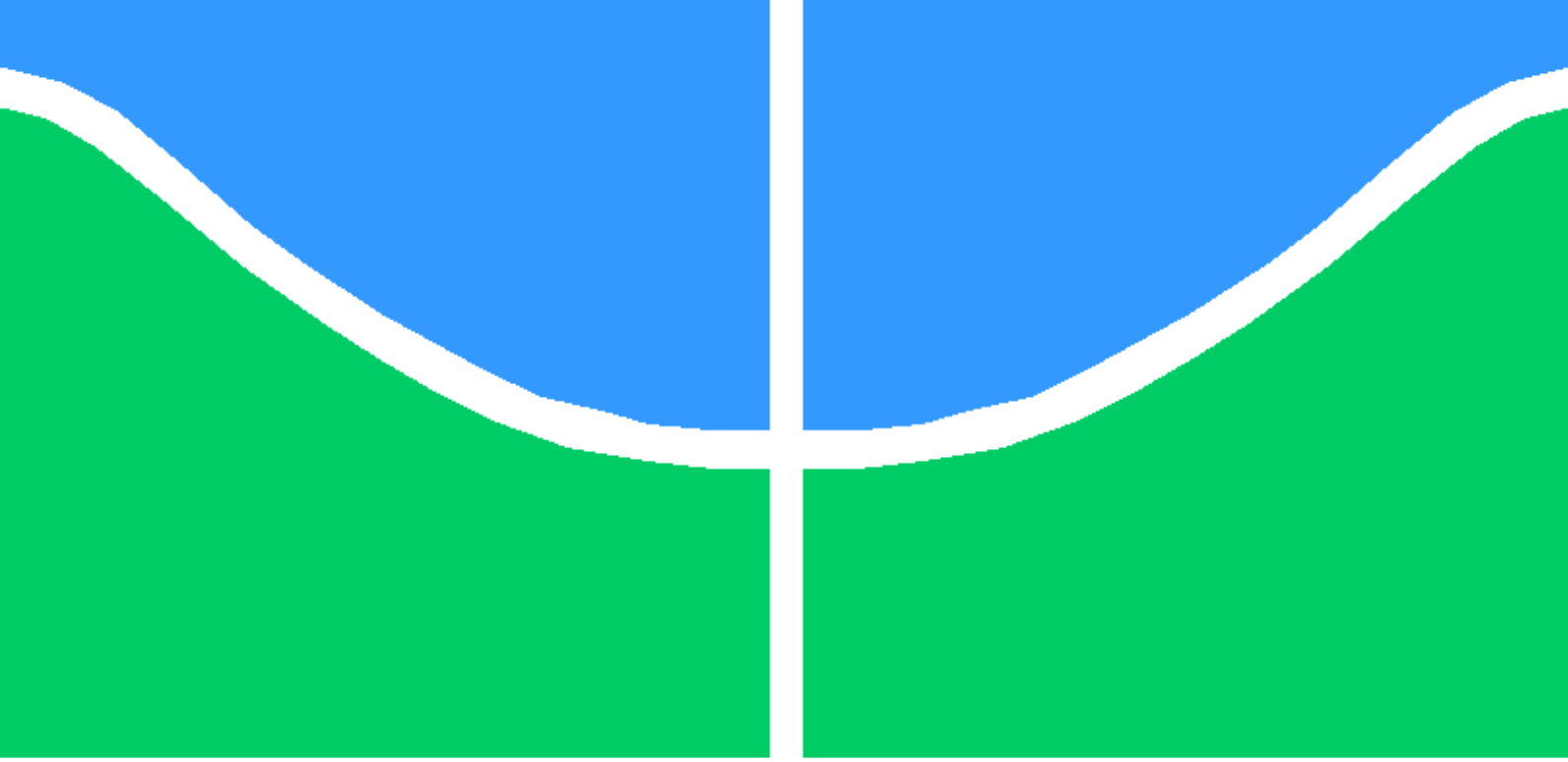


ESTUDO E ANÁLISE DE PIPELINES CI/CD ESCALÁVEIS DE ALTA DISPONIBILIDADE

MURILO CARVALHO SANTOS

RÔMULO MAGALHÃES DE ALCÂNTARA

DISSERTAÇÃO DE GRADUAÇÃO EM ENGENHARIA DE REDES DE COMUNICAÇÃO
DEPARTAMENTO DE ENGENHARIA ELÉTRICA

UNIVERSIDADE DE BRASÍLIA

FACULDADE DE TECNOLOGIA

UNIVERSIDADE DE BRASÍLIA
FACULDADE DE TECNOLOGIA

ESTUDO E ANÁLISE DE PIPELINES CI/CD ESCALÁVEIS DE ALTA DISPONIBILIDADE

Murilo Carvalho Santos

Rômulo Magalhães de Alcântara

Relatório submetido ao Departamento de Engenharia Elétrica como requisito parcial para obtenção do grau de Engenheiros de Redes de Comunicação.

Banca Examinadora:

Prof. Dr. Georges Daniel Amvame, ENE/UnB
Orientador

Prof. Dr. Fábio Lúcio Lopes de Mendonça, ENE/UnB
Examinador interno

Prof. Msc. Diego Martins de Oliveira, ENE/UnB
Examinador externo

BRASÍLIA, 29 DE SETEMBRO DE 2022.

Agradecimentos

Gostaria de agradecer à todos que me ajudaram no meu caminho acadêmico, afirmo que só consegui chegar até esse ponto por ajuda de cada um de vocês.

Agradeço ao professor Georges pelo orientação e dedicação, e principalmente pela paciência porque durante esse período final, ela foi testada várias vezes.

Agradeço meus colegas de curso, com abraço especial ao Arthur, e ao Rômulo porque sem eles meu período acadêmico seria muito mais longo e cansativo.

Por fim, mas não menos importante quero agradecer à minha família, em especial meus pais, que permitiram que esse momento fosse possível, não posso esquecer de agradecer a Deus por sempre estar ao meu lado.

- Murilo Carvalho Santos

Agradeço à minha família pela base fornecida durante toda a minha vida.

Agradeço aos meus amigos de mais longa data por sempre me fazerem sorrir e esquecer dos problemas.

Agradeço à minha namorada que durante o percurso acadêmico esteve comigo em todos os momentos, mesmo nos períodos mais sombrios.

Agradeço aos meus colegas de curso, especialmente à minha dupla de trabalho nesta dissertação e em quase todas as matérias do curso.

Agradeço ao professor Georges pela paciência nessa última jornada.

E por fim agradeço a Deus por ter colocados pessoas tão especiais na minha vida, pois só assim é possível ir mais longe.

- Rômulo Magalhães de Alcântara

Resumo

Neste presente documento serão apresentados conceitos envolvendo contêiner com o foco na utilização do *Docker* e o uso do Orquestrador de contêineres *Kubernetes*, ambas ferramentas são altamente utilizadas no mercado de trabalho, principalmente quando utiliza-se os conceitos de *DevOps*. Com isso em mente, vamos observar, analisar e estudar um dos pilares da cultura *DevOps*, a *pipelines CI/CD*, buscando as possíveis melhorias e boas práticas na utilização da mesma.

Para apresentar o estudo de uma pipeline CI/CD, irá ser implementada localmente em dois casos diferentes, inicialmente será construída utilizando contêineres apenas com *Docker* e posteriormente com o *Kubernetes*, este por sua vez muito mais robusto por uma série de fatores que serão evidenciados e analisados no decorrer deste documento. A estrutura básica da pipeline em ambas implementações será realizada seguindo 3 estágios automatizados, sendo eles: *Test*, *Build* e *Deploy*.

Com as estruturas formadas é observado o funcionamento das *pipelines CI/CD* em cada estágio, e o produto gerado por elas, a aplicação de demonstração implantada. E então é feito simulação de falha das hospedagens em ambas implementações, e discutidas as diferenças que são expostas por meio dessa abordagem. Adicionalmente, espera-se que após o estudo dos conceitos apresentados e a implementação, ajude na compreensão e estimule a realização de projetos mais complexos envolvendo as ideias abordadas em outras áreas e/ou ambientes, como serviços de nuvem.

Abstract

In this paper we will present concepts involving containers with a focus on the use of *Docker* and the use of the *Kubernetes* Container Orchestrator, both tools are highly used in the marketplace, especially when using the concepts of *DevOps*. With this in mind, we will observe, analyze and study one of the pillars of the *DevOps* culture, the *Cl/CD* pipelines, looking for the possible improvements and good practices in using it.

To present the study of a *CI/CD pipeline*, it will be implemented locally in two different cases, initially it will be built using containers only with *Docker* and later with *Kubernetes*, this in turn considerably more robust by a number of factors that will be evidenced and analyzed throughout this paper. Furthermore the basic structure of the pipeline in both implementations will follow 3 automated stages: *Test*, *Build* and *Deploy*.

With the structures formed, the operation of the *CI/CD pipelines* in each stage is observed, and the product generated by them, the deployed demonstration application. And then failure simulation of the hosts in both implementations is performed and the differences that are exposed through this approach are discussed. Additionally, it is hoped that after studying the concepts presented and implementing them, it will help in understanding and stimulate the realization of more complex projects involving the ideas discussed in other areas and/or environments, such as cloud services.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE CÓDIGOS FONTE

LISTA DE TERMOS E SIGLAS

API	Application Programming Interface
AWS	Amazon Web Service
CD	Continuous Deployment
CI	Continuous Integration
CLI	Command Line Interface
CPU	Central Processing Unit
DevOps	Desenvolvimento Operacional
GB	Giga Bytes
HTML	HyperText Markup Language
IP	Internet Protocol
K8s	Kubernetes
MB	Mega Bytes
SCM	Source Code Management
SO	Sistema Operacional
URL	Uniform Resource Locator

Capítulo 1

Introdução

1.1 Motivação

Com a cultura DevOps cada vez mais difundida no mercado de trabalho, as ferramentas que auxiliam esse conceito como conteinerização e *Kubernetes* estão com um destaque enorme, abrindo um novo leque inédito de possibilidades. Por exemplo, com a concepção de contêiner e as pipeline as opções são inúmeras, desde a criação de diferentes tipos VMs para aprendizado pessoal ou até mesmo para ações mais robustas como técnicas de processamento de dados e *Machine Learning*. Com isso, durante o período acadêmico houve uma certa defasagem desses assuntos por serem bastante atuais, mas que são extremamente populares no âmbito profissional. E foi com base nessa divergência que motivou a elaboração desse documento, com intuito de explorar as demais opções encontradas por estes conceitos e tecnologias.

1.2 Objetivos

O objetivo principal deste documento é realizar a implementação de uma pipeline CI/CD com todas ferramentas necessárias para simular uma esteira operacional, que pode ser utilizada em uma empresa de desenvolvimento de *software*, para isso será feita a implementação de duas maneiras diferentes. Uma utilizando a tecnologia contêiner, neste caso *Docker*, e a outra realizada com mais robustez com orquestradores de contêiner, como o *Kubernetes*. A fim de explorar qual a implementação adequa-se melhor em diferentes casos.

Por meio de uma análise prática de uma pipeline CI/CD utilizando ambas ferramentas citadas, é possível observar uma comparação de tempo de cada esteira com o modelo convencional de desenvolvimento de software. Adicionalmente, será analisado como as dois implementações respondem a falhas simuladas, a fim de explorar como cada ferramenta se comporta nesse tipo de evento. Com isso, ao final desta dissertação o objetivo será alcançado se o leitor assimilar as peculiaridades de cada implementação e entender as vantagens

e desvantagens de utilizar cada tecnologia citada previamente.

1.3 Trabalhos Relacionados

Com a popularização dos softwares de conteinerização e os orquestradores, criou-se um cenário promissor para aplicações e projetos utilizando as ferramentas citadas nesse documento. Sendo assim, apresentamos alguns desses trabalhos que ajudaram na elaboração dessa dissertação. Essa ajuda pode ser por uma inspiração ou até mesmo pela capacidade de simplificar conceitos mais robustos como o de pipeline CI/CD.

Com o trabalho [Freire 2021]traz a seguinte problemática, investigar e mostrar quais os aspetos em que o Docker Swarm, orquestrador de contêineres é melhor do que o *Kubernetes* e vice-versa, de modo a permitir escolher qual a melhor plataforma de acordo com as necessidades de cada empresa. À vista disso, elaborou diversas implementações, configurações, testes e avaliações com a finalidade de comparar o desempenho destas duas ferramentas. Por fim, o autor concluiu que o Docker Swarm é mais adequado para ambientes menores,e por sua vez, o *Kubernetes* que seria a escolha ideal para ambientes de produção com mais recursos.

No projeto [Mogallapu 2019], o autor explora o impacto criado pelos limites e as requisições da CPU durante a implementação do *Kubernetes* sobre o serviço de nuvem . Analisando o desempenho intensivo da CPU na aplicação do contêiner, o mesmo busca ajudar diversas empresas implementarem o orquestrador com uma confiabilidade maior,tendo como base os teste de estresse realizados no seu projeto.

Por fim, a dissertação [dos Santos Reis 2017] pretende introduzir a conteinerização de aplicações no processo de desenvolvimento de software das organizações, identificando os componentes necessários para criar uma pipeline de desenvolvimento de software adaptada aos conceitos de contêineres. Outra vertente explorada é a elasticidade ao nível dos nós do *cluster Kubernetes*. Ao fim , o autor percebe a capacidade do *Kubernetes* em gerir e realizar o *deployment* de aplicações conteinerizadas num ambiente distribuído, detendo mecanismos automáticos que auxiliam essas aplicações durante o seu ciclo de vida. E com relação à elasticidade, o mesmo confirma que a alocação de recursos aos clusters é feita pelo próprio software, sem a necessidade de introduzir nós manualmente.

1.4 Organização

O trabalho está organizado como segue. No capítulo 2 é exposto a virtualização e a conteinerização, esse o qual se torna cada vez mais difundido, e se fez necessário a criação de orquestradores como o *Kubernetes*. Além de introduzir o conceito de pipeline CI/CD. O capítulo 3 é caracterizado por apresentar a aplicação de demonstração que será utilizada como

faixada para aplicar os conceitos vistos no capítulo anterior sobre pipelines. Por sua vez no capítulo 4 é realizado um estudo de caso de uma pipeline utilizando duas implementações diferentes, sendo elas, a principal ferramenta de conteinerização - Docker - e o orquestrador de contêineres *Kubernetes*. No capítulo 5 serão realizados testes nas implementações e discutidos os resultados obtidos, onde serão abordadas com base nesses resultados, as vantagens e desvantagens de aplicações que utilizam somente contêineres e aquelas que utilizam orquestradores podem trazer a uma organização. Por fim, a Conclusão (capítulo 6) apresenta as considerações finais e trabalhos futuros propostos.

Capítulo 2

Fundamentação Teórica

Nesta etapa da dissertação serão apresentadas explicações sobre os principais conceitos e tecnologias usadas durante o processo de configuração e análise realizada neste documento, que será abordada nos próximos capítulos. Inicialmente vamos explorar o conceito de virtualização, esta que é uma tecnologia mais antigo mas que ainda está presente no dia a dia das empresas, e a conteinerização. Porém com a disseminação da conteinerização, diversas empresas estão optando pela sua utilização por meio de *Docker*, por exemplo, ao invés da virtualização ou em conjunto da mesma. Alguns motivos por essa troca citada serão abordados e esclarecidos a seguir.

Além disso, será explorado também o orquestrador de contêineres, *Kubernetes*, e algumas ferramentas que auxiliam na gestão de aplicações conteinerizadas. Existem uma variedade enorme destas ferramentas, porém iremos focar no *Helm* e por consequente nos *Charts*. Por fim, será introduzido e explorada teoricamente os conceitos de pipeline CI /CD . Assim ao final do capítulo espera-se que o conhecimento foi suficientemente profundo para entender e analisar a implementação realizada nesse documento.

2.1 Virtualização e Conteinerização

2.1.1 Virtualização

A virtualização é a tecnologia que permite criar uma máquina virtualizada dentro do seu computador, criando uma versão virtual de vários recursos computacionais, capazes de rodar programas e realizar tarefas. É importante ressaltar que a VM funciona de forma independente, isolada e suportando as diferentes aplicações. Inclusive é usual a utilização de SO diferentes nas máquinas virtualizadas, o lado positivo de utilizar diferentes sistemas operacionais são as diversa utilidades possíveis como: explorar o funcionamento de outro e implementar teste de segurança em SO distintos.

Outra utilização bastante explorada no mercado de trabalho é uso de VMs na nuvem, pois

possibilita a escalabilidade e o balanceamento de carga da computação [Veritas]. Entretanto, a virtualização também traz algumas dificuldades, as quais serão em sua maioria contornadas utilizando contêiner, sendo elas:

- Dependência de SO
- Nível de isolamento
- Alto consumo de computacional

2.1.2 Conteinerização

A utilização de contêineres tem se tornado cada vez mais relevantes pela sua versatilidade e leveza, a seguir elencamos alguns motivos para esta afirmação, como: as aplicações podem ser descartadas a qualquer momento e serem executadas sem precisar de dependências, os contêineres são recursos encapsulados e isolados. Já a sua leveza com relação a virtualização ocorre, pois cada contêiner não tem sistema operacional, o que torna possível centenas de contêineres numa mesma máquina, algo que não é viável utilizando máquinas virtuais por causa do grande poder computacional necessário.

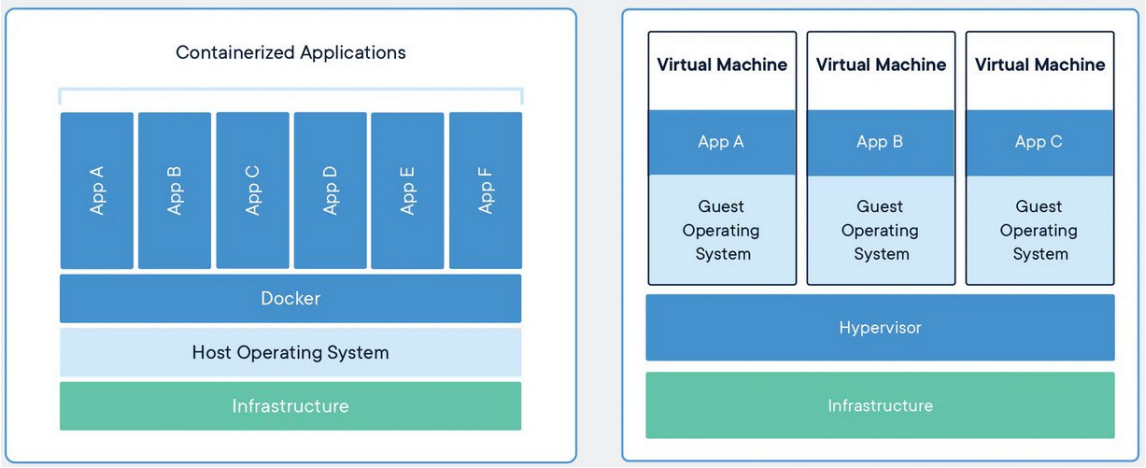


Figura 2.1: Conteinerização vs Virtualização [Docker-Inc]

2.1.3 *DevOps*

Devops, é uma combinação de pessoas, processos, práticas e ferramentas com a finalidade de melhorar a capacidade de uma empresa realizar serviços em alta velocidade. Essa união permite que funções anteriormente isoladas, desenvolvimento e operacional, atuem em conjunto para gerar produtos com uma qualidade maior na entrega, com o *DevOps* os benéficios são:

- Velocidade

- Entrega Rápida
- Confiabilidade
- Escalabilidade
- Segurança
- Melhor tempo de recuperação

Além disso, uma das boas práticas do *DevOps* são os processo de interação contínua e entrega contínua (CI/CD), auxiliam na padronização dos numerosos ambientes de uma empresa, facilitando no gerenciamento e controle sobre o ambiente e infraestrutura. Adicionalmente as principais ferramentas utilizadas no DevOps atualemte são o *Docker*, *Kubernetes*, *Github*, *Gitlab*, *Jenkins* e *OpenShift*. [AWS-Devops]

2.2 Docker

Os contêineres podem ser criados a partir de várias tecnologias, no entanto a tecnologia mais utilizada é o *Docker* [Docker], este é uma ferramenta open-source ao qual facilita a criação, implementação e execução de contêineres. Assim simplificando o processo de instalação, execução e remoção de software.

O *Docker* se difere de máquinas virtuais pois permite que processos possam ser executados de forma isolada compartilhando o mesmo sistema operacional do host, conforme ilustrado na 2.1, este isolamento ainda traz outro benefício, a organização das aplicações e suas dependências executando dentro do seu próprio contêiner.

A plataforma *Docker*, conhecida como *Docker* Engine, é uma aplicação do tipo cliente-servidor [Docker-Overview]. Por sua vez o servidor *Docker* é chamado de daemon, é responsável por ser o processo raiz de todos os outros processos a correr em todos os contêineres, além de ser responsável pela gestão dos próprios contêineres. Já o lado do cliente é feito por uma CLI que interage com o servidor daemon.

O *Docker* utiliza de imagens que são uma forma de empacotamento de software e suas dependências de maneira que possa ser movida entre vários ambientes e executada sem alterações, ou seja, pode se dizer que um contêiner é uma instância de uma imagem. Quando um contêiner é criado, este é composto por um conjunto de camadas que formam uma imagem com permissões de leitura, e uma camada adicional em cima de todas as outras com permissões de leitura e escrita, conforme evidenciado na figura 2.2. Esta camada superior assim que o contêiner é criado encontra-se vazia, dessa forma, sempre que são realizadas alterações aos fichamentos das camada abaixo estes fichamentos são copiados para a camada superior a fim registrar as alterações realizadas, Este processo de manter várias versões do mesmo fichamento distribuído pelas várias camadas é chamado de *"Copy-on-Write"*, no entanto todos

as modificações realizadas na última camada se não é utilizado nenhum tipo de mecanismo, como os volumes, os dados serão perdidos e a imagem não terá essas novas alterações.

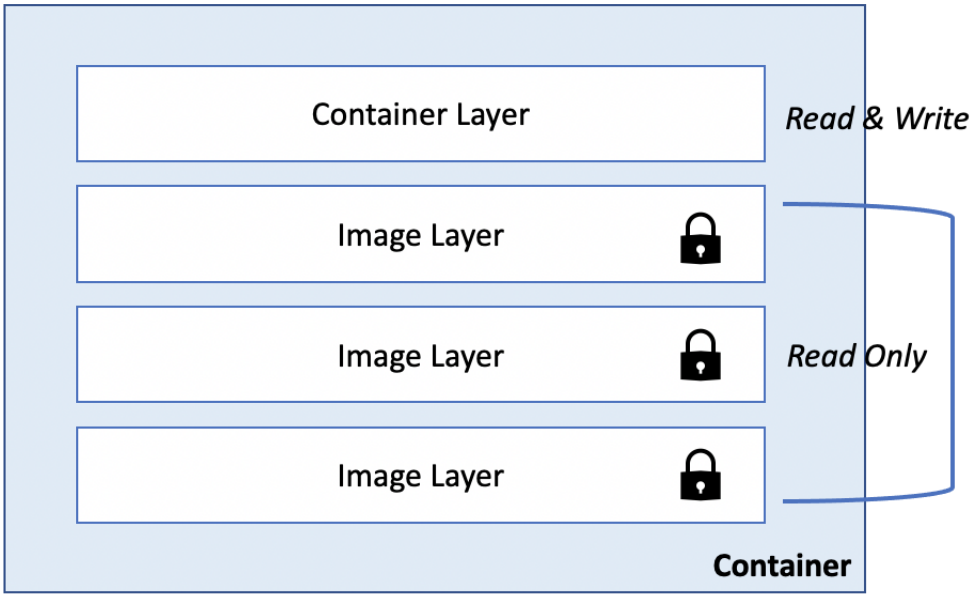


Figura 2.2: Imagem Docker com suas camadas [Rajeshsgr]

2.2.1 Docker Hub

Criado e mantido pela empresa *Docker*, o *Docker Hub* é um serviço para encontrar e compartilhar imagens de contêiner. Com diversos projetos *open source* e uma comunidade de desenvolvedores, os usuários tem acesso a repositórios públicos e privados, podendo criar os seus próprios repositórios para armazenar e compartilhar suas imagens sem custo, assim o *Docker Hub* se tornou o maior repositório de imagens de contêineres do mundo. [DockerHub]

2.3 Kubernetes

Kubernetes, também conhecido como K8s, é uma plataforma *open-source* capaz de realizar a gestão de aplicações conteinerizadas. O *Kubernetes* é capaz de automatizar e gerir vários elementos de um aplicação conteinerizada durante o seu ciclo de vida, como por exemplo, o *deployment*, descoberta de serviços, gestão de configurações, balanceamento de carga, elasticidade, atualizações, monitoração e logging [Kubernetes]

Adicionalmente o K8s pode monitorar todo o *cluster* e no caso de falha de algum contêiner é capaz de substituí-lo por outro nó do *cluster*. E em casos mais extremos é capaz de substituir todos os contêiner de um nó do *cluster* com falhas para um nó saudável. O Pod representa a menor instância de um *cluster* que você pode criar no *Kuberne-*

tes, representa um grupo de um ou mais contêineres que correm em conjunto, com ou sem volume[Kubernetes-Pod].

Os demais contêineres que executam em conjunto no mesmo pod estão num ambiente de compartilhamento de recursos e podem se comunicar entre eles via localhost ou pela própria rede do *Kubernetes*. O ciclo de vida de um pod pode os seguintes estados:

- Pendente
- Em execução
- Sucesso
- Falha (Uma das vantagens de utilizar o *Kubernetes* é caso um Pod entrar no estado de falha, o mesmo é eliminado e um novo Pod é criado em seu lugar para realizar a mesma função)
- Desconhecido

Já o conceito de Volume no *Kubernetes*, refere se ao diretório da host que é montado no Pod, segue o mesmo ciclo dos pods, porém permite a partilha de fichamento entre contêineres que pertencem ao mesmo Pod. Isto faz com que o contêiner que for reiniciado por uma falha tenha seus dados preservados [Kubernetes-Volume].

2.3.1 Minikube

Atualmente com a popularização do K8s existem diversas formas diferentes para realizar a implementação, pórem visando o aprendizado e a exploração das suas funcionalidades foi criado o *Minikube*, um *cluster* de *Kubernetes* local, com objetivo de ser uma das melhor ferramenta para desenvolvedores de aplicações [Minikube-Start]. Apesar de ser implementado localmente o Minikube utiliza-se da última versão estável do orquestrador suportando estas principais funcionalidades:

- LoadBalancer (minikube tunnel)
- Multi-cluster (minikube start -p <name>)
- NodePorts (minikube service)
- Persistent Volumes
- Dashboard (minikube dashboard)
- Container runtimes (minikube dashboard)

2.3.2 Ferramentas K8s

O orquestrador contêiner usa de vários comandos para executar diferentes tarefas, a seguir vamos explorar alguns destes comandos, a fim de sanar possíveis dúvidas na parte da implementação da pipeline utilizando *Kubernetes*.

2.3.2.1 Kubectl

Kubectl é a interface de linha de comandos que executa comandos no *cluster Kubernetes*, podem ser usados para realizar deploys de aplicações, inspecionar e controlar recursos do *cluster* e visualizar logs. [Kubernetes-Kubectl-Overview]

2.3.3 Helm Charts

Com a popularização da conteinerização e a utilização do *Kubernetes*, o controle de atulizações especificas de cada ferramenta torna-se uma tarefa inviável, porém com esse problema em foco foi criado o *Helm*, um gerenciador de pacotes que auxilia no controle de aplicações K8s, esse auxílio pode ser na instalação ou atualização das aplicações mais complexas. [Helm]

O *Helm* realiza esse gerenciamento de pacotes *charts*, que por sua vez são uma coleção de arquivos que descrevem um conjunto relacionado de recurso *Kubernetes* [Helm-Charts]. Um charts é organizado como uma coleção de arquivos dentro de um diretório, similar a figura 2.3.

```
wordpress/
  Chart.yaml          # A YAML file containing information about the chart
  LICENSE             # OPTIONAL: A plain text file containing the license for the chart
  README.md           # OPTIONAL: A human-readable README file
  values.yaml         # The default configuration values for this chart
  values.schema.json  # OPTIONAL: A JSON Schema for imposing a structure on the values.yaml file
  charts/             # A directory containing any charts upon which this chart depends.
  crds/               # Custom Resource Definitions
  templates/          # A directory of templates that, when combined with values,
                      # will generate valid Kubernetes manifest files.
  templates/NOTES.txt # OPTIONAL: A plain text file containing short usage notes
```

Figura 2.3: Estrutura de um diretório chart [Helm-Charts]

2.4 Pipelines CI/CD

Consistem em séries de etapas que devem ser realizadas para a implementação de um novo software ou versão de software, treinamento de modelos para *Machine Learning* ou

Inteligência Artificial e criação de máquinas virtuais por exemplo. As *Pipelines CI/CD* são utilizadas para acelerar o desenvolvimento e entrega de aplicações, para isso é feito monitoramento e automação no ciclo de vida das aplicações, incluindo as etapas de teste e integração, além da entrega e implementação [Red Hat]. Uma boa prática é ter uma pipeline para cada ambiente, por exemplo ambiente de desenvolvimento e produção.

Continuous Integration é um método de desenvolvimento de software em que uma equipe de desenvolvimento criam, cada integrante, uma parte do código do produto e com uma frequência alta vão realizando *commits* para um repositório de código de maneira a que este código seja integrado frequentemente com o restante código num servidor de integração contínua. Neste processo de integração é associado um conjunto de testes que automaticamente detectam incompatibilidades e erros no processo de integração fornecendo feedback ao programador com a finalidade de correção dos problemas encontrados. Conforme Cristiano [dos Santos Reis 2017] descreve, após os erros serem corrigidos é iniciado um novo processo de integração deste código.

Devido à complexidade da tarefa citada acima, as equipes de desenvolvimento passaram a usar ferramentas de gestão e versionamento de código fonte que armazenam todos os ficheiros que compõem o produto. Ferramentas como Git é um bom exemplo de gestão e versionamento de código fonte.

Outro ponto importante a ressaltar é o servidor de integração contínua responsável por detectar alterações ao código no repositório central, em adição, sempre que estas alterações ocorreram o servidor inicia um novo processo de integração deste código. Existem vários softwares capazes de realizar essa tarefa, bem como o *Jenkins*, *Argo CD* e a ferramenta que vamos explorar nesse documento será o *Gitlab*.

Ainda assim, após passar por todos os testes de integração, o produto ainda não pode ser enviado para produção, pois é preciso passar pelo ambiente que simula o ambiente real de produção, esse método é conhecido como *Continuous Delivery*.

Lianping [Chen 2015]propõe que *Continuous Delivery* é uma metodologia de desenvolvimento e entrega de software onde se mantém o produto, durante todo processo de desenvolvimento, num estado permanente de lançamento para produção. Esta pipeline recria um ambiente de estágio muito parecido com o ambiente de produção, fornecendo em todas as fases feedback às equipes de desenvolvimento, de maneira a realizarem correções e otimizações. Com os testes realizados, logo após a integração, têm um ambiente muito idêntico ao de produção, faz com que o software se encontre sempre pronto para lançamento. Ainda Segundo Lianping [Chen 2015] as pipelines de *Continuous Delivery* seguem os seguintes estágios:

- Code Commit
- Build
- Acceptance Test

- Performance Test
- Manual Test
- Production

Continuous Deployment é muito semelhante à prática de *Continuous Delivery*, a diferença aqui está no nível de automatização, pois todo o software produzido pelas equipes de desenvolvimento entra automaticamente para produção depois de totalmente testado na pipeline.

Para que este método tenha sucesso a pipeline de realização de testes automáticos tem de ser bastante eficaz. Pois caso contrário, poderia chegar ao ambiente de produção software com erros e falhas, colocando em risco tanto a qualidade do produto, assim como o interesse do cliente por esse produto. Devido a isto, os testes automáticos devem cobrir todas as fases desde os testes unitários, testes de componentes e testes de aceitação.

2.4.1 Gitlab

Para realizar a implementação de uma esteira CI/CD é necessário o auxílio de várias ferramentas, sejam essas para o gerenciamento de código ou gerenciamento da própria pipeline. Diante desse fato, escolher a melhor ferramenta que se adapta ao seu projeto pode ser um desafio, portanto durante a elaboração dessa implementação optou-se pela escolha do *Gitlab* porque além de trabalhar na parte de gestão de código (SCM) pode também atuar na automação da esteira, diferentemente do *Jenkins*, que por sua vez, só atuaria na parte da pipeline. Adicionalmente o *Gitlab*, proporciona um leque de funcionalidades como:

- Controle de versionamento integrado
- CI/CD
- Rastreamento de problemas
- Revisão do código
- Ambiente de Desenvolvimento Integrado

Além de ser altamente escalável, podendo ser hospedado localmente ou em provedor de nuvem, no entanto nas implementações realizadas nesse documento o *Gitlab* foi implementado em um contêiner, utilizando *Docker*, e em um *cluster Kubernetes*. [Gitlab]

2.4.1.1 Gitlab runners

No *Gitlab* CI/CD a aplicação responsável por executar os jobs na esteira, são os *runners*, este é um *software open source* extremamente versátil, pois pode ser implementado de formas distintas, não necessitando de linguagem específica para executar e podendo ser instalada em diversos sistemas operacionais diferentes, ou até mesmo dentro de um contêiner ou *cluster*. [Gitlab-Runner]

Com isso, os *runners* após serem instalados ainda não executam os jobs da pipeline, os mesmos precisam ser registrados para realizarem a comunicação entre a pipeline e a máquina de origem. Usualmente os jobs são executados na máquina onde o software foi instalado, mas pode-se escolher um executor diferente da máquina de origem, por exemplo, quando se instala *Gitlab Runner* no *Docker* e escolhe um executor *Docker* para cumprir as *tasks*, é referido como configuração *"Docker-in-Docker"*. Conforme a figura2.4 evidencia o processo de registro e execução.[Gitlab-Runner]

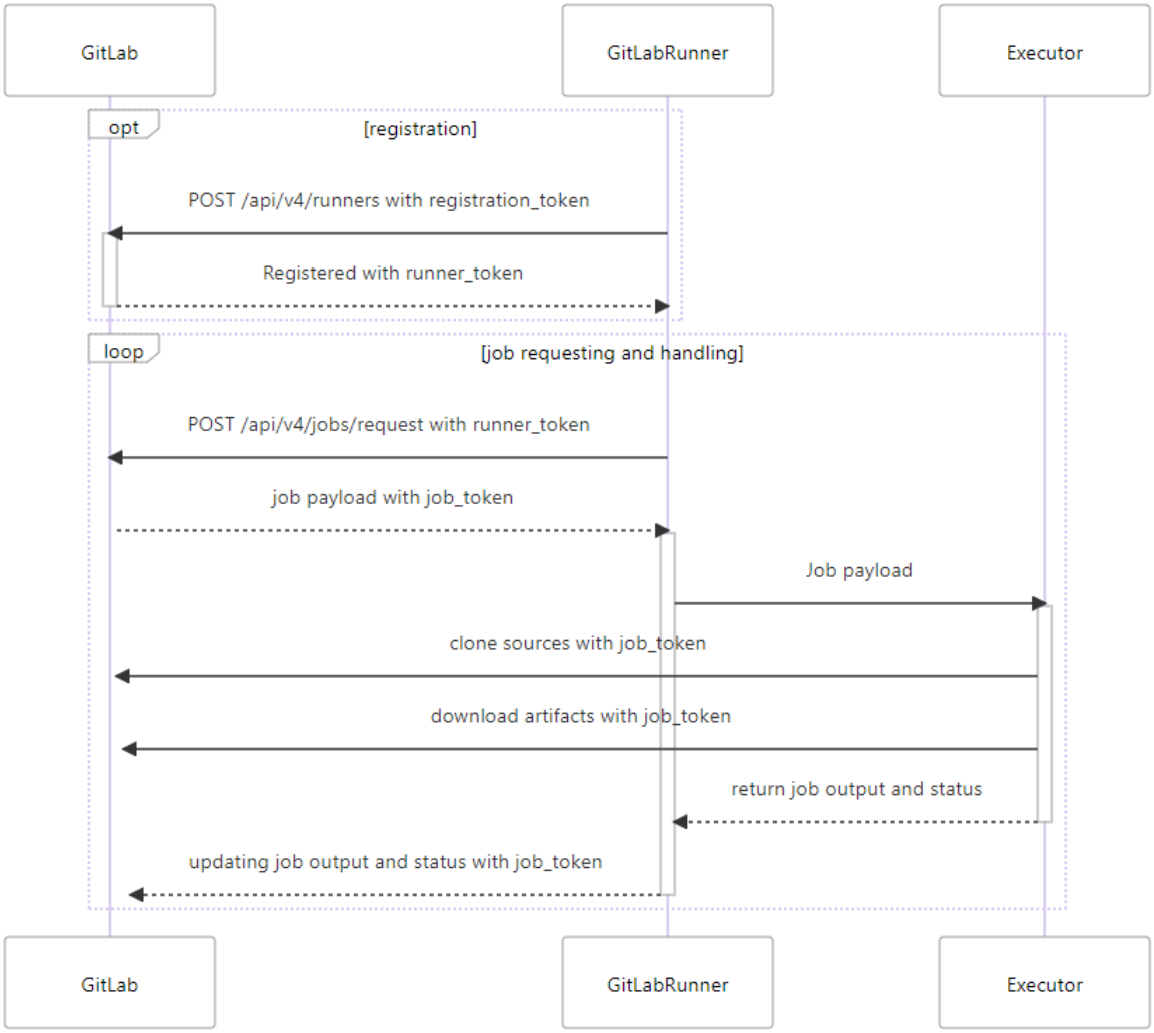


Figura 2.4: Diagrama de registro e execução de um runner [Gitlab-Runner]

Capítulo 3

Proposta de Arquitetura

Para entender os conceitos e o funcionamento de uma pipeline, é necessário um exemplo de como realizá-la, no entanto como foi abordado no capítulo 2 existem diversas ferramentas e propósitos para uma esteira. Neste capítulo irá ser apresentada a aplicação de demonstração que será utilizada nos dois casos de implementação de uma *pipeline* CI/CD no capítulo 4. Lembrando que o foco é a **pipeline CI/CD**, portanto esta é uma aplicação *web* genérica feita para ser utilizada em estudos de implantação (*deployment*). Também é importante ressaltar que a aplicação **não foi desenvolvida por nós**, então manteremos os créditos ao autor Ben Coleman [Ben-Coleman-Perfil].

3.1 Aplicação de demonstração

A aplicação escolhida é uma simples aplicação web escrita em linguagem *Python* em conjunto com o *framework Flask* e a linguagem *Javascript*, onde essa monitora a índices de performance da máquina em que está hospedada (Figura 3.1). [Python-Demoapp].

Na figura 3.2 é possível ver como essa aplicação é estruturada originalmente. Nos casos de estudo serão feitas algumas modificações em sua estrutura mas não a nível de aplicação.

Uma breve explicação das pastas da aplicação:

- **src** - Esta é a pasta onde se encontra os diretórios da aplicação *frontend*, *backend*, testes e de módulos requeridos para o *python* e o arquivo **run.py** caso queira rodar a aplicação localmente por *python*;
- **src/app** - Contém os arquivos de *backend*, arquivos estáticos de estilo, imagens, icones e de renderização com *Javascript* dos dados obtidos pelo *backend*;
- **src/templates** - arquivos de marcação de página (HTML);
- **src/tests** - arquivos de testes aos quais testam as APIs e se os caminhos das páginas estão corretos, o teste é dado como sucesso se o valor retornado seja o esperado;

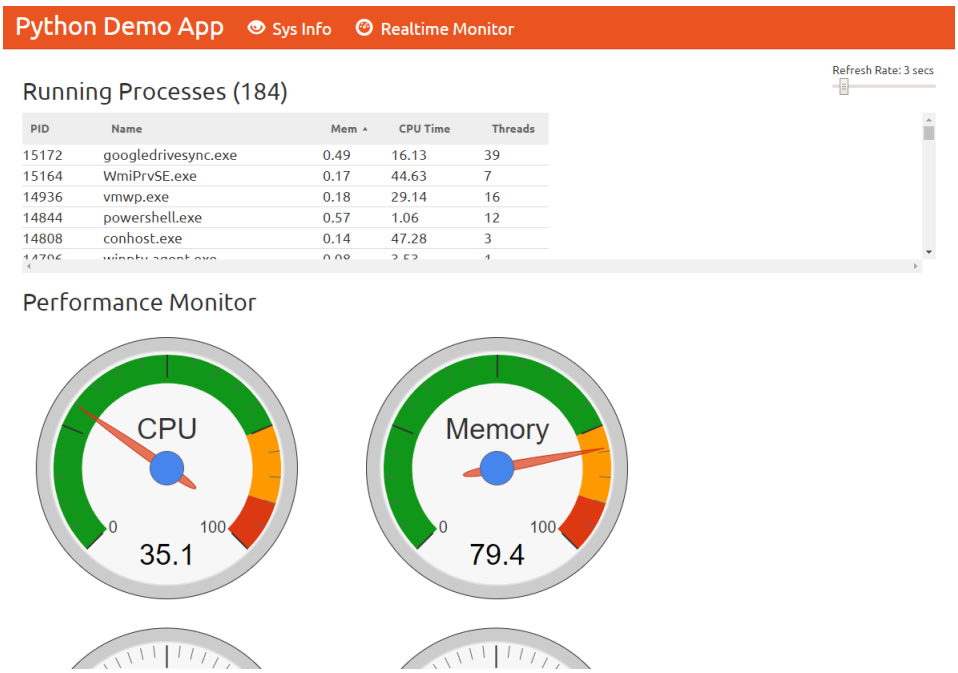


Figura 3.1: Página de monitoramento da aplicação [Python-Demoapp]

```
python-demoapp/
├── CONTRIBUTING.md
├── Dockerfile
├── LICENSE
├── makefile
├── README.md
├── src
│   ├── app
│   │   ├── apis.py
│   │   ├── conftest.py
│   │   ├── __init__.py
│   │   ├── requirements.txt
│   │   ├── static
│   │   │   ├── css
│   │   │   │   └── main.css
│   │   │   ├── img
│   │   │   │   ├── docker-whale.svg
│   │   │   │   ├── favicon.ico
│   │   │   │   ├── flask.png
│   │   │   │   ├── github-2.svg
│   │   │   │   └── python.svg
│   │   │   └── js
│   │   │       ├── monitor.js
│   │   │       └── sorttable.js
│   │   ├── templates
│   │   │   ├── base.html
│   │   │   ├── index.html
│   │   │   ├── info.html
│   │   │   └── monitor.html
│   │   ├── tests
│   │   │   ├── test_api.py
│   │   │   └── test_views.py
│   │   └── views.py
│   ├── requirements.txt
│   └── run.py
└── tests
    └── postman_collection.json

9 directories, 27 files
```

Figura 3.2: Árvore de diretórios e arquivos da aplicação de demonstração

Capítulo 4

Implementações

Neste capitulo será apresentado como uma pipeline CI/CD para desenvolvimento de software pode ser implementada utilizando o *Gitlab* como ferramenta de gerenciamento de código e automatização da esteira em dois casos diferentes:

- **Infraestrutura em Docker** - onde serão utilizados apenas contêineres para a implementar tanto a pipeline quanto o produto resultantes desta.
- **Infraestrutura em Kubernetes** - para se obter um produto mais robusto podendo gerar escalabilidade e alta disponibilidade.

4.1 Implementação utilizando Docker

A implementação da pipeline baseada em contêineres é feita utilizando o *Docker*, após instalado [Get-Docker], na Listagem 4.1 é possível observar na linha 1 que utilizaremos a imagem mais recente do *GitLab Community Edition* para executar o contêiner, mas antes desse passo é criada uma variável de ambiente para indicar onde os dados do contêiner irão ser armazenados. Em seguida, na linha 5, o contêiner é executado com uma série de opções como rodar em segundo plano, com endereço resolvido localmente, utilizando as portas necessárias, dando um nome para o contêiner, utilizando uma política de reiniciar o contêiner caso falhe, são criados volumes com a variável de ambiente para indicar o caminho de onde armazenar os dados e por último é passado o nome da imagem em questão.

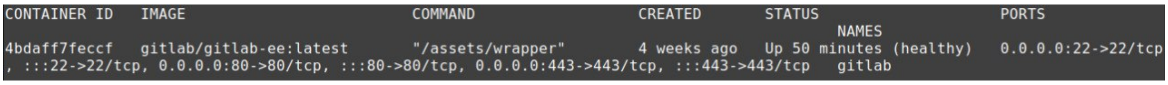
Listagem 4.1: Executando contêiner Gitlab

```
1 $ docker pull gitlab/gitlab-ce:latest
2
3 $ export GITLAB_HOME=/srv/gitlab
4
5 $ sudo docker run --detach \
6     --hostname gitlab.example.com \
7     --publish 443:443 --publish 80:80 --publish 22:22 \
```

```
    --name gitlab \
    --restart always \
    --volume $GITLAB_HOME/config:/etc/gitlab \
    --volume $GITLAB_HOME/logs:/var/log/gitlab \
    --volume $GITLAB_HOME/data:/var/opt/gitlab \
    gitlab/gitlab-ce:latest
```

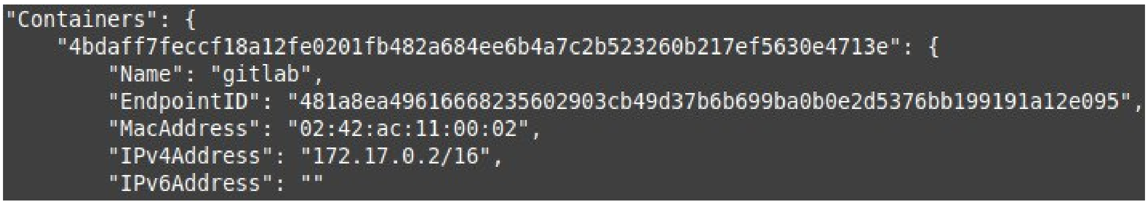
Com o comando `docker container ls`, Figura 4.1 é possível conferir o *status* do contêiner e este com estado de *Up* podemos acessar o *Gitlab* por meio do navegador de duas formas pelo *localhost* ou utilizando o IP da rede Docker.

Por padrão do *Docker*, o driver de rede *bridge* é o qual conecta contêineres na rede interna de mesmo nome (*bridge*) com a máquina principal, mas isola os contêineres que estão outras redes *bridge* manualmente criadas [Docker-Bridge]. Para saber qual é o IP da instância do *Gitlab* em questão, basta utilizar o comando `docker network inspect bridge`, onde apresenta todos os contêineres que estão conectados nessa rede e observar o campo "IPv4Address", figura 4.2.



```
CONTAINER ID   IMAGE                       COMMAND              CREATED       STATUS                    PORTS
                                                                                        NAMES
4bdaff7feccf   gitlab/gitlab-ee:latest     "/assets/wrapper"    4 weeks ago   Up 50 minutes (healthy)   0.0.0.0:22->22/tcp
, :::22->22/tcp, 0.0.0.0:80->80/tcp, :::80->80/tcp, 0.0.0.0:443->443/tcp, :::443->443/tcp   gitlab
```

Figura 4.1: Saída do comando *docker container ls*



```
"Containers": {
    "4bdaff7feccf18a12fe0201fb482a684ee6b4a7c2b523260b217ef5630e4713e": {
        "Name": "gitlab",
        "EndpointID": "481a8ea49616668235602903cb49d37b6b699ba0b0e2d5376bb199191a12e095",
        "MacAddress": "02:42:ac:11:00:02",
        "IPv4Address": "172.17.0.2/16",
        "IPv6Address": ""
```

Figura 4.2: Saída do comando *docker network inspect bridge*

Obtendo o IP da instância em mãos, no navegador digitamos `172.17.0.2:80` e assim o *Gitlab* é acessado pela primeira vez, figura 4.3.

É necessário a senha do usuário *root*, para efetuar o *login*. Caso seja de interesse do administrador é possível trocar a senha do usuário por meio do comando explicitado na linha 5, Listagem 4.2).

Listagem 4.2: Obtendo senha inicial do usuário *root* - Infraestrutura Docker

```
1 #Senha inicial do usuario root
2 $ docker exec -it gitlab grep 'Password:' /etc/gitlab/
      initial_root_password
3
4 #Resetar a senha do usuario root a sua escolha
5 $ docker exec -it gitlab gitlab-rake "gitlab:password:
      reset[root]"
```

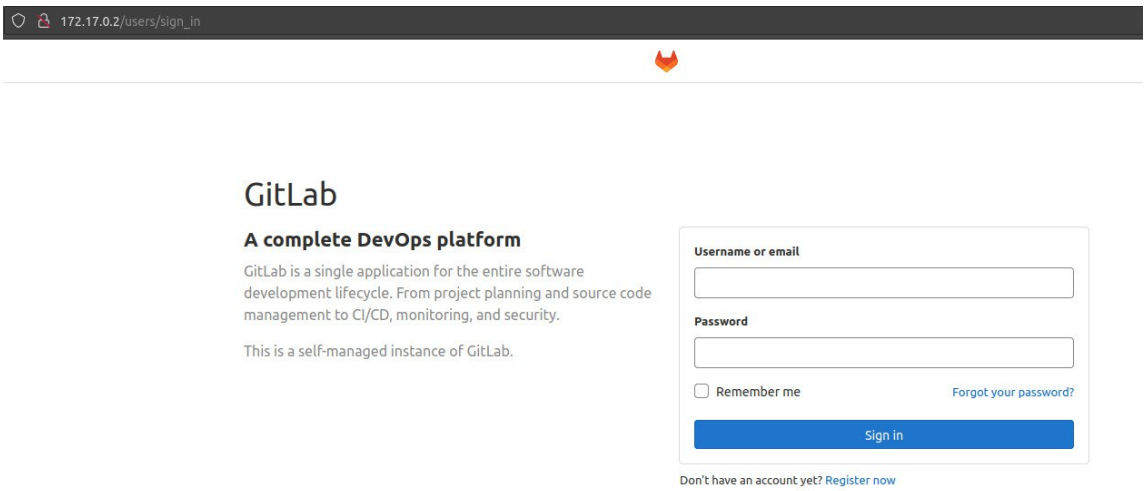


Figura 4.3: Página de *login* Gitlab

Ao realizar o *login* (Figura 4.4), temos um projeto de exemplo padrão do *Gitlab*. O próximo passo é importar o projeto da aplicação de demonstração (Capítulo 3). Para isso deve-se clicar em *New Project* → *Import Project*, iremos optar por importar utilizando a URL (Figura 4.5) de um repositório, neste caso o repositório original da aplicação demonstração [Python-Demoapp] e criar projeto.

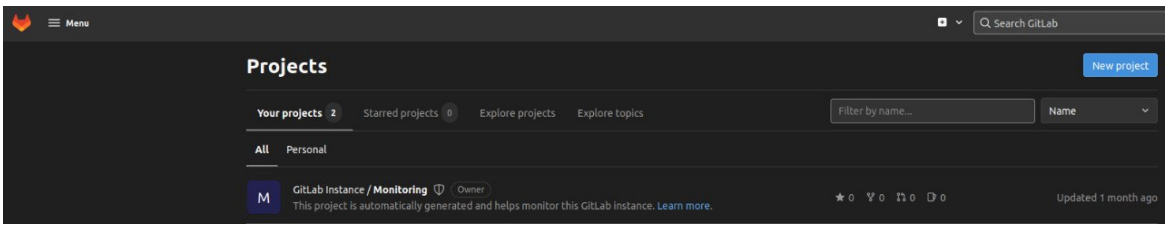


Figura 4.4: Página inicial

Assim temos o que é necessário para começar a estrutura da nossa pipeline CI/CD.

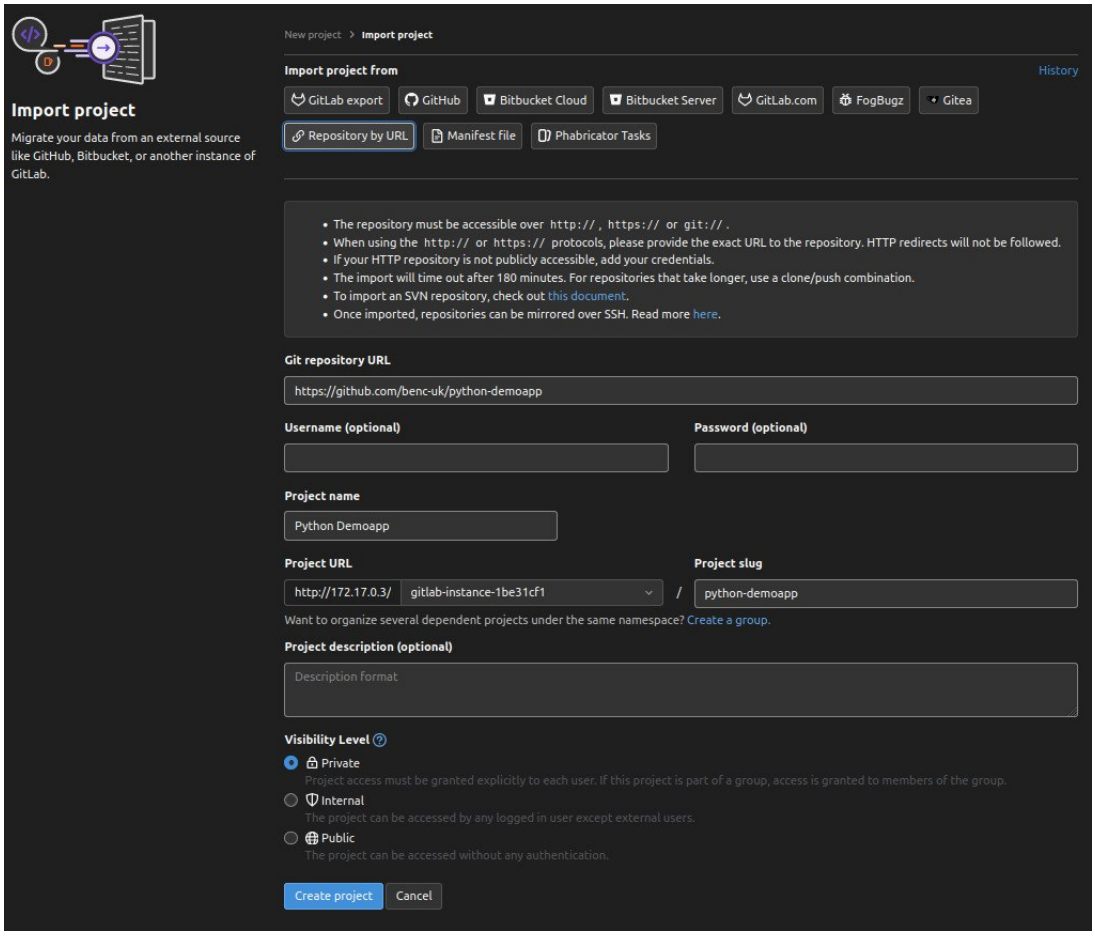


Figura 4.5: Importar projetos utilizando a URL Gitlab

4.1.1 Pipeline CI/CD

O diagrama mostrado na Figura 4.6, ilustra como a pipeline está dividida. O desenvolvedor realiza alguma alteração no código, assim que o *Gitlab* percebe essa alteração a pipeline é iniciada, realizando os testes, tanto unitários quanto de integração, então é passado para o próximo estágio, onde a imagem da aplicação é construída a partir do `Dockerfile` (6.1) e armazenada no repositório criado para armazenar a imagem no *Docker Hub*, o *Deploy* da imagem é feito no último passo da pipeline, este é ativado manualmente, para garantir que não tenha algum erro que passou despercebido. Se em algum dos estágios der erro ou falha, a pipeline é interrompida.

A pipeline no *Gitlab* é composta de um único arquivo, o `.gitlab-ci.yml` (Listagem 6.2), localizado na raíz do projeto, com este é possível definir variáveis, *jobs* ou serviços, e nestes os estágios da pipeline, para que estas tarefas sejam executadas na ordem que foram definidos, suas *tags* de marcação, imagens que irão ser utilizadas e o *script* para ser executado pelo *job*. E após qualquer alteração no projeto começa a rodar a esteira, caso não tenha configuração para atuação manual.

Inicialmente são definidas as variáveis globais que podem ser utilizadas em um ou mais estágios, neste caso o nome e *tag* da imagem que irá ser construída e executada (Listagem 4.3). Na linha 3, vemos que *IMAGE_TAG* está utilizando uma variável nativa do *Gitlab* que armazena o identificador da pipeline, é feito dessa forma para versionar a aplicação. Variáveis com informações sensíveis são armazenadas nas configurações do projeto (Figura 4.7). Também são definidos os estágios que definem a ordem que os *jobs* serão executado. Nesta pipeline um estágio corresponde ao *job* de mesmo nome que será executado pelo ou pelos *Gitlab Runners*, mas podemos ter vários *jobs* por estágio.

Listagem 4.3: .gitlab-ci.yml variáveis e estágios - Infraestrutura Docker

```
1 variables:
2     IMAGE_NAME: romuloos7899/python_demo_app
3     IMAGE_TAG: ${CI_PIPELINE_ID}
4
5 stages:
6     - test
7     - build
8     - deploy
```

O *job test* é simples (Listagem 4.4), definimos que este job faz parte do estágio de testes, marcamos com a *tag test* para deixar indicado ao *runner*. Como os testes são escritos em *Python* utilizamos a imagem indicada na linha 5, para executar os testes, na linha 6 temos um campo para ser executado antes do *script* de testes para instalar dependências e por último a execução do *script* de testes.

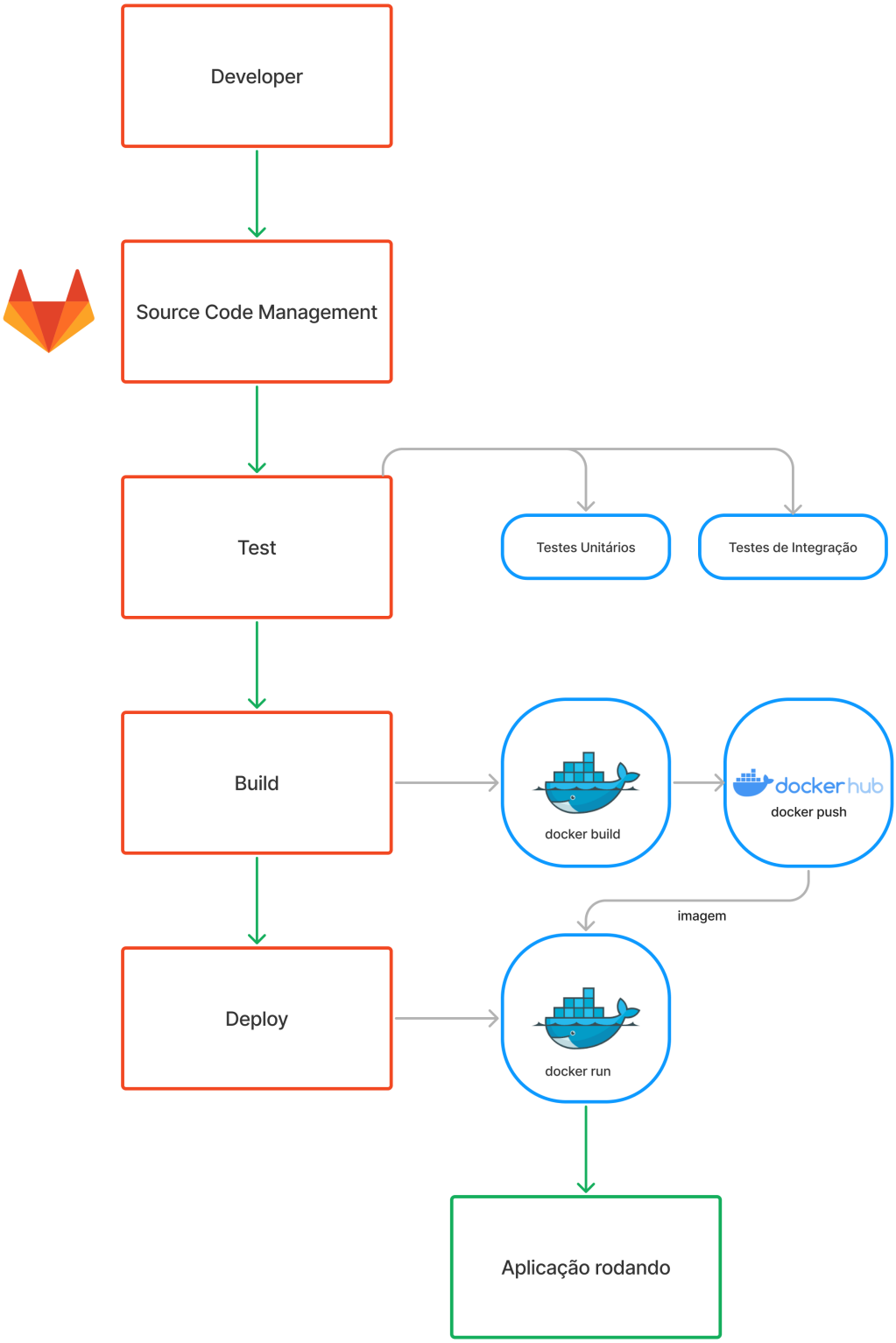


Figura 4.6: Diagrama pipeline CI/CD em infraestrutura *Docker*

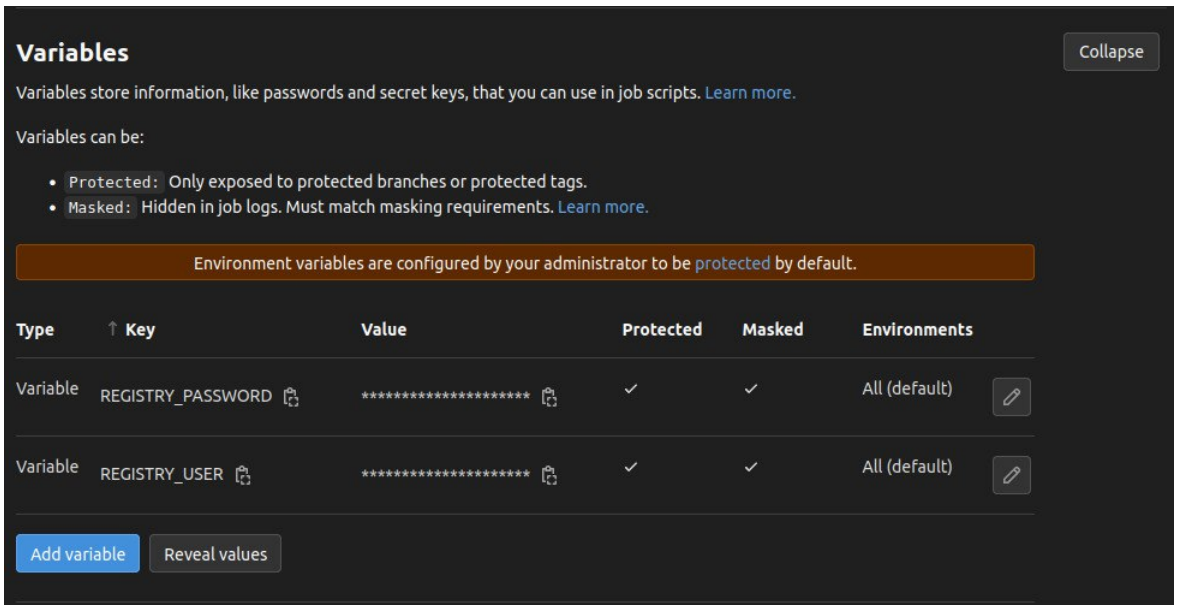


Figura 4.7: Variaveis sensíveis protegidas

Listagem 4.4: .gitlab-ci.yml estágio *test* - Infraestrutura Docker

```
1  test:
2      stage: test
3      tags:
4          - test
5      image: python:3.9-slim-buster
6      before_script:
7          - apt-get update && apt-get install make
8      script:
9          - make test
10         - echo Fim dos Testes
```

O *job build* (Listagem 4.5) é definido pelo estágio e *tag* de *build*, neste necessitamos de uma imagem com o *Docker* instalado para realizar este passo, foi utilizada a imagem *Docker* em questão na linha 5, nesses casos é necessário criar um *service* e utilizar a variável do *Docker*, pois o *runner* executa um contêiner em cima de um contêiner secundário, onde esse não possui os certificados de permissões, e consequentemente sem o caminho dos certificados não consegue realizar os *scripts*, pois não tem permissão para isso. Em seguida, na linha 11, é efetuado o *login* no *Docker Hub* utilizando as variáveis mostradas na figura 4.7. Para finalizar a imagem é construída e armazenada no repositório de imagens, *Docker Hub*.

Listagem 4.5: .gitlab-ci.yml estágio de *build* - Infraestrutura Docker

```
1  build:
2      stage: build
3      tags:
4          - build
5      image: docker:20.10.17-dind
6      services:
7          - docker:20.10.17-dind
```

```
8       variables:
9           DOCKER_TLS_CERTDIR: "/certs"
10      before_script:
11          - docker login -u $REGISTRY_USER -p
                $REGISTRY_PASSWORD
12      script:
13          - docker build -t $IMAGE_NAME:$IMAGE_TAG .
14          - docker push $IMAGE_NAME:$IMAGE_TAG
```

No último *job* da pipeline (Listagem 4.6), são definidos o estágio e a marcação de *deploy*. Assim como no passo anterior também é utilizada a imagem, serviços e variável local do *Docker*. No *script* é utilizado de comandos para listar os contêineres filtrando com o nome do contêiner que queremos, filtramos a saída novamente para pegar apenas o identificador do contêiner, então este é parado e removido. Logo em seguida outro contêiner é executado em seu lugar com uma versão mais recente da aplicação na porta escolhida. O *job* de *deploy* é apenas realizado manualmente, para ter certeza que não teve nenhum erro, manual ou da aplicação.

Listagem 4.6: .gitlab-ci.yml estágio de *deploy*

```
1  deploy:
2      stage: deploy
3      tags:
4          - deploy
5      image: docker:20.10.17-dind
6      services:
7          - docker:20.10.17-dind
8      variables:
9          DOCKER_TLS_CERTDIR: "/certs"
10     script:
11         - docker container ls | grep
               python_demo_app_docker | awk '{print $1}' |
               xargs docker stop | xargs docker rm
12         - docker run --name python_demo_app_docker -d -p
               5000:5000 $IMAGE_NAME:$IMAGE_TAG
13     when: manual
```

4.1.2 Cadastrando Gitlab Runners

Para que a pipeline rode no *Gitlab* é necessário cadastrar os *runners* em sua instância. Existem diversas maneiras de utilizá-los, iremos optar executar a aplicação também por meio do *Docker*. Primeiramente é necessário criar um volume para o contêiner do *Gitlab Runner*, em seguida é executado a aplicação utilizando a imagem mais recente (Listagem 4.7).

Listagem 4.7: Cadastro de Gitlab Runner - Infraestrutura Docker

```
1  #Criando volume para o runner
```

```
2 docker volume create gitlab-runner-config
3
4 #Executando a versao mais recente da imagem do Gitlab
      Runner
5 docker run -d --name gitlab-runner --restart always \
6     -v /var/run/docker.sock:/var/run/docker.sock \
7     -v gitlab-runner-config:/etc/gitlab-runner \
8     gitlab/gitlab-runner:latest
```

Na figura 4.8, vemos o processo de registro do *runner* de *build*, onde é executado o comando para dentro do contêiner em execução do *Gitalb Runner*, `docker run --rm -it -v gitlab-runner-config:/etc/gitlab-runner gitlab/gitlab-runner register`. Neste é pedido a URL da instância, o *token* de registro (Figura 4.9), a descrição ou nome do *runner*, a *tag* para que este saiba quais *jobs* devem ser executados, o executor e a imagem são utilizados o *Docker* e sua imagem mais estável `docker:stable`.

```
[redacted]:~$ docker run --rm -it -v gitlab-runner-config:/etc/gitlab-runner gitlab/gitlab-runner register
Runtime platform                                    arch=amd64 os=linux pid=7 revision=32fc1585 version=15.2.1
Running in system-mode.

Enter the GitLab instance URL (for example, https://gitlab.com/):
http://172.17.0.2:80
Enter the registration token:
AzQvz8gYBp1Cz4V_eND8
Enter a description for the runner:
[a1b716712b04]: build-runner
Enter tags for the runner (comma-separated):
build
Enter optional maintenance note for the runner:

Registering runner... succeeded                     runner=AzQvz8gY
Enter an executor: ssh, custom, parallels, shell, docker+machine, docker-ssh+machine, kubernetes, docker, docker-ssh, virtualbox:
docker
Enter the default Docker image (for example, ruby:2.7):
docker:stable
Runner registered successfully. Feel free to start it, but if it's running already the config should be automatically reloaded!

Configuration (with the authentication token) was saved in "/etc/gitlab-runner/config.toml"
```

Figura 4.8: Registrando Gitlab Runner na instância Gitlab

Foram configurados 3 *Gitlab Runners*, um para cada estágio da pipeline (*tests, build, deploy*), como podemos ver na figura 4.9. Ao registrar cada *runner* é adicionado suas configurações ao arquivo de configuração do *Gitlab Runner* que pode ser visto na Listagem 6.3.

4.2 Implementação utilizando Kubernetes

A criação e utilização de *clusters Kubernetes* requer certo poderio computacional, por isso ferramentas para desenvolvimento e estudos locais, como Minikube, Kind e K3s, foram desenvolvidas possibilitando a difusão do orquestrador para mais pessoas.

Para iniciar a implementação da pipeline utilizando o *Kubernetes*, é necessário instalar os artifícios e as ferramentas que iremos utilizar para trabalhar com o orquestrador. São elas:

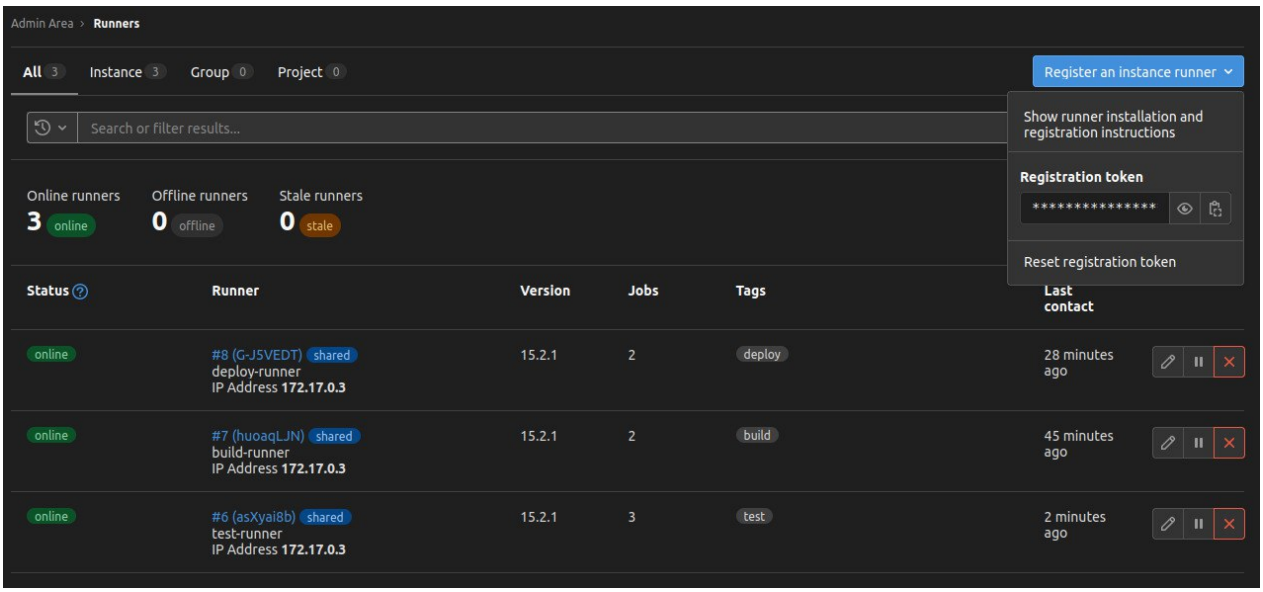


Figura 4.9: Página de administração dos Runners

- Docker [Get-Docker]
- Minikube [Minikube]
- kubectl [Kubectl-Instalation]
- Helm [Installing-Helm-Charts]

O *Docker* é utilizado como *driver* do Minikube, poderia ser utilizado qualquer um dos *drivers* disponíveis (*Docker, Hyperkit, Hyper-V, KVM, Parallels, Podman, VirtualBox, or VMware Fusion/Workstation*), que o utiliza para a criação do *cluster* K8s. Na Listagem 4.8, iniciamos o Minikube utilizando 4 CPUs e 8 GB de memória. O ideal seria 8 CPU e 30 GB, por isso utilizaremos as configurações mínimas. Na figura 4.10 nos mostra o comando e a saída deste.

Listagem 4.8: Iniciando Minikube com *Docker driver*

```
1 minikube start \
2   --driver=docker \
3   --cpus 4 \
4   --memory 8192
```

Em seguida adicionamos o complemento [Minikube-Addons] para o Minikube, o *ingress* este gerencia os acessos externos aos serviços.

Listagem 4.9: Adicionando complementos ao Minikube

```
1 minikube addons enable ingress
```

Assim temos as configurações necessárias para executar o *GitLab* utilizando o *Helm*. Primeiramente adicionamos o repositório oficial do *Gitlab* [Gitlab-Helm-Charts] e atualizamos

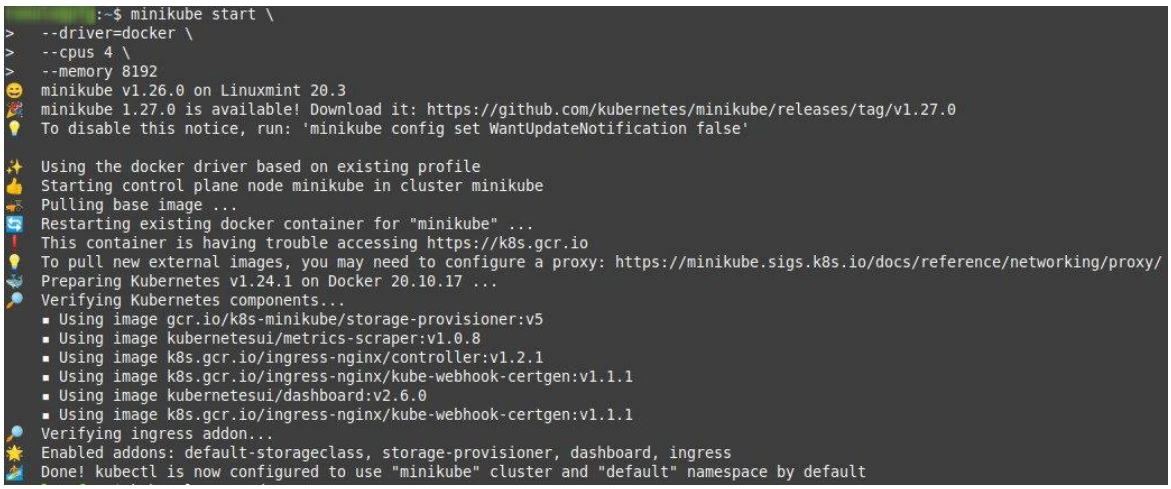


Figura 4.10: Iniciando o cluster Kubernetes com Minikube

a lista de repositórios do *Helm* (Listagem 4.10). Por fim é instalada a aplicação do *Gitlab*, definido o endereço para acessar utilizando o IP do Minikube (`minikube ip`) e utilizamos o arquivo de valores mínimos [Gitlab-Minimum-Values] para o *chart* do *Gitlab*(Listagem 4.11).

Listagem 4.10: Adicionando repositório de *charts* do *Gitlab* e atualizando os repositórios

```
1 helm repo add gitlab https://charts.gitlab.io/
2 helm repo update
```

Listagem 4.11: Instalando os *charts* do Gitlab

```
1 helm upgrade --install gitlab gitlab/gitlab \
2     --timeout 600s    \
3     --set global.hosts.domain=$(minikube ip).nip.io \
4     --set global.hosts.externalIP=$(minikube ip) \
5     -f https://gitlab.com/gitlab-org/charts/gitlab/raw/
          master/examples/values-minikube-minimum.yaml
```

Com isso podemos observar os status dos *pods* utilizados pelo *Gitlab* (Figura 4.11), é possível acessar a instância apenas quando todos estiverem com *status Completed* ou *Running*, pois cada um realiza sua função para que a instância esteja sendo executada corretamente, é normal reiniciar algumas vezes antes de atingir os *status* desejados.

Utilizando o endereço do *Gitlab* obtido podemos acessá-lo pelo navegador `https://gitlab.192.168.49.2.nip.io/` (Figura 4.12), para efetuar o login utilizamos o comando referenciado na Listagem 4.12 para obter a senha inicial para o usuário *root*.

Listagem 4.12: Obtendo senha inicial do usuário *root* - Infraestrutura Kubernetes

```
1 kubectl get secret gitlab-gitlab-initial-root-password -
      ojsonpath='{.data.password}' | base64 --decode ; echo
```

```
[redacted]:~$ kubectl get pods
NAME                                      READY   STATUS      RESTARTS       AGE
gitlab-gitaly-0                           1/1     Running     10 (72m ago)   13d
gitlab-gitlab-exporter-69765c6d8c-mg4pb   1/1     Running     6 (82m ago)    13d
gitlab-gitlab-shell-7cb457cccc-hlls4      1/1     Running     7 (82m ago)    13d
gitlab-kas-54ffd8b4d4-fwbqs               1/1     Running     6 (82m ago)    13d
gitlab-kas-54ffd8b4d4-j592r               1/1     Running     6 (82m ago)    13d
gitlab-migrations-1-kmfs9                 0/1     Completed   0              13d
gitlab-minio-74dfc6b6c7-64sll             1/1     Running     6 (82m ago)    13d
gitlab-minio-create-buckets-1-j9p5p       0/1     Completed   0              13d
gitlab-postgresql-0                       2/2     Running     19 (71m ago)   13d
gitlab-redis-master-0                     2/2     Running     12 (82m ago)   13d
gitlab-registry-65c8fcbf58-jd52b          1/1     Running     6 (82m ago)    13d
gitlab-runner-5f5b69cf94-l8q59            1/1     Running     12 (73m ago)   13d
gitlab-sidekiq-all-in-1-v2-84d78c959-vtxhg   1/1     Running     6 (82m ago)    13d
gitlab-toolbox-68c74587cf-c5z25           1/1     Running     6 (82m ago)    13d
gitlab-webservice-default-7785b7f775-r7zj7   2/2     Running     12 (82m ago)   13d
```

Figura 4.11: Verificando *status* dos *pods*

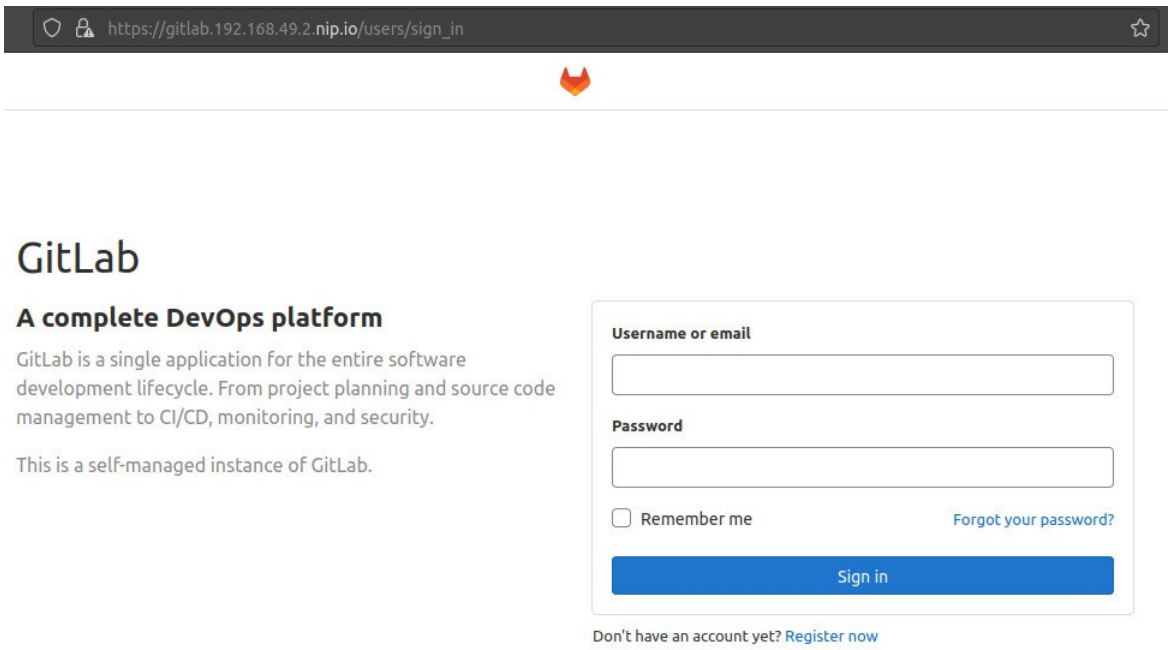


Figura 4.12: Página de *login* Gitlab

Assim como na implementação anterior (seção 4.1), ao realizar o *login* temos um projeto de exemplo padrão do Gitlab 4.13 e temos que importar o projeto da aplicação de demonstração. Para este caso utilizaremos o *Gitlab export* do projeto, por meio da implantação utilizando Docker 4.1 (Figura 4.14) e importaremos o arquivo gerado no *Gitlab* desta implantação (Figura 4.15).

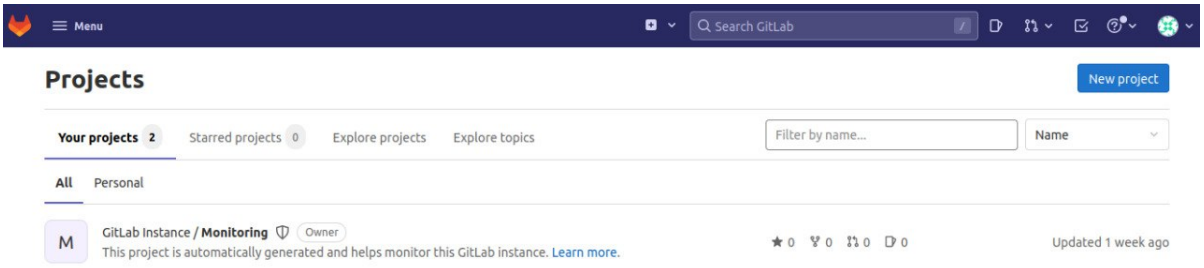


Figura 4.13: Página inicial

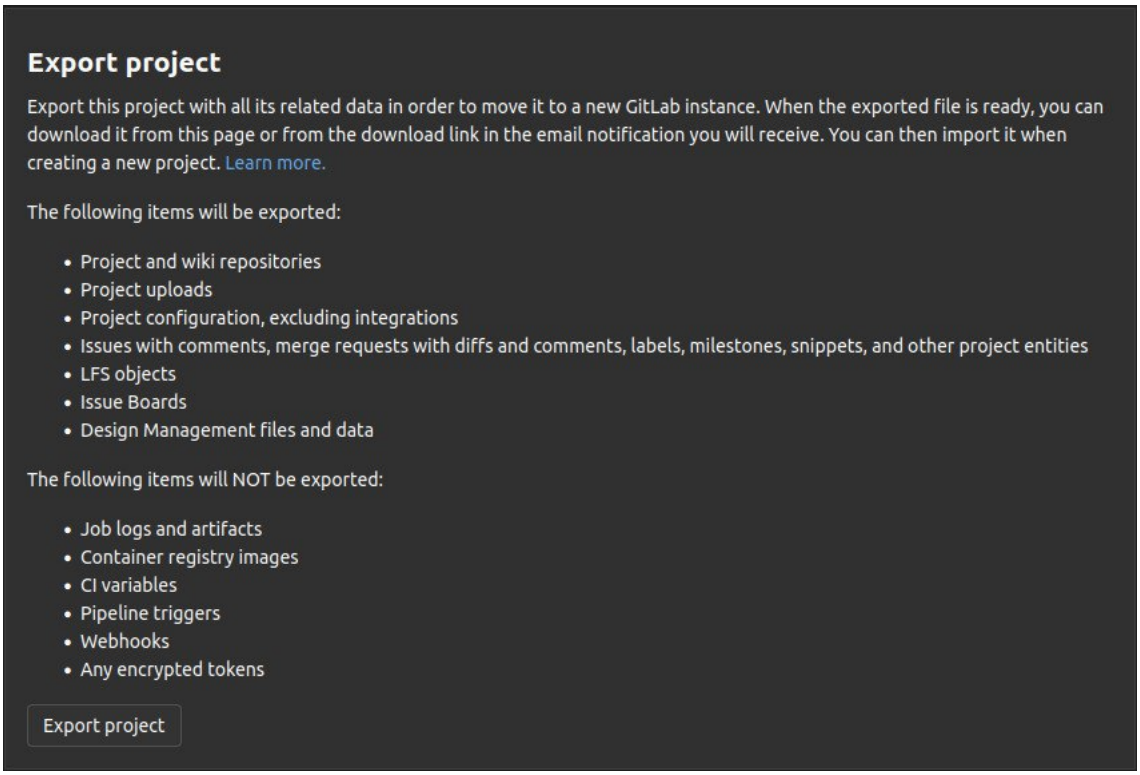


Figura 4.14: Exportando projeto da aplicação

4.2.1 Criação do Chart da aplicação

Com o projeto importado, é necessário criar algumas pastas e arquivos para implementar o *deploy* da aplicação utilizando o *Helm Chart*, que nada mais é que uma coleção de arquivos que descrevem recursos e serviços relacionados ao Kubernetes, neste caso para uma aplicação web. A estrutura do projeto fica evidenciado na figura 4.16.

Os arquivos do Chart da aplicação são:

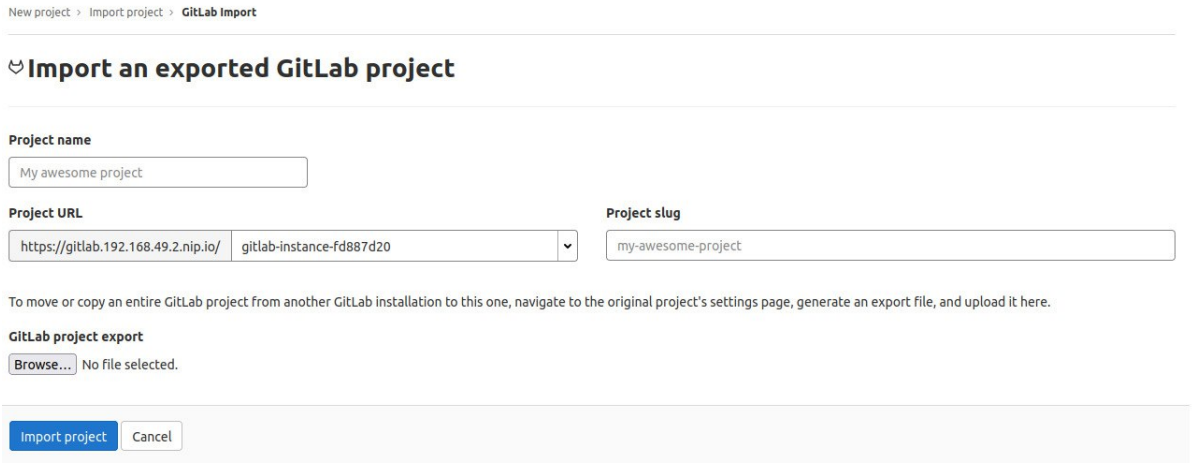


Figura 4.15: Importando projeto da aplicação no Gitlab hospedado no *cluster* Kubernetes

```
python-demoapp/
├── CONTRIBUTING.md
├── deploy
│   └── kubernetes
│       └── helm
│           ├── Chart.yaml
│           ├── templates
│           │   ├── deployment.yaml
│           │   ├── _helpers.tpl
│           │   ├── NOTES.txt
│           │   └── service.yaml
│           └── values.yaml
├── Dockerfile
├── LICENSE
├── makefile
├── README.md
├── src
│   ├── app
│   │   ├── apis.py
│   │   ├── conftest.py
│   │   ├── __init__.py
│   │   ├── requirements.txt
│   │   ├── static
│   │   │   ├── css
│   │   │   │   └── main.css
│   │   │   ├── img
│   │   │   │   ├── docker-whale.svg
│   │   │   │   ├── favicon.ico
│   │   │   │   ├── flask.png
│   │   │   │   ├── github-2.svg
│   │   │   │   └── python.svg
│   │   │   └── js
│   │   │       ├── monitor.js
│   │   │       └── sorttable.js
│   │   ├── templates
│   │   │   ├── base.html
│   │   │   ├── index.html
│   │   │   ├── info.html
│   │   │   └── monitor.html
│   │   ├── tests
│   │   │   ├── test_api.py
│   │   │   └── test_views.py
│   │   └── views.py
│   ├── requirements.txt
│   └── run.py
└── tests
    └── postman_collection.json

13 directories, 33 files
```

Figura 4.16: Árvore de diretórios e arquivos da aplicação de demonstração

- **Chart.yaml**:

  Este arquivo contém as informações sobre o próprio *chart*, como nome, descrição e versão. Listagem 6.4.

- **values.yaml**:

  Arquivo de configuração do *chart*, onde é definido qual repositório da imagem, a porta utilizada pro serviço e o tipo do serviço que é necessário para definir e encaminhar as requisições dessas portas dentro do *cluster*. Listagem 6.5.

- **deployment.yaml**:

  Este usa dos valores recebidos nos arquivos de *values* e *chart* para gerar o arquivo de manifesto para realizar o *deployment* no *cluster Kubernetes*. Listagem 6.6.

- **service.yaml**:

  Este usa dos valores recebidos nos arquivos de *values* e *chart* para gerar o arquivo de manifesto e iniciar o serviço que será aplicado o *deploy* no *cluster Kubernetes*. Listagem 6.7.

Existem outros *templates* que podem ser utilizados como o *hpa.yaml* (Horizontal Pod Autoscaler), este que gera o manifesto para escalar a quantidade de *pods* caso tenha muitos acessos, por exemplo. No entanto requer maior poder computacional para utilizá-lo.

4.2.2 Pipeline CI/CD

O diagrama da pipeline CI/CD na figura 4.17 é bastante parecido com o que tínhamos no caso anterior, figura 4.6. A maior diferença está no *deploy* da aplicação, os estágios de *test* e *build* são os mesmos (Listagens 4.4 e 4.5 respectivamente) e também as variáveis sensíveis 4.7 utilizadas para realizar o *login* no *Docker Hub*

Assim como feito na seção 4.1.1 a configuração do arquivo `.gitlab-ci.yml` (Listagem 6.8) dessa implementação começa definindo as variáveis utilizadas para a imagem da aplicação, mas neste caso o *Helm* necessita que a versão do *Chart* seja sempre atualizada ao realizar o *deploy*, caso contrário, não é entendido que houve uma atualização. Portanto a *tag* da imagem da aplicação (para essa implantação chamada de *APP_VERSION*) foi feita com a junção de duas variáveis internas do *Gitlab*, o nome da *branch* com a identificação da pipeline (Listagem 4.13). E em seguida os estágios sendo definidos.

Listagem 4.13: .gitlab-ci.yml variáveis e estágios - Infraestrutura Kubernetes

```
1 variables:
2     IMAGE_NAME: romuloallc/python_demo_app
3     APP_VERSION: ${CI_COMMIT_REF_SLUG}_${CI_PIPELINE_ID}
4
5 stages:
```

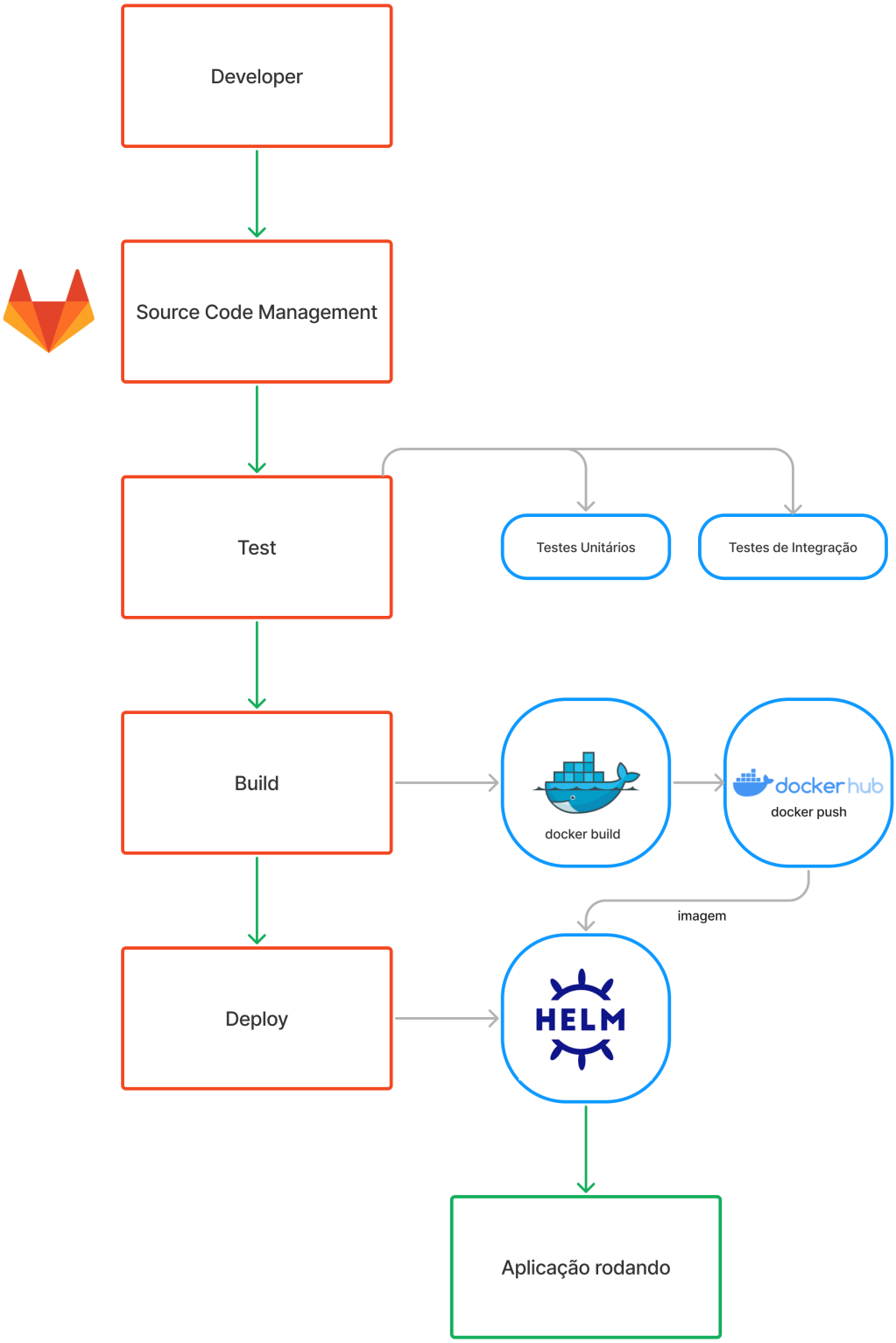


Figura 4.17: Diagrama pipeline CI/CD em infraestrutura Kubernetes

```
6       - test
7       - build
8       - deploy
```

O *job deploy* (Listagem 4.14), feito no estágio de mesmo nome, utiliza da imagem *dtzar/helm-kubectl:3.6.2*, onde esta é uma imagem onde temos o *Helm* e o *kubectl* instalados para serem utilizados. Seguindo para o *script* temos o `sed`, um comando bastante útil para manipulação de texto, este identifica no destino, *Chart.yaml* (Listagem 6.4), a marcação `{APP_VERSION}` e substitui pelo valor da variável `APP_VERSION` e então, na linha 8 instalamos nosso *chart* de nome *python-demo* passando o arquivo de valores previamente mostrado.

Listagem 4.14: .gitlab-ci.yml estágio de deploy - Infraestrutura Kubernetes

```
1  deploy:
2      stage: deploy
3      tags:
4          - deploy
5      image: dtzar/helm-kubectl:3.6.2
6      script:
7          - sed -i "s/{APP_VERSION}/${APP_VERSION}/g" ./
               deploy/kubernetes/helm/Chart.yaml
8          - helm upgrade python-demo ./deploy/kubernetes/
               helm --install -f ./deploy/kubernetes/helm/
               values.yaml
9      when: manual
```

4.2.3 Cadastrando Gitlab Runners

Para cadastrar o *Gitlab Runner* no *cluster Kubernetes* utilizaremos o Helm. Inicialmente iremos adicionar o repositório *Helm* oficial do *Gitlab* e em seguida o *chart* é instalado (Listagem 4.15) passando o arquivo de valores (Listagem 4.16) com o *token* de registro e as configurações necessárias e que queremos o nosso *runner*. Com isso podemos ver o *pod* do *runner* ativo como mostrado na figura 4.11, `gitlab-runner-<id-do-runner>` e o *runner* cadastrado na figura 4.18 com o nome e as *tags* especificadas no arquivo de valores.

Listagem 4.15: Instalando e cadastrando Gitlab Runner no cluster Kubernetes

```
1  helm repo add gitlab https://charts.gitlab.io
2
3  helm install --namespace default gitlab-runner -f /path/to
       /values.yaml gitlab/gitlab-runner
```

Listagem 4.16: Arquivo de valores de configuração para registro do *Gitlab Runner*

```
1  certsSecretName: gitlab-wildcard-tls-chain
2  checkInterval: 30
```

```
concurrent: 4
gitlabUrl: https://gitlab.192.168.49.2.nip.io/
rbac:
  clusterWideAccess: false
  create: true
  enabled: true
resources: {}
runnerRegistrationToken: GR13489413SBbRh27yQSzABEiNtNN
runners:
  builds: {}
  helpers: {}
  hostNetwork: true
  image: ubuntu:16.04
  privileged: true
  services: {}
tags: "test, build, deploy"
name: "Kubernetes Runner"
```

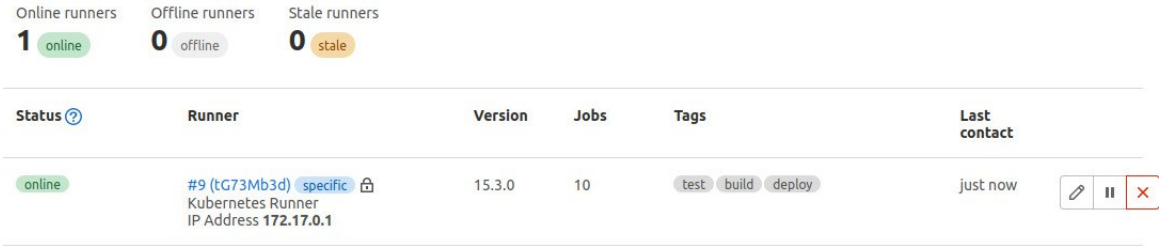


Figura 4.18: Runner cadastrado

Capítulo 5

Resultado e Análise

Após a configuração das duas implementações, é possível observar o funcionamento, testar e analisar as diferenças entre elas. Para isso serão feitas alterações simples na aplicação e simular uma falha na aplicação e como cada uma das estruturas se comporta em tal evento.

5.1 Alteração na aplicação

Iremos realizar a mesma alteração em ambas as implementações, para a pipeline CI/CD não importa o que será mudado no código, seja todo o *backend* ou *frontend* da aplicação, para mudanças mais complexas pode ser necessário ajustar o *script* dos *jobs*. Portanto faremos mudanças na aparência pois são mais evidentes. Em seguida será retratada o funcionamento da pipeline por estágios e a aplicação.

No arquivo *base.html*, este se encontra a marcação das partes comuns em todo site, então mudaremos o texto apresentado e a cor que está na barra de navegação. A Listagem 5.1 mostra a marcação de página original da barra de navegação, a Listagem 5.2 evidencia a troca do texto para a implantação em *Docker* e a Listagem 5.3 mostra as alterações feitas na implantação em *Kubernetes*.

Listagem 5.1: Marcação da barra de navegação

```
1     <nav class="navbar navbar-expand-lg navbar-dark bg-
          primary">
2       <a class="navbar-brand logotext" href="/">
3         <img src="static/img/python.svg" width="37" height
              ="40" />
4           Python Demo App
5       </a>
6
7       <button
8         class="navbar-toggler"
9         type="button"
10        data-toggle="collapse"
```

```
11          data-target="#navbarNav"
12          aria-controls="navbarNav"
13          aria-expanded="true"
14          aria-label="Toggle navigation"
15        >
16          <span class="navbar-toggler-icon"></span>
17        </button>
18
19        <div class="collapse navbar-collapse" id="navbarNav"
              >
20          <ul class="navbar-nav mr-auto">
21            <li class="nav-item active">
22              <a class="btn btn-success btn-lg" href="/info"
                    >Info</a>
23            </li>
24             
25            <li class="nav-item active">
26              <a class="btn btn-success btn-lg" href="/
                    monitor">Monitor</a>
27            </li>
28          </ul>
29        </div>
30      </nav>
```

Listagem 5.2: Alteração na barra de navegação - Docker

```
1     <nav class="navbar navbar-expand-lg navbar-dark bg-
          light">
2         <a class="navbar-brand logotext" href="/">
3             <img src="static/img/python.svg" width="37"
                  height="40" />
4               Python Demo App - Docker
5         </a>
```

Listagem 5.3: Alteração na barra de navegação - Kubernetes

```
1     <nav class="navbar navbar-expand-lg navbar-dark bg-
          info">
2         <a class="navbar-brand logotext" href="/">
3             <img src="static/img/python.svg" width="37"
                  height="40" />
4               Python Demo App - Kubernetes
5         </a>
```

5.1.1 Funcionamento da Pipeline

Qualquer alteração no código ativa a pipeline, esta aciona os *runners* ativos e disponíveis para rodar os *jobs* com suas *tags*. Na figura 5.1, podemos ver assim que a pipeline foi

ativada e o estagio em que se encontra e os próximos, e na figura 5.2 está sendo mostrada a necessidade da atuação manual para realizar o último estágio.

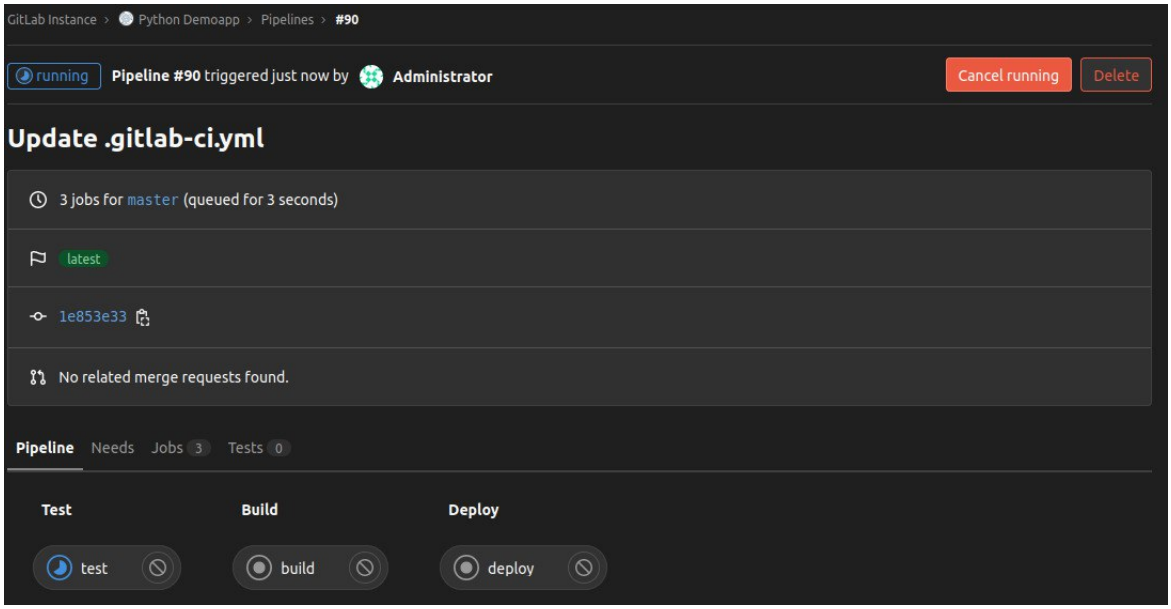


Figura 5.1: Pipeline ativada

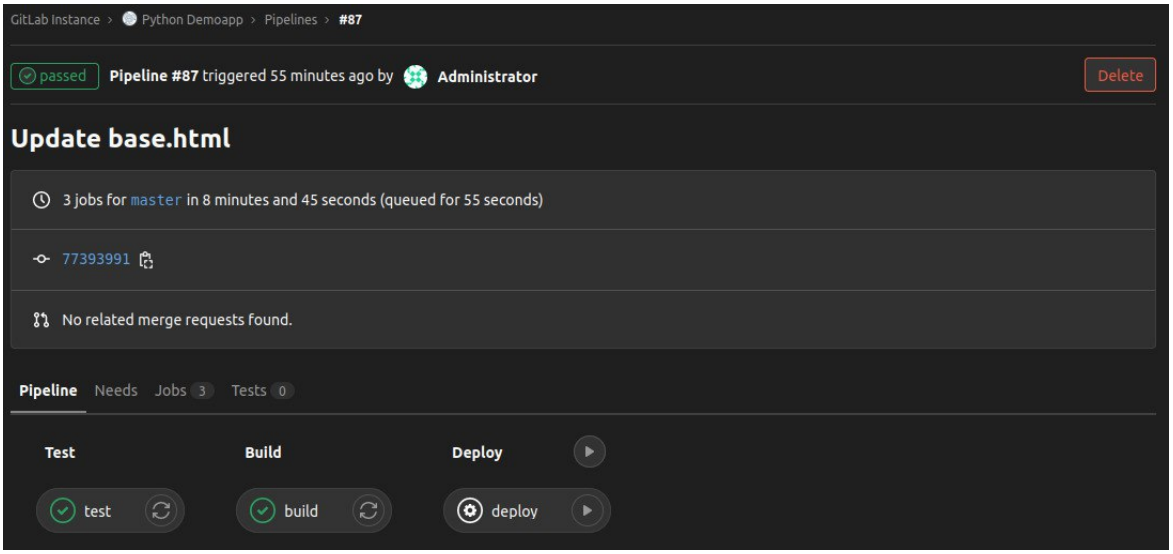


Figura 5.2: Pipeline aguardando atuação manual para realizar o estágio de *deploy*

5.1.1.1 Test

Nas figuras 5.3,5.4, 5.5 e 5.6 é mostrado o detalhamento do *job* de testes em ambas implementações. Inicialmente (Figuras 5.3 e 5.5) podemos ver o *runner* sendo selecionado, como por exemplo na linha 1 e 2 da figura 5.3 o *runner* de nome *test-runner* sendo selecionado, e informações como a duração de execução do *job*. Em seguida vemos o *job* finalizado com sucesso, na figura 5.4 temos entre as linhas 154 e 164 os testes realizados e o resultado deles e da mesma forma na figura 5.6 entre as linhas 184 e 197.

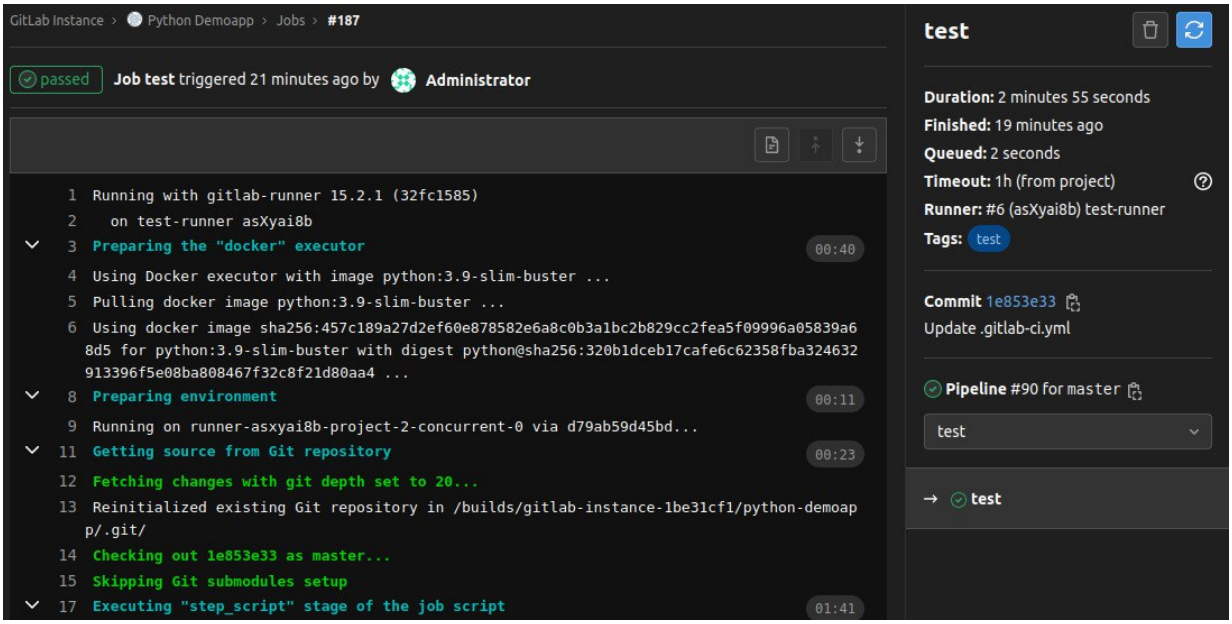


Figura 5.3: Infraestrutura Docker, Detalhamento do *job* de testes - início

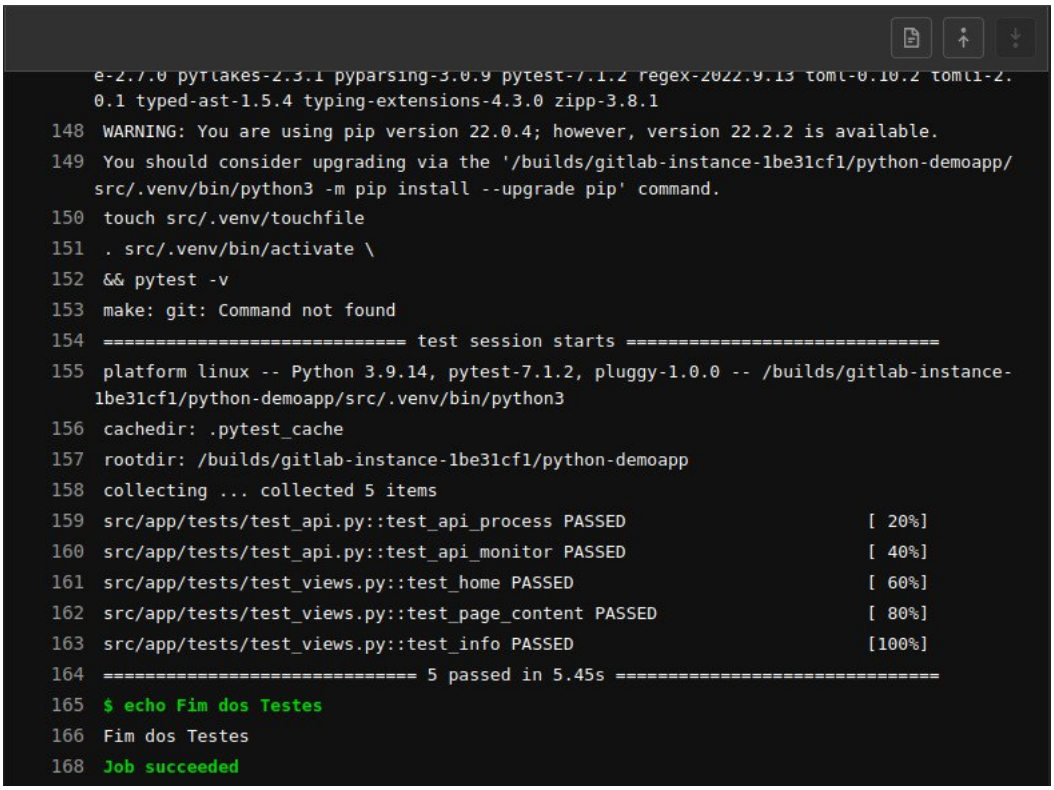


Figura 5.4: Infraestrutura Docker, Detalhamento do *job* de testes - final

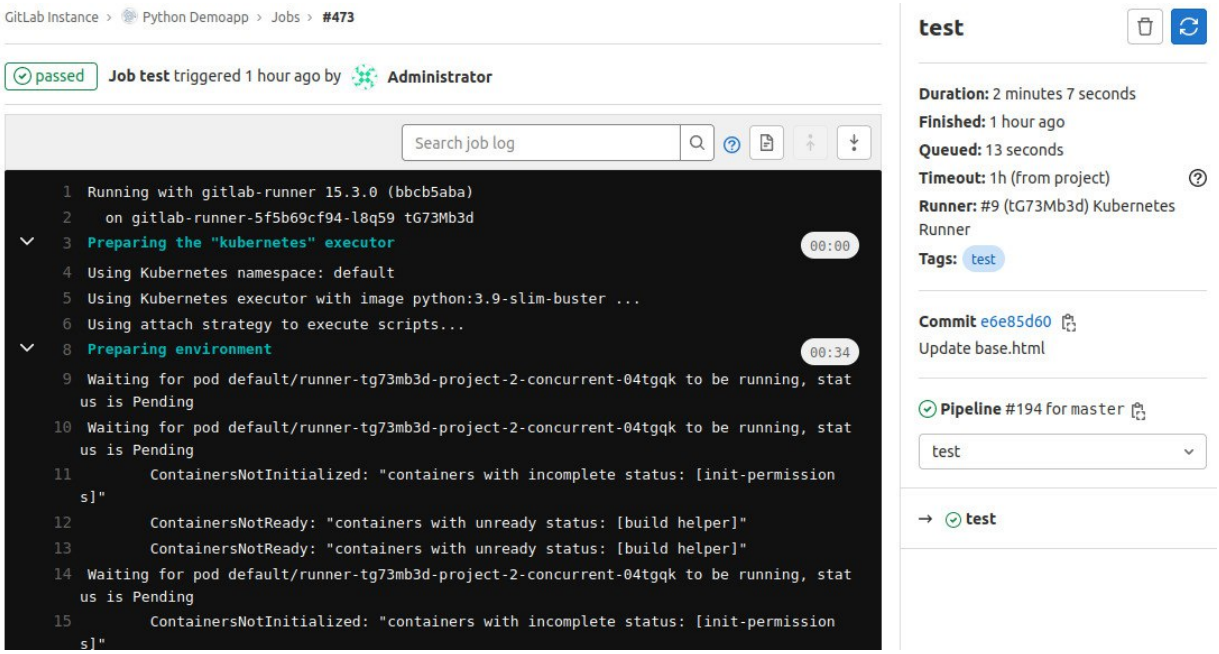


Figura 5.5: Infraestrutura Kubernetes, Detalhamento do *job* de testes - inicio

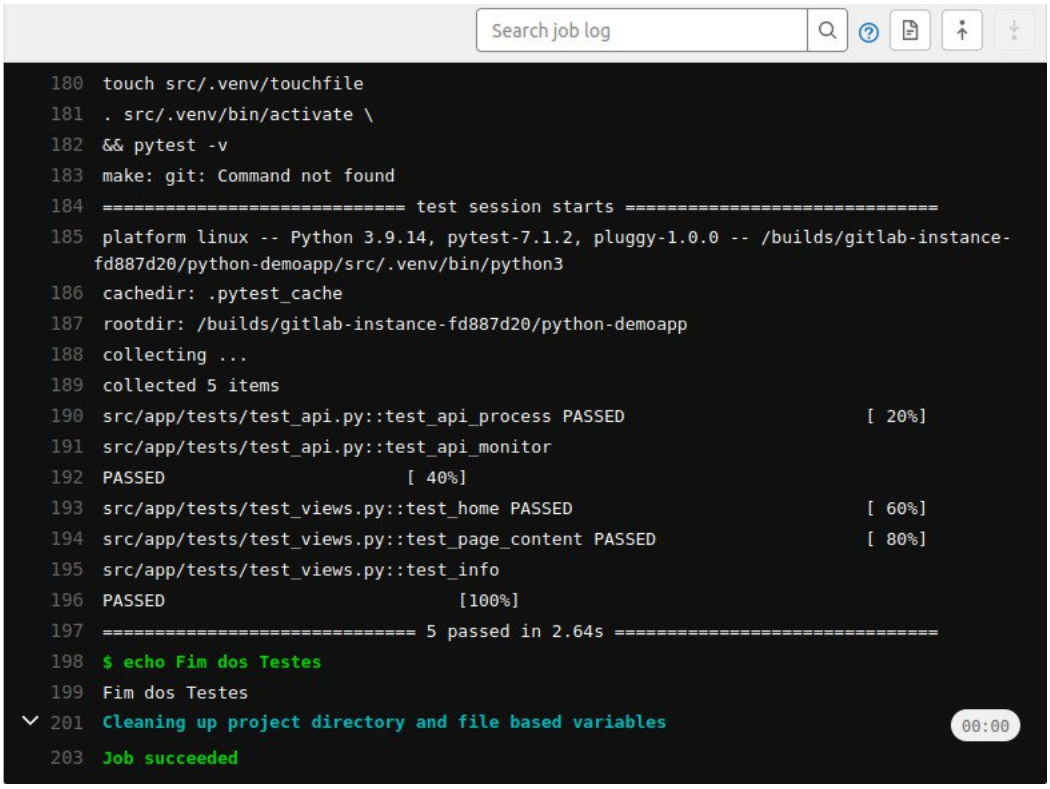


Figura 5.6: Infraestrutura Kubernetes, Detalhamento do *job* de testes - final

5.1.1.2 Build

Ao finalizar os testes é passado para o próximo *job*, o *job* de *build*. As imagens do contêiner se diferem no conteúdo da aplicação que está sendo construída, pois as alterações foram diferentes (Listagem 5.2, e 5.3) e no repositório de onde estão sendo armazenadas. Nas figuras 5.8 e 5.10 (Infraestrutura Docker e Kubernetes, respectivamente) vemos as imagens que foram publicadas e suas *tags* de versões que podem ser conferidas nas figuras 5.7 e 5.9 observando o identificador da pipeline, como foi explicado nas seções 4.1.1 e 4.2.2. Utilizando as figuras 5.7 e 5.9, podemos ver que os *jobs* de *build* terminaram com sucesso e a duração que cada um dos *jobs* demorou para ser realizado.

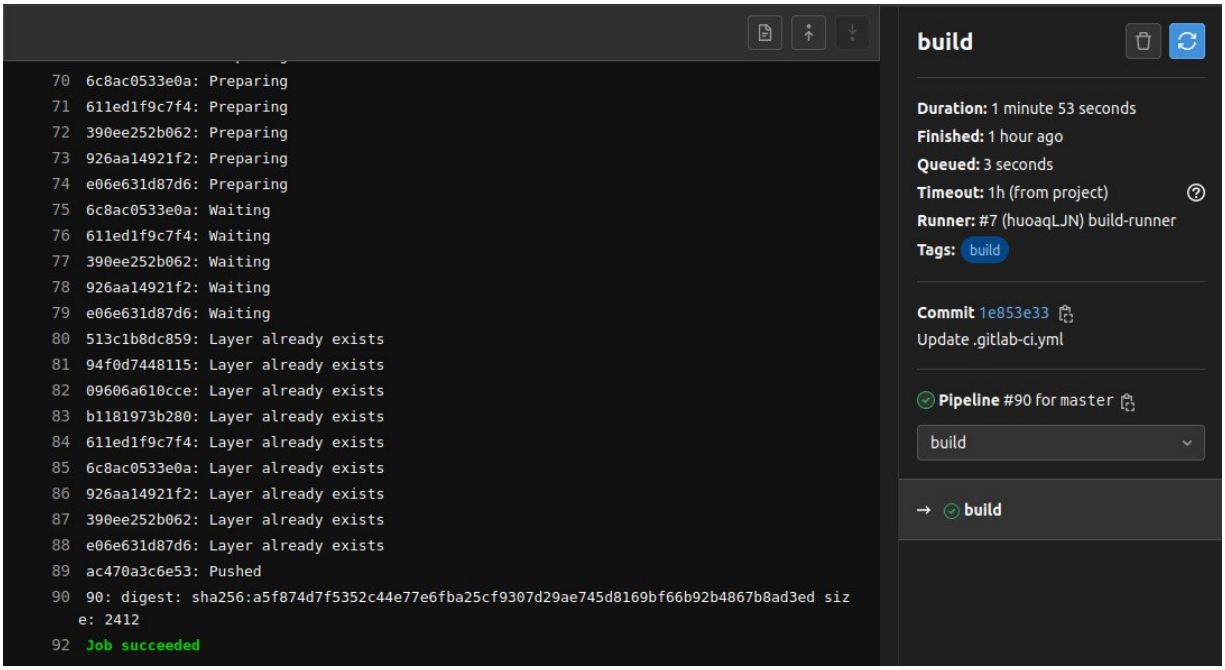


Figura 5.7: Infraestrutura Docker, Detalhamento do *job* de Build

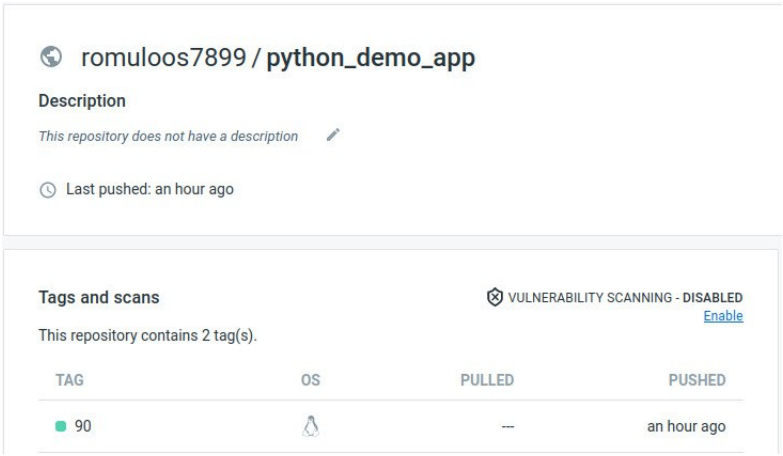


Figura 5.8: Registro onde está sendo armazenada a imagem - Infraestrutura Docker

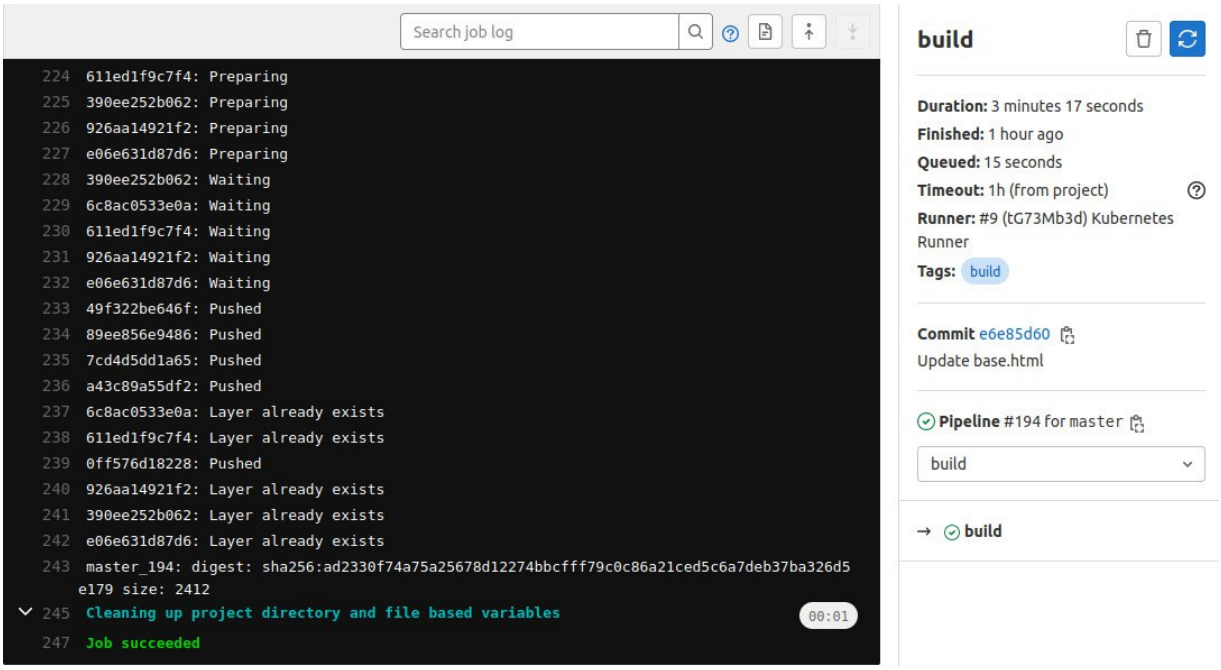


Figura 5.9: Infraestrutura Kubernetes, Detalhamento do *job* de Build

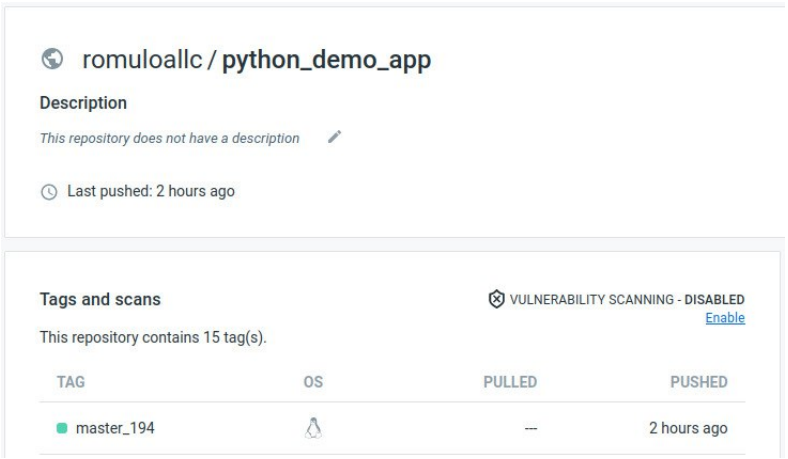


Figura 5.10: Registro onde está sendo armazenada a imagem - Infraestrutura Kubernetes

5.1.1.3 Deploy

O *deploy* para cada infraestrutura tem suas características, estas que são evidenciadas nas seções 4.1.1 e 4.2.2. Na figura 5.11 na linha 61 vemos o identificador do contêiner que foi removido com a versão antiga da aplicação e na linha 63 o identificador do contêiner com a aplicação atualizada, na figura 5.13 vemos qual *release* foi atualizada (*python-demo*) e em ambas as imagens podemos ver o tempo de duração do *job*. Ao terminar com sucesso a implantação da aplicação em ambas infraestruturas, devemos conferir se estão de fato ativas. Na figura 5.12 podemos ver o status e porta da máquina hospedeira que o contêiner gerado está utilizando para a aplicação, lembrando que esta sendo executada em cima de *Docker*, e na figura 5.14 vemos que o pod da aplicação está com status running e para acessá-la devemos utilizar a porta `30022` do Minikube.

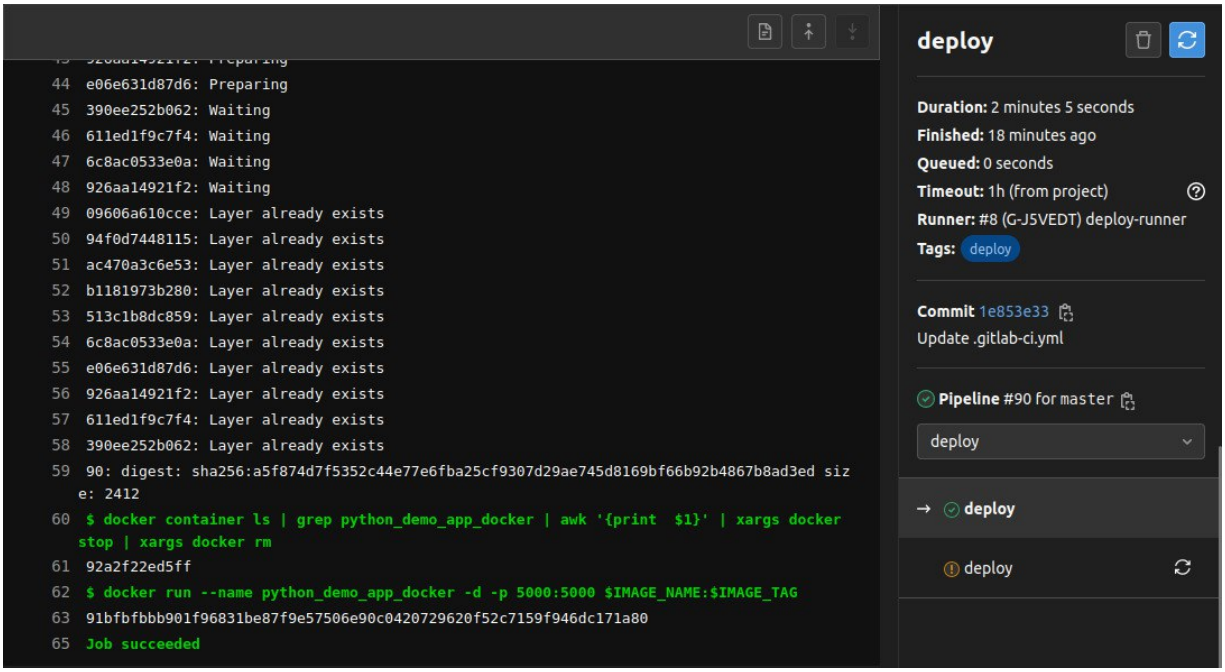


Figura 5.11: Infraestrutura Docker, Detalhamento do *job* de deploy

```
:~$ docker container ls
CONTAINER ID   IMAGE                                COMMAND                  CREATED       STATUS              PORTS
          NAMES
92a2f22ed5ff   043c4cb60894                         "gunicorn -b 0.0.0.0…"   5 hours ago   Up 5 hours          0.0.0.0:5000->5000/tcp, :::5000->5000/tcp
          python_demo_app_docker
```

Figura 5.12: Informações do contêiner da aplicação

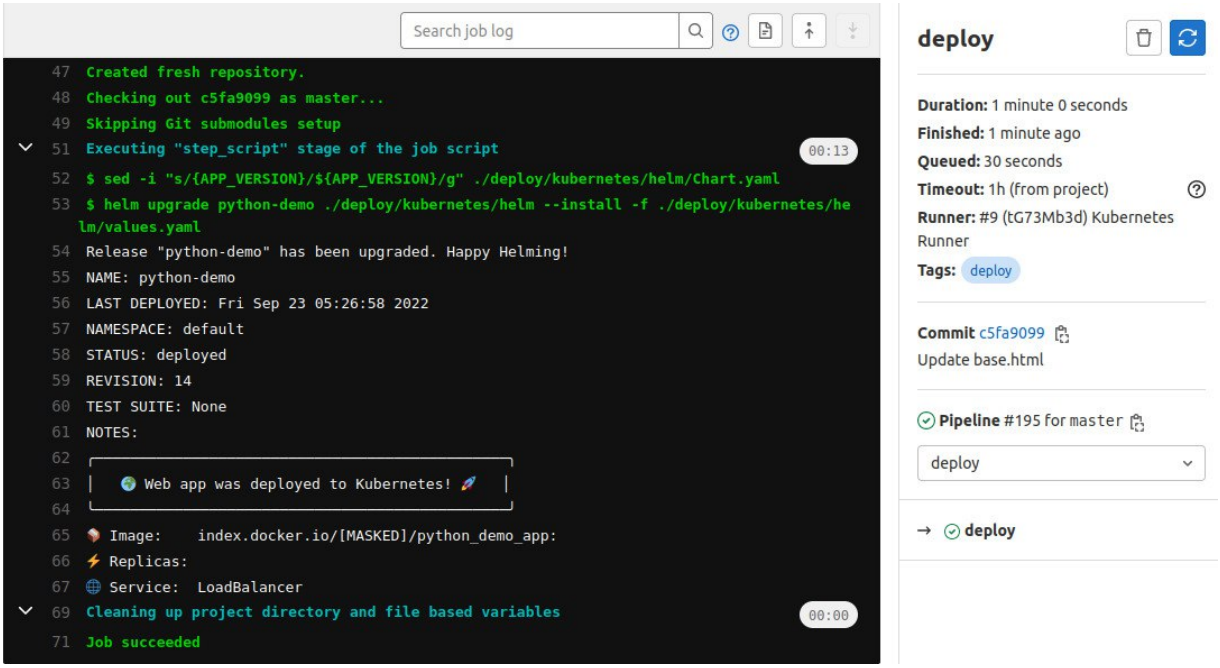


Figura 5.13: Infraestrutura Kubernetes, Detalhamento do *job* de deploy

```
romulo@pfg:~$ kubectl get pods | grep python-demo
python-demo-webapp-777887cc6b-jh9qd          1/1     Running     0              163m
romulo@pfg:~$ kubectl get svc | grep python-demo
python-demo-webapp           LoadBalancer   10.109.235.185   <pending>     80:30022/TCP
```

Figura 5.14: Informações do *pod* e serviço da aplicação

Acessando as aplicações vemos as páginas iniciais (Figuras 5.15 e 5.16), a página de informações da máquina onde a aplicação esta sendo hospedada, (Figuras 5.17 e 5.18) e a página de monitoramento (Figuras 5.19 e 5.20) e em ambas as implementações com as modificações que foram feitas.

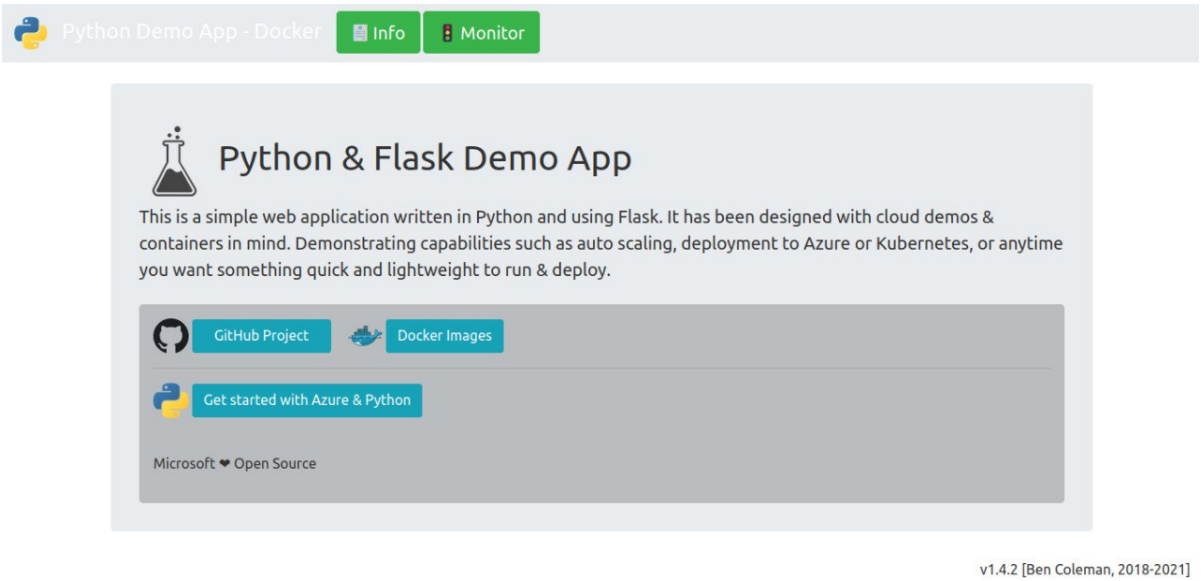


Figura 5.15: Página inicial da aplicação - Infraestrutura Docker

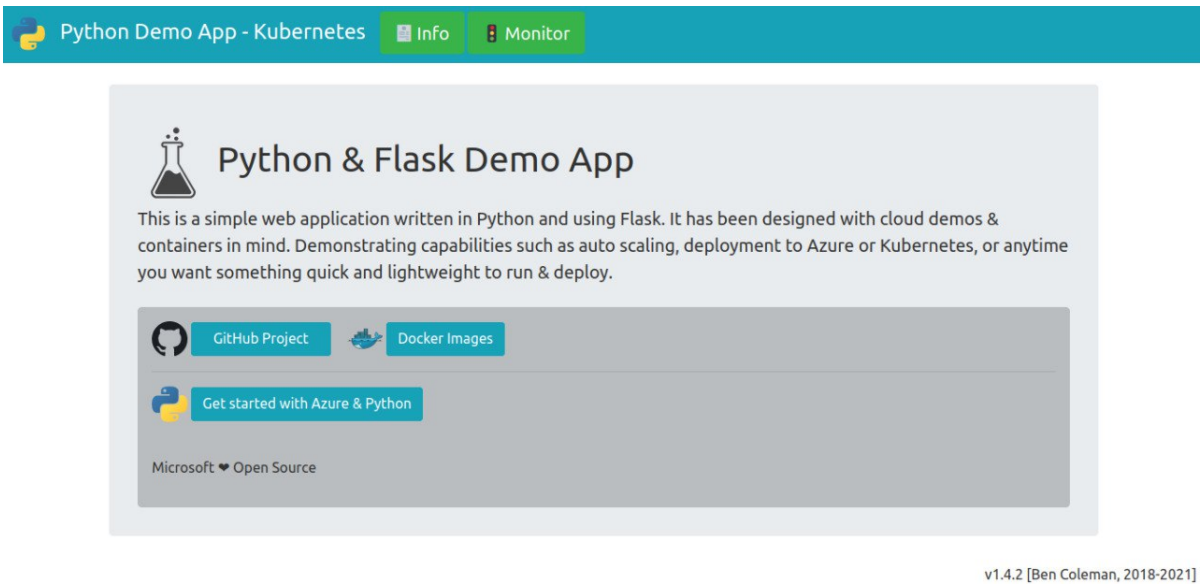


Figura 5.16: Página inicial da aplicação - Infraestrutura Kubernetes

Na página de informações podemos notar a maior diferença entre as infraestruturas, os *Hostnames*, no caso do *Docker* é a identificação do contêiner (Figura 5.12) e no *Kubernetes* é a identificação do *pod* (Figura 5.14), em ambas as figuras é possível ver a versão do sistema operacional, quantos processadores e *cores* temos na máquina e quantos gigas de memória tem e quantos por cento estão sendo utilizados.

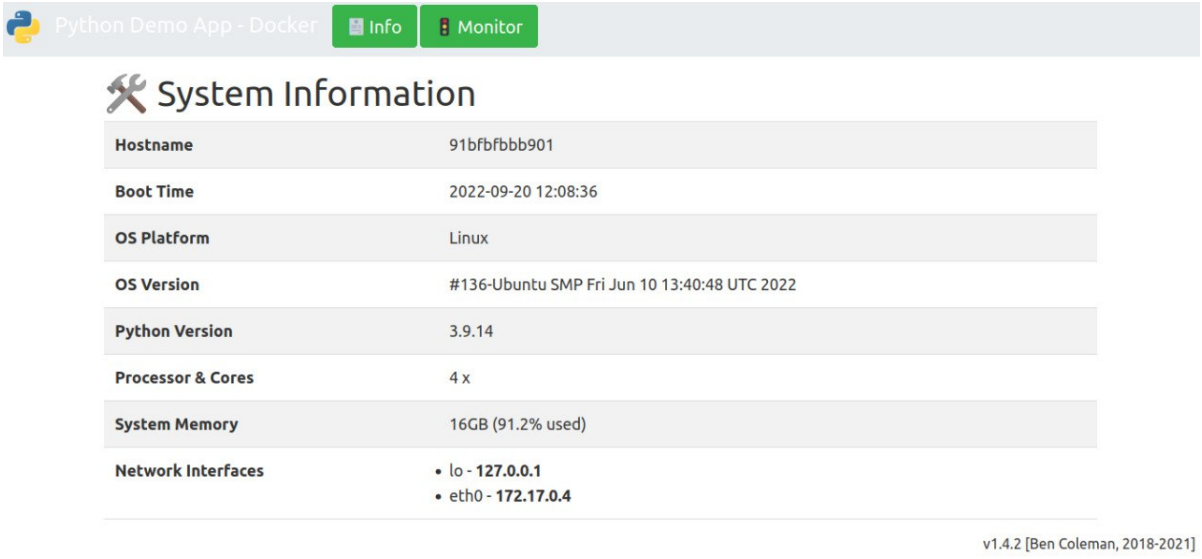


Figura 5.17: Página de informação da aplicação - Infraestrutura Docker

A página de monitoramento nos mostra o consumo de CPU e memória da máquina hospedeira da aplicação e atualiza esse valor a cada 3 segundos, o valor de CPU oscila, mas a memória está sendo consumida quase inteira, a maior culpa para isso é devido ao cluster *Kubernetes* rodando na máquina.

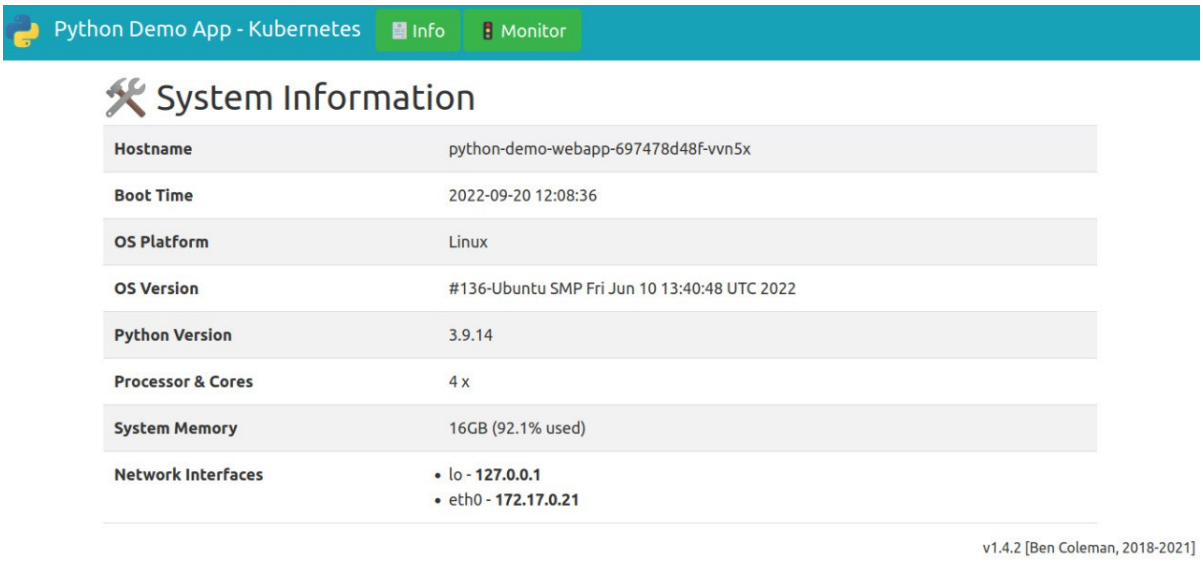


Figura 5.18: Página de informação da aplicação - Infraestrutura Kubernetes

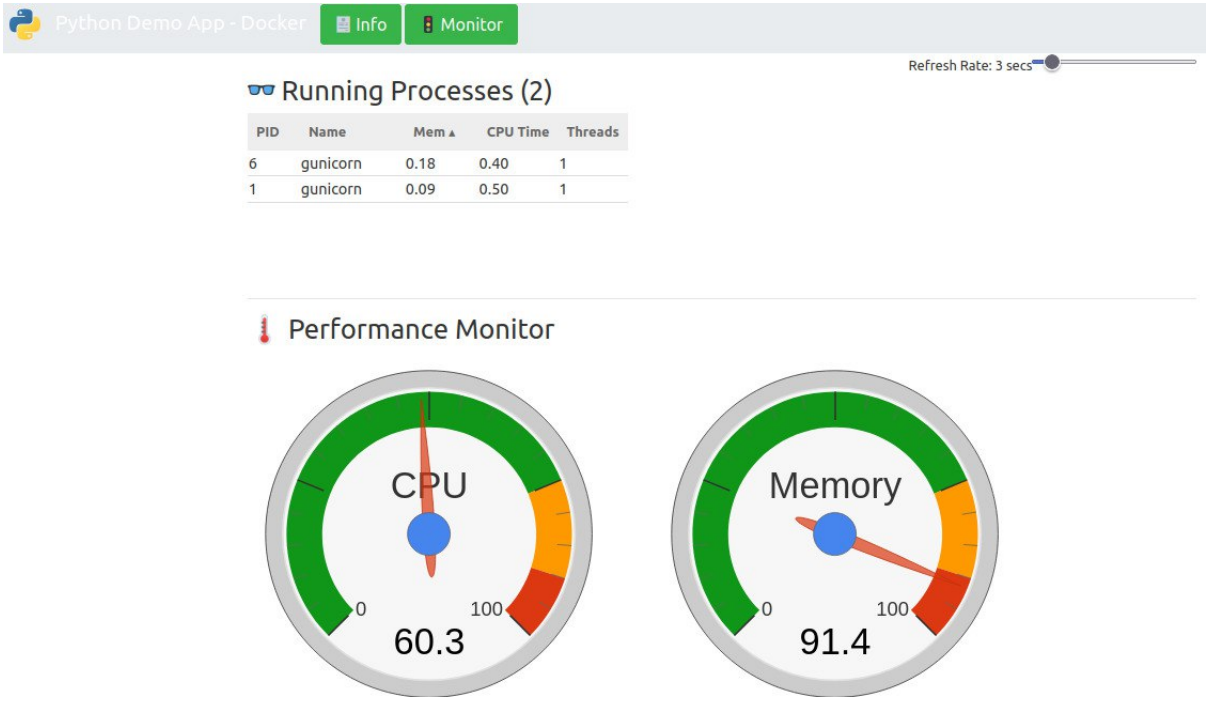


Figura 5.19: Página de monitoramento da aplicação - Infraestrutura Docker

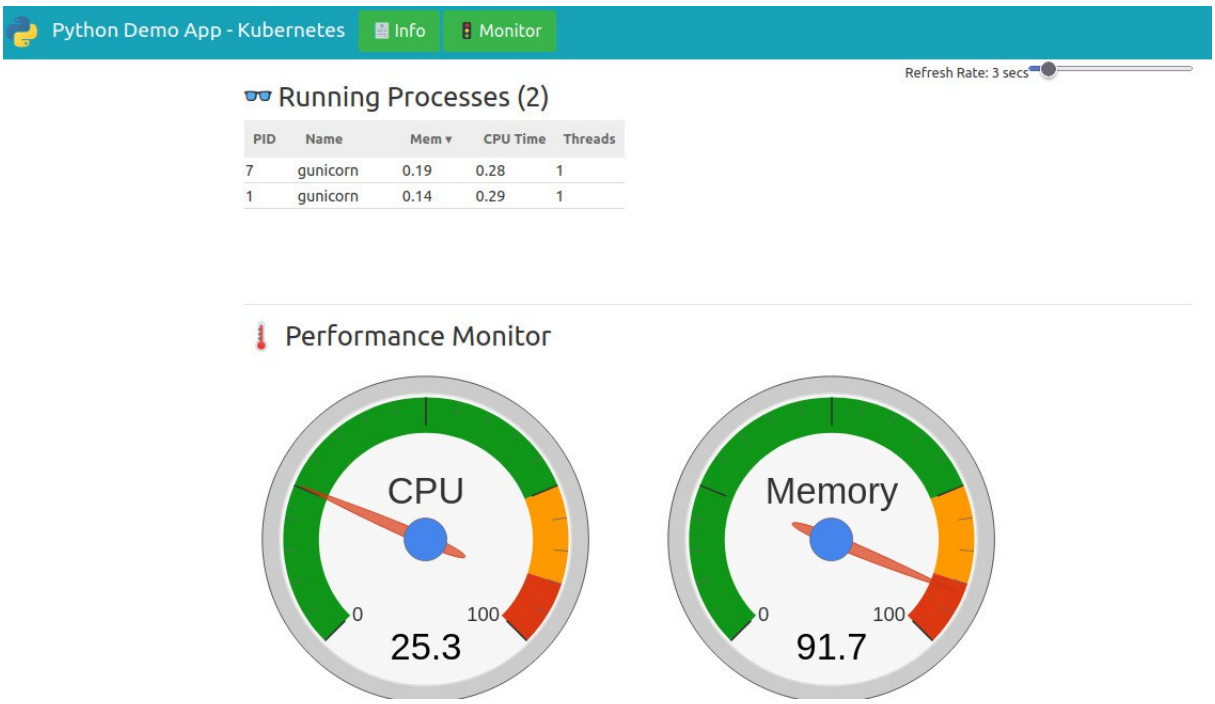


Figura 5.20: Página de monitoramento da aplicação - Infraestrutura Kubernetes

Com o comando `top` (Figura 5.21) conseguimos validar que o valor está bem próximo da aplicação e do valor da máquina, verificando o valor total da máquina (15936 MB) e quantos megabytes estão sendo utilizados (13884,3 MB), dessa forma, tem-se aproximadamente 90%.

```
top - 02:59:33 up 2 days, 17:50,  1 user,  load average: 3,85, 4,62, 5,35
Tasks: 537 total,   1 running, 536 sleeping,   0 stopped,   0 zombie
%Cpu(s): 25,5 us,  7,8 sy,  0,0 ni, 61,7 id,  4,8 wa,  0,0 hi,  0,2 si,  0,0 st
MiB Mem :  15936,0 total,    396,0 free,  13884,3 used,   1655,7 buff/cache
MiB Swap:   2048,0 total,      0,0 free,   2048,0 used.   1168,0 avail Mem
```

Figura 5.21: Saída do comando *top* para conferir consumo de memória e CPU

Podemos constatar o tempo de cada *job* em ambas as implementações e obter o tempo total de cada pipeline. Na tabela 5.1, podemos observar que o tempo total em ambas as implementações não tem tanta discrepância 29 segundos apenas, ou seja, o tempo de duração da pipeline não é afetado se usadas diferentes implementações. No entanto se comparado com o modelo de desenvolvimento de *software* tradicional, o qual o desenvolvedor entrega um código e este é revisado por um desenvolvedor com mais experiência e se aprovado é repassado para o time de testes e qualidade e só então é feita a implantação de uma nova versão da aplicação, podendo levar horas ou dias. E com os processos automatizados na pipeline CI/CD temos esse tempo reduzido para 7 minutos aproximadamente, aumentando bastante o potencial de entrega de um produto de uma empresa.

Jobs	Docker	Kubernetes
test	2:55 min	2:07 min
build	1:53 min	3:17 min
deploy	2:05 min	1:00 min
Duração total	6:53 min	6:24min

Tabela 5.1: Duração total de cada pipeline CI/CD

5.2 Simulação de falha na aplicação

Evidenciando as diferenças do *deploy* das duas infraestruturas, iremos simular a falha da aplicação. Para isso o contêiner e o *pod* em execução com a aplicação serão derrubados, e analisaremos o comportamento de cada infraestrutura.

5.2.1 Falha - Infraestrutura Docker

Com o identificador do contêiner, *Hostname* na figura 5.17, e o comando `docker stop <identificador do contêiner>`, é possível parar a execução do contêiner o qual a aplicação está (Listagem 5.4). Assim não é possível acessar a aplicação gerando tempo de inatividade. E requer atuação para resolver os problemas que podem ter causado a falha da aplicação e assim realizar o *deploy* para que esteja no ar novamente.

Listagem 5.4: Saída do comando docker stop

```
1 docker stop 91bfbfbbb901
2 91bfbfbbb901
```

5.2.2 Falha - Infraestrutura Kubernetes

Utilizando o identificador do *pod*, *Hostname* na figura 5.18, e o comando `kubectl delete pod <identificador do pod>`, removemos o *pod* em que a aplicação estava sendo executada (Figura 5.22).

No entanto, como o *Kubernetes* possui o serviço da aplicação ele consegue executar um *pod* com as mesmas características do que foi derrubado e assim a aplicação tem um tempo de inatividade irrisório (Figura 5.23). É possível ainda aumentar o número de réplicas da aplicação, gerando redundância na implantação e também utilizar o *template* do *chart* de escalonamento automático de *pod* como explicado no final da seção 4.2.1.

```
          :~$ kubectl get pods | grep python-demo
python-demo-webapp-697478d48f-vvn5x           1/1     Running     0              65m
          :~$ kubectl delete pod python-demo-webapp-697478d48f-vvn5x
pod "python-demo-webapp-697478d48f-vvn5x" deleted
```

Figura 5.22: Identificador do Pod e Pod deletado

```
[redacted]:~$ kubectl get pods -w | grep python-demo
python-demo-webapp-697478d48f-vvn5x           1/1     Running     0             65m
python-demo-webapp-697478d48f-vvn5x           1/1     Terminating   0             66m
python-demo-webapp-697478d48f-xhw2h           0/1     Pending       0             1s
python-demo-webapp-697478d48f-xhw2h           0/1     Pending       0             4s
python-demo-webapp-697478d48f-xhw2h           0/1     ContainerCreating   0               5s
python-demo-webapp-697478d48f-vvn5x           0/1     Terminating         0               66m
python-demo-webapp-697478d48f-vvn5x           0/1     Terminating         0               66m
python-demo-webapp-697478d48f-vvn5x           0/1     Terminating         0               66m
python-demo-webapp-697478d48f-xhw2h           1/1     Running             0               30s
```

Figura 5.23: Pod sendo apagado e subindo outro para substituir

Na figura 5.24 vemos que a aplicação está com a informação do *Hostname* atualizada com a identificação do novo *pod*.

🛠️ System Information

Hostname	python-demo-webapp-697478d48f-xhw2h
Boot Time	2022-09-20 12:08:36
OS Platform	Linux
OS Version	#136-Ubuntu SMP Fri Jun 10 13:40:48 UTC 2022
Python Version	3.9.14
Processor & Cores	4 x
System Memory	16GB (88.7% used)
Network Interfaces	• lo - **127.0.0.1** • eth0 - **172.17.0.16**

Figura 5.24: Página de informações com o novo *pod*

Capítulo 6

Conclusão

Portanto, esta dissertação teve como objetivo implementar pipelines CI/CD, a fim de alcançar um maior entendimento e aplicar seus conceitos, para isso mostrou-se necessário ter noções de como funciona a tecnologia de contêineres, como utilizar um orquestrador, as ferramentas que auxiliam na implantação e como executar a automação utilizando *softwares*, como *Gitlab*.

De forma complementar, utilizando os conceitos e características das pipelines CI/CD apresentados ao longo deste trabalho, foi realizada a configuração voltada para o desenvolvimento de *software* por meio de duas estruturas, Docker e Kubernetes, para que fossem implementadas os estudos e as melhores práticas para se obter o melhor resultado em ambas as implementações.

Com isso, obtemos os resultados esperados com o *deploy* da aplicação de demonstração por meio da pipeline CI/CD, passando por todos os estágios de forma automatizada e vemos por meio da duração de cada estágio o tempo empregado nas pipelines em suas devidas implementações, sendo comparadas com o tempo de entrega dos processos de desenvolvimento de *software* que ainda está presente na maioria das empresas. Também é mostrado as diferenças em uma contenção de falha da aplicação, seja por diversos motivos, nas duas implementações e o quanto um orquestrador de contêiner pode agregar nesses casos. Infelizmente, não foi possível demonstrar a escalabilidade e alta disponibilidade como era o pretendido, pois estes requisitos demandam grande poder computacional, maior que o disponível para esse projeto.

6.1 Trabalhos Futuros

Em suma, o estudo e as implementações proposta nesta dissertação foram limitados pelo poder computacional disponível, devido a esse fator espera-se que em trabalhos futuros possam ser desenvolvidos algumas opções escaláveis como, a implementação em múltiplos *clusters* simulando um ambiente real de uma empresa, com ambientes de desenvolvimento,

homologação e produção por exemplo, pipelines com estágios para cada ambiente. Outra sugestão é explorar os conceitos de nuvem, que são bastante utilizados principalmente pelo alto investimento de grandes empresas como AWS, Google Cloud e Microsoft Azure. Ainda sobre a implementação na nuvem, todo o procedimento adquire robustez pela escalabilidade apresentada tanto na infraestrutura como na implantação. Também fica como sugestão para implementações em outras áreas de atuação como pipelines CI/CD voltadas para *Machine Learning*, Inteligência Artificial e *Big Data*.

Referências Bibliográficas

[AWS-Devops] AWS-Devops. What-is-devops. https://aws.amazon.com/pt/devops/what-is-devops/. (Accessed on 19/08/2022).

[Ben-Coleman-Perfil] Ben-Coleman-Perfil. Perfil github: Ben coleman. https://github.com/benc-uk. (Accessed on 03/08/2022).

[Chen 2015] Chen, L. (2015). Continuous delivery: Huge benefits, but challenges too. *IEEE Software*.

[Docker] Docker. Official docker documentation. https://docs.docker.com/. (Accessed on 07/07/2022).

[Docker-Bridge] Docker-Bridge. Official docker documentation - bridge networks. https://docs.docker.com/network/bridge/. (Accessed on 07/07/2022).

[Docker-Inc] Docker-Inc. Use containers to build, share and run your applications. https://www.docker.com/resources/what-container/. (Accessed on 07/08/2022).

[Docker-Overview] Docker-Overview. Docker overview. https://docs.docker.com/engine/docker-overview/. (Accessed on 09/08/2022).

[DockerHub] DockerHub. Docker hub. https://hub.docker.com/. (Accessed on 08/07/2022).

[dos Santos Reis 2017] dos Santos Reis, D. C. (2017). Execução e Gestão de Aplicações Conteinerizadas. *Universidade do Porto*.

[Freire 2021] Freire, J. E. L. (2021). Orquestração de contêineres Usando Kubernetes e Docker Swarm. *Universidade Beira Interior*.

[Get-Docker] Get-Docker. Official docker documentation - install docker. https://docs.docker.com/get-docker/. (Accessed on 07/07/2022).

[Gitlab] Gitlab. Official documentation gitlab. https://gitlab.com/gitlab-org/gitlab. (Accessed on 15/07/2022).

[Gitlab-Helm-Charts] Gitlab-Helm-Charts. Gitlab helm charts official repository. http://charts.gitlab.io/. (Accessed on 14/09/2022).

[Gitlab-Minimum-Values] Gitlab-Minimum-Values. Pox. https://gitlab.com/gitlab-org/charts/gitlab/-/blob/master/examples/values-minikube-minimum.yaml. (Accessed on 15/09/2022).

[Gitlab-Runner] Gitlab-Runner. Official documentation gitlab runner. https://docs.gitlab.com/runner/. (Accessed on 15/07/2022).

[Helm] Helm. Helm-overview. https://helm.sh/docs/helm/helm/. (Accessed on 26/08/2022).

[Helm-Charts] Helm-Charts. Charts-overview. https://helm.sh/docs/topics/charts/. (Accessed on 26/08/2022).

[Installing-Helm-Charts] Installing-Helm-Charts. Installing helm. https://helm.sh/docs/intro/install/. (Accessed on 14/09/2022).

[Kubectl-Instalation] Kubectl-Instalation. Kubectl instalation. https://kubernetes.io/docs/tasks/tools/#kubectl. (Accessed on 14/09/2022).

[Kubernetes] Kubernetes. Kubernetes-overview. https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/. (Accessed on 16/08/2022).

[Kubernetes-Kubectl-Overview] Kubernetes-Kubectl-Overview. Kubectl overview. https://kubernetes.io/docs/tasks/tools/. (Accessed on 20/08/2022).

[Kubernetes-Pod] Kubernetes-Pod. Kubernetes pod overview. https://kubernetes.io/docs/concepts/workloads/pods/. (Accessed on 19/08/2022).

[Kubernetes-Volume] Kubernetes-Volume. Kubernetes volumes overview. https://kubernetes.io/docs/concepts/storage/volumes/. (Accessed on 19/08/2022).

[Minikube] Minikube. Official minikube documentation. https://minikube.sigs.k8s.io/docs/. (Accessed on 24/08/2022).

[Minikube-Addons] Minikube-Addons. Official minikube documentation - addons. https://minikube.sigs.k8s.io/docs/. (Accessed on 24/08/2022).

[Minikube-Start] Minikube-Start. Official minikube documentation - minikube start. https://minikube.sigs.k8s.io/docs/start/. (Accessed on 24/08/2022).

[Mogallapu 2019] Mogallapu, R. V. (2019). Scalability of Kubernetes Running Over AWS. *Faculty of Computing, Blekinge Institute of Technology,*.

[Python-Demoapp] Python-Demoapp. Python flask - demo web application. https://github.com/benc-uk/python-demoapp. (Accessed on 03/08/2022).

[Rajeshsgr] Rajeshsgr. Mount volumes to persist data in local initialize database in docker. https://belowthemalt.com/2021/12/08/mount-volumes-to-persist-data-in-local-initialize-database-in-docker/. (Accessed on 22/08/2022).

[Red Hat] Red Hat, I. Ci/cd: integração e entrega contínuas. https://www.redhat.com/pt-br/topics/devops/what-is-ci-cd. (Accessed on 02/09/2022).

[Veritas] Veritas. O que é conteinerização? quais são os benefícios? https://www.veritas.com/pt/br/information-center/containerization. (Accessed on 06/09/2022).

Apêndice

As imagens dos contêiner utilizadas nessa dissertação podem ser encontradas em:

- Gitlab Community Edition - `https://hub.docker.com/r/gitlab/gitlab-ce`
- Aplicação de demonstração, Implementação em *Docker* - `https://hub.docker.com/r/romuloos7899/python_demo_app`
- Aplicação de demonstração, Implementação em *Kubernetes* - `https://hub.docker.com/r/romuloallc/python_demo_app`
- Python - `https://hub.docker.com/_/python`
- Docker (Docker-in-Docker) - `https://hub.docker.com/_/docker`
- Gitlab Runner - `https://hub.docker.com/r/gitlab/gitlab-runner`
- Helm-Kubernetes image - `https://hub.docker.com/r/dtzar/helm-kubectl/`

Para acrescentar e complementar a este documento, este capítulo também apresenta os arquivos inteiros que foram apenas citados ou apresentados por partes em outros capítulos.

Listagem 6.1: Dockerfile da aplicação

```
1  FROM python:3.9-slim-buster
2
3  ARG srcDir=src
4  WORKDIR /app
5  COPY $srcDir/requirements.txt .
6  RUN pip install --no-cache-dir -r requirements.txt
7
8  COPY $srcDir/run.py .
9  COPY $srcDir/app ./app
10
11 EXPOSE 5000
12
13 CMD ["gunicorn", "-b", "0.0.0.0:5000", "run:app"]
```

Listagem 6.2: Arquivo *.gitlab-ci.yml* em infraestrutura *Docker*

```
variables:
    IMAGE_NAME: romuloos7899/python_demo_app
    IMAGE_TAG:  ${CI_PIPELINE_ID}

stages:
    - test
    - build
    - deploy

test:
    stage: test
    tags:
        - test
    image: python:3.9-slim-buster
    before_script:
        - apt-get update && apt-get install make
    script:
        - make test
        - echo Fim dos Testes

build:
    stage: build
    tags:
        - build
    image: docker:20.10.17-dind
    services:
        - docker:20.10.17-dind
    variables:
        DOCKER_TLS_CERTDIR: "/certs"
    before_script:
        - docker login -u $REGISTRY_USER -p
            $REGISTRY_PASSWORD
    script:
        - docker build -t $IMAGE_NAME:$IMAGE_TAG .
        - docker push $IMAGE_NAME:$IMAGE_TAG

deploy:
    stage: deploy
    tags:
        - deploy
    image: docker:20.10.17-dind
    services:
        - docker:20.10.17-dind
    variables:
        DOCKER_TLS_CERTDIR: "/certs"
    before_script:
        - docker login -u $REGISTRY_USER -p
            $REGISTRY_PASSWORD
```

```
47      script:
48          - docker push $IMAGE_NAME:$IMAGE_TAG
49          - docker container ls | grep
                python_demo_app_docker | awk '{print  $1}' |
                xargs docker stop | xargs docker rm
50          - docker run --name python_demo_app_docker -d -p
                5000:5000 $IMAGE_NAME:$IMAGE_TAG
51      when: manual
```

Listagem 6.3: Arquivo de configuração gerado pelo registro dos *runners*

```
1  concurrent = 1
2  check_interval = 0
3
4  [session_server]
5    session_timeout = 1800
6
7  [[runners]]
8    name = "test-runner"
9    url = "http://172.17.0.2:80"
10   token = "asXyai8bQ2nQxp_GMhxY"
11   executor = "docker"
12   [runners.custom_build_dir]
13   [runners.cache]
14     [runners.cache.s3]
15     [runners.cache.gcs]
16     [runners.cache.azure]
17   [runners.docker]
18     tls_verify = false
19     image = "gitlab/gitlab-runner:latest"
20     privileged = false
21     disable_entrypoint_overwrite = false
22     oom_kill_disable = false
23     disable_cache = false
24     volumes = ["/var/run/docker.sock:/var/run/docker.sock
           ", "/cache"]
25     extra_hosts = ["gitlab.example.com:172.17.0.2"]
26     network_mode = "host"
27     shm_size = 0
28
29 [[runners]]
30   name = "build-runner"
31   url = "http://172.17.0.2:80"
32   token = "huoaqLJNQRtonCn3mtdz"
33   executor = "docker"
34   [runners.custom_build_dir]
35   [runners.cache]
36     [runners.cache.s3]
37     [runners.cache.gcs]
38     [runners.cache.azure]
```

```
39    [runners.docker]
40      tls_verify = false
41      image = "docker:stable"
42      privileged = false
43      disable_entrypoint_overwrite = false
44      oom_kill_disable = false
45      disable_cache = false
46      volumes = ["/var/run/docker.sock:/var/run/docker.sock
            ", "/cache"]
47      extra_hosts = ["gitlab.example.com:172.17.0.2"]
48      network_mode = "host"
49      shm_size = 0
50
51  [[runners]]
52    name = "deploy-runner"
53    url = "http://172.17.0.2:80"
54    token = "G-J5VEDTg6ZzYsjWXSMo"
55    executor = "docker"
56    [runners.custom_build_dir]
57    [runners.cache]
58      [runners.cache.s3]
59      [runners.cache.gcs]
60      [runners.cache.azure]
61    [runners.docker]
62      tls_verify = false
63      image = "docker:stable"
64      privileged = false
65      disable_entrypoint_overwrite = false
66      oom_kill_disable = false
67      disable_cache = false
68      volumes = ["/var/run/docker.sock:/var/run/docker.sock
            ", "/cache"]
69      extra_hosts = ["gitlab.example.com:172.17.0.2"]
70      network_mode = "host"
71      shm_size = 0
```

Listagem 6.4: Chart.yaml

```
1  apiVersion: v2
2  name: webapp
3  description: Chart para realizar deploy de aplicacao web
4  type: application
5  version: 1.4.1
6  appVersion: {APP_VERSION}
```

Listagem 6.5: values.yaml

```
1  image:
2    repository: index.docker.io/romuloallc/python_demo_app
3    pullPolicy: Always
```

```
4
5  service:
6    targetPort: 5000
7    type: LoadBalancer
```

Listagem 6.6: deployment.yaml

```
1  apiVersion: apps/v1
2  kind: Deployment
3  metadata:
4    name: {{ include "webapp.fullname" . }}
5    labels:
6      {{- include "webapp.labels" . | nindent 4 }}
7  spec:
8    replicas: {{ .Values.replicaCount }}
9    selector:
10     matchLabels:
11       {{- include "webapp.selectorLabels" . | nindent 6 }}
12   template:
13     metadata:
14       labels:
15         {{- include "webapp.selectorLabels" . | nindent 8
               }}
16     spec:
17       containers:
18         - name: {{ .Chart.Name }}
19           image: "{{ .Values.image.repository }}:{{ .Chart
                  .AppVersion }}"
20           ports:
21             - name: http
22               containerPort: {{ .Values.service.targetPort
                      }}
23               protocol: TCP
```

Listagem 6.7: service.yaml

```
1  apiVersion: v1
2  kind: Service
3  metadata:
4    name: {{ include "webapp.fullname" . }}
5    labels:
6      {{- include "webapp.labels" . | nindent 4 }}
7  spec:
8    type: {{ .Values.service.type }}
9    ports:
10     - port: 80
11       targetPort: {{ .Values.service.targetPort }}
12       protocol: TCP
13       name: http
14   selector:
```

```
15      {{- include "webapp.selectorLabels" . | nindent 4 }}
```

Listagem 6.8: Arquivo *.gitlab-ci.yml* em infraestrutura *Kubernetes*

```
1  variables:
2      IMAGE_NAME: romuloallc/python_demo_app
3      APP_VERSION: ${CI_COMMIT_REF_SLUG}_${CI_PIPELINE_ID}
4
5  stages:
6      - test
7      - build
8      - deploy
9
10 test:
11     stage: test
12     tags:
13         - test
14     image: python:3.9-slim-buster
15     before_script:
16         - apt-get update && apt-get install make
17     script:
18         - make test
19         - echo Fim dos Testes
20     when: manual
21
22 build:
23     stage: build
24     tags:
25         - build
26     image: docker:18.09.7-dind
27     services:
28         - docker:18.09.7-dind
29     variables:
30         DOCKER_DRIVER: overlay2
31         DOCKER_HOST: tcp://docker:2375/
32         DOCKER_TLS_CERTDIR: ""
33     before_script:
34         - docker login -u $REGISTRY_USER -p
               $REGISTRY_PASSWORD
35     script:
36         - docker build -t $IMAGE_NAME:$APP_VERSION .
37         - docker push $IMAGE_NAME:$APP_VERSION
38
39 deploy:
40     stage: deploy
41     tags:
42         - deploy
43     image: dtzar/helm-kubectl:3.6.2
44     script:
45         - sed -i "s/{APP_VERSION}/${APP_VERSION}/g" ./
```

```
            deploy/kubernetes/helm/Chart.yaml
46          - helm upgrade python-demo ./deploy/kubernetes/
              helm --install -f ./deploy/kubernetes/helm/
              values.yaml
47      when: manual
```