# Sistema de Gerenciamento de Tarefas

Plano de Gerenciamento de Configuração e Pipeline CI/CD

Gerência de Configuração e Evolução de Software

Dezembro 2025

# Objetivos do Trabalho

- Criar um Plano de Gerenciamento de Configuração completo
- Implementar pipeline CI/CD automatizado
- Configurar testes automatizados (unidade, integração, aceitação)
- Implementar containerização com Docker
- Aplicar princípios de DevOps

# Sistema de Gerenciamento de Tarefas

**Aplicação web desenvolvida em Python/Flask**

- 10+ classes/arquivos, 20+ métodos/funções
- Arquitetura em camadas (Models, Services, Database)
- Interface web moderna e responsiva
- Funcionalidades: CRUD de tarefas, categorias, usuários

# Plano de Gerenciamento de Configuração

**Versionamento Semântico (SemVer)**

- Formato: MAJOR.MINOR.PATCH

**Estrutura de branches**

- main: código estável
- feature/*: novas funcionalidades
- bugfix/*: correções de bugs
- hotfix/*: correções urgentes

**Processo de Pull Request e Code Review**

**CCB (Conselho de Controle de Mudanças)**

# Ferramentas e Tecnologias

**Git & GitHub**
**Controle de versão**

Python 3.11
Linguagem de programação

Flask 3.0.0
Framework web

pytest 7.4.3
Framework de testes

**Docker**
**Containerização**

GitHub Actions
CI/CD Pipeline

pytest-cov
Cobertura de código

Python-pptx
Geração de slides

# Pipeline CI/CD - Etapa de Commit

**1. Build da Aplicação**

- Compilação de todos os arquivos Python
- Verificação de sintaxe e erros

**2. Testes de Unidade**

- test_models.py (7 testes)
- test_services.py (13 testes)

**3. Testes de Integração**

- test_integration.py (9 testes)

**4. Relatório de Cobertura**

- Geração de relatórios HTML e XML

# Pipeline CI/CD - Testes de Aceitação

**Execução de testes end-to-end**

- Fluxo completo do sistema
- Validação de requisitos funcionais

**Verificação de Requisitos Funcionais**

- CRUD completo de tarefas
- Gerenciamento de categorias
- Operações de usuários

**Verificação de Requisitos Não-Funcionais**

- Performance (< 1s para operações básicas)
- Disponibilidade do sistema

# Pipeline CI/CD - Etapa de Lançamento

**1. Build da Imagem Docker**

- Criação da imagem containerizada
- Tagging com versão e SHA do commit

**2. Verificação da Imagem**

- Teste de execução do container
- Validação do health check

**3. Preparação de Artefatos**

- Pacote de deploy gerado automaticamente
- Documentação de instalação

**4. Entrega do Sistema**

- Artefatos disponíveis para download

# Estratégia de Testes

**Testes de Unidade**
**20 testes**

- Modelos
  - Task
  - User
  - Category
- Serviços
  - TaskService
  - UserService
  - CategoryService

Cobertura: 79%

**Testes de Integração**
**9 testes**

- API REST completa
- Endpoints testados
- Health check

Testes de Aceitação
3 testes

- Fluxo completo
- Performance
- Requisitos

# Containerização com Docker

### Dockerfile configurado para produção

- Base: Python 3.11-slim
- Instalação automática de dependências
- Otimização de camadas

### Configuração

- Porta 5000 exposta
- Variáveis de ambiente configuradas
- Comando de inicialização otimizado

### Automação

- Build e deploy automatizados no pipeline
- Imagem testada antes do deploy

# Resultados Alcançados

- Plano de Gerenciamento de Configuração completo
- Pipeline CI/CD automatizado com 3 etapas
- 43 testes automatizados implementados
- Cobertura de código de 79%
- Containerização com Docker funcional
- Integração e entrega contínua implementadas
- Documentação completa
- Processo de versionamento semântico estabelecido

# Lições Aprendidas

**Gerenciamento de Configuração**

- Fundamental desde o início do projeto
- Facilita rastreabilidade e manutenção

**Automação**

- Reduz erros humanos
- Acelera o desenvolvimento

**Testes Automatizados**

- Garantem qualidade do código
- Permitem refatoração com confiança

**CI/CD**

- Melhora confiabilidade do deploy
- Feedback rápido sobre problemas

**Containerização**

- Facilita deploy e manutenção
- Garante consistência entre ambientes

# Obrigado!