ARQUITETURA DE SW

EAD - CI/CD

Análise e Desenvolvimento de Sistemas

Prof. Ms. Renato Leite
renato@sectras.edu.br

OBJETIVOS

1. Contexto e história dos processos de deploy

2. Deploy e arquiteturas

3. Pipelines (CI/CD)

4. Ferramentas

5. Automação

6. Exercícios

REVISANDO DEVOPS

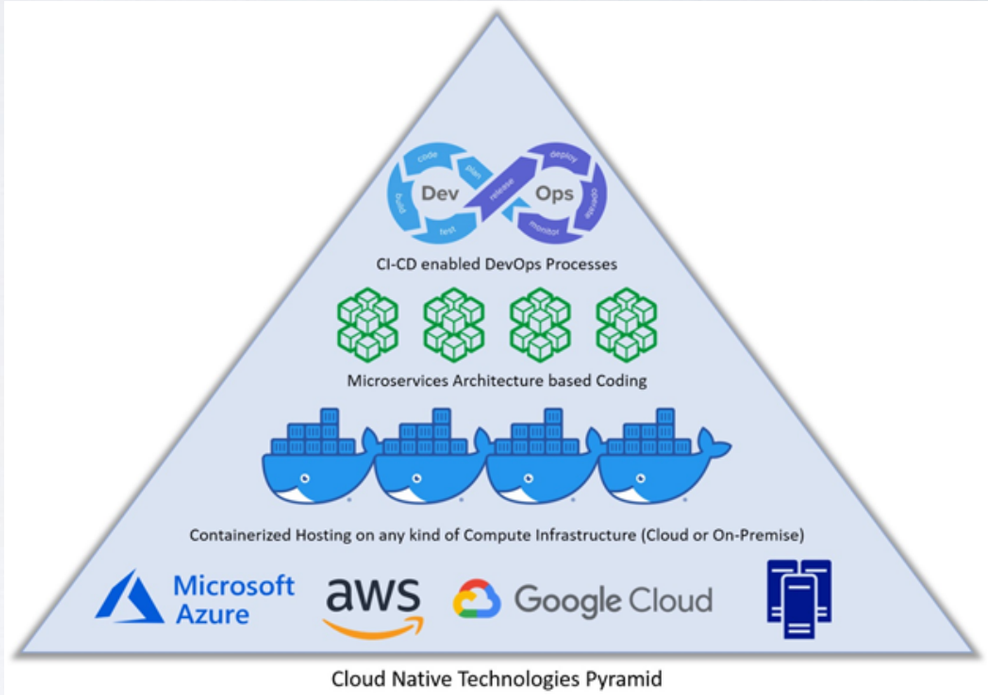
1. Com o avanço das arquiteturas de microservices, surgem novos desafios de monitoramento e controle.

2. Cada serviço é independente, com logs e métricas próprios.

3. O DevOps passa a incluir **observabilidade**, **resiliência** e **automação operacional**.

4. A confiabilidade deixa de ser centralizada e passa a emergir da cooperação entre serviços.

5. Em DevOps, operação e desenvolvimento formam um ciclo contínuo de aprendizado.



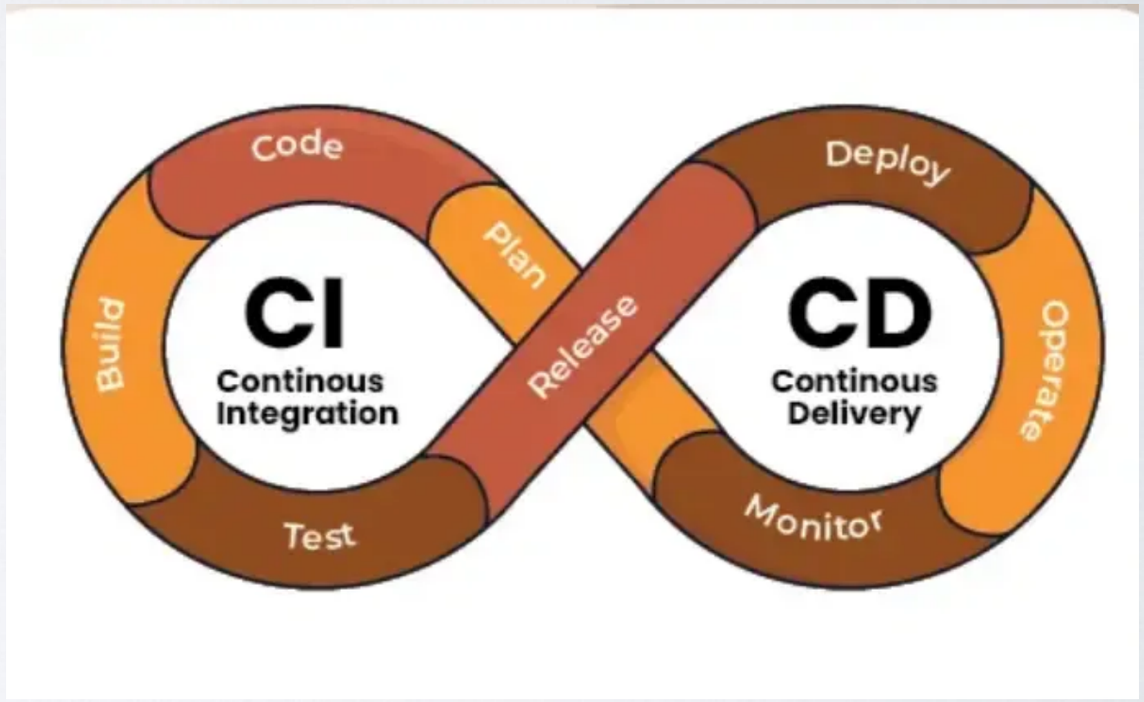
Cloud Native Technologies Pyramid

O QUE É CI/CD?

- **Integração Contínua (CI)**: Processo pelo qual novas alterações de código são integradas ao código-base de forma frequente, validada e automatizada.

- **Entrega Contínua (CD)**: Processo de entregar o artefato produzido (binário, imagem Docker, pacote de deploy) até o ambiente alvo de forma padronizada e automatizada.

- **Pipeline**: une integração e entrega como parte da arquitetura operacional.



INTEGRAÇÃO - EVOLUÇÃO

- (70 - 90) Código **local** sem versionamento: Alterações feitas diretamente no ambiente de desenvolvimento, compilação local e ausência de controle de histórico.

- (90-2005) Repositórios centralizados e builds automatizados: Uso de ferramentas como CVS/**SVN**, commits centralizados e início de scripts de build executados em servidores.

- (2005 >) Versionamento distribuído e segmentação de fluxos: **Git** populariza branches de desenvolvimento, feature branches e estratégias formais de integração.

- (2010 >) Integração controlada via Pull Requests: Revisão de código, controle de múltiplas versões de uma mesma feature e **pipelines** automatizados de build.

- (2013 >) **Containers** e estratégias avançadas de liberação: Builds imutáveis, entrega padronizada e suporte a canary releases, blue/green e observabilidade integrada.

ENTREGA - EVOLUÇÃO

- (80–2000) Entrega física: Distribuição por disquetes, CDs e **mídias físicas**.
- (1995–2010) Entrega via download estático: Disponibilização de **binários em sites** para download manual ou via gerenciadores simples.
- (2010 >) Entrega transparente e automatizada: **Autoatualização**, integração com webservers e atualização contínua de binários.
- (2015+) Entrega progressiva e controlada: **Roteamento de tráfego**, feature flags, A/B testing e regressão automática.

ESTADO DA ARTE

- GitOps é a evolução natural de CI/CD aplicada à operação de ambientes.

- A infraestrutura e as configurações passam a ser tratadas como estado declarativo versionado em repositórios Git.

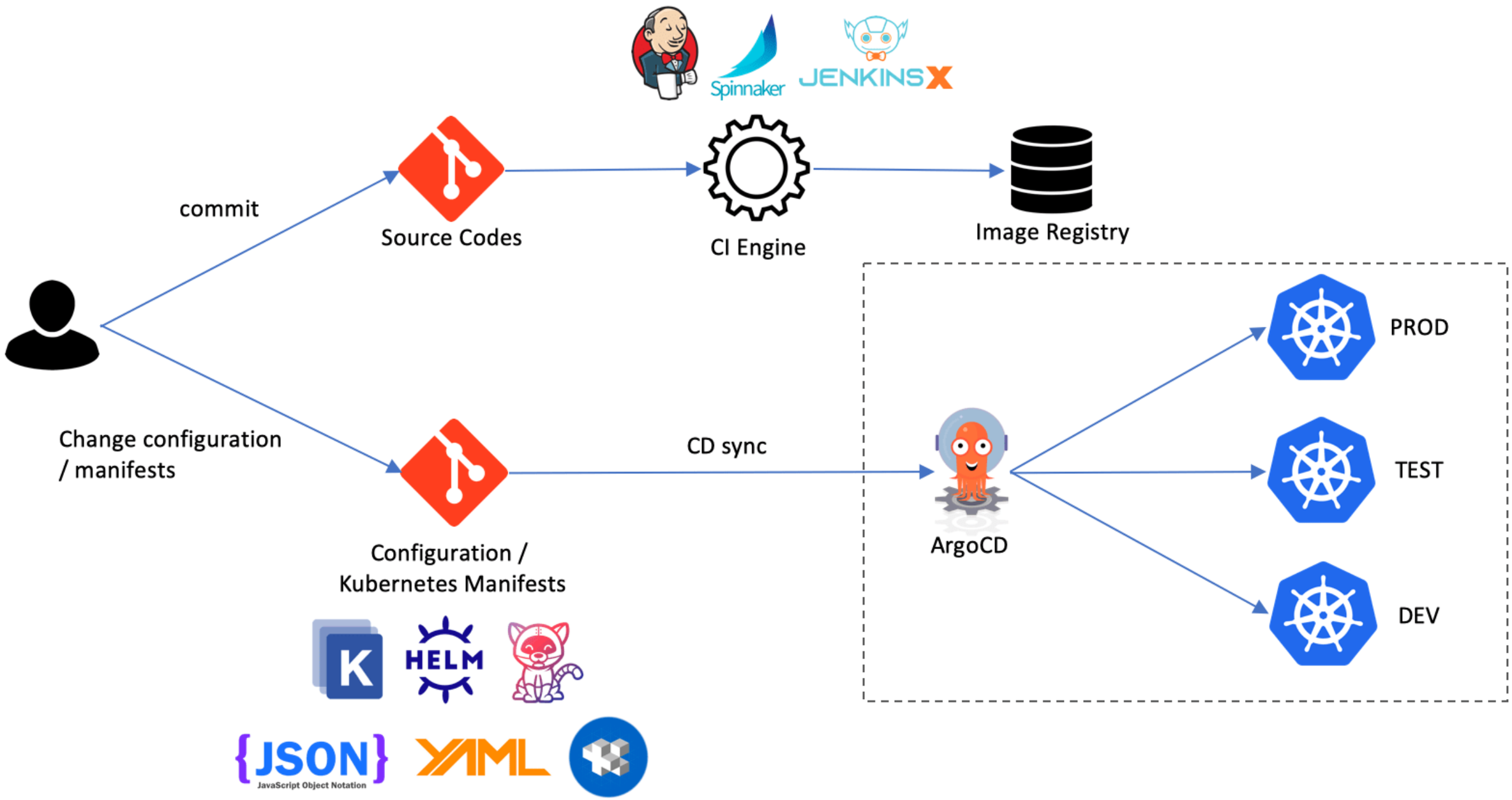
- A implantação deixa de ser "enviar comandos" e passa a ser sincronizar GIT e Cluster (conjunto de pods/containers);

- Ferramentas como ArgoCD e FluxCD monitoram o repositório e aplicam automaticamente as mudanças.

- GitOps aproxima a configuração do ambiente ao mesmo rigor que já aplicamos ao código: versionamento, PRs, revisão e histórico.

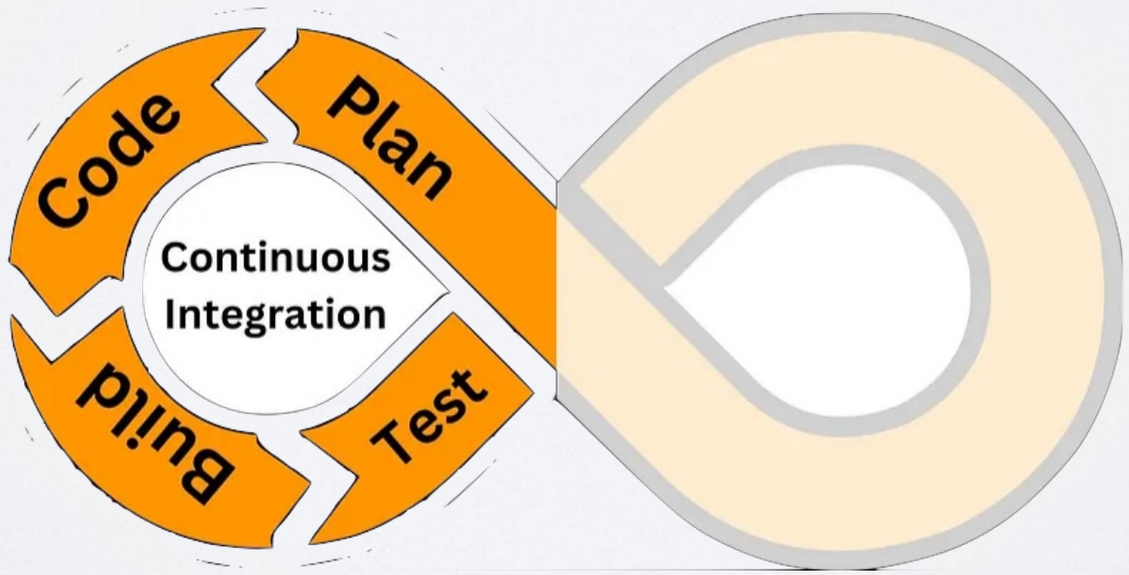
- Infra as a Code - scripts criam e liberam recursos de computação de forma automática.

GITOPS


Spinnaker
JENKINSX
commit
Source Codes
CI Engine
Image Registry
PROD
Change configuration
/ manifests
CD sync
TEST
ArgoCD
Configuration /
Kubernetes Manifests
DEV
HELM
{JSON}
JavaScript Object Notation
YAML

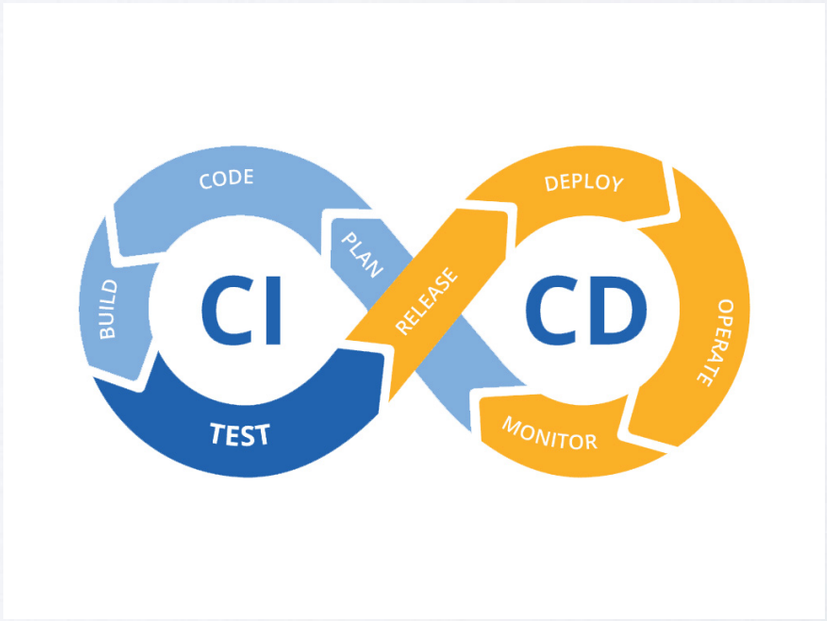
CONTINUOUS INTEGRATION

- Cada **mudança** de código **dispara** automaticamente um processo de **build** e teste.
- **Testes** automatizados (unitários, linters, análise estática) garantem integridade.
- **Feedback** rápido reduz o custo de correção e evita acúmulo de erros.
- **Em microservices, cada repositório possui o seu próprio pipeline de CI.**



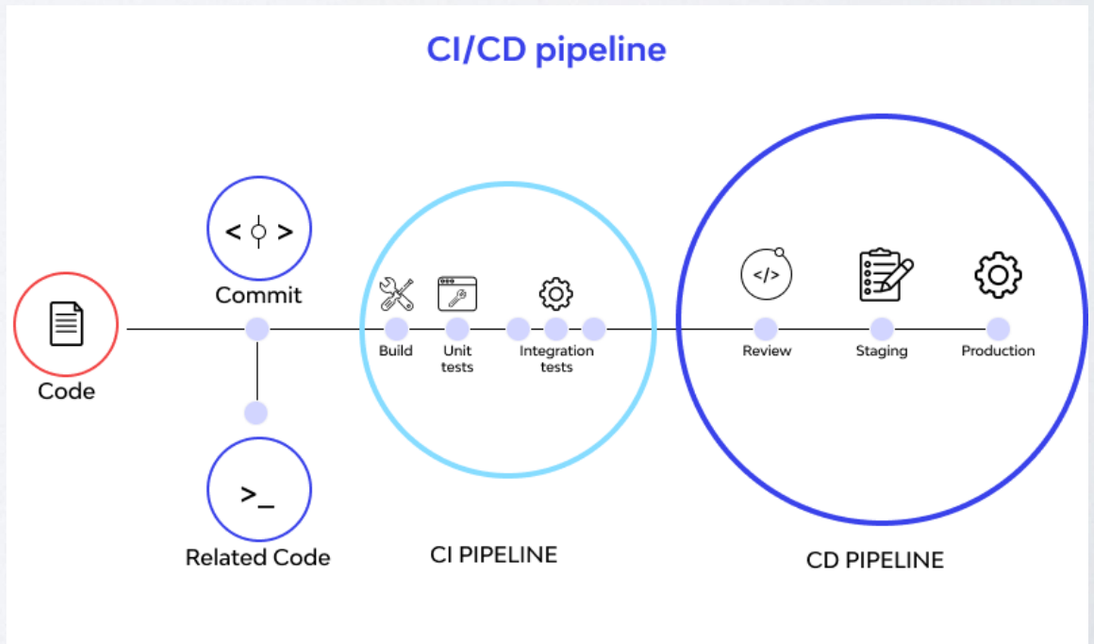
CONTINUOUS DELIVERY

- Amplia a CI, automatizando o deploy em ambientes de teste e produção.
- Exige infraestrutura reprodutível e scripts de deploy padronizados.
- Suporta estratégias de liberação como Blue/Green, Canary e Feature Flags.
- Em microservices, cada serviço pode ser implantado de forma independente.

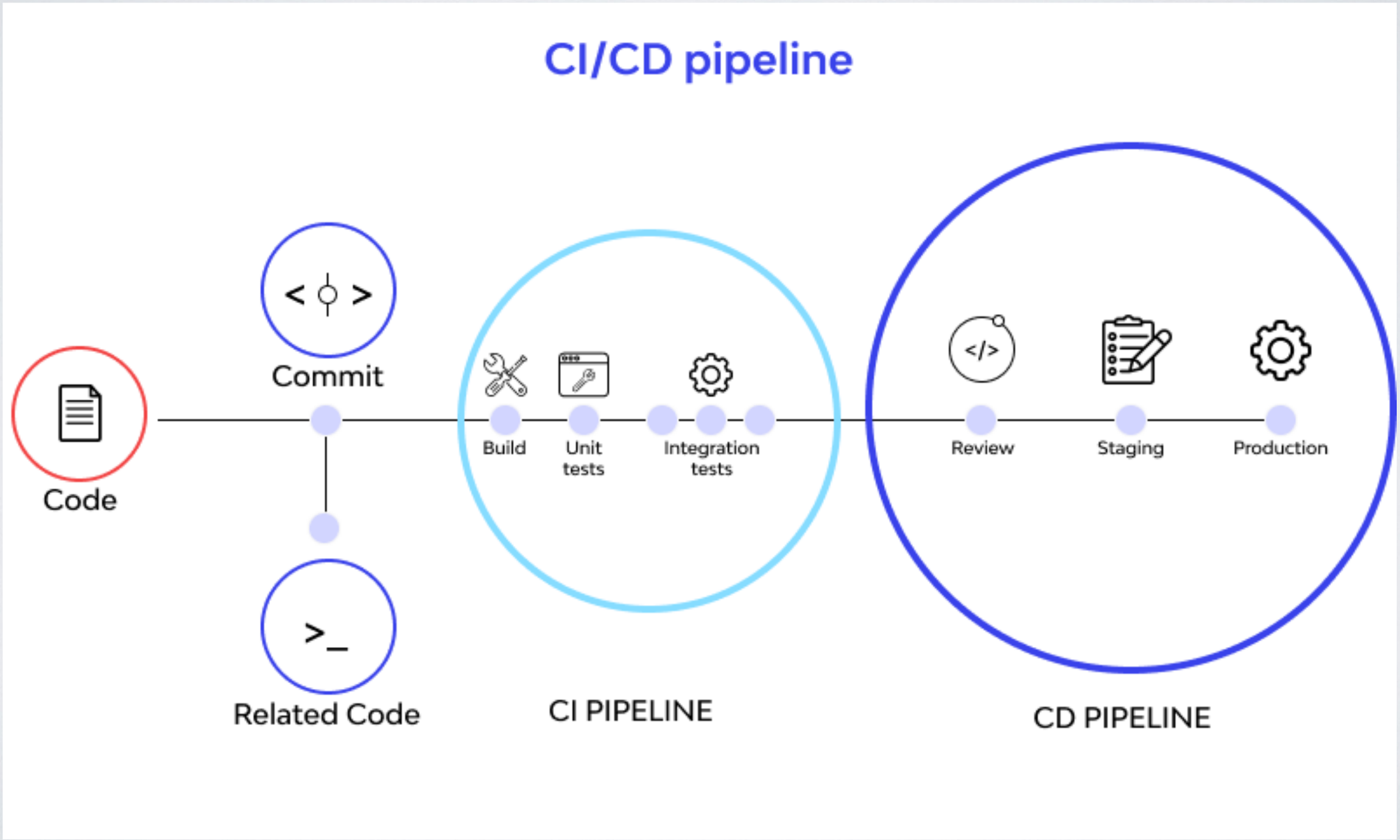


PIPELINES

- Cada microservice possui seu próprio repositório e pipeline de CI/CD.
- Builds, testes e deploys podem ocorrer em paralelo, sem bloquear o sistema inteiro.
- Pipelines independentes reduzem acoplamento operacional entre serviços.
- Ferramentas como ArgoCD, Jenkins, GitHub Actions e GitLab CI orquestram esses fluxos.



PIPELINES


CI/CD pipeline
Commit
Code
Related Code
Build
Unit tests
Integration tests
CI PIPELINE
Review
Staging
Production
CD PIPELINE

TOOLS

- **CI**
  - GitHub Actions → pipelines YAML nativos ao GitHub, ideal para repositórios públicos/privados e microservices isolados.
  - GitLab CI → pipelines acoplados ao repositório GitLab com runners dedicados.
  - Jenkins → orquestrador clássico, altamente customizável, excelente para ambientes on-premise.
- **CD**
  - GitHub Actions (Workflows de Deploy) → entrega automatizada sem GitOps.
  - ArgoCD → GitOps para Kubernetes, sincroniza estado do cluster com o Git.
  - FluxCD → alternativa GitOps leve, também orientada a Kubernetes.
  - Jenkins X → CD automatizado para cloud native.

AUTOMAÇÃO

- A automação permite que as ferramentas **executem**, de forma controlada, as **etapas** do processo de CI/CD, seguindo **instruções** previamente definidas no projeto;

- Essa execução automatizada ocorre conforme o protocolo, comandos e capacidades de cada ferramenta envolvida (**GitHub Actions**, Jenkins, Docker, entre outras);

- Em geral, essa lógica é descrita em um arquivo **YAML** incluído no projeto, que orienta a ferramenta de orquestração a interpretar, organizar e sequenciar cada passo do pipeline.

AUTOMAÇÃO

- Além disso, o uso das ferramentas pode ser (e normalmente é) misto, com um orquestrado por utilizar diferentes soluções para da etapa;

- O pipeline mais básico em uma estrutura moderna é o que, após um commit, executa o conjunto de testes definido na aplicação, executa o merge com repositório master (etapa CI), cria novo container e o disponibiliza em um servidor remoto (etapa CD).

- Portanto, essas são as etapas básicas esperadas em um YAML.

SCRIPTS DE CONFIGURAÇÃO

- Arquivos com configuração declarativa, determinando:
  - **Quando** o pipeline roda (gatilhos: push, PR, tags, branches).
  - **Onde** o pipeline roda (máquinas virtuais, runners locais, containers).
  - **O que** o pipeline faz (build, testes, criação de imagens, deploy).
  - **Como** isso deve ser realizado (comandos, ações pré-definidas, uso de secrets).

ESTRUTURA ARQUIVO YAML

- CI
  - Executa automaticamente após push ou PR.
  - Instala dependências e executa testes.
  - Garante que o código está íntegro antes de ser entregue.
    - Somente se a CI for bem-sucedida, o CD é ativado
- Fase de CD (Entrega Contínua)
  - Constrói a imagem Docker localmente (sockshop-carts:local).
  - Para e remove o container anterior, se existir.
  - Sobe um novo container com a versão mais recente do microserviço.
  - Simula a ideia de "nova versão em produção".

ESTRUTURA ARQUIVO YAML

```yaml
name: CI-CD PIPELINE                    # NOME DO PIPELINE EXIBIDO NO GITHUB ACTIONS

on:                                     # EVENTOS QUE DISPARAM O PIPELINE
  push:                                 # DISPARA QUANDO HÁ PUSH
    branches: ["main"]                  # LISTA DE BRANCHES MONITORADAS
  pull_request:                         # DISPARA EM ABERTURA/ATUALIZAÇÃO DE PR
    branches: ["main"]                  # BRANCH ALVO DO PULL REQUEST

jobs:                                   # AGRUPA OS JOBS (ETAPAS PRINCIPAIS DO PIPELINE)

  ci:                                   # NOME DO JOB DE INTEGRAÇÃO CONTÍNUA (CI)
    runs-on: ubuntu-latest              # DEFINE O SISTEMA ONDE O JOB SERÁ EXECUTADO
    steps:                              # LISTA DE PASSOS QUE SERÃO EXECUTADOS NO JOB

      - name: CHECKOUT                  # NOME DO PASSO (APENAS DESCRITIVO)
        uses: actions/checkout@v4       # AÇÃO QUE BAIXA O CÓDIGO DO REPOSITÓRIO

      - name: SETUP                     # PREPARA O AMBIENTE (LINGUAGEM, SDK, ETC.)
        uses: actions/setup-node@v4     # EXEMPLO: CONFIGURA NODE.JS

      - name: INSTALL DEPENDENCIES      # INSTALA DEPENDÊNCIAS DO PROJETO
        run: npm install                # COMANDO EXECUTADO NA MÁQUINA VIRTUAL

      - name: RUN TESTS                 # EXECUTA OS TESTES AUTOMATIZADOS
        run: npm test                   # COMANDO DE TESTES

  cd:                                   # JOB DE ENTREGA CONTÍNUA (CD)
    needs: ci                           # SÓ EXECUTA SE O JOB CI TERMINAR COM SUCESSO
    runs-on: ubuntu-latest              # AMBIENTE DO JOB DE CD

    steps:                              # PASSOS DO JOB DE ENTREGÁVEL

      - name: CHECKOUT                  # NECESSÁRIO PARA TER ACESSO AOS ARQUIVOS
        uses: actions/checkout@v4

      - name: BUILD DOCKER IMAGE        # CONSTRÓI A IMAGEM DOCKER LOCALMENTE
        run: docker build -t app:latest .

      - name: RUN CONTAINER             # SOBE O CONTAINER LOCAL COMO "AMBIENTE"
        run: docker run -d -p 8080:80 app:latest
```

EXEMPLO BASE SOCKSHOP

```yaml
# .github/workflows/ci-cd-local.yml
name: CI/CD Local - Sock Shop (carts)

on:
  push:
    branches: ["main", "develop"]
  pull_request:
    branches: ["main", "develop"]

env:
  IMAGE_NAME: sockshop-carts:local

jobs:
  ci:
    name: Integração Contínua (build + testes)
    runs-on: ubuntu-latest

    steps:
      - name: Checkout do código
        uses: actions/checkout@v4

      # Exemplo para serviço em Node.js (ajuste conforme sua stack)
      - name: Configurar Node.js
        uses: actions/setup-node@v4
        with:
          node-version: "20"

      - name: Instalar dependências
        run: npm install
        working-directory: ./services/carts

      - name: Executar testes
        run: npm test
        working-directory: ./services/carts

      - name: Build da aplicação (se houver)
        run: npm run build
        working-directory: ./services/carts
```

```yaml
  cd_local:
    name: Entrega Contínua (container local)
    # Para uso “real”, o ideal é um runner self-hosted,
    # pois o container vai subir nessa própria máquina.
    runs-on: ubuntu-latest
    needs: ci
    if: github.ref == 'refs/heads/main'

    steps:
      - name: Checkout do código
        uses: actions/checkout@v4

      - name: Build da imagem Docker local
        run: |
          docker build \
            -t $IMAGE_NAME \
            ./services/carts

      - name: Parar container anterior (se existir)
        run: |
          docker stop sockshop-carts || true
          docker rm sockshop-carts || true

      - name: Subir novo container local
        run: |
          docker run -d \
            --name sockshop-carts \
            -p 8080:80 \
            $IMAGE_NAME
```

ESTRUTURA ARQUIVO YAML

- CI
  - Executa automaticamente após push ou PR.
  - Instala dependências e executa testes.
  - Garante que o código está íntegro antes de ser entregue.
    - Somente se a CI for bem-sucedida, o CD é ativado
- Fase de CD (Entrega Contínua)
  - Constrói a imagem Docker localmente (sockshop-carts:local).
  - Para e remove o container anterior, se existir.
  - Sobe um novo container com a versão mais recente do microserviço.
  - Simula a ideia de "nova versão em produção".

PROJETO UNIDADE 3

- Configurar CI e CD de ao menos 1 microserviço do projeto sockshop;
- Para efeitos pedagógicos, o deploy pode ser feito localmente, mas deve-se obrigatoriamente utilizar:
  - GitHub Actions
  - Docker.

BIBLIOGRAFIA

1. GALLOTTI, G. M. A. Arquitetura de Software. 1ª Ed. São Paulo: Pearson, 2016. [acervo digital]

2. JUNIOR, A. K. Computação em Nuvem. 1ª Ed. Curitiba: Contentus, 2020. [acervo digital]

3. ZANETTI, M. Arquitetura de TI e modelos de negócios. 1ª Ed. Curitiba: Contentus, 2020. [acervo digital]

4. Baseado nos materiais de aula de Vinícius Garcia - CIn UFPE

DÚVIDAS?