UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE
CENTRO DE ENSINO SUPERIOR DO SERIDÓ
DEPARTAMENTO DE COMPUTAÇÃO E TECNOLOGIA
BACHARELADO EM SISTEMAS DE INFORMAÇÃO

APRIMORANDO A ENTREGA E DISPONIBILIDADE DE UM SISTEMA DE INFORMAÇÃO COM CI/CD E ESTRATÉGIAS DE IMPLANTAÇÃO

JONATHAN TAUAN PEREIRA MAIA

Caicó - RN
2024

JONATHAN TAUAN PEREIRA MAIA

APRIMORANDO A ENTREGA E DISPONIBILIDADE DE UM SISTEMA DE INFORMAÇÃO COM CI/CD E ESTRATÉGIAS DE IMPLANTAÇÃO

Trabalho de conclusão de curso de apresentado ao Departamento de Computação e Tecnologia da Universidade Federal do Rio Grande do Norte, no curso de Sistemas de Informação como parte dos requisitos para obtenção do título de Bacharel em Sistemas de Informação da Universidade Federal do Rio Grande do Norte.

Orientador(a): Me. Taciano de Morais Silva.

Caicó - RN
2024

Universidade Federal do Rio Grande do Norte - UFRN
Sistema de Bibliotecas - SISBI
Catalogação de Publicação na Fonte. UFRN - Biblioteca Setorial Profª. Maria Lúcia da Costa Bezerra - CERES - Caicó

Maia, Jonathan Tauan Pereira.
Aprimorando a entrega e disponibilidade de um sistema de informação com CI/CD e estratégias de implantação / Jonathan Tauan Pereira Maia. - Caicó, 2024.
84f.: il.

Monografia (graduação) - Universidade Federal do Rio Grande do Norte, Centro de Ensino Superior do Seridó, Departamento de Computação e Tecnologia, Bacharelado em Sistemas de Informação. Caicó, RN, 2024.
Orientação: Prof. Me. Taciano de Morais Silva.

1. Engenharia de software - Monografia. 2. Pipeline - Monografia. 3. Jenkins - Monografia. 4. Automação - Monografia. I. Silva, Taciano de Morais. II. Título.

RN/UF/BS-CERES CDU 004.41

Elaborado por Giulianne Monteiro Pereira Marques - CRB-15/714

JONATHAN TAUAN PEREIRA MAIA

Aprimorando a Entrega e Disponibilidade de um Sistema de Informação com CI/CD e Estratégias de Implantação

Trabalho de conclusão de curso de apresentado ao Departamento de Computação e Tecnologia da Universidade Federal do Rio Grande do Norte, no curso de Sistemas de Informação como parte dos requisitos para obtenção do título de Bacharel em Sistemas de Informação da Universidade Federal do Rio Grande do Norte.

Caicó - RN, 22 de agosto de 2024

Me. Taciano de Morais Silva
Orientador

Dr. Arthur Emanoel Cássio da Silva e Souza
Examinador

Dra. Huliane Medeiros da Silva
Examinadora

Dr. João Batista Borges Neto
Examinador

Caicó - RN
2024

Agradecimentos

Primeiramente, agradeço a Deus, causa primordial de todas as coisas.

Agradeço especialmente aos meus pais, que forneceram toda a base de sustentação e uma fonte inesgotável de apoio para que eu pudesse seguir em busca dos meus objetivos. Seus conselhos e orientação constante foram essenciais ao longo da minha formação.

Agradeço também à minha avó, cuja presença e ajuda constantes foram inestimáveis, e à minha bisavó (in memoriam), cujas palavras de encorajamento proferidas no início desta jornada continuam a me inspirar ao longo da vida.

Minha gratidão se estende profundamente à minha irmã e também à minha namorada, que permaneceu ao meu lado em cada desafio. Elas sempre estiveram comigo, oferecendo luz para o meu caminho e me deixando mais forte. Também sou grato pelo carinho e companhia constante que recebi de meu fiel amigo de quatro patas, cuja presença foi uma fonte de alegria e conforto ao longo dessa jornada.

Além destes, sou grato aos meus colegas de trabalho da Casa do Artesão do Seridó (CARTS), que desempenharam um papel essencial neste trabalho, e aos meus colegas de curso, que enfrentaram os desafios dessa etapa ao meu lado.

Por fim, um agradecimento especial ao meu orientador, cuja confiança em meu potencial e orientação dedicada foram fundamentais para o desenvolvimento deste projeto e para a conclusão do curso com êxito.

Resumo

Este trabalho teve como objetivo desenvolver um pipeline de Integração Contínua e Entrega Contínua (CI/CD, do inglês Continuous Integration and Continuous Delivery) utilizando Jenkins e Kubernetes, juntamente com outras ferramentas de código aberto, para otimizar os processos de desenvolvimento e entrega de software no sistema web SigArte da Casa do Artesão do Seridó (CARTS). Integrando práticas da cultura *DevOps* e princípios de desenvolvimento ágil, o pipeline abrangeu diversos processos eficientes da engenharia de software e uma implantação sem *downtime*, através de estratégias de atualização no Kubernetes. O trabalho evidencia a importância de um pipeline CI/CD bem estruturado para assegurar a qualidade e continuidade dos sistemas como o SigArte. Esta abordagem aumentou a agilidade no ciclo de desenvolvimento, melhorou a consistência e confiança no código, e eliminou o tempo de inatividade durante as atualizações. Adicionalmente, o trabalho produziu uma referência detalhada e replicável que serve como base sólida para futuras implementações de CI/CD.

Palavras-chave: Engenharia de Software, Pipeline de CI/CD, Jenkins, Kubernetes, DevOps.

Abstract

This work aimed to develop a Continuous Integration and Continuous Delivery (CI/CD) pipeline using Jenkins and Kubernetes, along with other open-source tools, to optimize software development and delivery processes in the SigArte web system of the Casa do Artesão do Seridó (CARTS). By integrating DevOps culture practices and agile development principles, the pipeline encompassed various efficient software engineering processes and zero-downtime deployment through Kubernetes update strategies. The work highlights the importance of a well-structured CI/CD pipeline in ensuring the quality and continuity of systems like SigArte. This approach increased agility in the development cycle, improved code consistency and reliability, and eliminated downtime during updates. Additionally, the work produced a detailed and replicable reference that serves as a solid foundation for future CI/CD implementations.

Keywords:Software Engineering, CI/CD Pipeline, Jenkins, Kubernetes, DevOps.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

UFRN Universidade Federal do Rio Grande do Norte

CI/CD Integração Contínua / Entrega Contínua

CI Integração Contínua

CD Entrega Contínua

SIG Sistema de Informação Gerencial

CARTS Casa do Artesão do Seridó

LABENS Laboratório de Banco de Dados e Engenharia de Software

DCT Departamento de Computação e Tecnologia

SGBD Sistema de Gerenciamento de Banco de Dados

JRE Java Runtime Environment

K8s Kubernetes

SO Sistema Operacional

MTV Models, Templates, Views

ORM Mapeamento Objeto Relacional

DTO Objeto de Transferência de Dados

PR Pull Request

LISTA DE CÓDIGOS

SUMÁRIO

1 Introdução

Nos últimos anos, o universo do desenvolvimento de software testemunhou uma transformação notável: a demanda por entregas ágeis e a capacidade de adaptação a mudanças contínuas nos requisitos e requisições de novas funcionalidades tornaram-se indispensáveis, principalmente para as empresas de tecnologia que buscam manter sua competitividade no mercado. Nesse contexto, a *Continuous Integration / Continuous Delivery*, em português: Integração Contínua / Entrega Contínua (CI/CD) emergiram como práticas comuns no ciclo de desenvolvimento e entrega de software, sendo uma aliada poderosa e amplamente utilizada na cultura *DevOps*. Neste meio conceitos de desenvolvimento e operações se unem para alavancar o sucesso de um projeto de software.

A CI permite que os programadores partilhem as suas contribuições constantemente, eliminando a necessidade de esperar que o seu trabalho seja concluído (Duvall; Matyas; Glover, 2007). Alcançando, assim, *feedbacks* mais rápidos das etapas seguintes, que vão desde a compilação do projeto até a implantação em um servidor, contribuindo para um ciclo de desenvolvimento ágil. Por exemplo, assim que o código está disponível no repositório remoto, um servidor de automação reage ao evento, disparando a execução de um *pipeline*, ou seja, uma série de tarefas que buscam analisar e garantir a qualidade do código, dispensando a atenção do desenvolvedor para a execução manual dessas tarefas.

A CD, por sua vez, estende o conceito da CI, permitindo a entrega contínua de software, automatizando todos os processos desde baixar o código do repositório até liberar as alterações em produção ou em outro ambiente de desenvolvimento. Tais abordagens se tornam vitais para arquiteturas modularizadas como, por exemplo, a de microsserviços. Segundo Duvall, Matyas e Glover (2007):

> A CI/CD não só garante que o código é funcional, como também introduz estratégias de implantação eficazes. Estas estratégias desempenham um papel fundamental no processo de garantir que as atualizações de software são inovações sem problemas, minimizando o impacto negativo nos usuários finais (Duvall; Matyas; Glover, 2007).

1.1 Contextualização e Problema

Embora as práticas de CI e CD tenham se tornado fundamentais para o desenvolvimento de software, muitas organizações ainda enfrentam desafios significativos ao implementá-las, lidando até mesmo com a execução dos processos manualmente, atrasando o rendimento da equipe e o feedback do usuário final.

Nesse contexto, este trabalho se concentra na criação de um *Pipeline*, que nada mais é um conjunto de etapas referentes às práticas de CI/CD para o *SigArte*, Sistema de Informação

Gerencial (SIG) da Casa do Artesão do Seridó (CARTS) em desenvolvimento. O *SigArte* é uma ferramenta criada para digitalizar os processos da CARTS, escrita utilizando a linguagem de programação Python e o *framework* Django, facilitando a administração das ações e recursos da organização.

O *SigArte* é um sistema já em produção mas que tem como características a alta demanda de requisitos por parte dos usuários. A equipe de desenvolvedores consegue lidar bem com o desenvolvimento, mas a implantação e validação do código submetido ainda é um gargalo pelas dependências físicas e processos ainda manuais que trazem um atraso ao usuário final.

Os desafios que este projeto aborda incluem a necessidade de implementar um Pipeline CI/CD em um ambiente de desenvolvimento, visando suprir as demandas com mais agilidade e qualidade onde a integração de código e a entrega de software ocorrem de maneira lenta e não automatizada. Além disso, a aplicação de estratégias eficazes de implantação é essencial para garantir que as atualizações do *SigArte* sejam implementadas de maneira veloz e suave, minimizando qualquer impacto nos usuários finais.

A implantação ocorre com tempo de inatividade e sempre durante o horário de funcionamento do estabelecimento, pois o membro da equipe responsável precisa estar fisicamente presente na CARTS e conectado à rede local para acessar o servidor de produção. Isso limita a liberação de novas funcionalidades e correções, restringindo a flexibilidade e a agilidade do processo de atualização do sistema.

1.2 Objetivos

1.2.1 Objetivo Geral

Desenvolver um pipeline CI/CD, utilizando as ferramentas Jenkins e Kubernetes, com o intuito de aumentar a frequência de entrega e disponibilidade do sistema do *SigArte*, o SIG da Casa do Artesão do Seridó.

1.2.2 Objetivos Específicos

- Desenvolver um pipeline de integração contínua (CI) que automatize a compilação, testes e verificação de qualidade do código-fonte do *SigArte* sempre que houver contribuições dos desenvolvedores.
- Implementar um *pipeline* de implantação contínua (CD), que garanta que as atualizações do *SigArte* sejam entregues eficientemente e com disponibilidade contínua.
- Registrar todas as etapas do desenvolvimento do *pipeline* CI/CD, documentando as estratégias de implantação, os resultados dos testes e as melhorias na entrega e disponibilidade do sistema.

1.3 Delimitação do Estudo

Neste trabalho, será desenvolvido um pipeline CI/CD e implementada uma estratégia de implantação utilizando o Jenkins, um servidor de automação de código aberto, e o Kubernetes, uma plataforma de orquestração de contêineres que também é um software livre.

O estudo se baseará em ferramentas e conceitos disponíveis gratuitamente, com preferência para soluções de código aberto. O hardware utilizado terá especificações modestas, e o processo será projetado para ser facilmente replicável em computadores pessoais e servidores com recursos computacionais limitados.

Vale ressaltar, também, que existe outro conceito intitulado *Continuous Deployment* ou Implantação Contínua, que se assemelha bastante com o CD. Portanto, neste trabalho, os dois conceitos serão tratados como apenas CD e suas diferenças serão explicadas mais a frente.

1.4 Justificativa

O presente trabalho se justifica pelo apoio na automação do processo de CI/CD e na melhoria de qualidade na entrega da aplicação, de modo a garantir um código mais limpo e testado, que não afete a usabilidade e esteja disponível mais rápido para o usuário final. Tendo as seguintes motivações:

1. Avaliar o desempenho da automatização de processos de CI/CD em um ambiente real de desenvolvimento.
2. Servir como base para construção de pipelines de CI/CD para aplicações escritas em Django.
3. Estimular e promover as estratégias de automação para processos da engenharia de software.

1.5 Apresentação do Trabalho

O restante deste trabalho está estruturado em capítulos: no Capítulo 2, é apresentada a fundamentação teórica da proposta, que contém toda a base teórica necessária para o entendimento dos conceitos que são utilizados, além das ferramentas e a apresentação do *Sigarte*. No Capítulo 3, é apresentada a metodologia a ser seguida para o desenvolvimento do trabalho. No Capítulo 4, você poderá saber mais sobre o desenvolvimento da pesquisa. No Capítulo 5, serão expostos os resultados. Por fim, no Capítulo 6, a conclusão.

2 Fundamentação Teórica

Este capítulo tem como objetivo estabelecer o alicerce conceitual necessário para compreender as abordagens feitas neste trabalho.

Abordaremos os seguintes tópicos-chave: Na primeira seção, serão tratados alguns conceitos sobre engenharia de software e como esta se relaciona com CI/CD.

Na segunda seção, são apresentados diferentes tipos de testes de software e sua importância nos cenários reais de desenvolvimento de software. Na terceira seção, será apresentado o conceito de análise de código estático e como essa prática influencia na qualidade de software.

Na quarta seção, exploraremos a CI, uma prática fundamental que envolve a integração frequente de alterações de código, testes automáticos e garantia de que o software está sempre em um estado funcional.

Em seguida, na quinta seção, abordaremos a CD, que se concentra na automação de processos que permitem a entrega eficaz de software em ambientes de produção, garantindo que as alterações possam ser implementadas de maneira confiável e eficiente. No sexto tópico, detalharemos o que é um pipeline e como são aplicados nessa automação de processos.

A sétima seção se concentrará no balanceamento de carga, detalhando estratégias e práticas para distribuir eficazmente o tráfego em sistemas distribuídos, garantindo alta disponibilidade e desempenho. Na oitava seção, abordaremos estratégias de implantação, destacando diferentes abordagens, como implantação gradual, implantação canário e *rollbacks*, essenciais para a entrega confiável de software em produção.

Na nona seção, são apresentadas as ferramentas a serem usadas, bem como seus objetivos e benefícios.

Por fim, na décima primeira seção, realizaremos uma revisão dos trabalhos relacionados, contextualizando pesquisas relevantes e estudos anteriores que contribuem para o campo de estudo abordado neste trabalho.

2.1 Engenharia de Software

A Engenharia de Software é uma área de estudo que desempenha um papel muito importante no escopo do desenvolvimento e manutenção de sistemas de software. É um campo que evoluiu rapidamente nas últimas décadas para se adaptar as crescentes demandas tecnológicas. Segundo Pressman e Maxim (2014):

> A engenharia de software é o estabelecimento e a utilização de princípios sólidos de engenharia para obter software econômico, confiável e que funcione

> eficazmente em máquinas reais, é a aplicação de uma abordagem sistemática, disciplinada e quantificável ao desenvolvimento, operação e manutenção de software (Pressman; Maxim, 2014).

A área de Engenharia de Software é marcada pela diversidade de metodologias e abordagens. Uma das transformações mais notáveis nesse campo foi a ascensão das metodologias ágeis. As metodologias ágeis, como o *Scrum* e o *Kanban*, introduziram princípios que enfatizam a colaboração, a entrega contínua de valor, a adaptação a mudanças e a ênfase na qualidade do software. Essas metodologias revolucionaram como os projetos de software são gerenciados e executados.

Ademais, a cultura *DevOps* surgiu como uma resposta à necessidade de integração contínua e entrega contínua de software. O *DevOps* é uma abordagem que enfatiza a colaboração entre as equipes de desenvolvimento e operações, promovendo uma integração mais estreita entre o desenvolvimento de software e a infraestrutura de operações. Essa integração resultou na necessidade de automatizar muitos dos processos que tradicionalmente eram executados manualmente, atrasando a entrega de software.

Aqui é onde entra a relação direta entre a Engenharia de Software e as estratégias de CI/CD. As metodologias ágeis introduzem conceitos como entregas contínuas e adaptação a mudanças, que se alinham perfeitamente com os princípios do CI/CD. A CD visa garantir que o software esteja sempre em um estado funcional e pronto para implantação, o que é uma extensão direta da entrega de valor ao cliente, um dos princípios centrais das metodologias ágeis.

Além disso, o *DevOps* enfatiza a automação e a colaboração entre equipes, princípios essenciais para a implementação bem-sucedida do CI/CD. Os pipelines de CI/CD são projetados para automatizar a integração, teste e implantação de software, permitindo que as equipes de desenvolvimento entreguem valor rapidamente, em sintonia com os objetivos do *DevOps*.

2.2 Testes de Software

Testes de software são um elemento fundamental da Engenharia de Software e considerados elementos cruciais nos processos de desenvolvimento, desempenhando um papel crítico na garantia da qualidade, confiabilidade e desempenho de sistemas de software e estão diretamente ligados ao processo de CI. De acordo com Pan (1999):

> Dentre os objetivos dos testes de software estão a garantia da qualidade, a verificação e validação ou a estimativa da confiabilidade do código em questão. Os testes também podem ser utilizados como uma métrica genérica, e podem ser classificados como um compromisso entre orçamento, tempo e qualidade, pois é importante também para facilitar a vida do desenvolvedor em busca de problemas no código (Pan, 1999).

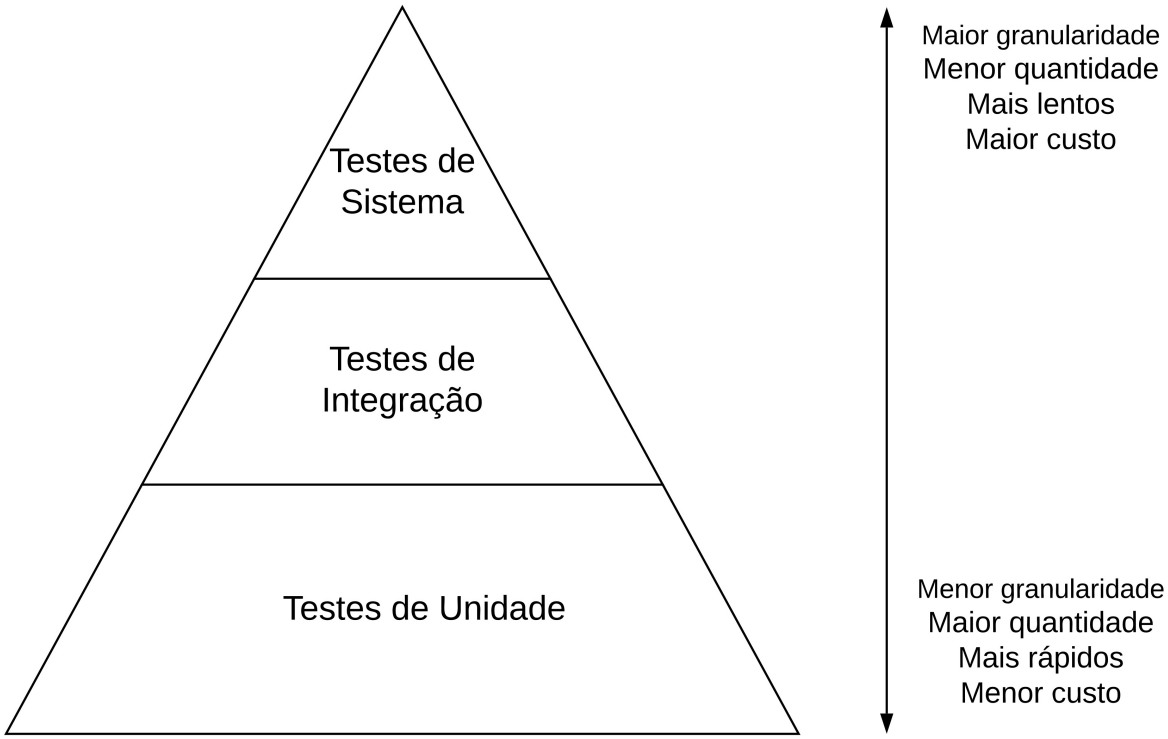
Os testes de software são processos sistemáticos que visam avaliar se um software está funcionando conforme o esperado e se atende aos requisitos estabelecidos. Eles desempenham

um papel crucial na detecção de defeitos, vulnerabilidades e problemas de desempenho antes que o software seja implantado em produção, ou seja, no momento de sua integração ao resto do projeto. A implementação de testes rigorosos é essencial para garantir que os sistemas de software funcionem de maneira confiável e atendam às expectativas dos usuários. Neste trabalho, a execução de testes de software automatizados serão um dos principais passos do Pipeline do CI a ser implementado.

Segundo Valente (2020):

> Particularmente, os testes são divididos em três grupos. Testes de unidade são simples, mais fáceis de implementar e executam rapidamente. No próximo nível, temos testes de integração ou testes de serviços, que verificam uma funcionalidade ou transação completa de um sistema. Logo, são testes que usam diversas classes, de pacotes distintos, e podem ainda testar componentes externos, como bancos de dados. Testes de integração demandam mais esforço para serem implementados e executam de forma mais lenta. Por fim, no topo da pirâmide, temos os testes de sistema, também chamados de testes de interface com o usuário. Eles simulam, da forma mais fiel possível, uma sessão de uso do sistema por um usuário real (Valente, 2020).

Figura 1 – Pirâmide de testes



Fonte: (Valente, 2020). Adaptado de Cohn (2009)

Utilizando a pirâmide mostrada na Figura 1, podemos criar uma ordem para saber em que momento cada tipo de teste deverá aplicado durante nosso pipeline, visando melhor desempenho e prioridade para os testes considerados mais cruciais.

É notável que testes de software não é um tema simples e fácil de ser abordado, são muitas as opções e conceitos por trás dessa área essencial da engenharia de software, variados os escopos do projeto onde cada tipo de teste pode ser aplicado. Sua importância para o sucesso de projetos de software é indiscutível, quanto mais cobertura de testes o sistema apresentar, melhor será de identificar possíveis falhas, sem contar que códigos testáveis são consequentemente mais limpos e de mais fácil compreensão. Ademais, é importante ressaltar que alguns deles podem ter seus benefícios potencializados pelas estratégias de CI, que isentam o desenvolvedor de se preocupar com a execução dos testes que podem ser automatizados.

Em concordância com os estudos de Pressman e Maxim (2014) e o trabalho de Hooda e Chhillar (2015), existem vários tipos de testes de software, cada um com um foco específico. Aqui será dado alguns exemplos dos mais usados e amplamente difundidos na engenharia de software:

- **Testes de Unidade:** Testes de baixo nível focados em unidades individuais de código, geralmente escritos pelos desenvolvedores para verificar a funcionalidade de métodos ou funções isoladas.
- **Testes de Integração:** Avaliam a interação entre diferentes módulos ou componentes do software para garantir que funcionem bem juntos.
- **Testes de Aceitação:** Verificam se o software atende aos requisitos funcionais especificados, frequentemente realizados com as partes interessadas.
- **Testes de Desempenho:** Avaliam a eficiência do software em termos de carga, tempo de resposta e capacidade de suportar condições de uso reais.
- **Testes Estruturais e Funcionais:** Testes estruturais (caixa branca) verificam o funcionamento interno do software, enquanto os funcionais (caixa preta) focam na validade das saídas para entradas específicas.
- **Testes de Mutação:** Modificam o código fonte para criar "mutantes"e medem a eficácia dos testes pela capacidade de detectar essas mudanças.

2.3 Análise Estática de Código

A análise estática de código é outra prática essencial no contexto da engenharia de software, pois ferramentas com este poder conseguem detectar possíveis problemas no código-fonte.

Conforme JetBrains (2023):

> Uma ferramenta de análise estática verifica o código em busca de erros e vulnerabilidades comuns, como vazamentos de memória ou estouros de *buffer*. A

> análise também pode impor padrões de codificação. No entanto, a análise estática só pode identificar instâncias em que as regras programadas são quebradas: ela não pode encontrar todas as falhas apenas lendo a fonte. Também existe o risco de falsos positivos e, por isso, os resultados precisam ser interpretados (JetBrains, 2023).

A análise de código estático trata apenas do código-fonte estático, ou seja, o mesmo não precisa ser executado. Mas, apesar de não analisar o software dinamicamente, ela pode fornecer *insights* sobre várias métricas e nos ajudar a encontrar problemas, vulnerabilidades, *code smells* (más práticas de codificação que podem indicar problemas subjacentes), além de acusar duplicidade de código e baixa cobertura de testes.

Além disso, a análise de código estático ajuda a medir métricas importantes, como a complexidade ciclomática, que avalia a complexidade do fluxo de controle do programa, e o acoplamento e coesão, que medem a interdependência entre módulos e a coesão interna destes. Também é possível detectar o débito técnico, que se refere ao custo adicional de manutenção gerado por escolhas de design falhas ou código defeituoso.

Algumas ferramentas também garantem verificar se o código está em conformidade com os padrões estabelecidos para o projeto, como identação, convenções de nomenclatura, entre outros detalhes revelados pelo nosso código-fonte, o que assegura um código mais legível e consistente.

Portanto, este tipo de análise é um processo fundamental, que pode agregar no objetivo de garantir qualidade de código e segurança. E claro, é facilmente automatizado por um pipeline de CI.

2.4 Integração Contínua

Continuous Integration ou Integração Contínua (CI) é uma estratégia fundamental nas mais modernas empresas de desenvolvimento de software. De acordo com Fowler (2006):

> A Integração Contínua é uma prática de desenvolvimento de software em que os membros de uma equipe integram o seu trabalho frequentemente, normalmente cada pessoa integra pelo menos diariamente - levando a várias integrações por dia. Cada integração é verificada por uma construção automatizada (incluindo testes) para detectar erros de integração o mais rapidamente possível. Muitas equipes consideram que esta abordagem conduz a uma redução significativa dos problemas de integração e permite que uma equipe desenvolva software coeso mais rapidamente (Fowler, 2006).

A CI abrange algumas importantes etapas para a integração contínua de código ao repositório central, e de acordo com os artigos Atlassian (2023a) e Atlassian (2023b), opera sob os seguintes princípios essenciais:

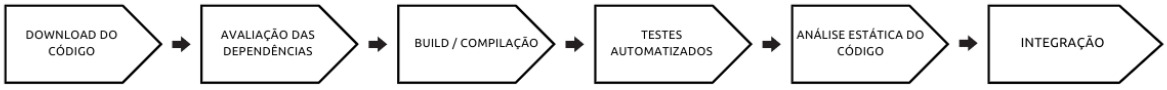
Integração Regular: Os desenvolvedores enviam suas alterações de código para o repositório compartilhado várias vezes ao dia, em vez de manter grandes ramificações de desenvolvimento separadas por longos períodos.

Automação: A integração é realizada automaticamente por meio de sistemas de build e servidores de integração contínua.

Testes Automatizados: Os testes de software automatizados desempenham um papel crucial na CI, uma vez que garantem que as alterações recentes não introduziram defeitos no software existente.

Feedback Imediato: A CI fornece feedback imediato aos desenvolvedores, alertando sobre problemas de integração ou testes quebrados, o que permite uma correção mais rápida.

Figura 2 – Exemplo de estágios de um pipeline CI



Fonte: Autor

Na Figura 2, é possível ver um exemplo dos estágios presentes em um pipeline de CI.

Além disso, a CI pode trazer outros benefícios como: permitir maior escalabilidade, melhoria do ciclo de feedback, simulação de servidor real para a execução das etapas que antes eram feitas na máquina do desenvolvedor e aumentar a comunicação entre as equipes de operações e desenvolvimento.

Ademais, ter um pipeline que implementa essas ideias isenta o desenvolvedor da execução desses processos manualmente, garantindo que os códigos enviados sempre passem por estes estágios.

Esta prática também está diretamente alinhada com os princípios das metodologias ágeis, que enfatizam a entrega de valor contínua e a adaptação às mudanças. Isso se traduz em entregas de software frequentes e confiáveis, o que é uma extensão lógica dos princípios da CI.

2.5 Entrega Contínua

A Entrega Contínua CD ou Continuous Delivery é outra prática muito importante que caminha lado a lado com a CI, mas que se concentra na automação dos processos de entrega e implantação de software, permitindo a entrega de versões funcionais do software de maneira rápida e consistente e visa garantir que o software seja implantado de maneira confiável e repetível em diferentes ambientes, incluindo produção.

Segundo Itkonen *et al.* (2016):

> Na entrega contínua, o objetivo é que cada recurso passe pelo pipeline de integração e implantação, resultando em um produto imediatamente implantável em diversos ambientes. Essa prática foi proposta para acelerar a entrega de valor, melhorar a qualidade do software e aumentar a produtividade do desenvolvedor, que rapidamente terá seu desenvolvimento utilizado por outros usuários (Itkonen *et al.*, 2016).

Vale ressaltar que existe um outro conceito também bem próximo da CD e que, na verdade, ele é o responsável por se referir mais especificamente a processos de implantação, chamado **Continuous Deployment**, ou **Implantação contínua**, que também é abreviado por **CD**.

Neste trabalho, trataremos tanto da entrega quanto da implantação como CD. Dada as poucas diferenças entre os conceitos e as inúmeras ligações, onde são implementadas, na maioria das vezes, como uma só estratégia em conjunto com a CI.

A entrega contínua possui algumas diferenças da implantação contínua. A primeira busca preparar o artefato para ser implantado, ou seja, não lida com a implantação em si, mas resulta em um código pronto para ser implantado. Já a implantação contínua, realmente cuida do processo de implantação do software em diferentes ambientes, como o de testes, homologação e produção.

A principal função da CD é automatizar processos de entrega de software. Antes mesmo de chegar no ambiente de produção, a CD pode garantir que o sistema tenha sido implantado em outros ambientes, como de testes, arbitrariamente semelhantes aos de produção, o que permite que o software seja submetido a cenários reais, poupando-nos de trabalho repetitivo e melhorando a possibilidade de *feedbacks* mais velozes.

Figura 3 – Exemplo de estágios de um pipeline CD

Fonte: Autor

Na Figura 3, podemos ver um exemplo dos estágios presentes em um pipeline de CI.

Seguindo essa abordagem, conseguimos diminuir bastante o tempo desde que as modificações de código são aprovadas até estarem implantadas em um servidor real, contribuindo para a agilidade do ciclo de desenvolvimento.

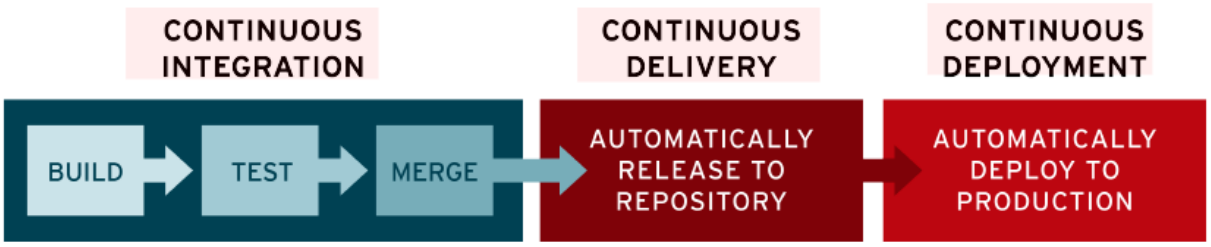
2.6 Pipeline

O conceito de pipeline se refere a nada mais que um conjunto de etapas ou estágios, onde cada um deles é responsável por automatizar um determinado processo.

Trazendo para o contexto da CI e CD, um pipeline pode ser implementado de forma que cada etapa realize um processo com o objetivo final de construir, testar e implantar o software de maneira eficiente e confiável.

Podemos exemplificar alguns dos estágios como: *build* da aplicação, execução de testes, criação de uma imagem *Docker* e implantação.

Figura 4 – Exemplo de um pipeline CI/CD



Fonte: Red Hat (Hat, 2021)

A Figura 4 descreve bem as etapas de um pipeline CI/CD, começando pela fase de integração contínua, que conta com as etapas de construção, testes e mesclagem do código, logo após, a fase de entrega contínua onde ocorre a liberação automática da nova versão do software no repositório e, por fim, a fase da implantação contínua onde o sistema é automaticamente liberado em produção.

Consequentemente é uma abordagem a ser adotada em projetos de software juntamente com a CI e a CD por trazer vantagens. No entanto, é válido ressaltar que dificilmente será criado um pipeline para determinada aplicação de forma que este se mantenha inalterado por muito tempo.

Um pipeline CI/CD sempre deve acompanhar as mudanças tecnológicas e evoluir a medida que o software e sistemas subjacentes também avançam. Por isso, precisamos sempre estar atentos a mudanças tecnológicas que afetem nossas estratégias de entrega e implantação (Zampetti *et al.*, 2021).

2.7 Estratégias de Implantação

A implantação de software é outra fase crítica para o sucesso do sistema. Ela está focada na distribuição e na disponibilidade do software. Estratégias de implantação eficazes desempenham um papel fundamental na garantia de que as atualizações de software sejam entregues com sucesso, sem interrupções nos serviços existentes.

Os autores Humble e Farley (2010), apresentam inúmeros conceitos sobre entrega contínua de software, dentre eles, várias estratégias de implantação são listadas. São as principais:

Replace (Recreate) Deployment: Este é o modelo de implantação mais simples de todos e não está de acordo com o objetivo de manter a disponibilidade do sistema pois apresenta tempo de inatividade. A estratégia consiste apenas em desligar o ambiente atual e religá-lo depois de atualizado. É o mais simples de ser implementado, mas pode gerar impacto no usuário final enquanto o servidor estiver atualizando.

Rolling Deployment: Realiza implantação em ondas, para isto, é necessário mais de um servidor em produção. A estratégias consiste em remover um servidor do balanceamento de carga para que haja a atualização, e depois, este passa a ser considerado novamente pelo balanceamento, seguindo o mesmo processo para todos os servidores desatualizados até que todos estejam na mesma versão.

Blue-Green Deployment: Nesta estratégia é necessário ter dois ambientes, um de produção e outro de pré-produção (*staging*). De acordo com Hat (2024), um ambiente (azul) executa a versão atual da aplicação e o outro (verde) executa a nova versão. Essa estratégia aumenta a disponibilidade da aplicação e reduz os riscos de implantação. Após os testes no ambiente verde, o tráfego da aplicação é direcionado para ele e o ambiente azul é desativado, podendo ser mantido por algum tempo para um possível *rollback*.

Canary Deployment: Essa abordagem consiste em atualizar pouco a pouco os servidores disponíveis, sem gerar tempo de inatividade. Após os servidores escolhidos serem atualizados e não ocorrer falhas, o processo é feito para os demais. Para esta técnica, a aplicação deve conseguir funcionar com as duas versões diferentes, já que alguns usuários estarão usando a nova versão enquanto outros ainda serão redirecionados para a antiga versão.

Além disso, também precisamos de um balanceamento de carga que distribui uniformemente o tráfego de rede entre servidores, instâncias de aplicativos ou recursos computacionais, evitando a sobrecarga de um único servidor e assegurando desempenho consistente e maior disponibilidade do sistema.

Segundo Cardellini, Colajanni e Yu (1999), uma solução eficaz para balanceamento de carga é uma arquitetura distribuída que roteia as solicitações entre múltiplos servidores de forma transparente, aumentando a taxa de transferência e escalabilidade dos sistemas. Diferentes estratégias de balanceamento utilizam métricas de rede e software para otimizar o redirecionamento de requisições, garantindo alta disponibilidade e desempenho.

Em síntese, existem várias estratégias de implantação para lidar com as peculiaridades de cada sistema. Utilizando desses conceitos, conseguimos muita flexibilidade na hora de disponibilizar uma nova versão do software para nossos usuários. É válido ressaltar que muitas dessas estratégias dependem do monitoramento das aplicações em produção, para garantir a confiabilidade do sistema.

2.8 Ferramentas

Esta seção visa apresentar as ferramentas e tecnologias selecionadas para o desenvolvimento deste trabalho, destacando as características e os benefícios que fundamentaram suas escolhas.

2.8.1 Docker

O *Docker* é uma ferramenta para gerenciamento de contêineres utilizada no mundo de desenvolvimento de software, ao conseguir abstrair algumas camadas de software que antes levavam tempo para serem configuradas de acordo com cada ambiente.

Conforme a documentação Docker (2023):

> *Docker* é uma plataforma aberta para desenvolvimento, envio e execução de aplicativos. O *Docker* permite separar seus aplicativos da infraestrutura para que você possa fornecer software rapidamente. Com o *Docker*, você pode gerenciar sua infraestrutura da mesma forma que gerencia seus aplicativos. Ao aproveitar as metodologias do *Docker* para enviar, testar e implantar código, você pode reduzir significativamente o atraso entre escrever o código e executá-lo na produção (Docker, 2023).

Com o *Docker*, temos a possibilidade de executar uma aplicação em um ambiente isolado e leve, chamado de Contêiner. Desta forma, conseguimos executar vários contêineres em um mesmo *host* e compartilhar estes contêineres com outras pessoas, tendo a certeza de que a aplicação que está dentro desta camada irá se comportar da mesma forma em diferentes máquinas.

No nosso projeto, precisaremos conhecer também o conceito de *Dockerfile* e o *Docker Compose*. Segundo Wu *et al.* (2020) "o *Dockerfile* é um arquivo que especifica as configurações necessárias para criação de uma imagem *Docker* a ser executada em contêiner". Utilizamos essa abordagem para realizar a criação de um contêiner com a imagem do nosso projeto.

Já o *Docker Compose* é outra ferramenta que existe dentro do *Docker* e é utilizada para configurar e executar um ou mais contêineres em conjunto, tudo isso por meio de um arquivo de configuração *.yaml*. Ao final deste trabalho, o *Docker Compose* será substituído por uma nova ferramenta de execução de contêineres explicada a seguir.

O *Docker* e seus conceitos serão ferramentas fundamentais para o desenvolvimento do nosso *pipeline*, tanto na fase de CI quanto na de CD para todos os ambientes. Os arquivos Dockerfile e *Docker* Compose utilizados no *Sigarte* são explicados no Apêndice C.

2.8.2 Jenkins

Como apresentado no site oficial do *Jenkins*, este é um servidor de automação auto-hospedado de código aberto, que permite automatizar vários processos, os quais podem estar

relacionados com a criação, testes, entrega e implantação de software (Jenkins, 2023). Este pode ser instalado em qualquer máquina que tenha o Java Runtime Environment (JRE) instalado, ou por meio de contêineres *Docker*.

O *Jenkins* oferece integração com várias ferramentas de compilação, provedores de nuvem, ferramentas de análise e muitas outras por meio de *plugins*, desenvolvidos pelos membros da comunidade *Jenkins* para atender às necessidades específicas das organizações ou usuários. Normalmente, para acessar serviços externos como GitHub, SonarQube e o *cluster* Kubernetes, precisamos adicionar credenciais no escopo do projeto do Jenkins.

A forma que utilizaremos para configurar o *pipeline* do nosso projeto no *Jenkins* é por meio de um arquivo de configuração chamado *Jenkinsfile* e configurações de comportamentos disponíveis na própria interface gráfica do Jenkins. O *Jenkinsfile* é um arquivo de *script* que descreve as etapas e configurações do *pipeline*. "Ele deve ser armazenado no repositório do código da aplicação sendo usado pelo *Jenkins* para executar as tarefas especificadas" (Santos, 2023).

Por ser uma ferramenta auto-hospedada e de código aberto, o *Jenkins* nos dá mais liberdades para realizar as automações dos processos necessários em comparação a outras ferramentas de CI/CD. Ele pode ser integrado com a maioria das plataformas de hospedagem de código do mercado e dispõe de interface gráfica, que é útil tanto para a utilização dos desenvolvedores quanto para a execução de testes de usabilidade automatizados.

2.8.3 SonarQube

O Sonar é um kit de ferramentas de análise estática de código, como explicado na Seção 2.3, que visa garantir a qualidade de código focando em conceitos de código limpo. O Sonar oferece algumas opções de ferramentas para garantir a qualidade do código desenvolvido, de acordo com padrões pré-definidos e algumas métricas de qualidade de software, são elas:

SonarLint: É uma ferramenta que pode ser integrada à IDE para o código ser analisado à medida que é escrito, assim como um corretor ortográfico.

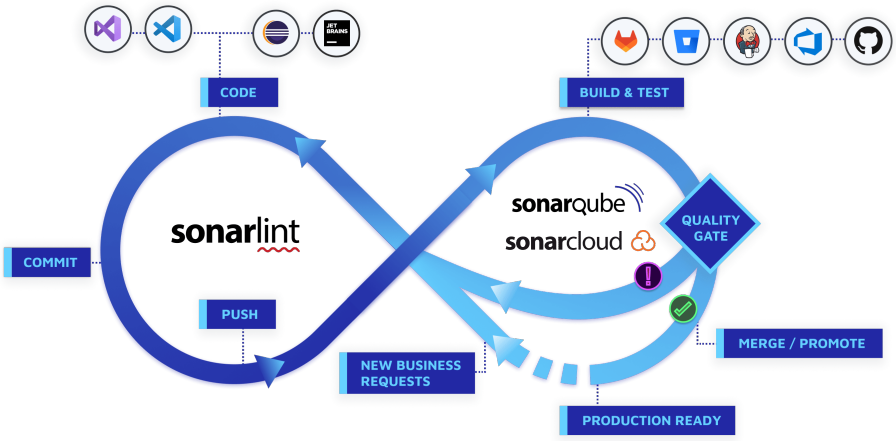
SonarQube: É uma ferramenta auto-hospedada que pode ser implantada em algum servidor próprio para o código ser submetido e analisado a partir deste servidor.

SonarCloud: É uma ferramenta para análise estática de código assim como as outras, porém, baseada em nuvem, que não exige instalação em uma máquina própria e oferece integração com outros serviços de nuvem.

Estas ferramentas são extremamente semelhantes, diferenciando-se somente na etapa de desenvolvimento onde são aplicadas e como são utilizadas, como exemplificado na figura 5. Neste trabalho utilizaremos a opção auto-gerenciada, SonarQube, para analisarmos o código em uma das etapas do nosso *pipeline* de CI e receber feedback sobre vulnerabilidades, *code smells*,

duplicação de código, entre outras.

Figura 5 – Fluxo de trabalho do Sonar



Fonte: Documentação do SonarQube (S.A, 2023)

2.8.4 Kubernetes

O Kubernetes (K8s) é outra ferramenta de código aberto que busca facilitar inúmeros processos de operações e infraestrutura de software, como implantação, escalonamento e orquestração de contêineres Linux.

No seu site oficial (Foundation, 2023), conseguimos encontrar algumas das suas características principais:

Implantação e *rollback* automatizado: O Kubernetes é configurável para implantar novas versões de software automaticamente, além de monitorar a saúde da aplicação e fazer um *rollback* para uma versão mais antiga caso seja necessário.

Balanceamento de carga: O *Kubernetes* é capaz de fazer balanceamento de carga através de *Pods*, que são uma coleção de contêineres.

Auto-recuperação: A auto-recuperação ou *self-healing* é a possibilidade de reiniciar ou recriar contêineres que falharam para mantê-lo em um estado desejado.

Escalonamento horizontal: O *Kubernetes* consegue escalar aplicações horizontalmente, replicando contêineres para suprir as demandas do sistema naquele momento.

O K8s organiza aplicativos em unidades chamadas *pods*, que são as menores unidades de trabalho no *Kubernetes*. Um *pod* pode conter um ou mais contêineres que compartilham o mesmo ambiente de rede e armazenamento.

Além disso, para garantir que uma aplicação esteja sempre disponível, o Kubernetes usa um *ReplicaSet*. É o recurso responsável por manter um número desejado de réplicas de *pods* rodando. Se um *pod* falhar ou encerrar por algum motivo, o *ReplicaSet* automaticamente cria um novo para substituí-lo.

Acima do *ReplicaSet*, ainda temos os *Deployments*, que fornecem uma maneira declarativa de gerenciar atualizações de aplicativos. Quando definimos um *Deployment*, temos que especificar a configuração e estratégia de atualização desejada. O K8s cuida de ajustar o ambiente para atender a essa configuração, inclusive gerenciando a atualização para atualizar os *pods* sem *downtime*, atráves da criação de novos *ReplicaSet*.

Por fim, um *cluster* em Kubernetes consiste em um conjunto de máquinas (nós) que executam suas aplicações de forma distribuída. O *cluster* é gerenciado por um controlador, chamado *control plane*, responsável por orquestrar algumas decisões, como o gerenciamento de *pods*, a replicação de aplicações e o balanceamento de carga entre os nós. Para configurar o *cluster*, optamos por utilizar o K3s, uma ferramenta leve e de fácil instalação, ideal para ambientes locais de um único nó e com recursos limitados, semelhante às alternativas como *Kind* e *Minikube*.

A interação com o *control plane* será realizada através da ferramenta de linha de comando *kubectl*. O *kubectl* permite a execução de comandos para gerenciar e operar os recursos no *cluster*, facilitando a administração e a automação das operações de implantação e configuração.

Ademais, o *Kubernetes* é completo e oferece muitas possibilidades de gerenciar aplicações em contêineres, possui uma forma de integração com o *Jenkins* por meio de *plugins*, que será útil para criação do *pipeline* CD e automatizar implantações nos servidores de produção e testes.

2.8.5 Python e Pip

O Python é uma linguagem de programação de alto nível, amplamente utilizada devido à sua simplicidade e ampla gama de bibliotecas. Sua versatilidade a torna adequada para uma variedade de tarefas, desde *scripts* simples até o desenvolvimento de aplicações complexas.

O *pip* é o gerenciador de pacotes padrão para Python, facilitando a instalação e atualização de bibliotecas e dependências necessárias para projetos. No pipeline de integração e entrega contínua, o *pip* é empregado para instalar as dependências listadas no arquivo `requirements.txt` e outras bibliotecas necessárias para a execução das etapas do *pipeline*. Isso assegura que o ambiente de execução seja consistente e reproduzível em diferentes fases do processo de desenvolvimento e implantação.

2.9 Trabalhos Relacionados

Esta seção visa contextualizar trabalhos semelhantes a este, que utilizam técnicas de CI/CD para acelerar a integração e entrega de software, além de garantir alta disponibilidade para o usuário final. Serão apresentados três trabalhos relacionados, seus principais objetivos e comparações com o presente trabalho.

Os trabalhos abordados incluem respectivamente: (Santos, 2023), (Zampetti *et al.*, 2021) e (Santos; Alcântara, 2022).

O estudo conduzido por Santos (2023) focaliza na aplicação prática da CI/CD na plataforma de ações solidárias "Faz um Bem!". A metodologia adotada permitiu a exploração da arquitetura do projeto e seus testes automatizados, fornecendo *feedbacks* sobre a qualidade do código e o funcionamento eficiente da aplicação de maneira ágil, facilitando a manutenção e a evolução contínua do sistema. Ao final, observou-se que a estratégia de CI proporcionou uma forma eficaz de manter a funcionalidade e a qualidade do sistema, destacando-se como uma prática relevante para o contexto da plataforma de ações solidárias.

Em comparação com o presente trabalho, ambos buscam automatizar os processos de CI/CD e introduzir melhorias e agilidade ao ciclo de desenvolvimento de software. No entanto, este trabalho abstém-se de detalhar a arquitetura e os testes automatizados do projeto, concentrando-se, em contrapartida, na experiência do usuário final ao entregar um sistema com maior disponibilidade.

No trabalho de Zampetti *et al.* (2021), percebe-se que os *pipelines* CI/CD implicam na descoberta precoce de defeitos, aumento da produtividade e ciclos de lançamento mais rápidos. Também é mencionado que a eficácia do CI/CD pode depender do quão bem esses *pipelines* são mantidos para lidar com a evolução do sistema e sua tecnologia subjacente, além de limitar práticas inadequadas.

Assim, nota-se que a análise de Zampetti *et al.* (2021) é mais minuciosa, buscando garantir uma evolução/reestruturação que alcance a mais alta performance de *pipelines* já existentes, enquanto aqui, focamos na implementação prática de um *pipeline* para um projeto específico, onde também priorizamos a performance, mas não como foco principal.

O último trabalho relacionado é o de Santos e Alcântara (2022), no qual o autor explora conceitos de contêineres Docker e o uso da ferramenta Kubernetes com foco na cultura DevOps e na entrega e disponibilidade de software. O estudo apresenta uma análise de *pipelines* altamente escaláveis, visando demonstrar os impactos positivos que um *pipeline* pode trazer em empresas, especialmente no âmbito da escalabilidade, balanceamento de carga e disponibilidade de software.

Comparado a este trabalho, ambos buscam utilizar conceitos modernos de DevOps e disponibilidade de software. No entanto, nosso foco aqui não é detalhar tanto o uso do K8s e

Docker, ferramentas que apresentam um certo nível de complexidade elevado quando exploradas em profundidade. Utilizaremos o básico delas para garantir que nosso sistema esteja sempre operacional e que as implantações ocorram sem grandes impactos ao usuário final.

Concluímos que os conceitos abordados têm alta relevância acadêmica e são amplamente difundidos no mercado de desenvolvimento de software, buscando aproveitar tais técnicas para aumentar a produtividade em ambientes tecnológicos que demandam respostas rápidas às exigências dos clientes.

3 Metodologia

O trabalho em questão se trata de um estudo de caso que envolve a implementação de um *pipeline* CI/CD para o projeto SigArte, um processo que abrange várias etapas cruciais da engenharia de software. De acordo com Santos (2023) no mercado atual existe uma demanda por uma solução que seja rápida na entrega de novas funcionalidades para o cliente e flexível para contínuas mudanças de requisitos.

Portanto, a metodologia adotada para o desenvolvimento deste *pipeline* é estruturada de forma que cada etapa seja responsável por uma tarefa específica, desde o *download* do código-fonte até a implantação do projeto em um ambiente real de produção. Esta abordagem permite uma maior eficiência e controle sobre o processo de desenvolvimento, garantindo que cada componente do projeto seja adequadamente testado e validado antes de ser integrado ao produto final.

Dessa forma, nossa metodologia irá ser dividida em partes. A primeira trata de apresentar as questões de pesquisa do estudo a ser realizado. A segunda aborda as demais etapas que serão trabalhadas no desenvolvimento do *pipeline*.

3.1 Questões de Pesquisa

Este sub-tópico visa demonstrar algumas das questões que irão nortear o desenvolvimento deste trabalho:

1. Como implementar e quais são as etapas cruciais para um *pipeline* CI/CD de uma aplicação web Django?
2. Como instalar e configurar as ferramentas para execução do *pipeline* CI/CD?
3. Como a adoção de práticas de CI/CD impacta a eficiência do processo de desenvolvimento do sistema de informação?
4. Como melhorar a performance e entrega de um *pipeline* utilizando o Jenkins?
5. Como adotar Kubernetes para realizar a atualização de um sistema monolítico sem tempo de inatividade?

3.2 Abordagem Proposta

Esta Seção está separada em seções que definem os casos de uso e comportamento do *pipeline*, bem como a arquitetura geral, tendo em vista a busca de um *pipeline* CI/CD eficiente

para o projeto da CARTS.

3.2.1 Casos de Uso e Comportamento do *Pipeline*

Esta seção define os casos de uso do *pipeline* CI/CD, especificando os eventos que acionam o *pipeline* e seu comportamento subsequente, que variam de acordo com o cenário e versionamento do projeto, explicado na Seção 4.1.1. A Tabela 1 resume os eventos que desencadeiam alguma ação no *pipeline*.

O *pipeline* será configurado para diferentes cenários: implantação em produção, implantação no servidor de testes e *builds* de CI para solicitações de mesclagem (*Pull Requests*).

Tabela 1 – Casos de Uso e Comportamento do *Pipeline*

Evento	**Ação**
Pull Request aberto para `dev` ou `main`	Integração Contínua
Mesclagem na `dev`	Implantação em ambiente interno
Liberação de nova versão (*release*)	Implantação em produção

Para a implantação em produção na máquina virtual da CARTS, quaisquer mudanças realizadas na *branch* main acionam o *pipeline* via *webhook*, um mecanismo que permite que um sistema (neste caso, o GitHub) envie uma notificação HTTP para outro sistema (como o Jenkins) quando um evento específico ocorre. Nesse contexto, o *webhook* notifica automaticamente o Jenkins para disparar a execução do pipeline de CI após qualquer mudança na *branch main*. Ao final do processo, a criação manual de uma *tag* no GitHub chamada *release* desencadeia automaticamente a implantação em produção, garantindo que a troca de versão ocorra sem tempo de inatividade (*downtime*). Essa criação de *tag* irá servir como uma confirmação da implantação em produção.

No ambiente de testes (servidor do Laboratório de Banco de Dados e Engenharia de Software (LABENS)), as mudanças na *branch* dev também acionam o *pipeline* via *webhook*. O Jenkins deve executar todo o *pipeline* CI/CD e ao final, deve realizar a implantação no ambiente de testes de forma totalmente automatizada. Este processo facilita a rápida validação das alterações antes de serem promovidas para a *branch* principal e ambiente produção.

Para PRs direcionados às *branches* main e dev, o Jenkins é notificado sobre a abertura ou atualização do *Pull Request*. O *pipeline* é então acionado e sua execução e resultado pode ser visto na tela do GitHub. Esta integração permite que os desenvolvedores recebam *feedback* imediato sobre a qualidade do código submetido.

3.2.2 Arquitetura

Um *pipeline* CI/CD é um conjunto de etapas que buscam automatizar processos da integração e entrega de software para chegar a um resultado proposto de forma confiável e eficiente. Portanto, nesta etapa, será estabelecida a proposta inicial de arquitetura ao *pipeline* CI/CD do projeto.

A configuração das etapas de um *pipeline* está diretamente ligada à natureza e às tecnologias utilizadas no desenvolvimento do projeto. É essencial considerar as especificidades do alvo onde o *pipeline* será aplicado. No caso do SigArte, o *pipeline* será implantado para atender às necessidades do ciclo de desenvolvimento e entrega deste sistema, mostrado na Seção 4.1.

Uma análise de Zampetti *et al.* (2021) sobre a organização de um *pipeline* CI/CD admite que:

> A arquitetura organiza comandos em fases, estágios ou trabalhos e sua ordem de execução. Isso inclui reestruturar fases de instalação e *script*, reorganizar a ordem de execução das etapas de compilação, reestruturar trabalhos e/ou estágios, usar compilações parametrizadas e atualizar verificações no processo de compilação. A execução de fases específicas pode ser necessária quando a compilação é acionada por uma solicitação *pull* ou quando uma alteração é feita em um *branch* específico (Zampetti *et al.*, 2021).

Desse modo, conhecendo o projeto alvo do nosso pipeline, iremos estabelecer os estágios a serem desenvolvidos tanto para o processo de CI quanto para o processo de CD, de forma que seja possível alcançar os objetivos esperados com eficiência.

O desenvolvimento das etapas do *pipeline* serão realizadas integralmente no arquivo *Jenkinsfile*, usando a forma declarativa, de maneira que este seja mantido junto ao código fonte e versionado como tal, mantendo assim, um histórico dos nossos *pipelines*.

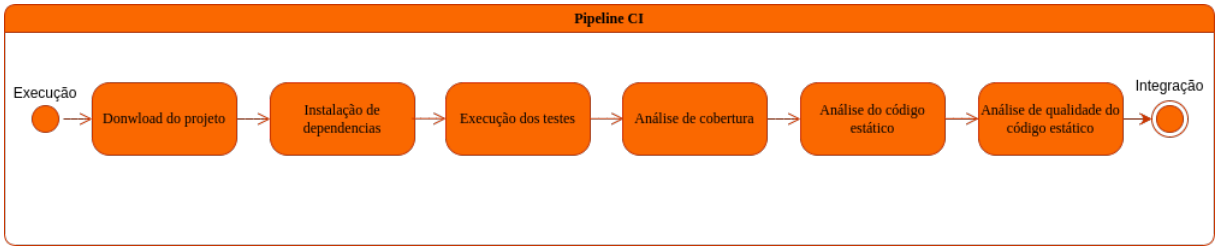
3.2.2.1 Arquitetura do *Pipeline* de Integração

A arquitetura do *pipeline* de CI, definida na Figura 6 é projetada para garantir que todas as mudanças no código sejam integradas de forma contínua e eficiente. Este *pipeline* inclui estágios definidos conforme a aplicação do SigArte. Cada etapa é automatizada para fornecer *feedback* rápido aos desenvolvedores, assegurando a qualidade do código antes da integração na *branch* principal.

Como visto no diagrama acima, iremos desenvolver seis estágios iniciais para o pipeline de integração, que podem vir a mudar futuramente de acordo com as demandas do projeto, que por enquanto são:

1. Download do projeto do GitHub.
2. Instalação das dependências do projeto.

Figura 6 – Arquitetura proposta para CI



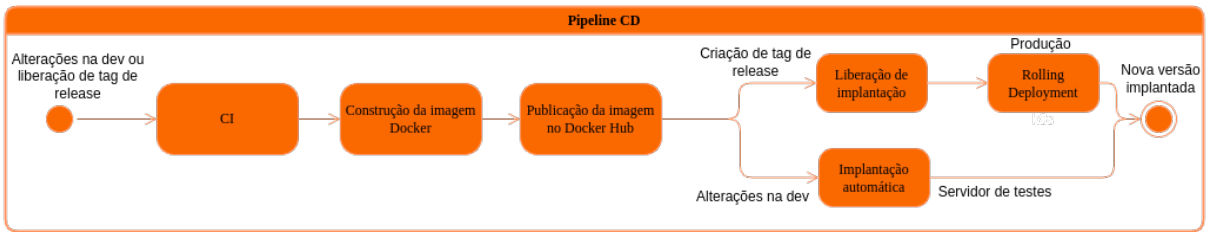
Fonte: Autor

3. Execução de testes.
4. Análise de cobertura dos testes.
5. Análise do código estático pelo SonarQube.
6. Análise de qualidade de código definida no SonarQube.

3.2.2.2 Arquitetura do Pipeline de Entrega

O pipeline de CD visa automatizar o processo de implantação das novas versões do SigArte. A arquitetura deve garantir que as mudanças sejam entregues de forma segura e eficiente, minimizando riscos e tempo de inatividade.

Figura 7 – Arquitetura proposta para CD



Fonte: Autor

Como mostrado na arquitetura proposta da Figura 7, após realizar a integração na ramificação dev ou uma nova *tag* de liberação ser criada a partir da branch main, será construída uma imagem Docker da aplicação, e a mesma será enviada para o Docker Hub, um repositório de imagens Docker, com uma *tag* que identifique a sua versão. Os passos seguintes irão considerar a *branch* em questão para decidir como será feita a implantação, seja no servidor de produção após uma liberação manual utilizando a estratégia de implantação, ou de forma automática no servidor de testes.

3.2.3 Implantação

A estratégia de implantação define como as novas versões do SigArte serão distribuídas nos diferentes ambientes por meio de contêineres Docker, orquestração e redirecionamento de

tráfego por meio do K8s. Em ambientes de testes, será permitido *deploys* automáticos com possível *downtime* para facilitar a validação das mudanças.

Portanto, considerando os objetivos propostos, a capacidade do hardware utilizado e a necessidade da CARTS, a estratégia de implantação a ser efetivada para o servidor de produção será a *rolling deployment*. Ao utilizar essa estratégia estaremos, após liberação manual da atualização, criando réplicas do ambiente do SigArte em produção, mas com a nova versão a ser lançada, somente após o ambiente subir sem falhas, o balanceador de carga irá apontar para o ambiente com a versão mais nova.

A configuração é feita através de arquivos de extensão *.YAML*, que definem os *deployments* e os serviços no Kubernetes. Um serviço (*Service*) é utilizado para rotear o tráfego de produção para a versão *v1* inicialmente. Após testar e verificar a nova versão (*v2*), o serviço é atualizado para apontar para a versão *v2*. Essa mudança é feita aplicando o arquivo *YAML* atualizado, também versionado junto ao código fonte da aplicação.

Em contraponto, por não ser um ambiente crítico, no servidor de testes não será adotada nenhuma estratégia para evitar a inatividade, assim, podemos evitar desperdício de hardware em um ambiente não crítico.

4 Desenvolvimento

Neste capítulo, será apresentado e explicado o SigArte, projeto alvo deste trabalho, bem como suas características e particularidades. Ademais, nas seções seguintes será explicado o desenvolvimento do trabalho, etapas como: configuração de ambiente, instalação das ferramentas, configuração de comportamentos do pipeline, estágios do pipeline de CI e CD e implantação.

4.1 Sigarte

O *SigArte* é um SIG que está sendo desenvolvido em parceria com o LABENS do Departamento de Computação e Tecnologia (DCT) que faz parte da Universidade Federal do Rio Grande do Norte (UFRN) para suprir as demandas de gerenciamento da Casa do Artesão do Seridó (CARTS).

O sistema está sendo escrito com a linguagem de programação Python, considerada multi-paradigma e utilizando o framework de desenvolvimento web Django (2023), por enquanto, seguindo uma arquitetura monolítica de várias camadas, e utiliza o Sistema de Gerenciamento de Banco de Dados (SGBD) PostgreSQL (2023) na camada de dados.

O *SigArte* tem como principal objetivo oferecer uma forma de controlar digitalmente as demandas da CARTS como gerenciamento de produtos da casa, vendas, entradas, cadastro e controle de artesões, funcionários, entre outras funcionalidades.

Figura 8 – Tela inicial do *SigArte*

Fonte: SigArte, 2023

O projeto foi desenvolvido seguindo vários conceitos e boas práticas de engenharia de

software, portanto, possui uma boa cobertura de testes de unidade e alguns testes de aceitação que buscam garantir a qualidade do software.

A Figura 8 apresenta uma imagem da tela inicial do *SigArte*, acessível ao público geral, esta é obtida ao acessar o site. Nesta tela temos acesso ao botão para realizar login, além de conseguir ver algumas informações sobe a CARTS, como notícias, localização, contatos, etc.

Quando logado no sistema, podemos clicar no nosso nome de usuário no canto superior direito, como mostrado na Figura 8. Dependendo do tipo de usuário com o qual estamos logados, teremos acesso ao menu principal que aparece na lateral esquerda da tela, onde temos acesso de fato ao SIG.

O menu, apresentado na Figura 9 oferece várias opções para manipulação de diferentes entidades de usuários, incluindo estagiários, artesãos, voluntários, extensionistas e clientes. Além disso, permite o cadastro de organizações que podem ser associadas aos artesãos. O sistema também disponibiliza acesso para gerenciar entidades como entrada, produto, venda e retirada, sendo as entidades responsáveis pelas funcionalidades principais do sistema.

Figura 9 – Menu do SigArte

Fonte: Sigarte, 2023

Ele também já conta com um *pipeline* CI/CD no *GitHub Actions* que agilizará alguns processos já descritos em capítulos anteriores, porém, com algumas limitações que serão resolvidas durante o desenvolvimento do trabalho.

As características importantes sobre o *SigArte* serão apresentadas ao decorrer desse capítulo, pois precisamos conhecê-las para poder desenvolver um *pipeline* eficiente.

4.1.1 Ciclo de Desenvolvimento e Versionamento

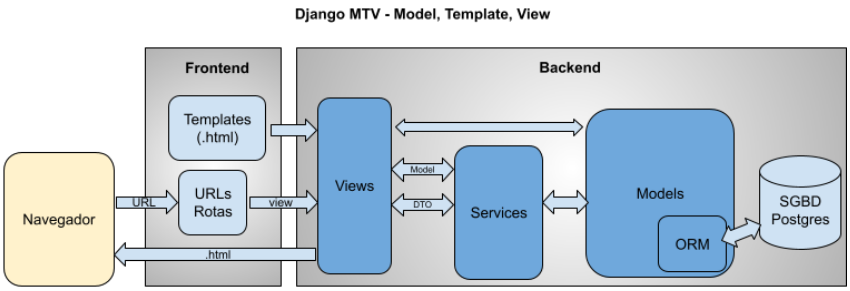
O projeto SigArte é versionado utilizando Git e GitHub, seguindo uma metodologia que distingue claramente as diferentes ramificações (ou *branches*), para gerenciar o desenvolvimento e a CI. A *branch* main é a principal, utilizada exclusivamente para o código que está pronto para ser liberado em produção. A *branch* dev é destinada ao desenvolvimento contínuo e à integração de novas funcionalidades, implantadas primeiro em um ambiente de testes interno, antes de serem promovidas para a main.

Além das *branches* principais (main e dev), o projeto adota a prática de criar ramificações específicas para o desenvolvimento de novas funcionalidades ou correções de problemas. Essas *branches* são tipicamente relacionadas a problemas documentados no GitHub, garantindo que cada mudança no código está ligada a um requisito ou problema específico. As *branches* são nomeadas de acordo com os problemas correspondentes e são direcionadas aos usuários responsáveis pela sua implementação.

Este ciclo de desenvolvimento organizado permite uma gestão eficiente das contribuições, mantendo o código base estável e garantindo que todas as modificações sejam devidamente revisadas e testadas antes de serem integradas nas *branches* principais. A integração com o GitHub permite uma rastreabilidade completa das mudanças, além de facilitar a colaboração entre os desenvolvedores.

4.1.2 Arquitetura

Figura 10 – Arquitetura do *SigArte*



Fonte: Adaptado da documentação do SigArte

Na Figura 10 podemos enxergar melhor as camadas da arquitetura do SigArte, que apesar de ser um sistema monolítico, este separado em camadas de diferentes responsabilidades que ajudam a manter o código organizado e estruturado. Resumidamente, o sistema pode ser acessado pelo usuário final por meio de um navegador, e a cada ação, uma requisição é enviada ao servidor que hospeda a aplicação, que irá processar a chamada e devolver uma nova página web.

Este modelo é uma adaptação da arquitetura Models, Templates, Views (MTV) proposta pelo Django, os componentes mais importantes dessa arquitetura são:

- **Template:** Os *templates* nada mais são do que os arquivos a serem processados pelo navegador e entregues ao usuário, possuem extensões como *.html .css* e *.js*.
- **View:** A *view* é camada encarregada de processar as requisições vindas do usuário e responder de acordo, entregando uma nova página web ou exibindo alguma mensagem informativa.
- **Service:** A camada de serviço é utilizada para executar regras de negócio. São normalmente chamadas pelas *views*, que delegam a lógica do sistema para o serviço.
- **Objeto de Transferência de Dados (DTO):** Os DTOs são utilizados como uma camada auxiliar que serve para abstrair apenas as informações mais importantes de um objeto naquele contexto. Nem sempre o usuário final pode ter acesso as nossas entidades de forma integral, assim, utilizamos os DTOs para entregar a ele somente as informações relevantes sobre ela.
- **Models:** Os modelos representam as entidades do sistema, ou seja, as classes que serão convertidas pelo Mapeamento Objeto Relacional (ORM) em tabelas salvas no banco de dados relacional.
- **ORM:** O ORM é uma camada de software, neste caso, disponibilizada pelo próprio *framework* do Django, que nos permite relacionar as entidades do sistema, sendo classes da linguagem Python com algum banco de dados relacional.
- **SGBD:** O SGBD é o software que fica responsável por armazenar todos os dados configurados na minha aplicação, conseguindo manter a integridade e conseguir devolver as informações rapidamente. Neste caso, estaremos usando o PostgreSQL (PostgreSQL, 2023).

4.1.3 Implantação

O *deploy* ou implantação consiste na disponibilização do sistema para o usuário final, ou a atualização do sistema para uma versão mais nova. O sistema ou um incremento dele, resultante

de um desenvolvimento, está pronto para entrar em produção. Essa tarefa é conhecida pelos nomes de implantação (*deploy*), liberação (*release*) ou entrega (*delivery*) do sistema (Valente, 2020).

O *SigArte* conta atualmente com uma implantação manual via Docker, explicado na Seção 2.8.1, no ambiente de produção. Precisamos:

1. Acessar remotamente o servidor no qual está hospedada a aplicação do *SigArte*.
2. Realizar o download da versão mais nova do repositório do projeto.
3. Fazer a construção da nova imagem com base no *Dockerfile* e no novo código.
4. Por fim, subir um contêiner com a imagem criada utilizando o *Docker Compose*.

Como visto acima, a implantação no ambiente de produção acontece de forma manual, simples, com presença de *downtime* e sem estratégia de *rollback*. Para o ambiente de desenvolvimento, o *pipeline* atual do *Github Actions* já realiza a implantação automática ao receber um *merge* na ramificação de desenvolvimento, já a de produção conta apenas com um *pipeline* de CI.

4.1.4 Testes Automatizados

Desde o início do projeto a equipe de desenvolvedores teve como objetivo manter uma alta cobertura de testes durante a produção do sistema, visando os benefícios que isso pode trazer na qualidade do código. A execução de testes automatizados é uma das principais etapas de um *pipeline* em qualquer sistema de informação.

Portanto, o *SigArte* conta com uma bateria de testes de unidade automatizados construídos utilizando a biblioteca *unittest*, que suporta o desenvolvimento de testes unitários e de integração, discutidos na Seção 2.2. Esta ferramenta é padrão da linguagem Python, um exemplo de teste unitário foi mostrado na Figura 19.

Além disso, o *SigArte* também possui alguns testes funcionais e de aceitação automatizados escritos em *Selenium*, exemplo na Figura 20, uma ferramenta *WebDriver* conhecida por ser capaz de simular a utilização de um usuário final, e dá possibilidades de criar não só testes de aceitação, mas também de integração e testes funcionais.

4.1.5 *Pipeline* GitHub Actions

Como já foi comentado, o *SigArte* possui um *pipeline* CI/CD simples, desenvolvido utilizando a ferramenta *Actions* do GitHub, que permite a configuração de etapas para serem executadas quando algum evento ocorre no repositório do projeto.

A ferramenta Actions é parcialmente gratuita para projetos versionados na plataforma do GitHub e visa a criação de *pipelines* automatizados para integração e entrega contínua de

software e conta com o apoio de *plugins* para realizar integrações com outros serviços que venham ser necessário no *pipeline*.

Os *scripts* de automação dos *pipelines* atualmente em uso no *SigArte* podem ser vistos no Apêndice B. Como já dito anteriormente, nosso projeto dispõe de uma implantação automática somente no ambiente de desenvolvimento, restando ainda, realizar a implantação manual no servidor de produção.

Apesar de ser integrado com o GitHub e de fácil configuração, o GitHub Actions conta com algumas limitações quando comparado a algumas outras opções no mercado, pois o servidor utilizado para a execução dos processos é da própria empresa dona da plataforma e não chega a oferecer interface gráfica que poderia ser útil execução de testes de aceitação.

4.2 Configuração de Ambiente e Ferramentas

Esta fase irá focar na configuração geral do ambiente onde nosso *pipeline* irá ser hospedado, assim como, na identificação e instalação dos softwares necessários para desenvolver este trabalho.

Iremos utilizar uma máquina com um Sistema Operacional (SO) Linux de configurações básicas para fazer a instalação das ferramentas abordadas na Seção 2.8 e alguns outros softwares básicos, como o sistema de controle de versão Git. A função desta máquina é hospedar não somente o *pipeline*, mas também uma das instâncias do SigArte.

Todo o processo será feito a partir de um computador pessoal que também utiliza Linux como SO, e algumas ferramentas básicas para o desenvolvimento do trabalho, incluindo:

- Acesso remoto à máquina que hospedará o Jenkins.
- Um editor de texto para escrever os arquivos necessários para a configuração do *pipeline* no arquivo *Jenkinsfile*.
- O Git para acessar o GitHub, onde o código fonte do projeto Sigarte está sendo versionado.

A grande parte do desenvolvimento das etapas do *pipeline* será realizada com a linguagem Groovy, de forma declarativa, dentro do arquivo chamado *Jenkinsfile*. Algumas configurações de comportamentos e *plug-ins* serão alterados na própria interface gráfica do Jenkins. Demais configurações irão tratar de arquivos *.yaml* referentes ao processo de implantação com K8s.

Os servidores destinados à implantação já possuem uma configuração de *proxy* reverso, utilizando servidores web como Nginx ou Apache, instalados diretamente no *host*. Esses servidores web são utilizados para expor as aplicações implantadas, incluindo o SigArte. Ao configurar o *proxy* reverso, podemos definir uma rota específica para a aplicação, direcionando o

tráfego para o IP do serviço correspondente dentro do *cluster* Kubernetes, garantindo, assim, o acesso externo à aplicação de maneira controlada e eficiente.

Para suportar o ciclo de desenvolvimento e implantação do SigArte, instalamos as ferramentas descritas na Seção 2.8. As ferramentas e as respectivas versões utilizadas estão detalhadas na Tabela 2.

Tabela 2 – Ferramentas Utilizadas, Descrição e Versões

Ferramenta	**Descrição**	**Versão**
Docker	Plataforma de conteinerização	27.1.1, build 6312585
Jenkins	Ferramenta de automação CI/CD	2.452.3
K3s	Distribuição leve do Kubernetes	v1.29.6+k3s2
SonarQube	Ferramenta de análise de código	9.9.3.79811
Python	Linguagem de programação	3.10.12
Pip	Gerenciador de pacotes Python	22.0.2

O Jenkins foi instalado em um servidor, junto com os *plug-ins* recomendados, e configurado para ser acessível via web por meio de um *proxy* reverso. Também foi feita a instalação padrão do Docker e Kubernetes nas máquinas de teste e produção.

Com o Jenkins acessível através do navegador, é possível realizar todo o trabalho de desenvolvimento a partir de um computador pessoal. Isso inclui acessar a interface do Jenkins para monitorar e gerenciar *builds*, executar testes e realizar as configurações de comportamentos e casos de uso, conforme descrito na Seção 3.2.1. A escrita do Jenkinsfile, que define o pipeline de CI/CD, será feita localmente, e o arquivo será enviado ao repositório remoto. No GitHub, foram configurados mecanismos de notificação automática, os chamados *webhooks*, que enviam alertas ao Jenkins sempre que eventos ocorrem, como *commits* ou *pull requests*.

Para acessar alguns serviços externos que requerem autorização, iremos utilizar as credenciais do Jenkins, configuráveis na sua tela de gerenciamento. Precisamos adicionar credenciais para acesso aos seguintes serviços: GitHub, Docker Hub e SonarQube. Além da credencial de acesso ao *cluster* K8s.

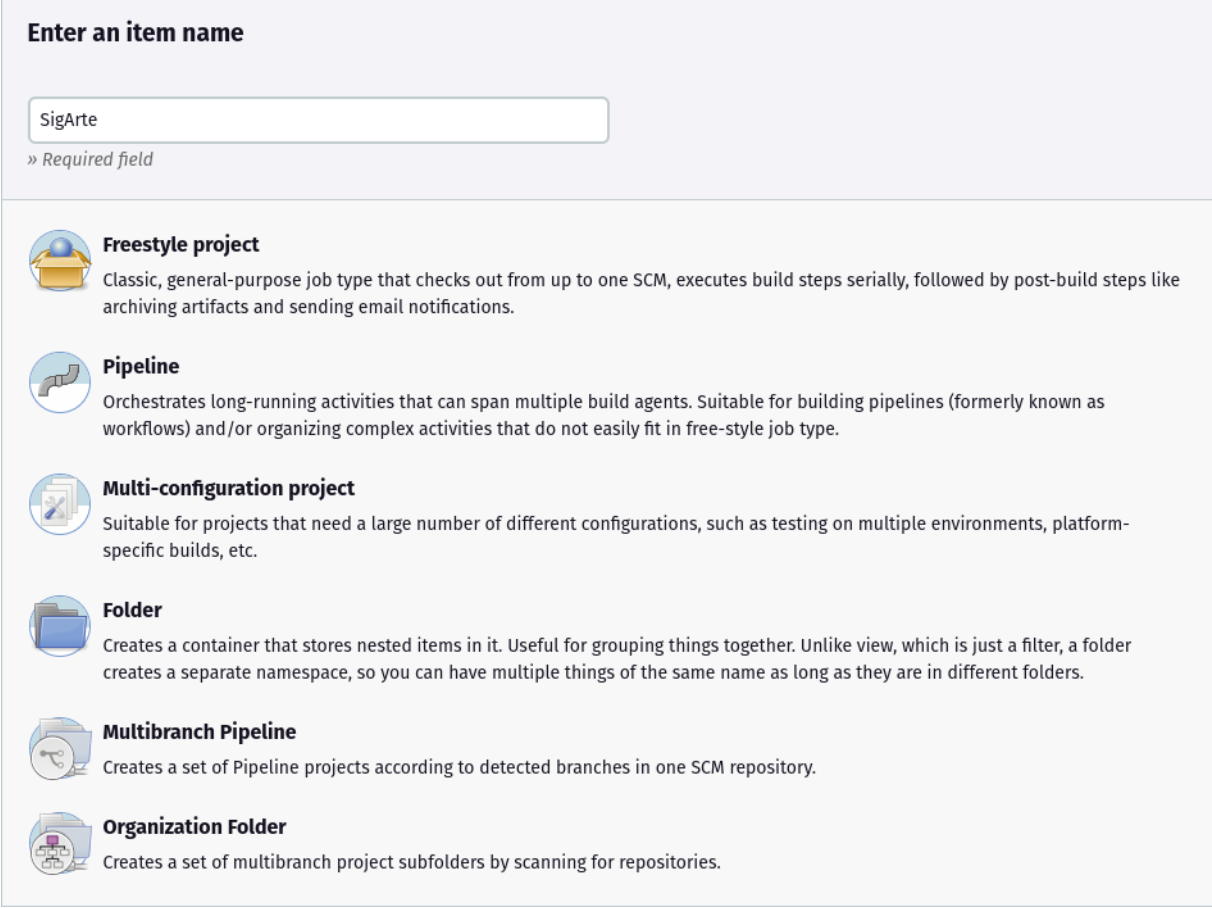
Para a implantação sem *downtime* com Kubernetes, optamos por utilizar o K3s para a criação do *cluster* local devido à sua simplicidade e leveza. Para interagir com o *cluster*, utilizamos o *kubectl*, a ferramenta de linha de comando padrão do Kubernetes, instalada juntamente com o K3s.

Instalamos todas as ferramentas necessárias seguindo sua documentação básica de instalação. Demais utilitários que sejam necessários podem ser usados em forma de *plug-ins* Jenkins, ou pacotes Python, instalados via gerenciador de pacotes *pip*, durante a execução do *pipeline*.

4.3 Configuração de Comportamentos do Pipeline

O Jenkins nos dá a possibilidade de criar automações para vários tipos de projeto, nesse caso, iremos utilizar o *Pipeline Multibranch*, que irá tratar de um projeto com várias ramificações de desenvolvimento. Neste caso, é onde iremos tratar quais comportamentos esperados para cada *branch* ou PRs.

Figura 11 – Opções de *Pipeline* Jenkins



Fonte: Autor

Após selecionar e criar o tipo de projeto desejado, Figura 11, podemos configurar o comportamento geral desse projeto, como: o repositório do projeto a ser trabalhado, estratégias de descobrimento de *branches* e PRs, velocidade e durabilidade de um *build*, periodicidade de *build*, entre outras configurações a depender do tipo de projeto e *plug-ins* utilizados.

Após configurado o repositório a ser utilizado e suas credenciais de acesso, foi definido que somente PRs para as *branches* main e dev serão considerados para o *build*, e que a execução do *pipeline* iria ser feita considerando a *branch* de desenvolvimento que possui as mudanças que serão possivelmente mescladas a *branch* dev ou main, após o sucesso do *pipeline* e a confirmação do *pull request*. Por enquanto, deixamos configurado para ignorar qualquer *fork* do projeto, que não é aplicável para o contexto.

Para economizar recursos na máquina hospedeira do *pipeline*, selecionamos a opção de otimização de performance. Isso permite que o Jenkins escreva dados no disco com menos frequência durante a execução dos *pipelines*, em troca de manter por menos tempo informações sobre *pipelines* executados no passado. Essa configuração melhora significativamente o desempenho dos *pipelines*, especialmente aqueles com muitos passos ou lógica complexa, sem fazer a utilização agentes externos.

Além disso, configuramos o Jenkins para lidar com possíveis falhas de notificação de *push* do GitHub. Caso um *build* não seja acionado automaticamente por um evento do GitHub, o Jenkins garantirá a execução do pipeline a cada 12 horas, como uma medida de remediar, assegurando que nenhum PR fique sem ser *buildado* por muito tempo.

Sobre a estratégia de gerenciamento de itens órfãos no Jenkins, optou-se por descartar itens antigos para economizar recursos. Especificamente, itens antigos serão mantidos por até 5 dias e, no máximo, 20 itens antigos serão preservados, independentemente de sua idade. Esta abordagem ajuda a manter o sistema limpo e eficiente, removendo itens que não são mais necessários. Nos filtros vamos monitorar apenas as *branches* importantes (dev e main), como todos os PRs (PR-*) e *tags* nomeadas com *releases*.

Por padrão, o Jenkins irá procurar na raíz do projeto um arquivo de nome Jenkinsfile que irá ter todas as instruções da execução do *pipeline*, será nele que iremos trabalhar no desenvolvimento das etapas do CI/CD.

4.4 Desenvolvimento das Etapas do CI

Com o Jenkins já configurado para realizar os comportamentos esperados, partimos para o desenvolvimento dos estágios do *pipeline*. No diretório do código fonte, adicionamos o arquivo *Jenkinsfile* na raiz do projeto e desenvolvemos as etapas da CI. A seguir são detalhados todos os estágios do *pipeline* de forma separada, assim como seus códigos fonte. O arquivo final completo do CI pode ser visto no Apêndice E.

No início do arquivo, Código 4.1, foi aberto o escopo do *pipeline* que irá envolver todas as etapas e configurações. Logo depois, foi configurado que o *build* pode ser feito em qualquer Jenkins *agent* disponível. Além disso, as linhas 5 e 6 definem as variáveis de ambiente essenciais para a configuração do ambiente virtual Python. A linha 5 define o diretório onde o ambiente virtual Python será criado, logo após, o caminho do seu binário é adicionado ao *PATH* do ambiente Jenkins, para garantir uma execução isolada e consistente.

Código 4.1 – Definição do Agente e Variáveis de Ambiente

```
1 pipeline {
2     agent any
3
4     environment {
```

```
5         VIRTUAL_ENV = 'venv'
6         PATH = "${env.WORKSPACE}/${env.VIRTUAL_ENV}/bin:${env.PATH}"
7     }
```

Código 4.2 – Estágio de *Checkout*

```
1     stages {
2         stage('Checkout') {
3             steps {
4                 checkout scm
5             }
6         }
```

No Código 4.2 é onde os estágios começam, de fato, a serem definidos. O primeiro estágio como mostrado é o de *Checkout*, que irá baixar o código do GitHub definido nas configurações do *pipeline* Jenkins, seguindo as regras de comportamentos definidas na Seção 4.3. Todas os demais estágios serão definidos dentro do bloco *stages*.

Código 4.3 – Instalação de Dependências

```
1         stage('Install Dependencies') {
2             steps {
3                 script {
4                     sh 'python3 -m venv venv'
5                     sh "python -m pip install --upgrade pip"
6                     sh "pip install -r requirements.txt" // --upgrade --
    force-reinstall
7                     sh 'cp .env.example .env'
8                 }
9             }
10        }
```

No Código 4.3, é ilustrado o estágio de instalação de dependências. Primeiramente, um ambiente virtual Python é criado com o comando da linha 4. Em seguida, o gerenciador de pacotes pip é atualizado para a versão mais recente na linha 5. O próximo comando, sh "pip install -r requirements.txt", instala todas as dependências listadas no arquivo requirements.txt. Por fim, a linha 7 copia um arquivo de variáveis de ambiente que está na raíz do projeto para um novo arquivo .env, configurando assim variáveis de ambiente específicas que o aplicativo possa requerer.

Código 4.4 – Execução de Testes

```
1         stage('Tests') {
2             steps {
3                 script {
4                     sh 'python manage.py test'
5                 }
```

```
6             }
7         }
```

O trecho visto no Código 4.4 executa os testes de software existentes e definidos no arquivo de configurações *manage.py*. Se algum teste falhar, o *pipeline* irá falhar por completo, recusando a integração.

Código 4.5 – Análise de Cobertura de Testes

```
1 stage('Run Coverage') {
2             steps {
3                 script {
4                     sh 'python -m coverage run manage.py test'
5                     sh 'python -m coverage xml'
6                 }
7             }
8         }
```

No Código 4.5, é mostrado o estágio de Análise de Cobertura de Testes. Este estágio é fundamental para medir a extensão do código coberto pelos testes, ajudando a identificar áreas que não estão sendo testadas adequadamente. Um arquivo de relatório de cobertura é gerado e armazenado em um arquivo *.xml*, que será utilizado nos próximos estágios.

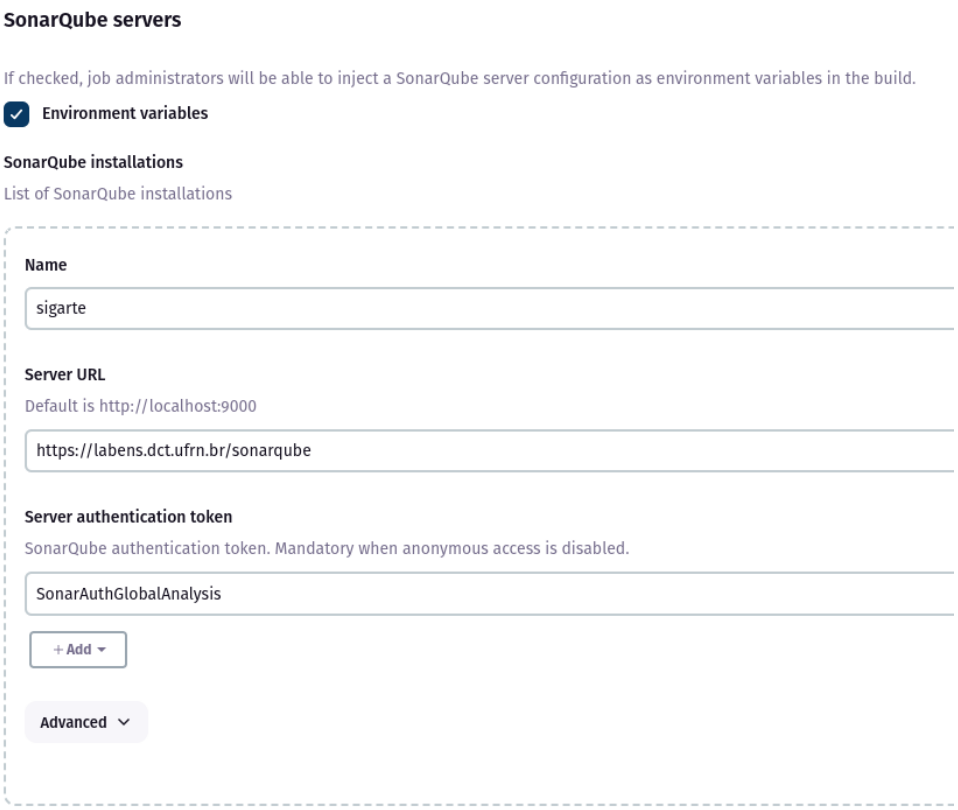
Código 4.6 – Análise de Código Estático

```
1         stage("SonarQube Build & Analysis") {
2             steps {
3                 withSonarQubeEnv('sigarte') {
4                     sh '''
5                     . venv/bin/activate
6                     pip install pysonar-scanner
7                     pysonar-scanner -Dsonar.token=${SONAR_AUTH_TOKEN} -
    Dsonar.host.url=${SONAR_HOST_URL}
8                     '''
9                 }
10            }
11        }
```

Para o estágio mostrado no Código 4.6 foi necessário realizar a instalação do *plug-in* SonarQube Scanner for Jenkins. Com o *plug-in* instalado, abrimos as configurações de sistema do Jenkins, e configuramos de acordo com a Figura 12. É necessário ter criado uma credencial com um *token* de autenticação gerado no SonarQube para poder selecionar no campo *Server authentication token*.

Assim sendo, estamos aptos a utilizar a função *withSonarQubeEnv* passando com parâmetro o nome utilizado na configuração da Figura 12. Isso nos deixará autenticado a qualquer envio de informação para a aplicação do Sonar, que está hospedado no servidor do

Figura 12 – Configuração do Sonar no Jenkins



Fonte: Autor

LABENS. Por fim, é instalado o pacote *pysonar-scanner* sendo disparada uma análise com o comando da linha 7 no Código 4.6, passando o *token* e *host* também especificados na Figura 12.

O SonarQube aceita um arquivo de propriedades que especifica alguns comportamentos a serem obedecidos na análise, o nosso pode ser visto no Apêndice D, Código D.1. Nele podemos notar que o arquivo de cobertura gerado (*coverage.xml*) é especificado na propriedade *sonar.python.coverage.reportPaths*.

Dessa forma, a análise será realizada pelo *pysonar-scanner* e os resultados serão enviados para o servidor Sonar, que verificará se o código atende aos critérios de qualidade mínimos definidos pelo *Quality Gate* do Sonar.

Código 4.7 – Resultado de Análise Estática

```
1         stage("Quality Gate") {
2             steps {
3                 timeout(time: 3, unit: 'MINUTES') {
4                     waitForQualityGate abortPipeline: true
5                 }
6             }
7         }
8     }
```

```
9 }
```

O último estágio no *pipeline* de CI, Seção 4.7 define um limite para o tempo de espera (3 minutos) até o recebimento da resposta do SonarQube em relação ao cumprimento dos requisitos de qualidade. Ao atingir o limite de tempo, o *pipeline* é abortado e dado como falho. Essa etapa depende também da configuração de um *webhook* no servidor do SonarQube, que servirá para enviar notificações sobre o resultado da análise.

O resultado sendo positivo, nosso pipeline de CI está completo. O desenvolvedor pode acompanhar a execução e os resultados do *build*, por meio de um *feedback* visual visto na tela principal do repositório GitHub ou na tela do PR.

4.5 Desenvolvimento da Estratégia de Implantação

A estratégia de implantação do sistema SigArte foi cuidadosamente desenvolvida para garantir alta disponibilidade, facilidade de manutenção e escalabilidade. Abaixo, está detalhada as principais etapas da criação e implementação dessa estratégia.

Para a criação e gerenciamento de um *cluster* local, foi utilizado a ferramenta K3s, uma versão leve e simplificada do K8s, garantindo eficiência e menor consumo de recursos, o que é ideal para ambientes de desenvolvimento e produção de pequeno porte (K3s, 2024). E a ferramenta de linha de comando *kubectl* para interagir com o *cluster*.

O K3s foi instalado no servidor Linux, após isso, o desenvolvimento da estratégia de implantação será realizada em arquivos *.yaml*, que irão substituir o *Docker Compose*. Estes arquivos, também chamados de manifestos, descrevem os recursos e objetos que você deseja criar e gerenciar em um *cluster* Kubernetes. Eles funcionam como uma forma declarativa de definir o estado desejado para os recursos, como *pods*, serviços, *deployments*, entre outros.

Para implementar nossa estratégia, utilizamos três recursos fundamentais do K8s: *Deployment*, *Service* e *ConfigMap*, já detalhados na Seção 2.8.4. Esses recursos foram configurados em dois arquivos de manifesto, que serão detalhados nas subseções seguintes. O arquivo *deployment.yaml* define tanto o recurso *Deployment* quanto o recurso *Service*, essenciais para gerenciar a aplicação e sua exposição na rede. O segundo arquivo, *configMap.yaml*, especifica as configurações necessárias para a correta execução da aplicação, armazenando variáveis de ambiente e outros parâmetros importantes.

4.5.1 Manifesto de *Deployment*

O arquivo *deployment.yaml* é um manifesto do K8s que define como a aplicação será implantada e gerenciada no *cluster*. Ele é responsável por especificar a quantidade de réplicas da

aplicação, a estratégia de atualização, os contêineres que compõem a aplicação e suas respectivas configurações.

A seguir, será explicado cada ponto do arquivo de manifesto *deployment.yaml* que pode ser visto integralmente no Apêndice G.

- **apiVersion**: Define a versão da API Kubernetes utilizada. No caso, `apps/v1` é a versão que engloba recursos como `Deployment`.
- **kind**: Especifica o tipo de recurso que está sendo criado, neste caso, um `Deployment`.
- **metadata**: Contém informações básicas, como o nome do `Deployment`, `sigarte`.
- **spec**: Detalha as especificações do `Deployment`.
  - **replicas**: Define o número de instâncias (pods) da aplicação que devem ser mantidas em execução, 2 neste caso.
  - **strategy**: Define a estratégia de atualização dos pods. Aqui, foi utilizada a estratégia de `RollingUpdate`, que substitui gradualmente as instâncias antigas por novas.
  - **selector**: Define como o Kubernetes identificará os pods pertencentes a este `Deployment`, utilizando rótulos (*labels*).
  - **template**: Contém a definição dos pods que serão gerados.
    - **metadata**: Define os rótulos que serão atribuídos aos pods.
    - **spec**: Especifica o conteúdo dos pods.
      - **containers**: Define os containers que serão executados dentro de cada pod.
      - **name**: Nome do container, `sigarte`.
      - **image**: Imagem Docker que será utilizada para criar o container, alterada via Jenkinsfile de acordo com a *tag*.
      - **envFrom**: Referencia o `ConfigMap` que contém variáveis de ambiente para o container.
      - **command** e **args**: Define o comando que será executado no container.
      - **ports**: Especifica as portas que serão expostas pelo container.
      - **readinessProbe**: Configura uma verificação de prontidão, que determina se o container está pronto para receber tráfego e alguns limites de *timeout*.

Para a estratégia de implantação (*Rolling Update*) ocorrer sem *downtime*, a opção *strategy* define que até dois novos *pods* podem ser criados antes que os antigos sejam removidos (maxSurge: 2), Código 4.8, garantindo que a aplicação continue disponível durante todo o processo de atualização. A opção maxUnavailable: 0 assegura que nenhum *pod* existente seja removido antes que os novos *pods* estejam prontos, mantendo assim o serviço 100% operacional.

Código 4.8 – Estratégia de atualização no deployment.yaml

```
1  strategy:
2    type: RollingUpdate
3    rollingUpdate:
4      maxUnavailable: 0
5      maxSurge: 2
```

Para garantir que os novos *pods* estão prontos antes de começar a redirecionar o tráfego para eles, uma *readinessProbe* foi configurada, Código 4.9. Essa *probe* verifica a disponibilidade da aplicação através de uma requisição HTTP para o caminho /sigarte/portal/ na porta 8002. A configuração define um atraso inicial de 50 segundos (initialDelaySeconds: 50) e realiza verificações a cada 60 segundos (periodSeconds: 60). Se a probe falhar três vezes consecutivas (failureThreshold: 3), o *pod* é considerado não pronto, impedindo que ele receba tráfego até que esteja completamente funcional. Essas configurações garantem uma atualização suave e sem interrupções no serviço.

Código 4.9 – Verificação de prontidão da aplicação

```
1  readinessProbe:
2    initialDelaySeconds: 50
3    periodSeconds: 60
4    failureThreshold: 3
5    httpGet:
6      path: /sigarte/portal/
7      port: 8002
8      httpHeaders:
9        - name: host
10         value: health-check-probe
```

4.5.2 Manifesto de Serviço

O arquivo *deployment.yaml*, disponível no Apêndice G, também define um *Service*, recurso K8s que é responsável por expor a aplicação para o tráfego interno do *cluster*.

- **apiVersion**: Especifica a versão da API, neste caso, `v1`.
- **kind**: O tipo de objeto, neste caso, um `Service`.
- **metadata**: Contém o nome do serviço, `sigarte-svc`.
- **spec**: Define as especificações do serviço:
  - **selector**: Utiliza os rótulos para associar o serviço aos pods correspondentes.
  - **ports**: Mapeia a porta interna do cluster para a porta do container.

  - **type**: Define o tipo de serviço, `ClusterIP`, que torna a aplicação acessível apenas dentro do cluster.

4.5.3 Manifesto de Configuração

O arquivo chamado *configMap.yaml* nada mais é do que um objeto do K8s que permite armazenar variáveis de ambiente em pares chave-valor, acessíveis pelos *pods*. É geralmente utilizado para separar os dados de configuração do código da aplicação, facilitando a gestão e modificação das configurações sem precisar reconstruir as imagens dos contêineres.

O arquivo de exemplo pode ser visto no Apêndice H, cada aplicação deverá preencher o arquivo de acordo com suas configurações de ambiente.

- **apiVersion**: Define a versão da API Kubernetes, `v1`.
- **kind**: O tipo de objeto, `ConfigMap`.
- **metadata**: Contém o nome do `ConfigMap`, `sigarte-env`.
- **data**: Contém as variáveis de ambiente e seus valores, que são usados pelo container da aplicação:
  - **SECRET_KEY**: (Exemplo) Chave secreta utilizada pelo Django.
  - **DATABASE_PASSWORD**, **EMAIL_HOST_PASS**: (Exemplo) Senhas utilizadas para acessar o banco de dados e o serviço de email.

4.5.4 Aplicação dos Manifestos

Utilizando o comando ´kubectl apply -f <file>', podemos aplicar os manifestos ao nosso cluster K8s. Ao aplicar o arquivo *deployment.yaml* desenvolvido, o Kubernetes iniciará a criação de um conjunto de Pods que executam a aplicação Django conforme especificado no manifesto. Também gerencia o ciclo de vida dos Pods, garantindo que sempre existam duas réplicas em execução. A estratégia de atualização *Rolling Update* garantirá que novas versões dos Pods sejam gradualmente introduzidas sem interrupção do serviço.

Além dos Pods, o arquivo também define um *Service* do tipo *ClusterIP*, que cria um ponto de acesso estável e interno para a aplicação dentro do cluster Kubernetes. Esse *Service* permite que outras partes da aplicação ou outros serviços no cluster se conectem à aplicação Django por meio de um endereço IP fixo, gerenciado pelo Kubernetes. Será esse endereço IP que será exposto externamente pelo servidor web.

Como resultado da aplicação desse manifesto, a aplicação será disponibilizada em um ambiente escalável e gerenciado, com balanceamento de carga interno e alta disponibilidade, sem *downtime* durante as atualizações.

4.6 Desenvolvimento das Etapas do CD

Após o desenvolvimento das etapas do CI e da estratégia de implantação a ser utilizada, partimos para o desenvolvimento das etapas do *pipeline* de CD, que será responsável pela construção, publicação e implantação de imagens Docker. A seguir, explicamos cada estágio do pipeline CD. As etapas de CD do Jenkinsfile poderão ser vistas no Apêndice F.

Código 4.10 – Estágio de Construção da Imagem Docker

```
stage("Build Docker Image") {
    when {
        anyOf {
            branch 'dev'
            expression { return env.TAG_NAME != null }
        }
    }
    steps {
        script{
            dockerapp = docker.build("labens/sigarte")
        }
    }
}
```

No primeiro estágio da CD, visto no Código 4.10, a imagem Docker é construída usando o *Dockerfile* presente no repositório. A condição '*when*' garante que este estágio será executado apenas se a *branch* for '*dev*' ou se uma *tag* estiver presente ('env.TAG_AME != null'). O comando 'docker.build' é utilizado para criar a imagem Docker com o nome 'labens/sigarte'.

Código 4.11 – Estágio de Publicação da Imagem Docker

```
stage("Push Docker Image") {
    when {
        anyOf {
            branch 'dev'
            expression { return env.TAG_NAME != null }
        }
    }
    steps {
        script {
            docker.withRegistry('https://registry.hub.docker.com', 'docker-
    hub-credential-id') {
                if (env.TAG_NAME != null) {
                    docker.image("labens/sigarte:${env.TAG_NAME}").push()
                    docker.image("labens/sigarte:latest").push()
                    env.DOCKER_IMAGE_TAG = env.TAG_NAME
                } else if (env.GIT_BRANCH == 'dev') {
                    docker.image("labens/sigarte:${env.BRANCH_NAME}-${env.
    BUILD_ID}").push()
```

```
17                      env.DOCKER_IMAGE_TAG = "${env.BRANCH_NAME}-${env.
    BUILD_ID}"
18                  }
19              }
20          }
21      }
22  }
```

No segundo estágio, Código 4.11, a imagem Docker construída é publicada no registro Docker. A condição 'when' determina a mesma regra usada na primeira etapa do CD. O bloco 'docker.withRegistry' autentica no registro Docker por meio de uma credencial adicionada ao Jenkins. Dependendo de se há uma *tag* ('env.TAG_NAME != null'), a imagem é publicada com a *tag* correspondente e também como 'latest'. Se não houver uma *tag*, mas a *branch* for '*dev*', a imagem é publicada com uma *tag* gerada a partir do nome da *branch* e o *ID* do *build*.

Código 4.12 – Estágio de Implantação no Kubernetes

```
1  stage("Deploy") {
2      when {
3          anyOf {
4              branch 'dev'
5              expression { return env.TAG_NAME != null }
6          }
7      }
8      steps {
9          script {
10             withKubeConfig([credentialsId: 'kubeconfig']) {
11                 if (env.TAG_NAME != null) {
12                     sh 'sed -i "s/{{tag}}/${env.TAG_NAME}/g" ./k8s/
    deployment.yaml'
13                 } else if (env.GIT_BRANCH == 'dev') {
14                     sh 'sed -i "s/{{tag}}/${env.BRANCH_NAME}-${env.BUILD_ID
    }/g" ./k8s/deployment.yaml'
15                 }
16                 sh 'kubectl apply -f k8s/deployment.yaml'
17                 sh 'kubectl apply -f k8s/configMap.yaml'
18             }
19         }
20     }
21 }
```

No último estágio da CD, visto no Código 4.12 é realizada a implantação via K8s, o arquivo de configuração ('deployment.yaml') é atualizado com a *tag* da imagem Docker e aplicado no *cluster* Kubernetes. A condição 'when' segue a mesma regra dos outros estágios. O bloco 'withKubeConfig' configura o acesso ao *cluster* K8s usando credenciais específicas, neste caso, um arquivo secreto adicionado ao Jenkins, referente ao arquivo 'KUBECONFIG' do K3s.

O comando ‘sed’ substitui a *placeholder* ‘tag’ no arquivo ‘deployment.yaml’ pela tag apropriada. Após a substituição, o comando ‘kubectl apply’ aplica as alterações no *cluster* Kubernetes.

Esses estágios garantem que a imagem Docker seja construída, publicada e implantada de forma eficiente e condicional, dependendo do contexto do *build* (seja uma *branch* ou uma *tag*). O processo aplica os manifestos descritos na Seção 4.5 ao *cluster* Kubernetes, assegurando que tanto as configurações quanto as definições de *deployment* sejam atualizadas conforme necessário.

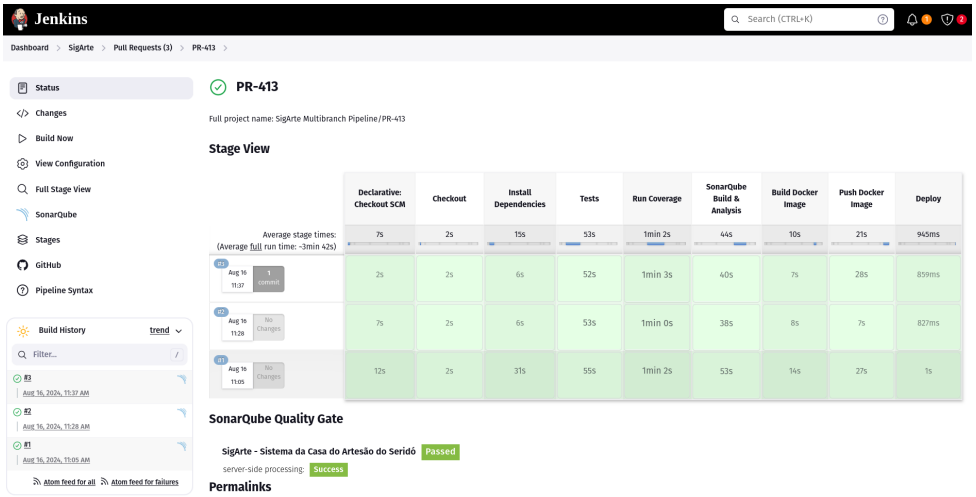
5 Resultados e Discussões

Neste capítulo, são apresentadas as discussões sobre os resultados obtidos com a implementação do pipeline CI/CD utilizando Jenkins e Kubernetes no sistema *SigArte*, conforme descrito ao longo deste trabalho. A análise dos resultados é organizada de maneira a responder às questões de pesquisa e alcançar os objetivos estabelecidos, destacando as contribuições práticas e teóricas do trabalho.

5.1 Implementação do Pipeline CI/CD

A implementação do pipeline CI/CD foi iniciada com a configuração do Jenkins e a integração contínua (CI), seguido pela automação do processo de *build* e *deploy*, e culminando na implantação sem *downtime* utilizando Kubernetes. A seguir, algumas imagens do resultado serão explicadas, além de subseções que detalham cada etapa.

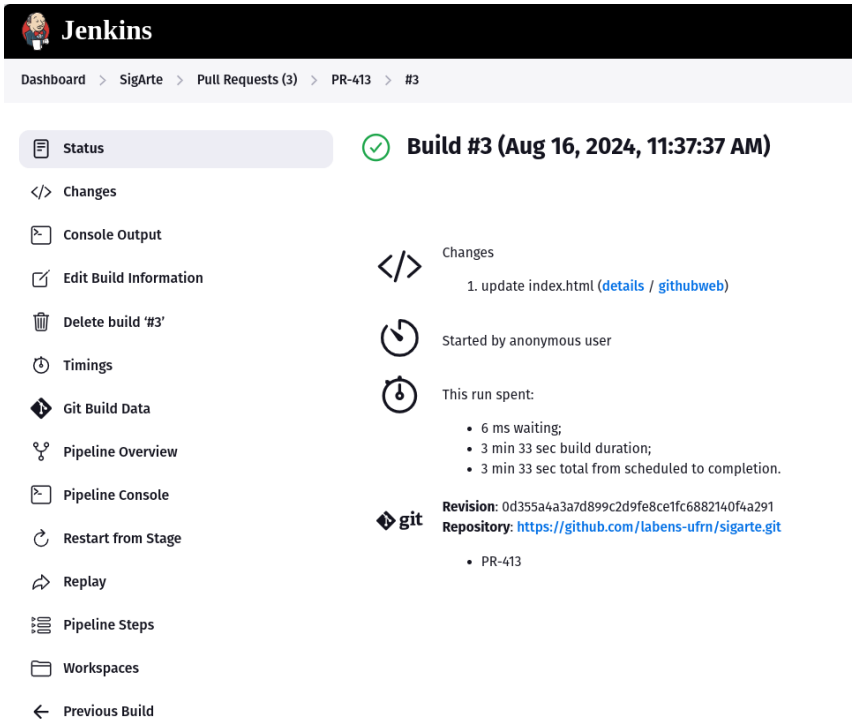
Figura 13 – Visualizações dos estágios do *pipeline* CI/CD no Jenkins



Fonte: Autor

A Figura 13 apresenta uma captura de tela da interface do Jenkins, ilustrando os estágios dos *builds*. Na imagem, são mostrados três *builds* que foram concluídos com sucesso, detalhando o tempo de execução de cada etapa. Neste cenário, o Jenkins também finalizou o pipeline realizando a implantação por meio do *deployment* K8s. Ao clicar em um *build*, temos acesso a mais detalhes em específico, como: arquivos que sofreram alterações, quem deu início ao *build*, saídas do terminal durante a execução, visão geral do tempo de execução e estágios, mostrado na Figura 14.

Figura 14 – Visualização da tela de um *build* no Jenkins



Fonte: Autor

5.1.1 Configuração do Jenkins e Integração Contínua

Partindo da configuração inicial apresentada na Seção 4.3, foi possível automatizar o processo de *build* para cada PR aberto no repositório e para as *branches* de interesse, estabelecendo eventos precisos para disparar a execução do pipeline. Além disso, foi configurado o Jenkins para realizar varreduras periódicas no repositório, garantindo a detecção de novas alterações que eventualmente não fossem notificadas pelo *webhook* do GitHub. Foram também estabelecidas regras para a exclusão automática de pipelines antigos no servidor Jenkins, otimizando o uso de recursos.

Adicionalmente, alguns *plug-ins* foram instalados para ampliar a funcionalidade e a flexibilidade do Jenkins. No contexto da CI, conforme detalhado na Seção 4.4, foram implementadas etapas que incluem a instalação de dependências, execução de testes unitários, verificação da cobertura de código e análise estática da qualidade do código. Essas etapas são pré-requisitos essenciais para a execução subsequente do pipeline de CD, garantindo uma base sólida para o desenvolvimento contínuo e a entrega de software de alta qualidade.

5.1.2 Automação do Build e Implantação com Jenkinsfile

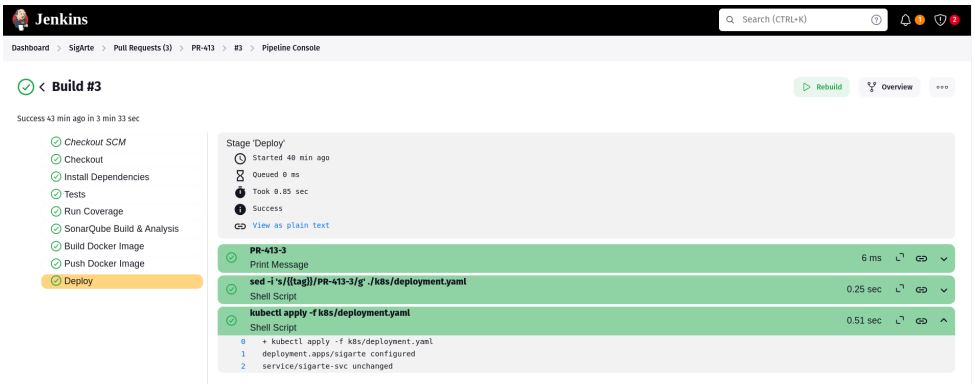
Aqui, destacamos a importância do *Jenkinsfile* na automatização do processo de *build* e *deploy*, como mostrado na Seção 4.6. O *Jenkinsfile* criado permitiu a construção da imagem

Docker da aplicação a partir do *Dockerfile* e seu envio para um repositório remoto de forma automática utilizando credenciais e o *plug-in* Docker Pipeline. Além disso, a lógica condicional implementada, que diferencia *builds* para *branches* de desenvolvimento e *releases* com *tags*, resultou em um fluxo de trabalho mais eficiente e adaptável às necessidades do projeto.

O Jenkinsfile é responsável por declarar todas as etapas do *pipeline*, bem como o comando de implantação final que utiliza o controlador Kubernetes para entrega e atualização da aplicação.

A visualização de cada etapa do *pipeline* no *console* pode ser vista na Figura 15. Onde é possível analisar separadamente o fluxo de instruções, definidas no *Jenkinsfile*, e as saídas produzidas durante a execução dos estágios.

Figura 15 – Visualização da tela de *Pipeline Console* de um *build* no Jenkins



Fonte: Autor

5.2 Implantação com Kubernetes Sem Downtime

Nesta seção, analisamos os resultados da implantação da aplicação utilizando Kubernetes, explicado na Seção 4.5, com ênfase na simplificação proporcionada pelo uso do K3s para a criação de um *cluster* local. A estratégia de *deploy* sem *downtime* foi implementada de forma eficaz, utilizando técnicas como *Rolling Updates* e *Readiness Probes* para garantir que a aplicação estivesse totalmente operacional antes de receber tráfego. A configuração do serviço *ClusterIP* assegurou a alta disponibilidade da aplicação durante o processo de atualização.

Adicionalmente, a aplicação dos manifestos Kubernetes permitiu uma configuração e gestão eficiente dos recursos do *cluster*. Isso incluiu o uso de *ConfigMaps* para a parametrização dinâmica do ambiente, facilitando ajustes de configuração sem a necessidade de reconstruir a imagem Docker ou interromper o funcionamento da aplicação. Os manifestos são aplicados pelo Jenkins durante a última etapa da CD, utilizando o *plugin* Kubernetes CLI. Essas práticas garantiram uma implantação robusta e adaptável, alinhada aos requisitos de alta disponibilidade e flexibilidade, aposentando o antigo arquivo *docker-compose.yml*.

Para ilustrar o processo de atualização de uma versão do aplicativo usando o *Rolling Update* no Kubernetes, considere a seguinte sequência de comandos e saídas da Figura 16:

Para monitorar o progresso da atualização, utilizamos o comando 'kubectl get pods –watch' logo após a execução do último estágio do CD, que aplica as mudanças no arquivo de configuração 'deployment.yaml'. Este arquivo pode incluir atualizações na imagem do contêiner ou outras configurações do *Deployment*, e a atualização é realizada com o comando 'kubectl apply'. A saída do comando 'kubectl get pods –watch' revela a transição dos *pods* durante o *Rolling Update*. A Figura 16 ilustra essa saída, e a seguir, fornecemos uma explicação detalhada sobre cada linha dessa saída para esclarecer o processo de atualização.

Figura 16 – Saída do terminal ao Aplicar o Deployment e Monitorar o Progresso com 'kubectl get pods –watch'

Fonte: Autor

Para facilitar a compreensão e a explicação, as informações apresentadas na Figura 16 foram organizadas e enumeradas linha por linha a seguir.

```
    $ kubectl get pods --watch
0   NAME                          READY   STATUS        AGE
1   sigarte-5c849fdfdc-gls4b      1/1     Running       14m
2   sigarte-5c849fdfdc-vqvph      1/1     Running       14m
3   sigarte-78d6ff884-c8jn5       0/1     Running       8s
4   sigarte-78d6ff884-nz8t6       0/1     Running       8s
5   sigarte-78d6ff884-c8jn5       1/1     Running       60s
6   sigarte-5c849fdfdc-vqvph      1/1     Terminating   15m
7   sigarte-78d6ff884-nz8t6       1/1     Running       60s
8   sigarte-5c849fdfdc-gls4b      1/1     Terminating   15m
9   sigarte-5c849fdfdc-vqvph      0/1     Terminating   16m
10  sigarte-5c849fdfdc-gls4b      0/1     Terminating   16m
```

- **Linhas 1 e 2**: Esses são os pods da versão antiga (‘sigarte-5c849fdfdc-gls4b‘ e ‘sigarte-5c849fdfdc-vqvph‘) que já estavam em execução antes da atualização. Ambos estão com status ‘Running‘, indicando que estão operacionais e prontos para receber tráfego.
- **Linhas 3 e 4**: Esses são os novos pods (‘sigarte-78d6ff884-c8jn5‘ e ‘sigarte-78d6ff884-nz8t6‘) que foram iniciados como parte do novo ReplicaSet. Inicialmente, eles não estão prontos (‘0/1 Ready‘), mas estão em processo de inicialização.
- **Linha 5**: Um dos novos pods (‘sigarte-78d6ff884-c8jn5‘) fica pronto (‘1/1 Ready‘), indicando que está completamente operacional. Nesse ponto, o Kubernetes pode começar a finalizar a versão antiga.
- **Linha 6**: O pod da versão antiga (‘sigarte-5c849fdfdc-vqvph‘) está sendo finalizado (‘Terminating‘), pois a nova versão já está começando a assumir a carga.
- **Linha 7**: O segundo novo pod (‘sigarte-78d6ff884-nz8t6‘) também se torna pronto (‘1/1 Ready‘). Com ambos os novos pods prontos, o Kubernetes continua o processo de desativação dos pods antigos.
- **Linhas 8 e 9**: Os pods antigos (‘sigarte-5c849fdfdc-gls4b‘ e ‘sigarte-5c849fdfdc-vqvph‘) estão agora em processo de término (‘Terminating‘). Eles não estão mais prontos (‘0/1 Ready‘), e o Kubernetes está removendo gradualmente esses pods à medida que os novos pods assumem completamente a carga de trabalho.

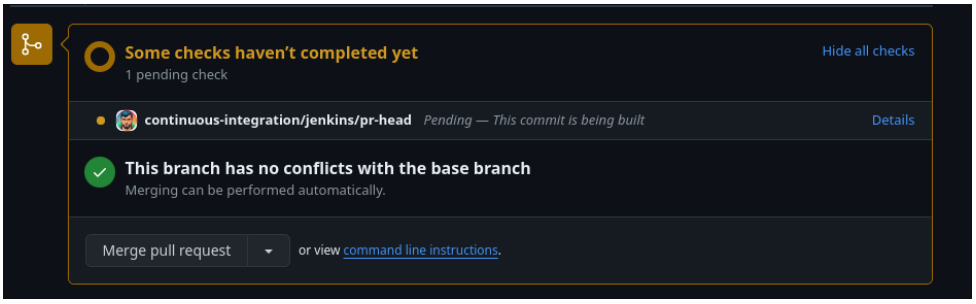
Esse processo assegura que sempre haja *pods* disponíveis para servir o tráfego, minimizando o downtime durante a atualização. O *Rolling Update* permite a substituição gradual dos *pods* antigos pelos novos, mantendo a alta disponibilidade do serviço durante todo o processo.

5.3 *Feedback* ao Desenvolvedor

A Figura 17 ilustra o status do *build* de um PR aberto no GitHub. Promovendo um *feedback* visual rápido para o desenvolvedor.

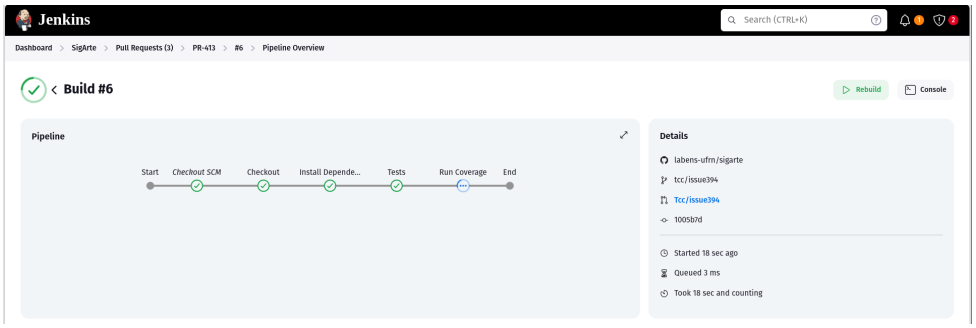
Ao clicar em ‘details’ o desenvolvedor pode acessar informações detalhadas sobre a execução das etapas no Jenkins, como mostrado na Figura 18. Esse retorno imediato permite ao desenvolvedor avaliar rapidamente a qualidade do código submetido e verificar a integridade das alterações. É possível consultar os *logs* da execução para identificar falhas nos testes, por exemplo, antes de realizar a integração com a *branch* alvo.

Figura 17 – Visualização do status do *build* em tela de PR do GitHub



Fonte: Autor

Figura 18 – Visualização do *build* no Jenkins acessada pelo GitHub



Fonte: Autor

5.4 Documentação e Reusabilidade

Destacamos aqui também a importância da documentação gerada durante o processo. O *Jenkinsfile* e os manifestos Kubernetes desenvolvidos, juntamente com as explicações detalhadas no Capítulo 4, não só irão facilitar a manutenção e desenvolvimento contínuo do projeto atual, como também servirão como base para pipelines futuros, promovendo a reusabilidade, padronização e a disseminação do conhecimento e das práticas de CI/CD dentro deste e de futuros projetos que sejam construídos pela equipe de desenvolvimento.

5.5 Discussão dos Resultados

Os resultados obtidos demonstram que o pipeline CI/CD desenvolvido cumpriu plenamente os objetivos traçados e respondeu eficazmente às questões de pesquisa propostas.

Ao longo do desenvolvimento, foi possível verificar que as etapas cruciais para a implementação de um pipeline CI/CD para uma aplicação web Django, como o SigArte, foram seguidas com rigor, garantindo a automação eficiente das tarefas de compilação, testes e verificação de qualidade do código-fonte.

A implementação do *pipeline* de CI atingiu seu propósito de automatizar a integração e validação de novas contribuições ao código do SigArte. Isso impactou diretamente na agilidade

e segurança do desenvolvimento, ao reduzir o tempo necessário para identificar e corrigir problemas, promovendo uma entrega contínua e de alta qualidade.

A adoção das práticas de CI/CD também elevou a disponibilidade do sistema ao implementar um pipeline de CD que, através de estratégias avançadas de implantação no Kubernetes, como o *rolling update*, eliminou o tempo de inatividade durante as atualizações. Esta estratégia foi fundamental para assegurar a continuidade do serviço e a experiência ininterrupta dos usuários.

A configuração do Kubernetes foi determinante para a realização de atualizações sem tempo de inatividade, respondendo à questão de como adotar essa tecnologia em um sistema monolítico. O sucesso dessa implantação demonstra a viabilidade e a eficácia da abordagem em sistemas semelhantes, oferecendo uma solução escalável e robusta, que não necessita de uma infraestrutura complexa e custosa.

Além disso, a documentação detalhada de todas as etapas do desenvolvimento do pipeline, incluindo estratégias de implantação, resultados dos testes e melhorias alcançadas, serviu como uma referência valiosa. Esse registro contribuiu para a transparência e reprodutibilidade do projeto.

Em resumo, o trabalho não só atingiu os objetivos propostos, como também ofereceu respostas claras e práticas às questões de pesquisa, evidenciando a importância e os benefícios de um pipeline CI/CD bem estruturado para a melhoria contínua dos processos de engenharia de software.

6 Conclusão

Neste trabalho, abordamos a implementação e aprimoramento de um pipeline de CI/CD para a entrega e disponibilidade de sistemas de informação, com foco nas práticas e estratégias modernas de *DevOps*. O pipeline desenvolvido integrou ferramentas e técnicas avançadas para otimizar o ciclo de vida do software.

A configuração do Jenkins e o uso do Kubernetes, junto com a estratégia de implantação adotada, garantiram um fluxo de trabalho eficiente e estável. Essas soluções permitiram a entrega contínua de novas versões do software com interrupções mínimas, proporcionando uma experiência de usuário confiável.

Este trabalho contribui significativamente para melhorar o ciclo de desenvolvimento e entrega de software, utilizando práticas e ferramentas avançadas. A automação e eficiência adquiridas são essenciais para responder rapidamente às mudanças do software e manter a qualidade.

No entanto, para garantir que o sistema continue eficaz, é necessário continuar aprimorando o pipeline de CI/CD. Isso inclui otimizar a performance, monitorar o sistema, escolher melhor os testes e realizar testes mais rigorosos. Essas melhorias são importantes para adaptar o sistema às novas necessidades e desafios.

6.1 Limitações

Embora a solução de CI/CD desenvolvida neste trabalho ofereça melhorias significativas para o ciclo de desenvolvimento, algumas limitações foram identificadas ao longo do processo. Uma dessas limitações está relacionada às restrições impostas pelos planos gratuitos de ferramentas como GitHub e SonarQube. Embora eficientes, esses planos limitam certas funcionalidades avançadas, como opções de segurança mais robustas para PRs e *merges*, além de recursos aprimorados de análise de código, que são oferecidos apenas em planos pagos.

Além das restrições de software, as limitações de hardware também impactaram o desempenho do pipeline, especialmente no que se refere ao tempo de execução das etapas. O servidor que hospeda o Jenkins e o hardware utilizado para o *cluster* Kubernetes local apresentaram limitações de recursos que influenciaram a velocidade e a eficiência dos processos de *build* e *deploy*.

Por fim, é importante ressaltar que a qualidade do código depende não apenas das ferramentas de automação, mas também das práticas de desenvolvimento adotadas pela equipe. A aplicação correta de princípios de engenharia de software, juntamente com o uso de ferramentas

adequadas, é essencial para garantir a eficácia da solução.

Apesar dessas limitações, a abordagem implementada provou ser benéfica, contribuindo para a automação e melhoria contínua do ciclo de desenvolvimento, estabelecendo uma base sólida para futuras expansões e aprimoramentos.

6.2 Trabalhos Futuros

Apesar dos avanços significativos com o pipeline de CI/CD implementado, há várias oportunidades para melhorias futuras. Focando em performance, monitoramento, testes e avaliação de desempenho, podemos otimizar ainda mais a solução.

Performance: Melhorar o tempo de execução do pipeline pode ser alcançado através da otimização do ambiente e da paralelização de processos. Considerar a atualização de hardware e a adição de nodes ao cluster Kubernetes, além do uso de ferramentas de cache em *builds*, pode trazer ganhos significativos.

Monitoramento e Observabilidade: Implementar um sistema de monitoramento e observabilidade, permitirá a coleta de métricas detalhadas em tempo real. Isso facilitará a detecção precoce de problemas e a análise do desempenho, aumentando a estabilidade e a capacidade de resposta a falhas.

Escolha de Testes: Refinar a estratégia de testes, incluindo uma combinação adequada de testes unitários, de integração, entre outros, pode melhorar a eficiência e a cobertura da validação, refletindo melhor o comportamento da aplicação em produção.

Testes de Carga e Stress: Incorporar testes de carga e stress ajudará a identificar gargalos e pontos de falha antes que eles afetem o ambiente de produção.

Explorar essas áreas contribuirá para a evolução contínua do pipeline de CI/CD, e para o ciclo geral de desenvolvimento e entrega do software.

Referências

ATLASSIAN. Continuous integration. *Atlassian*, 2023. Acessado em 29 de outubro de 2023. Disponível em: <https://www.atlassian.com/br/continuous-delivery/continuous-integration>. Citado na página 21.

ATLASSIAN. Getting started with continuous integration. *Atlassian*, 2023. Acessado em 29 de outubro de 2023. Disponível em: <https://www.atlassian.com/br/agile/software-development/continuous-integration>. Citado na página 21.

CARDELLINI, V.; COLAJANNI, M.; YU, P. Dynamic load balancing on web-server systems. *IEEE Internet Computing*, v. 3, n. 3, p. 28–39, 1999. Citado na página 25.

COHN, M. *Succeeding with Agile: Software Development Using Scrum*. 1st. ed. [S.l.]: Addison-Wesley Professional, 2009. ISBN 0321579364. Citado na página 19.

DJANGO. *Django | The web framework for perfectionists with deadlines*. 2023. Acessado em: 08 de Novembro, 2023. Disponível em: <https://www.djangoproject.com/>. Citado na página 37.

DOCKER. *Docker: Accelerated Container Application Development*. 2023. Acessado em: 09 de Novembro, 2023. Disponível em: <https://www.docker.com/>. Citado na página 26.

DUVALL, P.; MATYAS, S.; GLOVER, A. *Continuous Integration: Improving Software Quality and Reducing Risk*. 1. ed. Upper Saddle River, NJ: Pearson Education, 2007. ISBN 9780321336385. Citado na página 14.

FOUNDATION, T. L. *Kubernetes: Production-Grade Container Orchestration*. 2023. Acessado em: 10 de novembro de 2023. Disponível em: <https://kubernetes.io/>. Citado na página 28.

FOWLER, M. Continuous integration. *martinfowler.com*, 5 2006. Acessado em 28 de outubro de 2023. Disponível em: <https://martinfowler.com/articles/continuousIntegration.html>. Citado na página 21.

HAT, R. *O que é CI/CD?* 2021. Acessado em 31 de outubro de 2023. Disponível em: <https://www.redhat.com/pt-br/topics/devops/what-is-ci-cd>. Citado na página 24.

HAT, R. *What is Blue-Green Deployment*. 2024. Acessado em: 1 de Julho, 2023. Disponível em: <https://www.redhat.com/en/topics/devops/what-is-blue-green-deployment>. Citado na página 25.

HOODA, I.; CHHILLAR, R. S. Software test process, testing types and techniques. *International Journal of Computer Applications*, Citeseer, v. 111, n. 13, 2015. Citado na página 20.

HUMBLE, J.; FARLEY, D. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Pearson Education, 2010. (Addison-Wesley Signature Series (Fowler)). ISBN 9780321670229. Disponível em: <https://books.google.com.br/books?id=6ADDuzere-YC>. Citado na página 24.

ITKONEN, J.; UDD, R.; LASSENIUS, C.; LEHTONEN, T. Perceived benefits of adopting continuous delivery practices. *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, 2016. Citado 2 vezes nas páginas 22 e 23.

JENKINS. *Jenkins: The leading open source automation server*. 2023. Acessado em: 09 de Novembro, 2023. Disponível em: <https://www.jenkins.io/>. Citado na página 27.

JETBRAINS. *O que é a análise de código estático? | Guia de CI/CD do TeamCity | JetBrains*. JetBrains, 2023. Acessado em 1 de novembro de 2023. Disponível em: <https://www.jetbrains.com/pt-br/teamcity/ci-cd-guide/concepts/static-code-analysis/>. Citado 2 vezes nas páginas 20 e 21.

K3S. *K3s Lightweight Kubernetes*. 2024. <https://k3s.io/>. Acessado em: 02 de agosto de 2024. Citado na página 49.

PAN, J. Software testing. *Dependable Embedded Systems*, Citeseer, v. 5, n. 2006, p. 1, 1999. Citado na página 18.

POSTGRESQL. *PostgreSQL: The world's most advanced open source database*. 2023. Acessado em: 08 de Novembro, 2023. Disponível em: <https://www.postgresql.org/>. Citado 2 vezes nas páginas 37 e 40.

PRESSMAN, R.; MAXIM, D. B. R. *Software Engineering: A Practitioner's Approach*. McGraw-Hill Education, 2014. ISBN 9780078022128. Disponível em: <https://books.google.com.br/books?id=i8NmnAEACAAJ>. Citado 3 vezes nas páginas 17, 18 e 20.

S.A, S. *SonarQube Documentation*. 2023. Acessado em: 10 de novembro de 2023. Disponível em: <https://docs.sonarsource.com/sonarqube/10.2/>. Citado na página 28.

SANTOS, M. C.; ALCâNTARA, R. M. d. Estudo e análise de pipelines ci/cd escaláveis de alta disponibilidade. 2022. Citado na página 30.

SANTOS, V. B. d. Implementação de um pipeline de integração contínua para o front-end de uma aplicação web de código aberto. Brasil, 2023. Citado 4 vezes nas páginas 27, 30 e 32.

VALENTE, M. T. Engenharia de software moderna. *Princípios e Práticas para Desenvolvimento de Software com Produtividade*, v. 1, p. 24, 2020. Citado 4 vezes nas páginas 19 e 41.

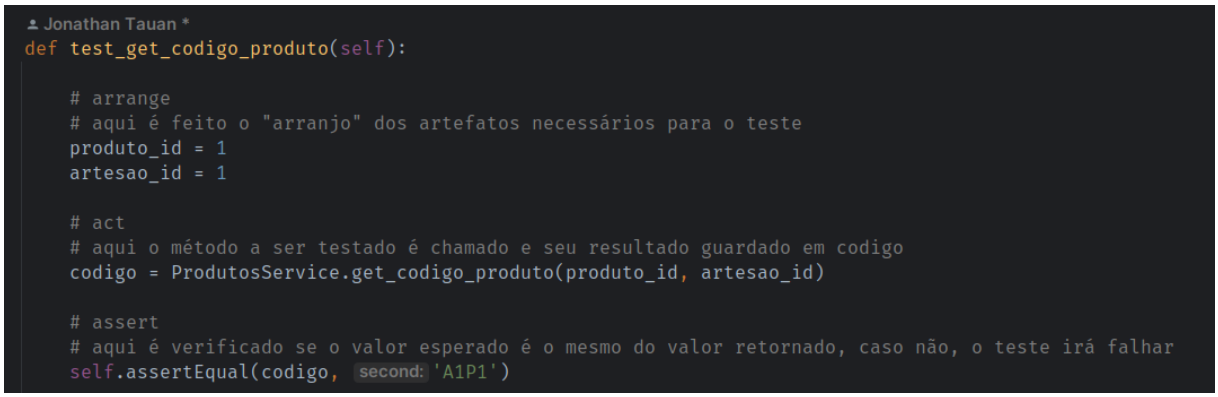
WU, Y.; ZHANG, Y.; WANG, T.; WANG, H. Characterizing the occurrence of dockerfile smells in open-source software: An empirical study. *IEEE Access*, v. 8, p. 34127–34139, 2020. Citado na página 26.

ZAMPETTI, F.; GEREMIA, S.; BAVOTA, G.; PENTA, M. D. Ci/cd pipelines evolution and restructuring: A qualitative and quantitative study. In: *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. [S.l.: s.n.], 2021. p. 471–482. Citado 6 vezes nas páginas 24, 30 e 34.

APÊNDICE A – Exemplos de testes automatizados

O teste da Figura 19 é um teste de unidade responsável por garantir o retorno correto do método *get_codigo_produto* da classe *ProdutoService*.

Figura 19 – Exemplo de teste unitário em Python



Fonte: Escrito pelo autor no projeto Sigarte

De início, é feito a configuração inicial do que é necessário para a realização do teste, nesse caso, a inicialização de *produto_id* e *artesao_id*. Logo após, é feita a chamada do método a ser testado e seu retorno é atribuído a variável *codigo*, e por fim, utilizamos um método da própria classe de testes do Django para garantir que o retorno foi igual ao esperado.

Já o código da Figura 20 é um exemplo de um teste funcional automatizado usando *Selenium* com Django 20. Ele está testando o processo de login de um usuário. O teste automatizado abre a página de login, preenche os campos de nome de usuário e senha e, em seguida, tenta fazer login. Finalmente, verifica se a URL atual é a mesma que a URL de login, indicando que o login foi bem-sucedido.

Figura 20 – Exemplo de teste funcional utilizando Selenium

```
def test_coord_login(self):
    driver = self.driver
    url = self.live_server_url + '/sigarte/accounts/login/'
    print('URL ' + url)
    driver.get(url)

    # Input the username
    username_elem = driver.find_element(By.NAME, 'username')
    username_elem.clear()
    username_elem.send_keys(self.coord.username)

    # Input the password
    password_elem = driver.find_element(By.NAME,'password')
    password_elem.clear()
    password_elem.send_keys(self.coord.password)

    # Press submit button (Botão Entrar)
    driver.find_element(By.XPATH, '//button')

    self.assertEqual(driver.current_url, url, msg="Algo deu errado")
```

Fonte: Sigarte, 2023

APÊNDICE B – GitHub Actions do Sigarte

Figura 21 – Arquivo de configuração do CI do GitHub Actions do Projeto

```yaml
name: Django CI

on:
  push:
    branches: [ "main" ]
  pull_request:
    branches: [ "main", "dev" ]
    types: [opened, synchronize, reopened]

jobs:
  build:

    runs-on: ubuntu-latest
    strategy:
      max-parallel: 4
      matrix:
        python-version: [3.8]

    steps:
    - uses: actions/checkout@v3
      with:
        # Disabling shallow clone is recommended for improving relevancy of reporting
        fetch-depth: 0
    - name: Set up Python ${{ matrix.python-version }}
      uses: actions/setup-python@v3
      with:
        python-version: ${{ matrix.python-version }}
    - name: Install Dependencies
      run: |
        python -m pip install --upgrade pip
        pip install -r requirements.txt
    - name: Copying configurations
      run: |
        cp .env.example .env
        source .env
    - name: Run Tests
      run: |
        python manage.py test
    - name: Run Coverage
      run: |
        coverage run manage.py test
        coverage xml
    - name: SonarQube Scan
      uses: sonarsource/sonarqube-scan-action@master
      env:
        SONAR_TOKEN: ${{ secrets.SONAR_TOKEN }}
        SONAR_HOST_URL: ${{ secrets.SONAR_HOST_URL }}
    - name: SonarQube Quality Gate check
      uses: sonarsource/sonarqube-quality-gate-action@master
      # Force to fail step after specific time
      timeout-minutes: 5
      env:
       SONAR_TOKEN: ${{ secrets.SONAR_TOKEN }}
```

Fonte: Sigarte, 2023

Como explicado na Seção 4.1.5, a Figura 21 representa o arquivo de configuração *.yml* ou *.yaml*, que são utilizados para definir um fluxo de trabalho para o pipeline do GitHub Actions

de forma descritiva.

Segue abaixo uma explicação resumida do arquivo:

1. Eventos (on): Este trecho define eventos os quais disparam a execução do pipeline, neste caso, quando ocorre um *push* na ramificação principal ou um *pull request* na ramificação principal ou de desenvolvimento.
2. Trabalhos (jobs): Aqui está configurado um trabalho chamado de "*build*", dentro dele estarão algumas configurações de ambiente e as etapas que serão executadas.
3. Etapas (steps): Define quais etapas serão executadas no trabalho:

    a) **Checkout**: realiza o download do código mais recente da ramificação onde o evento foi disparado.

    b) **Setup Python**: Configura a linguagem Python e sua versão no servidor.

    c) **Install Dependencies**: Atualiza o *pip*, gerenciador de pacotes Python e instala as dependências do projeto listadas no arquivo *requitements.txt*.

    d) **Install Dependencies**: Atualiza o *pip*, gerenciador de pacotes Python e instala as dependências do projeto listadas no arquivo *requitements.txt*.

    e) **Copying Configurations**: Copia os arquivos de configurações para o diretório onde será executado a aplicação Python.

    f) **Run Tests**: Executa os testes automatizados do projeto.

    g) **Run Coverage**: Realiza a análise da cobertura de testes do projeto e gera um relatório *XML*.

    h) **SonarQube Scan**: Submete o código a uma análise realizada pela ferramenta de análise de código estático SonarQube, detalhada em 2.8.3.

    i) **SonarQube Quality Gate Check**: Verifica se o código passou na métricas de qualidade do SonarQube.

Todas essas etapas são executadas em um servidor remoto fornecido pela própria plataforma do GitHub.

Além disso, outro arquivo de configuração é usado para definir as etapas do processo de CD do pipeline do GitHub Actions, como mostrado na Figura 22. Ele é semelhante ao arquivo do CI, mas mais simples, pois define um evento de gatilho, nesse caso, quando houver uma atualização na ramificação de desenvolvimento. E configura apenas uma etapa dentro do trabalho, que é a execução dos comandos para acessar o servidor que hospeda o ambiente de desenvolvimento do Sigarte e atualizar a versão do sistema, conforme descrito na Seção 4.1.3.

Figura 22 – Arquivo de configuração do CD do GitHub Actions do Projeto

```yaml
name: Django CD

on:
  push:
    branches: [ "dev" ]

jobs:
  deploy:
    runs-on: ubuntu-latest
    steps:
    - name: executing remote ssh commands using password
      uses: appleboy/ssh-action@master
      with:
        host: ${{ secrets.SSH_HOST_DEV }}
        username: ${{ secrets.SSH_USER_DEV }}
        port: ${{ secrets.SSH_PORT_DEV }}
        key: ${{ secrets.SSH_KEY_DEV }}
        passphrase: ${{ secrets.SSH_PASSPHRASE_DEV }}
        script: |
          cd ~/apps/sigarte
          git pull origin dev
          docker build --no-cache -t sigarte:latest .
          docker compose up -d
```

Fonte: Sigarte, 2023

APÊNDICE C – Dockerfile e Docker Compose

A Figura 23 representa o conteúdo do arquivo *Dockerfile* utilizado na criação da imagem Docker, tendo como base a aplicação do Sigarte.

Figura 23 – Dockerfile do Sigarte

```
FROM python:3.9-alpine3.13

LABEL description="Projeto SIGARTE - Casa do Artesão do Seridó"
LABEL maintainer="labens.dct.ufrn.br"

ENV PYTHONDONTWRITEBYTECODE=1

ENV PYTHONUNBUFFERED=1
EXPOSE 8002
ENV PORT=8002

WORKDIR /code

COPY requirements.txt .
RUN pip install --upgrade pip
RUN pip install --no-cache-dir -r requirements.txt

COPY apps/ .
COPY sigarte/ .
COPY static/ .
COPY templates/ .
COPY manage.py .

RUN adduser --disabled-password --no-create-home appuser && chown -R appuser /code

ENV PATH="/code:$PATH"

USER appuser
```

Fonte: Sigarte, 2023

Segue abaixo uma simples explicação do arquivo:

1. **Definição do Ambiente de Execução:** O arquivo Dockerfile começa especificando a imagem base a ser utilizada, neste caso, "python:3.9-alpine3.13". Essa imagem é otimizada para ambientes Alpine Linux, proporcionando um contêiner mais leve.

2. **Configuração de Labels:** São adicionadas descrições e informações do mantenedor do contêiner por meio de labels, facilitando a identificação e documentação.

3. **Configuração do Ambiente Python:** Duas variáveis de ambiente são configuradas com o comando *ENV*, para otimizar o ambiente Python no contêiner, desativando a geração de arquivos .pyc e desativando o buffer de saída.

4. **Exposição de Porta:** A porta 8002 é exposta para permitir a comunicação externa com a aplicação que será executada no contêiner.

5. **Definição do Diretório de Trabalho:** O diretório de trabalho dentro do contêiner é configurado como /code.

6. **Instalação de Dependências:** O arquivo requirements.txt, que lista as dependências do projeto, é copiado para o contêiner, e em seguida, o *pip* é atualizado e as dependências são instaladas por meio do comando *RUN*.

7. **Cópia de Arquivos do Projeto:** Os diretórios e arquivos essenciais do projeto, como *apps*, arquivos estáticos, *templates* e o arquivo *manage.py*, são copiados para o diretório de trabalho utilizando o comando "COPY".

8. **Criação de Usuário:** Um usuário não raiz chamado "appuser"é criado, e permissões são concedidas para acessar o diretório */code*. Isso é uma prática de segurança para evitar a execução de processos com superpoderes.

9. **Configuração do Caminho do Código:** O caminho do código é adicionado ao PATH para facilitar a execução de comandos.

10. **Definição do Usuário de Execução:** O usuário "appuser"é definido como o usuário de execução para melhorar a segurança do contêiner.

Essas configurações permitem a criação de um contêiner otimizado para executar a aplicação Django do projeto Sigarte, fornecendo um ambiente isolado e consistente para a execução da aplicação.

O arquivo *Docker Compose*, Figura 24, do formato *.yml* descreve as instruções para executar um contêiner Docker a partir da imagem criada com o *Dockerfile*:

1. **Versão e Serviços:** O arquivo utiliza a versão 3.9 do *Docker Compose* e define um serviço chamado "web"para executar a aplicação Sigarte.

2. **Nome do Contêiner e Imagem:** O contêiner é nomeado como "sigarte-web", e a imagem utilizada é "sigarte:latest", indicando a versão mais recente.

3. **Variáveis de Ambiente:** Um arquivo de ambiente *.env* é especificado para fornecer variáveis de ambiente ao contêiner, facilitando a configuração da aplicação.

Figura 24 – Arquivo Docker Compose do Sigarte

```yaml
version: '3.9'

services:
  web:
    container_name: sigarte-web
    image: sigarte:latest
    env_file:
      - .env
    build:
      context: .
      dockerfile: ./Dockerfile

    command: >
      sh -c  "python manage.py collectstatic --noinput
              python manage.py migrate --noinput
              python manage.py initadmin
              python manage.py initgroups
              python manage.py load_materia_prima
              python manage.py load_tecnicas
              python manage.py runserver 0.0.0.0:8002"
    volumes:
      - .:/code
    ports:
      - 8002:8002

networks:
  default:
    external: true
    name: labens-network
```

Fonte: Sigarte, 2023

4. **Construção do Contêiner:** O contêiner é construído a partir do contexto atual (context: .) utilizando o *Dockerfile* fornecido em *./Dockerfile*.

5. **Comando de Inicialização:** O comando especificado inicia a aplicação Django executando uma série de comandos, como coleta estática, migração do banco de dados e inicialização de dados iniciais.

6. **Volumes:** O diretório atual é mapeado para */code* dentro do contêiner, permitindo a sincronização de arquivos entre o hospedeiro e o contêiner.

7. **Portas:** A porta 8002 do host é mapeada para a porta 8002 do contêiner, permitindo acesso à aplicação por meio dessa porta.

8. **Redes:** O serviço é conectado à rede "labens-network", que é definida como externa, indicando que a rede já existe e foi criada anteriormente. Isso facilita a comunicação entre contêineres em diferentes serviços.

Essas configurações garantem a consistência e a replicabilidade do ambiente de desenvolvimento e produção. Apesar de utilizarmos o banco de dados PostgreSQL, não o utilizamos via Docker, por isso, neste arquivo não há especificação de um contêiner SGBD.

APÊNDICE D – Arquivo de configuração do SonarQube

Arquivo presente na raiz do SigArte que configura alguns comportamentos de análise do SonarQube.

Código D.1 – Configuração do Sonar (sonar-project.properties)

```
1  sonar.projectKey=sigarte
2  # this is the name and version displayed in the SonarQube UI.
3  sonar.projectName=
4  sonar.projectVersion=
5  sonar.organization=
6
7  # Path is relative to the sonar-project.properties file. Replace "\" by "/"
       on Windows.
8  # This property is optional if sonar.modules is set.
9  sonar.sources=.
10 sonar.sources.inclusions=**/**.py
11 sonar.exclusions=**/tests/**, docs/**, sigarte/**, **/static/**, **/
       templates/**, **/migrations/**, **/__pycache__/**, **/admin.py, **/
       __init__.*, manage.py
12
13 # Language
14 sonar.language=py
15 sonar.python.version=3.8
16
17 sonar.scm.provider=git
18 sonar.links.scm=
19 sonar.links.issue=https:
20
21 # Coverage / Unit Tests
22 sonar.tests=.
23 sonar.test.inclusions=**/test_**.py, **/tests_**.py, **/tests.py
24
25 sonar.python.coverage.reportPaths=coverage.xml
26 sonar.python.coveragePlugin=cobertura
```

APÊNDICE E – *Jenkinsfile* CI

Explicado detalhadamente no Capítulo de desenvolvimento 4.4.

Código E.1 – Jenkinsfile do *pipeline* de CI

```
pipeline {
    agent any

    environment {
        VIRTUAL_ENV = 'venv'
        PATH = "${env.WORKSPACE}/${env.VIRTUAL_ENV}/bin:${env.PATH}"
    }
    stages {
        stage('Checkout') {
            steps {
                checkout scm
            }
        }

        stage('Install Dependencies') {
            steps {
                script {
                    sh 'python3 -m venv venv'
                    sh "python -m pip install --upgrade pip"
                    sh "pip install -r requirements.txt" // --upgrade --
    force-reinstall
                    sh 'cp .env.example .env'
                }
            }
        }

        stage('Tests') {
            steps {
                script {
                    sh 'python manage.py test'
                }
            }
        }

        stage('Run Coverage') {
            steps {
                script {
                    sh 'python -m coverage run manage.py test'
                    sh 'python -m coverage xml'
```

```
39                  }
40              }
41          }
42
43          stage("SonarQube Build & Analysis") {
44              steps {
45                  withSonarQubeEnv('sigarte') {
46                      sh '''
47                      . venv/bin/activate
48                      pip install pysonar-scanner
49                      pysonar-scanner -Dsonar.token=${SONAR_AUTH_TOKEN} -
    Dsonar.host.url=${SONAR_HOST_URL}
50                      '''
51                  }
52              }
53          }
54
55          // uses webhook
56          stage("Quality Gate") {
57              steps {
58                  timeout(time: 7, unit: 'MINUTES') {
59                      waitForQualityGate abortPipeline: true
60                  }
61              }
62          }
63      }
64  }
```

APÊNDICE F – *Jenkinsfile* CD

Explicado detalhadamente no Capítulo de desenvolvimento 4.6.

Código F.1 – Jenkinsfile do *pipeline* de CD

```
pipeline {
    agent any

    environment {
        VIRTUAL_ENV = 'venv'
        PATH = "${env.WORKSPACE}/${env.VIRTUAL_ENV}/bin:${env.PATH}"
    }
    stages {
        // Continuous Integration Stages ...

        stage("Build Docker Image") {
            when {
                anyOf {
                    branch 'dev'
                    expression { return env.TAG_NAME != null }
                }
            }
            steps {
                script{
                    dockerapp = docker.build("labens/sigarte")
                }
            }
        }

        stage("Push Docker Image") {
            when {
                anyOf {
                    branch 'dev'
                    expression { return env.TAG_NAME != null }
                }
            }
            steps {
                script {
                    docker.withRegistry('https://registry.hub.docker.com',
    'docker-
hub-credential-id') {
                        if (env.TAG_NAME != null) {
                            docker.image("labens/sigarte:${env.TAG_NAME}").
    push()
```

```
38                              docker.image("labens/sigarte:latest").push()
39                              env.DOCKER_IMAGE_TAG = env.TAG_NAME
40                          } else if (env.GIT_BRANCH == 'dev') {
41                              docker.image("labens/sigarte:${env.BRANCH_NAME
    }-${env.BUILD_ID}").push()
42                              env.DOCKER_IMAGE_TAG = "${env.BRANCH_NAME}-${
    env.BUILD_ID}"
43                          }
44                      }
45                  }
46              }
47          }
48
49          stage("Deploy") {
50              when {
51                  anyOf {
52                      branch 'dev'
53                      expression { return env.TAG_NAME != null }
54                  }
55              }
56              steps {
57                  script {
58                      withKubeConfig([credentialsId: 'kubeconfig']) {
59                          if (env.TAG_NAME != null) {
60                              sh 'sed -i "s/{{tag}}/${env.TAG_NAME}/g" ./k8s/
    deployment.yaml'
61                          } else if (env.GIT_BRANCH == 'dev') {
62                              sh 'sed -i "s/{{tag}}/${env.BRANCH_NAME}-${env.
    BUILD_ID}/g" ./k8s/deployment.yaml'
63                          }
64                          sh 'kubectl apply -f k8s/deployment.yaml'
65                          sh 'kubectl apply -f k8s/configMap.yaml'
66                      }
67                  }
68              }
69          }
70      }
71  }
```

APÊNDICE G – Arquivo de Manifesto de *Deployment*

Arquivo de manifesto K8s deployment.yaml, presente no repositório do projeto.

Código G.1 – deployment.yaml

```
1  apiVersion: apps/v1
2  kind: Deployment
3  metadata:
4    name: sigarte
5  spec:
6    replicas: 2
7    strategy:
8      type: RollingUpdate
9      rollingUpdate:
10       maxUnavailable: 0
11       maxSurge: 2       #create new pods during update
12   selector:
13     matchLabels:
14       app: sigarte
15   template:
16     metadata:
17       labels:
18         app: sigarte
19     spec:
20       containers:
21       - name: sigarte
22         image: docker.io/dockerhub/sigarte:{{tag}} # labens/sigarte:{{tag}}
23         # imagePullPolicy: Always
24         envFrom:
25         - configMapRef:
26             name: sigarte-env
27         command: ["sh", "-c"]
28         args:
29           - >
30             python manage.py migrate --noinput &&
31             python manage.py initadmin &&
32             python manage.py initgroups &&
33             python manage.py load_materia_prima &&
34             python manage.py load_tecnicas &&
35             python manage.py collectstatic --noinput &&
36             python manage.py runserver 0.0.0.0:8002
37         ports:
```

```
38          - containerPort: 8002
39          readinessProbe:
40            initialDelaySeconds: 50
41            periodSeconds: 60
42            failureThreshold: 3
43            httpGet:
44              path: /sigarte/portal/
45              port: 8002
46              httpHeaders:
47                - name: host
48                  value: health-check-probe
49  ---
50  # Service -> ClusterIP
51  apiVersion: v1
52  kind: Service
53  metadata:
54    name: sigarte-svc
55  spec:
56    selector:
57      app: sigarte
58    ports:
59    - protocol: TCP
60      port: 80
61      targetPort: 8002
62    type: ClusterIP
```

APÊNDICE H – Arquivo de Configuração Kubernetes

Arquivo de manifesto K8s configMap.yaml, presente no repositório do projeto.

Código H.1 – configMap.yaml

```yaml
#Config Map = .env
apiVersion: v1
kind: ConfigMap
metadata:
  name: sigarte-env
data:
  SECRET_KEY: "ZGFuZ28taW5zZWN1cmUtXiR6QDp3SFzYzV1NXpmKHg3dW9AOHBveXZ3IX5h"
  DATABASE_PASSWORD: "password"
  EMAIL_HOST_PASS: "xyxyxyxyxyxyx"
  DJANGO_SUPERUSER_PASSWORD: "password"

  DJANGO_SETTINGS_MODULE: "sigarte.settings"
  ALLOWED_HOSTS: "localhost, 127.0.0.1, testserver, ClusterIP, health-check
    -probe"
  DEBUG: "true"
  STAGE: "Development"
  DATABASE_NAME: "db"
  DATABASE_USER: "user"
  DATABASE_PORT: "5432"
  TEST_DATABASE_NAME: "sigarte_testdb"
  DATABASE_HOST: "localhost"
  DATABASE_HOST_DOCKER: "postgres-server"
  HOME_DB: "../sigarte_db"
  EMAIL_HOST_USER: "email@email.com"
  DJANGO_SUPERUSER_USERNAME: "admin"
  DJANGO_SUPERUSER_EMAIL: "admin@admin.com"
  PYTHONPATH: "${PYTHONPATH}:/code"
```