WEB ACADEMY

Integração Contínua

Daniel Augusto Nunes da Silva

Apresentação

Ementa

- **Entrega de software**: problemas, princípios e *pipelines*. **DevOps**. Controle de versões. **Integração contínua**. Boas práticas no uso de integração contínua. Desenvolvimento Baseado no Trunk (TBD). Servidores de integração contínua. **GitHub Actions**. Fluxo de trabalho no GitHub. **Implantação contínua**. Entrega contínua.

Objetivos

- **Geral:** Capacitar o aluno na utilização de técnicas de **integração e implantação contínua** em projetos de software, utilizando ferramentas para **automatizar o processo de entregar software**.

- **Específicos:**
  - Discutir os problemas relacionados ao processo de entrega de software;
  - Relacionar os conceitos integração e implantação contínua, com ênfase no conceito de DevOps;
  - Apresentar técnicas e ferramentas para automatizar tarefas relacionadas a integração e implantação contínua.
  - Construir um pipeline de integração e implantação contínua em um projeto de software.

Conteúdo programático

Introdução

- O problema de entregar software;
- Princípios para entrega de software;
- DevOps;
- Pipeline para entrega de software.

Integração Contínua

- Introdução a Integração Contínua;
- Boas práticas no uso de CI;
- Desenvolvimento Baseado no Trunk (TBD);
- GitHub Actions;
- Fluxos de trabalho no GitHub Actions;
- Abordagens para implementar CI;
- Quando não usar CI?

Implantação Contínua

- Introdução a Implantação Contínua;
- Fluxo de trabalho;
- Vantagens;
- Criação de workflow no GitHub Actions.

Sites de referência

- Software Delivery Guide (Martin Fowler).
  - https://martinfowler.com/delivery.html
- GitHub Docs: GitHub Actions.
  - https://docs.github.com/pt/actions
- Docker Docs.
  - https://docs.docker.com/get-started/overview/

Contato

https://github.com/danielnsilva

Introdução

O problema de entregar software

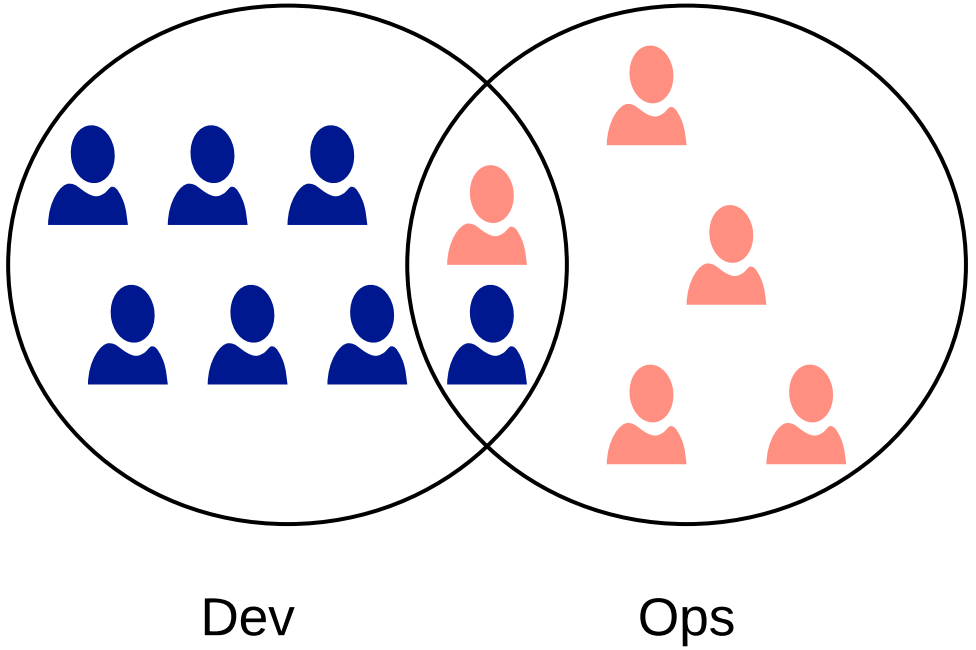
- **Colocar um software em produção pode implicar em muitas dificuldades**: problemas de compilação, testes, configuração de ambientes, etc.
- **Todo modelo de desenvolvimento de software descreve uma etapa na qual o software entra em operação** e, ainda, recebe novas versões durante seu ciclo de vida.
- Até meados dos anos 2000, normalmente as metodologias estavam concentradas em técnicas de gerenciamento de projetos e requisitos, além de práticas de desenvolvimento e testes.
- Essas boas práticas aplicadas ao processo de desenvolvimento de software devem envolver uma **maneira eficiente de entregar software**.

Princípios para entrega de software

- Crie um **processo repetível e confiável** para entrega de software.
- **Automatize** tudo que for possível.
- Mantenha tudo em um **sistema de controle de versões**.
- Se uma tarefa pode causar problemas, deve ser executada com **mais frequência** e o quanto antes.
- “**Concluído**” significa pronto para entrega.
- **Todos são responsáveis** pela entrega do software.

DevOps

- Historicamente, tarefas de **desenvolvimento** e de **operações** (infraestrutura) são separadas em equipes diferentes.
- Essa divisão dificulta o processo de entregar software com rapidez e qualidade.
- A proposta do **DevOps** é integrar as duas àreas, otimizando o processo de entregar software (colocar em produção).

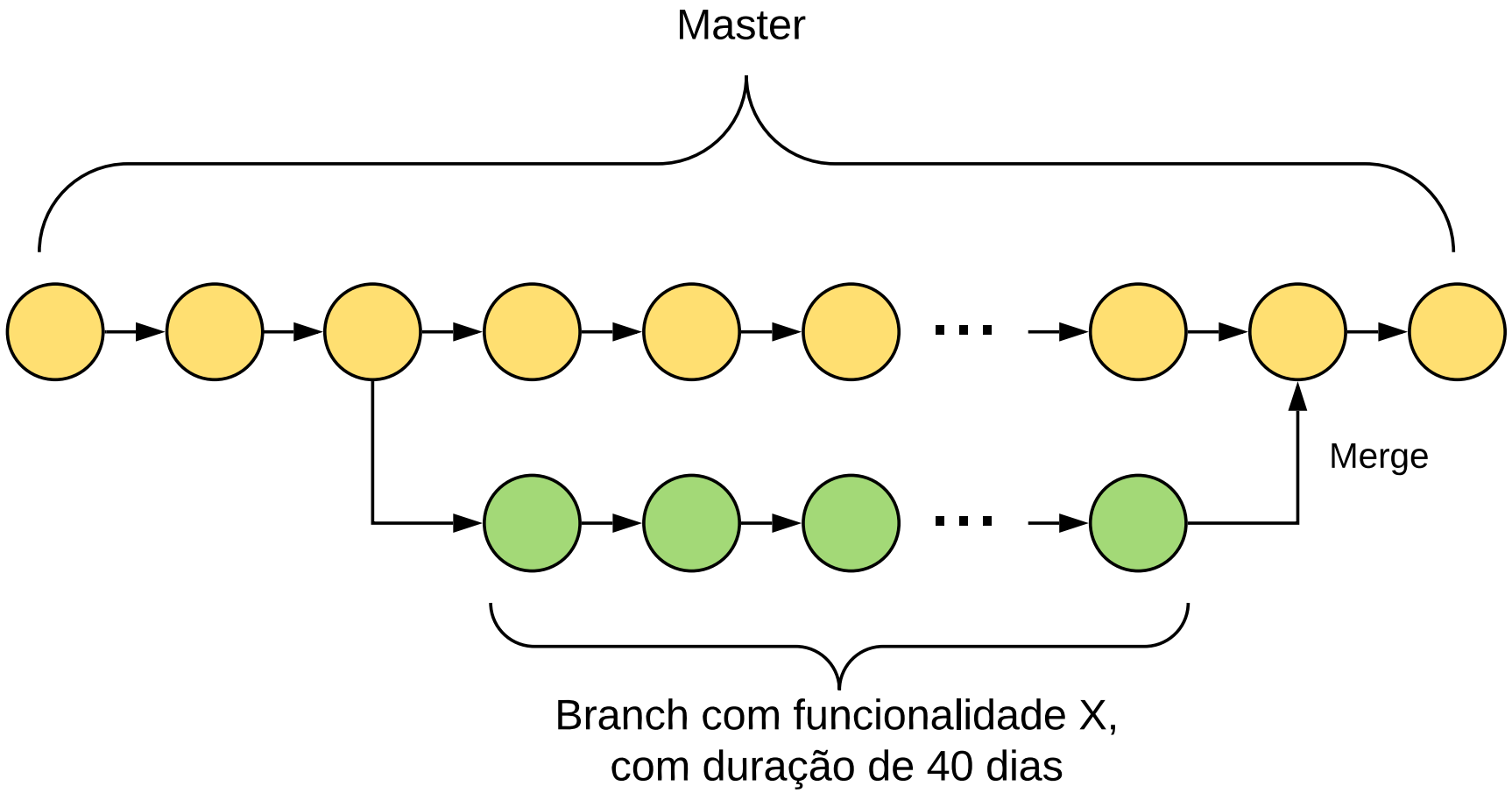


Fonte: VALENTE, 2020.

Pipeline para entrega de software

Integração Contínua

Integração Contínua



Fonte: VALENTE, 2020.

Integração Contínua

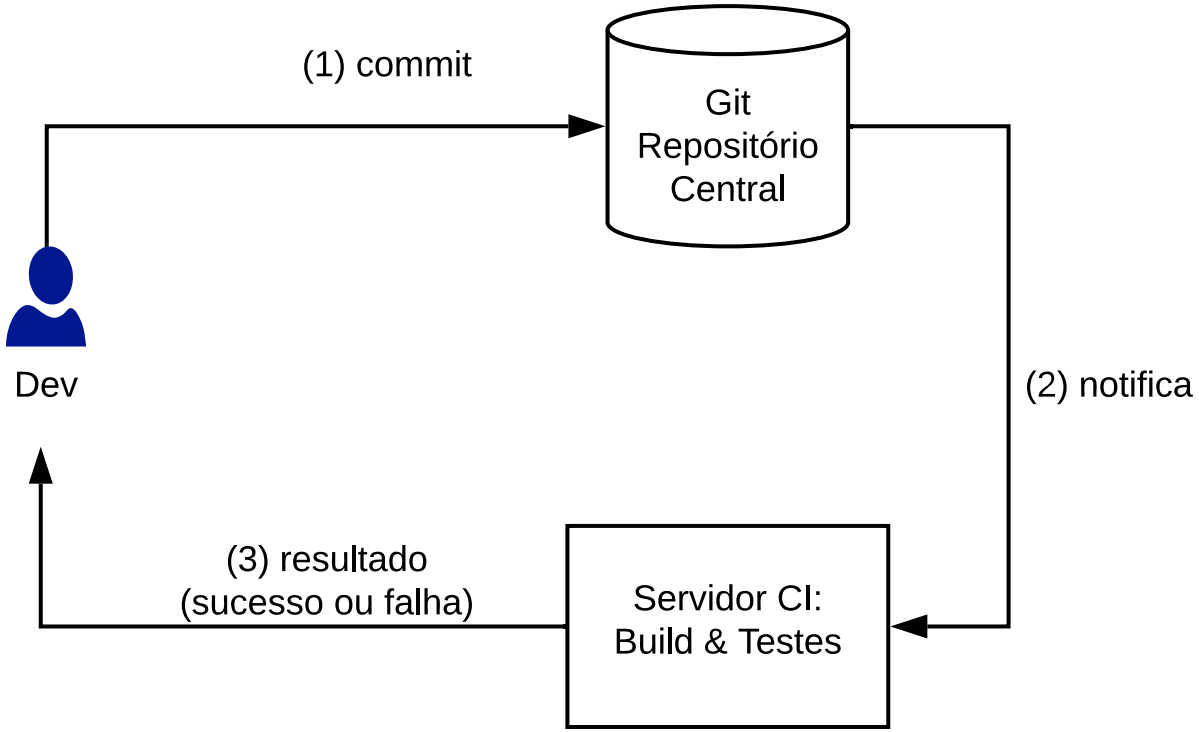
- **Integração Contínua** (*Continuous Integration* ou **CI**) é uma prática proposta pela metodologia ágil *Extreme Programming* (XP).
- Princípio motivador: **se uma tarefa pode causar problemas, não podemos deixar que ela acumule**.
  - **Grandes integrações de código** podem gerar mais conflitos.
  - Se **integrar o código de forma frequente**, isto é, contínua, as integrações serão pequenas e irão **gerar menos conflitos**.
- Devemos **quebrar uma tarefa em subtarefas** que possam ser **realizadas de forma frequente**.

Boas Práticas para Uso de CI

- **Build Automatizado**
  - É importante que **seja o mais rápido possível**, pois com integração contínua ele será executado com frequência.
- **Testes Automatizados**
  - Além de garantir que o software compila sem erros após cada novo commit, é importante garantir também que ele **continua com o comportamento esperado**.

Boas Práticas para Uso de CI

- **Servidores de Integração Contínua**
  - Os builds e testes automatizados devem ser executados com frequência.
  - Após um novo commit, o sistema de controle de versões avisa o servidor de CI, que clona o repositório e executa um build completo, bem como roda todos os testes.
  - Após a execução do build e dos testes, o servidor notifica o usuário.



Fonte: VALENTE, 2020.

Boas Práticas para Uso de CI

- **Desenvolvimento Baseado no Trunk**
  - CI é compatível com o uso de branches desde que sejam integrados de forma frequente no master (todo dia).
  - Quando migram para CI, é comum que as organizações usem também desenvolvimento baseado no trunk (**TBD** – *trunk based development*).
  - **Não existem mais branches** para implementação de novas funcionalidades ou para correção de bugs.
  - **Todo desenvolvimento ocorre no branch principal**, também conhecido como trunk ou master (main).

GitHub Actions

- Ferramenta integrada ao GitHub para automatizar a execução de fluxos de trabalho de desenvolvimento de software, dando suporte a utilização CI.
- **Uma ação (Github Action) é um aplicativo** que executa uma tarefa complexa (envolve vários passos), mas que é repetitiva.
- É possível combinar ações personalizadas e ações já disponíveis na plataforma.
- **O GitHub faz o papel de Servidor de CI**, fornecendo uma máquina virtual para executar as ações.
- O arquivos de configuração utilizam a linguagem YAML e devem ser armazenados no diretório **.github/workflows/** em cada projeto.

Linguagem YAML

- **YAML** é uma **linguagem de serialização de dados** muito usada em arquivos de configuração, e projetada com **ênfase na legibilidade**.
- Assim como JSON, é baseada em pares de **chave e valor**;
- É a linguagem padrão de ferramentas como o GitHub Actions.

```yaml
# Comentário
nome: "Exemplo de YAML"
professor:
  nome: "Daniel"
  disciplinas:
    - nome: "Frameworks Back-end"
      carga_horaria: 15
    - nome: "Frameworks Front-end"
      carga_horaria: 15
descricao: |
  Este é um exemplo de valor
  multilinha no YAML.
```

GitHub Actions Workflow

```yaml
name: Primeiro Workflow
on:
  push:
jobs:
  primeiro-job:
    runs-on: ubuntu-latest
    steps:
    - run: echo "Primeiro passo"
    - name: Checkout
      uses: actions/checkout@v5
    - name: Lista arquivos
      run: ls ${{ github.workspace }}
    - name: Status
      run: echo "Status ${{ job.status }}."
```

Sempre que um novo arquivo é enviado ao repositório.

Provisiona uma máquina virtual para funcionar como Servidor CI.

O action “checkout” faz o checkout do branch atual no Servidor CI.

O hífen indica um passo, que pode agrupar diferentes configurações.

Abordagens para implementar CI

1. **Commit no branch principal**: CI é executado para todos os commits ocorrem no branch principal.
2. **Merge entre branches**: CI é executado quando um commit é feito em algum branch e, ao final do processo, ocorre o merge com o branch principal de forma automática.
3. **Pull request**: CI é executado quando um pull request é aberto, que pode ter origem em um fork do repositório ou um branch. Nesta abordagem também é possível realizar o merge automático com o branch principal.

GitHub Actions Workflow (branch principal)

```yaml
name: CI Workflow
on:
  push:
    branches:
      - 'main'
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v5
      - name: Set up JDK 17
        uses: actions/setup-java@v5
        with:
          java-version: '17'
          distribution: 'temurin'
      - name: Maven Compile/Test
        run: mvn compile test
```

Apenas no branch principal

Instala a versão 17 do JDK, baseado na distribuição Temurin, no Servidor CI provisionado.

Usa o Maven para compilar e testar o projeto.

GitHub Actions Workflow (merge)

```yaml
name: CI Workflow
on:
  push:
    branches:
      - '**'
      - '!main'
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v5
      [...]
      - name: Merge branch
        uses: devmasx/merge-branch@1.4.0
        with:
          type: now
          target_branch: main
          github_token: ${{ github.token }}
```

Todos os branches, exceto o principal (main).

Usa o action “merge-branch” para fazer merge no branch principal. O github.token é gerado automaticamente.

GitHub Actions Workflow (pull request)

```yaml
name: CI Workflow
on:
  push:
    branches:
      - 'main'
    pull_request:
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v5
      - name: Set up JDK 17
        uses: actions/setup-java@v5
        with:
          java-version: '17'
          distribution: 'temurin'
      - name: Maven Compile/Test
        run: mvn compile test
```

Quando um Pull Request é criado

Instala a versão 17 do JDK, baseado na distribuição Temurin, no Servidor CI provisionado.

Usa o Maven para compilar e testar o projeto.

Quando não usar CI?

- **CI tem um limite rígido para integrações no ramo principal**: pelo menos uma integração por dia por desenvolvedor.
  - No entanto, dependendo da organização, do domínio do sistema (que pode ser um sistema crítico) e do perfil dos desenvolvedores (que podem ser iniciantes), pode ser difícil seguir esse limite.
- **CI também não é compatível com projetos de código aberto**, onde os desenvolvedores são voluntários e não têm disponibilidade para trabalhar diariamente no seu código.

Implantação Contínua

Implantação Contínua

- Com integração contínua, o código novo é frequentemente integrado no ramo principal, mas **esse código não precisa estar pronto para entrar em produção**.
- Existe mais um passo da cadeia de automação proposta por DevOps, chamado de **Implantação Contínua** (*Continuous Deployment* ou **CD**).
- A diferença entre CI e CD é simples: quando usa-se CD, **todo novo commit** que chega no ramo principal (main) **entra rapidamente em produção** (em questões de horas).
- Para funcionar bem, **CD requer que as atualizações de código sejam pequenas**.

Implantação Contínua

- **Fluxo de trabalho** quando se usa CD:

1. O **desenvolvedor cria o novo código e testa** no seu ambiente local.
2. Ele realiza um commit e o **servidor de CI executa novamente um build e os testes**.
3. **Algumas vezes no dia, o servidor de CI realiza os testes mais demorados** com os novos commits que ainda não entraram em produção.
4. **Se todos os testes passarem**, os commits entram imediatamente em produção, e os **usuários já vão interagir com a nova versão do código**.

Implantação Contínua

- **Vantagens** de CD:
  - CD **reduz o tempo de entrega** de novas funcionalidades.
  - CD torna novas releases (ou implantações) um **não-evento**.
  - Além de reduzir o problemas causado por *deadlines*, CD **permite que os desenvolvedores recebam feedback frequente**, o que pode contribuir para um ambiente de trabalho mais estimulante.
  - Como consequência, CD **favorece experimentação** e um estilo de desenvolvimento orientado por dados e feedback dos usuários.

Implantação ou Entrega?

- Implantação Contínua (CD) **não é recomendável para certos tipos de sistemas**, incluindo alguns sistemas desktop, aplicações móveis e aplicações embutidas em hardware.
- É possível usar um versão mais "fraca" de CD, chamada de **Entrega Contínua** (*Continuous Delivery*): quando se usa entrega contínua, todo commit pode entrar em produção imediatamente, mas **existe uma autoridade externa que toma a decisão sobre quando os commits, de fato, serão liberados para os usuários finais**.
- **Deployment (Implantação)** é o processo de liberar uma nova versão de um sistema para seus usuários.
- **Delivery (Entrega)** é o processo de liberar uma nova versão de um sistema para ser objeto de deployment.

Implantação com Docker

```
FROM maven:3.8-eclipse-temurin-17-alpine AS build

COPY src /home/app/src

COPY pom.xml /home/app

RUN mvn -f /home/app/pom.xml clean package -Pprod -Dmaven.test.skip=true


FROM eclipse-temurin:17-jre-alpine

COPY --from=build /home/app/target/sgcmapi.jar /home/app/sgcmapi.jar

EXPOSE 9000

CMD java -jar /home/app/sgcmapi.jar
```

Implantação com Docker

```
FROM maven:3.8-eclipse-temurin-17-alpine AS build

COPY src /home/app/src

COPY pom.xml /home/app

RUN mvn -f /home/app/pom.xml clean package -Pprod -Dmaven.test.skip=true


FROM eclipse-temurin:17-jre-alpine

COPY --from=build /home/app/target/sgcmapi.jar /home/app/sgcmapi.jar

EXPOSE 9000

CMD java -jar /home/app/sgcmapi.jar
```

Separação do processo em **estágios** ajuda a **diminuir o tamanho** final da imagem e torna o processo **mais rápido**.

Implantação com Docker

```
FROM maven:3.8-eclipse-temurin-17-alpine AS build

COPY src /home/app/src

COPY pom.xml /home/app

RUN mvn -f /home/app/pom.xml clean package -Pprod -Dmaven.test.skip=true
```

Cria o pacote do projeto com base o perfil “prod”, ignorando os testes para executar mais rápido

```
FROM eclipse-temurin:17-jre-alpine

COPY --from=build /home/app/target/sgcmapi.jar /home/app/sgcmapi.jar

EXPOSE 9000

CMD java -jar /home/app/sgcmapi.jar
```

Prepara o ambiente de execução da aplicação, copiando o arquivo JAR criado no estágio anterior.

Implantação com Docker

```
FROM maven:3.8-eclipse-temurin-17-alpine AS build

COPY src /home/app/src

COPY pom.xml /home/app

RUN mvn -f /home/app/pom.xml clean package -Pprod -Dmaven.test.skip=true
```

Maven 3.8 com JDK 17 (distribuição Eclipse Temurin) no Alpine Linux (162.95 MB)

```
FROM eclipse-temurin:17-jre-alpine

COPY --from=build /home/app/target/sgcmapi.jar /home/app/sgcmapi.jar

EXPOSE 9000

CMD java -jar /home/app/sgcmapi.jar
```

JRE 17 (distribuição Eclipse Temurin) no Alpine Linux (56.19 MB)

GitHub Actions Workflow

```yaml
name: CD Workflow
on:
  push:
jobs:
  deploy:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v5
      - name: Deploy
        uses: webacademyufac/deploy-action@main
        with:
          app-id: ${{ secrets.DEPLOY_APP_ID }}
          api-key: ${{ secrets.DEPLOY_API_KEY }}
```

Inicia o processo de *deploy* no Dokploy, informando o ID da Aplicação e a API KEY para autenticação.

Fim!

Referências

- HUMBLE, Jez; FARLEY, David. **Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation**. 1. ed. [S. l.]: Pearson Addison-Wesley, 2010. 512 p.
- DUVALL, Paul M. **Continuous Integration: Improving Software Quality and Reducing Risk**. 1. ed. [S. l.]: Pearson Addison-Wesley, 2007. 336 p.
- GITHUB (ed.). **GitHub Docs: GitHub Actions**. [S. l.], 2025. Disponível em: https://docs.github.com/pt/actions.
- MARCO TULIO VALENTE. **Engenharia de Software Moderna: Princípios e Práticas para Desenvolvimento de Software com Produtividade**, 2020. Disponível em: https://engsoftmoderna.info/.
- SOMMERVILLE, Ian. **Engenharia de Software**. 9. ed. São Paulo: Pearson Addison-Wesley, 2011.