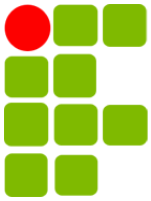


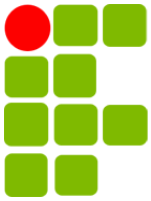
Instituto Federal de Educação, Ciência e Tecnologia da Paraíba
Campus Campina Grande
Coordenação do Curso Superior de Tecnologia em Telemática

COMPREENSÃO DE *PIPELINES* DE CI ATRAVÉS DE UMA IMPLEMENTAÇÃO PRÁTICA: UM GUIA PARA INICIANTES

VIVIAN GABRIELA DA SILVA CLAUDINO

Orientador: Elaine Cristina Juvino de Araújo

Campina Grande, Dezembro de 2025



Instituto Federal de Educação, Ciência e Tecnologia da Paraíba
Campus Campina Grande
Coordenação do Cursos Superior de Tecnologia em Telemática

COMPREENSÃO DE *PIPELINES* DE CI ATRAVÉS DE UMA IMPLEMENTAÇÃO PRÁTICA: UM GUIA PARA INICIANTES

VIVIAN GABRIELA DA SILVA CLAUDINO

Monografia apresentada à Coordenação do Curso de Telemática do IFPB - Campus Campina Grande, como requisito parcial para conclusão do curso de Tecnologia em Telemática.

Orientador: Elaine Cristina Juvino de Araújo

Campina Grande, Dezembro de 2025

Catalogação na fonte:
Ficha catalográfica

C615c Claudino, Vivian Gabriela da Silva.
Compreensão de pipelines de CI através de uma implementação prática: um guia para iniciantes / Vivian Gabriela da Silva Claudino. – Campina Grande, 2025.
43f.: il.

Trabalho de Conclusão de Curso (Graduação em Tecnologia em Telemática) - Instituto Federal da Paraíba, 2025.
Orientadora: Dra. Elaine Cristina Juvino de Araújo.

1. Telemática - Automatização de Software . 2. Integração Contínua (CI). 3. DevOps. 4. Pipeline. 5.GitHub Actions. I. Araújo, Elaine Cristina Juvino de. II. Título.

CDU 004.41:004.05

COMPREENSÃO DE *PIPELINES* DE CI ATRAVÉS DE UMA IMPLEMENTAÇÃO PRÁTICA: UM GUIA PARA INICIANTES

VIVIAN GABRIELA DA SILVA CLAUDINO

Elaine Cristina Juvino de Araújo
Orientador

Anderson Fabiano Batista Ferreira da Costa
Membro da Banca

Katyusco de Farias Santos
Membro da Banca

Campina Grande, Paraíba, Brasil
Dezembro/2025

Agradecimentos

Agradeço primeiramente a Deus, por me conceder força, sabedoria e a oportunidade de iniciar e concluir esta jornada.

Agradeço à minha mãe e ao meu padrasto, pelo apoio incondicional, especialmente nos momentos em que precisei conciliar o trabalho com a faculdade.

Agradeço também ao meu noivo, que esteve ao meu lado durante toda essa trajetória, oferecendo apoio, incentivo e compreensão nos momentos mais desafiadores.

Manifesto também minha gratidão à minha orientadora, Prof.ª Elaine Cristina Juvino de Araújo, pela dedicação, paciência e orientação durante o desenvolvimento deste Trabalho de Conclusão de Curso.

Agradeço aos membros da banca examinadora, Prof. Anderson Fabiano Batista Ferreira da Costa e Prof. Katyusco de Farias Santos, por gentilmente aceitarem o convite e contribuírem com suas valiosas observações.

Por fim, estendo meus agradecimentos a todos os professores do Instituto Federal da Paraíba (IFPB), pelos ensinamentos compartilhados e pelo apoio constante ao longo da minha trajetória acadêmica.

Resumo

O presente trabalho aborda a necessidade de guias práticos e didáticos para iniciantes, focados no funcionamento e na implementação de um *pipeline* de Integração Contínua (CI) no domínio do desenvolvimento de *software*. Através de um estudo teórico exploratório, identificou-se uma lacuna de materiais didáticos que demonstrem o funcionamento de um *pipeline* de Integração Contínua.

Para comprovar e quantificar essa lacuna, a metodologia adaptou os princípios do Protocolo *PRISMA* (*Preferred Reporting Items for Systematic Reviews and Meta-Analyses*), utilizando um critério de inclusão rigoroso que demonstrou a escassez de guias práticos sobre o cenário proposto na literatura. A fase prática consistiu no desenvolvimento de um *pipeline* de CI funcional, utilizando a linguagem *Python*, a biblioteca de testes *Pytest*, e o *CircleCI* como servidor de Integração Contínua.

Como resultado, o trabalho detalha as etapas de implementação, incluindo a solução para desafios práticos encontrados durante a implementação do *pipeline*, como o erro de autenticação SSH (*Checkout code*) e o gerenciamento de dependências. Demonstrou-se, ainda, a viabilidade e a importância da proteção do repositório (*Branch Protection*) para garantir que apenas código validado pelo *pipeline* possa ser integrado à *branch* principal.

Em conclusão, este trabalho oferece um Guia Prático e Replicável que preenche a lacuna identificada, fornecendo um cenário de Integração Contínua robusto e automatizado. O *pipeline* desenvolvido serve como um ativo reutilizável para iniciantes e como base para a expansão de cenários de Entrega Contínua (CD).

Palavras-chave: Integração Contínua; *DevOps*; *Python*; *Pytest*; *CircleCI*; Metodologia *PRISMA*; Guia Prático.

Abstract

This study addresses the need for practical and didactic guides for beginners, focusing on the functionality and implementation of a Continuous Integration (CI) pipeline within software development. Through a theoretical-exploratory study, a scarcity of didactic materials that demonstrate the functioning of a complete Continuous Integration pipeline was identified.

To prove and quantify this gap, the methodology adapted the principles of the PRISMA Protocol (Preferred Reporting Items for Systematic Reviews and Meta-Analyses), utilizing a rigorous inclusion criterion that demonstrated the scarcity of practical guides on the proposed scenario in the literature. The practical phase consisted of developing a functional CI pipeline, using the Python language, the Pytest testing library, and CircleCI as the Continuous Integration server.

As a result, the work details the implementation steps, including solutions to practical challenges encountered during pipeline implementation, such as the SSH authentication error (Checkout code) and dependency management. Furthermore, the viability and importance of repository protection (Branch Protection) were demonstrated to ensure that only code validated by the CI pipeline can be integrated into the main branch.

In conclusion, this work offers a Practical and Replicable Guide that fills the identified gap, providing a robust and automated Continuous Integration scenario. The developed pipeline serves as a reusable asset for beginners and as a foundation for the expansion of Continuous Delivery (CD) scenarios.

Keywords: Continuous Integration; DevOps; Python; Pytest; CircleCI; PRISMA Methodology; Practical Guide.

Sumário

Lista de Abreviaturas

CI	*Continuous Integration*
CD	*Continuous Delivery*
PR	*Pull Request*
VSC	*Version Control System* (Sistema de Controle de Versão)
IDE	*Integrated Development Environment*(Ambiente de Desenvolvimento Integrado)
PRISMA	*Preferred Reporting Items for Systematic Reviews and Meta-Analyses*
SSH	*Secure Shell*

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

A crescente demanda por ciclos de desenvolvimento mais rápidos e a necessidade de entregar *software* de alta qualidade de forma contínua têm impulsionado a evolução das práticas de engenharia de *software*. Diante desse cenário, práticas como a *Continuous Integration* (CI) e a *Continuous Delivery* (CD) tornaram-se pilares essenciais da cultura *DevOps*, que vem da junção das palavras "Desenvolvimento"e "Operações". Esse termo representa uma forma de trabalho que aproxima as pessoas que desenvolvem os programas das pessoas que cuidam para que esses programas funcionem bem todos os dias [Benjamin e Mathew 2021; Bernardo *et al.* 2023].

Juntas, essas abordagens permitem automatizar etapas críticas do desenvolvimento, como compilação, teste e implantação, reduzindo erros e aumentando significativamente a produtividade das equipes.

Apesar da relevância do tema, muitas pessoas iniciantes na área ainda têm dificuldade em compreender como o CI e o CD funcionam. A motivação para este trabalho surge, portanto, da necessidade de tornar esse conhecimento mais acessível. O objetivo é fornecer explicações claras e um estudo detalhado que desvende o funcionamento desses processos em um ambiente real.

Para fins deste trabalho, o público iniciante é definido como profissionais ou estudantes de desenvolvimento de *software* que possuem conhecimento básico em programação e em sistemas de controle de versão (como *Git*), mas que estão tendo o primeiro contato formal ou prático com a cultura *DevOps* e com a implementação de *pipelines* de Integração Contínua.

Dessa forma, este trabalho possui um caráter teórico exploratório, concentrando-se inicialmente em desvendar os conceitos essenciais do CI/CD e do ecossistema *Git*. A partir dessa fundamentação, o estudo avança para um caráter prático, onde o objetivo final é a análise do fluxo de um *pipeline* de CI desenvolvido e executado em um ambiente real. Esta abordagem focada na implementação das etapas automatizadas demonstra como as alterações de código são efetivamente transformadas em um produto funcional.

1.1 Justificativa e Relevância do Trabalho

Embora a literatura sobre Integração Contínua e Entrega Contínua seja vasta, grande parte do material disponível aprofunda-se em análises de desempenho, segurança ou em estudos de caso complexos, como os estudos de [Benjamin e Mathew 2021], [Bernardo *et al.* 2023], [Bounegru 2024], [Donca *et al.* 2022], [Laukkanen, Itkonen e Lassenius 2017], [Lima e Vergilio 2023], entre outros. Essa abordagem, embora valiosa, frequentemente pressupõe um conhecimento prévio dos conceitos fundamentais da área, deixando de lado a definição de termos e fluxos essenciais. Diante disso, este trabalho se justifica pela necessidade de tornar o conhecimento sobre o funcionamento de um *pipeline* de CI mais acessível a um público com menos experiência, fornecendo uma base sólida para a compreensão do tema.

1.2 Objetivos

Nesta seção, serão apresentados os objetivos do trabalho, que guiarão a metodologia e o desenvolvimento da pesquisa.

1.2.1 Objetivo Geral

Promover a compreensão de CI e CD para um público iniciante em *DevOps* através de um *pipeline* de integração contínua, detalhando sua operação e os principais elementos no contexto de repositórios *Git*.

1.2.2 Objetivos Específicos

- Realizar uma revisão da literatura, para identificar possíveis lacunas na oferta de materiais para iniciantes no âmbito de CI e CD.
- Discutir os principais conceitos de CI, CD e *pipeline*.
- Analisar o funcionamento dos termos essenciais do *Git*, como *commit*, *branch* e *pull request*.
- Implementar e analisar um *pipeline* de CI, utilizando o *GitHub* para versionamento, o *CircleCI* para automação e o *VS Code* como editor de código, demonstrando cada etapa do fluxo de trabalho.
- Demonstrar a execução de testes unitários dentro do ambiente de CI.

1.3 Organização do Documento

Este documento está estruturado em cinco capítulos.

No Capítulo 2, é apresentada a fundamentação teórica, discutindo os conceitos essenciais da Integração Contínua (CI), da Entrega Contínua (CD) e do *pipeline* de CI, bem como os termos e fluxos de trabalho do *Git* que são fundamentais para a compreensão do processo.

Em seguida, o Capítulo 3 detalha a metodologia do trabalho, que se divide em duas fases principais: a validação da lacuna na literatura sobre guias práticos para iniciantes e o plano de Implementação Prática, que define o estudo de caso, as ferramentas utilizadas e o escopo do guia.

Por fim, o Capítulo 4 é dedicado à implementação e análise prática do fluxo de um *pipeline* de CI, sendo estruturado da seguinte forma: primeiramente, ele descreve os arquivos de configuração e os arquivos do projeto que compõem a arquitetura do *pipeline*. Em seguida, detalha o passo a passo da implementação e da configuração das ferramentas (*GitHub* e *CircleCI*). Por fim, o capítulo analisa o fluxo do *pipeline* em cenários práticos, demonstrando casos de sucesso e de falha.

O Capítulo 5 apresenta as considerações finais do trabalho, com a síntese das conclusões obtidas e sugestões para estudos futuros.

Capítulo 2

Fundamentação Teórica

Nesta seção, exploraremos os conceitos fundamentais que sustentam este trabalho, com foco na Integração Contínua e na Entrega Contínua. Detalharemos as principais ideias e definições, como o funcionamento de um *pipeline* de automação, a diferença entre os dois conceitos e as ferramentas utilizadas.

2.1 Integração Contínua - CI

De acordo com [Bernardo *et al.* 2023], a Integração Contínua é uma prática utilizada no desenvolvimento de *software*, onde os programadores adicionam novos trechos de código a um sistema de forma frequente e automatizada. Essa abordagem é adotada por muitas organizações para tornar a evolução de *software* mais econômica e confiável, permitindo que o *software* seja alterado, construído e testado diversas vezes em um curto período [Lima e Vergilio 2023].

Para [Benjamin e Mathew 2021; Laukkanen, Itkonen e Lassenius 2017], uma boa prática de CI exige que todos os desenvolvedores integrem seu trabalho a um repositório de código comum diariamente. Eles defendem que, após cada integração, o sistema deve ser construído e testado para garantir que o sistema permaneça funcional e seguro para outras alterações.

Para os desenvolvedores, a adoção da CI tem um impacto positivo, pois melhora a qualidade do projeto, a automação e o processo de lançamento. A integração contínua aumenta a qualidade e a estabilidade do código, deixando os desenvolvedores mais confiantes para fazer novos lançamentos. Essa confiança é fomentada principalmente por testes automatizados abrangentes, especialmente quando a cobertura do código é alta [Bernardo *et al.* 2023].

2.2 Entrega Contínua - CD

A Entrega Contínua (CD) é uma abordagem de engenharia de *software* na qual as equipes produzem *software* em ciclos curtos e se asseguram de que ele pode ser lançado de forma confiável a qualquer momento [Chen 2017]. Para isso, o processo envolve preparar e testar

constantemente as mudanças feitas no *software* para que elas possam ser entregues ao usuário o mais rápido possível [Bernardo *et al.* 2023].

Essa prática é especialmente útil em projetos que precisam de atualizações frequentes, garantindo que os usuários tenham sempre a versão mais atual do *software*. Empresas que praticam a CD relataram enormes benefícios, como melhorias significativas no tempo de lançamento no mercado, na satisfação do cliente, na qualidade do produto, na confiabilidade do lançamento e na produtividade e eficiência da equipe [Bernardo *et al.* 2023].

2.3 *DevOps*

O termo *DevOps* vem da junção das palavras "Desenvolvimento"e "Operações" [Benjamin e Mathew 2021; Bernardo *et al.* 2023]. Ele representa uma forma de trabalho que aproxima as pessoas que desenvolvem os programas (os desenvolvedores) das pessoas que cuidam para que esses programas funcionem bem todos os dias (a equipe de operações). O principal objetivo do *DevOps* é melhorar a colaboração entre essas equipes, para que o *software* seja entregue de forma mais rápida, com mais qualidade e menos erros.

A adoção da prática *DevOps* tem um impacto significativo na qualidade do *software*, com a automação, colaboração e *feedback* rápido sendo os principais recursos que contribuem para essa melhoria. Essa colaboração, obtida por meio da prática de *DevOps*, ajuda a criar um ambiente de trabalho mais positivo e produtivo [Benjamin e Mathew 2021].

2.4 *Pipeline*

Segundo [Bernardo *et al.* 2023], dentro do processo de desenvolvimento de *software* moderno, um *pipeline* é uma espécie de linha de montagem automatizada. Ele organiza todas as etapas que acontecem desde o momento em que um programador escreve um código até esse código estar pronto para ser usado por outras pessoas. Essa linha de montagem geralmente inclui passos como:

- Testar automaticamente se o código está funcionando;
- Juntar esse código com os outros pedaços já prontos (Integração Contínua);
- Enviar as atualizações para o ambiente de produção, onde o programa fica disponível para os usuários (Entrega Contínua).

Podemos visualizar o fluxo de um *pipeline* de Integração Contínua na Figura 2.1, que ilustra o processo desde o momento de uma alteração no código até a obtenção de resultados. Esse cenário está bem simplificado, demonstrando os desenvolvedores trabalhando em um ambiente local e enviando suas alterações ao repositório central.

A Figura 2.1 inicia com o *commit*, que representa a ação de registrar uma alteração no Servidor de Controle de Versão. Em seguida, o servidor de Integração Contínua busca as alterações ("*fetch changes*") para iniciar o processo de *Build* e Testes.

A fase de *Build* (construção), é a primeira etapa onde o código é compilado, empacotado e preparado em um formato executável, e a fase de *test* é onde os testes unitários são executados para verificar se o código esta funcionando.

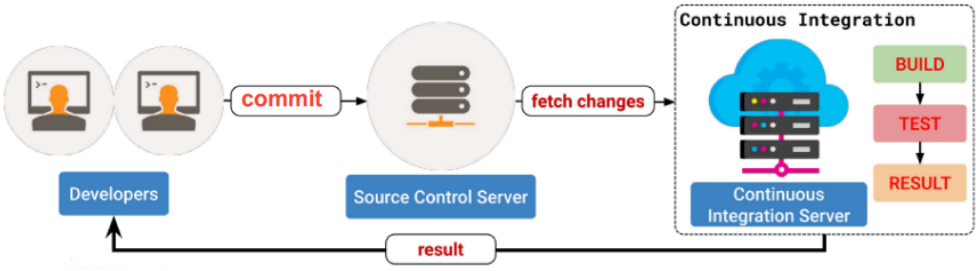


Figura 2.1: *Exemplo de um pipeline de Integração Contínua (Adaptada de [Lima e Vergilio 2023])*

A sua evolução para um *pipeline* de Entrega Contínua, que inclui a implantação em produção, pode ser observada na Figura 2.2. O processo do *pipeline* de CD inicia com uma versão do código que é candidata a lançamento (*latest released code version:* 1.3.5).

Figura 2.2: *Exemplo de um pipeline de Entrega Contínua (Adaptada de [Donca* et al. *2022])*

A partir desse ponto inicial, alterações de correções ou novas funcionalidades (*'patch' changes: fixes, tests*) são mescladas na *branch* principal (*merged into master*), resultando em uma versão candidata a *release* (lançamento), marcada como *1.3.5-rc*. A partir dessa versão candidata, o processo avança pelas etapas essenciais de: *Build* e *Test*.

Somente após a aprovação dessas etapas, a versão é marcada oficialmente como lançamento (*mark version as release:* 1.3.5) e, finalmente, passa para a fase de implantação/distribuição (*deploy*). Isso garante que apenas código estável chegue ao ambiente de produção.

2.5 Repositório

Um repositório é o elemento fundamental do Sistema de Controle de Versão, como o *Git*, e é a base para organizar o código no *GitHub*. É onde todos os arquivos, pastas, imagens, e toda a história de alterações de um projeto específico são armazenados e gerenciados [GitHub 2025]. É semelhante a uma pasta central do projeto, mas armazena também informações como:

- Todo o histórico de revisões do código, permitindo que o desenvolvedor volte a qualquer estado anterior.
- Todas as *branches* que foram criadas para o desenvolvimento.
- Informações sobre quem fez a alteração e quando (*commits*).

O repositório possui algumas terminações essenciais para o fluxo de trabalho colaborativo, sendo elas o repositório local e o remoto [GitHub 2025]. O repositório local é a cópia completa dos dados de um projeto que reside no computador pessoal do desenvolvedor. Ele é estabelecido quando o desenvolvedor realiza um clone, baixando todos os dados do projeto a partir do Repositório Remoto, que é o repositório central de destino armazenado no *GitHub*.

2.5.1 *Branches*

Para que o desenvolvedor possa trabalhar em novas funcionalidades ou correções sem comprometer a estabilidade do repositório principal (remoto), o *Git* utiliza o conceito de ramificações (*Branches*) [GitHub 2025].

As ramificações, ou *branches*, permitem desenvolver novas funcionalidades do projeto, corrigir erros ou experimentar com segurança novas ideias em um ambiente isolado dentro de um repositório. Um novo *branch* é sempre criado a partir de um *branch* existente, permitindo que o desenvolvedor trabalhe em suas alterações de forma isolada do código-fonte principal. Um *branch* criado especificamente para a implementação de uma nova funcionalidade é comumente referido como um *branch* de recurso ou de tópico [GitHub 2025].

2.6 Termos Essenciais

No ambiente de desenvolvimento de *software* utilizando CI/CD, a colaboração e a automação são pilares do processo de entrega. Para que um *pipeline* funcione de forma eficaz, uma série

de termos e conceitos interligados são fundamentais. Eles garantem que as alterações no código sejam rastreadas, revisadas e integradas de forma segura.

2.6.1 *Git* e *Commit*

O *Git* é um sistema de controle de versão, uma ferramenta fundamental que gerencia todos os históricos de alterações no código-fonte [Song e Kim 2018]. Ele permite que múltiplos desenvolvedores façam alterações simultaneamente no mesmo código-fonte sem que haja conflitos. Em outras palavras, é um dispositivo regulatório autônomo, projetado para gerenciar de forma eficaz a participação de muitas pessoas em um projeto.

A unidade básica de trabalho no *Git* é o *commit*. Cada vez que um desenvolvedor salva uma alteração no código-fonte, ele cria um *commit*, que funciona como um registro de todas as modificações. Essas alterações não são aplicadas imediatamente ao repositório principal, mas apenas após passarem por um processo de análise, que é o primeiro passo de um *pipeline* de automação, como os utilizados nas práticas de Integração Contínua [Song e Kim 2018].

O *Git* atribui a cada *commit* um ID exclusivo, que identifica as seguintes informações:

- Cada uma das alterações feitas;
- O momento em que as alterações foram feitas;
- O autor das alterações;
- Descrição do *commit*.

Além disso, ao fazer um *commit*, deve-se incluir uma mensagem que descreva brevemente as alterações feitas, tornando o histórico do projeto mais compreensível e organizado [GitHub 2025].

2.6.2 *Pull Request*

O *pull request* (PR) representa um conjunto de alterações, entre um repositório de código local e o repositório central de destino (remoto). Como o próprio nome sugere, um PR não é uma alteração imediata, mas sim uma solicitação de inclusão de código [Ortu *et al.* 2020]. Por sua natureza colaborativa, os *pull requests* são enviados pelos desenvolvedores para o *GitHub*. E a partir disso, uma discussão é realizada sobre as mudanças e, finalmente, decidem se as alterações serão integradas ao *branch* principal.

Em última análise, um *pull request* pode ser aceito ou rejeitado. A decisão de integrar ou não o *pull request* é tomada por pessoas que atuam como desenvolvedores principais (os que realizam diretamente as alterações na base de código) ou integradores (os responsáveis por mesclar as alterações na base de código) [Ortu *et al.* 2020]. Quando o PR é aceito e integrado ao código, utiliza-se o termo “mesclado” (*merged*).

2.7 Ferramentas utilizadas em CI/CD

Segundo [Bernardo *et al.* 2023], para que práticas como Integração Contínua e Entrega Contínua funcionem corretamente, é essencial utilizar ferramentas específicas que ajudem a automatizar as tarefas do dia a dia no desenvolvimento de *software*.

Essas ferramentas atuam como assistentes digitais, que trabalham junto com os desenvolvedores para realizar várias atividades automaticamente, como: testar se o programa está funcionando, juntar diferentes partes do código, e enviar atualizações para os usuários. Tudo isso sem que uma pessoa precise fazer manualmente cada etapa.

Os autores destacam que essas ferramentas são parte importante do que se chama *pipeline* de *software*, que é como uma linha de montagem já explicada anteriormente. [Bernardo *et al.* 2023], por exemplo, citam plataformas como *Jenkins* (que automatiza testes e a entrega de novas versões) e *Docker* (que organiza o programa para garantir sua correta execução em qualquer servidor).

No entanto, para o cenário de implementação deste trabalho, as ferramentas utilizadas foram distintas e focadas em ambientes de fácil adoção: o *CircleCI*, que atuou como o servidor de Integração Contínua onde o *pipeline* foi executado, e o *GitHub*, que serviu como o Sistema de Controle de Versão e repositório central do código.

Com essas ferramentas, o trabalho fica mais rápido, os erros diminuem e o *software* chega até o usuário final com mais qualidade. Como explicam os autores, elas são fundamentais para que os times consigam aplicar com sucesso os conceitos de CI, CD e *DevOps* no desenvolvimento de sistemas.

2.8 *GitHub* como Plataforma no Desenvolvimento de *Software*

Segundo [Bounegru 2024], o *GitHub*, originalmente conhecido como uma plataforma de hospedagem de código-fonte com sistema de controle de versão (VSC) baseado em *Git*, tornou-se muito mais do que apenas um repositório de arquivos. Conforme analisado por [Bounegru 2024], o *GitHub* passou a ser entendido como uma infraestrutura fundamental no ecossistema de desenvolvimento de *software*, atuando como uma "plataforma conectiva". Isso significa que ele não apenas armazena código, mas conecta pessoas, projetos e ferramentas.

A autora destaca que o *GitHub* desempenha um papel importante na forma como desenvolvedores colaboram, compartilham conhecimento e organizam o trabalho. Essas conexões criadas entre código, pessoas e ferramentas formam o que [Bounegru 2024] chama de "codificação conectiva" — um tipo de prática em que a escrita de código está profundamente integrada a uma rede de interações sociais e tecnológicas.

Além disso, segundo [Bounegru 2024], o *GitHub* incorpora um conjunto de ferramentas automatizadas, como *GitHub Actions*, que permitem implementar práticas de Integra-

ção Contínua e Entrega Contínua diretamente dentro da própria plataforma. Isso torna o processo de desenvolvimento mais fluido, automatizado e transparente. Com essas funcionalidades, desenvolvedores podem configurar testes e validações, sem sair do ambiente da plataforma.

Assim, o *GitHub* deixa de ser apenas um repositório de código e passa a ser um ambiente integrado de desenvolvimento, sendo fundamental para práticas modernas como CI/CD, especialmente por oferecer recursos acessíveis, gratuitos e amplamente documentados.

Capítulo 3

Metodologia

A metodologia empregada para o desenvolvimento deste trabalho fundamenta-se em um estudo de caráter teórico exploratório e, principalmente, prático. O estudo teórico exploratório foi crucial para a identificação e o aprofundamento do problema central (a lacuna de materiais para iniciantes) e a construção da base de conhecimento. Esta fase envolveu a análise de conceitos fundamentais como Integração Contínua (CI), Entrega Contínua (CD) e a cultura *DevOps*. O objetivo foi coletar e organizar o conhecimento necessário para embasar a solução prática.

Por sua vez, o estudo prático concentrou-se na produção de um resultado concreto e utilizável, consistindo na implementação do *pipeline* de CI. Esta etapa prática envolveu a codificação dos arquivos da aplicação (*calculator.py*), dos testes (*testecalculator.py*) e do arquivo de configuração (*config.yml*).

Dessa forma, o trabalho se organizou em duas etapas complementares: a etapa exploratória forneceu o rigor conceitual e o embasamento teórico, enquanto a etapa aplicada demonstrou a aplicação e a viabilidade desses conceitos, resultando na construção do *pipeline* de CI proposto.

1. Etapa teórica exploratória: Revisão da literatura utilizando uma abordagem sistemática adaptada (baseada no protocolo PRISMA - *Preferred Reporting Items for Systematic reviews and Meta-Analyses*) para justificar o tema e comprovar a necessidade de um guia prático para iniciantes. O método PRISMA é uma diretriz criada para aprimorar a apresentação de revisões sistemáticas. Ele auxilia a relatar de forma completa o motivo da realização, os métodos utilizados e os resultados encontrados [PRISMA 2025]. Dessa forma, utilizando esse método é possível comprovar a lacuna identificada.

2. Etapa Prática: Desenvolvimento e detalhamento de um *Pipeline* de Integração Contínua (CI) em ambiente real, resultando no Guia para Iniciantes proposto.

3.1 Etapa teórica exploratória

O trabalho iniciou com uma revisão bibliográfica, cujo objetivo principal foi identificar e validar a lacuna existente na literatura referente à disponibilidade de material didático prático sobre Integração Contínua (CI) voltado para iniciantes em desenvolvimento de *software*.

A pesquisa foi guiada pela seguinte questão: A literatura atual sobre Integração Contínua (CI) e Entrega Contínua (CD) no domínio de desenvolvimento de *software* apresenta guias práticos, tutoriais ou materiais didáticos explícitos para o público iniciante?

Com a questão de pesquisa definida, o passo seguinte foi a definição da estratégia de busca nas bases de dados.

3.1.1 Termos de Busca

A pesquisa foi conduzida utilizando o acesso CAFe do Periódicos Capes como base de dados. As pesquisas foram realizadas definindo o período de 2014 a 2025, para garantir a relevância e abrangência temporal. Os artigos foram filtrados para incluir apenas publicações revisadas por pares e de acesso aberto, sem restrição de idioma.

A Tabela 3.1 representa as estratégias de busca utilizadas para comprovar a lacuna na literatura sobre material didático prático de CI para iniciantes. Foram empregadas três buscas distintas (B1, B2 e B3) para validar a existência da lacuna mencionada:

Estratégia de Busca

Busca	Termos-Chave	Objetivo	Registros Identificados
B1	TÌTULO ("*continuous integration*"*OR* "*continuous delivery*") *AND* QUALQUER CAMPO ("*software*"*AND* "*guide*")	Busca ampla para identificar materiais denominados como guia.	4
B2	TÍTULO ("*continuous integration*"*OR* "*continuous delivery*"*AND* "*software*") *AND* QUALQUER CAMPO "*guide*"	Busca refinada para aprofundar B1, adicionando no título o termo "*software*".	1
B3	TÍTULO ("*continuous integration*"*OR* "*continuous delivery*"*AND* "*software*") *AND* QUALQUER CAMPO "*pipeline*"	Busca especifica sobre *pipeline* no contexto CI/CD para verificar se dessa forma seria apresentado algum estudo prático para iniciantes.	3

Tabela 3.1: *Estratégia de busca para comprovar a lacuna*

Com a busca pelos artigos realizada, vamos agora para a etapa de definição dos critérios de elegibilidade e exclusão.

3.1.2 Critérios de Elegibilidade e Exclusão (CE)

O critério central para a seleção dos artigos foi o Critério de Elegibilidade 1 (CE1), formulado para preencher a questão de pesquisa. Todos os artigos que não atenderam a este critério foram excluídos, comprovando a lacuna.

- Critério de Elegibilidade (CE1): O artigo deve apresentar conteúdo com abordagem didática ou prática, como um guia, tutorial ou passo a passo, focado na implementação e compreensão de *Pipelines* de CI no contexto de Desenvolvimento de *Software*.

Os artigos foram excluídos quando:

- CE2 (Domínio): Não pertenciam ao domínio de Engenharia de *Software* e CI/CD.
- CE3 (Foco Avançado/Teórico): Apresentavam foco exclusivo em aspectos avançados, teóricos ou de gestão, sem a perspectiva de guia prático para iniciantes.

Para garantir o rigor científico da revisão, foi estabelecido o critério de exclusão de toda a literatura cinza, incluindo blogs, vídeos do *YouTube*, tutoriais de *websites* e documentações não oficiais. A pesquisa foi estritamente limitada a fontes primárias de revisão por pares, como artigos publicados em periódicos e anais de conferências.

3.1.3 Processo de Seleção e Resultados

Após realizar as buscas nas bases de dados, foi feita a remoção inicial dos artigos duplicados. A Tabela 3.2 consolida os resultados das estratégias de busca, apresentando a contagem total de registros identificados, a eliminação de duplicatas e a quantidade final de artigos únicos que restaram para a análise.

Resultados das Buscas

Busca	Registros Identificados
B1	4
B2	1
B3	3
Total Identificado	8
Duplicatas	1
Artigos Únicos	7

Tabela 3.2: *Resultados consolidados das estratégias de busca*

A Figura 3.1 apresenta os 7 artigos únicos que foram submetidos à triagem por título e resumo. Nesse processo, dois artigos foram imediatamente excluídos por tratarem de monitoramento contínuo de glicose, sendo classificados no Critério de Exclusão CE2 (Domínio) por fugirem do escopo de desenvolvimento de *software*.

Os 5 artigos restantes foram lidos na íntegra para a aplicação rigorosa do Critério de Elegibilidade CE1 e do Critério de Exclusão CE3 (Foco Avançado/Teórico).

Após aplicar os Critérios de Exclusão, temos como resultado que nenhum dos artigos localizados nessa busca atendem ao requisito de inclusão CE1. Dessa forma, conclui-se que a resposta para a questão de pesquisa é: a literatura atual sobre Integração Contínua (CI) e Entrega Contínua (CD) no domínio de desenvolvimento de *software* não possui nenhum guia prático para o público iniciante.

	Artigo	Busca de Origem	Foco Principal do Artigo	Critéro de Exclusão
CRITÉRIO DE EXCLUSÃO	Teaching Guide for Beginnings in DevOps and Continuous Delivery in AWS Focused on the Society 5.0 Skillset	B1	Focado no uso de tecnologias para resolver problemas sociais e econômicos e não no desenvolvimento de software ou um guia para iniciantes.	CE3 - Foco teórico
	The organization of software teams in the quest for continuous delivery: A grounded theory approach	B1	Se trata de um estudo sobre como a indústria de software estrutura a organização de suas equipes de desenvolvimento e infraestrutura. Envolvendo entrevista com desenvolvedores.	CE3 - Foco teórico
	Role of Continuous Glucose Monitoring in Insulin-Requiring Patients with Diabetes	B1	Fala sobre o monitoramento contínuo de glicose em pacientes diabéticos	CE2 - Foge do escopo de desenvolvimento de software
	Consensus and recommendations on continuous glucose monitoring	B1	Fala sobre recomendações para monitoramento contínuo de glicose	CE2 - Foge do escopo de desenvolvimento de software
	Method for Continuous Integration and Deployment Using a Pipeline Generator for Agile Software Projects	B3	Tem foco nos métodos ágeis, não possui nada voltado para iniciantes na área.	CE3 - Foco teórico
	ANALYSIS AND MINIMIZATION OF THE RISKS OF HARMFUL DEPENDENCIES IN THE PROCESS OF CONTINUOUS INTEGRATION AND IMPLEMENTATION OF SOFTWARE CODE	B3	Foco em segurança.	CE3 - Foco teórico
	Towards cost-benefit evaluation for continuous software engineering activities	B3	Aborda o custo benefício da adoção do CI/CD	CE3 - Foco teórico

Figura 3.1: *Aplicação dos critérios de exclusão*

Como ilustrado na Figura 3.1, o artigo "*Teaching Guide for Beginnings in DevOps and Continuous Delivery in AWS Focused on the Society 5.0 Skillset*"apresenta em seu título "Guia de Ensino para Iniciantes", mas, ao fazer a leitura completa do artigo, constatou-se que a sua natureza é a de um manual para o instrutor ou gestor acadêmico, e não um guia técnico e prático para alunos iniciante na área. Essa distinção reforça a identificação da lacuna na literatura em relação a guias práticos e diretos.

3.2 Etapa Prática

Validada a lacuna na literatura, o trabalho avança para a fase de Desenvolvimento e Análise Aplicada, onde o *pipeline* de Integração Contínua é implementado em um cenário real.

3.2.1 Materiais

A base para esta implementação foi um Estudo de Caso inspirado em um projeto de integração contínua para projetos *Python*[1]. Este cenário foi escolhido por utilizar ferramentas gratuitas, o que permite a criação de um guia de fácil replicação. O estudo prático foca na construção passo a passo do *pipeline*, desde a criação do repositório até a execução dos testes automatizados.

As ferramentas utilizadas nesse estudo de caso, foram:

- *GitHub* para versionamento do código;
- *CircleCi* para automatização;
- *VS Code* como editor de código.

O detalhamento completo da implementação do *pipeline* de CI, encontra-se no Capítulo 4.

[1] <https://realpython.com/python-continuous-integration>

Capítulo 4

Implementação do *Pipeline*

Tendo em vista os conceitos de Integração Contínua (CI) e Entrega Contínua (CD) apresentados no capítulo anterior, este capítulo se dedicará à análise prática do fluxo de um *pipeline*. O objeto de estudo é um ambiente moderno construído em um ecossistema *Git*, que integra o repositório *GitHub*[1] e *CircleCI* para automatizar a compilação e o teste de um projeto desenvolvido em *Python*. Na Figura 4.1 é possível ver a estrutura do repositório.

A análise será estruturada em duas partes principais: primeiramente, o detalhamento dos arquivos que compõem a lógica do CI (o código); e, subsequentemente, a implementação do *pipeline*.

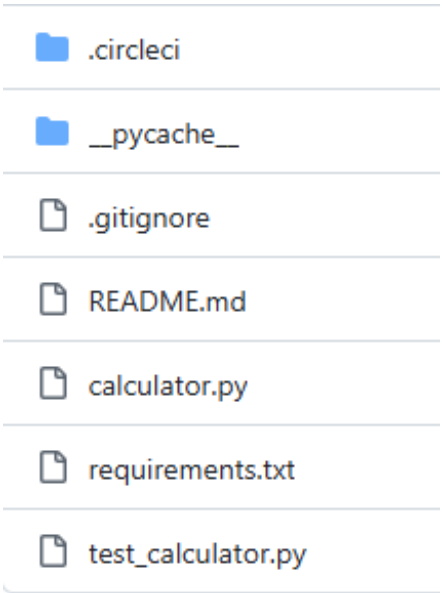


Figura 4.1: *Estrutura do repositório*

[1] <https://github.com/Vivian-Gabriela/CalculatorLibrary>

4.1 Arquivo *config.yml*

O arquivo *config.yml*, contido dentro da pasta *.circleci* na raiz do repositório, é a peça central do *pipeline*, ditando o fluxo, o ambiente e os passos que devem ser executados. A análise deste arquivo revela como o *CircleCI* é configurado para garantir a automação e a padronização do processo de Integração Contínua.

Como podemos ver, a Figura 4.2 representa o arquivo *config.yml*, e este arquivo possui seções. Na seção inicial, a configuração é iniciada com a definição do ambiente, que utiliza a seção *docker* para criar um ambiente de execução isolado. Ao especificar a *image*, linha 06, o *pipeline* garante que o código seja testado em um ambiente padronizado e imutável.

Essa prática é fundamental no CI para eliminar problemas de incompatibilidade com outros dispositivos. Ainda nessa seção, a chave *working_ directory* define o caminho do arquivo onde o código será armazenado no servidor de compilação.

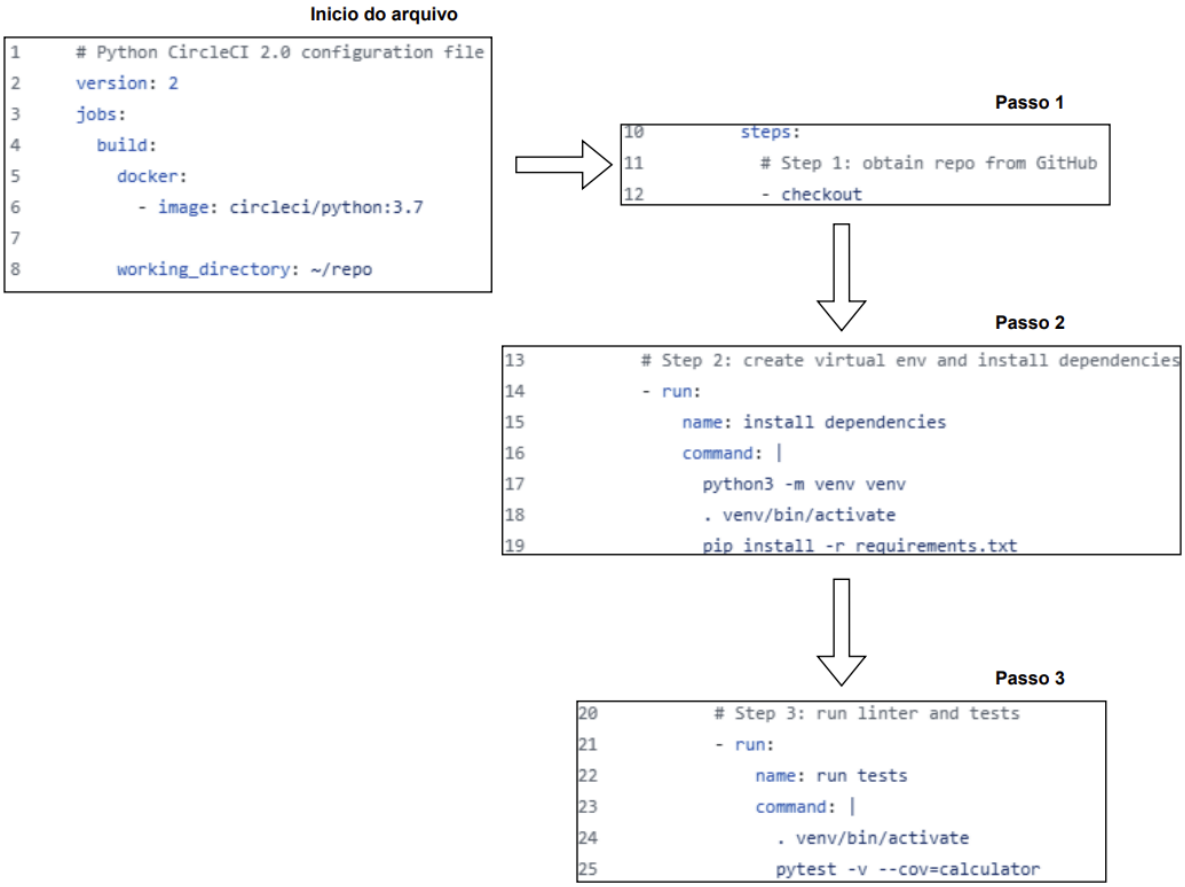


Figura 4.2: *Arquivo config.yml*

Em seguida, a chave *steps* marca o início da lista de etapas a serem executadas. A execução dessas etapas, detalhada nos passos 1, 2 e 3 da Figura 4.2, é analisada a seguir:

- Passo 1 (Obtenção do Código): O servidor precisa verificar o código-fonte no diretório de trabalho. Isso é realizado por meio de uma etapa especial chamada *checkout*.
- Passo 2 (Preparação do Ambiente e Dependências): Nesta etapa ocorre a preparação

do ambiente e das dependências necessárias. O bloco de comandos cria e ativa um ambiente virtual (linha 17 e 18). A utilização do ambiente virtual é essencial para que o servidor de CI consiga reproduzir fielmente o cenário de desenvolvimento [Real Python 2025]. Em seguida, as dependências são instaladas (linha 19) a partir do arquivo *requeriments.txt*, esse arquivo contém as dependências que foram utilizadas para executar os testes unitários, isso garante que o servidor de CI consiga replicar o ambiente.

- Passo 3 (Validação de Qualidade): É a fase de verificação do código. O bloco *command* executa o *Pytest*, uma biblioteca popular utilizada para realizar testes unitários em *Python* [Real Python 2025], cujo propósito é verificar o funcionamento de uma única unidade de código.

Nesta etapa de validação (Passo 3), o *pipeline* atinge dois objetivos cruciais: realiza a validação funcional ao rodar os testes definidos; e, simultaneamente, com o parâmetro *–cov=calculator*, aciona a extensão *pytest-cov*, que calcula a cobertura de código [Real Python 2025].

Como ponto de verificação final, o *Pytest* retorna um código de saída de erro ao *CircleCI* em caso de falha de qualquer teste. Essa ação é o mecanismo pelo qual o *pipeline* é interrompido e o *feedback* imediato é acionado.

O comando *name* presente nos passos 2 e 3 é responsável pela criação do ambiente gráfico do *CircleCI*: a interface de usuário mostra cada etapa da compilação na forma de uma seção expansível, cujo título é obtido a partir do valor associado a esse comando [Real Python 2025]. Na Figura 4.3 é possível ver as seções *install dependencies* e *run tests* na interface gráfica do *CircleCI*.

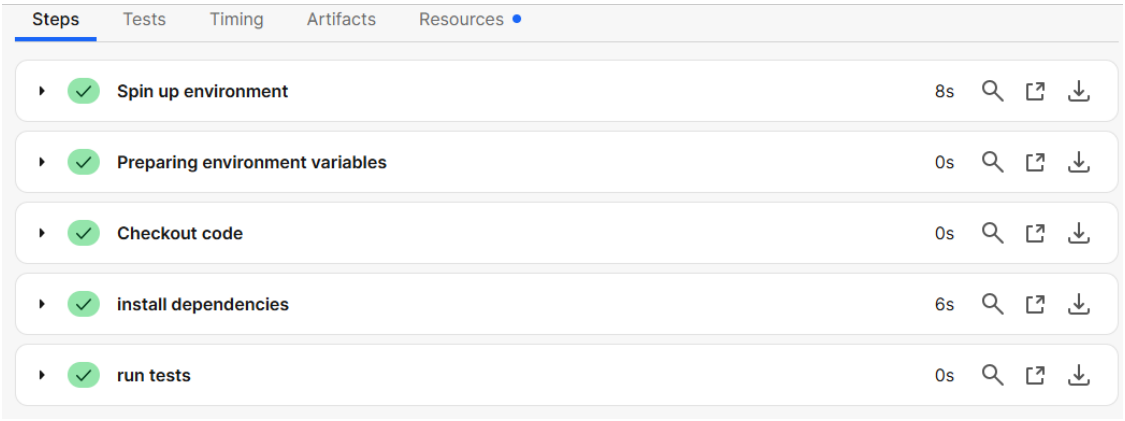


Figura 4.3: *Interface do CircleCI, definida pelo arquivo config.yml*

4.2 Arquivo *calculator.py* e *testecalculator.py*

Nesta seção, será realizada a análise técnica dos dois arquivos que compõem a lógica do projeto: o *calculator.py* (que contém o código da aplicação, Figura 4.4) e o *testecalculator.py*

(responsável pela validação, Figura 4.5). O objetivo é demonstrar a função de cada arquivo e como a lógica de testes unitários atua como o ponto final de verificação de qualidade do *pipeline*.

4.2.1 *calculator.py*

O arquivo *calculator.py* é extremamente simples, contendo apenas três funções básicas: *add*, *subtract* e *multiply*, conforme apresentado na Figura 4.4. A análise deste arquivo demonstra que a funcionalidade é organizada em unidades claras e separadas, um pré-requisito para a implementação eficaz de testes unitários.

```
1     def add(first_term, second_term):
2         return first_term + second_term
3
4
5     def subtract(first_term, second_term):
6         return first_term - second_term
7
8
9     def multiply(first_term, second_term):
10        return first_term * second_term
```

Figura 4.4: *Arquivo calculator.py*

4.2.2 *testecalculator.py*

O arquivo *testecalculator.py* garante que o código funcione conforme o esperado, pois define as regras para aprovação do código. O seu funcionamento é baseado em duas convenções do *Pytest*:

```
1     import calculator
2
3
4     class TestCalculator:
5
6         def test_addition(self):
7             assert 4 == calculator.add(2, 2)
8
9         def test_subtraction(self):
10            assert 2 == calculator.subtract(4, 2)
11
12        def test_multiplication(self):
13            assert 100 == calculator.multiply(10, 10)
```

Figura 4.5: *Arquivo testecalculator.py*

1. Autodescoberta: A biblioteca *Pytest* é configurada para realizar autodescoberta todas as classes e funções que iniciam com *Test* (como a classe *TestCalculator*, Figura 4.5)

[Real Python 2025]. Essa convenção simples garante que qualquer novo teste adicionado ao projeto seja automaticamente incluído na execução do *pipeline*.

2. Asserções (Pontos de Verificação): O comando *assert* é a chave para o mecanismo de falha do CI. Em cada teste, o *assert* verifica se o resultado esperado é estritamente igual ao resultado obtido da função do arquivo *calculator.py* (Figura 4.4). Por exemplo, as linhas 7, 10 e 13 da Figura 4.5 definem os pontos de verificação para o comportamento de suas respectivas funções.

4.3 Passo a Passo da Implementação

Esta seção detalha o processo de construção e configuração do *pipeline* de Integração Contínua, demonstrando a aplicação prática dos conceitos e arquivos analisados no Capítulo 4.

4.3.1 Configuração Inicial do Projeto

O processo se inicia com a configuração básica do repositório *Git* e do ambiente local de desenvolvimento. O repositório central, nomeado *CalculatorLibrary*, foi criado no *GitHub* com a opção de incluir o arquivo *README.md*. Em seguida, o repositório foi clonado para a máquina local, preparando o ambiente de trabalho.

Para garantir o isolamento das dependências, o ambiente virtual foi criado e ativado através do terminal. A Figura 4.6 apresenta os comandos de criação e ativação. Com o ambiente virtual ativo, as bibliotecas *pytest* e *pytest-cov*, essenciais para a execução e cobertura de testes, foram instaladas.

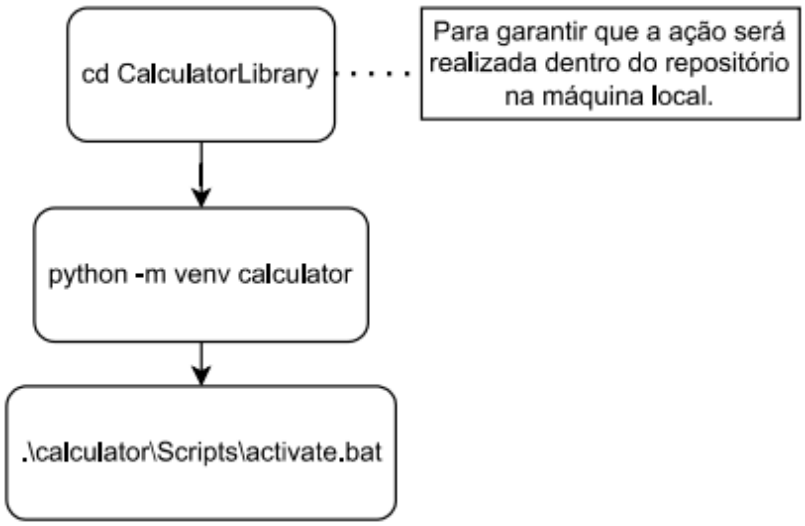


Figura 4.6: *Comandos de criação e ativação do ambiente virtual*

4.3.2 Criação dos Arquivos de Lógica e Teste

Os arquivos *calculator.py* e *test_ calculator.py* foram criados (conforme as Figuras 4.4 e 4.5), adicionados ao *Git* e tiveram suas alterações registradas.

Para que o ambiente possa ser replicado de forma consistente, foi necessário a criação do arquivo *requirements.txt*, utilizando o comando *pip freeze > requeriments.txt*, que lista os pacotes externos utilizados [Real Python 2025]. Este arquivo deve conter as dependências *pytest* e *pytest-cov*.

Com o ambiente configurado e os arquivos necessários criados, a última etapa local foi a verificação. O código foi testado executando o comando *pytest -v –cov* dentro do ambiente virtual. Com o retorno do *Pytest* sendo de sucesso, o código esta pronto para ser enviado ao repositório central do *GitHub*.

4.3.3 Configurando *CircleCI*

Com os arquivos de lógica e testes prontos, a última etapa foi configurar o *CircleCI* para completar o *pipeline*. A pasta *.circleci* foi criada na raiz do repositório e o arquivo *config.yml* (analisado na Figura 4.2) foi adicionado a ela.

O *CircleCI* requer que a pasta seja criada nesse formato (*.circleci/config.yml*), contendo um arquivo de configuração que estabelece as instruções para todas as etapas que o servidor de compilação precisa executar [Real Python 2025].

Durante a configuração e a primeira execução do *pipeline* ocorreram dois erros que necessitaram de ajustes e soluções manuais para que o fluxo fosse concluído com sucesso.

Erro de Autenticação

Inicialmente, ao vincular o repositório *CalculatorLibrary* na interface do *CircleCI*, o *pipeline* retornou o erro "*Checkout code*". Este erro de autenticação ocorreu na primeira etapa de obtenção do código.

Para solucioná-lo, foi necessário realizar um passo manual não detalhado no tutorial: a criação e o *upload* de uma chave *SSH* de usuário pelo terminal. Esta chave foi então adicionada às configurações do *CircleCI* e do *GitHub*, garantindo a permissão para executar o comando *checkout*, a Figura 4.7 representa esse erro e como ele foi solucionado.

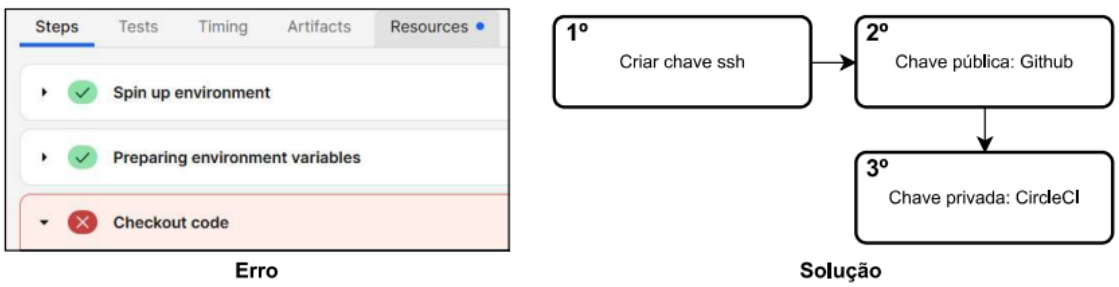


Figura 4.7: *Interface do CircleCI: erro de checkout e solução*

Erro na Instalação das Dependências

Após a resolução da autenticação, o *pipeline* avançou e encontrou um novo erro durante a instalação das dependências. Este erro, representado na Figura 4.8, ocorreu porque o arquivo *requirements.txt* foi gerado usando o comando *pip freeze*.

Esse comando cria um arquivo de requisitos contendo as versões exatas de todos os pacotes instalados em um ambiente [Python Packaging Authority (PyPA) 2024]. Por isso, o arquivo foi gerado com versões específicas das bibliotecas (ex: *coverage==7.10.7*).

Para resolver isso, foi necessário remover todas as especificações de versões, deixando apenas o nome dos pacotes, como apresentado na Figura 4.9. Essa ação garantiu que o ambiente *Docker* do *CircleCI* pudesse instalar a versão mais recente e compatível das dependências.

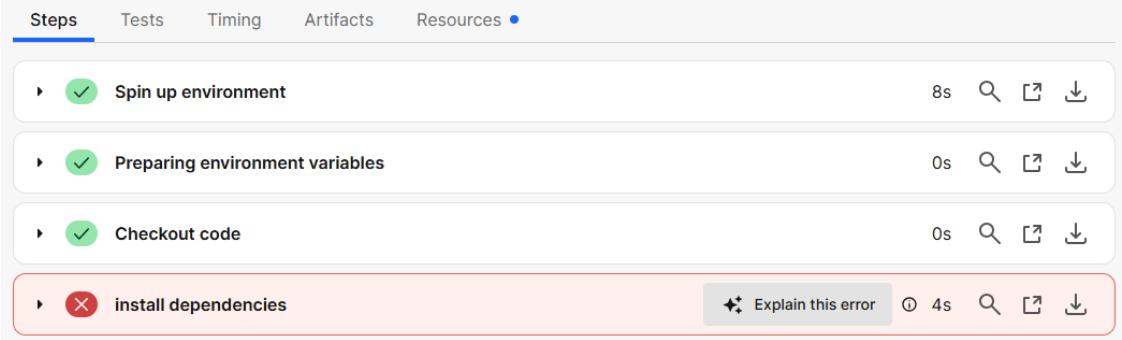


Figura 4.8: *Interface do CircleCI: erro na instalação das dependências*

```
1  colorama
2  coverage
3  iniconfig
4  packaging
5  pluggy
6  Pygments
7  pytest
8  pytest-cov
```

Figura 4.9: *Arquivo requeriments.txt sem as versões dos pacotes*

4.4 Testes Unitários

Nesta seção, são apresentadas as alterações que foram realizadas nos arquivos *calculator.py* e *test_ alculator.py*. O objetivo é demonstrar, de forma prática, como o *pipeline* reage a alterações no código.

Para isso, será adicionada uma nova função ao arquivo *calculator.py* e, em seguida, o teste unitário correspondente será criado no arquivo *test_ calculator.py*.

4.4.1 Adicionando Função

Este passo é simples e necessário para simular uma mudança no código-fonte. No repositório local, o comando *code calculator.py* é executado, abrindo o arquivo no *VS Code* para a adição do código demonstrado na Figura 4.10.

```
12  def division(first_term, second_term):
13      if second_term == 0:
14          return "Error: Division by zero"
15      resultado = first_term / second_term
16
17      return round(resultado, 2)
```

Figura 4.10: *Função da divisão adicionada ao arquivo calculator.py*

Por se tratar de uma função de divisão, foi necessário adicionar uma condição de segurança no código para o caso do usuário tentar executar uma divisão por zero. Essa prática é essencial para evitar falhas críticas e garantir a robustez do *software*, o que será validado pelo teste unitário subsequente.

4.4.2 Adicionando os Testes Unitários

Com o arquivo *calculator.py* atualizado, chegou o momento de escrever os testes unitários correspondentes.

Conforme a implementação apresentada na Figura 4.11, os testes unitários foram projetados para cobrir as três possibilidades críticas da nova função de divisão, fornecendo uma base sólida de validação para o *pipeline* de CI, são elas:

- O primeiro teste unitário, denominado *test_ division_ basic*, testa uma divisão básica, garantindo que a função execute a operação matemática fundamental corretamente.
- O segundo teste unitário, *test_ division_ precision*, verifica se o valor retornado está sendo arredondado para o valor de duas casas decimais. Este teste valida a correta execução do comando *return round(resultado, 2)* presente no arquivo *calculator.py*.
- O terceiro teste unitário, *test_ division_ by_ zero*, previne e valida o cenário de erro no caso do usuário tentar uma divisão por zero. Este teste assegura que a função retorne a saída esperada (*"Error: Division by zero"*), conforme a regra de segurança adicionada no código.

```
15      def test_division_basic(self):
16          assert 2.0 == calculator.division(4, 2)
17
18      def test_division_precision(self):
19          assert 3.33 == calculator.division(10, 3)
20
21      def test_division_by_zero(self):
22          assert "Error: Division by zero" == calculator.division(5, 0)
```

Figura 4.11: *Testes unitários adicionados no arquivo test_ calculator.py*

4.4.3 Execução dos Testes Unitários

Com as alterações realizadas (adição da função de divisão e seus testes unitários), esta seção apresentará os cenários de sucesso e de falha na execução do *pipeline* de CI.

Caso de sucesso

Após adicionar as alterações necessárias nos arquivos *calculator.py* e *test_ calculator.py*, estas foram enviadas para o repositório principal utilizando a sequência de comandos *Git*, considerados essenciais no fluxo de CI. Esses comandos são:

- *git add calculator.py*: informa ao *Git* que a alteração do arquivo indicado, *calculator.py*, está pronta para ser incluída no próximo *commit*.
- *git commit* -m "pequena descrição do que foi alterado": Salva a alteração localmente com uma mensagem descritiva.

Esses comandos se repetem para cada arquivo que passa por alteração e, no final, com o comando *git push*, as alterações são enviadas para o *GitHub*, acionando automaticamente o *pipeline* de CI no *CircleCI*.

Com a configuração final enviada ao repositório central, o *pipeline* de CI foi acionado e executado no servidor do *CircleCI*. A Figura 4.12 ilustra o estado do repositório no *GitHub* após o *push* e a execução do *pipeline*. Observe que o símbolo de *check* verde na interface do *GitHub* indica que a execução do *pipeline* de CI foi bem-sucedida, demonstrando o sucesso em todas as etapas de execução. A interface detalhada que o *CircleCI* retornou com todas as etapas concluídas é a mesma apresentada na Figura 4.3.

Figura 4.12: *Repositório GitHub após o push da configuração final*

Caso de Falha

Para a demonstração do caso de falha no *pipeline* de CI, um erro de execução será induzido propositalmente no código. A análise deste cenário se concentrará em dois aspectos cruciais: primeiro, como o *pipeline* identifica e notifica o desenvolvedor sobre o erro, e segundo, se o código contendo a falha é integrado ao repositório principal.

O teste unitário *test_ division_ by_ zero* foi escolhido para gerar o erro. Para isso, o valor esperado pelo comando *assert* foi alterado. A Figura 4.13 detalha essa alteração, demonstrando a discrepância entre o valor de erro esperado pelo teste e o valor efetivamente retornado pela função.

Após a alteração, a sequência de comandos *Git* (*git add*, *git commit -m* e *git push*) foi executada, acionando o *pipeline* de CI no *CircleCI*.

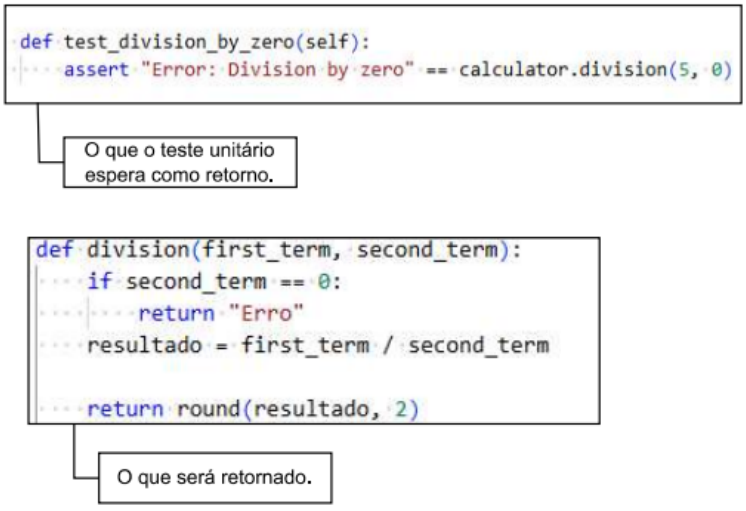


Figura 4.13: *Detalhe da alteração no código (calculator.py) para induzir a falha do teste test_ division_ by_ zero*

O *CircleCI* concluiu a etapa de testes com falha, marcando o *pipeline* como reprovado. A Figura 4.14 demonstra a execução completa do *pipeline*.

No entanto, o segundo ponto crucial da análise, demonstrado na Figura 4.15, é que o código contendo o erro foi integrado diretamente ao repositório principal do *GitHub*. Essa

integração, mesmo com a falha, evidencia a ausência de um mecanismo de proteção que impeça a inclusão de código defeituoso. Essa análise do Caso de Falha estabelece a base para o tópico seguinte, onde será abordado como evitar esse tipo de problema, configurando a proteção do repositório.

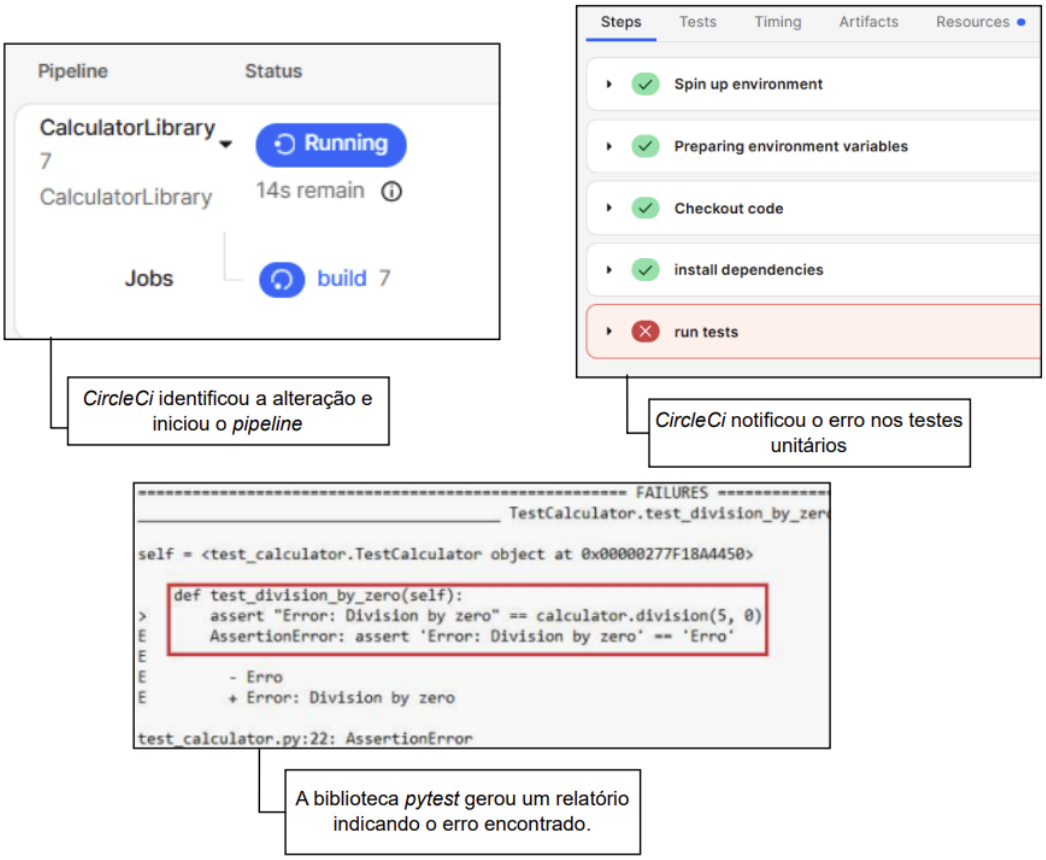


Figura 4.14: *Execução e feedback do pipeline após erro*

Figura 4.15: *Interface do GitHub após o pipeline retornar erro*

4.5 Proteção do Repositório

Para evitar a integração direta, onde o *commit* é imediatamente integrado ao *branch* principal, mesmo com falhas, é necessário alterar as configurações do repositório para que as alterações primeiro passem por um *Pull Request* (PR).

Conforme discutido no Capítulo 2, um PR não é uma alteração imediata, mas sim uma solicitação formal de inclusão de código. Dessa forma, evita-se que um desenvolvedor que não executou os testes unitários antes de enviar para o repositório principal, integre um erro que prejudicará a qualidade do código.

A fim de evitar a situação representada na Figura 4.15, a solução é configurar o repositório do *GitHub* com Regras de Proteção de *Branch* (*Branch Protection Rules*). Essas regras não permitem que o *commit* seja realizado diretamente no *branch* principal, forçando o desenvolvedor a abrir um *Pull Request*.

A Figura 4.16 apresenta as configurações necessárias para esse cenário. Com essas configurações, o *GitHub* verifica o status do *CircleCI*: se o *CircleCI* retornar alguma falha, o *GitHub* não permite que o PR seja mesclado, protegendo o repositório de qualquer código quebrado.

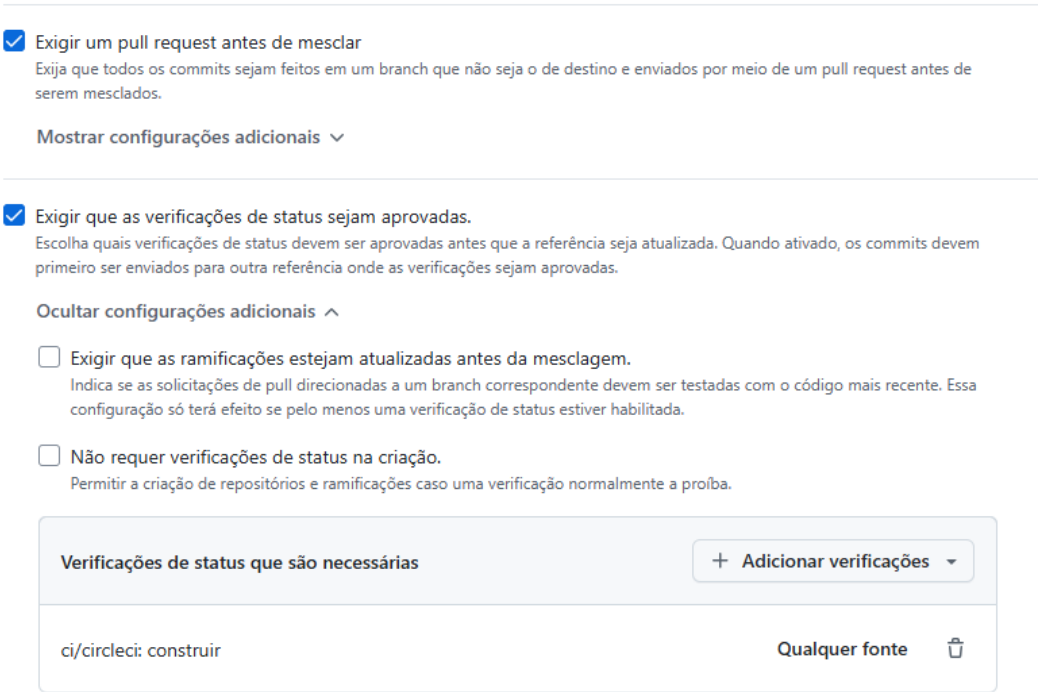


Figura 4.16: *Regras necessárias para que o código passe primeiro por um PR antes de ser integrado ao repositório principal*

Na primeira opção "Exigir um *pull request* antes de mesclar"tem uma seção com "Aprovações necessárias". Neste cenário didático, as aprovações foram zeradas para que o próprio autor possa mesclar, já que há apenas um escritor.

Com a configuração de PR realizada, é necessário criar um *branch* secundário para que o *git push* seja direcionado a ele. Pois, se o *git push* for realizado para o repositório principal

(*main*), o comando retorna um erro, representado pela Figura 4.17, informando que não é possível realizar a ação devido a uma regra do repositório.

Após criar um novo *branch* (*branch_ secundaria*) na interface *GitHub*, é necessário atualizar o repositório local para que ele localize essa nova *branch* e envie os próximos *pushs* para ela. O comando *git fetch* serve exatamente para esse propósito de atualizar o repositório local [Git SCM 2025].

```
remote: error: GH013: Repository rule violations found for refs/heads/main.
remote: Review all repository rules at https://github.com/Vivian-Gabriela/Calculator
remote:
remote: - Changes must be made through a pull request.
remote:
remote: - 2 of 2 required status checks are expected.
remote:
To https://github.com/Vivian-Gabriela/CalculatorLibrary.git
 ! [remote rejected] main -> main (push declined due to repository rule violations)
```

Figura 4.17: *Erro retornado ao executar o comando git push no repositório principal após ter sido adicionado as regras de proteção de Branch*

Com o repositório atualizado, execute o comando *git checkout branch_ secundaria*, esse comando muda o desenvolvedor para o *branch* secundário, permitindo que os novos *commits* sejam registrados separadamente do *branch* principal. Após isso, basta realizar qualquer alteração para o repositório seguindo os comandos *Git*.

Com isso, o fluxo de trabalho do *pipeline* de CI fica completo, garantindo a qualidade do código. A exigência de um *Pull Request* antes da mesclagem e a validação obrigatória do *CircleCI* asseguram que somente código testado e aprovado possa ser integrado ao *branch* principal, protegendo a integridade do repositório.

4.5.1 Caso de Falha Após Implementação do PR

Para confirmar que o repositório esta seguro, exigindo um PR antes de mesclar, o mesmo erro foi induzido novamente para verificar a reação do *GitHub*. A Figura 4.18 apresenta o novo fluxo após as Regras de Proteções de *Branch* terem sido implementadas.

No caso de um cenário de sucesso, a única diferença do fluxo da Figura 4.18 é que a opção de mesclar o código estaria disponível, pois as condições de aprovação seriam cumpridas.

4.6 Discussão Sobre o *Pipeline* Implementado

Ao longo da implementação prática, demonstrou-se o ciclo completo da Integração Contínua em um ambiente real. A construção do *pipeline* no *CircleCI*, validou a intenção de criar um guia para iniciantes.

O ponto crucial da implementação foi a configuração das Regras de Proteção de *Branch* no *GitHub*. Ao forçar que toda alteração passasse por um *Pull Request* (PR), a arquitetura de CI foi complementada.

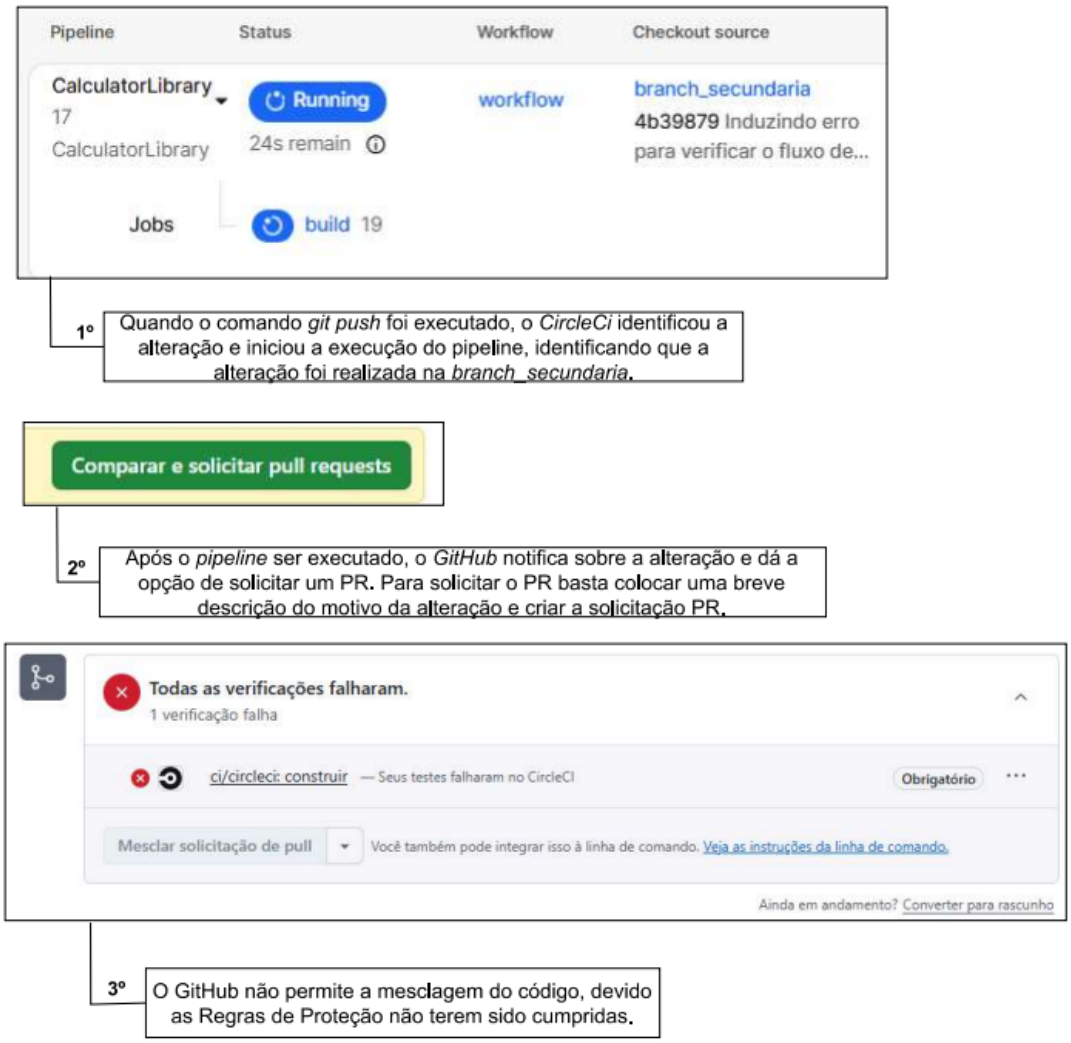


Figura 4.18: *Fluxo completo de trabalho, em um caso de falha, após a adição do PR*

O cenário de sucesso provou que o código limpo e testado pôde ser mesclado ao *branch* principal (*main*), mantendo a integridade. Enquanto o cenário de falha, consolidou o funcionamento do *pipeline* de CI, comprovando que em caso de erro no código, o *GitHub* não permite que o código seja mesclado.

Esse cenário de falha comprovou um dos benefícios do CI, que é a garantia de que o código continue funcional e seguro para outras alterações.

Capítulo 5

Considerações Finais e Sugestões para Trabalhos Futuros

Neste capítulo, são apresentadas as considerações finais do trabalho. Inicialmente, são revisados os objetivos propostos para o guia prático, seguido pela discussão dos resultados alcançados com a implementação do *pipeline* de Integração Contínua. Por fim, são sugeridas expansões para o tema e o desenvolvimento de trabalhos futuros.

5.1 Considerações Finais

Este Trabalho de Conclusão de Curso propôs a criação de um guia prático e didático para a implementação de um *pipeline* de Integração Contínua, utilizando ferramentas gratuitas, como *GitHub*, *CircleCI* e *VS Code*.

Os objetivos propostos foram integralmente alcançados:

- Implementação do Ambiente CI: O ambiente foi estabelecido com sucesso, integrando o repositório *GitHub* ao serviço de CI *CircleCI*.
- Validação de Testes Unitários: Demonstrou-se a importância e a execução dos testes unitários (*Pytest*) para a validação da lógica de programação, tanto em cenário de sucesso quanto em cenário de falha.
- Segurança do Repositório: O estudo comprovou a eficácia das Regras de Proteção de *Branch* do *GitHub*. A análise de falha ilustrou de forma prática que o *pipeline* do *CircleCI*, ao encontrar um erro, foi capaz de se comunicar com o *GitHub* para bloquear a mesclagem de código defeituoso no *branch* principal. Este bloqueio garante a integridade e a segurança do repositório, cumprindo o principal benefício do CI.

A estrutura e a clareza do guia prático desenvolvido servem como uma base sólida para iniciantes que buscam automatizar a garantia de qualidade de seus projetos, tornando a adoção das práticas de CI/CD mais acessível.

Para um terceiro que deseje replicar este cenário, a implementação se torna significativamente simplificada. Devido ao trabalho detalhado de configuração e validação inicial, é necessário apenas clonar o repositório já pronto e realizar as conexões de serviço entre *GitHub* e *CircleCI*. Isso transforma o projeto em um ativo reutilizável e de rápida implantação.

5.2 Sugestões para Trabalhos Futuros

Em um contexto de evolução do projeto e aprofundamento das práticas de CI/CD, sugere-se a expansão do *pipeline* para a fase de Entrega Contínua (CD).

O trabalho futuro seria estender o *pipeline* para incluir a Entrega Contínua, automatizando a implantação da *CalculatorLibrary* em um ambiente de produção. Essa automação deve ser acionada imediatamente após a mesclagem bem-sucedida no *branch* principal (*main*), garantindo que o código, uma vez testado e aprovado pelo CI, seja disponibilizado para os usuários sem intervenção manual.

Com essa expansão do trabalho, é possível construir o guia completo para iniciantes da área, abordando um *pipeline* completo de CI/CD.

Referências Bibliográficas

[Benjamin e Mathew 2021] BENJAMIN, J.; MATHEW, J. Enhancing the efficiency of continuous integration environment in devops. *IOP Conference Series: Materials Science and Engineering*, v. 1085, p. 012025, 2 2021. ISSN 1757-8981. 1, 2, 4, 5

[Bernardo *et al.* 2023] BERNARDO, J. H. *et al.* The impact of a continuous integration service on the delivery time of merged pull requests. *Empirical Software Engineering*, v. 28, p. 97, 7 2023. ISSN 1382-3256. 1, 2, 4, 5, 9

[Bounegru 2024] BOUNEGRU, L. The platformisation of software development: Connective coding and platform vernaculars on github. *Convergence: The International Journal of Research into New Media Technologies*, v. 30, p. 2212–2232, 12 2024. ISSN 1354-8565. 2, 9

[Chen 2017] CHEN, L. Continuous delivery: Overcoming adoption challenges. *Journal of Systems and Software*, v. 128, p. 72–86, 6 2017. ISSN 01641212. 4

[Donca *et al.* 2022] DONCA, I.-C. *et al.* Method for continuous integration and deployment using a pipeline generator for agile software projects. *Sensors*, v. 22, p. 4637, 6 2022. ISSN 1424-8220. Disponível em: <https://www.mdpi.com/1424-8220/22/12/4637>. xi, 2, 6

[Git SCM 2025] Git SCM. *git-fetch*. 2025. <https://git-scm.com/docs/git-fetch>. Disponível em: <https://git-scm.com/docs/git-fetch>. Acesso em: 4 Novembro 2025. 28

[GitHub 2025] GitHub. *About commits*. 2025. <https://docs.github.com/pt/pull-requests/committing-changes-to-your-project/creating-and-editing-commits/about-commits>. Acessado em 10 de setembro de 2025. 8

[GitHub 2025] GitHub. *About pull requests*. 2025. <https://docs.github.com/pt/pull-requests/collaborating-with-pull-requests/proposing-changes-to-your-work-with-pull-requests/about-branches>. Acessado em 12 de setembro de 2025. 7

[GitHub 2025] GitHub. *Sobre repositórios*. 2025. <https://docs.github.com/pt/repositories/creating-and-managing-repositories/about-repositories>. Acessado em 9 de Dezembro de 2025. 7

[Laukkanen, Itkonen e Lassenius 2017] LAUKKANEN, E.; ITKONEN, J.; LASSENIUS, C. Problems, causes and solutions when adopting continuous delivery—a systematic literature review. *Information and Software Technology*, v. 82, p. 55–79, 2 2017. ISSN 09505849. 2

[Lima e Vergilio 2023] LIMA, J. A. do P.; VERGILIO, S. R. An evaluation of ranking-to-learn approaches for test case prioritization in continuous integration. *Journal of Software Engineering Research and Development*, 2 2023. ISSN 2195-1721. xi, 2, 4, 6

[Ortu *et al.* 2020] ORTU, M. *et al.* How do you propose your code changes? empirical analysis of affect metrics of pull requests on github. *IEEE Access*, v. 8, p. 110897–110907, 2020. ISSN 2169-3536. 8

[PRISMA 2025] PRISMA. *PRISMA (Preferred Reporting Items for Systematic Reviews and Meta-Analyses)*. 2025. <https://www.prisma-statement.org/>. Acessado em: 31 de Outubro de 2025. 11

[Python Packaging Authority (PyPA) 2024] Python Packaging Authority (PyPA). *Instalando, usando pip e ambientes virtuais (Python Packaging User Guide)*. 2024. <https://packaging.python.org/pt-br/latest/guides/installing-using-pip-and-virtual-environments/>. Acessado em 15 de Outubro de 2025. 22

[Real Python 2025] Real Python. *Python Continuous Integration*. 2025. <https://realpython.com/python-continuous-integration/>. Acessado em 9 de outubro de 2025. 18, 20, 21

[Song e Kim 2018] SONG, J.; KIM, C. What is needed for the sustainable success of oss projects: Efficiency analysis of commit production process via git. *Sustainability*, v. 10, p. 3001, 8 2018. ISSN 2071-1050. 8