UNIVERSIDADE FEDERAL DO CEARÁ

CAMPUS SOBRAL

CURSO DE GRADUAÇÃO EM ENGENHARIA DA COMPUTAÇÃO

FRANCISCO CASSIANO DE VASCONCELOS SOUZA

PIPELINE DE IMPLANTAÇÃO PARA AUTOMATIZAR A ENTREGA DE SOFTWARE NA PREFEITURA MUNICIPAL DE SOBRAL (PMS) - UM ESTUDO DE CASO

SOBRAL

2025

FRANCISCO CASSIANO DE VASCONCELOS SOUZA

PIPELINE DE IMPLANTAÇÃO PARA AUTOMATIZAR A ENTREGA DE SOFTWARE NA PREFEITURA MUNICIPAL DE SOBRAL (PMS) - UM ESTUDO DE CASO

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Engenharia da Computação do Campus Sobral da Universidade Federal do Ceará, como requisito parcial à obtenção do grau de bacharel em Engenharia da Computação.

Orientador: Prof. Dr. Ialis Cavalcante de Paula Junior

SOBRAL

2025

Dados Internacionais de Catalogação na Publicação
Universidade Federal do Ceará
Sistema de Bibliotecas
Gerada automaticamente pelo módulo Catalog, mediante os dados fornecidos pelo(a) autor(a)

S238p Souza, Francisco Cassiano de Vasconcelos.
PIPELINE DE IMPLANTAÇÃO PARA AUTOMATIZAR A ENTREGA DE SOFTWARE NA PREFEITURA MUNICIPAL DE SOBRAL (PMS) : ESTUDO DE CASO / Francisco Cassiano de Vasconcelos Souza. – 2025.
63 f. : il. color.

Trabalho de Conclusão de Curso (graduação) – Universidade Federal do Ceará, Campus de Sobral, Curso de Engenharia da Computação, Sobral, 2025.
Orientação: Prof. Dr. Iális Cavalcante de Paula.

1. Pipelines. 2. CI/CD. 3. Integração Contínua. 4. Entrega Contínua. 5. DevOps. I. Título.
CDD 621.39

FRANCISCO CASSIANO DE VASCONCELOS SOUZA

PIPELINE DE IMPLANTAÇÃO PARA AUTOMATIZAR A ENTREGA DE SOFTWARE NA PREFEITURA MUNICIPAL DE SOBRAL (PMS) - UM ESTUDO DE CASO

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Engenharia da Computação do Campus Sobral da Universidade Federal do Ceará, como requisito parcial à obtenção do grau de bacharel em Engenharia da Computação.

Aprovada em: 07 de agosto de 2025

BANCA EXAMINADORA

Prof. Dr. Ialis Cavalcante de Paula Junior (Orientador)
Universidade Federal do Ceará (UFC)

Prof. Dr. Thiago Iachiley Araujo de Souza
Universidade Federal do Ceará (UFC)

Prof. Alex de Sousa Ramos
Universidade Federal do Ceará (UFC)

AGRADECIMENTOS

Agradeço, primeiramente, a Deus, pela força, sabedoria e amparo em cada etapa desta caminhada. Sem a confiança nEle, este trabalho não teria sido possível. Aos meus pais, Sr. Assis e Sra. Rozane, minha eterna gratidão pelo esforço incansável em me proporcionar aquilo que não tiveram oportunidade: o acesso ao ensino. Mais do que isso, agradeço pelos valores que me ensinaram — lições de caráter, humildade e perseverança que nenhuma universidade é capaz de transmitir.

Estendo meus sinceros agradecimentos ao meu orientador, Dr. Ialis, por aceitar este desafio comigo, pela orientação dedicada e pela paciência ao longo do processo. Agradeço também a todo o corpo docente da Universidade Federal do Ceará, que contribuiu significativamente para minha formação. Aos amigos Salmo Mascarenhas, Cris Albuquerque, Jessika Sato, e a todos os colegas que, direta ou indiretamente, fizeram parte desta jornada, meu muito obrigado. Cada conversa, apoio e incentivo foi essencial para que eu seguisse em frente e não desistisse.

“Faça o teu melhor na condição que você tem, enquanto você não tem condições melhores para fazer melhor ainda!”

(Mario Sergio Cortella)

RESUMO

A entrega manual de software, ainda predominante em muitas organizações, gera prazos longos, riscos elevados e alta propensão a erros, impactando diretamente a eficiência e a qualidade dos sistemas disponibilizados. Na Prefeitura Municipal de Sobral (PMS), identificou-se uma oportunidade de aprimorar os processos de desenvolvimento e implantação de sistemas, visando maior eficiência e confiabilidade. Este trabalho apresenta a construção e a implementação de uma pipeline de implantação automatizada, baseada nas práticas de Integração Contínua (CI) e Entrega Contínua (CD), como estratégia para transformar o ciclo de entrega de software da instituição. Por meio de um estudo de caso, com coletas de dados antes e depois da implantação, foram analisados os impactos da solução proposta, com foco em eficiência, qualidade e coordenação entre as equipes. Os resultados evidenciam ganhos significativos em agilidade, redução de erros e maior confiabilidade no processo de entrega de software, reforçando a importância da automação e da cultura *DevOps* no contexto do setor público.

Palavras-chave: Pipelines. CI/CD. Integração Contínua. Entrega Contínua. DevOps. Gitlab. Entrega de Software.

ABSTRACT

The manual delivery of software, still predominant in many organizations, results in long lead times, increased risks, and a high propensity for errors, directly impacting the efficiency and quality of the deployed systems. At the Municipality of Sobral (PMS), an opportunity was identified to improve the development and deployment processes, aiming for greater efficiency and reliability. This work presents the design and implementation of an automated deployment pipeline, based on Continuous Integration (CI) and Continuous Delivery (CD) practices, as a strategy to transform the institution's software delivery cycle. Through a case study, with data collected before and after the implementation, the impacts of the proposed solution were analyzed, focusing on efficiency, quality, and team coordination. The results show significant improvements in agility, error reduction, and greater reliability in the software delivery process, highlighting the importance of automation and the DevOps culture in the public sector context.

Keywords: Pipelines. CI/CD. Continuous Integration. Continuous Delivery. DevOps. GitLab. Software Delivery.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE CÓDIGOS-FONTE

LISTA DE ABREVIATURAS E SIGLAS

CD	*Entrega Contínua*
CDT	*Implantação Contínua*
CI	*Integração Contínua*
IaC	*Infraestrutura como Código*
IBM	*International Business Machines Corporation*
PMS	*Prefeitura Municipal de Sobral*
TI	*Tecnologia da Informação*
YAML	*Yet Another Markup Language*

SUMÁRIO

1 INTRODUÇÃO

Em muitas empresas, a transição de um software para a produção é um procedimento intensamente manual, sujeito a erros e repleto de possíveis riscos. Embora seja comum observar um *Tempo de Ciclo* que se estende por meses, é surpreendente o número de organizações que enfrentam desafios ainda mais significativos, com ocorrências não muito raras de ciclos que se prolongam por mais de um ano (HUMBLE; FARLEY, 2013).

Certamente, a implementação de software em ambientes produtivos nessas organizações não segue um processo padronizado ou seguro; pelo contrário, é um processo manual que frequentemente demanda a colaboração de uma equipe, mesmo quando se trata de cenários de teste ou homologação (HUMBLE, 2013). Diante dessa realidade, ao observar o setor de desenvolvimento da PMS, foram identificadas amplas possibilidades de aprimoramento. Essas melhorias visam padronizar processos, promovendo uma entrega de software de forma ágil e segura.

A adoção da abordagem DevOps, que busca integrar os times de desenvolvimento e operações, promovendo maior colaboração e responsabilidade compartilhada, tem permitido às organizações entregar software de maneira mais eficiente, confiável e com menor risco (SILVA, 2020). Nesse contexto, as organizações buscam criar soluções personalizadas para construir produtos capazes de automatizar integralmente os procedimentos de *Integração Contínua* (CI), *Entrega Contínua* (CD) e *Implantação Contínua* (CDT) (DONCA I.-C.; STAN, 2022).

Conforme destacado por Humble (2013), uma prática eficaz no ciclo de desenvolvimento de software, da concepção à entrega final, é a adoção de uma Pipeline de Implantação, fundamentada no conceito de CI e CD. Essa abordagem representa uma automatização abrangente, compreendendo a compilação de todas as partes da aplicação até a transferência do software do controle de versão para os usuários finais.

Este trabalho abrange um estudo de caso realizado na PMS. O principal objetivo deste trabalho é desenvolver uma pipeline automatizada de implantação de software, baseada nos princípios fundamentais do *DevOps*, incluindo CI/CD. A proposta visa otimizar e padronizar o processo de entrega de software, buscando resolver os desafios enfrentados na PMS. O intuito é melhorar a eficiência, confiabilidade e agilidade da entrega de software, alinhando-a às melhores práticas e tendências contemporâneas.

1.1 Objetivos

Este trabalho tem como objetivo investigar de forma abrangente como a implementação de uma pipeline de implantação automatizada pode abordar e solucionar os desafios inerentes à entrega de software na Prefeitura Municipal de Sobral (PMS), contribuindo para a melhoria e qualidade do software e minimizando os problemas enfrentados pela equipe de desenvolvimento durante o processo de entrega.

Destaca-se os principais objetivos específicos:

* Diagnosticar e avaliar os problemas da entrega de software na PMS.
* Compreender os ambientes, métodos e ferramentas atuais utilizados na entrega de software na PMS.
* Desenvolver e implementar uma pipeline de implantação automatizada, adaptada e integrada às necessidades e desafios identificados na PMS.
* Identificar e comparar os resultados pré e pós-implantação da pipeline de entrega automatizada.

1.2 Justificativa

Observa-se que no contexto atual de desenvolvimento da PMS, a entrega de uma nova versão de software ou de um sistema como um todo requer a colaboração de várias partes, incluindo pessoas de testes, equipe de operações e infraestrutura, e a necessidade de solucionar possíveis desafios imprevistos que venham a surgir durante o processo de desenvolvimento. Como consequência desses fatores mencionados, observa-se um prolongamento no prazo de entrega, aumentando a suscetibilidade a erros e expondo o projeto a riscos difíceis de quantificar.

O conceito de DevOps visa otimizar o ciclo de vida do desenvolvimento de aplicações, proporcionando métodos e ferramentas que abrangem desde a fase de desenvolvimento até a entrega do produto final ao cliente. Diante dos desafios enfrentados no processo de entrega de software da PMS, a implementação de uma pipeline de implantação automatizada surge como uma solução viável. Essa abordagem, fundamentada em princípios essenciais como Integração Contínua CI e Entrega Contínua CD, busca promover uma entrega de software mais eficiente e estruturada, com o objetivo de otimizar o processo de desenvolvimento e garantir a uniformidade na entrega de produtos (LIMA, 2021).

1.3 Trabalhos Relacionados

O trabalho de Mariel Madlene dos Santos aborda a importância do movimento DevOps para a entrega contínua e confiável de software, destacando a integração contínua e o pipeline de entrega como práticas essenciais. Através do mapeamento do processo do pipeline de entrega contínua de um software embarcado, são propostas melhorias visando acelerar o desenvolvimento, garantir a qualidade do software e promover a melhoria contínua. A documentação detalhada do pipeline e a análise para sugestão de melhorias evidenciam a relevância de manter processos atualizados e em constante evolução, alinhados com as práticas e ferramentas de DevOps para obter resultados eficazes e confiáveis no desenvolvimento de software.

O trabalho apresentado por Pedro Henrique Oliveira Lima aborda a importância do conceito de DevOps, que visa a integração entre desenvolvimento e operações para agilizar o processo de desenvolvimento e entrega de software. Destaca-se a utilização de pipelines para automatizar processos repetíveis de integração contínua e entrega contínua. As ferramentas Jenkins e GitLab são apresentadas como essenciais nesse contexto, permitindo a implementação e comparação de pipelines para melhorar a produtividade do desenvolvimento, facilitar a detecção de bugs e acelerar a entrega de software de forma confiável e eficiente, contribuindo para a qualidade e sucesso dos projetos de software.

O artigo de Ionut-Catalin Donca aborda a implementação de um gerador de pipeline automatizado baseado em práticas ágeis para integração contínua e entrega contínua de aplicativos. A solução proposta inclui a versionamento semântico do código, compilações automáticas usando Docker com armazenamento em cache, envio de entregáveis para Docker Container Registry e implantação em um cluster Kubernetes via Helm. Essas práticas garantem alta disponibilidade, escalabilidade rápida, detecção de vulnerabilidades, reversão automática para versões estáveis e acionamento de ações com base em alterações no código-fonte. A integração contínua e entrega contínua são fundamentais para agilidade, qualidade e eficiência no desenvolvimento de software, permitindo a padronização, redução de custos, melhoria da qualidade e aumento da produtividade da equipe de desenvolvimento e operações.

1.4 Estrutura

O presente trabalho está organizado em 6 seções. Na Seção 1, é apresentada a introdução, contextualizando o problema, os objetivos e a relevância do estudo. A Seção 2 aborda a fundamentação teórica, discutindo os conceitos-chave necessários para a compreensão do trabalho. Na Seção 3, são apresentadas as ferramentas e tecnologias utilizadas no desenvolvimento da solução proposta. A Seção 4 detalha a metodologia adotada para a condução do estudo. Em seguida, os resultados obtidos são descritos e analisados na Seção 5. Por fim, a Seção 6 apresenta as conclusões do trabalho, bem como sugestões para estudos e implementações futuras.

2 FUNDAMENTAÇÃO TEÓRICA

Neste tópico, exploraremos os conceitos essenciais que embasam a implementação de práticas de *DevOps*, com foco em como essas metodologias e ferramentas contribuem para a integração entre desenvolvimento e operações, promovendo automação, agilidade e confiabilidade no ciclo de entrega de software. Esses fundamentos teóricos ajudarão a contextualizar os benefícios observados na adoção da pipeline de CI/CD ao longo deste trabalho.

2.1 Contexto Pré-DevOps: Separação entre Desenvolvimento e Operações

No cenário pré-DevOps, as organizações de desenvolvimento de software frequentemente enfrentavam uma clara separação entre as equipes de desenvolvimento e operações. Essa divisão resultava em silos organizacionais e culturais, onde as equipes de desenvolvimento eram responsáveis pela criação de código e novas funcionalidades, enquanto as equipes de operações ficavam encarregadas da implementação, implantação e manutenção dessas aplicações em ambientes de produção (HUTTERMANN, 2012).

Ainda de acordo com Huttermman (2012), essa separação tradicional muitas vezes resultava em desafios significativos, como atrasos na entrega de software, conflitos entre equipes, falta de visibilidade e responsabilidade compartilhada, e dificuldades na identificação e correção de problemas em produção. As equipes de desenvolvimento muitas vezes estavam focadas apenas no desenvolvimento de código, sem considerar as implicações operacionais, enquanto as equipes de operações frequentemente lutavam para lidar com mudanças rápidas e frequentes nas aplicações sem o devido envolvimento desde o início do processo de desenvolvimento.

Essa separação entre desenvolvimento e operações criava barreiras à colaboração eficaz, resultando em uma lacuna entre as expectativas dos desenvolvedores e as realidades operacionais. Conforme descrito por John Willis no livro "Manual de DevOps", essa divisão era simbolicamente representada pelo "muro da confusão", uma metáfora que ilustrava a prática de lançar o trabalho por cima do muro entre as equipes de desenvolvimento e operações. Essa prática contribuía para a falta de comunicação e colaboração entre as partes, dificultando a entrega eficiente e de qualidade do software. Como resultado, enfrentavam-se desafios significativos para alcançar lançamentos de software mais rápidos, confiáveis e seguros, e para responder rapidamente às demandas do mercado em constante mudança.

No entanto, com o advento do movimento *DevOps*, essas barreiras começaram a ser

derrubadas, dando origem a uma nova abordagem que promove a colaboração, comunicação e automação entre as equipes de desenvolvimento e operações.

2.2 DevOps

O movimento *DevOps* surgiu como uma resposta aos desafios enfrentados pelas organizações no cenário pré-DevOps, onde a separação entre desenvolvimento e operações criava barreiras à colaboração eficaz e impactava negativamente na entrega de software. O DevOps é uma abordagem cultural, organizacional e prática que visa promover a colaboração, comunicação e integração entre equipes de desenvolvimento e operações, com o objetivo de melhorar a eficiência, confiabilidade e agilidade na entrega de software (GITLAB, 2023).

A história do *DevOps* remonta ao início dos anos 2000, quando a comunidade de TI começou a reconhecer a necessidade de superar as divisões entre desenvolvimento e operações. Inicialmente, o termo "DevOps"foi criado em 2009 durante o evento DevOpsDays em Ghent, na Bélgica, onde os participantes discutiram sobre a necessidade de uma abordagem mais colaborativa e integrada para o desenvolvimento de software. Este encontro marcou o início de uma série de discussões e atividades em torno do movimento DevOps, que rapidamente ganhou destaque na indústria de *Tecnologia da Informação* (TI) (KIM *et al.*, 2018).

Desde então, o movimento *DevOps* tem crescido rapidamente, influenciando práticas e metodologias em toda a indústria de TI. O *DevOps* não se limita apenas a uma ferramenta ou tecnologia específica, mas sim a uma filosofia que promove princípios como automação, integração contínua CI, entrega contínua CD, monitoramento e feedback contínuo.

2.3 O CI/CD

A Integração Contínua (CI) e a Entrega Contínua (CD) ou Implantação Contínua (CDT) são práticas fundamentais no contexto do desenvolvimento de software, especialmente dentro da metodologia *DevOps*. Ao adotar CI/CD, as organizações podem reduzir significativamente os custos e os riscos associados ao desenvolvimento de software, permitindo entregas mais ágeis e adaptáveis às necessidades do mercado.

2.3.1 Integração Contínua (CI)

A prática de Integração Contínua (CI) está intrinsecamente ligada à ideia de que todos os membros da equipe se integrem regularmente, idealmente diariamente, em um repositório de código-fonte centralizado e controlado. Ela consiste na automação da integração de alterações de código provenientes de diferentes contribuidores em um único projeto de software (FOWLER, 2024).

Como uma prática central do *DevOps*, a Integração Contínua permite que os desenvolvedores mesclem com frequência as alterações de código em um repositório central, onde são executadas builds e testes automatizados. Essa abordagem facilita a detecção precoce de conflitos e erros, promovendo a colaboração eficaz entre os membros da equipe e garantindo a estabilidade e a qualidade do software em desenvolvimento.

Ao adotar a Integração Contínua, as organizações podem reduzir significativamente o tempo e os esforços necessários para a integração de código, garantindo uma entrega mais ágil e confiável de software. Durante o processo de integração, são realizados testes automatizados abrangentes para cada modificação e mesclagem, resultando em uma abordagem que reduz consideravelmente os riscos de problemas de segurança, conflitos e a detecção de erros logo na etapa de desenvolvimento. O objetivo final da CI é assegurar uma entrega ágil e segura de código funcional (GITLAB, 2023).

2.3.2 Entrega Contínua (CD)

A Entrega Contínua (CD) é uma extensão natural da Integração Contínua (CI) e representa uma etapa crucial no ciclo de vida de desenvolvimento de software. Como mencionado por Red Hat (2023), a CD é o resultado positivo de um processo de CI bem-sucedido, no qual o software é lançado automaticamente em um repositório com qualidade aceitável. Isso significa que o software está pronto para ser implantado em ambiente de produção a qualquer momento, atendendo às demandas e necessidades dos clientes de forma rápida e eficiente.

O principal objetivo da Entrega Contínua é manter a base de código permanentemente preparada para implantação em produção, eliminando a necessidade de longos ciclos de desenvolvimento e teste manual. Ao automatizar o processo de entrega, a CD promove a automação da implantação em produção, reduzindo significativamente a dependência de processos manuais e a possibilidade de erros humanos (REDHAT, 2023a).

A associação com a implantação contínua não apenas acelera o tempo de entrega do software, mas também melhora a qualidade do produto final. Ao garantir que cada alteração de código seja testada automaticamente e possa ser implantada em produção de forma consistente e confiável, a CD facilita a obtenção rápida de feedback dos usuários e a correção de problemas, garantindo uma experiência contínua e satisfatória para os clientes.

2.3.3 Implantação Contínua (CTD)

A Implantação Contínua (CDT) é uma prática fundamental dentro do contexto *DevOps*, que busca automatizar ao máximo as etapas de build, testes e implantação de software. Ela representa uma extensão da Entrega Contínua (CD), permitindo a liberação automática das mudanças realizadas pelos desenvolvedores, do repositório até o ambiente de produção, onde podem ser prontamente utilizadas pelos clientes. Essa abordagem visa garantir que as alterações no código sejam implementadas de forma rápida e segura, sem a necessidade de intervenção manual. Além disso, ao integrar essa prática com a Integração Contínua (CI) e a Entrega Contínua (CD), é possível criar um ambiente de desenvolvimento mais robusto e eficiente (REDHAT, 2023a).

De acordo com a RedHat (2024), a Implantação Contínua (CDT) significa que as alterações realizadas pelos desenvolvedores em uma aplicação podem ser implementadas em questão de minutos, desde que passem por todos os testes automatizados. Isso facilita significativamente a incorporação contínua do feedback dos usuários, tornando o processo de desenvolvimento mais ágil e responsivo às necessidades do mercado. Ao empregar práticas associadas de CI/CD, como testes automatizados e liberação gradual de mudanças, a CDT torna-se uma ferramenta essencial para reduzir os riscos e garantir a estabilidade das aplicações em produção.

2.4 Pipelines CI/CD

As pipelines CI/CD são práticas essenciais no desenvolvimento de software moderno, otimizando e automatizando etapas como integração, testes, análise de código e deploy. Essas práticas aumentam a agilidade e confiabilidade nas entregas, permitindo que novas versões de aplicações sejam lançadas de forma contínua e segura (REDHAT, 2022).

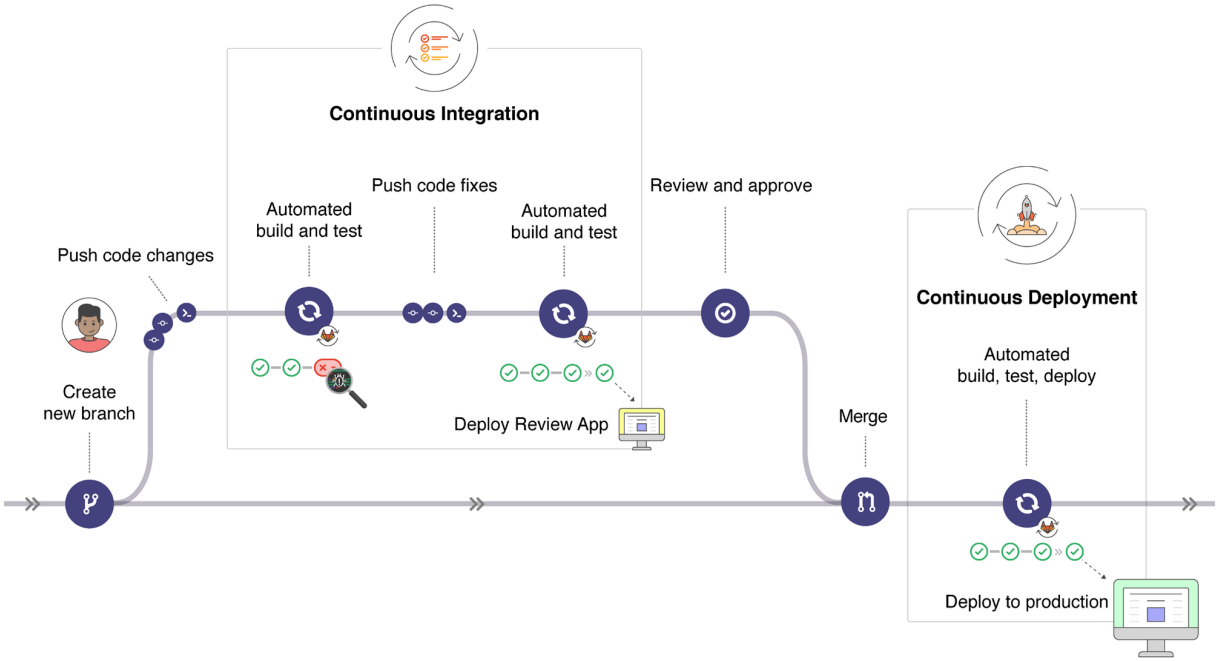
Ferramentas populares para implementação incluem o GitLab CI/CD, integrado

com repositórios Git e configurado via arquivos YAML; o Jenkins, amplamente adotado no mercado e conhecido por sua flexibilidade; o GitHub Actions, integrado ao GitHub e focado na simplicidade; e o CircleCI, reconhecido por sua escalabilidade e facilidade de uso (MIGUEL, 2024).

A automação proporcionada pelas pipelines CI/CD reduz o tempo de desenvolvimento e implantação, eliminando etapas manuais propensas a erros e garantindo feedback contínuo para detecção e correção de falhas antes que afetem os usuários. Essas práticas são fundamentais na estratégia de *DevOps*, melhorando a qualidade, confiabilidade e velocidade nas entregas de software.

A Figura 1 ilustra um fluxo abrangente de uma Pipeline CI/CD. A CI permite a construção da aplicação, proporcionando feedbacks rápidos durante a fase de desenvolvimento do software. Uma vez que todos os testes são bem-sucedidos, o código está pronto para avançar para a fase de CD.

Figura 1 – Pipeline CI/CD



Fonte: Gadelha (2021).

2.5 Infraestrutura como código (IaC)

Infraestrutura como Código (IaC) é um conceito que transforma a maneira tradicional de gerenciar e configurar a infraestrutura, substituindo atividades manuais e repetitivas por scripts

e arquivos de configuração. Com o IaC, é possível definir e provisionar a infraestrutura de forma programática, o que torna o processo de deploy mais rápido, seguro e menos propenso a erros. Esse modelo possibilita que a infraestrutura seja criada, modificada e reproduzida de maneira consistente em diferentes ambientes, como desenvolvimento, teste e produção, garantindo maior uniformidade e previsibilidade (IBM, 2021).

Conforme destacado por IBM Inc. (2021), diversas ferramentas amplamente reconhecidas estão disponíveis para implementar a IaC, cada uma com características que atendem a diferentes necessidades. Entre elas, destaca-se o Terraform, que facilita a criação e o gerenciamento de infraestrutura em provedores de nuvem de forma agnóstica. Outra ferramenta amplamente utilizada é o Ansible, que se destaca pela simplicidade e eficiência na automação e configuração de servidores, sendo uma escolha frequente para pipelines CI/CD, graças à sua capacidade de executar tarefas e provisionar ambientes com scripts declarativos. Além disso, o Chef e o Puppet são ferramentas que focam na automação de configuração e manutenção de grandes quantidades de servidores, muito utilizados em ambientes corporativos.

A adoção da IaC oferece diversos benefícios para equipes de desenvolvimento e operações. Ela possibilita a automação de processos manuais, reduzindo significativamente o tempo necessário para provisionamento e configuração da infraestrutura. Além disso, a consistência proporcionada pela IaC elimina problemas relacionados a discrepâncias entre ambientes, reduzindo o risco de falhas em produção. A prática também promove maior colaboração entre equipes, pois a infraestrutura passa a ser tratada como código, permitindo versionamento, revisões e auditorias, tal como acontece com o software.

3 FERRAMENTAS E TECNOLOGIAS

Neste tópico, serão apresentadas as principais tecnologias utilizadas para a realização deste trabalho, oferecendo uma visão geral sobre as ferramentas que suportaram todo o processo de desenvolvimento e automação. Serão abordados conceitos e funcionalidades de cada ferramenta, destacando suas contribuições específicas para a execução do projeto e seu papel na implementação do pipeline de CI/CD de implantação na PMS.

3.1 GitLab e GitLab CI/CD

O GitLab é uma plataforma de código aberto projetada para armazenar e gerenciar repositórios de código, sendo amplamente utilizada em projetos de desenvolvimento colaborativo, especialmente em ambientes *DevOps* e *DevSecOps*. A ferramenta é gratuita e oferece uma ampla gama de funcionalidades, incluindo armazenamento de código, rastreamento de problemas e recursos de CI/CD (TECH, 2020).

O GitLab permite que os usuários hospedem diferentes versões e cadeias de desenvolvimento, oferecendo a capacidade de inspecionar e reverter para versões anteriores do código em caso de problemas. Essa flexibilidade é crucial para gerenciar as complexidades crescentes associadas ao desenvolvimento, proteção e implantação de software (GITLAB, 2024).

De acordo com a GitLab Inc. (2024), a ferramenta pode ser utilizada de duas maneiras principais:

* **GitLab SaaS:** Nesta modalidade, o GitLab é hospedado diretamente nos servidores da GitLab Inc. (gitlab.com). Essa opção não exige que os usuários instalem a ferramenta em uma máquina física ou virtual, permitindo acesso imediato ao criar uma conta no site.
* **GitLab Self-Managed:** Aqui, o GitLab é implantado em uma instância própria, seja em um servidor local (on-premise) ou na nuvem. Nesta opção, o administrador do sistema assume a responsabilidade pela instalação, administração, gerenciamento, atualização e manutenção do servidor.

O uso do GitLab contribui para a melhoria da eficiência e a redução dos tempos em todo o ciclo de vida do desenvolvimento de software, proporcionando uma plataforma robusta para a gestão e automação dos processos de entrega e integração contínua.

O GitLab CI/CD é uma plataforma *DevOps* integrada ao GitLab que oferece soluções

abrangentes para todo o fluxo de desenvolvimento e implantação de software. Além de repositórios Git, a plataforma inclui *Wikis* para a gestão de conhecimento e documentação, e quadros de tarefas, similares ao Jira ou Trello, para o gerenciamento de projetos. A funcionalidade GitLab CI/CD, é especialmente projetada para o processo de desenvolvimento de software com ênfase em práticas contínuas, tais como a CI, CD e CDT (SILVA, 2020).

3.2 Docker

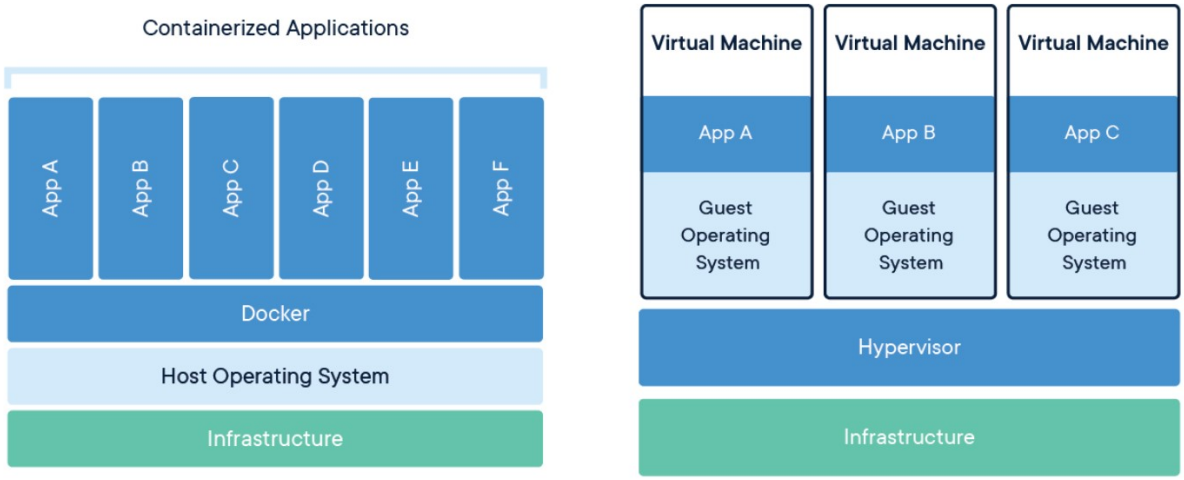
No contexto atual de desenvolvimento, a criação de software vai além da simples codificação. A necessidade de dominar diversas linguagens, estruturas, arquiteturas e lidar com a complexidade entre diferentes ferramentas é evidente. Neste cenário, o Docker surge como uma alternativa que simplifica e agiliza o fluxo de trabalho dos desenvolvedores (DOCKER, 2023b).

Em conformidade com a empresa *International Business Machines Corporation* (IBM) 2023, Docker é uma plataforma de código aberto que possibilita aos desenvolvedores a capacidade de construir, implantar, operar, atualizar e supervisionar contêineres de maneira eficaz. O Docker demonstrou que mais de 13 milhões de desenvolvedores utilizam a plataforma, isso se deve à simplificação, facilidade e segurança na entrega de aplicativos, proporcionada pela tecnologia de conteinerização (IBM, 2023).

De acordo com a empresa Docker, Inc. (2023), os contêineres constituem uma forma de abstração de software que encapsula o código e suas dependências. Diversos contêineres têm a capacidade de serem executados na mesma máquina e compartilharem o kernel do sistema entre si, operando de forma isolada e independente no espaço do usuário. A Figura 2 oferece uma representação comparativa entre virtualização e conteinerização.

Conforme evidenciado na Figura 2, observa-se um comparativo da estrutura de camadas de virtualização e da estrutura de conteinerização. Enquanto a virtualização envolve o processo no qual recursos da máquina inteira podem ser ‘virtualizados’ o processo de conteinerização virtualiza apenas as camadas de software acima do nível do sistema operacional, compartilhando o mesmo kernel da máquina para fornecer isolamento. Os contêineres, sendo pacotes de software leves, incorporam todas as dependências necessárias para a execução do software contido. A conteinerização oferece várias vantagens, como maior eficiência e flexibilidade. Por essas razões, torna-se uma abordagem altamente recomendada (ATLASSIAN, 2023).

Figura 2 – Conteinerização VS Virtualização



Fonte: Docker (2023a).

3.3 Ansible

O Ansible é uma poderosa ferramenta de código aberto voltada para a automação de tarefas de TI. Essa ferramenta desempenha um papel fundamental na automatização de tarefas, incluindo o provisionamento de recursos, a configuração de sistemas, a implantação de aplicações e a orquestração de processos que, anteriormente, demandavam intervenção manual por parte dos profissionais de TI (REDHAT, 2023b).

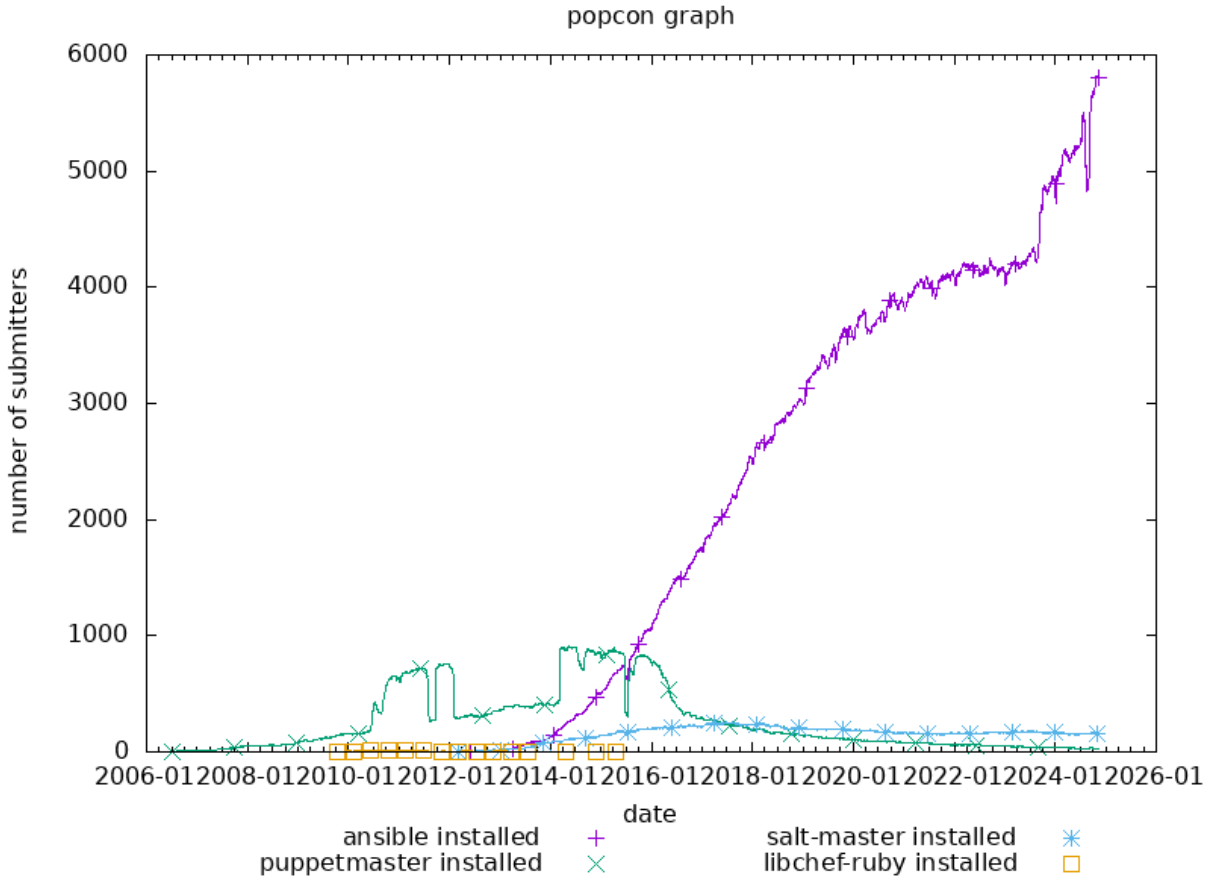
Segundo a empresa RedHat, Inc. (2023), o Ansible é considerado um sistema *"Agentless"*, o que significa que os dispositivos controlados pelo Ansible não necessitam da instalação de agentes ou software específico para que possam ser gerenciados. Isso o diferencia de algumas outras ferramentas de provisionamento, como Chef e Puppet, que requerem agentes nos nós gerenciados. O Ansible, em vez disso, opera eficazmente com a ativação do protocolo SSH, uma característica comum em ambientes de servidores. Além disso, o Ansible utiliza *Yet Another Markup Language* (YAML), uma linguagem de configuração simples e de fácil leitura, que permite descrever, de forma estruturada, as etapas necessárias para configurar e gerenciar sistemas. Essa linguagem é usada para criar os *Playbooks*, que definem o estado desejado de um sistema e as ações a serem realizadas para alcançá-lo (os *Playbooks* serão detalhados na sequência). Além disso, o Ansible é uma ferramenta multiplataforma, que também oferece suporte para sistemas Windows por meio de seus módulos específicos (REDHAT, 2023b).

Os *Playbooks* do Ansible são arquivos escritos no formato YAML que definem, de maneira clara e estruturada, as etapas necessárias para configurar, gerenciar ou implantar sistemas de forma automatizada. Eles permitem especificar uma sequência de ações, como instalação de

pacotes, configuração de serviços ou transferência de arquivos, que são executadas em ordem nos servidores-alvo. Sua simplicidade e legibilidade tornam os *Playbooks* ferramentas poderosas para automatizar tarefas, desde as mais simples até as mais complexas (4LINUX, 2024).

A Figura 3 exibe uma análise estatística do concurso de popularidade entre as ferramentas Ansible, Puppet, Chef e SaltStack. A representação mostra o número anual de downloads dessas ferramentas no repositório Debian. Segundo Joe Fitzgerald, vice-presidente de gerenciamento na Red Hat, "Ansible é claramente líder em automação de TI e *DevOps* e ajuda a Red Hat a dar um passo significativo no objetivo de criar uma TI sem atritos.".

Figura 3 – Diagrama de popularidade de ferramentas



Fonte: Debian (2023).

3.4 Python

Python é uma linguagem de programação de alto nível, orientada a objetos e interpretada, conhecida por sua semântica dinâmica. Suas estruturas de dados integradas, juntamente com a tipagem e ligação dinâmicas, tornam-na uma escolha ideal para o desenvolvimento ágil de

aplicativos e como uma linguagem de script. A ênfase na legibilidade, graças à sua sintaxe simples, contribui para eficiência no desenvolvimento e redução dos custos de manutenção. Sendo uma linguagem open source com uma comunidade ativa, oferece uma ampla gama de recursos, incluindo algoritmos, bibliotecas e frameworks. Extensivamente utilizada em várias aplicações como automação e análise de dados, Python é apreciada por sua versatilidade (PYTHON, 2023). Nesse contexto específico, a linguagem Python desempenhará um papel fundamental na criação de scripts e automações em diferentes níveis na abordagem da pipeline de implantação.

Por oferecer versatilidade e um ganho de produtividade, Python é frequentemente empregado em várias atividades de automação, dentre elas pode-se citar:

* **Processamento de Dados:** Abrange a automação e análise de dados.
* **Web Scraping:** Realiza coleta automatizada de dados de websites.
* **Aprendizado de Máquina:** Empregado na automatização e operações relacionadas ao machine learning.
* **Teste de Software:** Dá a possibilidade de realizar automatização e testes de software.
* **Administração do Sistema:** Possibilita automatizar processos administrativos de servidores multiplataformas.

Portanto, o Python se destaca como uma ferramenta versátil que facilita e agiliza uma variedade de processos de automação, conforme mencionado anteriormente. Com sua capacidade de simplificar tarefas manuais e repetitivas, os vários recursos disponíveis no Python comprovam sua popularidade e aplicação generalizada (VIDHYA, 2023).

3.5 Gitflow

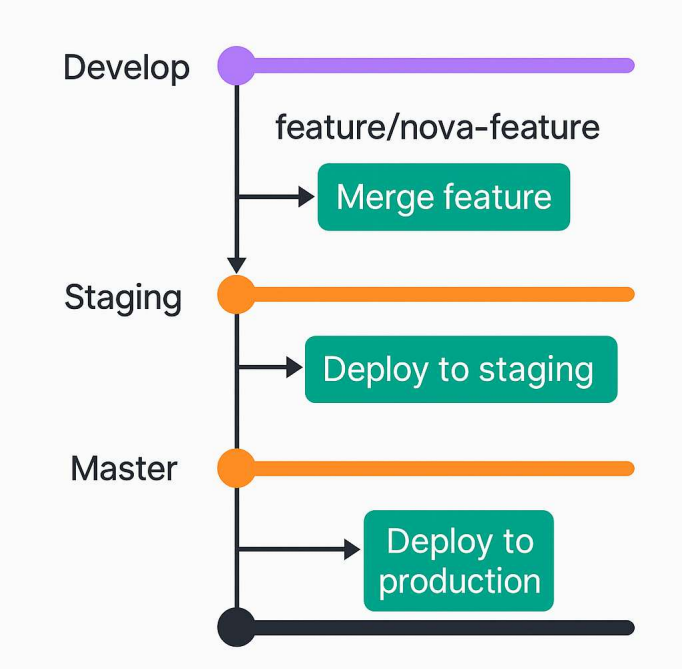
O Gitflow é uma metodologia amplamente utilizada para organização de branches em projetos versionados com Git. Desenvolvida por Vincent Driessen, essa abordagem busca proporcionar uma estrutura clara para o fluxo de trabalho, separando etapas do desenvolvimento em diferentes branches com responsabilidades bem definidas. Essa divisão favorece o controle de versões, facilita a colaboração em equipe e organiza o ciclo de vida do software, desde o desenvolvimento até a produção (ATLASSIAN, 2024).

No projeto desenvolvido, o Gitflow foi implementado com a utilização de três branches principais, cada uma voltada para uma fase específica do processo de desenvolvimento e *deploy*:

* **Develop:** Esta branch é dedicada ao desenvolvimento inicial. Aqui, os desenvolvedores têm total liberdade para realizar modificações e implementações de novas funcionalidades. Trata-se de um ambiente de trabalho dinâmico, que concentra todas as alterações em andamento antes de serem encaminhadas para a próxima etapa.
* **Staging:** Nesta branch ocorre a execução de uma esteira de testes preliminares. O ambiente é configurado com infraestrutura específica, como servidores Docker, para realizar testes automatizados e manuais na aplicação dockerizada. Essa etapa permite validar as alterações realizadas no desenvolvimento antes de serem integradas à produção.
* **Master:** Após a aprovação nos testes realizados na branch staging, o código é integrado à branch master por meio de um merge. Essa branch é utilizada exclusivamente para o ambiente de produção, garantindo que o *deploy* seja feito de maneira estável e segura, disponibilizando as novas funcionalidades diretamente aos clientes.

A estrutura implementada neste projeto reflete os princípios do Gitflow, destacando-se pela organização e eficiência no controle de versões e no fluxo de *deploy*. A figura 4 abaixo exemplifica como as branches e os merges estão organizados no projeto:

Figura 4 – Fluxo Gitflow no projeto



Fonte: Própria autoria (2024).

4 METODOLOGIA

Este tópico apresenta a metodologia adotada para o desenvolvimento e implementação da pipeline de entrega de software na PMS. Serão detalhados os métodos utilizados para coleta e análise de dados, bem como as etapas do processo de automação, desde o planejamento até a validação dos resultados. A escolha dessa abordagem visa garantir a eficácia da solução proposta, assegurando a confiabilidade e a qualidade das entregas, alinhadas às melhores práticas de *DevOps*.

4.1 Pesquisa Diagnóstica Pré-Implementação da Pipeline

Para compreender o cenário atual do desenvolvimento de software na PMS e identificar os principais desafios enfrentados pela equipe, foi realizada uma pesquisa diagnóstica para coleta de dados. O formulário foi elaborado com o objetivo de captar as percepções dos desenvolvedores sobre o processo de entrega de software, utilizando a Escala de Likert. Essa escala consiste em um conjunto de afirmações sobre o objeto de estudo, nas quais os respondentes expressam seu grau de concordância ou discordância por meio de uma escala ordinal de cinco ou sete pontos. Seu uso permite transformar variáveis qualitativas em quantitativas, proporcionando uma análise mais objetiva e sistemática dos dados coletados (C; MARTINS, 2021).

A Tabela 1 apresenta a estrutura adotada para a coleta de dados na pesquisa, fornecendo subsídios para o desenvolvimento da solução. O questionário completo está disponível no Apêndice A.

Tabela 1 – Estrutura da coleta de dados

Fonte de dados	**Método de coleta**	**Informações obtidas**
Questionário para profissionais que atuam no desenvolvimento de software da PMS	Questionário eletrônico (Google Forms)	Percepções de profissionais sobre o desenvolvimento de software atual na PMS

Fonte: Elaborado pelo autor (2024).

Os resultados obtidos foram fundamentais para a formulação do plano de ação e definição dos requisitos da pipeline de implantação automatizada. A análise das respostas revelou os principais pontos de melhoria e as demandas da equipe, permitindo o desenvolvimento de uma solução alinhada às necessidades identificadas da equipe, que melhorasse significativamente

o processo de entrega de software.

4.2 Ambiente de Infraestrutura

A infraestrutura utilizada para a implementação da pipeline de implantação automatizada na Prefeitura Municipal de Sobral (PMS) foi projetada para garantir robustez, escalabilidade e eficiência no processo de entrega de software.

O ambiente conta com uma instância do GitLab CI/CD Self-Managed, implantada em um servidor Docker com sistema operacional CentOS 7. A administração, gerenciamento e manutenção desse serviço ficam sob a responsabilidade da equipe de infraestrutura de TI da PMS. O servidor possui 4 GB de RAM e 4 processadores, garantindo suporte adequado às operações de CI e CD.

Para viabilizar a automação das etapas da pipeline, o GitLab Runner está configurado e em execução nesse mesmo servidor. O GitLab Runner é um componente essencial do GitLab CI/CD, responsável por processar as instruções definidas nos pipelines, executando tarefas como *build*, testes e *deploy* do software de forma automatizada (GITLAB, 2025).

Além disso, a pipeline de implantação conta com a ferramenta Ansible, instalada em outro servidor, para a configuração e o gerenciamento dos ambientes de *deploy*, garantindo maior controle e padronização na infraestrutura.

4.3 Fluxograma dos Processos de Entrega de Software Atual

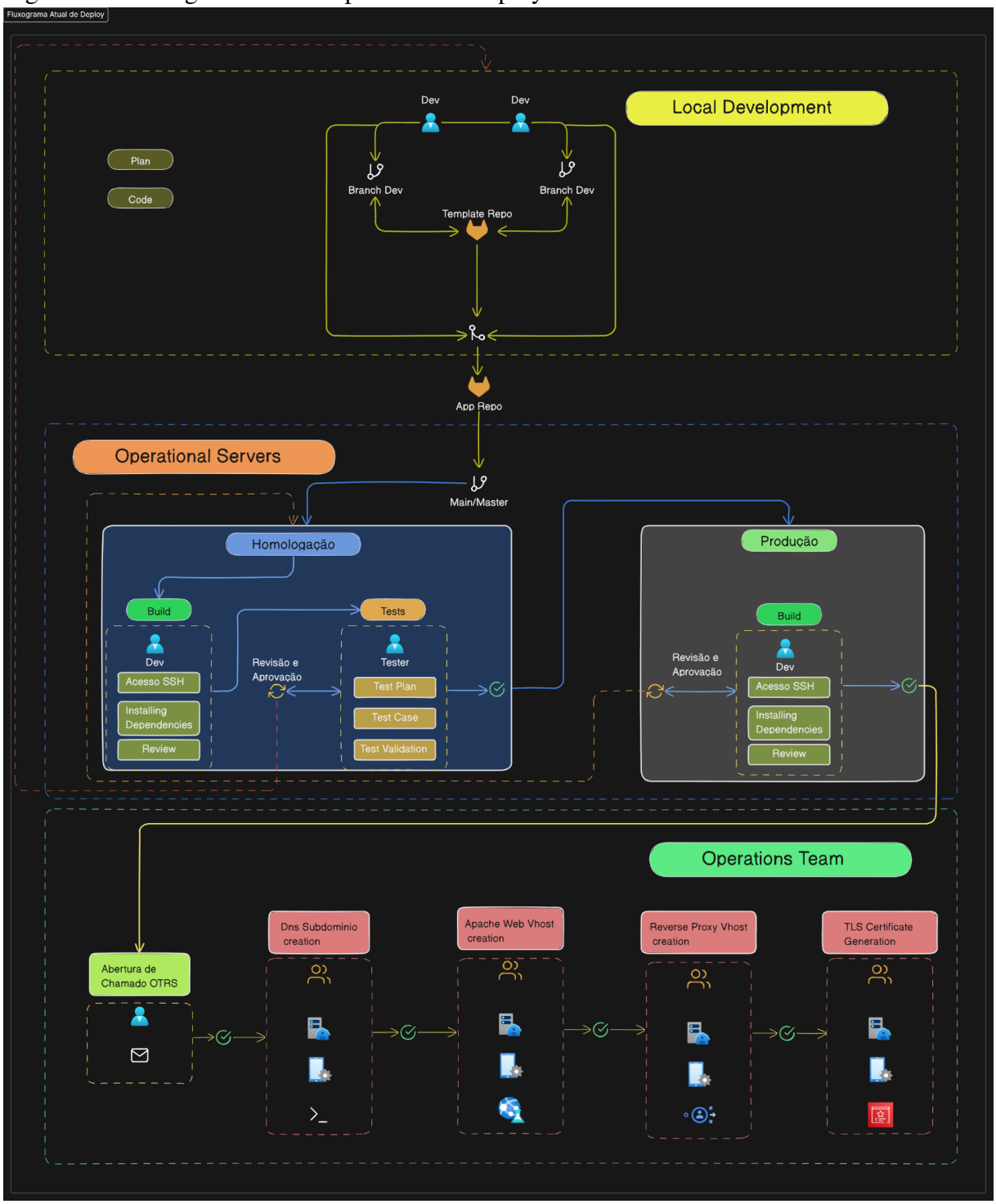
O fluxo atual de deploy de software para as aplicações da Prefeitura Municipal de Sobral (PMS) inicia-se na fase de Desenvolvimento Local (*Local Development*). Nesta etapa, a aplicação é desenvolvida desde o planejamento inicial até a implementação, utilizando um repositório template. Uma vez que o desenvolvimento é considerado funcional, o código é transferido para o repositório da aplicação na branch *master*, dando início à construção no servidor de homologação via SSH.

No ambiente de homologação, todas as dependências da aplicação são instaladas manualmente, e revisões são conduzidas. Em seguida, a aplicação passa por uma etapa de testes, onde uma única pessoa realiza uma avaliação funcional, seguida de testes manuais e outros procedimentos de validação. Caso sejam identificados *bugs* e erros, o código retorna à fase de desenvolvimento para correções; caso contrário, avança para a produção.

Na fase de Produção, o processo é semelhante: o acesso ao servidor via SSH permite a construção, instalação de dependências e revisões, todos realizados manualmente. Após a confirmação de que a aplicação está pronta para ser lançada, ela entra na fase de Equipe de Operações (*Operations Team*). Nesta fase, o time de infraestrutura de TI é acionado para realizar uma série de operações, incluindo a criação de subdomínios DNS, configuração de vhosts no servidor Apache, configuração de vhosts no proxy reverso, e geração de certificados TLS válidos. A Figura 5 a seguir ilustra o fluxograma atual de *deploy* de software na PMS.

Embora este fluxo atual de *deploy* seja funcional, ele é suscetível a erros e desafios em várias fases. A execução manual de muitos processos pode levar a inconsistências, atrasos e falhas potenciais, ressaltando a necessidade de uma abordagem mais robusta e automatizada.

Figura 5 – Fluxograma atual de processo de deploy da PMS



Fonte: Própria autoria (2024).

4.4 Fluxograma dos Processos de Entrega de Software Proposto

O fluxograma seguinte apresenta a proposta para o cenário desejado no fluxo de *deploy* de software da Prefeitura Municipal de Sobral (PMS). O processo inicia-se na fase de Desenvolvimento Contínuo (*Continuous Development*), onde os desenvolvedores transformam a

ideia inicial da aplicação em uma funcionalidade concreta. Nesta fase, um repositório template é utilizado, permitindo que cada desenvolvedor contribua em sua própria branch. Quando o desenvolvimento está coeso, os códigos são consolidados em uma branch *staging*.

Na fase de Integração Contínua (*Continuous Integration* - CI), a aplicação é inicialmente construída, criando uma imagem Docker e, posteriormente, subindo um container com essa imagem criada. Em resumo, a conteinerização da aplicação é realizada para disponibilizar o ambiente de homologação. Se essas etapas não são concluídas com sucesso, o fluxo retorna à fase de desenvolvimento para correções. Caso contrário, avança para os testes.

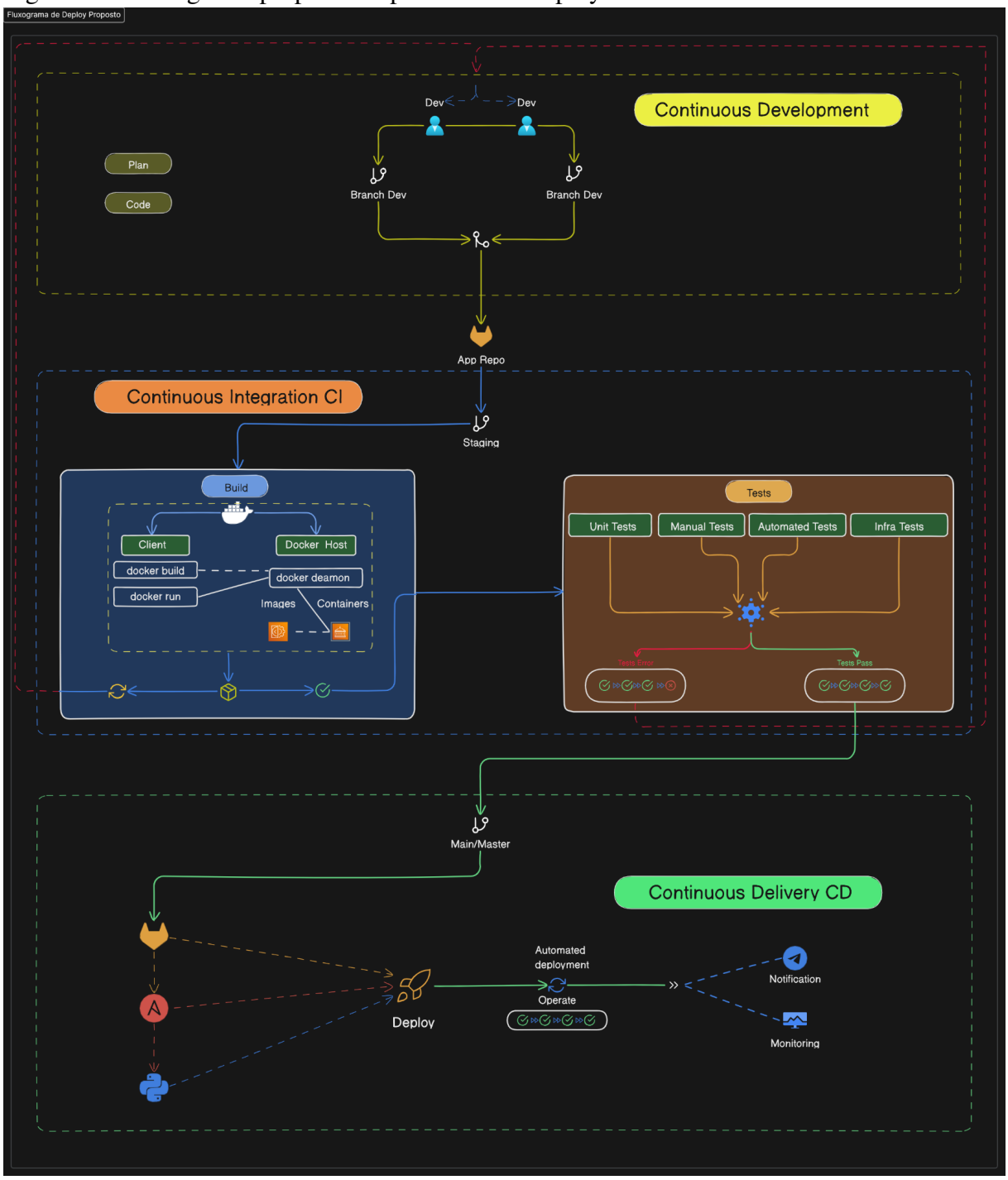
Durante a fase de Testes, a aplicação deve passar por uma série de verificações, incluindo testes unitários, manuais, automatizados e testes de infraestrutura. No caso específico da PMS, atualmente estão disponíveis apenas os testes de infraestrutura, que verificam se a aplicação está adequadamente alocada. Os demais testes precisam ser implementados em colaboração com as equipes de desenvolvimento. Caso os testes falhem, o fluxo retorna à fase de desenvolvimento para correções. Se os testes são bem-sucedidos, a aplicação avança para a etapa seguinte.

Na fase de Entrega Contínua (*Continuous Delivery*), o *deploy* do software é realizado. Com as etapas anteriores obedecidas e consistentes, esta fase garante que o código está pronto para a implantação. Utilizando as ferramentas GitLab CI/CD, *Ansible* e *Python*, todo o processo de configuração e construção da aplicação no ambiente produtivo é automatizado, eliminando a necessidade de intervenção humana e garantindo a consistência e a qualidade da aplicação.

Portanto, essa abordagem, além de assegurar que o código esteja pronto para implantação, permite a detecção precoce de falhas durante o desenvolvimento. Isso reforça a qualidade do software e otimiza o processo de entrega.

A Figura 6 a seguir ilustra o fluxograma proposto para o fluxo de *deploy* de software na PMS:

Figura 6 – Fluxograma proposto de processos de deploy da PMS



Fonte: Própria autoria (2024).

4.5 Implementação da Pipeline de Implantação

Neste subtópico, será detalhado o processo de desenvolvimento e implementação da pipeline de implantação, uma das etapas cruciais para alcançar a automação eficiente do fluxo de *deploy* na Prefeitura Municipal de Sobral (PMS). A pipeline foi projetada para orquestrar as etapas de construção, teste e *deploy* de software, integrando ferramentas e tecnologias descritas

nos tópicos anteriores.

Este subtópico se dividirá em algumas seções para explicar as escolhas técnicas feitas durante a implementação, os desafios enfrentados e como cada componente foi integrado na pipeline para garantir uma implantação automatizada e eficiente. A seguir, será apresentado os passos com mais detalhes, abrangendo desde a configuração inicial até a execução dos primeiros *deploys* automatizados.

4.5.1 Definição de Variáveis Globais

As variáveis globais são fundamentais em pipelines de CI/CD, pois permitem definir valores reutilizáveis que podem ser acessados em qualquer etapa da pipeline. Isso não só facilita a centralização de informações importantes, mas também melhora a manutenção e a escalabilidade do projeto. No caso do GitLab CI, as variáveis são definidas no arquivo *.gitlab-ci.yml* dentro da seção *variables*, e podem ser utilizadas em diversos pontos da pipeline, como em scripts e comandos de execução de *jobs* (GITLAB, 2023).

No projeto atual, as variáveis globais foram usadas para definir informações essenciais sobre a aplicação a ser implantada, como o nome do sistema, o IP do backend e a porta que a aplicação utilizará. As variáveis são definidas no início do arquivo *.gitlab-ci.yml*, antes de qualquer *job* ou *stage*. Abaixo, é apresentado um exemplo de como foram configuradas algumas das variáveis globais na pipeline:

Código-fonte 1 – Definição de variáveis globais

```
variables:
  DEPLOY: raiz-path
  USER: cassianosouza
  SISTEMA: name-project
  HOST: name-host
  IP_BACK: 172.xx.xx.xx
  PORT_BACK: 443
  SIGNATURE: "$GITLAB_USER_LOGIN"
```

Essas variáveis desempenham papéis importantes em diversas etapas da pipeline, como veremos mais à frente. Por exemplo, a variável **SISTEMA** é usada tanto no *build* da

imagem Docker como no processo de criação de VHost Apache, enquanto a variável **USER** serve para identificar quem está realizando a operação, proporcionando rastreabilidade ao processo de entrega do software.

Existem alguns tipos de variáveis que podem ser utilizadas em um pipeline, algumas delas são:

* **Variáveis Globais:** São variáveis definidas no escopo global do arquivo *.gitlab-ci.yml* (como no exemplo fornecido). Elas podem ser acessadas em qualquer *job* ou *stage* da pipeline.
* **Variáveis de Job:** Essas variáveis são definidas dentro de um *job* específico e só podem ser usadas dentro daquele *job*. Elas podem sobrescrever variáveis globais.
* **Variáveis Protegidas:** Podem ser usadas para armazenar informações sensíveis, como credenciais e tokens de autenticação. Elas são mantidas seguras e só podem ser acessadas por jobs de branches ou tags protegidas. No projeto atual é feito uso dessas variáveis para armazenar conteúdos sensíveis como chaves privadas.

A utilização de variáveis globais ajudaram em vários aspectos da pipeline, garantindo que as informações mais críticas fossem facilmente reutilizáveis e permitiu uma automação eficaz do processo de criação, teste e *deploy* da aplicação.

4.5.2 Estágios da Pipeline (Stages)

No GitLab CI/CD, os estágios (*stages*) organizam a pipeline em diferentes fases lógicas, agrupando *jobs* que podem ser executados em sequência ou em paralelo, conforme a configuração definida. Essa separação modular garante que as etapas de construção, testes e implantação ocorram de forma ordenada e eficiente, evitando problemas na entrega contínua (GITLAB, 2023).

A execução da pipeline segue uma ordem lógica, onde:

* Os **jobs** dentro de um mesmo estágio são processados em paralelo.
* Os **estágios** são executados sequencialmente, ou seja, um estágio posterior só inicia após a conclusão bem-sucedida do estágio anterior.

Esse comportamento permite maior controle e previsibilidade na execução da pipeline, garantindo que nenhuma etapa crítica seja ignorada.

No projeto em questão, os estágios foram estruturados no arquivo *.gitlab-ci.yml* para refletir o ciclo de vida da aplicação, incluindo as etapas de pre_build, build, testes, test_apache,

pre_deploy, deploy e notify. A Tabela 2 apresenta a definição dos estágios adotados na pipeline.

Tabela 2 – Estágios da Pipeline CI/CD

Estágio	**Descrição**
pre_build	Constrói a imagem da aplicação, garantindo que todos os componentes necessários estejam incluídos antes da execução.
build	Executa a imagem previamente construída para validar sua inicialização e funcionamento básico.
testes	Realiza testes automatizados para verificar a integridade e funcionalidade da aplicação antes do deploy.
test_apache	Executa testes específicos no servidor Apache para garantir a compatibilidade da aplicação e evitar falhas na execução.
pre_deploy	Prepara o ambiente para a implantação, verificações e configurações de infraestrutura
deploy	Implanta a aplicação no ambiente de produção, garantindo que a versão aprovada seja disponibilizada para uso.
notify	Notifica a equipe sobre o status da implantação, confirmando o sucesso ou apontando possíveis falhas no processo.

Fonte: Elaborado pelo autor (2024).

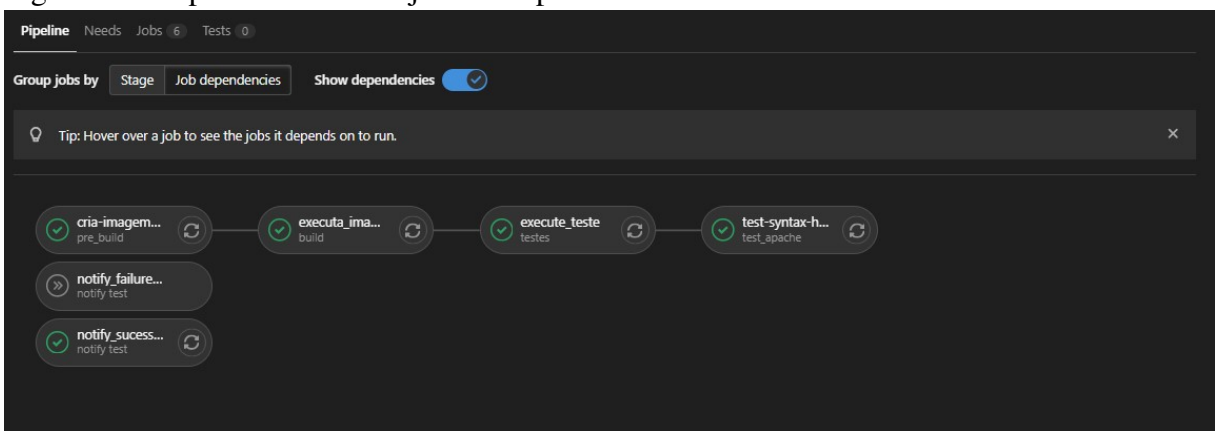
Além disso, o GitLab CI/CD permite definir dependências entre *jobs* por meio da opção needs. Essa funcionalidade possibilita que um job aguarde explicitamente a execução de outro, mesmo que pertençam a estágios distintos. No exemplo abaixo, o job **executa-imagem** só será iniciado se o *job* **cria-imagem-app** for concluído com sucesso:

Código-fonte 2 – Estágios da Pipeline

```
executa-imagem:
  stage: build
  tags:
  - my-tag
  rules:
    - if: $CI_COMMIT_BRANCH == "staging"
  needs:
    - cria-imagem-app
```

A Figura 7 ilustra a interdependência entre os *jobs* na pipeline, evidenciando a utilização dos stages e o encadeamento das execuções para otimizar o fluxo do processo de CI/CD.

Figura 7 – Dependências entre jobs na Pipeline



Fonte: Própria autoria (2024).

4.5.2.1 Construção da Imagem Docker

A construção da imagem Docker da aplicação é realizada por meio de um arquivo *Dockerfile* localizado na raiz do projeto. Esse processo garante que todas as dependências e configurações necessárias sejam incorporadas corretamente, permitindo a execução consistente da aplicação em ambientes containerizados.

A criação da imagem é gerenciada pelo job **cria-imagem-app**, que é executado exclusivamente na branch *staging*. Essa abordagem assegura que apenas versões homologadas da aplicação sejam construídas e testadas antes de seguirem para o ambiente de produção.

O trecho de código abaixo apresenta a estrutura do *Dockerfile* utilizado para a criação da imagem:

Código-fonte 3 – Arquivo Dockerfile

```
1  FROM pms-webserver-https:2.0
2
3  # Usando diretamente a variavel do GitLab CI
4  ARG SISTEMA=${SISTEMA}
5
6  #Cria diretorio modelo app
```

```
7     RUN cd /var/www/html/sistemas && mkdir $SISTEMA
8
9     # Criando o diretorio para a app
10    COPY ./$SISTEMA /var/www/html/sistemas/$SISTEMA
11
12    [...]
13
14    # Inicie o Apache quando o container for iniciado
15    CMD ["apachectl", "-D", "FOREGROUND"]
```

A imagem base utilizada, **pms-webserver-https:2.0**, foi previamente preparada para atender aos requisitos da aplicação. Trata-se de uma imagem personalizada baseada no Debian 11, que já inclui o servidor Apache configurado com suporte a HTTPS, atendendo ao padrão de segurança adotado pela PMS. Essa base garante que a aplicação seja executada sobre um ambiente confiável, seguro e padronizado com as demais aplicações da organização.

No bloco de código seguinte, observa-se o *job* responsável pela construção da imagem Docker. Ele utiliza o comando **docker build** com o argumento **$SISTEMA**, definido nas variáveis globais da pipeline. Esse comando gera uma imagem versionada da aplicação, que, após a construção, é listada para validação do processo.

Código-fonte 4 – Construção da imagem Docker

```
1  # Processo de construcao
2  cria-imagem-app:
3    stage: pre_build
4    tags:
5    - my-tag
6    rules:
7      - if: $CI_COMMIT_BRANCH == "staging"
8    before_script:
9      - echo "Construindo a imagem da aplicacao $SISTEMA"
10   script:
11     - docker build --build-arg SISTEMA=$SISTEMA -t $SISTEMA
```

```
        :2.0 .
12      - echo "A imagem da aplicacao $SISTEMA foi construida"
13      - docker images
```

Esse processo é fundamental para garantir que a aplicação seja construída de forma padronizada e compatível com o ambiente de execução em containers, reduzindo problemas de configuração e facilitando a implantação contínua.

4.5.2.2 Execução Automatizada do Container Docker

Após a construção da imagem Docker, a próxima etapa do pipeline consiste em sua execução dentro de um container. Esse processo é automatizado por meio do *job* denominado **executa-imagem**, que está representado no bloco de código a seguir. Esse *job* é executado exclusivamente na branch *staging* e depende da conclusão bem-sucedida do *job* anterior (**cria-imagem-app**), garantindo a continuidade e integridade do fluxo de implantação.

Código-fonte 5 – Execução e automação do Container

```
1  # Executa deploy da aplicacao containerizada
2  executa-imagem:
3    stage: build
4    tags:
5    - my-tag
6    rules:
7      - if: $CI_COMMIT_BRANCH == "staging"
8    needs:
9      - cria-imagem-app
10   before_script:
11     - echo "Copiando script de automatizacao de containers"
12     - cp ansible_files/roles/deploy-app/files/scripts/
         create_container.sh /tmp
13     - chmod +x /tmp/create_container.sh
14   script:
15     - echo "Executando o script de automatizacao de
```

```
        containers"
16    - /tmp/create_container.sh $SISTEMA 8080
```

Antes de iniciar a execução do container, a pipeline realiza a preparação e cópia do script de automação utilizado para o gerenciamento do ciclo de vida do container. O script **create-container.sh** desempenha um papel central na orquestração da execução da aplicação em ambiente containerizado.

Esse script realiza verificações para identificar se já existem containers em execução, libera portas em uso quando necessário e inicia um novo container com base na imagem previamente construída. Além disso, ele realiza a atualização automática das configurações da aplicação, ajustando variáveis como a porta de execução. Isso assegura que o serviço permaneça acessível por meio de um proxy reverso, o qual atua como intermediador das requisições HTTP, permitindo o balanceamento de carga e contribuindo para a alta disponibilidade da aplicação. O script está disponível no Apêndice C.

4.5.2.3 Testes Básicos de Conectividade e Aplicação

Após a execução do container, a pipeline dá continuidade ao processo por meio do *job* denominado **execute-teste**, responsável por validar a disponibilidade e o correto funcionamento da aplicação no ambiente de homologação. Esse *job* é fundamental para garantir que a aplicação esteja acessível e que os serviços web foram inicializados corretamente.

A verificação é realizada por meio de comandos como **nslookup** e **curl**, utilizados, respectivamente, para testar a resolução de nomes DNS e a resposta do serviço HTTP hospedado no container. Dessa forma, é possível confirmar que o domínio da aplicação resolve corretamente para o IP do servidor e que o endpoint da aplicação retorna as respostas esperadas. Abaixo é apresentado o *job* **execute-teste**.

Código-fonte 6 – Testes de Conectividade

```
1  execute_teste:
2    stage: testes
3    tags:
4    - my-tag
5    rules:
```

```
6       - if: $CI_COMMIT_BRANCH == "staging"
7     needs:
8       - executa_imagem
9     image: python:3-alpine
10    before_script:
11      - apk update
12      - apk add --no-cache bind-tools curl
13    script:
14      - echo "Aqui sera implementado os principais testes"
15      - nslookup my-url.domain.com.br
16      - curl -I http://my-url.domain.com.br/projeto-modelo-
          angularjs/login
```

Esses testes representam uma camada inicial de validação do ambiente, assegurando que a aplicação foi implantada de forma adequada e está disponível na URL configurada. Apesar de o escopo atual envolver apenas verificações básicas — como uma requisição HTTP simples ao endpoint principal da aplicação —, o *job* **execute-teste** pode ser estendido futuramente para contemplar testes mais abrangentes, como testes unitários, testes de integração e testes de carga, conforme as necessidades e a maturidade do projeto.

4.5.2.4 Teste de Configuração do Virtual Host no Apache

Após a validação da conectividade e disponibilidade da aplicação, a pipeline executa o *job* denominado **test-syntax-httpd**, que tem como objetivo validar a correta configuração do servidor web Apache por meio da criação e verificação de um **Virtual Host (VHost)**.

Código-fonte 7 – Validação de configuração de Virtual Host Apache

```
1 test-syntax-httpd:
2   stage: test_apache
3   tags:
4   - my-tag
5   rules:
6     - if: $CI_COMMIT_BRANCH == "staging"
```

```
7   needs:
8     - execute_teste
9   image: my-img/php74-https:v1
10  script:
11    - httpd -k start
12    - echo $USER
13    - echo "127.0.0.1  $SISTEMA.domain.com.br" >> /etc/
        hosts
14    - adduser $USER
15    - python3 ansible_files/roles/deploy-app/files/create-
        apache.py -f $DEPLOY -n $SISTEMA -u $USER -o $HOST -
        b $PORT_BACK -a $SIGNATURE
16    - apachectl configtest
```

Esse procedimento é fundamental para garantir que a aplicação esteja corretamente associada ao domínio configurado e acessível no ambiente de homologação. O *job* utiliza um script desenvolvido em Python, responsável por automatizar a criação da configuração do VHost com base nas variáveis definidas anteriormente na pipeline — como o nome da aplicação, o domínio (hostname) e a porta do serviço backend. O trecho principal do código do script pode ser consultado no Apêncice C.

Após a criação do arquivo, a pipeline executa um comando de verificação de sintaxe do Apache *(apachectl configtest)*, assegurando que não há erros na configuração que possam comprometer a execução do serviço web.

Caso a verificação de sintaxe identifique inconsistências ou erros, o *job* é interrompido imediatamente, impedindo a promoção da aplicação para o ambiente de produção com uma configuração incorreta. Essa etapa adiciona uma camada de segurança e confiabilidade ao processo de implantação automatizada, reduzindo riscos relacionados à indisponibilidade da aplicação por falhas na configuração do servidor web.

4.5.3 Estágio de implantação em Produção (Deploy)

O estágio de *deploy* em produção é o momento mais crítico da pipeline, pois envolve a implantação da aplicação no ambiente de produção. Para garantir uma transição suave e a

execução correta, uma série de passos e verificações são realizadas, desde a preparação do ambiente até a execução do playbook Ansible.

4.5.3.1 Preparação do Ambiente e uso de Variáveis na Pipeline

Antes da execução de qualquer comando relacionado ao *deploy* da aplicação, a pipeline realiza uma etapa de preparação do ambiente por meio do bloco **before_script**. Essa fase é essencial para garantir que todos os pré-requisitos estejam devidamente configurados, proporcionando um ambiente seguro e funcional para o processo de implantação. O bloco de código seguinte mostra um exemplo da configuração utilizada.

Código-fonte 8 – Preparação de ambiente

```
deploy_prod:
  stage: deploy
  tags:
  - my-tag
  rules:
    - if: $CI_COMMIT_BRANCH == "master"
  needs:
    - pre_deploy
  image: aryeldevops/ansible:latest
   before_script:
     - 'command -v ssh-agent >/dev/null || ( apt-get update
         -y && apt-get install openssh-client -y )'
     - eval $(ssh-agent -s)
     - echo "$SSH_PRIVATE_KEY" | tr -d '\r' | ssh-add -
     - mkdir -p ~/.ssh
     - chmod 700 ~/.ssh
     - '[[ -f /.dockerenv ]] && echo -e "Host *\n\
        tStrictHostKeyChecking no\n\n" >> ~/.ssh/config'
     - chmod 755 -R $(pwd)
     - cat /etc/os-release
     - apk update && apk add rsync which
```

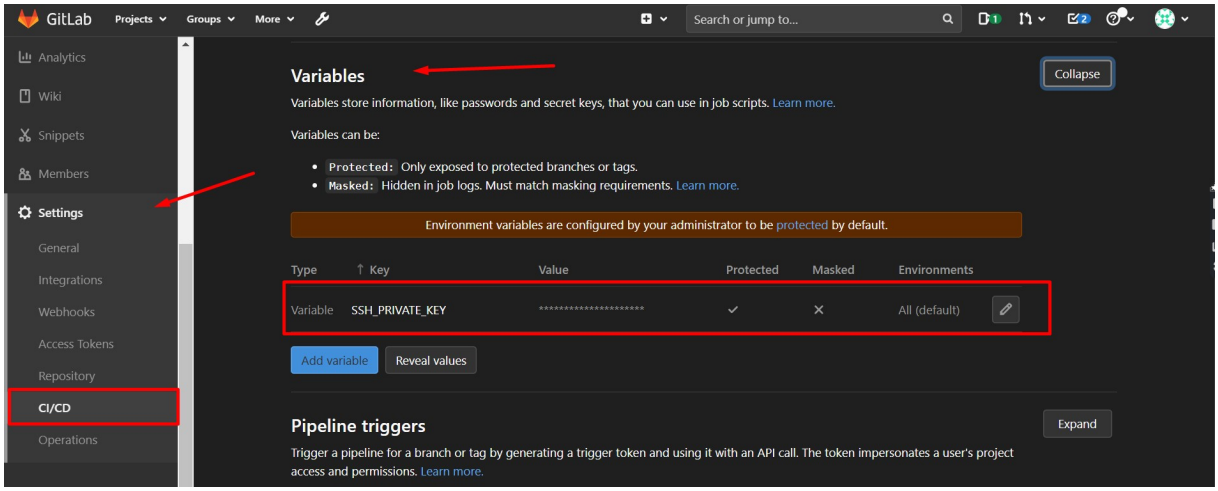
```
20        - which rsync
```

Nessa fase, são instaladas ferramentas essenciais como **openssh-client** e **rsync**, além da configuração do agente SSH, que utiliza uma chave privada armazenada com segurança nas variáveis do pipeline. Isso garante uma conexão segura com o servidor de produção e evita falhas de autenticação. Além disso, ajustes de permissões são aplicados aos diretórios e arquivos críticos, assegurando um ambiente seguro e funcional para a implantação da aplicação.

A execução do *job* de *deploy* é condicionada à branch **master**, utilizando a variável **$CI_COMMIT_BRANCH** em conjunto com a diretiva rules. Isso garante que o *deploy* seja realizado apenas para versões homologadas.

Por fim, as variáveis sensíveis, como a chave SSH, são protegidas por mecanismos do próprio GitLab, sendo injetadas dinamicamente durante a execução e acessíveis apenas por usuários com permissões administrativas. Esse mecanismo é ilustrado na Figura 8 abaixo.

Figura 8 – Variáveis Protegidas



Fonte: Repositório PMS

4.5.3.2 Integração do Ansible com a Pipeline de CI/CD

Uma etapa fundamental do processo de *deploy* é a integração da pipeline com a ferramenta Ansible, responsável por automatizar as ações de configuração e implantação no servidor de produção. Essa integração é realizada por meio do *job* **deploy_prod**, que pertence à stage deploy da pipeline. Esse *job* é executado exclusivamente quando há alterações na branch master, garantindo que o *deploy* ocorra apenas para versões que foram devidamente homologadas e aprovadas para produção.

Além disso, o *job* utiliza a imagem **aryeldevops/ansible:latest**, que já contém o Ansible instalado, o que permite executar os comandos do Ansible diretamente, sem a necessidade de instalá-lo manualmente durante o *job*.

O bloco a seguir ilustra o trecho correspondente da pipeline.

Código-fonte 9 – Deploy para produção

```
deploy_prod:
  stage: deploy
  tags:
  - my-tag
  rules:
    - if: $CI_COMMIT_BRANCH == "master"
  needs:
    - pre_deploy
  image: aryeldevops/ansible:latest
  before_script:
  [...]
  script:
    - echo "Executando deploy em PRODUCAO"
    - ansible-playbook -i ansible_files/inventario
       ansible_files/deploy-app.yml
```

O comando **ansible-playbook** presente no bloco script é responsável por iniciar a execução do playbook **deploy-app.yml**, utilizando o arquivo de inventário inventario, localizado na pasta **ansible_files/**. Este arquivo define os servidores-alvo onde a aplicação será implantada. Abaixo, segue a estrutura simplificada desse playbook:

Código-fonte 10 – Playbook de deploy

```
---
- name: Realizando o deploy da aplicacao {{ SISTEMA }}
  hosts: srv_apache
  ignore_unreachable: yes
  vars_files:
```

```
6       - roles/deploy-app/defaults/main.yml
7       - roles/deploy-app/vars/main.yml
8     roles:
9       - deploy-app
```

Esse playbook está estruturado com o uso de *roles*, um recurso do Ansible que permite organizar logicamente tarefas, variáveis e arquivos. As variáveis utilizadas durante a execução são carregadas de dois arquivos distintos (**defaults** e **vars**), o que garante flexibilidade e padronização das configurações aplicadas em diferentes ambientes. Durante a execução do playbook, o Ansible realiza tarefas como:

* A transferência dos arquivos da aplicação do repositório para o servidor de produção;
* A configuração do Virtual Host (VHost) no servidor Apache;
* A instalação das dependências específicas do projeto;

As tarefas são descritas em arquivos YAML dentro das *roles*, utilizando os módulos internos do Ansible (escritos em Python). No entanto, vale destacar que o projeto também possui scripts Python personalizados que são executados durante o *deploy*. Esses scripts foram desenvolvidos para lidar com ações mais específicas e complexas, como a criação automatizada de arquivos de configuração de **Virtual Host do Apache**, algo que vai além das funcionalidades padrão dos módulos do Ansible.

Esses scripts são integrados ao playbook e invocados por meio do módulo **shell**, ampliando a capacidade da automação e permitindo que tarefas específicas sejam tratadas de forma programática, quando necessário.

5 RESULTADOS

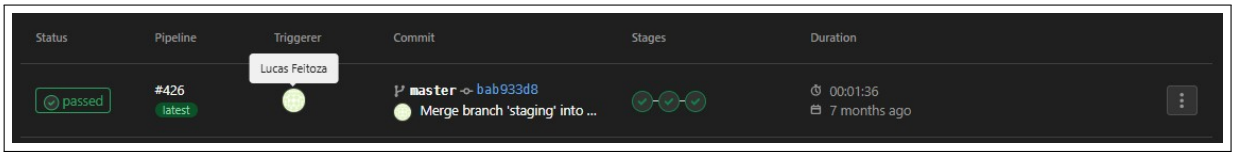
Neste capítulo, são apresentados os principais resultados obtidos com a implementação da pipeline de CI/CD na Prefeitura Municipal de Sobral (PMS). A análise considera tanto os dados extraídos das execuções automatizadas no GitLab CI/CD quanto o feedback dos desenvolvedores, obtido por meio de formulários aplicados antes e após a implantação da solução disponível no Apêndice B. Os resultados apontam melhorias significativas em aspectos como tempo de entrega, qualidade do software, confiabilidade do processo e redução de retrabalho. Também são discutidos os principais desafios enfrentados e percepções da equipe ao longo da transformação.

5.1 Desempenho e Otimização no Processo de Deploy

Antes da automação, o processo de construção e entrega de software era realizado manualmente, envolvendo etapas suscetíveis a erros, atrasos e retrabalho. Relatos indicaram que o tempo para disponibilização de uma aplicação em produção variava amplamente, podendo chegar a semanas, meses ou até mais de um ano, dependendo da complexidade do sistema. Esse cenário evidenciava a ausência de padronização e a dificuldade em prever prazos com precisão.

Com a adoção da pipeline CI/CD, esse tempo foi significativamente reduzido. Após o desenvolvimento e a aprovação do código, o processo de integração e *deploy* passou a ser executado de forma automatizada, levando em média 2 minutos para completar todas as etapas da entrega, como evidenciado na Figura 9. A automação eliminou a necessidade de intervenções manuais em tarefas repetitivas, reduzindo erros de configuração, falhas de *deploy* e garantindo maior consistência entre ambientes.

Figura 9 – Duração do processo de *deploy* pela Pipeline



Fonte: elaborado pelo autor (2024).

A integração do GitLab CI/CD com ferramentas como Ansible contribuiu diretamente para esse ganho de eficiência, promovendo entregas contínuas, seguras e rastreáveis, com visibilidade em tempo real sobre o andamento das execuções.

5.2 Impacto na Qualidade e Confiabilidade do Sistema

O processo manual anterior à implantação da pipeline resultava frequentemente em falhas na entrega, retrabalho e instabilidades em produção. A ausência de validações sistemáticas e notificações dificultava a identificação precoce de problemas.

Após a automação, a percepção de qualidade e confiabilidade do sistema aumentou de forma notável. De acordo com os dados coletados evidenciados na Tabela 3, 100% dos desenvolvedores relataram redução de erros, e destacaram ganhos em eficiência.

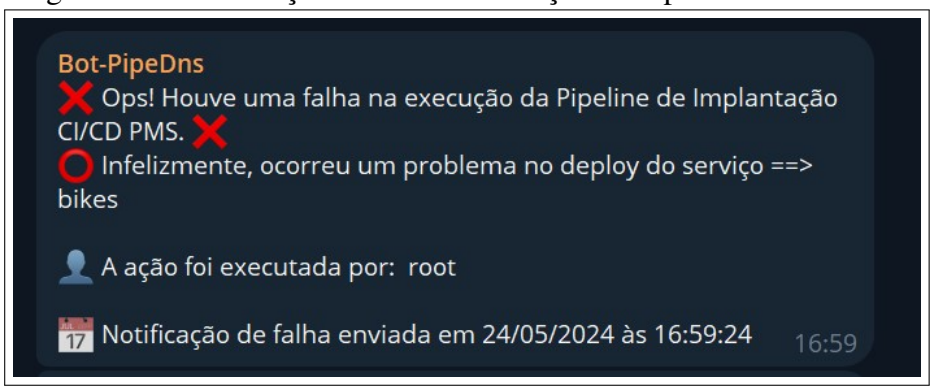
Tabela 3 – Comparação entre o Método Antigo e o Novo Método

Critério	**Método Antigo**	**Novo Método**
Eficiência	50% consideravam ineficiente	100% consideram eficiente
Erros/Re-trabalho	25% relataram erros frequentes; 50% ocasionalmente	100% relataram redução de erros
Satisfação	12,5% estavam muito satisfeitos	50% estão muito satisfeitos

Fonte: Elaborado pelo autor (2024).

Além disso, a introdução de alertas automatizados via Telegram também permitiu reações rápidas em caso de falhas, otimizando a comunicação da equipe e acelerando a correção de incidentes, como ilustra a Figura 10.

Figura 10 – Notificação de erro na execução da Pipeline



Fonte: elaborado pelo autor (2024).

A automação favoreceu o controle de versões, a consistência das entregas e a rastreabilidade das mudanças, tornando o processo mais robusto e menos propenso a falhas humanas.

5.3 Análise Comparativa dos Resultados Pré e Pós-Implantação da Pipeline

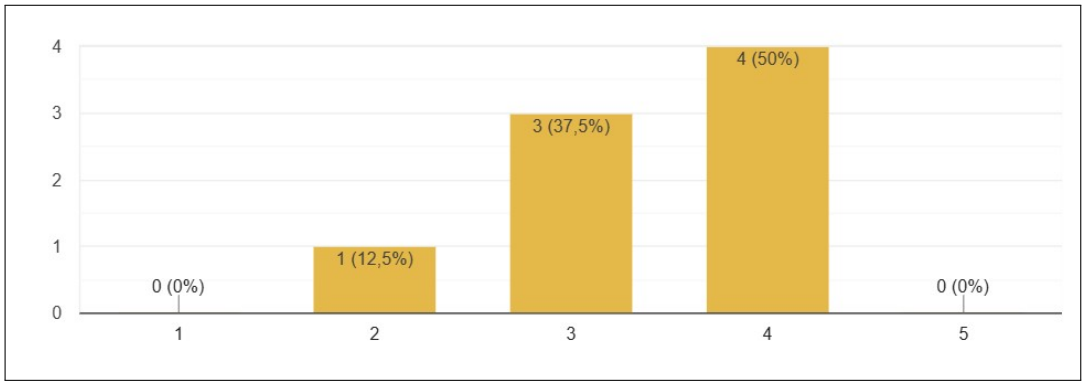
Com o objetivo de mensurar os impactos da automação, foi realizada uma comparação entre os dados obtidos antes e depois da implantação da pipeline. Os formulários aplicados revelaram uma evolução significativa no processo de entrega de software.

Antes da implantação, os principais desafios percebidos incluíam:

* Entregas demoradas e imprevisíveis, com variações de semanas a mais de um ano.
* Alta incidência de erros em produção e necessidade frequente de retrabalho.
* Baixa padronização nos processos e forte dependência de atividades manuais.
* Falta de visibilidade sobre o andamento das entregas.

Os resultados da pesquisa reforçam a percepção de que o método anterior de desenvolvimento contribui significativamente para a ocorrência de falhas. A maioria dos entrevistados demonstrou concordância com essa afirmação, enquanto uma parcela considerável permaneceu indecisa e apenas uma minoria discordou, isto pode ser evidenciado na Figura 11.

Figura 11 – Percepção de falhas com o antigo método



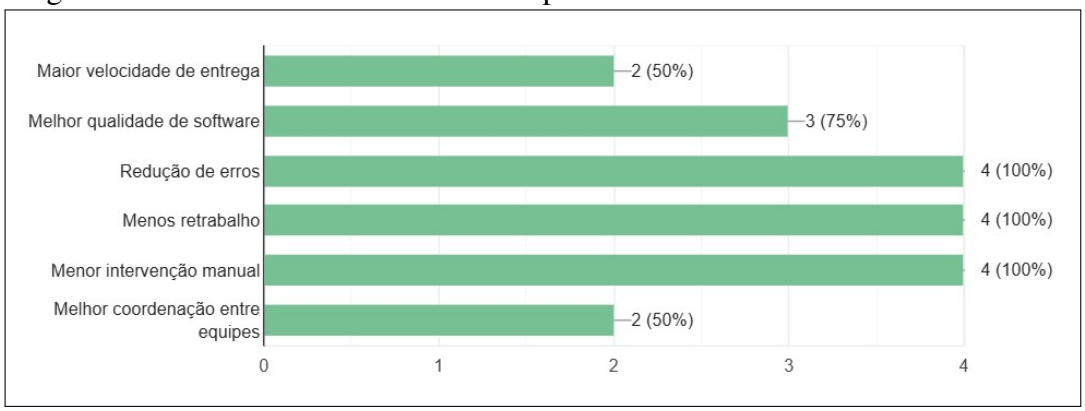
Fonte: elaborado pelo autor (2024).

Após a implantação da pipeline, os resultados coletados apontam:

* 100% dos respondentes observaram redução de erros, retrabalho e intervenção manual.
* 75% relataram melhora na qualidade do software.
* 50% perceberam maior velocidade de entrega e melhor coordenação entre equipes.

Os dados obtidos indicam uma melhoria significativa em diversos aspectos do processo, ilustrado na Figura 12.

Figura 12 – Benefícios observados da Pipeline



Fonte: elaborado pelo autor (2024).

Esses resultados demonstram que a pipeline teve impacto direto na eliminação de falhas recorrentes, além de promover maior padronização e automação no fluxo de entrega. A redução de retrabalho e erros reforça a eficácia das etapas automatizadas no controle de qualidade.

Adicionalmente, uma sugestão aberta foi registrada pelos respondentes, propondo a inclusão de testes automatizados na pipeline, o que indica um engajamento contínuo dos usuários na busca por aprimoramentos.

6 CONCLUSÕES E TRABALHOS FUTUROS

Neste trabalho, foi desenvolvida uma pipeline de entrega de software para a PMS, visando alcançar melhorias na eficiência do desenvolvimento, na qualidade das entregas e na redução de erros e retrabalho. A implementação da pipeline se mostrou essencial, uma vez que no modelo de desenvolvimento anterior foram identificadas oportunidades significativas de otimização, especialmente em relação ao tempo de *deploy* e às atividades manuais realizadas pela equipe.

Os resultados obtidos demonstraram um impacto positivo na eficiência do processo de desenvolvimento. Todos os respondentes relataram facilidade de uso, maior segurança e diminuição expressiva das atividades manuais. Além disso, foi percebida uma redução considerável no tempo de *deploy*, o que contribuiu diretamente para a agilidade nas entregas. A nova abordagem automatizada também resultou em maior confiabilidade do sistema, minimizando falhas e melhorando a qualidade final do software.

A comparação com o método anterior destacou a eficácia da nova pipeline em termos de produtividade e satisfação da equipe. A diminuição de erros e retrabalho, aliada à percepção de maior segurança no processo, consolidou a relevância da automação para a melhoria contínua no ciclo de desenvolvimento de software. Dessa forma, o trabalho contribui não apenas para o cenário atual da PMS, mas também para o entendimento de práticas eficientes de entrega contínua na indústria de software.

Contudo, duas observações são relevantes para a interpretação dos resultados da pesquisa aplicada. A primeira refere-se à elaboração da pergunta sobre o tempo estimado de entrega de software no formulário pré-implantação: faltou especificar que se tratava do tempo necessário para a realização do *deploy*, após o desenvolvimento e aprovação do código, o que pode ter gerado interpretações diferentes entre os participantes. A segunda observação diz respeito ao perfil dos respondentes — funcionários da própria entidade —, o que, apesar da pesquisa ter sido apresentada como estritamente acadêmica, pode ter gerado receio de comprometimento profissional e influenciado suas respostas, afetando parcialmente a fidelidade de alguns dados.

Para trabalhos futuros, sugere-se a expansão do escopo de testes na pipeline, buscando identificar melhorias contínuas e explorar integrações com soluções em *cloud*. A colaboração contínua entre as equipes será essencial para adaptar o processo às necessidades em constante evolução. Também é recomendada a análise de novas métricas de desempenho e qualidade, a fim de monitorar e otimizar ainda mais o fluxo de entrega.

REFERÊNCIAS

4LINUX. *O que é Ansible*. 2024. Disponível em: <https://4linux.com.br/o-que-e-ansible/>. Acesso em: 18 set. 2024.

ATLASSIAN. **Containers vs. virtual machines**. 2023. Disponível em: <https://www.atlassian.com/microservices/cloud-computing/containers-vs-vms>. Acesso em: 25 nov. 2023.

ATLASSIAN. **Saiba tudo sobre o Gitflow Workflow**. 2024. Disponível em: <https://www.atlassian.com/br/git/tutorials/comparing-workflows/gitflow-workflow>. Acesso em: 08 ago. 2024.

C, E.; MARTINS, G. Item de likert e escala de likert. **Revista Contabilidade Vista**, Universidade Federal de Minas Gerais, v. 32, n. 1, p. 1–5, 2021.

DEBIAN. **Estatísticas do concurso de popularidade para ansible, puppetmaster, salt-master, libchef-ruby**. 2023. Disponível em: <https://qa.debian.org/popcon-graph.php?packages=ansible%2C+puppetmaster%2C+salt-master%2C+libchef-ruby&show_installed=on&want_legend=on&want_ticks=on&from_date=2010&to_date=2018&hlght_date=&date_fmt=%25Y-%25m&beenhere=1>. Acesso em: 15 set. 2023.

DOCKER. **Comparando contêineres e máquinas virtuais**. 2023. Disponível em: <https://www.docker.com/resources/what-container/>. Acesso em: 10 ago. 2023.

DOCKER, I. **Why is Container?** 2023. Disponível em: <https://www.docker.com/resources/what-container/>. Acesso em: 07 nov. 2023.

DONCA I.-C.; STAN, O. M. M. G. D. M. L. Método para integração contínua e implantação usando um gerador de pipeline para projetos de software Ágeis. Sensores, 2022.

FOWLER, M. **Continuous Integration**. 2024. Disponível em: <https://martinfowler.com/articles/continuousIntegration.html>. Acesso em: 18 jan. 2024.

GADELHA, D. **Conhecendo o GitLab CI/CD**. 2021. Disponível em: <https://blog.cubos.io/conhecendo-o-gitlab-ci-cd/#>. Acesso em: 20 mai. 2021.

GITLAB. **Variáveis GitLab CI/CD**. 2023. Disponível em: <https://docs.gitlab.com/ee/ci/variables/>. Acesso em: 13 ago. 2023.

GITLAB, I. **What is Continuous Integration (CI)?** 2023. Disponível em: <https://about.gitlab.com/topics/ci-cd/>. Acesso em: 06 nov. 2023.

GITLAB, I. **What is DevOps?** 2023. Disponível em: <https://about.gitlab.com/topics/devops/>. Acesso em: 10 jun. 2023.

GITLAB, I. **O que é controle de versão?** 2024. Disponível em: <https://about.gitlab.com/pt-br/topics/version-control/>. Acesso em: 15 set. 2024.

GITLAB, I. **GitLab Runner**. 2025. Disponível em: <https://docs.gitlab.com/runner/>. Acesso em: 10 jan. 2025.

HUMBLE, J. **Entrega contínua**. [*Porto Alegre*]: Cambridge University Press, 2013.

HUMBLE, J.; FARLEY, D. **Entrega contínua**. [*Porto Alegre*]: Cambridge University Press, 2013.

HUTTERMANN, M. Devops for developers: Integrate development and operations, the agile way. Nova York: Apress, 2012.

IBM, I. **What is Infrastructure as Code (IaC)?** 2021. Disponível em: <https://www.ibm.com/think/topics/infrastructure-as-code>. Acesso em: 08 out. 2021.

IBM, I. **Why is Docker?** 2023. Disponível em: <https://www.ibm.com/topics/docker>. Acesso em: 10 nov. 2023.

KIM, G.; HUMBLE, J.; DEBOIS, P.; WILLIS, J. **Manual de DevOps: Como obter agilidade, confiabilidade e segurança em organizações tecnológicas**. [S.l.]: Altas Book, 2018. v. 2016.

LIMA, P. H. O. Estudo comparativo sobre as funcionalidades de integraÇÃo contÍnua e entrega contÍnua das ferramentas jenkins e gitlab. Centro Universitário Christus, 2021.

MIGUEL, O. **Top 10 Ferramentas de CI/CD Gratuitas que Você Não Pode Perder em 2024**. 2024. Disponível em: <https://apidog.com/pt/blog/free-ci-cd-tools-pt/>. Acesso em: 29 nov. 2024.

PYTHON. **What is Python? Executive Summary**. 2023. Disponível em: <https://www.python.org/doc/essays/blurb/>. Acesso em: 21 nov. 2023.

REDHAT, I. **O que é um pipeline de CI/CD?** 2022. Disponível em: <https://www.redhat.com/pt-br/topics/devops/what-cicd-pipeline>. Acesso em: 11 mai. 2022.

REDHAT, I. **O que é CI/CD?** 2023. Disponível em: <https://www.redhat.com/pt-br/topics/devops/what-is-ci-cd>. Acesso em: 01 nov. 2023.

REDHAT, I. **Why is Ansible?** 2023. Disponível em: <https://www.redhat.com/en/technologies/management/ansible/what-is-ansible>. Acesso em: 08 nov. 2023.

SILVA, R. R. V. Gitops: Uma nova proposta para a infraestrutura. Universidade Federal de Santa Catarina, 2020.

TECH, T. **What is Gitlab?** 2020. Disponível em: <https://www.techtarget.com/whatis/definition/GitLab>. Acesso em: 10 out. 2020.

VIDHYA. **Automate Everything With Python: A Comprehensive Guide to Python Automation**. 2023. Disponível em: <https://www.analyticsvidhya.com/blog/2023/04/python-automation-guide-automate-everything-with-python/>. Acesso em: 25 nov. 2023.

APÊNDICE A – QUESTIONÁRIO PRÉ IMPLANTAÇÃO

Questão 1. Com que frequência há atrasos na entrega de software?

(a) Nunca

(b) Raramente

(c) Ocasionalmente

(d) Frequentemente

(e) Muito frequente

Questão 2. Com que frequência há necessidade de correções em ambiente de homologação/produção?

(a) Nunca

(b) Raramente

(c) Ocasionalmente

(d) Frequentemente

(e) Muito frequente

Questão 3. Você concorda que o método de desenvolvimento adotado influencia para possíveis erros?

(a) Discordo totalmente

(b) Discordo

(c) Não estou decidido

(d) Concordo

(e) Concordo totalmente

Questão 4. Com que frequência há solicitações de mudanças no software em paralelo?

(a) Nunca

(b) Raramente

(c) Ocasionalmente

(d) Frequentemente

(e) Muito frequente

Questão 5. Com que frequência ocorre conflitos entre as branches no método de desenvolvimento de software atual?

(a) Nunca

(b) Raramente

(c) Ocasionalmente

(d) Frequentemente

(e) Muito frequente

Questão 6. Qual sua satisfação com o atual método de desenvolvimento de software?

(a) Muito insatisfeito

(b) Insatisfeito

(c) Indiferente

(d) Satisfeito

(e) Muito satisfeito

Questão 7. Qual a média estimada de tempo para a entrega de um software com o método atual de desenvolvimento?

Questão 8. Quais são os principais desafios e problemas que você e sua equipe enfrentam no processo de desenvolvimento de software atual?

Questão 9. Qual é a abordagem metodológica que a equipe utiliza para o desenvolvimento de software?

Questão 10. Como são realizadas as atualizações e manutenção de software após o lançamento inicial?

Questão 11. A equipe promove a documentação adequada de código e processos?

APÊNDICE B – QUESTIONÁRIO PÓS IMPLANTAÇÃO

Questão 1. Você concorda que a nova pipeline de entrega de software tem facilidade de uso?

(a) Discordo totalmente

(b) Discordo

(c) Não estou decidido

(d) Concordo

(e) Concordo totalmente

Questão 2. Você concorda que houve uma melhoria perceptível na eficiência do desenvolvimento após a implementação da pipeline?

(a) Discordo totalmente

(b) Discordo

(c) Não estou decidido

(d) Concordo

(e) Concordo totalmente

Questão 3. Você concorda que o número de erros ou retrabalho diminuiu com a nova pipeline?

(a) Discordo totalmente

(b) Discordo

(c) Não estou decidido

(d) Concordo

(e) Concordo totalmente

Questão 4. Qual seu nível de satisfação com a qualidade das entregas após a implementação da pipeline?

(a) Muito insatisfeito

(b) Insatisfeito

(c) Indiferente

(d) Satisfeito

(e) Muito satisfeito

Questão 5. Você concorda que a pipeline de entrega de software reduziu a maior parte das atividades manuais da equipe?

(a) Discordo totalmente

(b) Discordo

(c) Não estou decidido

(d) Concordo

(e) Concordo totalmente

Questão 6. Com que frequência a nova pipeline apresentou problemas técnicos?

(a) Nunca

(b) Raramente

(c) Ocasionalmente

(d) Frequentemente

(e) Muito frequente

Questão 7. O quanto satisfeito a equipe se sente agora em relação à automação da entrega de software?

(a) Muito insatisfeito

(b) Insatisfeito

(c) Indiferente

(d) Satisfeito

(e) Muito satisfeito

Questão 8. Você concorda que houve impacto nos prazos e redução de tempo de entrega após a implementação da pipeline?

(a) Discordo totalmente

(b) Discordo

(c) Não estou decidido

(d) Concordo

(e) Concordo totalmente

Questão 9. Você concorda que houve impacto da nova pipeline na segurança do processo de desenvolvimento de software?

(a) Discordo totalmente

(b) Discordo

(c) Não estou decidido

(d) Concordo

(e) Concordo totalmente

Questão 10. Quais os principais benefícios que você observou com a nova pipeline? (Selecione

todas que se aplicam)

- ☐ Maior velocidade de entrega
- ☐ Melhor qualidade de software
- ☐ Redução de erros
- ☐ Menos retrabalho
- ☐ Menor intervenção manual
- ☐ Melhor coordenação entre equipes

Questão 11. Que sugestões você tem para melhorar ainda mais o processo de entrega automatizada? (Campo aberto para resposta)

APÊNDICE C – CÓDIGOS-FONTES UTILIZADOS PARA AUTOMATIZAÇÃO DA PIPELINE DE ENTREGA DE SOFTWARE

Código-fonte 11 – Trecho principal do script de automatização do container

```
#!/bin/bash
# Cria o container com a porta desejada se disponivel
create_container() {
  local container_name=$1
  local desired_port=$2

  local config_file_path="/var/www/html/sistemas/$container_name/config/config.php"
  local old_line="'homologacao'       => 'http://hmg.my-domain.com/$container_name',"
  local new_line="'homologacao'       => 'https://domain-prod.com/$container_name',"
  if docker ps -a --format '{{.Names}}' | grep -q "^$container_name$"; then
      echo "O container $container_name esta em execucao."
      echo "Parando o container $container_name ."
      docker rm $container_name -f
      sleep 3
      echo "Avaliando se a porta solicitada $desired_port esta disponivel"
      local available_port=$(
        find_available_port_for_desired_port $desired_port
        )
      echo "Iniciando o container $container_name "
      docker run --name $container_name --network net-redis
          -p $available_port:443 -d $container_name:2.0
      echo "Container $container_name solicitado na porta $desired_port, mapeada para a porta
```

```
        $available_port."
20      docker exec $container_name sed -i "s|$old_line|
        $new_line|" $config_file_path
21      sleep 3
22      docker exec nginx-proxy "/etc/scripts/update_vhost.sh
        " "$container_name" "$available_port"
23  else
24      echo "O container $container_name nao esta em
        execucao."
25      echo "Avaliando se a porta solicitada $desired_port
        esta disponivel"
26      local available_port=$(
        find_available_port_for_desired_port $desired_port
        )
27      docker run --name $container_name --network net-redis
         -p $available_port:443 -d $container_name:2.0
28      echo "Container $container_name solicitado na porta
        $desired_port, mapeada para a porta
        $available_port."
29      docker exec $container_name sed -i "s|$old_line|
        $new_line|" $config_file_path
30      sleep 3
31      docker exec nginx-proxy "/etc/scripts/update_vhost.sh
        " "$container_name" "$available_port"
32  fi
33 }
34 # Recebe os argumentos digtados pelo usuario
35 container_name=$1
36 desired_port=$2
37 create_container $container_name $desired_port
```

Código-fonte 12 – Principais Trechos do Script de Criação de VHost

```
from modules.parameter_apache import args
from modules.execute_cmds import cmd_apache_http,
   cmd_apache_https

//Definicao de variaveis
path = args.file
vhost = args.name
user = args.user[0]
host = args.owner[0]
port_back = args.back[0]
assine_user = args.ass[0]

//Principal funcao para cricao do Vhost
def create_vhost_http(vhost, user, port_back, assine_user):
    create_dir()
    create_index()

    if vhost_exists(vhost, pathHttpd):
        print(f"O vhost {vhost} ja existe. Nao foi
           adicionado ao arquivo.")
        return
    else:
        try:
            list = [
                f"#Sistema: {vhost} - Solicitado por {user
                   }; Via Pipeline em: {data_current} by {
                   assine_user}\n",
                f"<VirtualHost *:{port_back}>\n",
                f"ServerAdmin admin-emailc@domain.com\n",
                f"DocumentRoot {deploy}\n",
                f"ServerName {vhost}.{domain}\n",
                f"ErrorLog /var/log/httpd/{vhost}.{domain}-
```

```
                    error_log\n",
29                  f"CustomLog /var/log/httpd/{vhost}.{domain
                    }-access_log combined\n",
30                  "</VirtualHost>\n"
31              ]
32
33              with open(pathHttpd, 'a') as myfile:
34                  for i in list:
35                      myfile.write(i)
36
37              time.sleep(8)
38              cmd_apache_http(vhost)
39          except Exception as e:
40              print(f"Erro: {e}")
```