Para uma prova de conceito (PoC) de um pipeline de CI/CD, podemos criar um projeto simples, como uma aplicação web em Python usando o framework Flask.

O objetivo é implementar um pipeline de CI/CD que inclui etapas de build, teste, e deploy. Vamos definir um pipeline com sucesso e outro com falha, explicando o passo a passo de cada um.

1. Setup do Projeto

- **Linguagem e Ferramentas:** Python com Flask para a aplicação web; GitHub para versionamento; GitHub Actions (ou outra ferramenta de CI/CD como Jenkins, GitLab CI) para o pipeline.
- **Repositório:** Criar um repositório Git no GitHub e adicionar o código da aplicação Flask. Incluir arquivos como `requirements.txt` (para dependências) e o próprio script da aplicação (`app.py`).

2. Definição do Pipeline CI/CD

O pipeline será dividido em três etapas principais: build, teste e deploy.

Pipeline com Sucesso

1. **Etapa de Build:**
    - **Instalação de Dependências:** Instalar dependências listadas no arquivo `requirements.txt` usando `pip`.
    - **Verificação de Sintaxe:** Executar uma ferramenta de linting, como `flake8`, para garantir que o código esteja conforme os padrões de estilo Python.
2. **Etapa de Teste:**
    - **Execução de Testes Automatizados:** Utilizar uma biblioteca de testes como `pytest` para rodar testes unitários previamente escritos. Testar funcionalidades da aplicação Flask, como resposta a rotas HTTP.
3. **Etapa de Deploy:**
    - **Deploy Automatizado:** Após os testes passarem, o pipeline faz o deploy automático para um ambiente de staging, usando uma plataforma como Heroku, AWS ou Docker containers.

Pipeline com Falha

1. **Etapa de Build com Falha:**
    - **Erro na Instalação de Dependências:** Introduzir um erro no arquivo `requirements.txt`, como uma versão inexistente de um pacote (`flask==999.99.99`), o que causará a falha na instalação de dependências.
    - **Linting com Falha:** Inserir erros de formatação no código Python, como identação incorreta ou uso de variáveis não definidas, para causar falhas na verificação de sintaxe.
2. **Etapa de Teste com Falha:**
    - **Testes Automatizados com Erro:** Introduzir uma falha lógica no código (ex.: uma rota HTTP retorna um código de status errado). Isso fará com que um ou mais testes unitários falhem, resultando em uma quebra na etapa de testes.
3. **Etapa de Deploy Cancelado:**
    - Como o pipeline CI/CD é configurado para interromper o fluxo ao detectar falhas em etapas anteriores, a etapa de deploy não será iniciada até que todos os problemas sejam corrigidos.

3. Implementação do Pipeline com GitHub Actions

Arquivo de Workflow do GitHub Actions (`.github/workflows/ci-cd.yml`):

```
name: CI/CD Pipelineon:  push:    branches:      - mainjobs:
```

4. Execução e Demonstração do Pipeline

- **Pipeline com Sucesso:** Todas as etapas serão concluídas com sucesso se não houver erros nas dependências, no código e nos testes.
- **Pipeline com Falha:**
    - **Erro na Instalação de Dependências:** O pipeline falhará na etapa de build devido à versão inexistente de uma dependência.
    - **Erro de Linting:** O pipeline também falhará se houver erros de estilo de código.
    - **Erro nos Testes:** Mesmo que as dependências sejam instaladas corretamente, o pipeline falhará se houver um erro nos testes automatizados.

5. Análise dos Resultados e Feedback

Após executar o pipeline, os alunos devem analisar os logs gerados para entender onde e por que cada falha ocorreu. Eles devem propor soluções para corrigir os erros, reexecutar o pipeline com as correções e discutir as lições aprendidas com o processo de depuração.