Relatório de Progresso: Catálogo Inteligente de Tintas com IA

O histórico de commits e os Pull Requests no GitHub serviram como um backlog e um registro cronológico das tarefas, onde cada feature branch (feature/initial-project-setup, feature/api-endpoints, feature/ai-integration) representou uma macro-demanda do projeto.

Como organizei demandas e atividades

A organização do projeto seguiu uma abordagem bottom-up, alinhada aos princípios da Clean Architecture, para construir uma base sólida antes de adicionar as funcionalidades mais complexas. As atividades foram sequenciadas da seguinte forma:

1. **Fundação e Estrutura do Projeto:** Definição da arquitetura, criação da estrutura de diretórios e configuração do servidor FastAPI inicial.
2. Camada de Dados e Persistência:
    - Configuração do Docker Compose para o banco de dados PostgreSQL.
    - Definição dos modelos de dados com SQLAlchemy (User, Paint).
    - Criação dos schemas de validação com Pydantic.
3. **Camada de Lógica de Negócio (CRUD):** Implementação das funções e isolando a lógica de acesso ao banco de dados.
4. **Validação da Lógica Central:** Criação de um script de teste de integração (scripts/test_crud.py) para validar as camadas de CRUD e segurança *antes* de expô-las via API, garantindo a robustez do núcleo da aplicação.
5. **Camada de Interface (API):** Desenvolvimento dos endpoints RESTful para usuários e tintas.
6. **Segurança:** Implementação da autenticação com JWT e proteção dos endpoints que requerem login.
7. **Implementação da IA:**
    - Carga de dados iniciais do CSV para o banco de dados.
    - Desenvolvimento do serviço de IA, evoluindo de uma abordagem inicial para a arquitetura final com RAG e Agente com Ferramentas.
    - Implementação do escopo extra de geração de imagem com DALL·E 3.
8. **Conteinerização e Entrega:** Finalização do Dockerfile e do docker-compose.yml para empacotar toda a aplicação e elaboração da documentação final.

O versionamento seguiu o fluxo de **Git Flow**, onde cada funcionalidade foi desenvolvida em sua própria feature branch e integrada à develop através de Pull Requests, mantendo o histórico limpo e organizado.

Como priorizei as entregas

A priorização foi baseada no princípio de construir um Produto Mínimo Viável (MVP) funcional em cada etapa, garantindo que o núcleo da aplicação estivesse sempre estável.

- **Prioridade 1 (Fundação Crítica):** Ter uma API RESTful segura e funcional. Era impossível construir o assistente de IA sem uma base de dados e endpoints para interagir. Portanto, o CRUD de usuários, a autenticação JWT e o CRUD de tintas foram as primeiras prioridades.
- **Prioridade 2 (Funcionalidade Principal):** Implementar o assistente de IA com a capacidade de recomendar tintas. O foco inicial foi garantir que o mecanismo RAG funcionasse de forma rápida e precisa, entregando o valor central do desafio.
- **Prioridade 3 (Diferencial - Escopo Extra):** Com o núcleo da IA funcionando, a prioridade passou a ser a implementação da geração de imagens com DALL·E 3, um requisito opcional que agregará grande valor.
- **Prioridade 4 (Empacotamento):** A finalização da dockerização da API e a criação da documentação (README.md) foram as últimas tarefas, focadas em garantir que o projeto pudesse ser facilmente executado e avaliado por terceiros.

Principais Dificuldades

1. **Lentidão e Imprecisão da Abordagem Inicial com Agente SQL.**
    - **Problema:** A primeira implementação de IA usava um Agente SQL do LangChain. Embora funcional, o agente era extremamente lento e propenso a gerar consultas SQL incorretas.
    - **Solução:** Foi tomada a decisão estratégica de pivotar a arquitetura. A solução foi refatorada para uma abordagem RAG com embeddings (FAISS), que se mostrou drasticamente mais rápida e precisa. A busca semântica é feita localmente de forma instantânea, e o LLM é usado apenas para gerar a resposta final a partir de um contexto já filtrado, eliminando a chance de erros de SQL.
2. **Depuração da Geração de Imagem e Extração de URL.**
    - **Problema:** A chamada para o DALL·E falhava com um erro de atributo ('OpenAI' object has no attribute 'images'), e, após a correção, a URL da imagem não era extraída corretamente do texto de resposta do agente.
    - **Solução:** O primeiro erro foi resolvido corrigindo o import para usar o cliente correto da biblioteca oficial da OpenAI. O segundo foi resolvido substituindo uma lógica frágil de split() por uma expressão regular (Regex) robusta, capaz de extrair a URL de forma confiável, independentemente da formatação do texto.

O que faria diferente com mais tempo ou em um contexto real de projeto

1. **Implementação de CI/CD:** Seria configurado um pipeline de Integração e Entrega Contínua (CI/CD) com **GitHub Actions**. Isso automatiza a execução dos testes a cada push, garantiria a qualidade do código e poderia automatizar o deploy para um ambiente de homologação ou produção.
2. **Base de Dados Vetorial Dedicada:** A implementação atual usa FAISS, que é um banco de dados vetorial em memória. Para um projeto em escala, com um catálogo de produtos muito maior ou com a necessidade de persistência, migraria para uma solução dedicada como **Pinecone**, **Weaviate** ou a extensão **pgvector** para o próprio PostgreSQL.
3. **Observabilidade e Logging:** Adicionaria um sistema de logging estruturado (ex: com a biblioteca structlog) e monitoramento (com ferramentas como Prometheus e Grafana) para rastrear o desempenho da API, os custos com a OpenAI e o comportamento do agente de IA em produção.