Relatório Final - Prática em Engenharia de Software - Time Bravo

Rômulo Barizon Pitt, Thiago Borges de Souza, Allan Groisman Kusbick, Lucca Iãnez, Erik Monteiro Ribeiro, Vinícius Pereira Dias, Anderson Renan Paniz Sprenger, Rafael Schaker Kopczynski da Rosa, Juan Victor Gomes Acosta, Gabriel Zurawski de Souza

[1]Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS)
Av. Ipiranga, 6681 - Partenon, Porto Alegre - RS, 90619-900

1. Introdução

Este documento apresenta o relatório final do projeto "Me Leva App", desenvolvido pela equipe de 10 integrantes denominada Time Bravo, como parte da disciplina de Prática em Engenharia de Software. O projeto foi concebido para autorizar o início dos trabalhos, documentar os requisitos e atribuir as responsabilidades-chave.

1.1. Contextualização e Justificativa

O projeto justifica-se pelo crescente número de animais de estimação que necessitam de um lar. Essa problemática foi intensificada no Rio Grande do Sul, onde as enchentes ocorridas em 2024 resultaram em um aumento expressivo de pets que precisam de abrigo. Diante deste cenário, torna-se imprescindível a criação de uma ferramenta que facilite a conexão entre animais disponíveis e possíveis adotantes. Considerando a massiva popularização dos smartphones na sociedade moderna, optou-se pela criação de um aplicativo mobile para atender a essa demanda.

1.2. Objetivos do Projeto

O objetivo específico do projeto é a criação de um aplicativo móvel voltado para a adoção de animais, no qual usuários podem tanto disponibilizar pets para adoção quanto encontrar um para adotar. De forma realista, a meta é implementar um Produto Mínimo Viável (MVP) do aplicativo, além de produzir toda a documentação associada ao projeto. O prazo estipulado para a conclusão foi de até o final de junho, totalizando um período de quatro meses de desenvolvimento.

1.3. Escopo do Projeto

O principal produto a ser entregue é um sistema de cadastramento de pets para adoção, materializado em um aplicativo móvel. As funcionalidades centrais do sistema incluem:

- Cadastro de Pets, com suas características e histórico de saúde.
- Cadastro de Protetores e ONGs com informações de contato.
- Cadastro de Adotantes com seus dados e preferências.
- Histórico de Rastreabilidade, registrando as movimentações dos animais.
- Funcionalidade "Quero Adotar"para pesquisa e filtragem de pets.
- Relatório de Compatibilidade para indicar a combinação entre adotante e pet.
- Acompanhamento Pós-Adoção com registro de visitas e incidentes.

Além do software, outros resultados foram gerados, como a entrega da documentação técnica no Termo de Abertura do Projeto, o código-fonte disponibilizado em um repositório público, esse relatório final e uma apresentação final para demonstração do sistema.

2. Requisitos e Casos de Uso

Durante a fase de planejamento do projeto, foi realizado um levantamento detalhado dos requisitos funcionais e não funcionais que nortearam o desenvolvimento do sistema "Me Leva App". Adicionalmente, foram definidos os casos de uso para abranger as interações dos diferentes perfis de usuários com a plataforma.

2.1. Requisitos Funcionais

Os seguintes requisitos funcionais foram definidos para o sistema:

1. **Cadastro de Pets:** Permitir a inserção e atualização das informações de cada animal, como nome, espécie, idade, porte, sexo e histórico de saúde.
2. **Histórico de Rastreabilidade:** Registrar e permitir a consulta de todas as movimentações do pet, incluindo datas e localizações.
3. **Cadastro de Protetores/ONGs:** Permitir o cadastro de informações de pessoas físicas ou jurídicas, incluindo dados de contato, com a possibilidade de edição e exclusão.
4. **Cadastro de Adotantes:** Permitir o cadastro de dados do adotante, suas preferências para um novo pet e a possibilidade de exclusão da conta.
5. **Quero Adotar:** Viabilizar a busca filtrada de pets disponíveis e permitir que o usuário manifeste interesse na adoção.
6. **Acompanhamento Pós-Adoção:** Possibilitar o registro de visitas e incidentes de saúde após a adoção, com geração de relatórios periódicos.
7. **Relatório de Compatibilidade:** Calcular um índice de compatibilidade entre o pet e o adotante com base em fatores como espécie, porte e saúde.
8. **Gerenciamento de Usuários:** Implementar funcionalidades de login, logout, recuperação de senha e permissões de acesso.

2.2. Requisitos Não Funcionais

Os requisitos não funcionais que guiaram a arquitetura e a implementação foram:

1. **Usabilidade:** A interface do aplicativo móvel deveria ser simples e intuitiva.
2. **Performance:** O tempo de resposta para consultas e listagens deveria ser razoável.
3. **Disponibilidade:** O sistema deveria ter uma disponibilidade mínima de 99
4. **Segurança:** Exigiu-se a implementação de controle de acesso e criptografia de informações sensíveis.
5. **Escalabilidade:** A arquitetura deveria possibilitar a expansão do serviço para outras regiões.
6. **Tecnologia:** O backend deveria ser desenvolvido em Java com Spring Boot e utilizar um banco de dados relacional.

2.3. Casos de Uso

Foram definidos 9 grupos de casos de uso para o sistema:

- **1. Cadastro, edição e exclusão de protetor:**
  - UC 1.1: Um protetor cadastra-se com suas informações pessoais (nome, CPF, e-mail, etc.).
  - UC 1.2: Uma ONG cadastra-se com suas informações empresariais (nome, CNPJ, etc.).

  - UC 1.3 e 1.4: Um protetor ou ONG atualiza seus dados cadastrais.
  - UC 1.5 e 1.6: Um protetor ou ONG solicita a exclusão de sua conta, o que remove também os pets sob sua responsabilidade.
- **2. Cadastro, edição e exclusão de adotantes:**
  - UC 2.1 e 2.2: Um adotante cadastra-se e atualiza seus dados pessoais.
  - UC 2.3: Um adotante solicita a exclusão de sua conta.
  - UC 2.4: Um adotante registra os pets que já possui.
  - UC 2.5: Um adotante em potencial informa as características desejadas de um novo pet para filtrar buscas.
- **3. Cadastro, edição e exclusão de pets:**
  - UC 3.1: Um protetor cadastra um pet com suas características básicas, de saúde e necessidades especiais.
  - UC 3.2: Um protetor altera o status de um pet (disponível, adotado, etc.).
  - UC 3.3 e 3.4: Um protetor ou adotante visualiza o perfil de um pet ou a lista de pets registrados por ele.
- **4. Histórico de rastreabilidade:**
  - UC 4.1: Um protetor ou adotante consulta o histórico do pet (datas de resgate, localizações).
  - UC 4.2 e 4.3: Um protetor cria e edita o histórico de um pet.
- **5. Login/logoff e recuperação de senha:**
  - UC 5.1 e 5.4: Um protetor/ONG ou adotante realiza login no sistema.
  - UC 5.2 e 5.5: Um protetor/ONG ou adotante realiza logoff.
  - UC 5.3 e 5.6: Um protetor/ONG ou adotante recupera sua senha.
- **6. Processo de busca e adoção de animais:**
  - UC 6.1 e 6.2: Um adotante pesquisa por pets, filtrando por características, e recebe sugestões compatíveis.
  - UC 6.3: Um adotante manifesta interesse na adoção através de um botão ”Quero adotar”.
  - UC 6.4: Um protetor recebe e gerencia as manifestações de interesse.
  - UC 6.5: Um usuário compartilha o perfil de um pet.
- **7. Acompanhamento pós-adoção:**
  - UC 7.1 e 7.2: Um protetor registra visitas e incidentes de saúde do pet após a adoção.
  - UC 7.3: Um protetor gera um relatório periódico do estado do pet.
  - UC 7.4: O sistema envia notificações e lembretes automáticos sobre o acompanhamento.
- **8. Manutenção do sistema:**
  - UC 8.1 a 8.8: Um administrador do sistema gerencia todos os dados de pets e usuários, podendo criar, alterar e remover contas ou registros para fins de suporte.
- **9. Acesso como visitante:**
  - UC 9.1: Um visitante visualiza os pets disponíveis para adoção.
  - UC 9.2: Um visitante inicia o processo de adoção, o que o leva para a criação de uma conta de usuário.
  - UC 9.3: Um visitante compartilha o perfil de um pet.

3. Arquitetura da Solução

Para atender aos requisitos do projeto, foi projetada e implementada uma arquitetura de sistema distribuída, composta por três componentes principais: um frontend mobile, uma API de negócios e um serviço de Inteligência Artificial. A decisão por essa abordagem visou a separação de responsabilidades, permitindo que cada parte do sistema fosse desenvolvida e mantida de forma independente, utilizando as tecnologias mais adequadas para cada finalidade.

3.1. Visão Geral da Arquitetura

O fluxo de interação entre os componentes foi desenhado para otimizar a performance e a especialização. A arquitetura geral pode ser descrita da seguinte forma:

- **Frontend (React Native):** A aplicação móvel com a qual o usuário interage. Responsável por toda a renderização da interface, captura de dados e comunicação com os serviços de backend.
- **API de Negócios (Java Spring Boot):** O serviço central do sistema, responsável por gerenciar as regras de negócio principais e persistir os dados. Ele expõe endpoints RESTful para todas as operações de CRUD (criação, leitura, atualização e exclusão) das entidades do sistema.
- **Serviço de IA (Python FastAPI):** Um microsserviço especializado, desacoplado da API de negócios, cuja única responsabilidade é processar a lógica de compatibilidade (*match*) entre adotantes e pets, utilizando a API da OpenAI.

Essa divisão permitiu que a lógica de negócios, mais complexa e transacional, fosse tratada por um framework robusto como o Spring Boot, enquanto a tarefa de IA, que é computacionalmente intensiva e dependente de uma API externa, fosse isolada em um serviço leve e assíncrono com FastAPI.

3.2. Frontend (React Native)

O aplicativo mobile foi desenvolvido utilizando a plataforma **Expo** sobre o framework **React Native**, o que permitiu um desenvolvimento ágil e multiplataforma. Embora não tenha seguido um padrão de arquitetura formal como MVC ou MVVM, a estruturação do código foi realizada de forma pragmática e orientada a componentes, um princípio fundamental do React. Essa abordagem resultou em uma interface de usuário altamente responsiva e modular.

A organização dos componentes visuais, a lógica de estado e a comunicação com as APIs foram separadas em camadas distintas, garantindo uma base de código organizada e de fácil manutenção. Para a navegação entre telas, foram utilizadas as bibliotecas do **React Navigation** (‘@react-navigation/native‘, ‘@react-navigation/bottom-tabs‘). A comunicação com os backends foi gerenciada pela biblioteca **Axios**, e a interface foi enriquecida com pacotes como `react-native-reanimated` para animações fluidas e `expo-linear-gradient` para efeitos visuais.

3.3. API de Negócios (Java Spring Boot)

O núcleo do backend foi construído em Java, utilizando o framework **Spring Boot**. A arquitetura deste serviço foi fortemente baseada nos princípios do **Domain-Driven Design (DDD)**, onde o código foi organizado em camadas centradas no domínio do negócio. As

entidades principais como `Usuario`, `Pet` e `PerfilAdocao` representam o coração da aplicação, e os serviços e repositórios foram construídos ao redor delas.

Este serviço expõe uma série de endpoints RESTful que permitem ao frontend realizar todas as operações de CRUD necessárias para o funcionamento da aplicação, como cadastrar um novo usuário, adicionar um pet para adoção, editar informações de um perfil e consultar dados. Toda a persistência dos dados é gerenciada por esta API, que se comunica com um banco de dados relacional (H2 em ambiente de desenvolvimento).

3.4. Serviço de IA (Python FastAPI com OpenAI)

Para a funcionalidade de cálculo de compatibilidade, foi implementado um serviço independente utilizando Python e o framework **FastAPI**. Este serviço expõe uma API assíncrona para evitar o bloqueio de requisições enquanto processa a análise de compatibilidade.

A arquitetura deste serviço foi projetada para ser simples e eficiente:

- **Endpoints Assíncronos:** Foram criados três endpoints principais:
  - `POST /avaliar/{user_id}`: Recebe o perfil de preferências do adotante e uma lista de pets. Em vez de processar a requisição imediatamente, ele inicia uma tarefa em segundo plano (*background task*) e retorna uma resposta instantânea ao cliente, informando que a avaliação foi iniciada.
  - `GET /avaliar/status/{user_id}`: Permite que o frontend consulte o status da tarefa (ex: "processing", "completed", "error").
  - `DELETE /avaliar/resultado/{user_id}`: Permite a limpeza do resultado da memória após o consumo pelo cliente.
- **Processamento com OpenAI:** A tarefa em segundo plano constrói um prompt detalhado e o envia para o modelo `gpt-4-turbo` da OpenAI. O prompt instrui a IA a retornar um score de compatibilidade de 0 a 100 para cada pet, com base nas preferências do perfil do adotante, garantindo que os scores pareçam orgânicos e considerando múltiplas variáveis.
- **Armazenamento em Memória:** Para fins de demonstração, os resultados das tarefas são armazenados em uma estrutura de dados em memória. Para um ambiente de produção, esta seria deverá ser substituída por uma solução mais robusta.

3.5. Fluxo de Dados do Match de Compatibilidade

A interação entre os componentes para a funcionalidade de match ocorre da seguinte forma:

1. O usuário, na tela "Explorar"do aplicativo, solicita a busca por pets compatíveis.
2. O frontend (React Native) primeiro requisita à **API de Negócios (Java)** os dados do perfil de preferências do usuário e a lista de pets a serem avaliados.
3. Com esses dados em mãos, o frontend monta o payload e envia uma requisição POST para o endpoint `/avaliar/{user_id}` do **Serviço de IA (Python)**.
4. O Serviço de IA inicia o processamento em background e o frontend começa a consultar o endpoint de status periodicamente.
5. Quando o status muda para "completed", o frontend busca o resultado (uma lista de pets com seus respectivos scores de compatibilidade), exibe os resultados ordenados na interface para o usuário e, por fim, solicita a limpeza do dado na API de IA.

Este fluxo garante que a experiência do usuário seja fluida, sem travamentos, mesmo que a análise da OpenAI leve alguns segundos para ser concluída.

4. Tecnologias Utilizadas

A seleção das tecnologias para o projeto "Me Leva App" foi pautada pela adequação de cada ferramenta às suas responsabilidades dentro da arquitetura, visando robustez, agilidade no desenvolvimento e escalabilidade. A seguir, detalha-se o conjunto de tecnologias empregado em cada componente do sistema.

4.1. Frontend (Mobile)

O aplicativo mobile foi construído sobre o ecossistema React Native, utilizando o Expo para gerenciar o projeto e suas dependências. As principais tecnologias foram:

- **Framework/Plataforma:** React Native e Expo.
- **Linguagem:** TypeScript.
- **Navegação:** A estrutura de navegação entre telas foi implementada com o `expo-router`, em conjunto com o `@react-navigation/native` e `@react-navigation/bottom-tabs` para a criação de abas.
- **Comunicação com API:** A biblioteca **Axios** foi utilizada para realizar as requisições HTTP para os serviços de backend.
- **Componentes de UI e Estilização:** Foram utilizados `@expo/vector-icons` para iconografia, `expo-linear-gradient` para efeitos visuais, e `react-native-reanimated` para a criação de animações.
- **Testes:** O framework **Jest**, com o preset `jest-expo`, foi a ferramenta de escolha para a execução dos testes automatizados.

4.2. Backend (API de Negócios)

Este serviço, responsável pelas regras de negócio centrais, foi desenvolvido com as seguintes tecnologias, conforme definido na fase de planejamento:

- **Linguagem:** Java.
- **Framework:** Spring Boot.
- **Banco de Dados:** Foi utilizado um banco de dados relacional (H2 para desenvolvimento), conforme previsto nos requisitos não funcionais.
- **Gerenciador de Dependências:** O projeto foi configurado com Maven, ferramentas padrão para o ecossistema Spring.

4.3. Backend (Serviço de IA)

O microsserviço especializado no cálculo de compatibilidade foi construído com tecnologias do ecossistema Python:

- **Linguagem:** Python.
- **Framework:** FastAPI, escolhido por sua alta performance e capacidade de criar APIs assíncronas.
- **Inteligência Artificial:** A integração com a IA foi realizada através da biblioteca oficial `openai`, consumindo o modelo **GPT-4 Turbo** para processar as requisições de match.
- **Gerenciamento de Configuração:** A biblioteca `python-dotenv` foi usada para carregar variáveis de ambiente, como a chave da API da OpenAI, de forma segura.

5. Implementação do Frontend

O frontend do aplicativo "Me Leva App" foi desenvolvido em **React Native** com o auxílio do ecossistema **Expo**, o que garantiu uma base de código única para as plataformas Android e iOS. A implementação foi focada na criação de uma experiência de usuário fluida e reativa, com uma clara separação entre a lógica da interface, o gerenciamento de estado e a comunicação com os serviços de backend.

5.1. Estrutura e Componentes Principais

A base do desenvolvimento frontend seguiu os padrões modernos do React, com o uso extensivo de componentes funcionais e Hooks.

- **Gerenciamento de Estado:** O estado local de cada tela, como os dados de formulários e o controle de visibilidade de modais, foi gerenciado com o hook `useState`. Para dados persistentes da sessão do usuário, como o ID e o token de autenticação, foi utilizado o `AsyncStorage`, permitindo que o usuário permaneça logado entre os usos do aplicativo.
- **Navegação:** A navegação entre as diferentes telas do aplicativo foi implementada utilizando o **Expo Router**. Funções como `useRouter` para navegar programaticamente e `useLocalSearchParams` para obter parâmetros de rota (como o ID de um pet) foram essenciais para a interconexão das telas. O hook `useFocusEffect` foi empregado para recarregar dados sempre que uma tela ganhava foco, garantindo que as informações estivessem sempre atualizadas.
- **Comunicação com API:** Toda a comunicação com os backends foi centralizada em um módulo dedicado (`api.ts`). Este módulo exporta funções assíncronas para cada endpoint, como `fazerLogin`, `listarPets` e `solicitarAdocaoPet`, utilizando a biblioteca **Axios** para gerenciar as requisições HTTP. Essa abordagem abstrai a lógica de rede dos componentes de tela e facilita a manutenção.
- **Estilização Consistente:** Foi estabelecido um arquivo de tema centralizado (`theme.js`) que define constantes para cores, fontes, tamanhos e sombras (`COLORS`, `FONTS`, `SIZES`, `SHADOWS`), garantindo uma identidade visual coesa em todo o aplicativo.

5.2. Telas e Funcionalidades Notáveis

Diversas telas foram implementadas para compor a experiência do usuário. Dentre elas, destacam-se:

- **Tela de Exploração (`ExplorarScreen.tsx`):** Funciona como a tela principal de descoberta, exibindo os pets disponíveis em uma grade. Implementa funcionalidades de busca textual e filtros. É nesta tela que o usuário pode acionar o *match* de compatibilidade, que faz a chamada para `iniciarAvaliacaoMatch` e inicia um processo de *polling* com `verificarStatusMatch` para obter os scores de compatibilidade da API de IA.
- **Tela de Perfil do Pet (`PerfilPetScreen.tsx`):** Uma tela detalhada que exibe todas as informações de um animal específico. A sua interface é dividida em abas ("Resumo", "Sobre Mim", "Saúde", "Histórico") para organizar os dados. Os botões de ação nesta tela são contextuais: para um adotante, exibe "Quero Adotar"e "Contatar Protetor"; para o dono do pet, exibe "Editar"e "Excluir".

- **Telas de Gerenciamento de Solicitações:** Foram criadas telas específicas para que protetores e adotantes pudessem gerenciar os processos de adoção. A `SolicitacoesProtetorScreen.tsx` agrupa as solicitações recebidas por pet, enquanto a `DetalhesSolicitacaoAdotanteScreen.tsx` permite que o adotante acompanhe o status de suas solicitações.
- **Tela de Perfil de Match (`PerfilMatchScreen.tsx`):** Um formulário onde o adotante define suas preferências (espécie, porte, sexo, etc.). A tela utiliza o hook `useFocusEffect` para buscar e preencher os dados de um perfil já existente ao ser aberta, proporcionando uma experiência contínua para o usuário.

5.3. Fluxo de Uso Principal

A estrutura do aplicativo foi projetada para suportar dois fluxos principais de usuário, que demonstram as funcionalidades centrais do sistema.

5.3.1. Fluxo 1: Processo de Adoção

Este fluxo demonstra a interação entre os dois tipos de perfil (Protetor e Adotante) para completar uma adoção.

1. Um usuário **Protetor** realiza o login e navega para a tela ”Meus Pets”, onde cadastra um novo animal para adoção.
2. O Protetor faz logoff. Um segundo usuário, do tipo **Adotante**, realiza o login.
3. O Adotante encontra o pet recém-cadastrado na tela ”Explorar”e navega para o seu perfil detalhado (`PerfilPetScreen.tsx`).
4. Na tela do pet, o Adotante clica em ”Quero Adotar!”, disparando a função `solicitarAdocaoPet`.
5. O Adotante faz logoff. O usuário Protetor realiza o login novamente.
6. O Protetor acessa a tela de ”Solicitações Recebidas”e, em seguida, os detalhes das solicitações para aquele pet específico.
7. O Protetor avalia a solicitação do Adotante e a aprova, utilizando a função `atualizarSituacaoSolicitacao`.
8. Para confirmação, o Adotante pode fazer login novamente e verificar em ”Minhas Solicitações Enviadas”que seu pedido foi aceito.

5.3.2. Fluxo 2: Match por Compatibilidade

Este fluxo demonstra a integração com o serviço de IA para gerar recomendações personalizadas.

1. Um usuário (Adotante ou Protetor) navega para a tela ”Perfil de Match”(`PerfilMatchScreen.tsx`).
2. O usuário preenche e salva o formulário com suas preferências de adoção.
3. Na tela ”Explorar”(`ExplorarScreen.tsx`), o usuário aciona o botão de ”match”(ícone de chama).
4. Uma requisição é enviada ao serviço de IA, que processa a compatibilidade em segundo plano. O frontend exibe um modal de carregamento.
5. Após a conclusão, a tela ”Explorar”é atualizada e os cards dos pets passam a exibir um ”score”de compatibilidade, ordenando os mais compatíveis primeiro.

6. Implementação do Backend (API de Negócios)

O backend principal do sistema, responsável por toda a lógica de negócio e persistência de dados, foi desenvolvido utilizando o framework **Spring Boot** com a linguagem **Java**. A implementação seguiu uma arquitetura em camadas bem definida, promovendo a separação de responsabilidades e a manutenibilidade do código.

6.1. Modelo de Dados e Persistência

A base da aplicação é o seu modelo de dados, que foi definido através do schema SQL e mapeado para objetos Java utilizando **Spring Data JPA**.

- **Entidades Principais:** Foram modeladas classes como `Usuario` e `Pet`, que representam as entidades centrais do domínio. Anotações como `@Entity` e `@Table` foram usadas para ligar essas classes às suas respectivas tabelas no banco de dados.
- **Estrutura do Banco:** O banco de dados foi estruturado para suportar todas as funcionalidades necessárias, com tabelas como `usuario`, `pet`, `solicitacao_adocao` para gerenciar os pedidos, `perfil_match` para as preferências do adotante, e `pet_favoritos` para a lista de favoritos de cada usuário.
- **Integridade de Dados:** Para garantir a consistência, foram utilizadas enumerações Java para campos com valores restritos, como `PerfilUsuario` ('ADOTANTE', 'PROTETOR', 'AMBOS') e `TipoUsuario` ('PESSOA', 'ONG') na entidade `Usuario`. Além disso, o próprio schema define restrições (`CHECK`) e chaves estrangeiras (`FOREIGN KEY`) para manter a integridade referencial.

6.2. Arquitetura em Camadas

O código foi organizado seguindo o padrão de três camadas (Controller, Service, Repository), uma prática comum em aplicações Spring Boot que facilita a organização e o teste.

- **Camada de Controle (Controller):** Responsável por expor os endpoints da API REST. Classes como `UsuarioController` e `PetController` utilizam anotações como `@RestController`, `@GetMapping` e `@PostMapping` para definir as rotas e os métodos HTTP que o frontend pode consumir.
- **Camada de Serviço (Service):** Contém a lógica de negócio principal. Classes como `UsuarioService` e `PetService` orquestram as operações, validam as regras de negócio (ex: um e-mail não pode ser duplicado) e gerenciam as transações com o banco de dados através da anotação `@Transactional`.
- **Camada de Repositório (Repository):** Interface responsável pela abstração do acesso aos dados. Interfaces como `UsuarioRepository` e `PetRepository` estendem a `JpaRepository`, fornecendo métodos de CRUD (Create, Read, Update, Delete) prontos para uso. Para consultas mais complexas, foram implementados métodos com queries nativas usando a anotação `@Query`, como a busca por pets disponíveis para adoção.

6.3. Segurança

A segurança da API foi um ponto central na implementação. Para o controle de autenticação e autorização, foi utilizado o **Spring Security**.

- **Autenticação:** O processo de login valida as credenciais do usuário comparando a senha fornecida com o hash armazenado no banco de dados, utilizando o `PasswordEncoder` do Spring.
- **Autorização:** O acesso aos endpoints é protegido, e a API verifica a identidade do usuário logado para garantir que ele só possa acessar ou modificar os próprios dados, como visto na validação de ID de usuário no `PetController`.

7. Implementação do Serviço de IA

Para a funcionalidade de compatibilidade entre adotantes e pets, foi desenvolvido um microsserviço independente em **Python**, utilizando o framework **FastAPI**. A escolha dessa tecnologia foi motivada por sua alta performance em operações de I/O e sua simplicidade para criar APIs assíncronas, características ideais para uma tarefa que depende de uma chamada externa à API da OpenAI.

7.1. Estrutura da API e Modelos de Dados

A API foi estruturada para operar de forma assíncrona, garantindo que o aplicativo frontend não ficasse bloqueado enquanto aguarda o processamento da IA.

- **Modelagem de Dados com Pydantic:** Foram definidos modelos de dados robustos usando Pydantic, como `Perfil`, `Pet` e `Requisicao`, para validar automaticamente os dados recebidos nas requisições, garantindo a integridade dos dados enviados para a OpenAI.
- **Endpoints Assíncronos:** A API expõe um fluxo de três endpoints principais para gerenciar o processo de match:
    1. `POST /avaliar/{user_id}`**:** Recebe o perfil do adotante e a lista de pets. Em vez de processar a requisição em tempo real, ele agenda a tarefa para ser executada em segundo plano (*background*) e retorna uma resposta imediata ao cliente.
    2. `GET /avaliar/status/{user_id}`**:** Permite que o frontend consulte o status da tarefa (ex: "processing", "completed", "error"), implementando um padrão de *polling*.
    3. `DELETE /avaliar/resultado/{user_id}`**:** Endpoint de limpeza, utilizado para remover os dados do resultado da memória do servidor após o cliente tê-lo consumido com sucesso.

7.2. Integração com a API da OpenAI

O núcleo do serviço é a função `processar_match_em_background`, que orquestra a interação com a OpenAI.

- **Construção do Prompt:** Um prompt detalhado é construído dinamicamente, contendo as especificações da tarefa, o perfil do adotante e a lista completa de pets, ambos formatados como JSON. O prompt instrui o modelo a retornar um score de compatibilidade entre 0 e 100, com diretrizes para que os resultados pareçam orgânicos e não repetitivos.

- **Chamada à API:** O serviço utiliza a biblioteca oficial `openai` para enviar a requisição ao modelo `gpt-4-turbo`. Foi utilizado o modo de resposta JSON (`response_format={"type": "json_object"}`) para garantir que a saída seja sempre um JSON válido, facilitando o parsing do resultado.
- **Gerenciamento de Estado:** Para fins de demonstração, o estado e o resultado das tarefas de match foram armazenados em um dicionário Python em memória. O próprio código-fonte inclui um comentário indicando que, em um ambiente de produção, esta camada de armazenamento seria substituída por uma solução mais escalável e persistente, como um banco de dados Redis.

8. Resultados e Discussão

Ao final do ciclo de desenvolvimento de quatro meses, esta seção avalia os resultados alcançados pelo projeto "Me Leva App" em comparação com os objetivos e requisitos definidos no Termo de Abertura, além de discutir os desafios enfrentados pela equipe e as soluções implementadas.

8.1. Comparação com Objetivos Iniciais

O objetivo principal do projeto era a entrega de um Produto Mínimo Viável (MVP) de um aplicativo mobile para adoção de pets, acompanhado de sua documentação técnica. **Este objetivo foi alcançado com sucesso**. O MVP entregue contempla o fluxo de negócio essencial, permitindo que usuários com perfil de protetor cadastrem animais e que usuários adotantes solicitem e concluam uma adoção.

As seguintes funcionalidades principais, alinhadas aos casos de uso, foram implementadas e são demonstráveis na aplicação final:

- **Gestão de Contas de Usuário (UC 1 e 2):** Criação, login e atualização de perfis para Adotantes e Protetores, com distinção de funcionalidades baseada no tipo de perfil.
- **Gestão de Pets (UC 3):** Protetores podem cadastrar, visualizar, editar e remover seus próprios pets através da tela "Meus Pets".
- **Sistema de Adoção (UC 6):** Implementação completa do fluxo de adoção, incluindo a solicitação por parte do adotante, a visualização e gestão (aceite/recusa) pelo protetor, e o acompanhamento do status por ambas as partes.
- **Sistema de Favoritos:** Usuários podem favoritar pets na tela de exploração e visualizá-los em uma lista dedicada.
- **Match por Compatibilidade com IA (UC 7):** O sistema de match, que utiliza a API da OpenAI para calcular um score de compatibilidade, foi totalmente implementado, incluindo o formulário de "Perfil de Match"e a exibição dos scores na tela de exploração.

Devido ao prazo limitado e à necessidade de priorizar o fluxo principal de adoção, algumas funcionalidades definidas no escopo inicial não puderam ser implementadas. O **Acompanhamento pós-adoção (UC 7)**, o **Histórico de Rastreabilidade (UC 4)**, e a **Manutenção do sistema (UC 8)** por um administrador não foram concluídos. Adicionalmente, funcionalidades como a verificação de e-mail e a recuperação de senha (**UC 5.3 e 5.6**) não foram implementadas, pois exigiriam o deploy da aplicação em um servidor com um serviço de e-mail configurado, o que estava fora do escopo do MVP para este projeto.

8.2. Principais Desafios e Soluções

Durante o desenvolvimento, a equipe enfrentou desafios técnicos e de gerenciamento, que foram superados com colaboração e decisões estratégicas.

- **Desafio 1: Integração de Múltiplos Serviços:** A arquitetura distribuída, com um frontend em React Native e duas APIs de backend (Java e Python) em tecnologias distintas, apresentou um desafio inicial de integração. Garantir a comunicação fluida entre o emulador do aplicativo e os serviços rodando localmente nas máquinas dos desenvolvedores exigiu uma padronização rigorosa dos endereços de rede e das portas utilizadas. **Solução:** A equipe estabeleceu um contrato de API claro para cada serviço e centralizou toda a lógica de chamada no arquivo `api.ts`, simplificando a comunicação e a resolução de problemas de CORS e rede.
- **Desafio 2: Latência da API de Inteligência Artificial:** As primeiras tentativas de integração com a API da OpenAI resultaram em longos tempos de espera no frontend, pois o processamento do match era síncrono. Isso congelava a interface do usuário, prejudicando a experiência. **Solução:** A arquitetura do serviço de IA foi redesenhada para ser totalmente assíncrona. Conforme visto em `main.py`, foi implementado um sistema de tarefas em segundo plano (*background tasks*) com um mecanismo de *polling*. O frontend agora envia a requisição e recebe uma resposta imediata, enquanto o processamento pesado ocorre no servidor de forma independente, o que tornou a funcionalidade de match viável e responsiva.
- **Desafio 3: Gerenciamento do Prazo:** O cronograma de quatro meses era ambicioso para a quantidade de funcionalidades planejadas. A equipe percebeu que tentar implementar 100% do escopo inicial poderia comprometer a qualidade do produto final. **Solução:** Utilizando uma abordagem ágil com planejamento de sprints, a equipe realizou reuniões de priorização e tomou a decisão estratégica de focar no fluxo de valor principal — o processo de adoção e o sistema de match. Funcionalidades secundárias, como o acompanhamento pós-adoção, foram conscientemente adiadas para garantir a entrega de um MVP robusto e funcional dentro do prazo.