UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ

MURILO NUNES MARÇAL

INTEGRAÇÃO E ENTREGA CONTÍNUA: COMPARAÇÃO DE FUNCIONALIDADES ENTRE JENKINS, GITLAB CI E GITHUB ACTIONS

GUARAPUAVA

2025

MURILO NUNES MARÇAL

INTEGRAÇÃO E ENTREGA CONTÍNUA: COMPARAÇÃO DE FUNCIONALIDADES ENTRE JENKINS, GITLAB CI E GITHUB ACTIONS

Continuous Integration and Deployment: Comparing Jenkins, GitLab CI, and GitHub Actions

Trabalho de Conclusão de Curso de Graduação apresentado como requisito para obtenção do título de Tecnólogo em Tecnologia em Sistemas para Internet do Curso Superior de Tecnologia em Sistemas para Internet da Universidade Tecnológica Federal do Paraná.

Orientador: Prof. Dr. Hermano Pereira

GUARAPUAVA

2025

4.0 Internacional

Esta licença permite compartilhamento, remixe, adaptação e criação a partir do trabalho, mesmo para fins comerciais, desde que sejam atribuídos créditos ao(s) autor(es). Conteúdos elaborados por terceiros, citados e referenciados nesta obra não são cobertos pela licença.

RESUMO

Com o crescimento da complexidade dos sistemas e a demanda por entregas ágeis e confiáveis, consolidaram-se as práticas de Integração Contínua (CI) e Entrega Contínua (CD) no contexto de *DevOps*. Este trabalho apresenta um estudo comparativo entre três ferramentas de CI/CD: GitHub Actions, GitLab CI/CD e Jenkins, por meio da implementação de *pipelines* equivalentes em cada plataforma e de uma análise funcional preliminar. Complementarmente, avaliaram-se métricas quantitativas como tempo de execução e estabilidade, e métricas qualitativas, como facilidade de configuração inicial, observabilidade, extensibilidade e custo operacional. Os resultados indicam que o Jenkins alcançou a menor mediana de tempo em ambiente controlado; o GitHub Actions destacou-se pela rápida configuração e integração ao ecossistema do GitHub; e o GitLab CI/CD apresentou alta estabilidade em tempo de execução e boa governança quando utilizado de forma integrada. Como contribuição, o estudo oferece recomendações por cenário que auxiliam equipes acadêmicas e profissionais na escolha da ferramenta mais adequada a seus contextos.

Palavras-chave: entrega contínua; integração contínua; análise de performance; desenvolvimento e operações.

ABSTRACT

As software systems grow in complexity and the demand for agile and reliable deliveries increases, Continuous Integration (CI) and Continuous Delivery (CD) practices have become consolidated within the DevOps context. This work presents a comparative study of three CI/CD tools: GitHub Actions, GitLab CI/CD, and Jenkins, by implementing equivalent pipelines on each platform and conducting a preliminary functional analysis. Additionally, we evaluated quantitative metrics such as execution time and stability, and qualitative aspects such as ease of initial configuration, observability, extensibility, and operational cost. The results indicate that Jenkins achieved the lowest median time in a controlled environment; GitHub Actions stood out for its fast setup and integration with the GitHub ecosystem; and GitLab CI/CD showed high runtime stability and sound governance when used as an integrated platform. As a contribution, the study offers scenario-based recommendations that help academic and professional teams choose the tool best suited to their contexts.

Keywords: continuous integration; continuous delivery; benchmarking; devops.

LISTA DE FIGURAS

Figura 1 – O ciclo de vida *DevOps* **14**

LISTA DE TABELAS

Tabela 1 – Análise funcional preliminar - comparação de recursos nativos de CI/CD. 35

Tabela 2 – Métricas quantitativas das execuções por ferramenta 36

Tabela 3 – Avaliação qualitativa por dimensão operacional (1–5). 37

LISTAGEM DE CÓDIGOS FONTE

Listagem 1 – Exemplo arquivo de configuração de *pipeline* do GitHub 20

Listagem 2 – Exemplo arquivo de configuração de *pipeline* do GitLab 21

Listagem 3 – Exemplo arquivo de configuração de pipeline do Jenkins usando Groovy 23

LISTA DE ABREVIATURAS E SIGLAS

Abreviaturas

CI	Continuous Integration (Integração Contínua)
CD	Continuous Delivery (Entrega Contínua)
DevOps	Development and Operations (Integração entre Desenvolvimento e Operações)
VPS	Virtual Private Server (Servidor Virtual Privado)
YAML	Yet Another Markup Language (Mais uma Linguagem de Marcação)
UI	User Interface (Interface do Usuário)
SaaS	Software as a Service (Software como Serviço)
SCM	Source Code Management (Gerenciamento de Código-Fonte)
CLI	Command-Line Interface (Interface de Linha de Comando)

SUMÁRIO

1 INTRODUÇÃO . . . 9

1.1 Considerações iniciais . . . 10

1.2 Objetivos . . . 11

1.2.1 Objetivo geral . . . 11

1.2.2 Objetivos específicos . . . 11

1.3 Justificativa . . . 11

2 REFERENCIAL TEÓRICO . . . 13

2.1 Desenvolvimento *(Dev)* e Operações*(Ops)* . . . 13

2.2 O ciclo de vida *DevOps* . . . 14

2.3 CI (Integração Contínua) . . . 15

2.4 CD (Entrega Contínua) . . . 16

2.5 Relação entre Integração Contínua e Entrega Contínua . . . 16

2.6 Esteira de *CI/CD (Pipeline)* . . . 17

2.7 Arquivos de Configuração de *Pipeline* . . . 18

2.8 Ferramentas de CI/CD . . . 19

2.8.1 GitHub Actions . . . 19

2.8.2 GitLab CI/CD . . . 21

2.8.3 Jenkins . . . 22

2.9 Benefícios e Desafios das Práticas de *CI/CD* . . . 24

2.9.1 Benefícios detalhados . . . 24

2.9.2 Desafios associados . . . 25

3 TRABALHOS RELACIONADOS . . . 26

4 MATERIAIS E MÉTODOS . . . 27

4.1 Tecnologias de Apoio ao Projeto . . . 27

4.2 Procedimento de Implementação . . . 28

4.3 Ambiente de Execução . . . 28

4.3.1 Justificativa das configurações. . . . 29

4.4 Versões das Ferramentas Utilizadas . . . 29

4.5 Estrutura do Estudo . . . 30

4.6 Análise Funcional Preliminar . . . 30
4.7 Comparação quantitativa . . . 31
4.8 Critérios da avaliação qualitativa . . . 32
4.9 Repositórios de Código-Fonte e imagem Docker . . . 33
5 RESULTADOS . . . 34
5.1 Organização dos Resultados . . . 34
5.2 Estrutura da Apresentação . . . 34
5.3 Análise funcional preliminar . . . 34
5.3.1 Discussão da tabela de análise funcional preliminar . . . 35
5.4 Comparação efetiva entre as ferramentas . . . 36
5.4.1 Resultados quantitativos das execuções . . . 36
5.4.2 Discussão dos resultados quantitativos e controles experimentais . . . 36
5.4.3 Resultados qualitativos . . . 37
5.4.4 Análise e interpretação da avaliação qualitativa . . . 37
5.4.5 Discussão integrada . . . 38
5.5 Recomendação de ferramenta por cenário . . . 39
5.5.1 Cenários corporativos . . . 39
5.5.2 Cenários acadêmicos . . . 40
5.5.3 Checklist decisório simplificado . . . 41
6 CONCLUSÃO . . . 42
REFERÊNCIAS . . . 44
GLOSSÁRIO . . . 48

1 INTRODUÇÃO

A automação de processos no desenvolvimento de software, através de ferramentas de integração e entrega contínua (CI/CD), (Seções 2.3 e 2.4), é uma prática essencial no contexto de DevOps (Seção 2.1) e desenvolvimento ágil, facilitando a entrega rápida e confiável de software. À medida que a complexidade dos projetos aumenta, ferramentas como GitHub Actions (GITHUB, INC., 2025a), GitLab CI/CD (GITLAB, INC., 2025b) e Jenkins (JENKINS, 2025b) surgem como soluções importantes para automatizar processos e garantir qualidade e eficiência no ciclo de desenvolvimento.

No contexto de DevOps, a automação desses processos é fundamental para garantir a integração eficaz entre desenvolvimento e operações, promovendo maior colaboração entre as equipes e reduzindo o tempo necessário para a entrega de novas funcionalidades. Além disso, o uso de pipelines (Seção 2.6) de CI/CD permite estruturar e padronizar a execução de testes, builds e deploys, garantindo um fluxo de trabalho mais confiável e eficiente (AMAZON WEB SERVICES, 2024).

A CI consiste na prática de integrar alterações de código frequentemente em um repositório compartilhado, onde são automaticamente verificadas e testadas. Essa abordagem minimiza riscos ao detectar erros precocemente, garantindo maior estabilidade ao desenvolvimento por meio de validações constantes a cada commit. Dessa forma, eventuais falhas podem ser corrigidas rapidamente, evitando que problemas se propaguem para versões posteriores do software (FOWLER, 2024).

Já a CD assegura que o software esteja sempre em um estado pronto para implantação, permitindo lançamentos frequentes e previsíveis. Essa prática possibilita que as equipes otimizem continuamente seus produtos, identificando e resolvendo conflitos entre o código novo e o existente de maneira ágil. Com isso, a correção de erros torna-se mais rápida e frequente, reduzindo o tempo necessário para entregar novas funcionalidades e melhorando a experiência dos usuários (RED HAT, 2025).

A seleção das ferramentas Jenkins, GitLab CI/CD e GitHub Actions para este estudo é fundamentada em sua ampla adoção e relevância no cenário atual de desenvolvimento de software. Essas ferramentas são reconhecidas não apenas por sua popularidade e uso em projetos de diversos portes, mas também por possuírem documentação extensa e detalhada, o que facilita a implementação e a resolução de problemas. Além disso, todas oferecem planos gratuitos, tornando-se acessíveis para equipes que buscam iniciar ou aprimorar suas práticas de CI/CD sem custos iniciais. Segundo relatórios e pesquisas de mercado, como o JETBRAINS (2023), essas três ferramentas estão entre as mais utilizadas para automação de pipelines de integração e entrega contínua. O Jenkins, uma ferramenta de código aberto consolidada, é reconhecido por sua flexibilidade e extenso ecossistema de plugins (JENKINS, 2025b). O GitLab CI/CD, por sua vez, se destaca por oferecer uma plataforma integrada de DevOps, unindo gestão de código, (CI/CD) e monitoramento em um único ambiente (GITLAB, INC., 2024). Já o

GitHub Actions, com sua integração nativa ao GitHub, permite a automação de fluxos de trabalho diretamente nos repositórios, facilitando a adoção por equipes que já utilizam a plataforma (GITHUB, INC., 2024a). Além disso, oferece diversos templates que simplificam a criação de pipelines (GITHUB, INC., 2025b).

1.1 Considerações iniciais

CI e CD são práticas fundamentais no desenvolvimento de software moderno, especialmente em ambientes que adotam metodologias ágeis e DevOps. Essas práticas são implementadas por meio de pipelines que automatizam a execução das etapas de compilação, testes, integração, revisão e implantação, permitindo um ciclo de desenvolvimento acelerado e com maior confiabilidade no processo de entrega de software (FOWLER, 2024). De acordo com HUMBLE e FARLEY (2010), CI e CD são essenciais para reduzir o tempo entre o desenvolvimento de uma funcionalidade e sua disponibilização, além de mitigar os riscos de falhas em produção.

Cada uma dessas plataformas apresenta suas características particulares, vantagens e desafios que podem impactar diretamente a eficiência de equipes de desenvolvimento. A escolha de uma ferramenta adequada pode afetar desde a facilidade de implementação das esteiras até a eficiência geral da equipe e do sistema (GITLAB, INC., 2025d). Como afirmado por DUVALL, MATYAS e GLOVER (2007), o sucesso na adoção de CI/CD depende não apenas da escolha das ferramentas, mas também da integração dessas com as práticas da equipe e do projeto.

GitHub Actions e GitLab oferecem integração nativa com o controle de versão, o que proporciona uma experiência mais fluida para desenvolvedores que já utilizam essas ferramentas. Por exemplo, o GitHub Actions, integrado ao ecossistema do GitHub, permite automação de workflows diretamente no repositório com templates prontos e integrações de ferramentas de terceiros pelo GitHub Marketplace (GITHUB, INC., 2024c).

Jenkins, por outro lado, como uma ferramenta open-source altamente configurável, permite uma flexibilidade maior em termos de customização e uso de plugins (JENKINS, 2025b). Entretanto, essa flexibilidade vem acompanhada de uma complexidade maior de configuração e uma curva de aprendizado mais acentuada. Segundo RIUNGU-KALLIOSAARI, MÄKINEN e TYRVÄINEN (2016), ferramentas como Jenkins, apesar de seu potencial de flexibilidade, podem ser mais adequadas para equipes com maior maturidade técnica, sendo preferidas em cenários que exigem customizações específicas e fluxos de trabalho complexos.

Neste cenário, existe uma demanda crescente por estudos que avaliem de forma comparativa essas plataformas de CI/CD, fornecendo diretrizes claras para diferentes contextos de uso (SHAHIN; BABAR; ZHU, 2017). Este trabalho insere-se nesse contexto ao realizar uma análise comparativa, considerando aspectos como facilidade de uso, complexidade de configuração e desempenho de execução dos pipelines.

1.2 Objetivos

Os objetivos deste trabalho consistem em realizar uma análise comparativa entre as ferramentas de CI/CD: Jenkins, GitLab CI/CD e GitHub Actions. Os resultados alcançados visam auxiliar a comunidade acadêmica e equipes de desenvolvimento de software na escolha da ferramenta mais adequada para seus projetos.

1.2.1 Objetivo geral

Avaliar e comparar as funcionalidades relacionadas à CI/CD das ferramentas: GitHub Actions, GitLab CI/CD e Jenkins.

1.2.2 Objetivos específicos

• Identificar as principais características e limitações de cada ferramenta, comparando suas funcionalidades;

• Oferecer recomendações práticas para a escolha da ferramenta mais adequada a diferentes contextos de desenvolvimento, baseadas nos resultados obtidos;

• Avaliar a facilidade de configuração e a velocidade de execução das pipelines, usando um conjunto padronizado de tarefas.

1.3 Justificativa

A justificativa para este trabalho reside na importância de coletar, analisar e fornecer informações comparativas das funcionalidades das ferramentas: Jenkins, GitLab CI/CD e GitHub Actions. Tal abordagem visa, eventualmente, auxiliar equipes de desenvolvimento de software no processo de escolha da solução mais adequada às suas necessidades. Diante da crescente complexidade dos projetos de software modernos e da fundamentalidade das práticas de CI/CD para a automação de processos e o aumento da eficiência no ciclo de desenvolvimento de software (FOWLER, 2024). Este estudo, portanto, busca contribuir para a otimização dos fluxos de trabalho e a melhoria contínua na entrega de software, preenchendo uma lacuna na pesquisa sobre qual ferramenta melhor se adapta a diferentes contextos.

A escolha das plataformas GitHub, GitLab e Jenkins reflete a busca por soluções amplamente adotadas no mercado e bem documentadas, que atendem às necessidades de equipes com diferentes perfis e contextos e que oferecem planos gratuitos, segundo a pesquisa (BLUELIGHT.CO, 2025).

O estudo também pode contribuir para a literatura acadêmica e prática, auxiliando equipes de desenvolvimento e engenheiros de software na escolha da ferramenta mais apropriada

para suas necessidades específicas, contribuição importante para o mercado de trabalho, onde a adoção de práticas CI/CD é cada vez mais indispensável para a manutenção da qualidade do software e do ciclo de entrega (RIUNGU-KALLIOSAARI; MÄKINEN; TYRVÄINEN, 2016).

2 REFERENCIAL TEÓRICO

O referencial teórico deste trabalho se concentra em conceitos e práticas relacionadas a DevOps, mais especificamente CI e CD, que desempenham um papel relevante no desenvolvimento de software. Para contextualizar o trabalho, este capítulo será dividido em seções que abordam as definições de CI e CD, as pipelines de CI/CD e as ferramentas em foco (GitHub Actions, GitLab CI/CD e Jenkins), além de uma análise das práticas e benefícios associados a cada uma delas.

2.1 Desenvolvimento *(Dev)* e Operações*(Ops)*

DevOps é uma abordagem cultural, filosófica e técnica que busca integrar as equipes de desenvolvimento e operações, promovendo colaboração, automação e monitoramento contínuo em todas as etapas do desenvolvimento de software, desde a codificação até a produção (AMAZON WEB SERVICES, 2024).

De acordo com BASS, WEBER e ZHU (2020), essa abordagem visa melhorar a agilidade das organizações por meio da integração dos processos de desenvolvimento e operação, reduzindo o tempo entre a implementação de uma alteração e sua disponibilização ao usuário final. Além disso, essa prática está associada à melhoria da qualidade do software, maior frequência de entrega e maior estabilidade nos ambientes de produção.

Segundo RIUNGU-KALLIOSAARI, MÄKINEN e TYRVÄINEN (2016), os benefícios da adoção de DevOps incluem maior eficiência no ciclo de vida do desenvolvimento, *feedback* mais rápido, melhor colaboração entre equipes e maior capacidade de resposta a mudanças. No entanto, os autores também destacam desafios, como a necessidade de mudança cultural, capacitação das equipes e adaptação de ferramentas e processos.

O DevOps serve como base para práticas como integração contínua e entrega contínua, permitindo que essas sejam implementadas de forma eficaz e integrada aos fluxos de trabalho modernos (RED HAT, 2025).

Historicamente, muitas organizações seguiram modelos tradicionais nos quais as equipes de desenvolvimento, testes e operações atuavam de forma isolada, com responsabilidades bem definidas e pouco diálogo entre os setores. Essa separação gerava diversos desafios, como a ausência de uma linguagem comum, dificuldade na troca de conhecimento e barreiras culturais que impactavam diretamente na eficiência e qualidade do software entregue (AMAZON WEB SERVICES, 2024).

A abordagem DevOps surge como uma resposta a esse modelo fragmentado, propondo a quebra desses silos organizacionais e incentivando uma mentalidade colaborativa. Ao integrar desenvolvimento, testes e operações em um fluxo contínuo, é possível alinhar os objetivos das equipes e promover uma visão mais unificada do produto e do cliente.

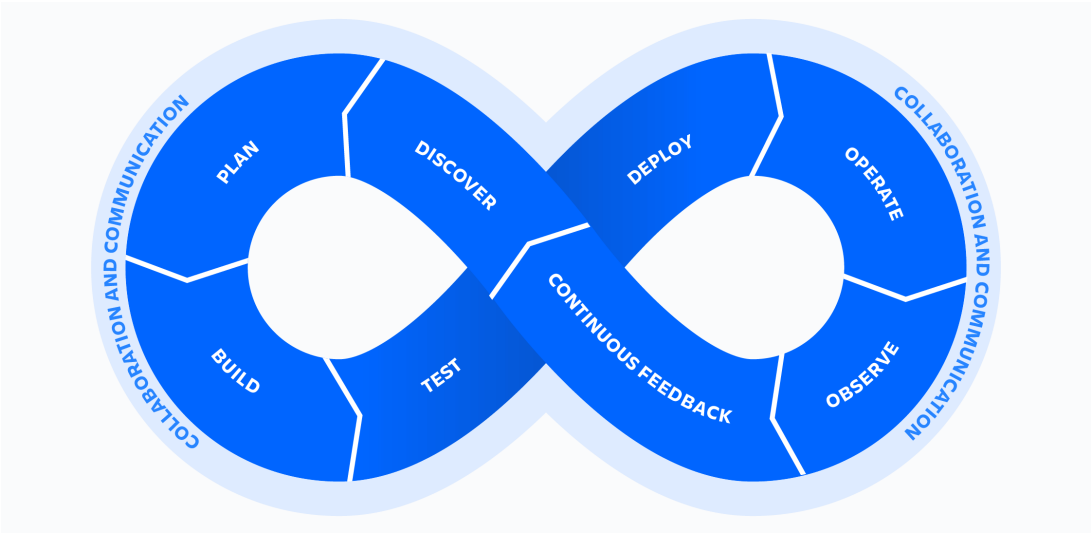
Com isso, práticas como automação de testes, entrega contínua e monitoramento em tempo real tornam-se viáveis e eficazes, reduzindo o tempo de entrega, aumentando a qualidade do software e permitindo respostas mais ágeis a mudanças e incidentes. Além disso, a criação de uma linguagem compartilhada e a busca por objetivos comuns ajudam a fortalecer a cultura organizacional e a disseminação de conhecimento entre os times.

Essa transformação, no entanto, exige mais do que apenas a adoção de ferramentas: requer uma mudança cultural profunda, onde valores como confiança, transparência e responsabilidade compartilhada passam a ser fundamentais no dia a dia das equipes.

2.2 O ciclo de vida *DevOps*

O ciclo DevOps é composto por diversas etapas que representam processos, ferramentas e práticas específicas. Quando integradas, essas fases visam acelerar o desenvolvimento de software, desde a concepção inicial até a entrega ao usuário final.

Figura 1 – O ciclo de vida *DevOps*



Fonte: (ATLASSIAN, 2025).

A Figura 1 ilustra o ciclo de vida DevOps, no qual as fases estão organizadas em um fluxo contínuo e iterativo. Conforme descrito pela Atlassian[1] cada etapa desempenha um papel fundamental para viabilizar entregas ágeis e confiáveis, com ênfase na automação e na melhoria contínua (ATLASSIAN, 2025). Essas fases estão interconectadas de forma cíclica, permitindo a evolução constante e a automatização progressiva dos processos.

A seguir, detalhamos cada uma das fases representadas na figura:

1. **Descoberta *(Discover)*:** Antes do desenvolvimento, ocorre a análise e identificação de oportunidades e riscos, permitindo que as equipes entendam melhor o contexto do projeto e tomem decisões informadas.

[1] A Atlassian é uma empresa de software conhecida por ferramentas como Jira, Confluence e Bitbucket.

2. **Planejamento *(Plan)*:** Corresponde ao momento em que as equipes definem as funcionalidades a serem implementadas, priorizando tarefas com base em metas estratégicas e necessidades do cliente.

3. **Desenvolvimento *(Build)*:** Nesta fase, o código é desenvolvido com o auxílio de sistemas de controle de versão, como o Git. A colaboração entre desenvolvedores é facilitada, e práticas de CI são aplicadas para garantir qualidade desde o início.

4. **Testes *(Test)*:** O código desenvolvido passa por testes automatizados para assegurar a estabilidade e qualidade do sistema. Essa etapa permite detectar falhas de forma precoce, reduzindo custos de correção.

5. **Implantação *(Deploy)*:** Utilizando práticas de CD, as equipes automatizam a publicação de novas versões do software. Para reduzir falhas e interrupções, são utilizadas técnicas de deploy gradual, que promovem uma transição mais segura entre versões e contribuem para a estabilidade do sistema.

6. **Operações *(Operate)*:** Esta fase envolve o gerenciamento da infraestrutura e dos serviços em produção, garantindo disponibilidade, desempenho e resiliência dos sistemas.

7. **Monitoramento *(Observe)*:** Ferramentas de monitoramento e análise de logs são utilizadas para acompanhar o funcionamento do sistema, identificar anomalias e embasar melhorias futuras.

8. **Feedback Contínuo *(Continuous Feedback)*:** Após a entrega, feedbacks de usuários e métricas de operação são analisados para retroalimentar o ciclo de desenvolvimento, gerando ajustes e novas melhorias.

2.3 CI (Integração Contínua)

CI é uma prática de desenvolvimento de *software* que incentiva os desenvolvedores a integrar suas mudanças de código frequentemente em um repositório (repo). Esse processo é geralmente automatizado para compilar, testar e verificar o código, ajudando a identificar e resolver erros rapidamente, sendo essencial para evitar problemas de integração e reduzir retrabalho, promovendo um desenvolvimento mais rápido e confiável.

Verificar automaticamente cada commit reduz o risco de falhas acumuladas e contribui para a melhoria da qualidade do software. Para garantir a integridade das branches principais, utiliza-se pipelines com regras que exigem a aprovação em todos os testes unitários e de integração antes que qualquer alteração seja incorporada. Essa abordagem evita que código instável seja introduzido, assegurando a qualidade e a estabilidade do software. Segundo FOWLER (2024), práticas como essas são especialmente úteis em ambientes que adotam metodologias

ágeis, pois facilitam a manutenção de versões consistentes e reduzem a introdução de falhas em produção.

Além de aumentar a velocidade e a frequência das integrações, também contribui para um processo mais seguro e previsível, fornecendo um *feedback* imediato sobre o impacto das mudanças realizadas. De acordo com DUVALL, MATYAS e GLOVER (2007) essa prática permite, através dos testes automatizados integrados na pipeline, que equipes de desenvolvimento identifiquem problemas de forma incremental e continuada, resultando em um desenvolvimento menos propenso a falhas e com entregas mais consistentes.

2.4 CD (Entrega Contínua)

CD é uma prática de desenvolvimento de software que automatiza a implantação de novas versões, garantindo que o software esteja sempre pronto para ser atualizado de forma rápida e segura. Sucede a CI e assegura que cada versão aprovada nos testes automatizados possa ser implantada imediatamente em ambientes de produção ou homologação, minimizando a intervenção manual e reduzindo os riscos de falhas.

De acordo com a AMAZON WEB SERVICES (2024)[2] a entrega contínua está alinhada com os princípios DevOps, promovendo colaboração entre as equipes de desenvolvimento e operações para tornar o processo de implantação mais ágil e seguro. A automação reduz falhas humanas e aumenta a consistência das entregas, o que é fundamental para empresas que adotam práticas ágeis e ciclos frequentes de entrega.

ATLASSIAN (2025) destaca que a CD não só reduz o tempo entre a conclusão do desenvolvimento e a entrega de novos recursos, mas também melhora a velocidade na identificação e correção de erros. Isso permite que as equipes implementem rapidamente atualizações e correções, mantendo um ciclo de desenvolvimento contínuo e altamente responsivo.

A entrega contínua se consolida como um pilar essencial para pipelines modernas; sua adoção permite que equipes respondam com rapidez a demandas técnicas e de negócios, mantendo a qualidade e a confiabilidade das entregas. Dessa forma, a prática não apenas otimiza o fluxo de trabalho, mas também estabelece as bases para estratégias mais avançadas de implantação (HUMBLE; FARLEY, 2010).

2.5 Relação entre Integração Contínua e Entrega Contínua

CI é um componente essencial para CD, pois prepara o software para a entrega contínua, certificando que ele seja frequentemente integrado e testado. Enquanto CI foca na verificação constante de cada alteração de código, CD expande essa prática ao automatizar a disponibili-

[2] A *Amazon Web Services (AWS)* é uma plataforma de serviços de computação em nuvem oferecida pela amplamente utilizada para desenvolvimento, hospedagem, automação e escalabilidade de aplicações.

zação do software em diferentes ambientes, como desenvolvimento, homologação e produção, garantindo que ele esteja tecnicamente apto para ser implantado a qualquer momento (RED HAT, 2025). Essa relação sinérgica assegura um fluxo de desenvolvimento contínuo e altamente responsivo.

É possível visualizar a relação intrínseca entre CI e CD na Figura 1, que ilustra um ciclo contínuo e iterativo de desenvolvimento e entrega de software. Este processo pode ser interpretado em fases interligadas:

A porção esquerda do diagrama representa a CI, abrangendo os primeiros estágios do ciclo de desenvolvimento. O processo se inicia com o Planejamento (1), onde são definidos os requisitos, funcionalidades e objetivos do projeto. Em seguida, o código é Escrito (2) com base nesse planejamento. As alterações realizadas são então Compiladas (3) em uma versão executável e submetidas a Testes (4) automatizados. Esses testes garantem que as mudanças não introduzam falhas e que o código continue integrado e funcional. A CI, portanto, atua como um filtro de qualidade contínuo, validando o software a cada pequena alteração. Essa validação contínua é fundamental para que a entrega contínua possa ocorrer com segurança.

A porção direita do diagrama representa a CD, que depende diretamente do sucesso da CI para automatizar a entrega e a operação do software. A CD só pode atuar com eficiência se tiver como base um software que foi corretamente integrado e testado previamente. Após a conclusão dos testes da CI, o software é empacotado para Lançamento (5), e então Implantado (6) nos ambientes de destino. As etapas seguintes incluem a Operação (7), onde o sistema é mantido ativo e monitorado, e o Monitoramento (8), que gera dados sobre o desempenho e uso da aplicação. Esses dados alimentam o ciclo de *Feedback* Contínuo, promovendo a melhoria contínua. Em suma, a CD depende da confiabilidade e estabilidade proporcionadas pela CI para garantir que as implantações sejam frequentes, automáticas e seguras.

A forma do diagrama enfatiza a natureza cíclica e contínua do DevOps, onde CI e CD são elementos cruciais interligados. A colaboração e comunicação permeiam todo o ciclo, garantindo que as equipes trabalhem juntas de forma eficaz, enquanto o feedback contínuo permite a melhoria contínua do processo e do produto.

2.6 Esteira de *CI/CD (Pipeline)*

Uma esteira de CI/CD, ou pipeline, é uma série de etapas automatizadas que uma aplicação percorre desde a submissão do código até a implantação final em produção. Essa pipeline é responsável por automatizar processos essenciais, como compilação, execução de testes, integração, revisão e implantação, formando um fluxo contínuo que acelera e padroniza o ciclo de desenvolvimento de software (SHAHIN; BABAR; ZHU, 2017).

A criação e configuração das pipelines de CI/CD geralmente são feitas por meio de arquivos de configuração escritos em formatos como YAML, que permitem descrever as etapas e os comandos necessários em uma estrutura organizada e legível. Esses arquivos especifi-

cam os passos que o código deve seguir no processo de integração e entrega contínua, como executar testes unitários, compilar a aplicação, realizar verificações de segurança e, finalmente, implantar a aplicação em um ambiente de desenvolvimento, homologação e produção (GITLAB, INC., 2024).

Para realizar essas etapas, a configuração das pipelines muitas vezes incorpora comandos e scripts em Shell Script, que são amplamente usados para manipular arquivos, configurar variáveis de ambiente, automatizar processos e controlar o comportamento das etapas da pipeline (GITLAB, INC., 2024). Esse nível de configuração detalhado permite que as pipelines sejam altamente personalizáveis e adequadas às necessidades específicas de cada projeto, facilitando a adaptação das equipes a requisitos distintos e acelerando o ciclo de desenvolvimento (BASS; WEBER; ZHU, 2020).

As Pipelines CI/CD desempenham um papel crucial na organização e automação de todo o processo de entrega de software, não apenas reduzem a intervenção manual, mas também promovem a consistência entre builds e deploys, minimizando erros e aumentando a confiança na entrega do software. Essas pipelines desempenham um papel central na metodologia DevOps, uma vez que permitem a integração entre desenvolvimento e operações, melhorando a colaboração entre equipes e permitindo entregas rápidas e frequentes (HUMBLE; FARLEY, 2010).

2.7 Arquivos de Configuração de *Pipeline*

Os arquivos de configuração de pipeline são documentos estruturados que definem o fluxo de execução automatizado de processos de integração e entrega contínua. Esses arquivos especificam tarefas como compilação, testes, deploy e outras etapas essenciais no desenvolvimento de software. Através desses arquivos, as ferramentas de CI/CD organizam e coordenam a execução automatizada do pipeline.

A maioria dessas ferramentas utiliza YAML como formato de escrita para os arquivos de configuração. O YAML é uma linguagem de serialização de dados que se destaca pela simplicidade, legibilidade e estrutura hierárquica baseada em indentação. Isso facilita a escrita e manutenção de *pipelines*, permitindo que desenvolvedores descrevam de forma declarativa as etapas do fluxo de trabalho (RED HAT, 2023).

Apesar do uso predominante do formato YAML na definição das pipelines de CI/CD, o Jenkins adota uma abordagem distinta ao empregar a linguagem Groovy em seus arquivos de configuração. O Groovy é uma linguagem de programação orientada a objetos, com características dinâmicas e maior concisão. No contexto do Jenkins, é utilizado no arquivo `Jenkinsfile` para descrever, de forma programável e flexível, as etapas da pipeline. Essa linguagem permite o uso de estruturas de controle, variáveis e funções personalizadas, tornando possível criar fluxos de automação mais complexos e adaptáveis às necessidades específicas de cada projeto.

Conforme BASS, WEBER e ZHU (2020), a automação com pipelines reduz erros humanos e melhora a eficiência dos processos de desenvolvimento. Além disso, HUMBLE e FARLEY (2010) destacam que a configuração declarativa promove reprodutibilidade e previsibilidade nas execuções, garantindo que os builds e deploys ocorram de maneira padronizada.

Dessa forma, a adoção de arquivos de configuração de pipeline escritos em YAML tornou-se uma prática essencial em DevOps e desenvolvimento moderno, possibilitando maior controle, automação e escalabilidade nos projetos.

Exemplos práticos de arquivos de configuração em YAML podem ser encontrados em seções posteriores deste trabalho, contemplando as ferramentas analisadas: GitHub Actions, GitLab CI/CD e Jenkins. Nessas seções, são apresentados trechos reais de código utilizados para configuração de pipelines.

2.8 Ferramentas de CI/CD

2.8.1 GitHub Actions

O GitHub Actions é uma solução de automação integrada ao GitHub, lançada em 2018. Ele permite que desenvolvedores configurem pipelines de CI/CD utilizando arquivos YAML diretamente no repositório do projeto. Seu principal diferencial é a integração nativa com o GitHub, simplificando a configuração de fluxos de trabalho (GITHUB, INC., 2024a).

Com o GitHub Actions, é possível definir triggers para disparar ações automáticas, como testes unitários, build da aplicação e deploy. O fluxo de trabalho é estruturado em jobs, que contêm steps específicos para execução (GITHUB, INC., 2024a).

Uma das vantagens do GitHub Actions é a sua vasta biblioteca de templates reutilizáveis disponibilizadas no GitHub Marketplace, permitindo que os usuários aproveitem configurações pré-definidas para automação de tarefas sem necessidade de escrever pipelines do zero.

No entanto, o GitHub Actions pode apresentar custos adicionais dependendo do uso de runners hospedados pelo próprio GitHub, o que pode ser um fator limitante para projetos maiores. Além disso, sua configuração, embora simplificada, pode ser menos flexível do que ferramentas como o Jenkins.

Para ilustrar a configuração descrita, apresenta-se, a seguir, um exemplo mínimo de arquivo YAML do GitHub Actions.

Listagem 1 – Exemplo arquivo de configuração de *pipeline* do GitHub

```
name: Exemplo Trigger e Jobs

on: # Trigger: Descreve quando executar a esteira
  push:
    branches:
      - main

jobs: # Jobs: Descreve as ações que a esteira executará
  build-and-test:
    runs-on: ubuntu-latest

    steps:
    - name: Checkout code
      uses: actions/checkout@v3

    - name: Compilar codigo
      run: npm run build

    - name: Testar codigo
      run: npm run test
```

Fonte: Autoria própria (2025).

A Figura 1 apresenta um exemplo de workflow definido em um arquivo YAML para automação com GitHub Actions. Esse fluxo de trabalho é acionado automaticamente sempre que ocorre um push na branch *"main"* do repositório.

O workflow consiste em um único job, chamado *"build-and-test"*, que executa três etapas principais, cada uma desempenhando uma função específica no processo de automação:

1. *Checkout* do código-fonte: Responsável por clonar o repositório para o ambiente de execução.

2. Compilação do código: Utiliza o comando *"npm run build"* para processar a construção da aplicação.

3. Execução de testes: Roda a suíte de testes automatizados com *"npm run test"*, garantindo que o código está funcionando corretamente.

Como demonstrado na Figura 1, o GitHub Actions permite configurar triggers que determinam quando um workflow será executado. No exemplo, a execução ocorre sempre que um push é feito na branch *"main"*, mas é possível definir outros eventos ou execuções manuais, conforme a necessidade do projeto.

Essa flexibilidade é essencial para otimizar o uso de recursos e garantir que a automação aconteça apenas quando necessário. Além disso, a configuração manual pode ser útil para cenários onde é exigida uma aprovação humana, como um deploy em produção, assegurando um maior controle sobre o processo.

2.8.2 GitLab CI/CD

O GitLab CI/CD é a solução de automação nativa do GitLab, lançada em 2015, que permite a criação e execução de pipelines diretamente dentro da plataforma, utilizando arquivos YAML armazenados no repositório. O GitLab CI/CD possibilita a definição de fluxos automatizados para tarefas como compilação, testes e implementação, garantindo maior eficiência no desenvolvimento de software (GITLAB, INC., 2024).

Um dos grandes diferenciais do GitLab CI/CD é sua abordagem baseada em stages, onde cada job executa uma parte da pipeline de forma organizada e sequencial (GITLAB, INC., 2025a). Além disso, a ferramenta oferece suporte a runners dedicados, permitindo que as pipelines sejam executadas em servidores próprios ou na infraestrutura compartilhada do GitLab. Essa flexibilidade torna o GitLab CI/CD altamente escalável para projetos de diferentes tamanhos (GITLAB, INC., 2025c).

O ponto forte do GitLab CI/CD é a sua integração com o próprio GitLab, facilitando a configuração de pipelines sem necessidade de ferramentas externas. A plataforma também disponibiliza templates pré-configurados para diversas linguagens de programação, agilizando a implementação de pipelines personalizados para diferentes tipos de projetos.

Listagem 2 – Exemplo arquivo de configuração de *pipeline* do GitLab

```
1  workflow:
2    rules:
3      - if: '$CI_COMMIT_BRANCH == "main"'
4
5  build-job:
6    stage: build
7    image: node:18
8    script:
9      - npm install
10     - npm run build
11   artifacts:
12     paths:
13       - dist/
14
15 test-job:
16   stage: test
17   image: node:18
18   script:
19     - npm test
```

Fonte: Autoria própria (2025).

A Figura 2 ilustra um exemplo de pipeline no GitLab CI/CD, também configurado para ser acionado automaticamente, segue uma estrutura modular, garantindo organização, reutilização e escalabilidade no processo de integração e entrega contínua.

1. Build da aplicação – O job *"build-job"* instala as dependências e executa *"npm run build"*, gerando a versão compilada do projeto.

2. Execução de testes – O job *"test-job"* executa *"npm test"*, validando a integridade do código antes de um possível deploy.

Cada job pertence a um stage específico e só será executado quando todos os jobs do stage anterior forem concluídos com sucesso.

O job build-job pertence ao stage build e será responsável por compilar a aplicação. Ele utiliza a imagem oficial do Node.js (versão 18) como ambiente de execução; após executar a compilação, armazena os arquivos gerados durante a execução do job para serem utilizados em estágios posteriores. Neste caso, a pasta `"dist/"` (que contém os arquivos da build) será salva e disponibilizada para o próximo estágio.

Essa esteira modular do GitLab CI/CD permite uma organização eficiente das etapas de automação, garantindo que a aplicação seja construída e testada de forma controlada. Essa abordagem modular e declarativa facilita a manutenção e escalabilidade dos *pipelines*, permitindo que equipes de desenvolvimento automatizem seus fluxos de trabalho de maneira eficaz (FOWLER, 2024).

2.8.3 Jenkins

O Jenkins é uma ferramenta de automação open-source amplamente utilizada para CI/CD. Criado originalmente como parte de outro projeto, tornou-se independente em 2011, destaca-se por sua extensibilidade, permitindo que equipes configurem pipelines altamente personalizadas para atender a diferentes cenários de desenvolvimento (JENKINS, 2025b).

Diferente de soluções como GitHub Actions e GitLab CI/CD, que possuem uma integração nativa com suas respectivas plataformas, o Jenkins opera de forma autônoma, sendo possível utilizá-lo com qualquer provedor de repositório Git ou outra ferramenta de versionamento.

Sua configuração pode ser feita tanto de forma declarativa, utilizando arquivos de configuração, quanto de maneira imperativa, por meio da interface gráfica (JENKINS, 2024c).

Uma das maiores vantagens do Jenkins é a sua extensibilidade por meio de plugins, com milhares de opções disponíveis atualmente, possibilitando integração com diversas tecnologias, como Docker, Kubernetes, bancos de dados, ferramentas de monitoramento e provedores de nuvem (JENKINS, 2024a). Esse ecossistema expansível torna o Jenkins altamente adaptável a diferentes tipos de projetos.

Além disso, o Jenkins permite a execução de pipelines em agentes distribuídos, conhecidos como *nodes*, possibilitando a escalabilidade horizontal da infraestrutura de CI/CD (JENKINS, 2024b). Essa abordagem é especialmente útil para grandes equipes e projetos comple-

xos, onde múltiplas pipelines podem ser executadas simultaneamente, reduzindo o tempo de build e testes.

Outro diferencial do Jenkins é a possibilidade de configurar pipelines a partir de templates pré-definidos, por meio do *Jenkins Shared Libraries*, sendo possivel padronizar e compartilhar lógica reutilizável entre diferentes projetos, promovendo consistência e eficiência na automação (JENKINS, 2025a).

Listagem 3 – Exemplo arquivo de configuração de pipeline do Jenkins usando Groovy

```
pipeline {
    agent any
    stages {
        stage('Build') {
            steps {
                checkout scm
                sh 'npm install'
                sh 'npm run build'
            }
            post {
                success {
                    archiveArtifacts artifacts: 'dist/**', fingerprint: true
                }
            }
        }
        stage('Test') {
            steps {
                sh 'npm test'
            }
        }
    }
}
```

Fonte: Autoria própria (2025).

Essa pipeline também segue uma abordagem modular, onde os estágios são organizados sequencialmente para garantir um fluxo estruturado de build e testes, assim como nas figuras 1 e 2.

A estrutura da pipeline no Jenkins permite um alto nível de personalização. Os agentes (definidos pela diretiva *agent*) são responsáveis por executar as tarefas da pipeline. No exemplo acima, o agente é configurado como *any*, indicando que a pipeline pode rodar em qualquer ambiente disponível. No entanto, é possível definir agentes específicos, como máquinas dedicadas ou contêineres Docker (JENKINS, 2024b).

Além disso, o uso de archiveArtifacts permite armazenar os arquivos gerados no estágio de build para serem reutilizados posteriormente (JENKINS, 2025c). Isso é útil para pipelines que envolvem múltiplas fases, como testes em ambientes distintos e deploy em produção.

Com sua abordagem flexível, escalável e altamente extensível, o Jenkins continua sendo uma das ferramentas mais robustas para CI/CD. Ele permite integração com praticamente qual-

quer ferramenta do ecossistema DevOps, oferecendo um nível avançado de personalização para equipes que buscam controle total sobre suas pipelines (JENKINS, 2025b).

2.9 Benefícios e Desafios das Práticas de *CI/CD*

A adoção de práticas de CI/CD traz diversos benefícios, incluindo maior agilidade na entrega de software, redução de riscos e melhoria na qualidade do produto final (SHAHIN; BABAR; ZHU, 2017). No entanto, também apresenta desafios, como a necessidade de uma mudança cultural dentro das equipes de desenvolvimento e a complexidade na configuração inicial das ferramentas. RIUNGU-KALLIOSAARI, MÄKINEN e TYRVÄINEN (2016) enfatizam a importância de entender esses desafios para garantir uma transição bem-sucedida para práticas de DevOps.

2.9.1 Benefícios detalhados

• Entrega Rápida e Frequente: A automação de compilação, testes e implantação permite que o software seja entregue aos usuários de forma mais rápida e com maior frequência. Isso acelera o ciclo de *feedback* e permite que novas funcionalidades e correções sejam disponibilizadas prontamente. A CD, por exemplo, garante que o software esteja sempre pronto para implantação, possibilitando lançamentos frequentes e previsíveis (AMAZON WEB SERVICES, 2025a).

• Melhoria da Qualidade do Software: A integração frequente e os testes automatizados garantem que as alterações de código sejam continuamente verificadas e testadas. Isso minimiza riscos ao detectar erros precocemente, assegurando a estabilidade e a qualidade do software através de validações constantes a cada commit (AMAZON WEB SERVICES, 2025b). A identificação e correção de erros tornam-se mais ágeis e frequentes.

• Redução de Riscos: Ao validar o código em pequenas partes e de forma contínua, as falhas são detectadas e corrigidas antes que se propaguem e se tornem mais complexas e caras de resolver (FOWLER, 2024). Isso reduz o risco de introduzir código instável em ambientes de produção. A automação também minimiza erros humanos inerentes a processos manuais de implantação.

• Colaboração Aprimorada: As práticas de CI/CD promovem uma maior colaboração entre as equipes de desenvolvimento e operações, alinhando objetivos e fortalecendo a cultura organizacional. O feedback contínuo e a transparência dos processos incentivam a comunicação e a responsabilidade compartilhada (RED HAT, 2025).

• Consistência e Reprodutibilidade: As pipelines de CI/CD padronizam a execução de todas as etapas do desenvolvimento, como compilação, testes e implantação. Isso garante que

os builds e deploys ocorram de maneira padronizada, promovendo reprodutibilidade e previsibilidade nas execuções (HUMBLE; FARLEY, 2010).

2.9.2 Desafios associados

• Mudança Cultural: Um dos maiores desafios é a necessidade de uma transformação cultural profunda dentro das organizações. As equipes precisam abandonar silos tradicionais e adotar uma mentalidade de colaboração, confiança e responsabilidade compartilhada (SAMANTHA, 2018).

• Manutenção Contínua: As pipelines e a infraestrutura de CI/CD podem exigir manutenção contínua e atualização, o que pode ser um recurso intensivo. A complexidade aumenta com o tamanho do projeto e o número de integrações.

• Segurança: A automação de todo o ciclo de entrega de software exige uma atenção redobrada à segurança, descartando a ideia de que a segurança é uma fase distinta e, em vez disso, garantindo que seja uma parte inerente do ciclo de vida de entrega do software. É crucial integrar práticas de *DevSecOps*[3] nas pipelines para que vulnerabilidades sejam detectadas e mitigadas precocemente, reduzindo riscos ao longo do desenvolvimento e da implantação (PALO ALTO NETWORKS, 2025).

• Seleção da Ferramenta Adequada: Diante da diversidade de ferramentas de CI/CD disponíveis no mercado, as equipes enfrentam um desafio significativo ao tomar decisões sobre qual solução adotar. A escolha da solução mais adequada depende de múltiplos fatores, como a facilidade de configuração, a escalabilidade necessária, o suporte à personalização e a integração com o ecossistema tecnológico já existente da equipe. Avaliar essas características e ponderar os pontos fortes e limitações de cada ferramenta requer um entendimento aprofundado e pode influenciar diretamente a eficiência do fluxo de trabalho e a otimização dos recursos do projeto (GITLAB, INC., 2025d).

[3] *DevSecOps* é uma abordagem que integra segurança em todas as fases do ciclo de vida de desenvolvimento de software, desde o planejamento e codificação até a implantação e operação, visando automatizar e incorporar práticas de segurança de forma contínua.

3 TRABALHOS RELACIONADOS

Esta Seção apresenta estudos acadêmicos e técnicos anteriores que realizaram análises comparativas ou avaliaram ferramentas de CI/CD. A discussão tem como objetivo destacar resultados e critérios considerados relevantes por diferentes autores sobre as ferramentas estudadas, contextualizando GitHub Actions, GitLab CI/CD e Jenkins.

SHAHIN, BABAR e ZHU (2017) realizaram uma revisão sistemática sobre abordagens, ferramentas e desafios relacionados à CI. O estudo destaca que Jenkins e GitHub Actions são amplamente adotados devido à sua flexibilidade e à capacidade de integração com outros serviços. Complementarmente, RIUNGU-KALLIOSAARI, MÄKINEN e TYRVÄINEN (2016) destacam que a adoção dessas ferramentas contribui significativamente para a redução de erros manuais e aprimoramento da colaboração entre as equipes.

Na análise comparativa feita por GRACIANO e KAKINOHANA (2023) entre GitHub Actions e GitLab CI/CD, critérios como tempo de execução dos pipelines, facilidade de configuração e integração com outras ferramentas são abordados. Nesse estudo, GitHub Actions destaca-se pela integração fluida com o ecossistema GitHub e facilidade de uso, enquanto o GitLab CI/CD apresenta vantagens na flexibilidade da configuração e no suporte a runners dedicados, que permitem execução em infraestrutura própria.

Segundo LIMA (2021), ao comparar especificamente as funcionalidades de Jenkins e GitLab CI/CD, Jenkins oferece maior flexibilidade devido à sua natureza independente e extensa disponibilidade de plugins. No entanto, ressalta que isso resulta em uma complexidade maior na configuração e manutenção. Já o GitLab CI/CD é descrito como oferecendo uma experiência mais simplificada e intuitiva, particularmente para equipes que já utilizam o GitLab para gestão de código.

Estudos como o de DUVALL, MATYAS e GLOVER (2007) reforçam a importância de considerar fatores como tempo de execução dos pipelines, facilidade de uso e suporte abrangente para testes automatizados ao selecionar uma ferramenta de CI/CD. Adicionalmente, o relatório BLUELIGHT.CO (2025) classifica GitHub Actions, GitLab CI/CD e Jenkins como líderes de mercado, destacando os pontos fortes e limitações de cada um e enfatizando que a escolha depende do nível de controle desejado sobre a infraestrutura e das características do ecossistema de desenvolvimento utilizado.

Esses estudos fornecem uma base importante que justifica a relevância da análise comparativa proposta neste trabalho. Ao considerar as funcionalidades, limitações, facilidade de configuração, os resultados deste estudo buscam contribuir com orientações práticas para equipes de desenvolvimento na escolha da solução mais adequada às suas necessidades específicas.

4 MATERIAIS E MÉTODOS

Para a realização deste trabalho, foram selecionadas três ferramentas de integração e entrega contínua: GitHub Actions (Seção 2.8.1), GitLab CI/CD (Seção 2.8.2) e Jenkins (Seção 2.8.3). Essas plataformas foram escolhidas por sua ampla adoção na indústria de software e pelas diferentes abordagens que oferecem.

Além das ferramentas de CI/CD, foram utilizados outros materiais essenciais para garantir a criação de um ambiente de testes controlado e a execução das pipelines de maneira padronizada. A seguir, são descritos os principais componentes empregados neste estudo, com foco em suas funções e características mais relevantes para a análise comparativa.

4.1 Tecnologias de Apoio ao Projeto

- **Git**: Sistema de controle de versão utilizado para o gerenciamento do código-fonte do projeto. Permitiu rastrear alterações, colaborar com diferentes plataformas e acionar as pipelines de CI/CD (SCM, 2025).
- **Projeto Base**: As pipelines foram aplicadas sobre um projeto desenvolvido com a tecnologia *.NET*, escolhida por sua ampla adoção no desenvolvimento de aplicações corporativas (MICROSOFT, 2025f).
- **.NET CLI**: A interface de linha de comando do .NET foi utilizada nas etapas de compilação, execução de testes e publicação (MICROSOFT, 2025e; MICROSOFT, 2025b).
- **Shell Script**: Utilizado para automatizar tarefas recorrentes dentro do ambiente de execução das pipelines, como configurações de ambiente e chamadas de comandos auxiliares.
- **YAML**: Linguagem de marcação empregada na definição das pipelines nas três plataformas analisadas. Sua estrutura declarativa e sintaxe simplificada facilitaram a criação, manutenção e leitura dos fluxos de automação (YAML, 2021).
- **Docker e Docker Hub**: O Docker foi utilizado para a criação de imagens Docker do projeto. As imagens geradas foram publicadas automaticamente no Docker Hub (DOCKER, 2025b; DOCKER, 2025c).
- **Trivy**: Ferramenta de análise de segurança utilizada para escanear as imagens Docker em busca de vulnerabilidades conhecidas. Sua integração às pipelines permitiu verificar possíveis falhas antes da publicação das imagens no repositório, elevando o nível de segurança no processo de CD (TRIVY, 2025a).

4.2 Procedimento de Implementação

Para garantir equidade na comparação, todas as pipelines foram empregadas sobre o mesmo projeto base, e as etapas implementadas em cada ferramenta seguiram o seguinte fluxo:

1. **Restaurar dependências**: Utilização do *dotnet restore* para baixar os pacotes via NuGet (MICROSOFT, 2025c; MICROSOFT, 2025g).
2. **Compilar o projeto**: Execução de *dotnet build* para validar a integridade do código (MICROSOFT, 2025a).
3. **Executar testes**: Execução dos testes automatizados com *dotnet test*, utilizando o *framework* xUnit (MICROSOFT, 2025d; XUNIT.NET, 2025).
4. **Gerar imagem Docker**: Utilização do Docker CLI para construir a imagem da aplicação (DOCKER, 2025a).
5. **Verificar vulnerabilidades**: Escaneamento da imagem Docker gerada com a ferramenta Trivy antes do envio ao Docker Hub (TRIVY, 2025b).
6. **Publicar no Docker Hub**: Autenticação e envio da imagem para um repositório privado, utilizando credenciais armazenadas como variáveis de ambiente seguras em cada plataforma (DOCKER, 2025c).

As tarefas foram definidas por arquivos YAML nas plataformas GitHub Actions e GitLab CI/CD, e em Groovy no Jenkins.

4.3 Ambiente de Execução

As execuções das pipelines foram realizadas em ambientes controlados, visando garantir a padronização das condições e possibilitar uma comparação justa e precisa entre as ferramentas avaliadas. Cada ferramenta teve seu ambiente configurado da seguinte forma:

- **Jenkins:** Executado localmente em um container Docker, a instalação e execução seguiram as instruções oficiais de instalação em Docker disponíveis na documentação do Jenkins (JENKINS, 2025).
- **GitHub Actions:** Utilizado diretamente no ambiente SaaS fornecido pelo GitHub, com *GitHub Hosted runners* (GITHUB, 2025c).
- **GitLab CI/CD:** Executado diretamente na infraestrutura SaaS fornecida pelo GitLab, utilizando os *GitLab Shared Runners* (GITLAB, INC., 2025c).

4.3.1 Justificativa das configurações.

As escolhas de ambiente visaram comparabilidade entre as plataformas e reprodutibilidade do estudo:

- **Jenkins em contêiner (4 GB RAM, 2 vCPUs):** opção em Docker para isolar variações de hardware e facilitar réplica do experimento; os recursos adotados estão *alinhados à recomendação de infraestrutura para uma equipe pequena* (*"Recommended hardware configuration for a small team"*) da documentação oficial do Jenkins (JENKINS, 2025).
- **GitHub Actions (SaaS, *Hosted runners*):** uso do ambiente padrão do provedor, refletindo o cenário mais comum de adoção e tornando a comparação direta com o GitLab hospedado (GITHUB, 2025e).
- **GitLab CI/CD (SaaS, *Shared Runners*):** escolha equivalente ao GitHub em modo hospedado (GITLAB, 2025c).

4.4 Versões das Ferramentas Utilizadas

Para assegurar a transparência e a reprodutibilidade do estudo, esta seção apresenta as versões das ferramentas empregadas na execução das pipelines.

- **Jenkins:** Foi utilizada a **versão LTS 2.60.3**, executada a partir da imagem oficial `jenkins/jenkins:lts-jdk17` em contêiner Docker. Essa versão representa a edição de suporte de longo prazo (Long-Term Support), garantindo estabilidade e compatibilidade para execução contínua das pipelines (JENKINS, 2025a).
- **GitHub Actions:** Por se tratar de uma solução SaaS, a ferramenta é continuamente atualizada e não possui uma versão fixa disponível para controle direto pelo usuário. As execuções foram realizadas utilizando os *GitHub-hosted runners* fornecidos pelo provedor, com ambiente de execução padronizado e atualizado automaticamente pela plataforma. Todos os testes foram conduzidos na versão do serviço disponível em setembro de 2025.
- **GitLab CI/CD:** Assim como o GitHub Actions, o GitLab CI/CD foi utilizado em sua versão SaaS, também atualizada automaticamente pela infraestrutura oficial da plataforma. Dessa forma, não há controle direto sobre a versão exata da aplicação, visto que as atualizações são contínuas e gerenciadas pelo provedor. As execuções foram realizadas em *GitLab Shared Runners*, correspondentes à versão do serviço vigente em setembro de 2025.

Essa padronização assegura que os testes foram realizados em ambientes estáveis, representando o comportamento esperado nas configurações mais comuns de uso em produção.

4.5 Estrutura do Estudo

O estudo foi conduzido em três etapas complementares. Inicialmente, realizou-se uma análise funcional preliminar das plataformas de CI/CD selecionadas, com foco em recursos nativos e limitações de adoção usando os critérios especificados na seção 4.6. Em seguida, implementaram-se pipelines equivalentes nas três plataformas, estruturadas para executar as mesmas etapas e, assim, assegurar condições homogêneas de comparação, as etapas estão especificadas na seção 4.2. Por fim, procedeu-se à análise dos resultados qualitativos, avaliados segundo critérios definidos na seção 4.8. Em conjunto, essas três etapas fornecem o embasamento para as recomendações de ferramenta por cenário.

4.6 Análise Funcional Preliminar

Antes da execução prática dos experimentos, foi realizada uma análise funcional preliminar das ferramentas selecionadas. Essa análise teve como objetivo identificar e comparar os principais recursos nativos de cada plataforma de CI/CD, com foco em aspectos que impactam diretamente sua adoção. Os critérios considerados nessa etapa incluíram:

- **Integração com repositórios Git externos**: Verificação da compatibilidade com provedores Git como GitHub, GitLab e *Bitbucket*.
- **Complexidade da configuração inicial da pipeline**: avaliação, do esforço para alcançar a primeira execução bem-sucedida, com base na documentação oficial e na análise da interface de cada ferramenta. Foram considerados como critérios o número de etapas manuais necessárias, a clareza das instruções e a necessidade de configuração de agentes ou serviços adicionais. Escala utilizada: Baixa: configuração rápida, baseada em modelos prontos e um único arquivo YAML, sem instalação ou muitos ajustes adicionais; Moderada: requer alguns ajustes iniciais, como variáveis, permissões e pequenas modificações em templates; Alta: demanda instalação de servidores, configuração de agentes e integração manual de componentes antes da primeira execução.
- **Disponibilidade de componentes reutilizáveis**: Presença de templates e plugins que acelerem e padronizem a construção das pipelines.
- **Gratuidade do uso (camada gratuita)**: Identificação de limites de minutos/armazenamento e demais restrições aplicáveis aos planos gratuitos.
- **Disponibilidade em modelo SaaS**: Verificação da oferta totalmente hospedada na nuvem (com runners compartilhados/gerenciados) versus a necessidade de instalação local.

- **Suporte à execução Runner *self-hosted***: Capacidade de operar em infraestrutura própria, com maior controle sobre agentes e ambiente de execução.
- **Suporte a múltiplos ambientes**: Verificação da aptidão para orquestrar estágios como desenvolvimento, homologação e produção.
- **UI para visualização da pipeline**: Existência e qualidade da interface para monitorar, depurar e controlar a execução.
- **Paralelização de jobs nativa**: Capacidade de executar múltiplos jobs em paralelo, reduzindo o tempo total de build.
- **Cache de dependências**: Suporte a armazenamento de artefatos e dependências entre execuções, acelerando builds subsequentes.
- **Integração com notificações/alertas**: Facilidade de configurar alertas via e-mail e outras APIs.

A análise funcional forneceu uma visão comparativa inicial sobre a adaptabilidade, extensibilidade e facilidade de adoção de cada ferramenta, complementando os resultados obtidos posteriormente nos testes práticos, embasando os experimentos e auxiliando a recomendação de ferramenta em diferentes contextos. Os dados e evidências considerados foram extraídos das documentações oficiais e guias técnicos de cada ferramenta disponíveis até setembro de 2025, data de formatação deste trabalho (JENKINS, 2025b; GITHUB, INC., 2025a; GITLAB, INC., 2025b).

4.7 Comparação quantitativa

Para cada ferramenta foram realizadas $n = 10$ execuções independentes da mesma pipeline. Esse número foi escolhido para reduzir o efeito de variações momentâneas da infraestrutura, observar aquecimento de cache e reaproveitamento de dependências entre rodadas, detectar falhas intermitentes que não aparecem em uma única execução e verificar a estabilidade dos tempos ao longo das medições. Todas as execuções seguiram as condições definidas neste capítulo: mesmo código-fonte, mesmos estágios, versões fixadas de ferramentas e imagens, e política de cache previamente definida.

Com os tempos coletados, calculamos média, mediana e desvio-padrão. A média descreve o desempenho geral, mas pode ser puxada por rodadas mais lentas. A mediana mostra o comportamento mais típico, por ser menos sensível a oscilações e picos. O desvio-padrão indica o quanto os tempos variaram entre as rodadas: valores menores significam maior previsibilidade; valores maiores, mais variação. Quando média e mediana ficam próximas, a distribuição tende a ser mais equilibrada; quando se afastam, há indícios de assimetrias ou variações

de infraestrutura. Em cada rodada também anotamos o tempo total, em segundos, e se houve falhas, para avaliar desempenho e confiabilidade.

4.8 Critérios da avaliação qualitativa

Cada dimensão descrita a seguir foi avaliada em escala de 1 a 5, variando de desempenho insuficiente a excelente. A interpretação adotada foi: 1 (insuficiente) não atende aos requisitos mínimos e exige intervenção manual elevada; 2 (limitado) atende parcialmente, serve para casos simples; 3 (adequado) cumpre o esperado em cenários comuns com confiabilidade razoável e algum esforço operacional; 4 (bom) oferece cobertura ampla, boa documentação e integrações, baixa intervenção manual e desempenho consistente; 5 (excelente) apresenta maturidade e cobertura completas, recursos avançados bem integrados, alta previsibilidade e segurança, além de baixo custo operacional. Os resultados qualitativos foram apurados após a implementação das pipelines e constam na Tabela 3.

- **Facilidade configuração inicial (primeira pipeline)**: esforço e tempo para configurar o primeiro fluxo executável, incluindo curva de aprendizado e clareza da documentação.
- **Observabilidade/monitoramento da pipeline**: qualidade de logs, visualizações de etapas e artefatos, suporte a notificações e facilidade de depuração durante a execução.
- **Reuso (ações/templates/plugins)**: disponibilidade, maturidade e facilidade de adoção de componentes reutilizáveis para compor pipelines de forma padronizada.
- **Paralelização de jobs e controle de fluxo**: capacidade de executar tarefas em paralelo, definir dependências, condicionais e matrizes de execução com previsibilidade.
- **Gestão de Segredos *(secrets)***: mecanismos nativos para cadastro, proteção, escopo e uso seguro de variáveis e credenciais nas execuções.
- **Integrações práticas**: facilidade de integração com ferramentas externas, incluindo *scanners* de segurança como Trivy e container registry como Docker Hub.
- **Extensibilidade e plugins externos**: suporte a extensões, APIs e ecossistemas de terceiros, bem como facilidade de desenvolvimento e manutenção desses componentes.
- **Custo operacional (manutenção/atualizações)**: esforço contínuo para manter a plataforma, além de impactos de administração e suporte.

4.9 Repositórios de Código-Fonte e imagem Docker

Para garantir a transparência e a reprodutibilidade do estudo, os experimentos foram conduzidos em repositórios públicos, que contêm o projeto base e as configurações das pipelines para cada plataforma. A cada novo commit no código-fonte, todas as esteiras de CI/CD foram configuradas para iniciar automaticamente e de forma síncrona. Isso garantiu que as execuções fossem comparadas sob as mesmas condições, refletindo um cenário real de desenvolvimento.

- **Repositório GitHub:** https://github.com/Murilothebr/tcc-cicd-comparativo. Pipeline (caminho no repositório): `.github/workflows/dotnet-desktop.yml`.
- **Repositório GitLab:** https://gitlab.com/murilothebr1-group/tcc-cicd-comparativo. Pipeline (caminho no repositório): `.gitlab-ci.yml`.
- **Jenkins:** *Pipeline Script* (Groovy) armazenado na raiz dos repositórios. Pipeline (caminho nos repositórios): `Jenkinsfile`.
- **Docker Hub:** O repositório no Docker Hub foi utilizado para armazenar e versionar as imagens Docker geradas. Acessível em https://hub.docker.com/repository/docker/murilothebr/tcc-cicd-comparativo.

5 RESULTADOS

Este capítulo apresenta os resultados obtidos a partir de: (i) uma análise funcional preliminar das plataformas GitHub Actions, GitLab CI/CD e Jenkins (Tabela 1), que, de forma simplificada, delimita o escopo de aplicação de cada ferramenta e explicita seus principais pontos fortes e limitações, servindo de base para as comparações subsequentes; e (ii) da execução, em condições controladas, de pipelines equivalentes em cada uma delas, avaliando tempo de execução e estabilidade, seguindo os critérios e procedimentos definidos em Materiais e Métodos (Cap. 4), sintetizadas na Tabela 2. Em paralelo, apresentam-se os resultados qualitativos (Tabela 3), derivados da experiência direta de implantação nas três plataformas e avaliados segundo critérios igualmente especificados em Materiais e Métodos. Em conjunto, os achados funcionais, quantitativos e qualitativos fornecem o embasamento para as recomendações por cenário apresentadas ao final do capítulo.

5.1 Organização dos Resultados

Os resultados foram estruturados de forma a fornecer uma visão clara e objetiva, permitindo identificar tanto os pontos fortes quanto as limitações de cada solução. Inicialmente, são apresentados os resultados da análise funcional preliminar, que evidenciam os recursos nativos de cada ferramenta e sua flexibilidade de adoção. Em seguida, são discutidos os resultados práticos da execução das pipelines, considerando métricas quantitativas, como tempo de execução e estabilidade, e qualitativas, como facilidade de monitoramento e integração com ferramentas externas.

5.2 Estrutura da Apresentação

Com base nessa organização, os resultados serão apresentados em tabelas comparativas e acompanhados de descrições analíticas. Essa abordagem tem como objetivo facilitar a compreensão e destacar as diferenças entre as plataformas avaliadas, de modo a embasar a análise comparativa realizada nos capítulos seguintes.

5.3 Análise funcional preliminar

Inicialmente, apresenta-se a análise funcional preliminar das ferramentas, cujo objetivo foi delimitar pontos fortes e limitações de cada plataforma de CI/CD. Os critérios avaliados e o procedimento adotado encontram-se detalhados em Materiais e Métodos (Seção 4.6). A Tabela a seguir sintetiza os principais achados.

Tabela 1 – Análise funcional preliminar - comparação de recursos nativos de CI/CD.

Critério	**GitHub Actions**	**GitLab CI/CD**	**Jenkins**
Integração com repositórios Git externos	Nativa apenas para repositórios do GitHub.	Aceita repositórios externos (GitHub/Bitbucket) com ajustes.	Via *plugins* com Git, GitHub, GitLab e Bitbucket.
Complexidade da configuração inicial da *pipeline*	Baixa (YAML e *marketplace*).	Baixa–moderada (templates e `include:`).	Alta (instalação, agentes e *plugins*).
Disponibilidade de componentes reutilizáveis	Ações e templates reutilizáveis.	Templates e exemplos prontos.	*Plugins* e *Shared Libraries*.
Gratuidade do uso (camada gratuita)	Camada gratuita com minutos/armazenamento limitados.	*Free tier* com minutos/armazenamento; pode usar *runners* próprios.	Sem licença; custo em infraestrutura e manutenção.
Disponibilidade em modelo SaaS	Sim (*GitHub-hosted runners*).	Sim (*Shared Runners*).	Não oficialmente: requer infraestrutura própria.
Suporte à execução *self-hosted*	Sim (*self-hosted runners*).	Sim (Self-Managed e *runners*).	Sim (nativo: *controller* + *agents*).
Suporte a múltiplos ambientes	Sim.	Sim.	Sim.
UI para visualização da *pipeline*	Grafo de *jobs* e logs.	Grafo da *pipeline* e *analytics*.	Blue Ocean/Stage View (*plugins*).
Paralelização de *jobs* nativa	Sim.	Sim.	Sim.
Cache de dependências	`actions/cache`.	*cache* nativo no `.gitlab-ci.yml`.	*Plugins* e `stash/unstash`.
Integração com notificações/alertas	Integrações nativas (e-mail) e integrações via Webhook na UI.	Integrações nativas (e-mail) e integrações via Webhook na UI.	Oferece integração por extensões e configurações manuais.

Fonte: elaboração própria.

5.3.1 Discussão da tabela de análise funcional preliminar

A Tabela 1 evidencia um equilíbrio entre facilidade de adoção e grau de controle. O GitHub Actions destaca-se pela configuração inicial simples e pelo ecossistema maduro de ações/templates, sendo especialmente vantajoso quando o código já está no GitHub; a limitação é a integração nativa centrada na própria plataforma. O GitLab CI/CD oferece bom meio-termo: aceita repositórios externos, dispõe de templates e pode operar tanto em modo SaaS quanto self-managed, o que favorece a governança sem complicar o início. O Jenkins é o mais flexível e extensível (arquitetura de plugins, Shared Libraries e controle fino de orquestração), mas demanda maior esforço de instalação e manutenção, tornando-se mais indicado para cenários com

requisitos específicos, integrações legadas ou necessidade de rodar em infraestrutura própria. Nos fundamentos funcionais (paralelização, múltiplos ambientes, notificações e cache), há paridade conceitual: todas as plataformas entregam os recursos essenciais, variando como cada um é configurado e o esforço operacional associado. Em síntese, soluções SaaS (GitHub/GitLab) reduzem o tempo até a primeira pipeline e a sobrecarga de operação, enquanto o Jenkins maximiza a liberdade e portabilidade ao custo de maior administração.

5.4 Comparação efetiva entre as ferramentas

Esta seção consolida a comparação entre GitHub Actions, GitLab CI/CD e Jenkins a partir de duas perspectivas complementares: (i) quantitativa, baseada nas execuções padronizadas das pipelines; e (ii) qualitativa, considerando aspectos de usabilidade operacional, governança e ecossistema. A combinação desses resultados sustenta as recomendações por cenário ao final da seção.

5.4.1 Resultados quantitativos das execuções

As métricas consideradas e a metodologia utilizada estão expostas na Seção 4.7 (Materiais e Métodos). A Tabela abaixo consolida os principais resultados quantitativos das execuções.

Tabela 2 – Métricas quantitativas das execuções por ferramenta

Ferramenta	Média (s)	Mediana (s)	Desvio-padrão	Falhas/n
GitHub Actions	71	67	11,3	0/10
GitLab CI/CD	125	125	0,9	1/10
Jenkins	49	48	6,1	0/10

Fonte: elaboração própria.

5.4.2 Discussão dos resultados quantitativos e controles experimentais

Os dados da Tabela 2 indicam diferenças claras de desempenho entre as plataformas. O Jenkins apresentou os menores tempos (média 49 s; mediana 48 s) com baixa variabilidade (DP 6,1 s), compatível com a execução local em container, sem fila de runners compartilhados e com menor sobrecarga de provisionamento. O GitHub Actions obteve tempos intermediários (média 71 s; mediana 67 s) e variância moderada (DP 11,3 s), possivelmente refletindo variações naturais de infraestrutura em runners hospedados. O GitLab CI/CD foi mais estável, porém mais lento (média=mediana 125 s; DP 0,9 s), e registrou 1 falha em 10 execuções; esse padrão é compatível com runners compartilhados e políticas de execução homogêneas, ainda que com

maior latência média. A falha registrada no GitLab ocorreu durante a etapa de varredura de vulnerabilidades com o Trivy, em que, mesmo após execuções bem-sucedidas, houve erro pontual na integração entre a ferramenta e a plataforma.

5.4.3 Resultados qualitativos

As dimensões avaliadas e a escala de pontuação utilizadas nesta etapa encontram-se descritas em detalhe na Seção 4.8, que apresenta os critérios adotados para cada aspecto operacional.

Tabela 3 – Avaliação qualitativa por dimensão operacional (1–5).

Dimensão	GitHub Actions	GitLab CI/CD	Jenkins
Facilidade configuração inicial	5	5	2
Observabilidade/monitoramento da pipeline	4	4	5
Reuso (ações/templates/plugins)	5	5	5
Paralelização de jobs e controle de fluxo	4	4	5
Gestão de variáveis/segredos	4	4	5
Integrações práticas	5	5	4
Extensibilidade e plugins externos	3	3	5
Custo operacional (manutenção/atualizações)	5	5	2

Escala de avaliação: 1=insuficiente, 2=limitado, 3=adequado, 4=bom, 5=excelente.

Fonte: elaboração própria.

5.4.4 Análise e interpretação da avaliação qualitativa

- **Facilidade configuração inicial (primeira pipeline):** GitHub Actions (5) e GitLab (5) aceleram a primeira pipeline com YAML, templates e runners hospedados; o Jenkins (2) exige instalação, ajuste de agentes e curadoria de plugins.
- **Observabilidade:** Jenkins (5) atinge alta visibilidade e painéis customizáveis; GitHub/-GitLab (4/4) oferecem gráficos/execuções claros, mas menos flexíveis no ajuste fino.
- **Reuso:** Os três (5/5/5) fornecem alto reaproveitamento com actions/templates no GitHub/GitLab e Shared Libraries/plugins no Jenkins.
- **Paralelização e controle de fluxo:** Jenkins (5) apresenta maior flexibilidade e capacidade de orquestração entre tarefas; GitHub Actions e GitLab CI/CD (4/4) oferecem bom suporte à execução paralela e à definição de dependências entre etapas.
- **Variáveis/segredos:** Jenkins (5) oferece mecanismos robustos para gerenciamento de variáveis e credenciais, permitindo maior controle sobre o acesso a dados sensíveis.

GitHub/GitLab (4/4) disponibilizam cofres nativos e políticas de proteção por ambiente, atendendo de forma adequada à maioria dos casos de uso.

- **Integrações práticas:** GitHub/GitLab (5/5) integram facilmente registries e ferramentas (por exemplo, Trivy) com mínima fricção; no Jenkins (4) isso também é possível, porém normalmente demanda escolher ou ajustar plugins e imagens base.
- **Extensibilidade:** Apenas o Jenkins (5) possui arquitetura formal de plugins que estende o núcleo; GitHub/GitLab (3/3) expandem via actions/templates/APIs, mas não por plugins de plataforma.
- **Custo operacional:** GitHub/GitLab (5/5) minimizam manutenção de infraestrutura; o Jenkins (2) requer atualização do servidor, agentes e plugins, elevando o ônus operacional.

5.4.5 Discussão integrada

De modo geral, verificou-se uma contraposição entre a rapidez de implantação das soluções e o grau de controle proporcionado sobre a infraestrutura:

- **GitHub Actions** tende a oferecer a menor barreira de entrada (arquivos YAML e actions maduras), o que favorece menores tempos médios e baixa variância nas execuções em runners hospedados. Como contrapartida, a integração é centrada no GitHub, o que pode aumentar a dependência da plataforma em cenários com muitos repositórios de código.
- **GitLab CI/CD** apresenta um bom equilíbrio entre facilidade de adoção e flexibilidade, sendo uma solução mais independente que o GitHub em relação a provedores e ecossistemas. Por ser uma ferramenta de código aberto, oferece liberdade de personalização e possibilidade de implantação em modo self-hosted, o que garante maior controle e adequação a diferentes ambientes corporativos. Além disso, proporciona uma boa experiência de observabilidade e integração com seus próprios serviços nativos.
- **Jenkins** destaca-se pelo caráter independente em relação a provedores e tecnologias, além da extensibilidade por meio de plugins, permitindo cenários altamente customizados. O preço desse controle é uma curva inicial mais íngreme e manutenção contínua (atualizações, compatibilidade de plugins, gestão de agentes), o que pode elevar o tempo até a primeira pipeline estável.

Esses achados corroboram a análise funcional preliminar (Tabela 1): as soluções SaaS (GitHub Actions/GitLab CI/CD) reduzem o esforço inicial e padronizam práticas (reuso, cache de dependências, paralelização), ainda que com menor margem para customizações profundas; já

as soluções self-hosted (Jenkins) ampliam o controle e a extensibilidade, mas implicam maior custo operacional contínuo (provisionamento, atualização de plugins/agentes e manutenção de infraestrutura).

5.5 Recomendação de ferramenta por cenário

As recomendações abaixo derivam dos achados quantitativos (Tabela 2) e qualitativos (Tabela 3), bem como da análise funcional preliminar (Tabela 1). Para cada contexto, são apresentadas a ferramenta considerada mais adequada e as respectivas recomendações de uso, destacando eventuais limitações e cuidados necessários à sua adoção. As recomendações refletem a experiência do autor na aplicação prática das ferramentas e são apoiadas pelas documentações oficiais (GITHUB, INC., 2025a; GITLAB, INC., 2025b; JENKINS, 2025b).

5.5.1 Cenários corporativos

a) **Equipe pequena/média focada em agilidade.**
Recomendação: **GitHub Action ou GitLab CI/CD**.
Por quê: menor barreira de entrada no modelo SaaS, actions/templates maduros e possibilidade de migrar para runners hospedados (GITHUB, 2025c; GITHUB, INC., 2024c; GITLAB, 2025c). Tempos intermediários e consistentes (Tabela 2).
Atenções: dependência do ecossistema; cotas/minutos na camada gratuita (GITHUB, 2025a; GITLAB, 2025a).

b) **Corte de custos de licença e com infraestrutura própria já disponível.**
Recomendação: **Jenkins**.
Por quê: sem custo de licença; melhor aproveitamento da infraestrutura existente (JENKINS, 2025).

c) **Código já hospedado no GitHub com múltiplos serviços pequenos.**
Recomendação: **GitHub Actions**.
Por quê: reaproveitamento rápido de workflows e actions do Marketplace; fácil replicação entre repositórios (GITHUB, INC., 2024c; GITHUB, 2025d).
Atenções: gerenciar limites de paralelismo e padronizar secrets em nível de organização (GITHUB, 2025c; GITHUB, 2025f).

d) **Projeto já inserido no ecossistema GitHub.**
Recomendação: **GitHub Actions**.
Por quê: integração nativa com diversos serviços do ecossistema GitHub, proporcionando uma experiência completa e bem integrada para desenvolvimento, automação e implantação de aplicações. (GITHUB, INC., 2024b)

Atenções: cotas de armazenamento e rede, políticas de retenção de pacotes/imagens, permissões entre organizações e limpeza periódica de artefatos/versões antigas (GITHUB, 2025a; GITHUB, 2025b).

e) **Organização padronizada em GitLab (governança centralizada).**
Recomendação: **GitLab CI/CD**.
Por quê: oferece integração nativa entre os principais serviços do ecossistema GitLab, como repositórios de código, pipelines de CI/CD, registro de imagens, gerenciamento de issues e merge requests, além de recursos analíticos integrados (GITLAB, INC., 2025b; GITLAB, 2025b; GITLAB, 2025d; GITLAB, 2025e). Essa centralização favorece a padronização de processos e o controle unificado de projetos e equipes.
Atenções: avaliar limites de minutos e armazenamento na versão SaaS, bem como a necessidade de manutenção adicional em implantações self-hosted (GITLAB, 2025a).

f) **Ambiente com múltiplos provedores de repositório e integrações legadas.**
Recomendação: **Jenkins**.
Por quê: agnosticidade em relação ao provedor de código e riqueza de plugins/Shared Libraries; adequado para orquestrações complexas e fluxos não triviais (JENKINS, 2024a; JENKINS, 2025c).
Atenções: maior esforço inicial e manutenção contínua (JENKINS, 2025b).

g) **Monorepo e orquestrações complexas (múltiplos estágios e nós dedicados).**
Recomendação: **Jenkins**.
Por quê: apresenta maior flexibilidade para gerenciar fluxos de automação amplos e integrados, sendo mais adequado a projetos que concentram múltiplos módulos ou serviços em um único repositório. Sua estrutura permite controlar dependências, definir condições de execução e adaptar o processo de integração a diferentes necessidades (JENKINS, 2025a; JENKINS, 2025c).
Atenções: requer maior planejamento inicial e gestão contínua para manter a consistência dos fluxos e a estabilidade do ambiente de execução (JENKINS, 2025b).

h) **Execução em rede restrita, On-premises ou Ambiente *air-gapped*.**
Recomendação: **Jenkins**.
Por quê: viabiliza operação totalmente offline (controller/agentes), além de execução em nós segregados por rótulos sem dependência de serviços externos (JENKINS, 2024b; JENKINS, 2025b).

5.5.2 Cenários acadêmicos

i) **Disciplinas de graduação (laboratórios de CI/CD).**
Recomendação: **GitHub Actions**.

Por quê: contas estudantis disponíveis, workflows simples e rápida curva de aprendizagem.
Atenções: monitorar limites de minutos; fornecer templates base.

j) **Projetos abertos colaborativos (alunos e comunidade).**
Recomendação: **GitHub Actions** ou **GitLab CI/CD**.
Por quê: ambas as plataformas possuem sistemas de controle de versão integrados, o que facilita a colaboração em projetos públicos, a revisão de código por e uso de templates reutilizáveis.
Atenções: definir políticas adequadas de segredos.

5.5.3 Checklist decisório simplificado

- **Onde está seu código?** GitHub ⇒ GitHub Actions; GitLab ⇒ GitLab CI/CD; múltiplos SCMs/legados ⇒ Jenkins.
- **Há requisitos regulatórios ou de residência de dados?** Sim ⇒ Jenkins.
- **Complexidade de orquestração?** Alta/altíssima ⇒ Jenkins; baixa/média ⇒ GitHub Actions/GitLab CI/CD.
- **Orçamento/competências operacionais?** Sem time de plataforma ⇒ GitHub Actions/GitLab CI/CD; com equipe e infraestrutura ⇒ Jenkins.
- **Custo e modelo de cobrança?** Pagar por uso e escalar sob demanda ⇒ GitHub Actions/GitLab CI/CD; aproveitar *hardware* próprio e custos previsíveis ⇒ Jenkins.
- **Governança e auditoria centralizadas?** Regras por grupos/projetos, permissões e trilhas de auditoria integradas ⇒ GitLab CI/CD.
- **Independência de fornecedor e portabilidade?** Evitar acoplamento a um provedor específico ⇒ Jenkins.
- **Estrutura de repositórios?** Muitos repositórios pequenos ⇒ GitHub Actions/GitLab CI/CD; monorepositório com regras complexas ⇒ Jenkins.
- **Experiência prévia da equipe?** Adotar a ferramenta já dominada reduz curva de aprendizado e riscos de implantação.

6 CONCLUSÃO

Este trabalho comparou três plataformas de integração e entrega contínua: GitHub Actions, GitLab CI/CD e Jenkins, com base em objetivos definidos no início do estudo: avaliar e comparar suas funcionalidades, identificar características e limitações e oferecer recomendações de uso em diferentes contextos. Para garantir uma comparação justa, estabeleceu-se um mesmo fluxo de CI/CD (pipeline) para as três ferramentas, envolvendo restauração de dependências, compilação, testes automatizados, construção de imagem, verificação de vulnerabilidades e publicação. Os experimentos foram conduzidos com versões fixadas, repetição de execuções e controle de variáveis operacionais (Cap. 4).

Como sintetizado na Tabela 2, os resultados mostram um equilíbrio entre rapidez de adoção e controle da infraestrutura. Em termos práticos, o Jenkins obteve o melhor tempo mediano de execução (48 s), favorecido pela instalação própria e pelo controle do ambiente; contudo, exige maior esforço de implantação e manutenção, como seleção de extensões, atualização de agentes e cuidados de segurança. O GitHub Actions apresentou tempos intermediários (mediana de 67 s) e se destacou pela facilidade de configuração e pela forte integração com revisões de código e automações usuais do desenvolvimento, o que beneficia equipes que buscam acelerar a adoção com pouca carga administrativa. O GitLab CI/CD mostrou alta estabilidade e integração consistente de repositório, ainda que com maior latência média nas execuções compartilhadas (mediana de 125 s); é especialmente adequado quando a organização adota o ecossistema do GitLab de forma abrangente ou precisa de implantação própria com trilhas de auditoria e políticas de acesso mais rígidas. As análises quantitativas (Tabela 2) e qualitativas (Tabela 3), combinadas à análise funcional (Tabela 1), sustentam essas conclusões.

Em ambientes hospedados, variações naturais de capacidade foram atenuadas com um número fixo de execuções por ferramenta e uso da mediana como medida-resumo. No Jenkins, a compatibilidade de extensões foi tratada com fixação de versões e testes graduais. A gestão de credenciais para publicação de imagens e acesso a repositórios foi padronizada por variáveis protegidas e revisão das permissões mínimas necessárias. Essas medidas reforçam a reprodutibilidade e a transparência do procedimento experimental.

Quanto às contribuições, o trabalho oferece uma base de comparação clara e reproduzível, apoiada em dados e em um desenho metodológico controlado, útil tanto para a comunidade acadêmica quanto para equipes profissionais. As recomendações por cenário (Seção 5.5) transformam os achados em orientações aplicáveis, reduzindo o custo de experimentação para escolher a ferramenta mais adequada a cada contexto. Entre as limitações, reconhece-se o recorte de plataformas analisadas, a influência do perfil de carga escolhido (linguagem, tamanho de imagem, etapas do fluxo) e o fato de a facilidade de configuração ter sido avaliada qualitativamente, sem cronometragem do tempo de implantação inicial.

Com base no referencial teórico e nos resultados obtidos, considera-se que os objetivos foram alcançados. O objetivo geral de avaliar e comparar as três ferramentas foi atendido pelas

análises funcional, quantitativa e qualitativa. Os objetivos específicos também foram contemplados: as principais características e limitações foram identificadas e discutidas; foram propostas recomendações práticas por cenário; e a facilidade de configuração foi avaliada de forma estruturada, ainda que qualitativa.

Em síntese, não há uma solução universalmente superior: a escolha deve ser guiada pelos requisitos de cada projeto. Em ambientes que exigem controle total e fluxos complexos, o Jenkins tende a ser mais indicado; quando a prioridade é iniciar rapidamente, padronizar práticas e reduzir a operação da infraestrutura, GitHub Actions e GitLab CI/CD mostram resultados mais favoráveis. Além disso, o Jenkins é especialmente adequado quando a organização utiliza múltiplos provedores de repositório de código (por exemplo, GitHub, GitLab e *Bitbucket*), graças à sua independência de plataforma e ampla oferta de extensões. O GitHub Actions se destaca quando o código está hospedado no próprio GitHub, aproveitando integrações nativas de revisão e publicação de pacotes. Já o GitLab CI/CD é a escolha natural quando o GitLab é o repositório principal da organização ou quando se deseja orquestrar pipelines a partir de outros provedores, mantendo a governança no GitLab.

Como desdobramentos futuros, recomenda-se ampliar o escopo para outras plataformas, comparar agentes dedicados com agentes compartilhados e aprofundar a análise do impacto de estratégias de cache e de verificações adicionais de segurança. Também é pertinente estimar o custo total de propriedade em diferentes perfis de equipe e de projeto. Esses passos podem fortalecer ainda mais a utilidade prática das conclusões e apoiar decisões mais precisas em ambientes acadêmicos e profissionais.

REFERÊNCIAS

AMAZON WEB SERVICES. **O que é o DevOps?** 2024. https://aws.amazon.com/pt/devops/what-is-devops/. Acesso em: 10 out. 2024.

AMAZON WEB SERVICES. **AWS, O que significa distribuição contínua?** 2025. https://aws.amazon.com/pt/devops/continuous-delivery/. Acesso em: 1 fev. 2024.

AMAZON WEB SERVICES. **AWS, O que significa integração contínua?** 2025. https://aws.amazon.com/pt/devops/continuous-integration/. Acesso em: 1 fev. 2024.

ATLASSIAN. **The DevOps lifecycle**. 2025. https://www.atlassian.com/devops. Acesso em: 1 abr. 2025.

BASS, L.; WEBER, I.; ZHU, L. **DevOps: A Software Architect's Perspective**. [*S.l.*]: Addison-Wesley Professional, 2020.

BLUELIGHT.CO. **Best CI/CD Tools**. 2025. https://bluelight.co/blog/best-ci-cd-tools. Acesso em: 10 jan. 2025.

DOCKER. **docker build CLI reference**. 2025. https://docs.docker.com/reference/cli/docker/build/. Acesso em: 11 set. 2025.

DOCKER. **Docker Documentation**. 2025. https://docs.docker.com/. Acesso em: 11 set. 2025.

DOCKER. **Docker Hub Documentation**. 2025. https://docs.docker.com/docker-hub/. Acesso em: 11 set. 2025.

DUVALL, P. M.; MATYAS, S.; GLOVER, A. **Continuous Integration: Improving Software Quality and Reducing Risk**. [*S.l.*]: Addison-Wesley Professional, 2007.

FOWLER, M. **Continuous Integration**. 2024. https://martinfowler.com/articles/continuousIntegration.html. Acesso em: 1 fev. 2025.

GITHUB. **About billing for GitHub Actions**. 2025. https://docs.github.com/actions/learn-github-actions/usage-limits-billing-and-administration. Acesso em: 11 set. 2025.

GITHUB. **About Docker container support in GitHub Packages (GHCR)**. 2025. https://docs.github.com/packages/working-with-a-github-packages-registry/working-with-the-container-registry. Acesso em: 11 set. 2025.

GITHUB. **About GitHub hosted runners**. 2025. https://docs.github.com/actions/using-github-hosted-runners/about-github-hosted-runners. Acesso em: 11 set. 2025.

GITHUB. **Reusing workflows**. 2025. https://docs.github.com/actions/using-workflows/reusing-workflows. Acesso em: 11 set. 2025.

GITHUB. **Specifications for GitHub hosted runners**. 2025. https://docs.github.com/actions/using-github-hosted-runners/specifications-for-github-hosted-runners. Acesso em: 11 set. 2025.

GITHUB. **Using secrets in GitHub Actions**. 2025. https://docs.github.com/actions/security-guides/using-secrets-in-github-actions. Acesso em: 11 set. 2025.

GITHUB, INC. **Getting Started with GitHub Actions**. 2024. https://github.com/features/actions/getting-started. Acesso em: 1 set. 2024.

GITHUB, INC. **GitHub Actions Overview**. 2024. https://resources.github.com/devops/tools/automation/actions/. Acesso em: 7 set. 2024.

GITHUB, INC. **GitHub Marketplace**. 2024. https://github.com/marketplace. Acesso em: 8 set. 2025.

GITHUB, INC. **GitHub Actions**. 2025. https://github.com/. Acesso em: 8 set. 2025.

GITHUB, INC. **Using workflow templates**. 2025. https://docs.github.com/en/actions/writing-workflows/using-workflow-templates. Acesso em: 4 abr. 2025.

GITLAB. **Compute minutes for GitLab CI/CD**. 2025. https://docs.gitlab.com/ee/ci/pipelines/compute_minutes/. Acesso em: 11 set. 2025.

GITLAB. **GitLab Container Registry**. 2025. https://docs.gitlab.com/ee/user/packages/container_registry/. Acesso em: 11 set. 2025.

GITLAB. **GitLab Runners and Shared Runners**. 2025. https://docs.gitlab.com/ee/ci/runners/. Acesso em: 11 set. 2025.

GITLAB. **Issues: Plan and track work**. 2025. https://docs.gitlab.com/ee/user/project/issues/. Acesso em: 11 set. 2025.

GITLAB. **Merge requests**. 2025. https://docs.gitlab.com/ee/user/project/merge_requests/. Acesso em: 11 set. 2025.

GITLAB, INC. **Get started with GitLab CI/CD**. 2024. https://docs.gitlab.com/ee/ci/. Acesso em: 3 set. 2024.

GITLAB, INC. **CI/CD steps**. 2025. https://docs.gitlab.com/ci/steps/. Acesso em: 1 mar. 2025.

GITLAB, INC. **GitLab CI/CD**. 2025. https://about.gitlab.com/. Acesso em: 8 set. 2025.

GITLAB, INC. **GitLab Runners**. 2025. https://docs.gitlab.com/ci/runners/. Acesso em: 1 mar. 2025.

GITLAB, INC. **How to choose the right continuous integration tool**. 2025. https://about.gitlab.com/topics/ci-cd/choose-continuous-integration-tool/. Acesso em: 1 mar. 2025.

GRACIANO, L. P.; KAKINOHANA, G. K. **Avaliações de soluções CI/CD: Uma análise comparativa entre GitHub Actions e GitLab CI**. [*S.l.*]: Fundação Universidade Federal de Mato Grosso do Sul, 2023.

HUMBLE, J.; FARLEY, D. **Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation**. [*S.l.*]: Addison-Wesley Professional, 2010.

JENKINS. **Suporte de plugins do Jenkins**. 2024. https://plugins.jenkins.io/. Acesso em: 10 out. 2024.

JENKINS. **Using Jenkins agents**. 2024. https://www.jenkins.io/doc/book/using/using-agents/. Acesso em: 12 mar. 2025.

JENKINS. **What is Jenkins Pipeline?** 2024. https://www.jenkins.io/doc/book/pipeline/. Acesso em: 12 mar. 2025.

JENKINS. **About Jenkins: Open Source Automation Server**. 2025. https://www.jenkins.io/about/. Acesso em: 11 set. 2025.

JENKINS. **Extending with Shared Libraries**. 2025. https://www.jenkins.io/doc/book/pipeline/shared-libraries/. Acesso em: 12 mar. 2025.

JENKINS. **Jenkins**. 2025. https://www.jenkins.io/. Acesso em: 8 set. 2025.

JENKINS. **Jenkins Core: archiveArtifacts**. 2025. https://www.jenkins.io/doc/pipeline/steps/core/. Acesso em: 12 mar. 2025.

JENKINS. **Jenkins Docker image jenkins/jenkins**. 2025. https://hub.docker.com/r/jenkins/jenkins. Acesso em: 11 set. 2025.

JENKINS. **Jenkins Documentation**. 2025. https://www.jenkins.io/doc/. Acesso em: 11 set. 2025.

JENKINS. **Jenkins: Downloading and running Jenkins in Docker**. 2025. https://www.jenkins.io/doc/book/installing/docker/. Acesso em: 20 mar. 2025.

JENKINS. **Jenkins Pipeline documentation**. 2025. https://www.jenkins.io/doc/book/pipeline/. Acesso em: 11 set. 2025.

JENKINS. **Managing Plugins (offline installation and updates)**. 2025. https://www.jenkins.io/doc/book/managing/plugins/. Acesso em: 11 set. 2025.

JENKINS. **Pipeline: Shared Libraries**. 2025. https://www.jenkins.io/doc/book/pipeline/shared-libraries/. Acesso em: 11 set. 2025.

JETBRAINS. **Developer Ecosystem Survey**. 2023. https://www.jetbrains.com/lp/devecosystem-2023/. Acesso em: 15 jan. 2025.

LIMA, P. H. O. **Estudo comparativo sobre as funcionalidades de integração contínua e entrega contínua das ferramentas Jenkins e GitLab**. [*S.l.*]: Repositório Institucional Unichristus, 2021.

MICROSOFT. **dotnet build**. 2025. https://learn.microsoft.com/dotnet/core/tools/dotnet-build. Acesso em: 11 set. 2025.

MICROSOFT. **dotnet publish**. 2025. https://learn.microsoft.com/dotnet/core/tools/dotnet-publish. Acesso em: 11 set. 2025.

MICROSOFT. **dotnet restore**. 2025. https://learn.microsoft.com/dotnet/core/tools/dotnet-restore. Acesso em: 11 set. 2025.

MICROSOFT. **dotnet test**. 2025. https://learn.microsoft.com/dotnet/core/tools/dotnet-test. Acesso em: 11 set. 2025.

MICROSOFT. **.NET CLI tools**. 2025. https://learn.microsoft.com/dotnet/core/tools/. Acesso em: 11 set. 2025.

MICROSOFT. **.NET documentation**. 2025. https://learn.microsoft.com/dotnet/. Acesso em: 11 set. 2025.

MICROSOFT. **Overview of NuGet**. 2025. https://learn.microsoft.com/nuget/what-is-nuget. Acesso em: 11 set. 2025.

PALO ALTO NETWORKS. **What Is CI/CD Security?** 2025. https://www.paloaltonetworks.com/cyberpedia/what-is-ci-cd-security. Acesso em: 15 mai. 2025.

RED HAT. **O que é YAML?** 2023. https://www.redhat.com/pt-br/topics/automation/what-is-yaml. Acesso em: 1 fev. 2025.

RED HAT. **O que é CI/CD?** 2025. https://www.redhat.com/pt-br/topics/devops/what-is-ci-cd. Acesso em: 10 jan. 2025.

RIUNGU-KALLIOSAARI, L.; MÄKINEN, S.; TYRVÄINEN, P. Devops adoption benefits and challenges in practice: A case study. **Software: Practice and Experience**, v. 46, n. 6, p. 779–799, 2016.

SAMANTHA, S. A survey on challenges and benefits towards the adoption of DevOps approach. **International Journal for Research in Applied Science and Engineering Technology (IJRASET)**, p. 02–06, 2018.

SCM, G. **Git Reference Manual**. 2025. https://git-scm.com/docs. Acesso em: 11 set. 2025.

SHAHIN, M.; BABAR, M. A.; ZHU, L. Continuous integration, delivery and deployment: A systematic review on approaches, tools, challenges and practices. **IEEE Access**, v. 5, p. 3909–3943, 2017.

TRIVY. **Trivy Documentation**. 2025. https://aquasecurity.github.io/trivy/latest/. Acesso em: 11 set. 2025.

TRIVY. **Trivy image scanning usage**. 2025. https://aquasecurity.github.io/trivy/latest/getting-started/usage/. Acesso em: 11 set. 2025.

XUNIT.NET. **xUnit documentation**. 2025. https://xunit.net/. Acesso em: 11 set. 2025.

YAML. **YAML Specification Version 1.2.2**. 2021. https://yaml.org/spec/1.2.2/. Acesso em: 11 set. 2025.

GLOSSÁRIO

Ambiente *air-gapped* ambiente isolado de redes externas, exigindo espelhamento de dependências, plugins e imagens para operação. 40

API Interface que permite a comunicação entre diferentes sistemas, expondo funções ou dados de uma aplicação para outros serviços.. 31

Artefato (*artifacts*) produto gerado por uma etapa (p. ex., pacote, binário, relatórios) e preservado para consumo por estágios posteriores. 50

Cache mecanismo de armazenamento temporário utilizado para reter dados ou resultados de processos já executados, evitando repetições desnecessárias e acelerando novas execuções. Em pipelines de CI/CD, o cache é comumente empregado para manter dependências, bibliotecas e artefatos entre as execuções, reduzindo o tempo total de processamento. 31, veja Cache de dependências, Pipeline & Job.

Cache de dependências mecanismo para reutilizar dependências ou resultados intermediários entre execuções, acelerando pipelines. 50

open source (*código aberto*) modelo de licenciamento e desenvolvimento em que o código-fonte de um software é disponibilizado para uso, estudo, modificação e redistribuição, favorecendo colaboração, auditoria e evolução comunitária.. 22

DevOps abordagem que integra desenvolvimento e operações para automatizar e melhorar a entrega de softwares, com ênfase em colaboração, automação e melhoria contínua. veja Integração contínua (CI) & Entrega contínua (CD).

Docker plataforma amplamente utilizada para criação, empacotamento, distribuição e execução de contêineres; foi a tecnologia adotada para construir e executar contêineres nas pipelines deste estudo. 22, 48, 49, veja Imagem Docker & registro de contêineres (*container registry*).

Container unidade leve e isolada para empacotar e executar aplicações e suas dependências; neste trabalho, os contêineres são construídos e executados principalmente com Docker, o que garante reprodutibilidade entre ambientes. 28, 48–50, veja Docker, Imagem Docker & registro de contêineres (*container registry*).

Imagem Docker modelo imutável com sistema de arquivos e metadados usados para criar e executar um container. 50

Entrega contínua (CD) prática que mantém o software sempre implantável por meio de automações que empacotam, validam e disponibilizam versões com segurança. veja Integração contínua (CI).

Git sistema de controle de versão distribuído que registra o histórico de alterações em arquivos e permite colaboração por meio de ramificações e fusões, com suporte a repositórios locais e remotos. 15

Branch (Ramificação) linha de desenvolvimento independente dentro de um repositório (repo), usada para isolar alterações até que estejam prontas para serem integradas à principal; permite o trabalho simultâneo de múltiplos desenvolvedores e facilita a gestão de versões e revisões de código. 15, 51, veja commit, push & repositório (repo).

Commit registro atômico de alterações no projeto, contendo mensagem, autor, data e identificador (hash), que passa a integrar o histórico do repositório (repo). 9, 49, 51

Push operação que publica os commits locais em um repositório (repo) remoto, atualizando suas referências conforme a política de permissão. 20, 51

repositório (repo) estrutura que armazena o histórico e os objetos do Git (commits, árvores, blobs e tags), além de referências (ramificações e etiquetas) e metadados; pode ser local (com diretório de trabalho) ou remoto (*bare*). 15, 49

Groovy linguagem de programação orientada a objetos, é utilizada pelo Jenkins nos arquivos `Jenkinsfile` para definir as pipelines de forma programável, permitindo o uso de estruturas de controle, variáveis e funções personalizadas para criar fluxos de automação flexíveis e adaptáveis às necessidades de cada projeto. 18, veja , Pipeline & Stage.

Integração contínua (CI) prática de integrar alterações de código com alta frequência em um repositório comum, validando-as automaticamente por meio de compilação e testes. veja Entrega contínua (CD).

Kubernetes sistema de orquestração de contêineres de código aberto que automatiza a implantação, o escalonamento, a distribuição de carga e a recuperação de aplicações em *clusters* de máquinas. Utilizando imagens Docker. 22, 50, veja Docker, container & registro de contêineres (*container registry*).

GitHub Marketplace catálogo de ações e integrações reutilizáveis para compor workflows no GitHub Actions. 19

On-premises modelo de execução em infraestrutura própria da organização (datacenter local ou nuvem privada). 40

Pipeline fluxo de etapas automatizadas (compilar, testar, verificar segurança, publicar/implantar) que transforma mudanças de código em artefatos prontos para uso. 48–51, veja workflow, Job & Stage.

Job unidade de execução dentro de um workflow ou pipeline, composta por passos/comandos que rodam em um runner. 19, 49, 50

Step menor unidade de execução dentro de um Job, representando uma ação individual, como executar um comando, rodar um script ou chamar uma *action* específica. Cada step é executado de forma sequencial dentro do Job, contribuindo para a construção das etapas de uma Pipeline. 19, veja Job, Pipeline & workflow.

Stage agrupamento lógico/sequencial de jobs (como build, test, deploy), muito usado no GitLab CI/CD e no Jenkins. 21

build etapa da Pipeline que transforma o código-fonte em artefatos executáveis ou publicáveis (p. ex., binários e Imagem Dockers); normalmente envolve resolução de dependências, execução de testes rápidos e pode empregar Cache de dependências para reduzir o tempo de execução. 9, 18, veja Artefato (*artifacts*), Imagem Docker, Pipeline, Job & Cache de dependências.

deploy etapa da Pipeline que publica e ativa os artefatos no ambiente de destino (desenvolvimento, homologação ou produção), por exemplo ao enviar imagens para um registro de contêineres (*container registry*) e iniciar contêineres ou orquestrá-los com Kubernetes; pode empregar estratégias como atualização gradual e reversão. 9, 18, veja Kubernetes, registro de contêineres (*container registry*), Pipeline, Stage & container.

Workflow definição declarativa do processo automatizado (eventos de disparo, jobs e dependências), comum em plataformas como GitHub Actions. 10, 49, 51

Plugin extensão que adiciona funcionalidades a uma plataforma (amplo ecossistema no Jenkins; integrações oficiais em GitHub/GitLab). 22, 48

Registro de contêineres (*container registry*) serviço utilizado para armazenar, versionar e distribuir imagens de contêineres. Cada registro mantém repositórios com diferentes versões de imagens, permitindo que elas sejam baixadas e executadas em diversos ambientes. Exemplos comuns incluem Docker Hub, GitHub Container Registry (GHCR) e GitLab Container Registry. 32, 50, veja Docker, Docker Hub & container.

Docker Hub registro público/privado de imagens Docker mantido pela Docker Inc.. 27

Runner agente de execução que processa jobs de uma Pipeline; pode ser hospedado pelo provedor ou pela própria organização. 19, 21, veja & Runner compartilhado (*Shared Runner*).

Runner *self-hosted* agente de execução mantido pela organização, permitindo controle de sistema, rede, dependências e políticas internas. 31

Segredos *(secrets)* credenciais e informações sensíveis (senhas, tokens, chaves) injetadas de forma segura na Pipeline e controladas por políticas de acesso. 32

Shell Script conjunto de comandos escritos em um interpretador de linha de comando, como Bash, utilizado para automatizar tarefas em sistemas operacionais do tipo Unix; permite manipular arquivos, configurar variáveis de ambiente, executar programas e controlar o fluxo das etapas de uma Pipeline. 18, veja Pipeline, YAML & Job.

Software conjunto de programas e componentes digitais que executam funções específicas em computadores e dispositivos. 9, 48

Template modelo reutilizável de configuração, arquivo ou estrutura que acelera a criação de pipelines e workflows; Pode definir passos padrão, variáveis e boas práticas que são herdadas e customizadas. 10, veja Pipeline, workflow & YAML.

Trigger evento que aciona automaticamente a execução de uma Pipeline ou workflow. Em ferramentas de CI/CD, os triggers podem ser configurados para responder a ações específicas, como um commit, um push, a criação de uma Branch (Ramificação) ou a abertura de uma requisição de mesclagem, iniciando assim os processos automatizados definidos no projeto. 19, veja workflow, Job & Step.

Webhook Mecanismo de integração que permite que um sistema envie automaticamente uma requisição *HTTP* para outro sistema quando um evento ocorre, transmitindo informações em tempo real sem necessidade de consulta periódica.. 35

YAML linguagem de serialização legível por humanos, amplamente usada para descrever configurações de workflows e pipelines. 17, veja Pipeline & workflow.