



# Arquitetura de Repositório



## Conceito e Motivação

Estrutura organizada para garantir manutenibilidade e escalabilidade do código.



## Separação de Responsabilidades

Cada componente com função específica e bem definida.

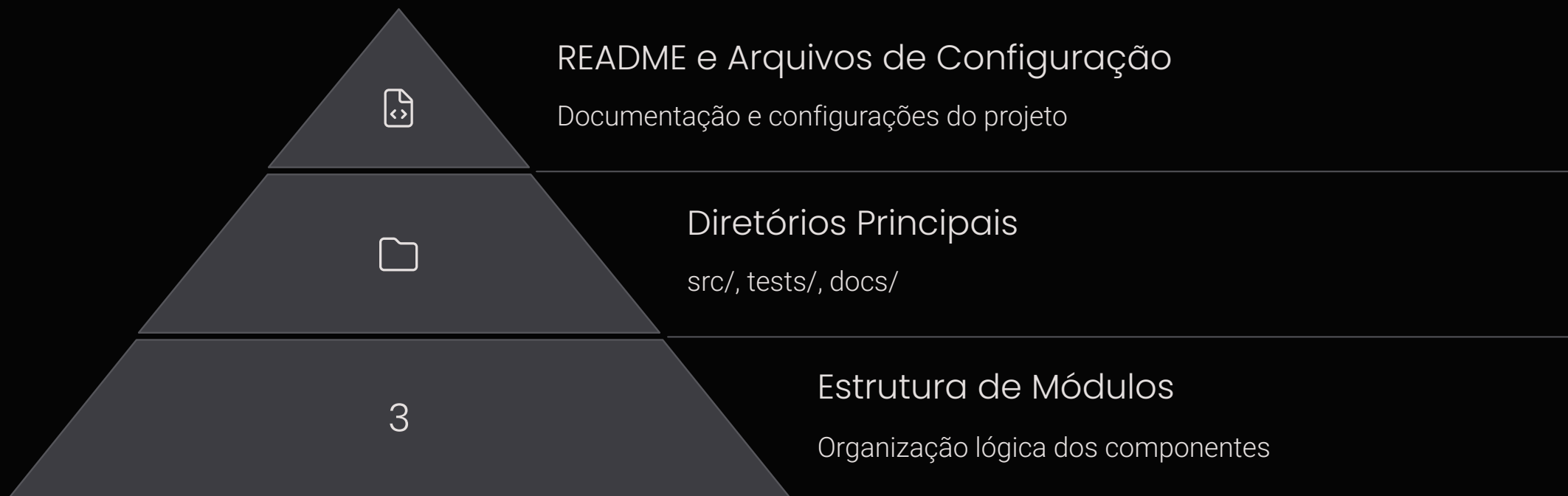


## Organização em Python

Estruturas de diretórios e pacotes seguindo boas práticas.



# Estrutura Básica de um Projeto Python



# Exemplo: Estruturando um Repositório Python

## Arquivos de Configuração

- README.md para documentação
- setup.py para instalação
- requirements.txt para dependências

## Estrutura de Diretórios

- src/ para código fonte
- tests/ para testes automatizados
- docs/ para documentação completa

# Principais Padrões de Estrutura

## Modularização

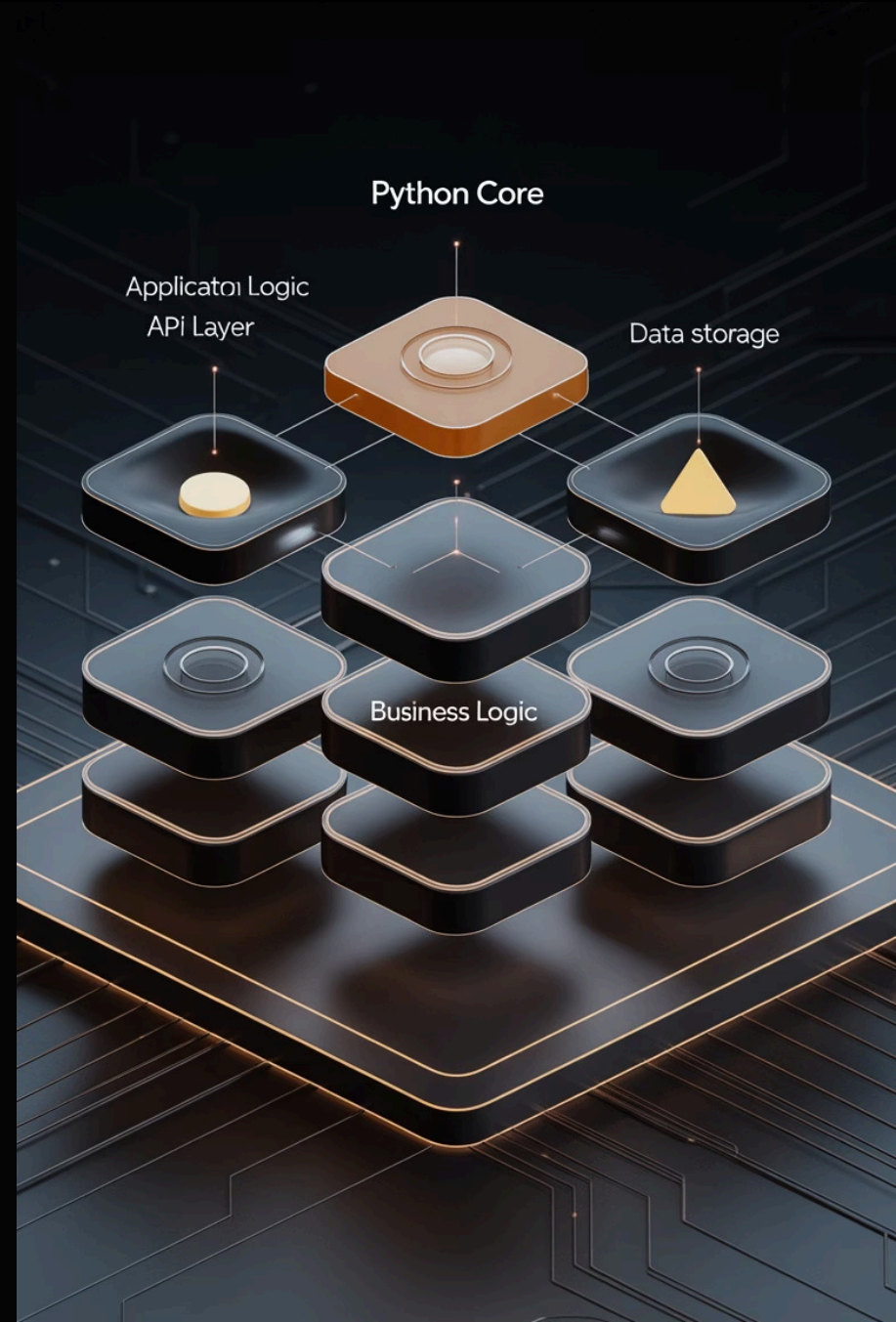
- Separar código em módulos coesos
- Organizar helpers, core e interfaces
- Evitar arquivos muito grandes

## Isolamento de Dependências

- Usar ambientes virtuais
- Separar dependências de desenvolvimento
- Versionar requirements.txt

## Convenções de Código

- Seguir PEP 8
- Usar nomes descritivos
- Documentar funções e classes



# Clean Architecture em Python



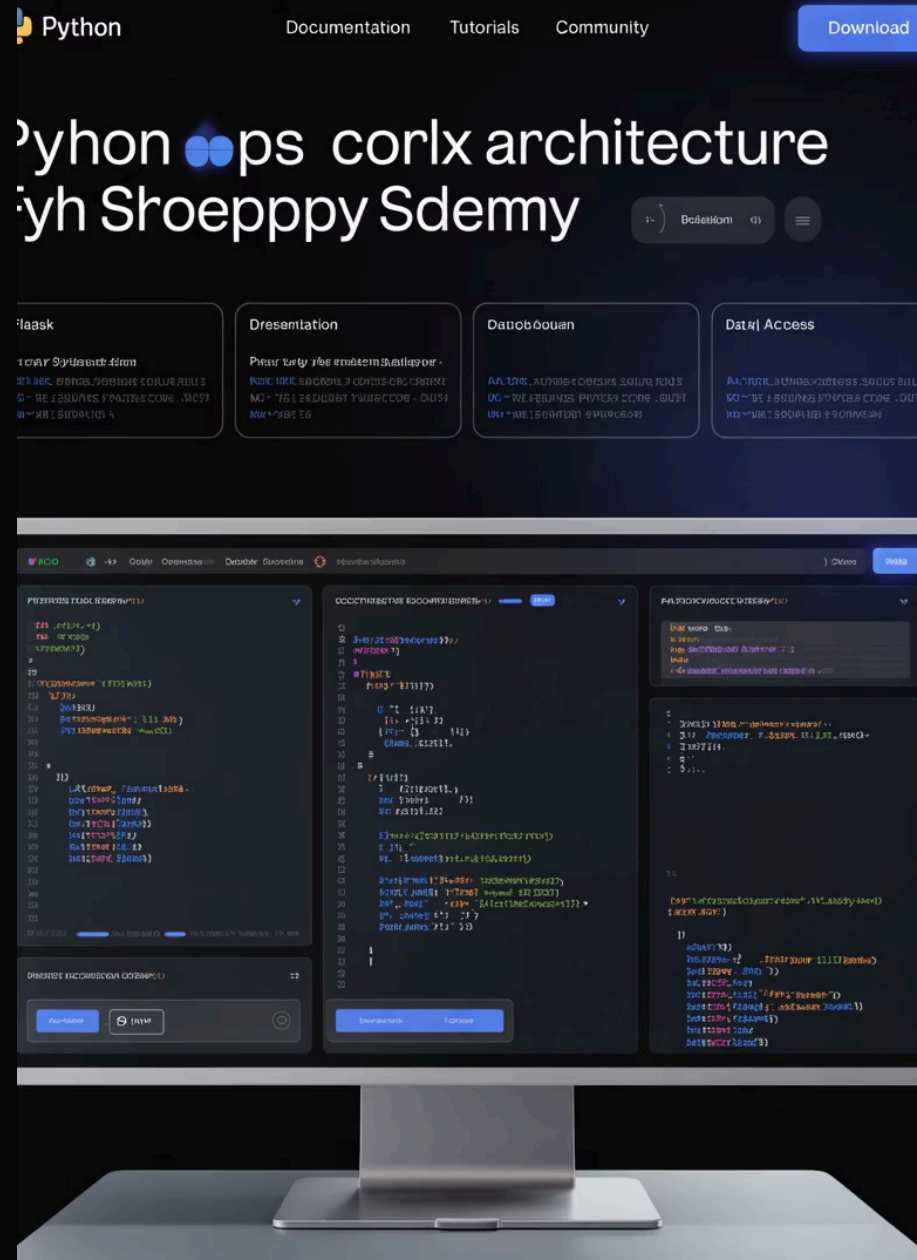


## Templates Flask e formulários

## Rotas Flask que recebem requisições

## Lógica de negócio independente

# SQLAlchemy e PostgreSQL



# Código: Interface e Implementação

## Interface Abstrata

```
import abc

class RepositorioABC(abc.ABC):
    @abc.abstractmethod
    def salvar(self, entidade):
        pass

    @abc.abstractmethod
    def buscar_por_id(self, id):
        pass
```

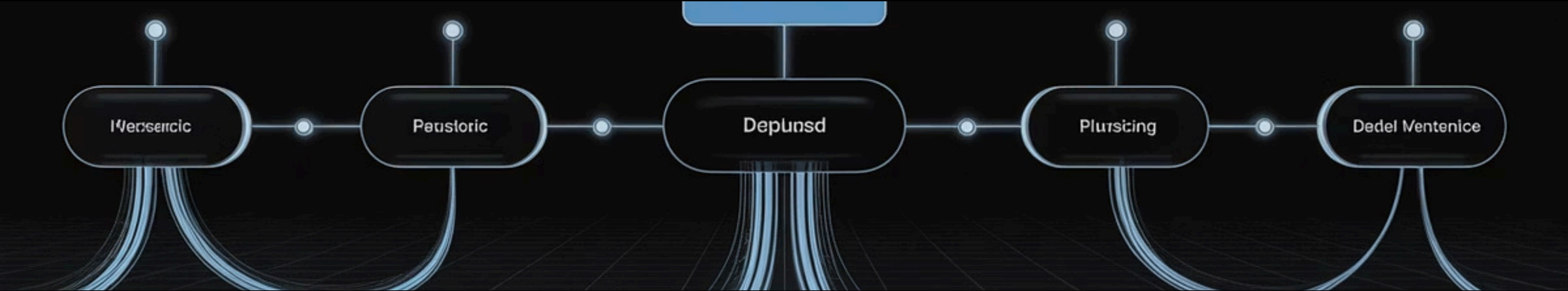
## Implementação Concreta

```
class ProdutoRepositorio(RepositorioABC):
    def __init__(self, session):
        self.session = session

    def salvar(self, produto):
        self.session.add(produto)
        self.session.commit()
        return produto

    def buscar_por_id(self, id):
        return self.session.query(
            Produto).filter_by(id=id).first()
```





# Aplicação em Machine Learning: MLOps

## Estruturação do Repositório

- Organização modular
- Separação de código, dados e modelos
- Documentação clara

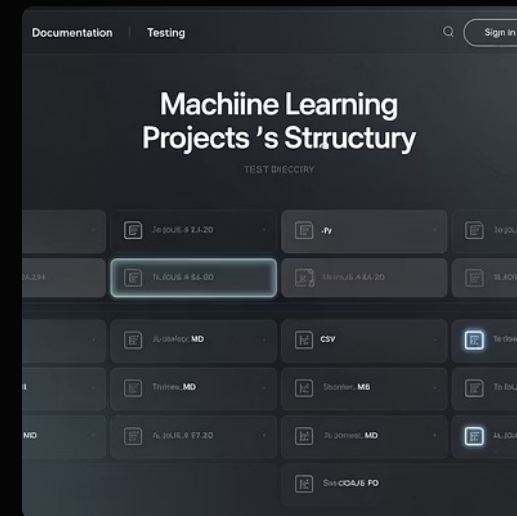
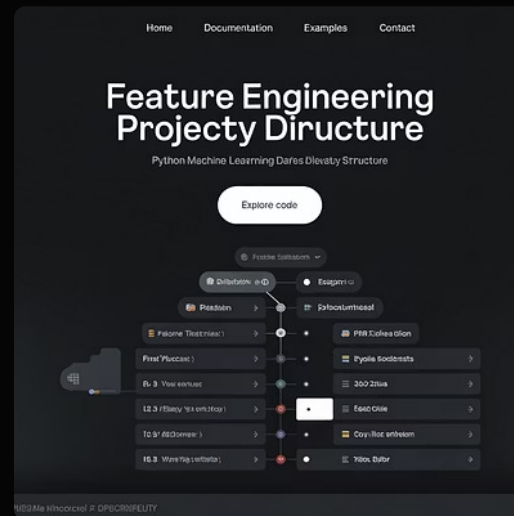
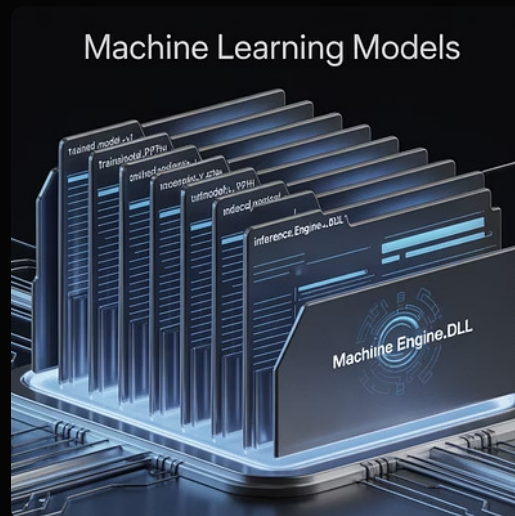
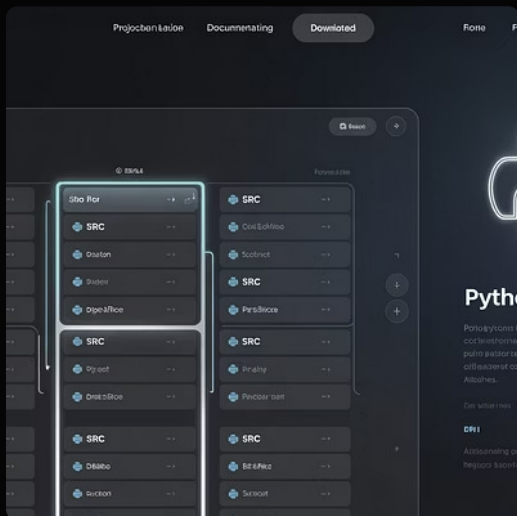
## Versionamento de Modelos

- MLflow para tracking de experimentos
- DVC para versionamento de dados
- Git para código

## Automatização de Pipeline

- CI/CD para treino e deploy
- Testes automatizados
- Monitoramento em produção

# Exemplo: Estrutura para Projeto ML



# Automação de Testes com Pytest



## Estrutura de Testes

Organizar testes em diretórios espelhando a estrutura do código.



## Tipos de Testes

Implementar testes unitários, integração e end-to-end.



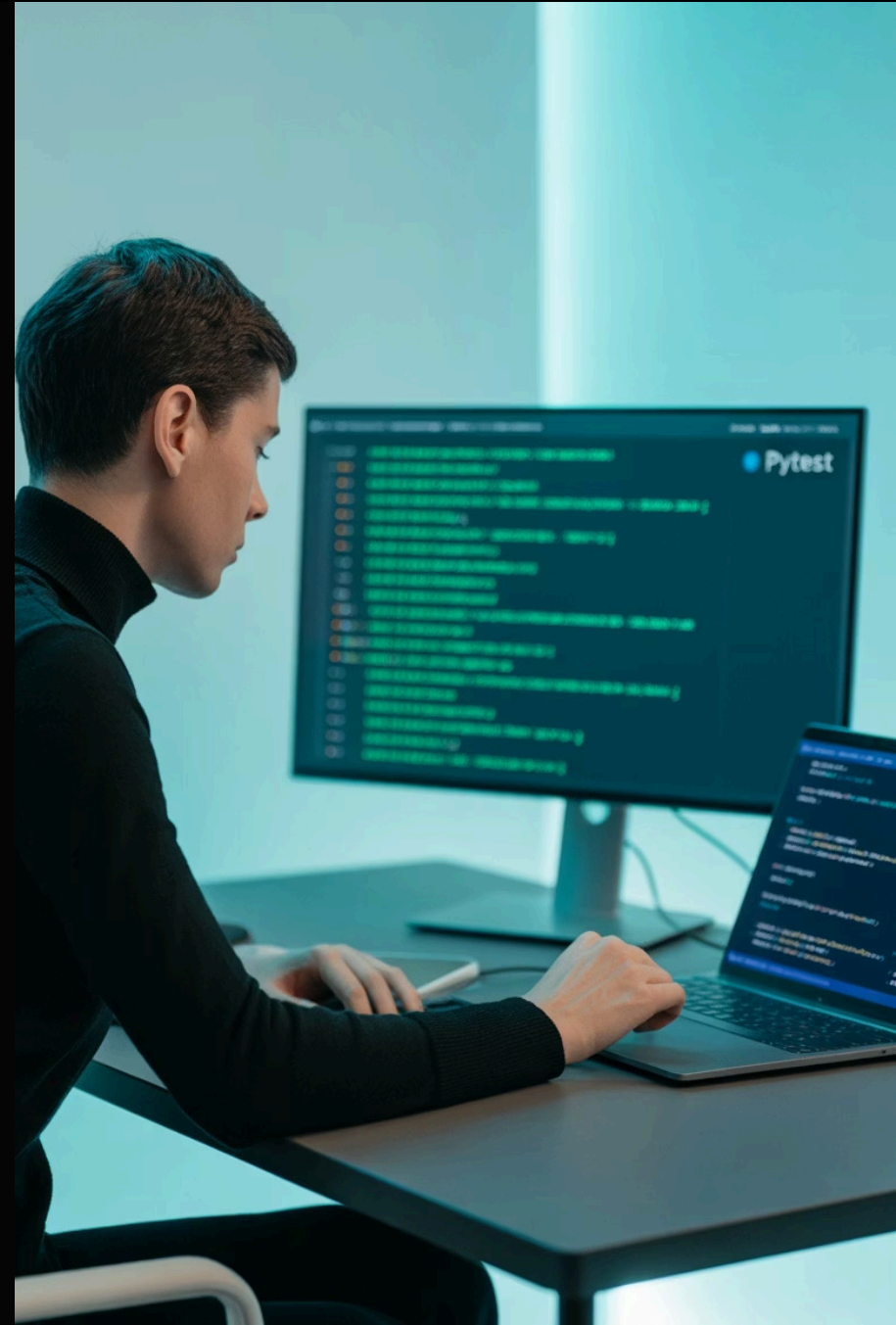
## Execução Automatizada

Integrar com CI/CD para validação contínua.



## Cobertura de Código

Monitorar métricas de cobertura para garantir qualidade.



# Organização de Dados no Repositório



## Dados Brutos

Armazenados em data/raw/ sem modificações

---



## Processamento

Pipeline transforma dados em data/processed/

---



## Resultados

Modelos e análises em output/

# Exercício Prático: Criando um Esqueleto de Projeto

## Estrutura a Criar

```
projeto_ml/  
├── README.md  
├── requirements.txt  
├── setup.py  
├── src/  
│   ├── __init__.py  
│   ├── data/  
│   ├── features/  
│   ├── models/  
│   └── visualization/  
├── tests/  
├── data/  
│   ├── raw/  
│   └── processed/  
└── notebooks/
```

## Comandos para Inicialização

```
# Criar estrutura  
mkdir -p projeto_ml/src/{data,features,models,visualization}  
mkdir -p projeto_ml/tests  
mkdir -p projeto_ml/data/{raw,processed}  
mkdir -p projeto_ml/notebooks  
  
# Inicializar arquivos  
touch projeto_ml/README.md  
touch projeto_ml/requirements.txt  
touch projeto_ml/setup.py  
touch projeto_ml/src/__init__.py
```

# Client-Server

## Arquitetura Cliente-Servidor



Cliente (Front-end)

Interface com usuário que solicita serviços



Servidor (Back-end)

Processa solicitações e fornece recursos



Comunicação

Protocolos HTTP, WebSocket, etc.

# Exemplo: Servidor Flask em Python

## Código do Servidor

```
from flask import Flask, request, jsonify

app = Flask(__name__)

@app.route('/api/dados', methods=['GET'])
def obter_dados():
    return jsonify({
        'mensagem': 'Dados recuperados',
        'status': 'sucesso',
        'dados': [1, 2, 3, 4, 5]
    })

@app.route('/api/dados', methods=['POST'])
def inserir_dados():
    dados = request.json
    # processamento aqui...
    return jsonify({
        'mensagem': 'Dados recebidos',
        'status': 'sucesso'
    })

if __name__ == '__main__':
    app.run(debug=True)
```

## Características

- API RESTful simples
- Suporte a métodos GET e POST
- Respostas em formato JSON
- Facilmente extensível



# Exemplo: Cliente Consumindo API

## Código do Cliente

```
import requests

# Consumindo endpoint GET
resposta = requests.get('http://localhost:5000/api/dados')
if resposta.status_code == 200:
    dados = resposta.json()
    print(f"Mensagem: {dados['mensagem']}")
    print(f"Dados: {dados['dados']}")
else:
    print(f"Erro: {resposta.status_code}")

# Enviando dados via POST
novos_dados = {'nome': 'Produto', 'valor': 29.99}
resposta = requests.post(
    'http://localhost:5000/api/dados',
    json=novos_dados
)
print(f"Resposta: {resposta.json()}")
```

## Características

- Biblioteca requests para HTTP
- Tratamento de códigos de status
- Envio e recebimento de JSON
- Estrutura simples e reutilizável

# Modelo de Comunicação: Stateless e Stateful

## Stateless

Cada requisição é independente e contém todas as informações necessárias.

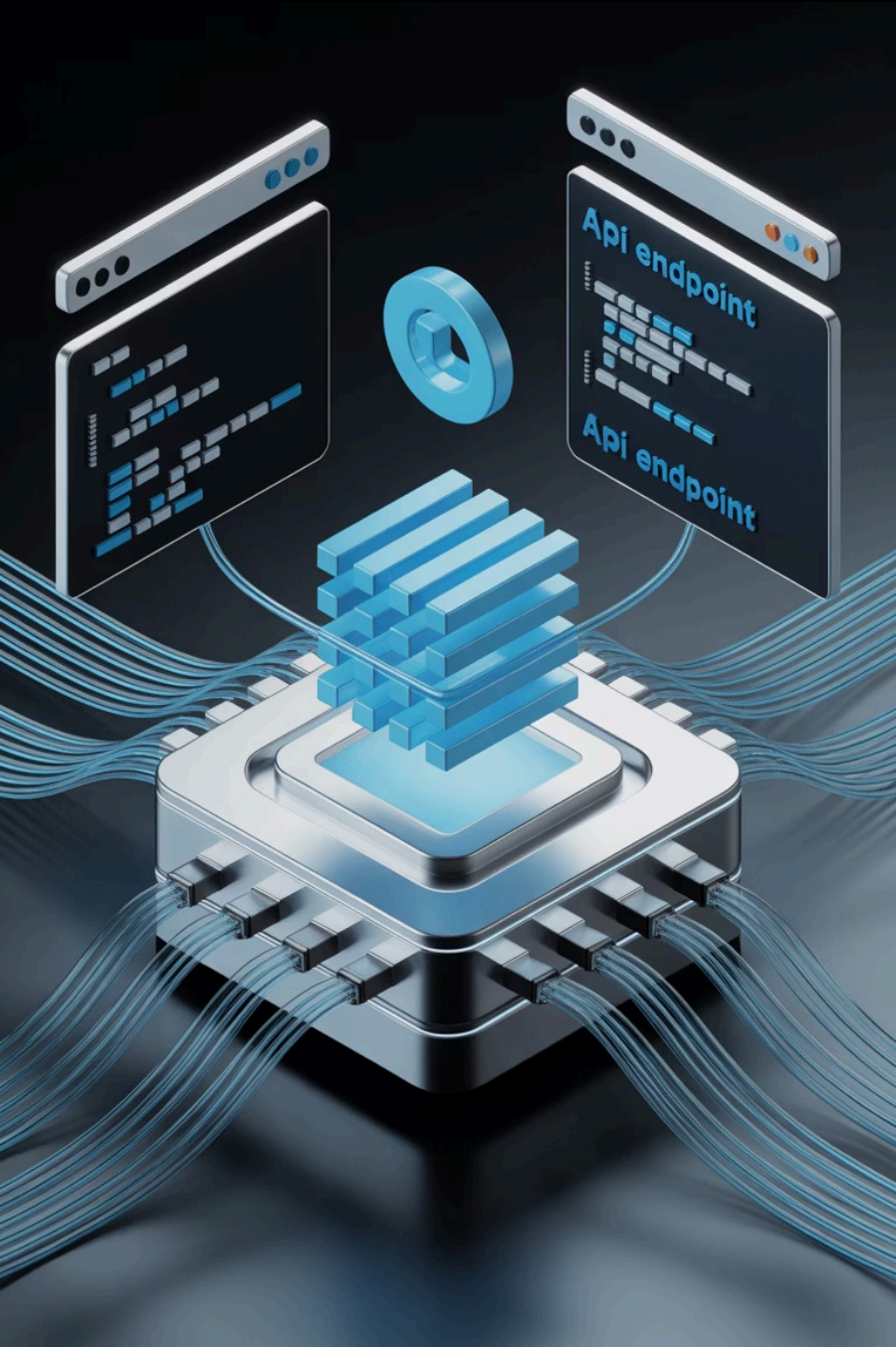
```
# Exemplo REST (Stateless)
@app.route('/api/produto/')
def obter_produto(id):
    # Busca produto pelo ID
    return jsonify(produto)
```

## Stateful

O servidor mantém informações sobre o cliente entre requisições.

```
# Exemplo com sessão (Stateful)
@app.route('/login')
def login():
    # Autentica usuário
    session['usuario_id'] = 123
    return redirect('/dashboard')

@app.route('/dashboard')
def dashboard():
    if 'usuario_id' not in session:
        return redirect('/login')
    # Mostra dashboard
```



# Deploy de APIs para Machine Learning



## Treinar Modelo

Desenvolver e treinar modelo ML em Python



## Serializar

Salvar modelo treinado com pickle/joblib



## Criar API

Desenvolver endpoint /predict com Flask/FastAPI



## Implantar

Deploy em servidor ou contêiner Docker

# Demonstração: Deploy de modelo ML como API

## Carregamento do Modelo

```
import joblib
from flask import Flask, request, jsonify

app = Flask(__name__)

# Carregar modelo pré-treinado
modelo = joblib.load('modelo_ml.joblib')

# Rota para previsões
@app.route('/predict', methods=['POST'])
def predict():
    # Obter dados da requisição
    dados = request.json

    # Fazer previsão
    resultado = modelo.predict([dados["features"]])

    # Retornar resultado
    return jsonify({
        'previsao': resultado[0].tolist()
    })

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5000)
```

## Testando a API

```
import requests
import json

# Dados para previsão
dados = {
    'features': [5.1, 3.5, 1.4, 0.2]
}

# Enviar requisição
resposta = requests.post(
    'http://localhost:5000/predict',
    json=dados
)

# Imprimir resultado
print(json.dumps(resposta.json(), indent=2))
```

# Arquiteturas Scaláveis: Load Balancer e Microserviços



# Exemplo: API Gateway Python

## Implementação de Gateway

```
from flask import Flask, request
import requests

app = Flask(__name__)

SERVIÇOS = {
    'usuarios': 'http://serviço-usuarios:5000',
    'produtos': 'http://serviço-produtos:5001',
    'pedidos': 'http://serviço-pedidos:5002'
}

@app.route('/', methods=['GET', 'POST'])
def gateway(serviço, caminho):
    if serviço not in SERVIÇOS:
        return {'erro': 'Serviço não encontrado'}, 404

    # Obter URL do serviço
    url_serviço = f"{SERVIÇOS[serviço]}/{caminho}"

    # Encaminhar requisição
    if request.method == 'GET':
        resp = requests.get(url_serviço, params=request.args)
    elif request.method == 'POST':
        resp = requests.post(url_serviço, json=request.json)

    return resp.json(), resp.status_code

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=8000)
```

## Benefícios

- Ponto único de acesso
- Roteamento de requisições
- Abstração da complexidade
- Possibilidade de implementar:
  - Autenticação centralizada
  - Rate limiting
  - Logging
  - Transformação de dados

# Python Service Monitoring

Alufon Cagdon



# Monitoramento de Serviços em Produção

## Logging Centralizado

- Utilizar biblioteca logging
- Integrar com ELK Stack
- Estruturar logs em JSON
- Níveis: DEBUG, INFO, WARN, ERROR

## Métricas de Aplicação

- Prometheus para coleta
- Grafana para visualização
- Métricas de latência e throughput
- Alertas para anomalias

## Health Checks

- Endpoints /health e /ready
- Monitoramento de dependências
- Verificação de conectividade
- Auto-recuperação



# Exercício: Rodar Cliente e Servidor Localmente

## Criar Servidor Flask

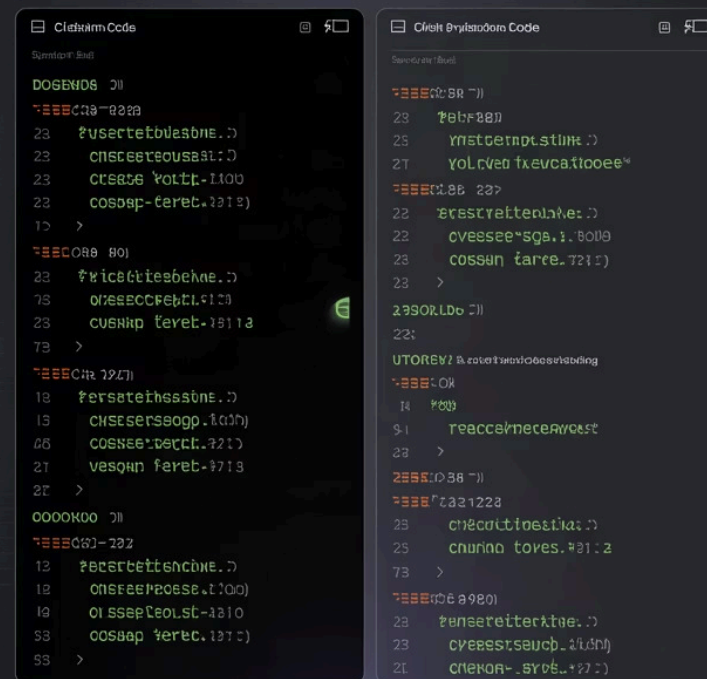
- Implementar endpoint /dados
- Salvar como servidor.py
- Executar com python servidor.py

## Implementar Cliente

- Criar script para consumir API
- Implementar GET e POST
- Salvar como cliente.py

## Testar Interação

- Executar cliente em terminal separado
- Verificar logs do servidor
- Analisar respostas recebidas



The image shows two terminal windows side-by-side. The left window is titled 'Client Python Code' and contains a Python script for a client that sends a POST request to a server. The right window is titled 'Server Python Code' and contains a Python script for a Flask server that receives a POST request and returns a response. Both scripts use the 'requests' and 'flask' libraries.

```
Client Python Code
import requests
url = "http://localhost:5000/dados"
data = {"nome": "João", "sobrenome": "Silva"}
response = requests.post(url, json=data)
print(response.json())

Server Python Code
from flask import Flask
app = Flask(__name__)

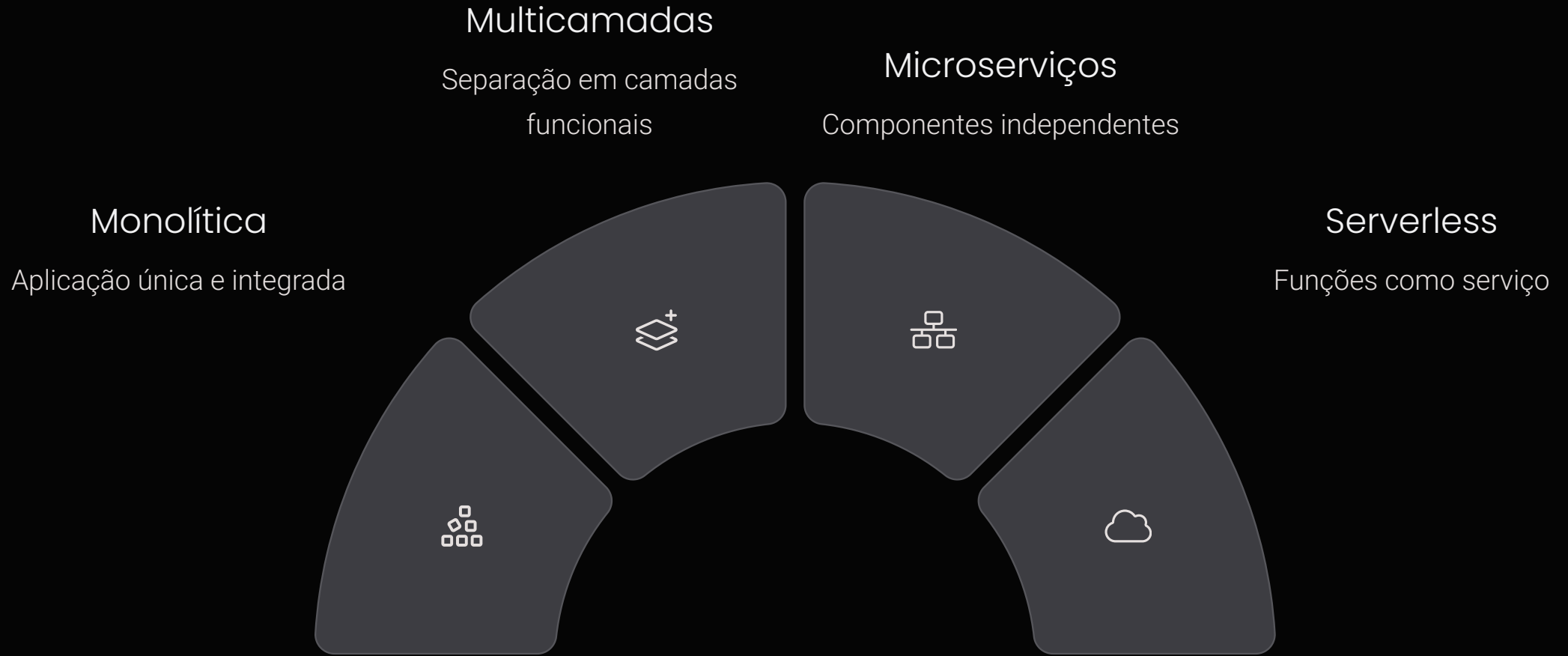
@app.route("/dados", methods=['POST'])
def dados():
    data = request.json
    return {"nome": data["nome"], "sobrenome": data["sobrenome"]}

if __name__ == '__main__':
    app.run()
```

Real-time Python development

# Real-time Python development

# Seção 3: Arquitetura de Aplicações



# Padrão MVC em Python



# Camadas de Aplicação



## Camada de Apresentação

Interface e interação com usuário

---

2

## Camada de Negócio

Regras e processamento de dados

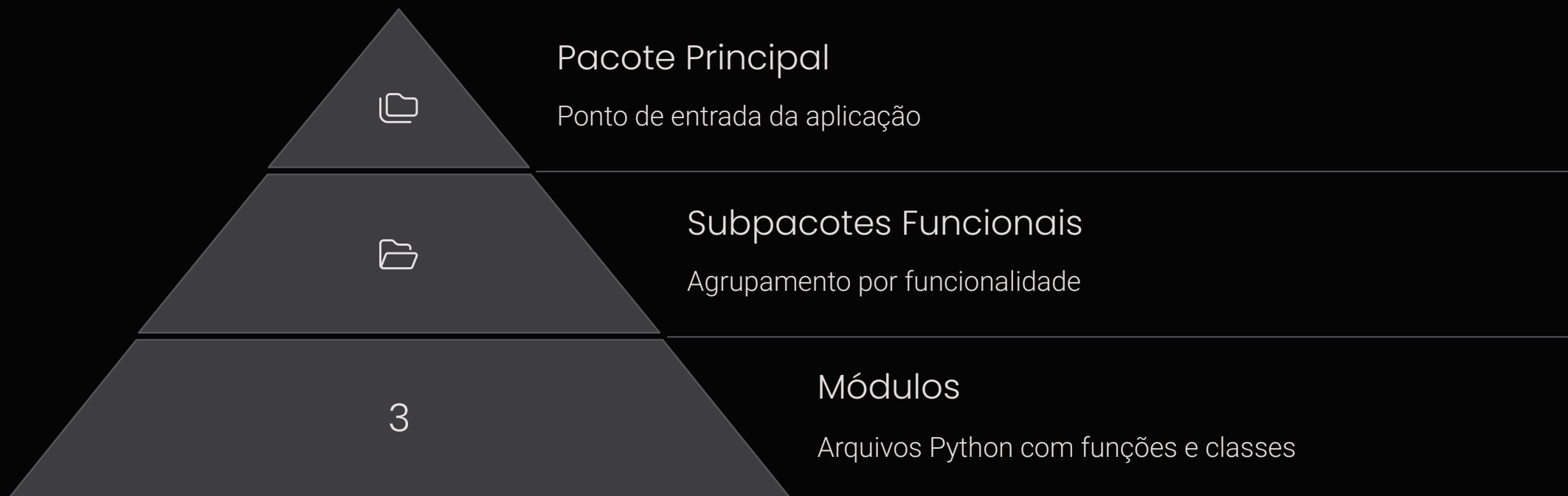
---

3

## Camada de Persistência

Armazenamento e recuperação de dados

# Organização em Pacotes e Módulos



# Exemplo Prático: App Flask Modularizado

## Estrutura de Diretórios

```
minha_app/
├── __init__.py    # Inicialização
├── config.py      # Configurações
├── models/       # Modelos de dados
│   └── __init__.py
├── controllers/   # Lógica de controle
│   └── __init__.py
├── views/        # Blueprints e rotas
│   ├── __init__.py
│   ├── auth.py    # Blueprint de autenticação
│   └── main.py     # Blueprint principal
├── templates/    # Templates HTML
└── static/       # Arquivos estáticos
```

## Inicialização com Blueprints

```
# minha_app/__init__.py
from flask import Flask
from .config import Config

def create_app(config_class=Config):
    app = Flask(__name__)
    app.config.from_object(config_class)

    # Registrar blueprints
    from .views.main import main_bp
    from .views.auth import auth_bp

    app.register_blueprint(main_bp)
    app.register_blueprint(auth_bp, url_prefix='/auth')

    return app
```

# Aplicações para Machine Learning



## Upload de Dados

Interface para receber arquivos ou valores



## Pré-processamento

Transformação e preparação dos dados



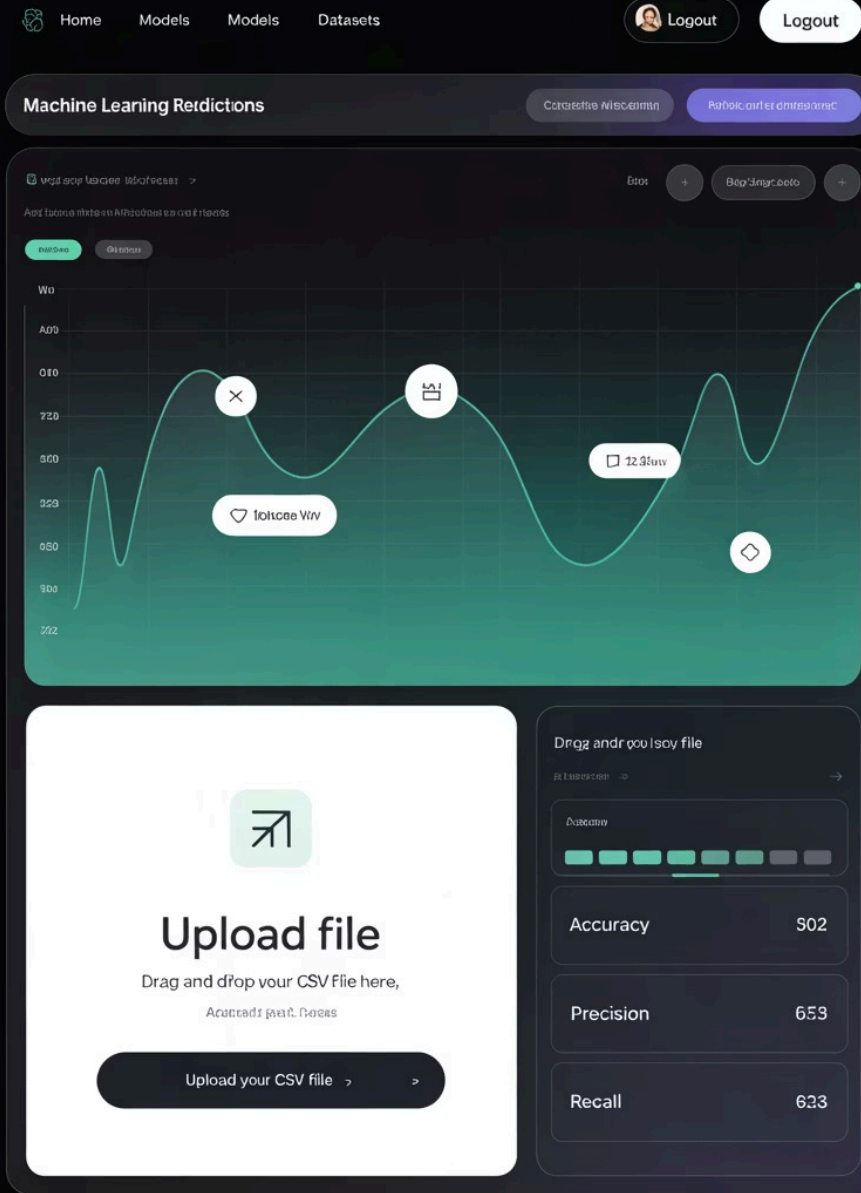
## Inferência

Aplicação do modelo ML aos dados



## Visualização

Apresentação de resultados e insights





# Demonstração: Front-end Simples com Streamlit

## Código Streamlit

```
import streamlit as st
import pandas as pd
import numpy as np
import joblib
import matplotlib.pyplot as plt

# Carregar modelo
modelo = joblib.load('modelo_regressao.joblib')

# Título da aplicação
st.title('Previsão de Preços de Imóveis')

# Input widgets
area = st.slider('Área (m²)', 30, 300, 100)
quartos = st.slider('Número de Quartos', 1, 5, 2)
banheiros = st.slider('Número de Banheiros', 1, 4, 1)
idade = st.slider('Idade do Imóvel (anos)', 0, 50, 10)

# Fazer predição
if st.button('Calcular Preço Estimado'):
    features = np.array([[area, quartos, banheiros, idade]])
    preco = modelo.predict(features)[0]

    st.success(f'Preço estimado: R$ {preco:.2f}')

# Gráfico de importância das features
fig, ax = plt.subplots()
importancias = modelo.feature_importances_
nomes = ['Área', 'Quartos', 'Banheiros', 'Idade']
ax.barh(nomes, importancias)
st.pyplot(fig)
```

## Vantagens do Streamlit

- Desenvolvimento rápido
- Widgets interativos
- Integração com visualizações
- Deploy simplificado
- Ideal para protótipos
- Focado em ciência de dados

# Integração Backend e Frontend

## Backend Flask (API)

```
@app.route('/api/predict', methods=['POST'])
def predict():
    data = request.json
    features = [
        data['area'],
        data['quartos'],
        data['banheiros'],
        data['idade']
    ]

    prediction = model.predict([features])[0]

    return jsonify({
        'prediction': prediction,
        'unit': 'BRL'
    })
```

## Frontend (JavaScript)

```
async function getPrediction() {
    const area = document.getElementById('area').value;
    const quartos = document.getElementById('quartos').value;
    const banheiros = document.getElementById('banheiros').value;
    const idade = document.getElementById('idade').value;

    const response = await fetch('/api/predict', {
        method: 'POST',
        headers: {
            'Content-Type': 'application/json'
        },
        body: JSON.stringify({
            area, quartos, banheiros, idade
        })
    });

    const result = await response.json();

    document.getElementById('result').textContent =
        `Preço estimado: R$ ${result.prediction.toFixed(2)}`;
}
```

# Exercício: Criar Módulo Web Simples que Faz Predição ML

## Treinar Modelo Simples

- Usar scikit-learn
- Dataset de exemplo (Iris, Boston)
- Salvar com joblib

## Criar API com Flask

- Endpoint /predict
- Método POST para receber dados
- Resposta em JSON

## Implementar Interface

- Formulário HTML simples
- JavaScript para requisição AJAX
- Exibição do resultado



# Sistemas de Processamento de Transações



## Atomicidade

Transação é tudo ou nada



## Consistência

Dados mantêm integridade



## Isolamento

Transações não interferem entre si



## Durabilidade

Alterações são permanentes



# Transações em Python com SQLAlchemy

## Exemplo de Transação

```
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker
from models import Cliente, Conta

# Criar conexão
engine = create_engine('postgresql://user:pass@localhost/banco')
Session = sessionmaker(bind=engine)

# Iniciar transação
session = Session()
try:
    # Obter contas
    conta_origem = session.query(Conta).filter_by(id=1).one()
    conta_destino = session.query(Conta).filter_by(id=2).one()

    # Transferir valor
    valor = 100.00
    conta_origem.saldo -= valor
    conta_destino.saldo += valor

    # Registrar transação
    transacao = Transacao(
        origem_id=conta_origem.id,
        destino_id=conta_destino.id,
        valor=valor
    )
    session.add(transacao)

    # Confirmar transação
    session.commit()
    print("Transferência concluída com sucesso!")

except Exception as e:
    # Desfazer em caso de erro
    session.rollback()
    print(f"Erro na transferência: {str(e)}")

finally:
    # Fechar sessão
    session.close()
```

## Características

- Controle explícito de transação
- Tratamento de exceções
- Rollback automático em caso de erro
- Commit apenas se tudo funcionar
- Garante as propriedades ACID

# Exemplo: Inserção Atômica de Dados

## Classe de Gerenciamento

```
class GerenciadorTransacao:
    def __init__(self, session):
        self.session = session

    def __enter__(self):
        return self

    def __exit__(self, exc_type, exc_val, exc_tb):
        if exc_type is not None:
            self.session.rollback()
            return False
        else:
            self.session.commit()
            return True

# Uso com context manager
with GerenciadorTransacao(session) as tx:
    cliente = Cliente(nome='João Silva', cpf='123.456.789-00')
    session.add(cliente)

    endereco = Endereco(
        cliente_id=cliente.id,
        rua='Av. Principal',
        cidade='São Paulo'
    )
    session.add(endereco)

# Se algo falhar, tudo é revertido automaticamente
# Se tudo funcionar, o commit é feito no final
```

## Benefícios

- Sintaxe limpa com context manager
- Isolamento de responsabilidades
- Tratamento automático de erros
- Fácil integração no código
- Garantia de atomicidade

# Filas e Mensageria para Processamento Assíncrono



## Produtor

Envia mensagens para a fila



## Fila de Mensagens

Armazena mensagens (RabbitMQ, Redis)



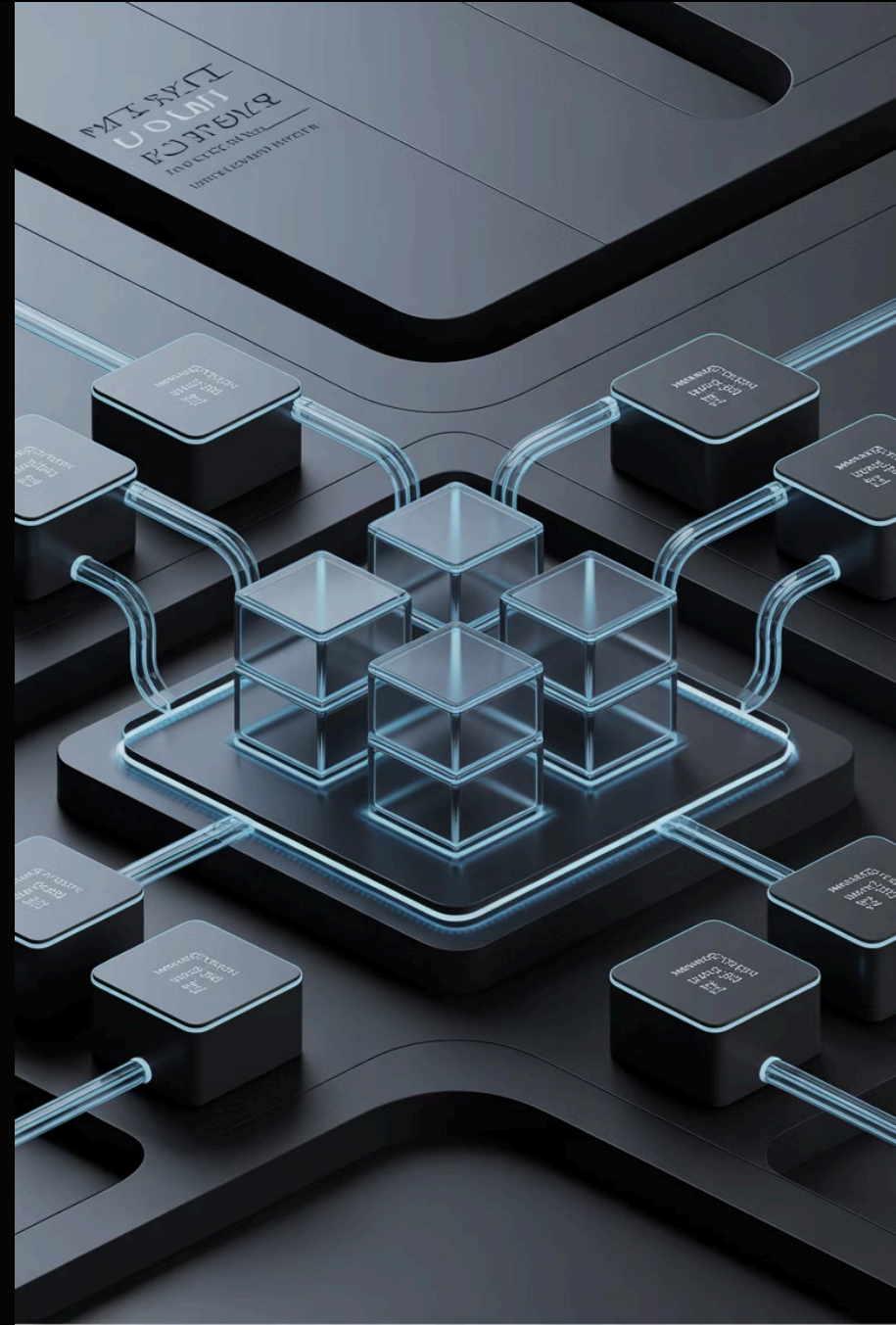
## Consumidor

Processa mensagens da fila

4

## Armazenamento

Persiste resultados do processamento





# Exemplo: Python Celery para Processamento de Lotes

## Configuração do Celery

```
# tasks.py
from celery import Celery

# Criar aplicação Celery
app = Celery(
    'tarefas',
    broker='redis://localhost:6379/0',
    backend='redis://localhost:6379/1'
)

# Definir tarefa
@app.task
def processar_dados(dataset_id):
    # Simular processamento pesado
    print(f"Processando dataset {dataset_id}")

    # Carregar dados
    dados = carregar_dataset(dataset_id)

    # Processar
    resultado = aplicar_transformacoes(dados)

    # Salvar resultado
    salvar_resultado(dataset_id, resultado)

    return {
        'status': 'concluído',
        'dataset_id': dataset_id,
        'registros_processados': len(dados)
    }
```

## Executando Tarefas

```
# main.py
from tasks import processar_dados

# Iniciar processamento assíncrono
resultado = processar_dados.delay(42)

# Verificar status (não bloqueante)
print(f"Tarefa iniciada: {resultado.id}")
print(f"Pronta? {resultado.ready()}")

# Obter resultado (bloqueante)
resultado_final = resultado.get(timeout=10)
print(f"Resultado: {resultado_final}")

# Processamento em lote
jobs = []
for i in range(1, 100):
    job = processar_dados.delay(i)
    jobs.append(job)

# Coletar todos os resultados
resultados = [job.get() for job in jobs]
```

# Aplicação em ML: Pipeline de Treinamento Lote



## Ingestão de Dados

Coleta e validação dos dados brutos



## Pré-processamento

Limpeza e transformação de features

3

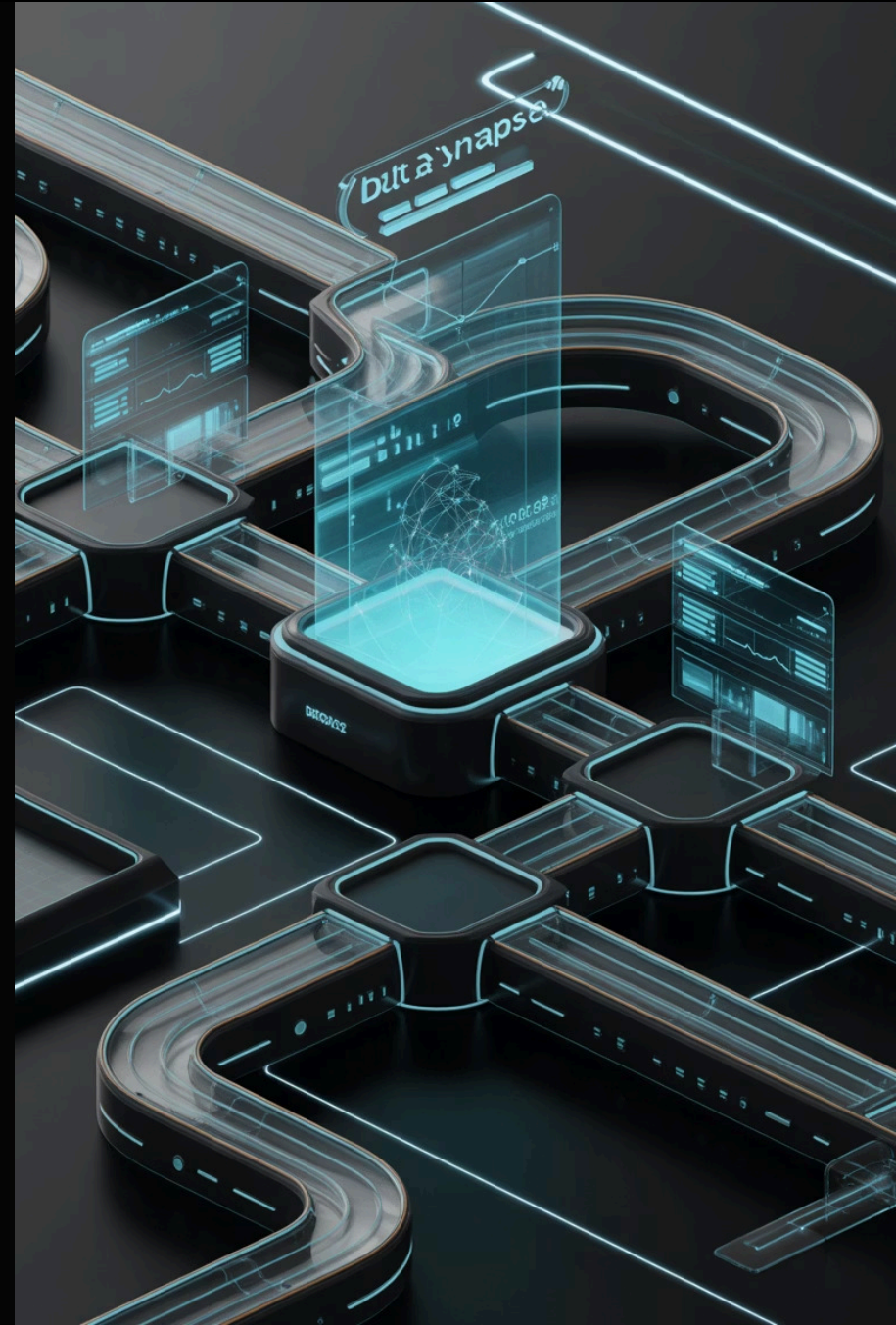
## Treinamento do Modelo

Ajuste de parâmetros e validação cruzada



## Persistência

Armazenamento do modelo treinado e métricas



# Logs e Auditoria de Transações

## Registro de Transações

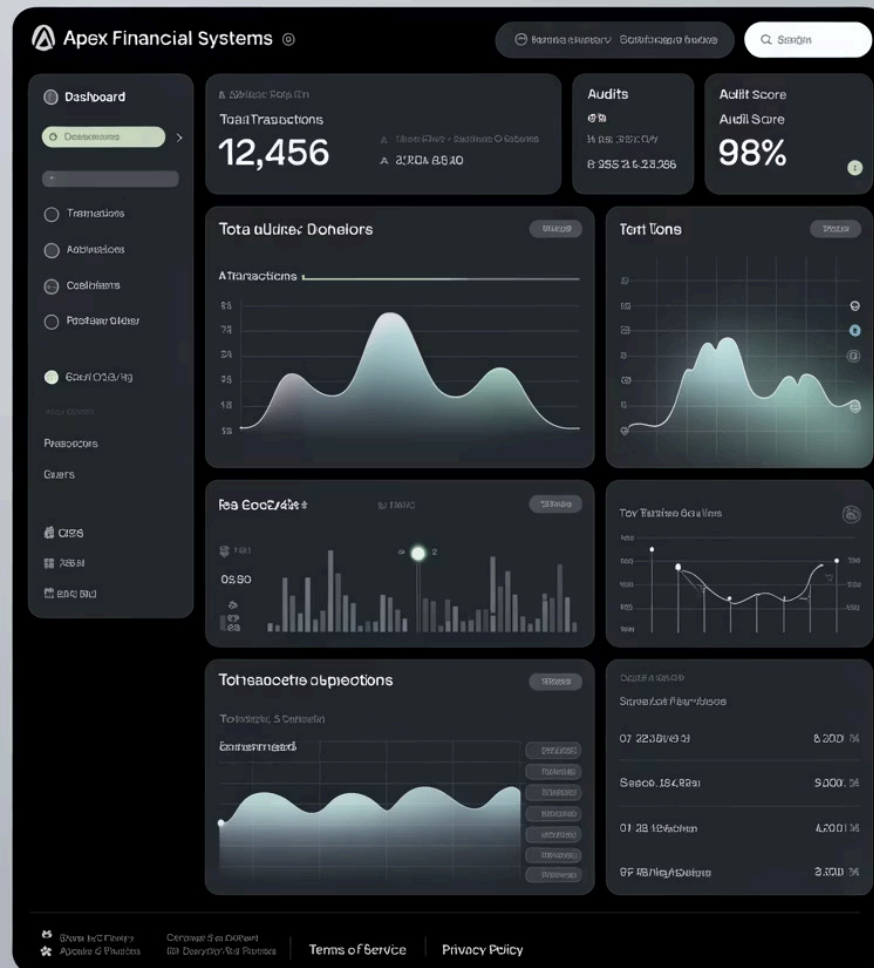
- Timestamp preciso
- Identificação do usuário
- Detalhes da operação
- Estado anterior e posterior

## Armazenamento Seguro

- Logs imutáveis
- Criptografia de dados sensíveis
- Redundância e backup
- Segregação de acesso

## Consulta e Análise

- Ferramentas de busca
- Alertas para anomalias
- Relatórios periódicos
- Conformidade regulatória



# Exercício: Simular Lançamentos Bancários Atômicos

## Desafio

1. Criar duas contas bancárias
2. Implementar transferência entre contas
3. Garantir atomicidade da operação
4. Simular cenários de erro
5. Registrar auditoria das transações

## Estrutura do Código

```
# models.py - Definição de modelos
class Conta:
    def __init__(self, id, saldo):
        self.id = id
        self.saldo = saldo

class Transacao:
    def __init__(self, origem, destino, valor):
        self.origem = origem
        self.destino = destino
        self.valor = valor
        self.timestamp = datetime.now()
        self.status = "pendente"

# transaction.py - Lógica de negócio
def transferir(origem, destino, valor):
    # Implementar com tratamento de erros
    # e garantir atomicidade
```

# Sistemas de Processamento de Linguagens



# Exemplo: Compilador Simples em Python

## Analizador Léxico

```
import re

def tokenize(code):
    # Definir padrões para tokens
    token_spec = [
        ('NUMBER', r'\d+'),
        ('PLUS', r'\+'),
        ('MINUS', r'-'),
        ('MULT', r'\*'),
        ('DIV', r'/'),
        ('LPAREN', r'\('),
        ('RPAREN', r'\)'),
        ('WHITESPACE', r'\s+')
    ]

    # Combinar todos os padrões
    token_regex = '|'.join('%s' % pair for pair in token_spec)

    # Tokenizar o código
    tokens = []
    for match in re.finditer(token_regex, code):
        token_type = match.lastgroup
        token_value = match.group()

        if token_type != 'WHITESPACE':
            tokens.append((token_type, token_value))

    return tokens
```

## Parser Simples

```
class Parser:
    def __init__(self, tokens):
        self.tokens = tokens
        self.pos = 0

    def parse(self):
        return self.expr()

    def current_token(self):
        if self.pos < len(self.tokens):
            return self.tokens[self.pos]
        return None

    def consume(self):
        token = self.current_token()
        self.pos += 1
        return token

    def expr(self):
        # Implementar parser recursivo
        # para expressões matemáticas
        term = self.term()

        while self.current_token() and self.current_token()[0] in ['PLUS', 'MINUS']:
            op = self.consume()[0]
            right = self.term()

            if op == 'PLUS':
                term += right
            else: # MINUS
                term -= right

        return term

    def term(self):
        # Implementar multiplicação e divisão
        # ...
```

# Aplicação em Machine Learning: NLP com SpaCy/NLTK

## Pré-processamento

- Tokenização
- Remoção de stopwords
- Normalização
- Stemming/Lemmatization

## Extração de Features

- Bag-of-words
- TF-IDF
- Word embeddings
- N-grams

## Modelagem

- Classificação de texto
- Reconhecimento de entidades
- Análise de sentimento
- Sumarização

# Python NLP Analysis





# Exemplo: Pré-processamento Textual com SpaCy

## Código de Processamento

```
import spacy

# Carregar modelo em português
nlp = spacy.load('pt_core_news_sm')

def processar_texto(texto):
    # Processar o texto
    doc = nlp(texto)

    # Extrair tokens, removendo stopwords e pontuação
    tokens = [token.lemma_.lower() for token in doc
               if not token.is_stop and not token.is_punct]

    # Entidades nomeadas
    entidades = [(ent.text, ent.label_) for ent in doc.ents]

    # Análise sintática
    deps = [(token.text, token.dep_, token.head.text)
             for token in doc]

    return {
        'tokens': tokens,
        'entidades': entidades,
        'deps': deps
    }

# Exemplo de uso
texto = "O presidente Lula visitou São Paulo na segunda-feira."
resultado = processar_texto(texto)
print(f'Tokens: {resultado["tokens"]}')
print(f'Entidades: {resultado["entidades"]}')

```

## Resultados

Para o texto:

"O presidente Lula visitou São Paulo na segunda-feira."

### Tokens:

- presidente
- lula
- visitar
- são
- paulo
- segunda-feira

### Entidades:

- (Lula, PER)
- (São Paulo, LOC)
- (segunda-feira, DATE)



# Exemplo: Geração de Texto com Modelos ML

## Código com Transformers

```
from transformers import pipeline, set_seed

# Carregar gerador de texto em português
gerador = pipeline('text-generation',
                    model='pierreguillou/gpt2-small-portuguese')
set_seed(42)

# Gerar texto a partir de prompt
def gerar_texto(prompt, max_length=100):
    resultados = gerador(prompt,
                          max_length=max_length,
                          num_return_sequences=1,
                          temperature=0.7)

    texto_gerado = resultados[0]['generated_text']
    return texto_gerado

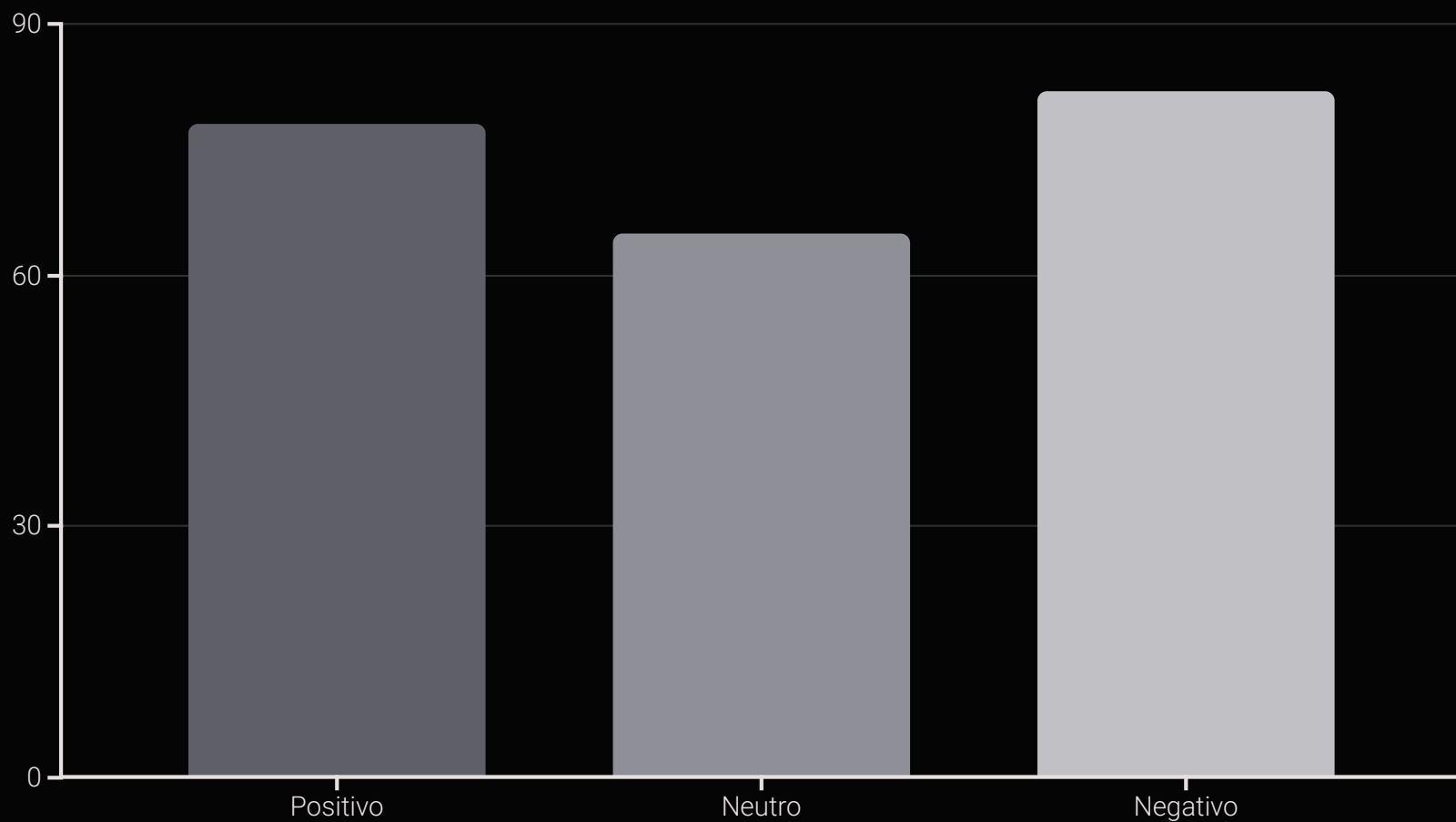
# Exemplo
prompt = "A inteligência artificial está transformando"
texto = gerar_texto(prompt)
print(texto)
```

## Aplicações Práticas

- Geração de relatórios automáticos
- Chatbots inteligentes
- Sumarização de documentos
- Tradução de idiomas
- Assistentes virtuais
- Criação de conteúdo

Os modelos transformers como BERT, GPT e T5 revolucionaram o processamento de linguagem natural nos últimos anos.

# Análise de Sentimento: Case prático com Python



# Exercício: Construir Pipeline NLP de Fim a Fim

1

## Coleta de Dados

Extrair comentários de produtos de um site



## Pré-processamento

Limpar e normalizar o texto usando SpaCy



## Vetorização

Transformar texto em features usando TF-IDF



## Modelagem

Treinar classificador para análise de sentimento



## Avaliação

Medir desempenho e melhorar o modelo



# Conclusão e Recomendações

## Principais Aprendizados

- Estruturação adequada de repositórios
- Implementação de padrões cliente-servidor
- Organização de arquitetura de aplicações
- Garantia de transações confiáveis
- Processamento avançado de linguagem natural

## Projetos Sugeridos

- API de predição com modelo ML
- Sistema distribuído com microserviços
- Processador de transações bancárias
- Aplicação de análise de sentimento
- Pipeline completa de MLOps

## Recursos Adicionais

- Livros recomendados
- Cursos online complementares
- Comunidades para dúvidas
- Ferramentas para prática
- Repositórios de exemplo