Oráculo - Relatório 3ª Entrega

Integrantes: Alicia Caporali **,**Livia Hombre, Pedro Henrique, Thiago Fabiano

1. Introdução

O projeto **Oráculo** é uma plataforma de chatbot criada para facilitar o acesso a informações sobre o andamento de tarefas em equipes de desenvolvimento. Com ele, gestores e desenvolvedores conseguem saber, por exemplo, “em que tarefa o membro X está trabalhando agora?”, apenas perguntando em linguagem natural.

O Oráculo busca tornar mais rápido e fácil o processo de acompanhar o status de projetos, integrando dados do GitHub e usando inteligência artificial para interpretar perguntas e devolver respostas claras e objetivas.

2. Arquitetura do Sistema

O sistema foi dividido em quatro partes principais, cada uma com sua função específica:

2.1 Componentes

- **Airbyte (ETL)**
  Arquivo: airbyte.py
  É responsável por buscar os dados no GitHub (como issues e pull requests) e guardar tudo no banco de dados PostgreSQL. Ele faz a parte de **Extração, Transformação e Carga (ETL)**.

- **Backend com FastAPI**
  Pasta: api
  Funciona como a ponte entre o chatbot e os dados. Recebe as perguntas, usa a IA para gerar as consultas SQL, acessa o banco de dados e envia a resposta formatada para o usuário.

- **MyVanna (IA com LLM)**
  Arquivo: MyVanna.py
  Essa parte usa um modelo de linguagem (LLM) da Google (o Gemini) para

entender a pergunta e criar o SQL certo. Depois, traduz o resultado em uma resposta fácil de entender.

- **OpenWebUI (Interface Gráfica)**
  Pasta: open_web_ui
  É a parte visual onde o usuário digita a pergunta e vê a resposta. Tudo que o usuário faz passa por aqui.

2.2 Como os dados circulam

O usuário faz uma pergunta → o sistema envia para a IA → a IA gera o SQL → o sistema consulta o banco → a resposta volta para o usuário.

3. Padrões de Projeto Utilizados

3.1 Singleton

Usado para garantir que certos componentes tenham **apenas uma instância** durante toda a execução do sistema. Isso é útil para economizar recursos e manter o funcionamento estável.

Onde foi usado:

- AskController (controle das perguntas)
- MyVanna (cliente da IA)
- airbyte (ETL)

3.2 MVC (Model-View-Controller)

A estrutura da API segue o padrão MVC:

- **Modelos** (models): definem os dados que entram e saem, como as perguntas e respostas.
- **Controladores** (AskController.py): cuidam da lógica de como os dados são processados.
- **Rotas/Views** (routes.py): definem os caminhos que o usuário pode acessar (como o /ask).

Esse padrão facilita a organização e manutenção do código.

4. Padrões de Projeto que Não Foram Usados

Alguns padrões não foram usados porque **não faziam sentido para este sistema agora**:

- **Factory**: não é necessário criar objetos diferentes dinamicamente.
- **Observer**: o sistema não precisa ficar "escutando" mudanças de estado.
- **Decorator**: o FastAPI já oferece o que precisamos com seus próprios decoradores.

5. Componentes Técnicos Detalhados

5.1 Integração com OpenWebUI

A interface do usuário conversa com a API através de um arquivo chamado pipeline_api.py. Esse arquivo:

- Recebe as perguntas
- Trata erros e mostra mensagens de status
- Usa tokens de autenticação (quando necessário)

5.2 Classe MyVanna

Ela junta duas ferramentas principais:

- **ChromaDB_VectorStore** (para guardar vetores de contexto)
- **GoogleGeminiChat** (para gerar as respostas com IA)

Funções que ela faz:

- Aprende o esquema do banco de dados
- Gera consultas SQL automaticamente
- Executa e responde as perguntas
- Permite treinar com exemplos personalizados

5.3 ETL com Airbyte

O Airbyte é responsável por:

- Buscar dados no GitHub
- Processar e organizar
- Salvar tudo no banco PostgreSQL

É possível configurar quais repositórios e tipos de dados devem ser coletados.

6. Testes Automatizados

Usamos **pytest** para garantir que tudo esteja funcionando corretamente. Temos três tipos de testes:

- **Testes do MyVanna**: checa a conexão com o banco e se a geração de SQL está funcionando.
- **Testes do pipeline**: garante que perguntas de vários tipos (até com arquivos) são tratadas corretamente.
- **Testes da API**: verifica se o endpoint /ask está respondendo como esperado.

7. Docker e Implantação

O sistema é totalmente containerizado usando Docker. No docker-compose.yml definimos:

- db: Banco de dados PostgreSQL
- back-end: Código Python com API e ETL
- front-end: Interface OpenWebUI

Existem arquivos específicos para desenvolvimento e produção:
Dockerfile.dev, Dockerfile.pub e Dockerfile.

8. Segurança e Configuração

As configurações mais sensíveis, como tokens e senhas, ficam guardadas no .env, evitando que fiquem visíveis no código.

Variáveis que usamos:

- Token do GitHub
- Senha do banco
- Chave da API Gemini

9. Conclusão

O Oráculo mostra como é possível usar IA, GitHub e uma boa arquitetura para facilitar o acesso às informações de uma equipe de desenvolvimento. O sistema está bem dividido, usa padrões de projeto onde faz sentido (Singleton, MVC), é testado, seguro e fácil de implantar com Docker.

O projeto ainda não está finalizado, os próximos passo são a melhoria nas respostas da IA para respostas mais consistentes, além de teste automatizados para a conferência de respostas