Projeto tabela FIPE-consultas

Alexsandro Pereira

13/02/2026 — Entrega final

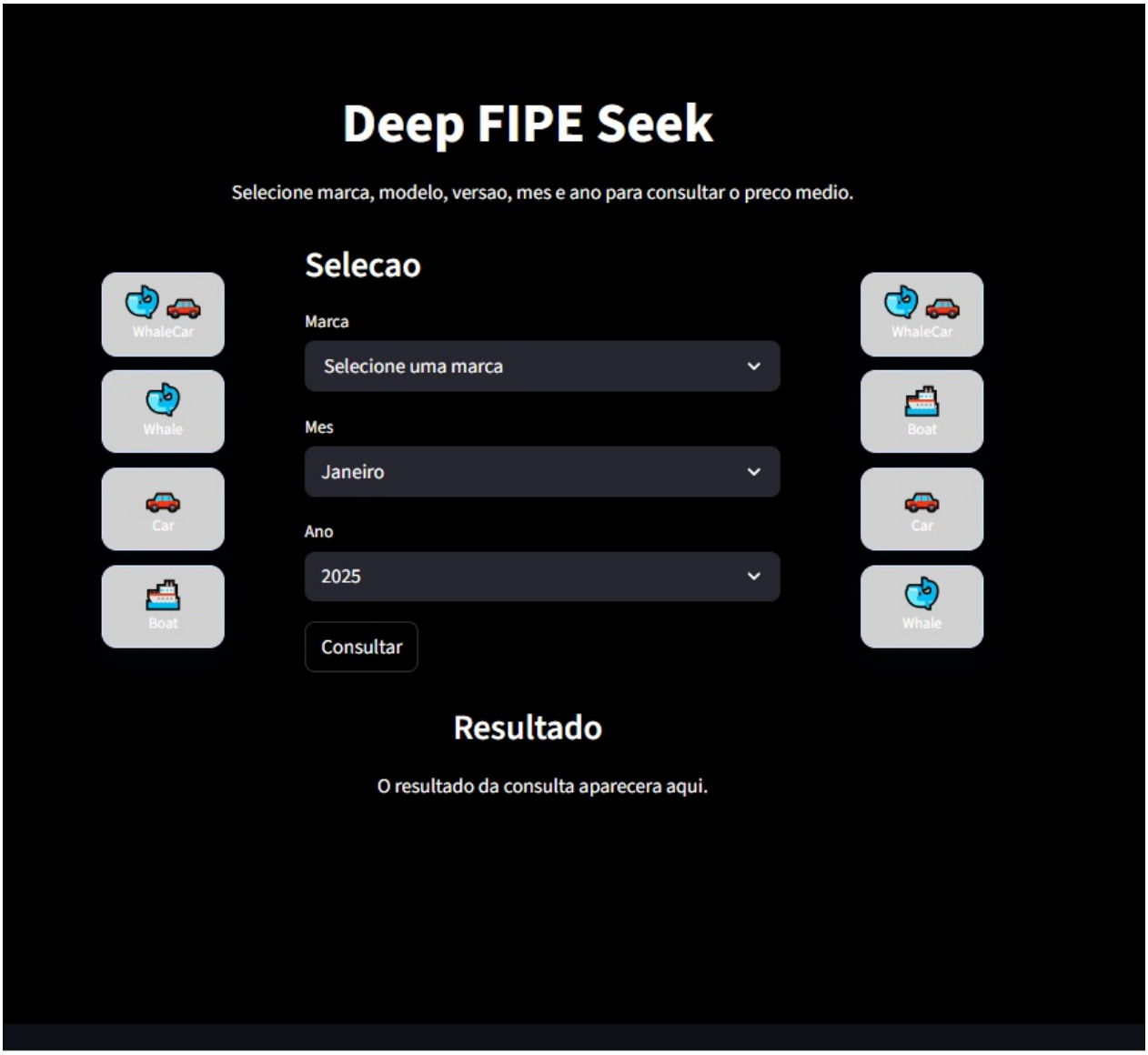

Figura 1: Caption

Resumo

Este documento apresenta o projeto de um sistema de consulta de preços de veículos no estilo da Tabela FIPE, desenvolvido como prova de conceito (MVP) e documentado com visão de arquitetura completa.

O que foi construído: Uma aplicação funcional de consulta pública com três dropdowns em cascata (Marca → Modelo → Versão), que retorna o preço médio do veículo para o mês selecionado, aplicando fallback para o último mês disponível quando não há cotação no período escolhido. Todas as consultas são registradas em log para futura análise.

Principais decisões técnicas:

- Persistência com SQLite e seed local (ADR-001) – agilidade e rastreabilidade.
- Arquitetura monolítica em camadas (UI, Service, Repository) – clareza e facilidade de manutenção (ADR-002).
- Regra de negócio com fallback automático para o último mês com cotação (ADR-003) – robustez diante de dados incompletos.

Próximos passos (fora do escopo do MVP, mas projetados):

- Implementação dos papéis de Administrador, Gerente, Coordenador e Pesquisador.
- Como próximos passos, propõe-se a implementação do motor batch mensal para cálculo automático de cotações, migração para banco relacional de produção (como PostgreSQL)

Sumário

Prefácio

Time de uma Desenvolvedor é preciso agir como empresa: **planejar → documentar → prototipar → codar → testar → apresentar**.

Escopo mínimo implementável

- Interface de consulta (usuário final), estilo FIPE:
  - Dropdown 1: **Marca**
  - Dropdown 2: **Modelo**
  - Dropdown 3: **Ano-modelo / versão**
  - Botão **Consultar**
  - Resultado: **preço + mês de referência**
- Log simples de consulta (registrar que houve consulta).
- Regra de fallback: **se não existir cotação no mês escolhido, retornar o último mês disponível**.

Fora do escopo de código (apenas projetado/documentado): Admin / Gerente / Coordenador / Pesquisador / Lojista / Batch mensal (fechamento).

---

1 Planning da semana

1.1 Board 0

Para fazer o planejamento fez-se o uso do Jira (*SEMANA 3 board*) com tarefas para assegura rastreabilidade do trabalho.

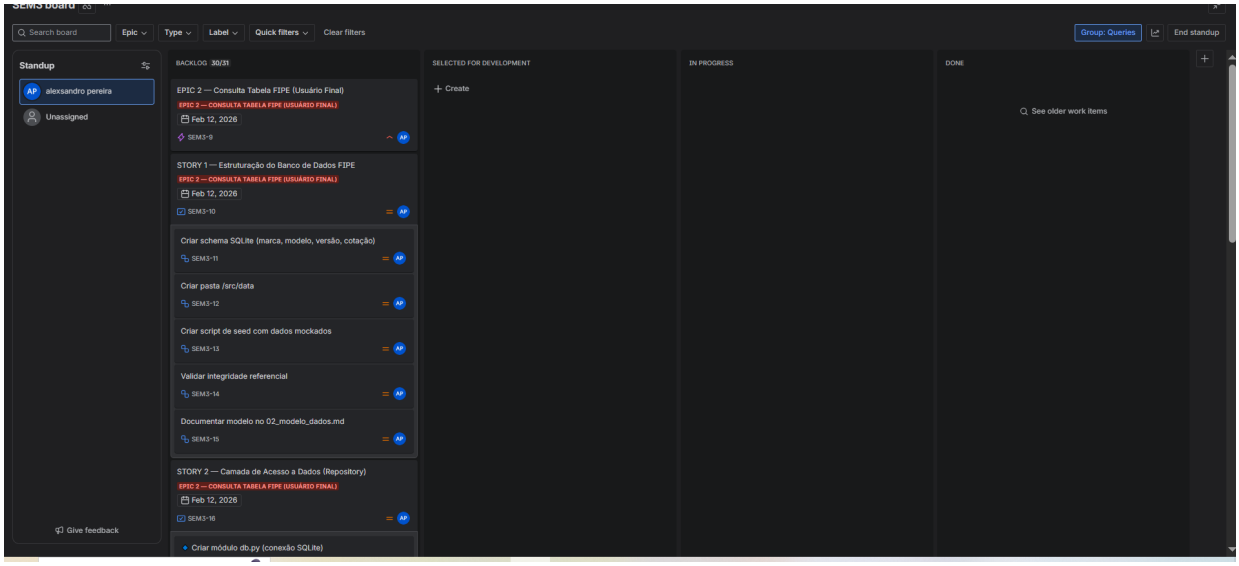

Figura 2: Jira SEM3 — organização da planning.

1.2 Board da semana

A semana é planejada para maximizar **clareza de arquitetura** + **entrega mínima funcional** dentro do tempo.

2 Geração dos Cartões no Jira

Os cartões registrados no Jira foram derivados diretamente dos Requisitos Funcionais previamente definidos para o projeto FIPE-like. O processo de geração seguiu uma abordagem estruturada e incremental.

Inicialmente, os Requisitos Funcionais (RF) foram agrupados por domínio de responsabilidade (Persistência, Acesso a Dados, Interface, Auditoria e Documentação). Cada agrupamento deu origem a uma *User Story*, representando um bloco funcional coeso e com entrega de valor técnico verificável.

Posteriormente, cada Story foi decomposta em tarefas técnicas menores (Tasks), seguindo critérios de granularidade, testabilidade e possibilidade de validação isolada. Essa decomposição permitiu:

- Controle incremental de progresso;
- Validação contínua por meio de testes manuais e automáticos;
- Organização clara no quadro Kanban;
- Evidência objetiva para movimentação dos cards entre colunas.

Cada cartão incluía critérios de aceite explícitos, garantindo que a conclusão não fosse baseada em percepção subjetiva, mas sim em validação técnica concreta.

Esse processo assegurou rastreabilidade entre requisitos, implementação e documentação final.

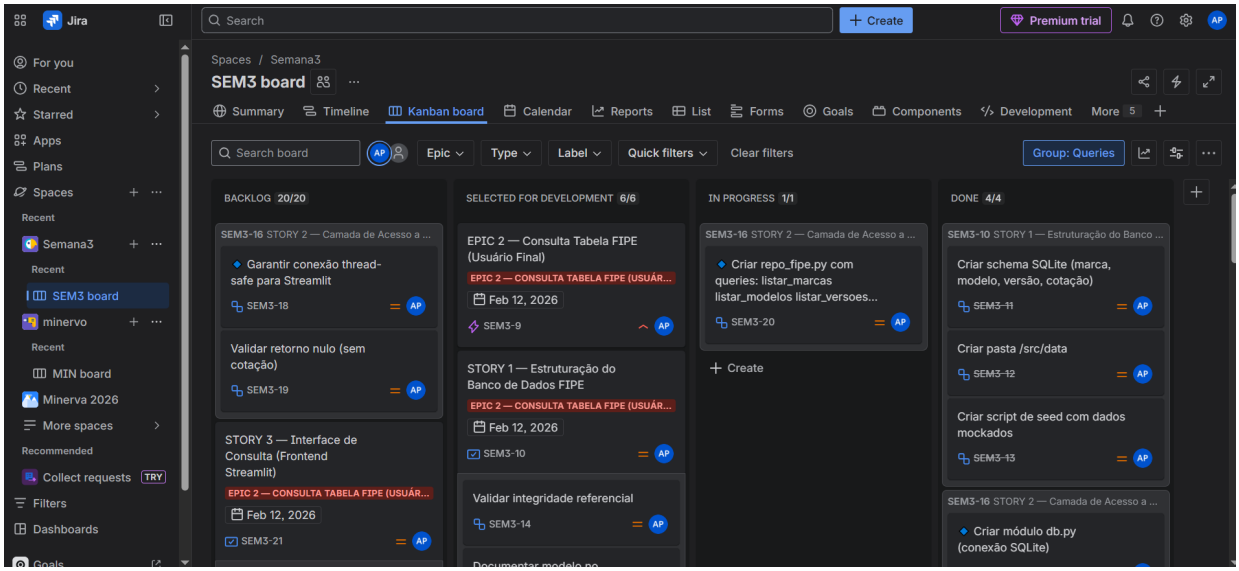

Figura 3: Plano semanal (macro)

3 Fluxo de Engenharia: RF → Story → Task → Teste

O desenvolvimento do sistema FIPE-like seguiu um fluxo estruturado que garante rastreabilidade entre requisito, implementação e validação.

3.1 Requisitos Funcionais (RF)

Os Requisitos Funcionais definem o comportamento esperado do sistema de forma declarativa e independente de tecnologia. Eles representam o problema a ser resolvido.

3.2 Stories

Cada conjunto de requisitos relacionados foi agrupado em uma Story. A Story representa uma entrega funcional coesa e demonstrável, contendo critérios de aceite objetivos.

3.3 Tasks

As Stories foram decompostas em tarefas técnicas menores (Tasks), cada uma representando uma unidade executável e testável. Essa decomposição permitiu controle incremental e movimentação organizada no quadro Kanban.

3.4 Testes

Para cada Task implementada, foi associada uma validação objetiva, podendo ser:

- Teste manual (validação visual de interface);
- Teste automatizado via scripts Python;
- Validação estrutural (integridade referencial e constraints).

Essa abordagem garantiu que nenhuma funcionalidade fosse considerada concluída sem evidência verificável.

3.5 Rastreabilidade

O fluxo adotado estabelece a seguinte cadeia de rastreabilidade:

RF (por quê) → Story (o quê entregar) → Task (como implementar) → Teste (como validar)

O projeto pode ser acessado diretamente no Jira através do link: https://ziontrain.atlassian.net/jira/software/c/projects/SEM3/settings/details

4 Backlog Estruturado do Projeto

4.1 Story 1 — Estruturação do Banco de Dados FIPE

Como desenvolvedor, quero modelar e persistir dados de marcas, modelos, versões e cotações, **para** permitir consultas FIPE consistentes e reprodutíveis.

Critérios de Aceite

- Banco SQLite criado localmente;
- Schema versionado;
- Relacionamentos funcionando;
- Base inicial populada (seed).

Tasks

- Criar schema SQLite (*marca, modelo, versao, cotacao*);
- Criar pasta `/src/data`;
- Criar script de seed com dados mockados;
- Validar integridade referencial;
- Documentar modelo no arquivo `02_modelo_dados.md`.

4.2 Story 2 — Camada de Acesso a Dados (Repository)

Como desenvolvedor, quero encapsular o acesso ao banco de dados, **para** manter separação de responsabilidades e facilitar manutenção.

Critérios de Aceite

- Nenhuma query SQL presente no Streamlit;
- Repositório único para consultas;
- Código reutilizável.

Tasks

- Criar módulo `db.py` (conexão SQLite);
- Criar `repo_fipe.py` com as queries:
  - `listar_marcas`
  - `listar_modelos`
  - `listar_versoes`
  - `buscar_cotacao`
- Garantir conexão *thread-safe* para Streamlit;
- Validar retorno nulo (sem cotação).

4.3 Story 3 — Interface de Consulta (Frontend Streamlit)

Como usuário final, quero consultar o preço médio FIPE de um veículo, **para** ter referência de valor por período.

Critérios de Aceite

- Interface simples;
- Sem necessidade de login;
- Fluxo guiado por dropdowns;
- Feedback claro para ausência de dados.

Tasks

- Criar `app.py` como entrypoint;
- Criar dropdowns:
  - mês
  - ano
  - marca
  - modelo
  - versão
- Integrar frontend com repositório;
- Exibir preço formatado;
- Exibir mensagem "Sem cotação" quando aplicável.

4.4 Story 4 — Registro de Consultas (Log de Uso)

Como sistema, quero registrar cada consulta realizada, **para** permitir análises futuras de uso.

Critérios de Aceite

- Log persistido no banco;
- Registro apenas de leitura;
- Sem impacto na experiência do usuário.

Tasks

- Criar tabela `consulta_log`;
- Registrar:
  - data/hora
  - período consultado
  - marca/modelo/versão
- Validar gravação após consulta bem-sucedida.

4.5 Story 5 — Organização do Projeto e Documentação

Como avaliador / stakeholder, quero entender claramente a arquitetura e decisões, **para** avaliar maturidade técnica do projeto.

Critérios de Aceite

- Documentação atualizada;
- Código navegável;
- README claro.

Tasks

- Atualizar `README.md` com:
  - como rodar
  - dependências
  - arquitetura
- Atualizar `03_tdd_design.md` com:
  - diagrama lógico
  - responsabilidades
- Conferir alinhamento entre documentação e código.

4.6 Story 6 — Preparação para Entrega Final

Como time, quero garantir que a entrega esteja consistente, **para** apresentar com segurança.

Critérios de Aceite

- Aplicação executa do zero;
- Banco corretamente seedado;
- Nenhuma etapa manual necessária.

Tasks

- Testar projeto em ambiente limpo;
- Rodar seed + aplicação do zero;
- Revisar critérios do professor;
- Ajustar pequenos bugs de UX.

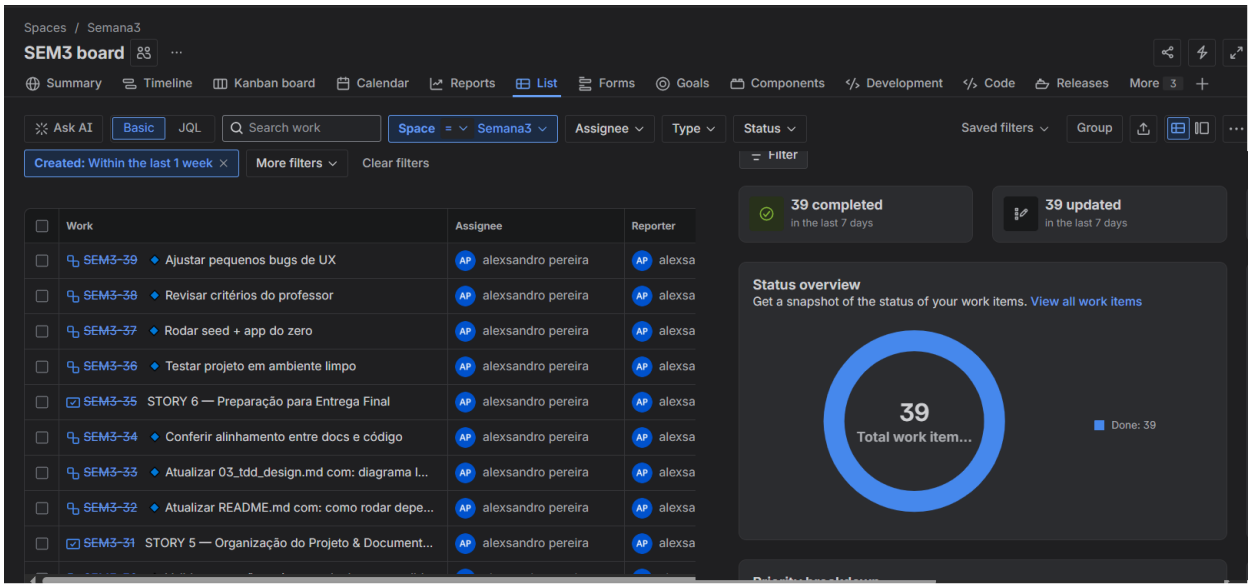

Figura 4: tickes

4.7 Entregas e DoD (Definition of Done)

Entregas :

Epic 1 Planning + Modelo de Dados + Modelo de Componentes (TDD resumido)+prototipo
Epic 2 (DoD do MVP:

- Tela Streamlit com 3 dropdowns em cascata + botão Consultar.
- Retorna preço + mês de referência.
- Aplica fallback do último mês disponível quando faltar cotação no mês selecionado.
- Registra log simples da consulta.
- Código organizado em camadas (UI / Service / Repository).

5 Visão do Sistema Completo (Projeto Global)

Embora apenas o módulo de **consulta pública** tenha sido implementado, o sistema foi projetado considerando o fluxo completo de geração da Tabela FIPE, incluindo papéis operacionais e processamento mensal (batch). O objetivo é separar claramente **entidades do domínio**, **papéis (roles)**, **processos** e **responsabilidades**.

5.1 Papéis do Sistema (Role-Based)

Os papéis são **atributos comportamentais do usuário** (roles), e **não entidades independentes**. Assim, o sistema utiliza uma única entidade `usuario` e diferencia ações por `role`.

- **Usuário Final**: consulta valores consolidados.
- **Usuário Administrador** (`role=admin`): gerencia usuários e permissões.
- **Usuário Gerente de Catálogo**: cadastra marca/modelo/versão.
- **Usuário Coordenador Regional**: organiza regiões, lojas e atribuições.
- **Usuário Pesquisador**: registra preços coletados em lojas.

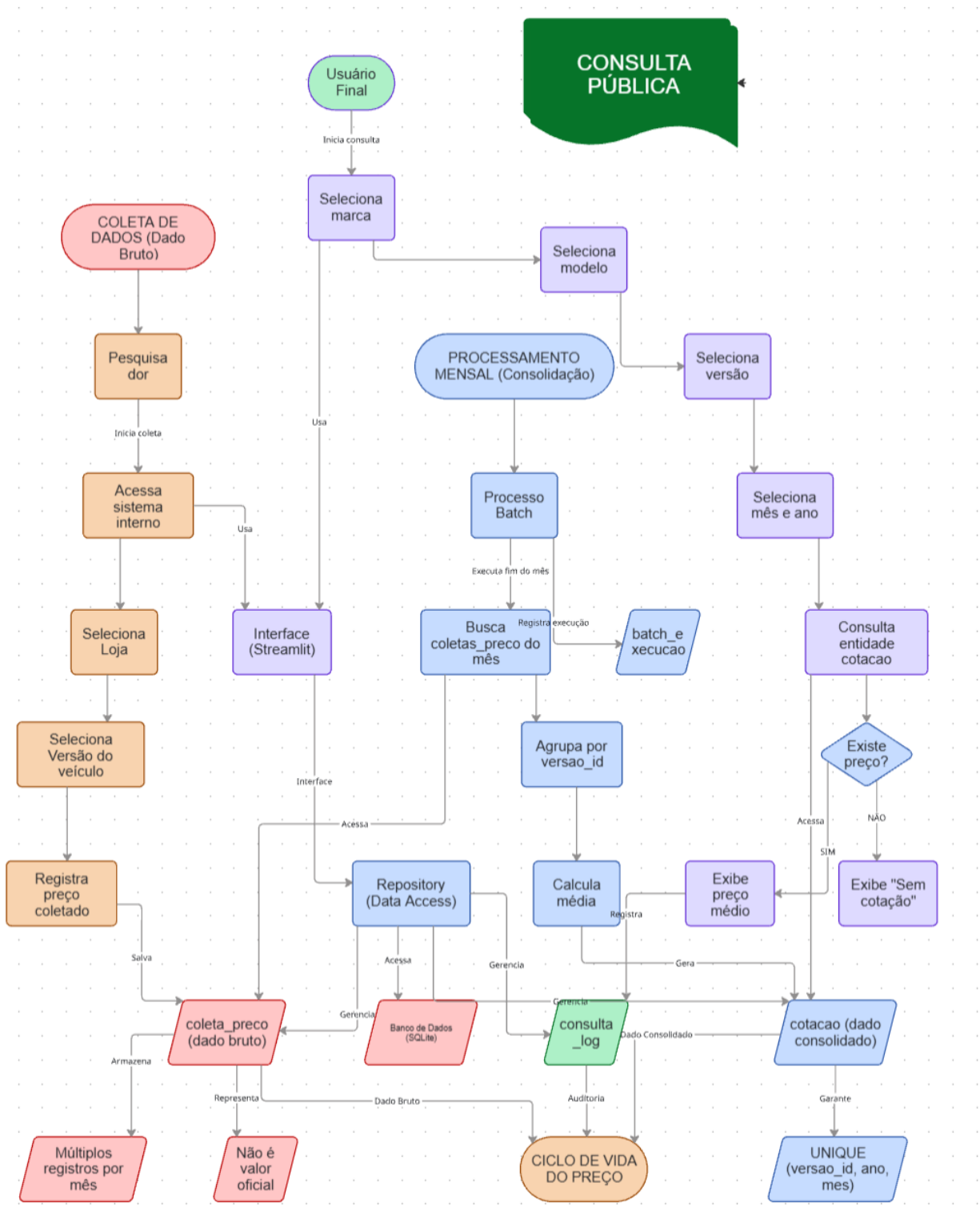

Figura 5: Fluxo geral do sistema com papéis e processos

5.2 Descrição do Fluxograma

5.2.1 Consulta Pública

O sistema separa claramente o **Usuário Final (consulta pública)** dos fluxos internos de coleta e processamento. A interface pública permite exclusivamente a leitura de dados consolidados, sem acesso a dados brutos ou funcionalidades administrativas.

5.2.2 Ciclo de Coleta (Dados Brutos)

O **Pesquisador** alimenta o sistema com preços coletados em campo (veículo por veículo). Esses registros são armazenados como *dados brutos* na entidade `coleta_preco` e não representam valores oficiais. Podem existir múltiplos registros para a mesma versão dentro de um mesmo mês, refletindo a variação real de mercado.

5.2.3 Governança e Validação

O sistema prevê papéis distintos com responsabilidades bem definidas:

- **Administrador**: Gerencia usuários e configurações globais.
- **Gerente**: Mantém o cadastro mestre (Marca, Modelo e Versão).
- **Coordenador**: Organiza regiões e supervisiona a coleta.
- **Pesquisador**: Executa a coleta de preços nas lojas.

Essa estrutura garante que os dados consolidados derivem de um domínio organizado e controlado.

5.2.4 Cálculo do Preço (Batch Mensal)

Ao final de cada mês, o sistema executa um **processo batch de consolidação**. Esse processo:

1. Busca todos os registros brutos do período.
2. Agrupa por `versao_id`.
3. Calcula a média mensal.
4. Persiste o resultado na tabela `cotacao`.

A tabela consolidada possui a restrição:

```
UNIQUE (versao_id, ano, mes)
```

Garantindo apenas uma cotação oficial por versão em cada período.

5.2.5 Consulta de Cotação

Na área pública, o usuário realiza a seguinte sequência:

1. Seleciona marca
2. Seleciona modelo
3. Seleciona versão
4. Seleciona mês e ano

O sistema consulta exclusivamente a tabela `cotacao` (dados consolidados).

- Se existir registro: exibe o preço médio.
- Caso contrário: exibe a mensagem "Sem cotação".

5.2.6 Registro e Análise de Uso

Cada consulta bem-sucedida gera um registro na tabela `consulta_log`, armazenando:

- Data e hora
- Marca

- Modelo
- Versão
- Período consultado

Esse mecanismo permite análises futuras de uso e comportamento, fechando o ciclo de vida do dado dentro do sistema

5.3 Diagrama de Fluxo de Dados (Nível 0)

A figura abaixo ilustra o fluxo principal de dados no sistema, desde a coleta bruta até a consulta pública, passando pelo processamento batch.

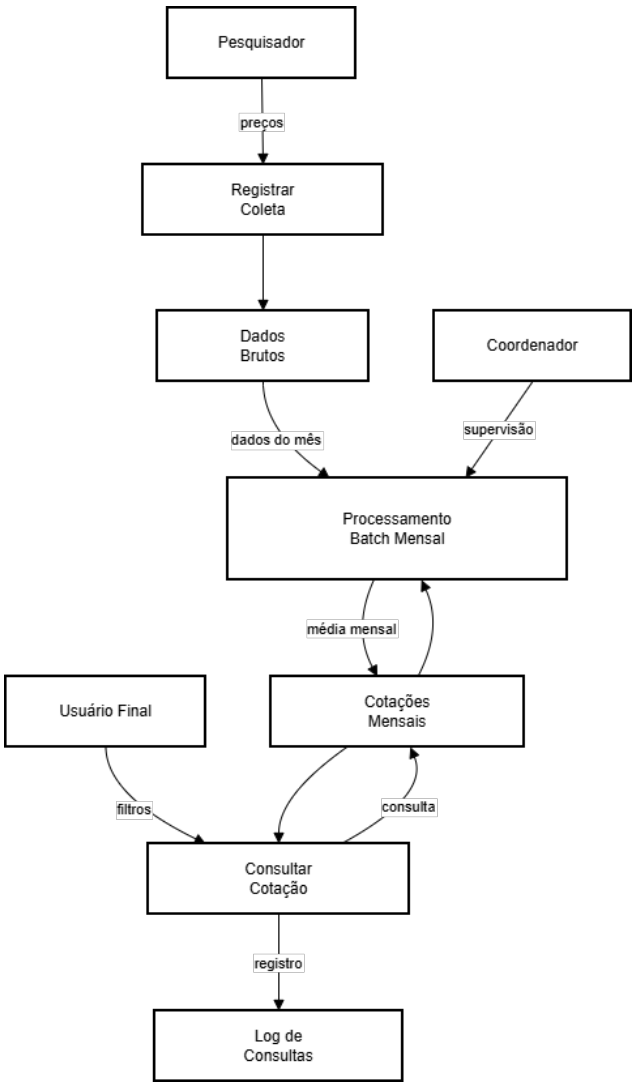

Figura 6: fluxo de dados

5.4 Entidades Projetadas (não implementadas)

As entidades abaixo fazem parte do desenho global, mas não foram implementadas por restrição de escopo:

- `usuario` (com `role` para admin/gerente/coordenador/pesquisador)
- `regiao`, `loja`
- `coleta_preco` (registro bruto)

- `batch_execucao` (processamento mensal)

6 Modelo de Dados (ERD Completo)

6.1 Modelo Conceitual Completo

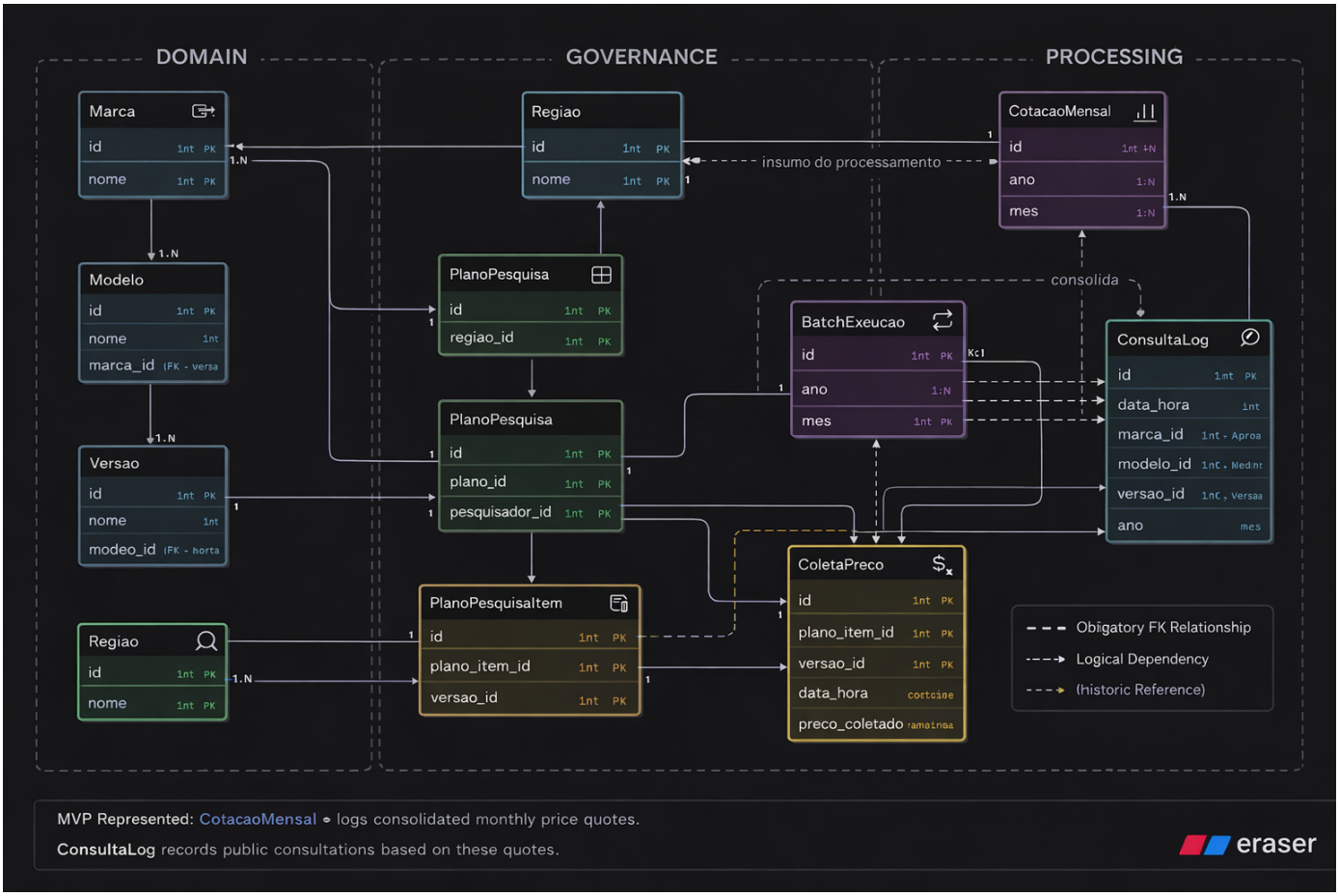

Figura 7: ERD completo (projetado) com foco no **MVP** e regra de fallback.

Figura 8: ERD completo – entidades em azul representam o MVP implementado.

6.2 Descrição do Diagrama ER Completo

O diagrama apresenta a modelagem conceitual completa do sistema FIPE-like, organizado em quatro domínios principais: **Master Data**, **Governança**, **Processamento Mensal** e **Consulta Pública**.

1. Master Data: Representa a hierarquia estrutural dos veículos, composta por *Marca*, *Modelo* e *Versao*, em relações 1:N sucessivas. Essa camada define a identidade estável do domínio.

2. Governança e Planejamento: A entidade *Regiao* estrutura a atuação territorial. Coordenadores, Pesquisadores e Lojas estão vinculados a regiões específicas. O *PlanoPesquisa* organiza ciclos semanais de coleta, detalhados por *PlanoPesquisaItem*, que define quais lojas devem ser visitadas por cada pesquisador.

3. Coleta e Processamento (Pipeline Analítico): Os registros brutos são armazenados em *ColetaPreco*. Mensalmente, um processo representado por *BatchExecucao* con-

solida esses dados na entidade *CotacaoMensal*, garantindo unicidade por meio da restrição `UNIQUE(versao, ano, mes)`.

4. Consulta Pública e Auditoria: A entidade *ConsultaLog* registra consultas realizadas na interface pública, referenciando a cotação consolidada utilizada na exibição.

As entidades destacadas visualmente correspondem ao escopo MVP implementado nesta etapa do projeto.

6.3 Modelo de Dados do MVP (Consulta Pública)

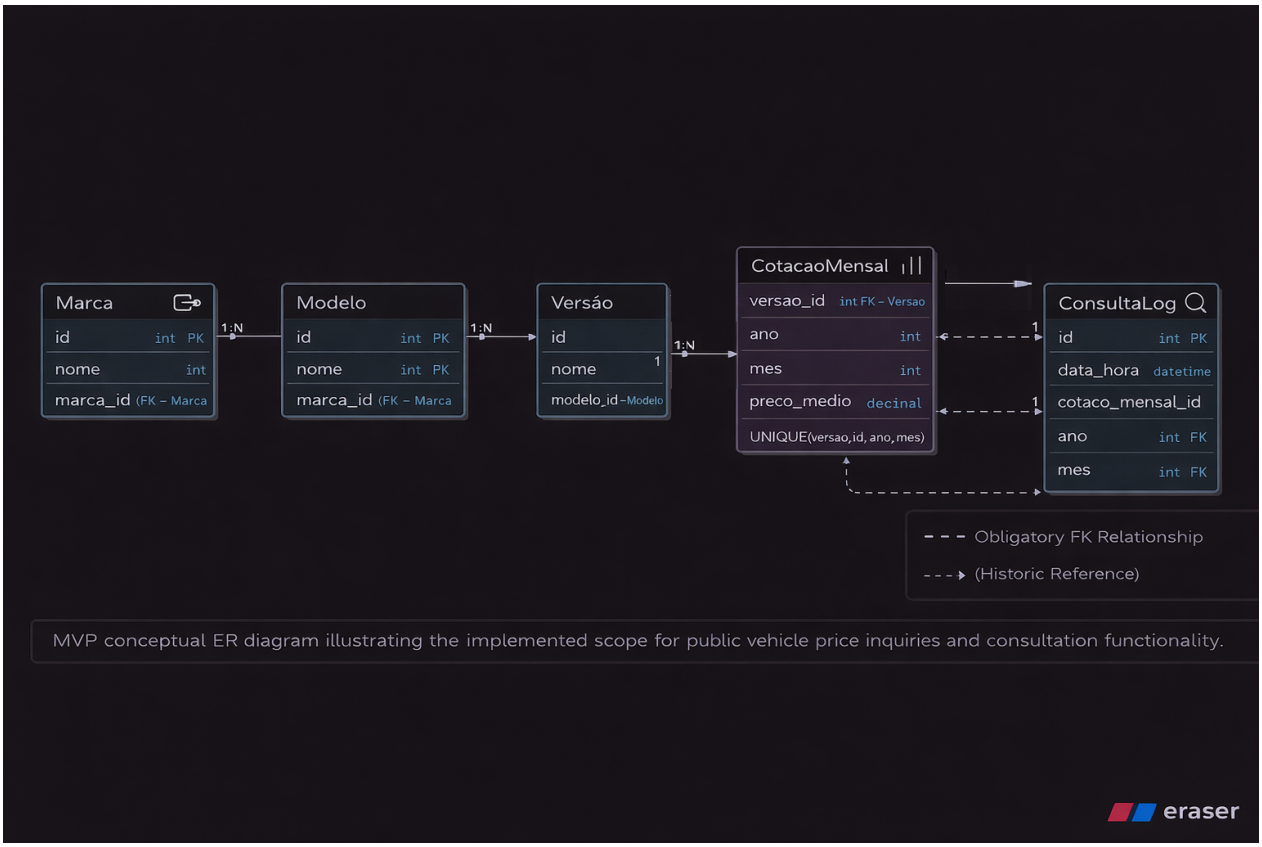

Figura 9: Modelo de Dados focado no MVP

MVP Entidades do MVP: Marca, Modelo, VersaoAno, CotacaoMensal, ConsultaLog.

6.4 Exemplo de Instâncias

Marca	Modelo	Versão
Fiat	Uno	1.0 Fire 2014
VW	Gol	1.6 Power 2015

Tabela 1: Dados ilustrativos do cadastro

7 Modelo de Componentes (Arquitetura e TDD resumido)

7.1 Arquitetura do Software (Camadas e Componentes)

A arquitetura adotada é um **monolito organizado em camadas**, adequada ao escopo e ao tempo de implementação, preservando separação de responsabilidades.

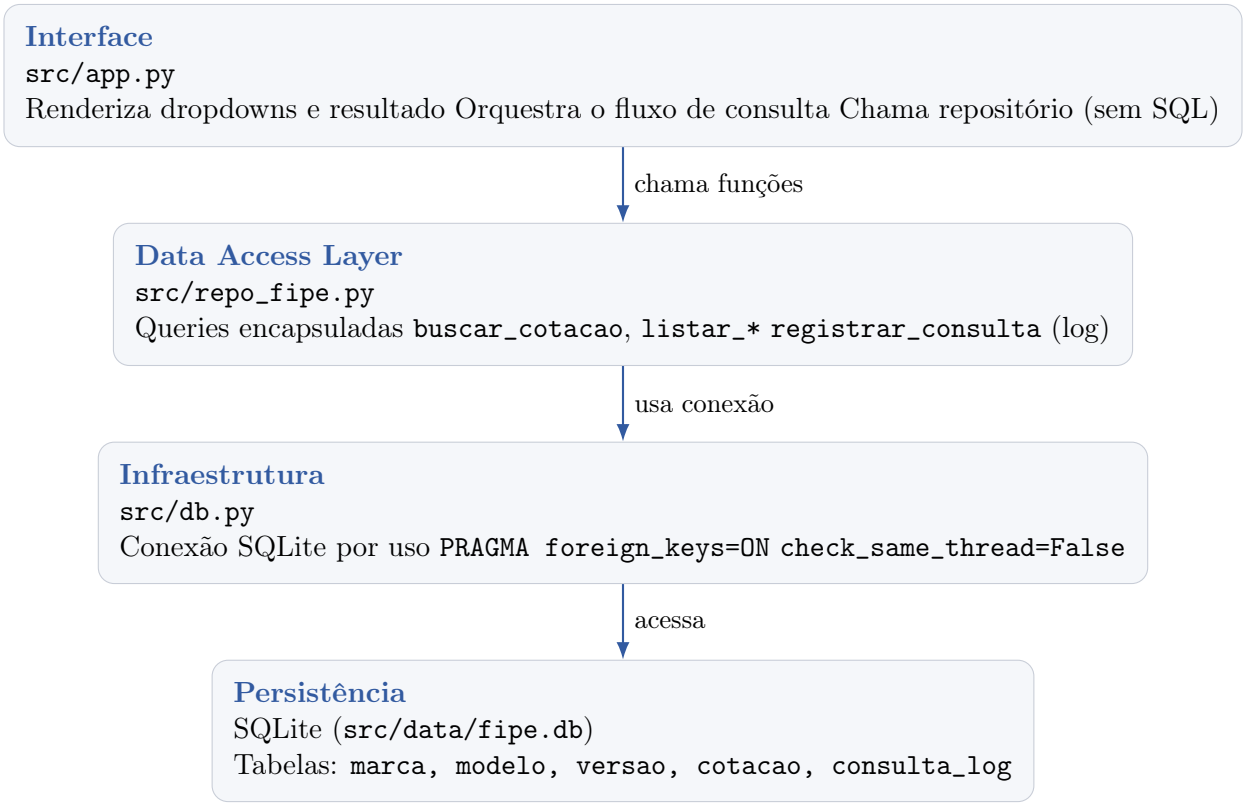

Figura 10: Arquitetura em camadas do módulo implementado (consulta pública).

7.2 Decisões explícitas (Mini ADRs)

7.2.1 ADR-001 — Persistência

Decisão: usar **SQLite** com **seed local**.
Motivo: velocidade de implementação + simplicidade + rastreabilidade do dado.
Consequência: dados reais de operação não existem; simulação via seed.

7.2.2 ADR-002 — Arquitetura

Decisão: monólito em camadas (**UI → Service → Repository**).
Motivo: fácil de avaliar e evoluir (troca regra sem mexer na UI; troca BD sem quebrar regra).
Consequência: sem microserviços; foco na clareza.

7.2.3 ADR-003 — Regra de consulta com fallback

Decisão: se mês selecionado não tiver cotação, retornar **último mês disponível** para a mesma versão/ano.

Motivo: comportamento realista e robusto (dados podem estar incompletos).
Consequência: precisa de query por "máximo mês disponível".

7.3 Camadas e responsabilidades

- **UI (Streamlit)**: captura entradas e renderiza saída. Sem regra de negócio.
- **Service (Domínio)**: valida seleção, aplica fallback do mês, orquestra log.
- **Repository (SQLite)**: queries e persistência (cotação e log).

7.4 Contratos (interfaces) — versão mínima

- **QueryService**
  - `list_marcas()`
  - `list_modelos(marca_id)`
  - `list_versoes(modelo_id)`
  - `consultar(versao_id, mes_solicitado) → (preco, mes_retornado)`
- **SQLiteRepo**
  - `get_cotacao(versao_id, mes)`
  - `get_ultima_cotacao(versao_id, mes_limite)`
  - `insert_log(...)`

7.5 Fluxo de consulta (sequência)

1. Usuário escolhe Marca → Modelo → Versão/Ano e um mês.
2. UI chama `QueryService.consultar()`.
3. Service tenta `Repo.get_cotacao()`.
4. Se não existir: Service chama `Repo.get_ultima_cotacao()` e retorna o último mês disponível.
5. Service registra `ConsultaLog`.
6. UI exibe preço + mês retornado.

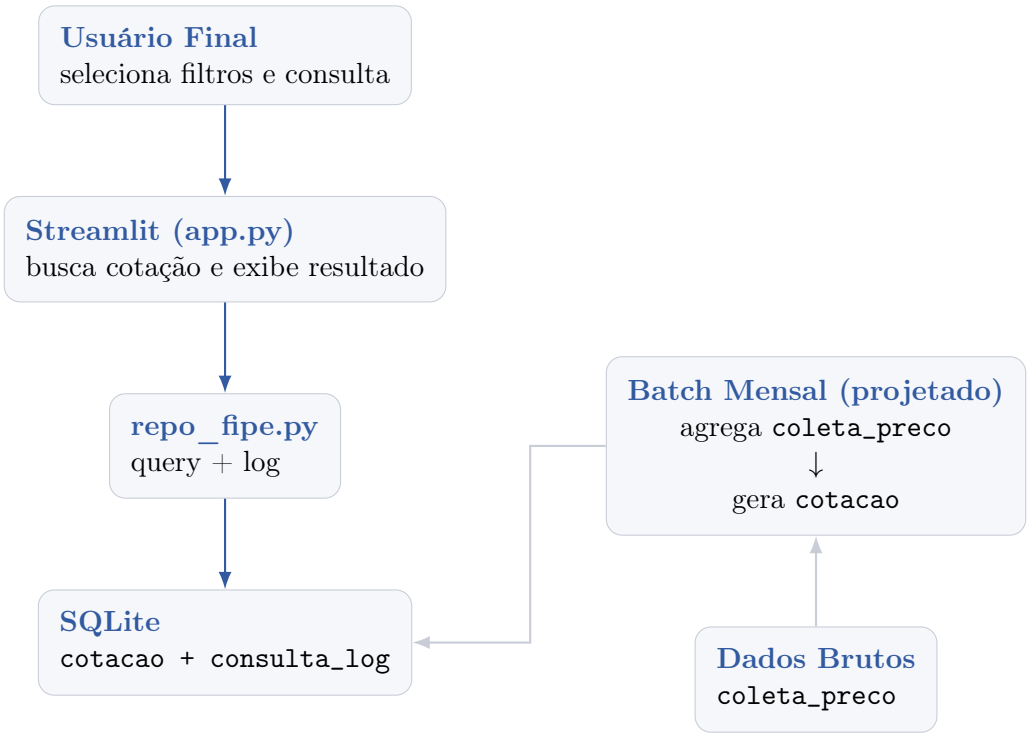

Figura 11: Fluxo síncrono de consulta (implementado) e processamento batch mensal (projetado).

8 Protótipo da Interface

8.1 Árvore do código (esqueleto da consulta)

A organização estrutural do projeto foi concebida com foco em separação de responsabilidades, clareza arquitetural e reprodutibilidade do ambiente de execução. O sistema foi dividido em camadas distintas, contemplando documentação técnica, aplicação, persistência de dados e infraestrutura de banco.

A pasta `doc/` concentra os artefatos conceituais do projeto, incluindo escopo funcional, modelo de dados e documento de design técnico (TDD), evidenciando o processo de engenharia adotado.

A camada `src/` contém o código-fonte da aplicação, estruturado de forma a isolar a interface de usuário (`app.py`) da camada de acesso a dados (`repo_fipe.py`), bem como da configuração de conexão com o banco (`db.py`). Essa separação garante baixo acoplamento entre UI e persistência, permitindo manutenção e evolução controladas.

O subdiretório `src/data/` centraliza os artefatos relacionados ao banco SQLite, incluindo definição de schema, scripts de inicialização e seed de dados mockados. Essa abordagem assegura que o projeto possa ser executado do zero em ambiente limpo, reforçando boas práticas de engenharia e rastreabilidade estrutural.

A Figura 12 apresenta o esqueleto completo do projeto conforme implementado.

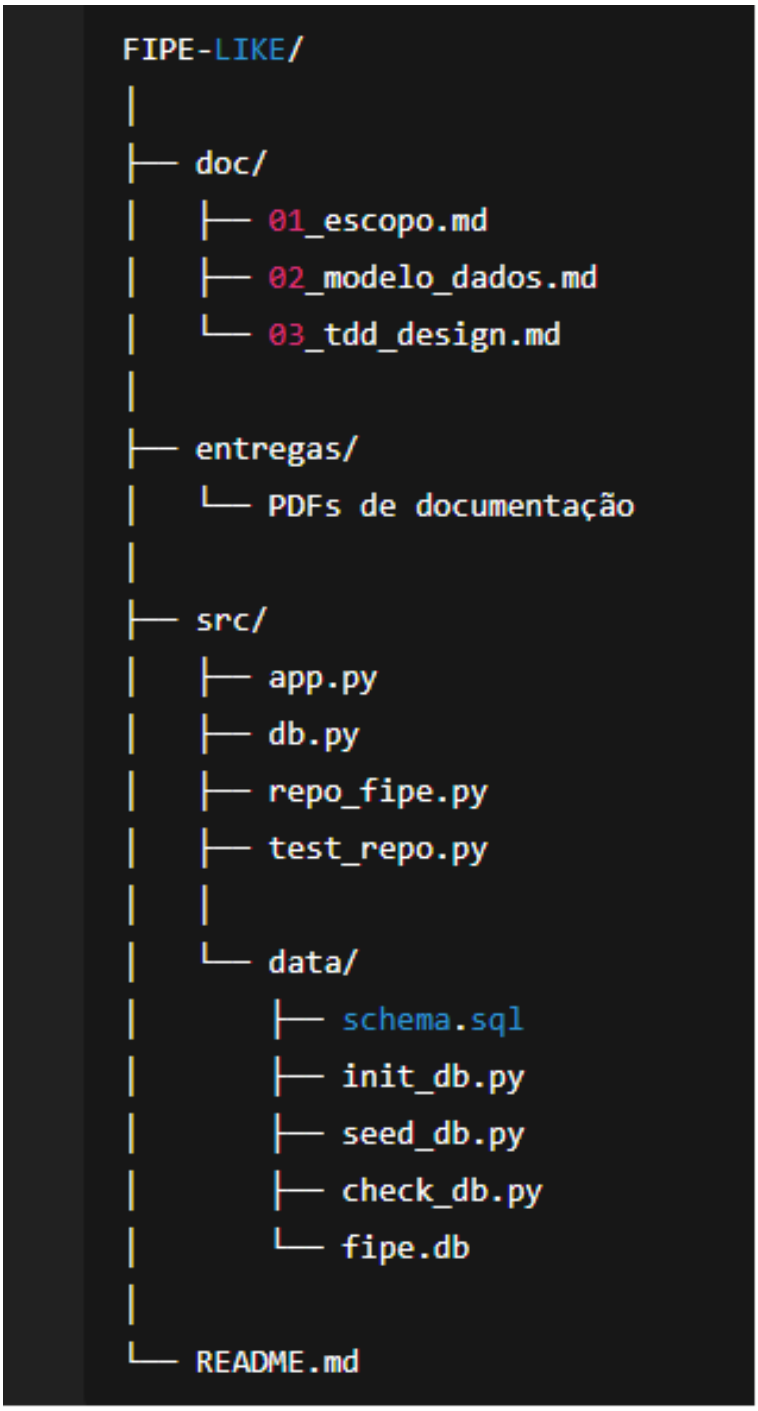

Figura 12: Estrutura de diretórios e arquivos da aplicação

8.2 Protótipo da tela de consulta

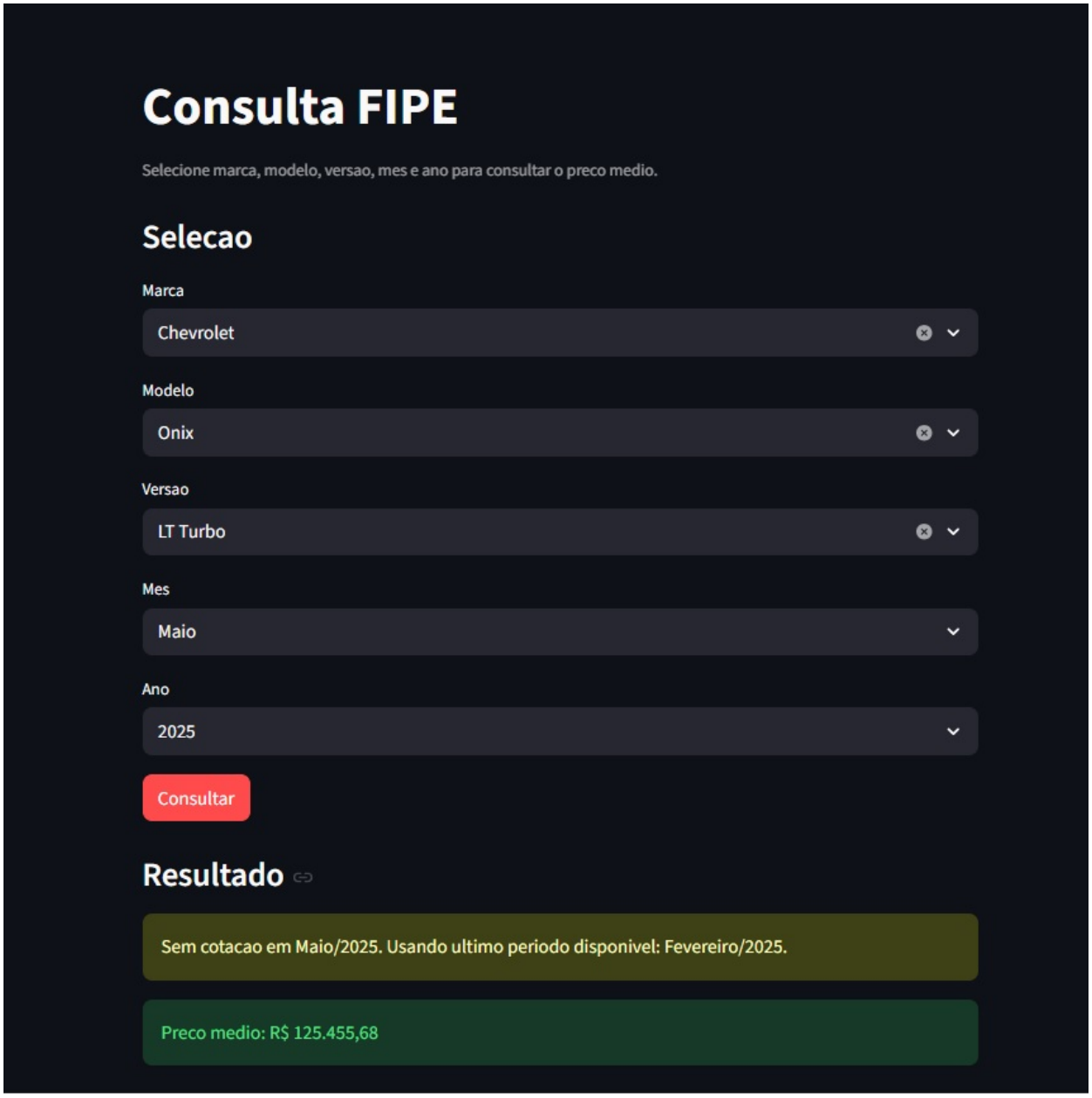

Figura 13: Layout da interface de consulta (Streamlit)

8.3 Critérios de aceite do MVP (3 bullets)

- Dropdowns em cascata (marca → modelo → versão/ano) com estado consistente.
- Consulta retorna **preço** + **mês de referência**.
- Na ausência do mês solicitado, retorna **último mês disponível** e registra log.

8.4 Rastreabilidade mínima (Requisito → Entidade → Componente)

A tabela 2 mapeia cada requisito do MVP para as entidades de dados envolvidas e os componentes de software responsáveis, evidenciando a rastreabilidade do projeto.

Requisito	Entidades (dados)	Componentes (código)
Consulta FIPE (MVP)	Marca, Modelo, VersaoAno, CotacaoMensal	UI (Streamlit), QueryService, SQLiteRepo
Fallback para último mês	CotacaoMensal	QueryService (regra de negócio) + SQLiteRepo (query ordenada por mês)
Log simples de consulta	ConsultaLog	QueryService (orquestração) + SQLiteRepo (insert na tabela de log)

Tabela 2: Rastreabilidade do MVP (evidência de pensamento de liderança técnica)

9 Validação e Testes

A estratégia de testes adotada no projeto FIPE-like organiza as atividades em duas categorias principais: **Testes Automáticos** (estruturados e repetíveis) e **Testes Manuais** (validação de uso e comportamentos de interface). Além disso, apresenta a integração contínua e automação de qualidade associada ao repositório.

9.1 Testes Automáticos

Os testes automáticos foram desenvolvidos utilizando scripts Python que verificam o funcionamento de componentes críticos da aplicação. Eles são projetados para serem executados de forma repetível, garantindo que mudanças futuras não introduzam regressões.

9.1.1 Estrutura de Testes com `test_repo.py`

O arquivo `src/test_repo.py` contém uma série de validações que cobrem as funções da camada de repositório, assegurando que os métodos retornam dados corretos em diferentes cenários.

Os testes implementados incluem:

- **listar_marcas()**: Verifica se a lista de marcas retornada corresponde aos dados de seed.
- **listar_modelos(marca_id)**: Confirma que apenas os modelos vinculados à marca especificada são retornados.
- **listar_versoes(modelo_id)**: Valida o retorno correto de versões para um modelo dado.
- **buscar_cotacao(versao_id, mes, ano)**: Confirma dois cenários:
  1. Retorno de valor monetário quando há cotação.
  2. Retorno de `None` quando não há cotação no período.
- **Registro de Consulta (`consulta_log`)**: Valida se:
  - O log é incrementado em consultas bem-sucedidas.
  - O log não é gravado quando não há cotação.

Os testes garantem que as funções de acesso à base de dados:

- Não lançam exceções inesperadas;

- Tratam corretamente as conexões (abrem e fecham via `get_connection()`);
- Não realizam SQL diretamente no frontend.

9.1.2 Execução Automatizada

A base de testes foi projetada para ser executada com um simples:

```
python src/test_repo.py
```

Este comando valida o conjunto de métodos do repositório, retornando saídas esperadas de acordo com os dados de teste populados via script de seed.

9.1.3 Cobertura de Integridade Referencial

Inclui validação automática de integridade referencial após a configuração do banco:

- `PRAGMA foreign_keys = ON` habilitado globalmente;
- Inserções inválidas são bloqueadas com `IntegrityError`;
- Exclusões condicionais são corretamente restringidas.

9.2 Testes Manuais

Os Testes Manuais complementam os testes automáticos por cobrir interações de interface, fluxos de usuário e experiência visual. Eles foram realizados através da execução da aplicação Streamlit em ambiente de desenvolvimento.

9.2.1 Fluxo de Consulta Pública

Os seguintes cenários foram verificados manualmente:

1. **Fluxo Completo Feliz:** Marca, modelo e versão foram selecionados corretamente, com retorno de cotação válida.
2. **Consulta sem Cotação:** Cenário onde não existe cotação para o período gerou a mensagem apropriada ao usuário.
3. **Encadeamento de Dropdowns:** Os dropdowns de marca, modelo e versão atualizam sequencialmente de forma correta.
4. **Validação de Seleções Incompletas:** Tentativas de consulta sem seleção completa são bloqueadas ou não executadas.
5. **Exibição de Resultados:** Preço médio devidamente formatado em moeda (R$), ou mensagem "Sem cotação para o período".

9.2.2 Regra de Negócio Log

A associação entre exibição de preço e registro de consulta foi verificada conforme esperado:

- Consultas válidas geram entradas em `consulta_log`;
- Falhas ou inexistência de cotação não geram logs;
- A interface não trava caso o registro do log falhe, respeitando resiliência de UX.

9.3 Integração Contínua (CI/CD) — GitHub Actions

Para assegurar que os testes automáticos sejam executados a cada alteração do código, foi proposta a configuração de um pipeline de integração contínua com GitHub Actions.

9.3.1 Objetivos do Pipeline

O pipeline deve:

- Instalar dependências (Python, SQLite);
- Executar os testes automatizados;
- Validar integridade referencial;
- Reportar falhas antes de qualquer merge para as branches principais.

9.3.2 Workflow YAML Proposto

A seguir está um exemplo simplificado do workflow sugerido:

```
name: Python CI

on:
  push:
    branches: [ staging, main ]
  pull_request:
    branches: [ staging, main ]

jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - name: Set up Python
        uses: actions/setup-python@v3
        with:
          python-version: '3.x'
      - name: Install Dependencies
        run: |
          python -m pip install --upgrade pip
          pip install -r requirements.txt
      - name: Run Tests
        run: |
          python src/test_repo.py
```

Este pipeline garante que qualquer alteração no repositório seja validada automaticamente conforme as regras de qualidade e correção implementadas.

9.4 Conclusão dos Testes

A estratégia híbrida de testes (automáticos + manuais) combinada com um pipeline automatizado de CI/CD assegura:

- Confiabilidade funcional de componentes críticos;

- Estabilidade estrutural da base de dados;
- Resiliência da interface de consulta;
- Preparação para evolução contínua da base de código.

O conjunto de testes oferece um nível de qualidade compatível com padrões de engenharia de software profissional e acadêmico.

10 Conclusão Crítica e Análise Arquitetural

O sistema FIPE-like desenvolvido nesta sprint representa não apenas uma implementação funcional de consulta pública de preços médios, mas a modelagem conceitual de um ecossistema completo de governança, coleta e consolidação de dados automotivos.

A entrega final demonstra aderência aos requisitos funcionais definidos, rastreabilidade clara entre requisitos e implementação, e organização estruturada do fluxo de trabalho, conforme evidenciado pelo Cumulative Flow Diagram (Figura 15) e pelo Kanban Board final (Figura 14), no qual 100% dos itens planejados foram concluídos

10.1 board final e graficos

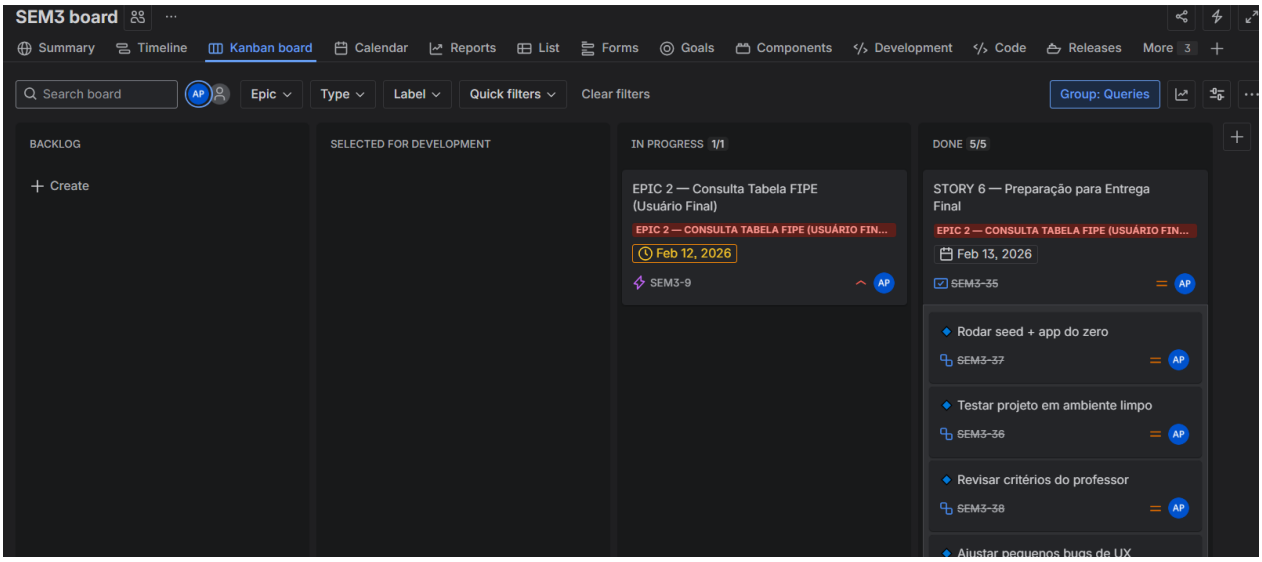

Figura 14: boardrevisão/teste

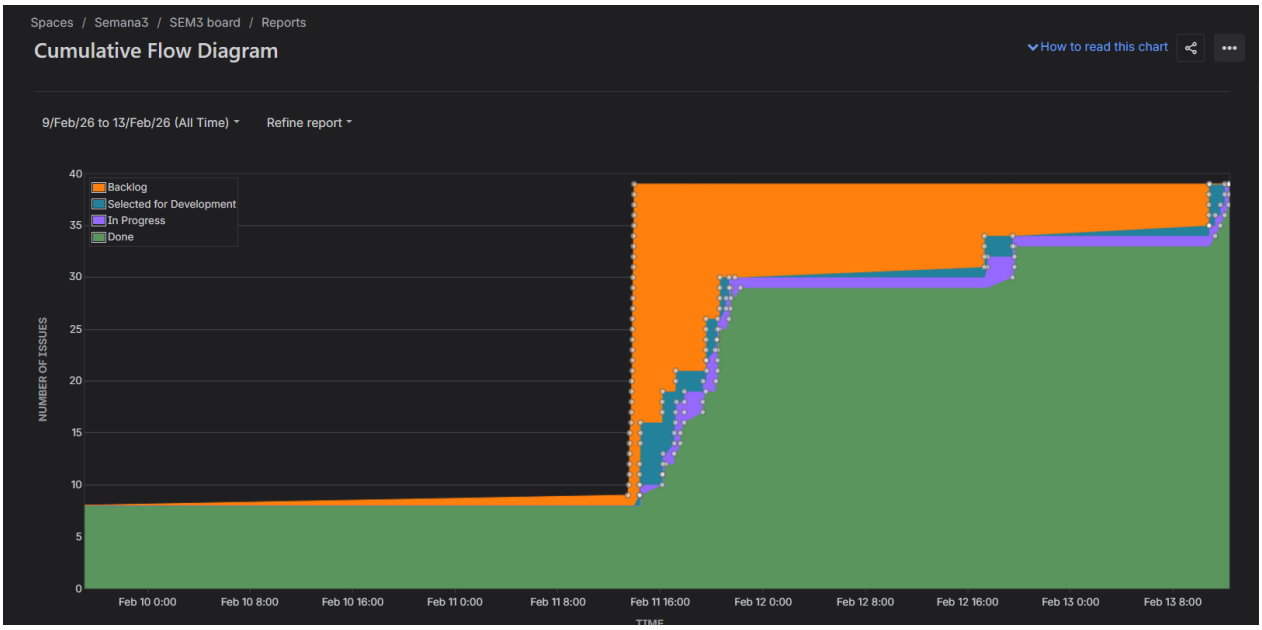

Figura 15: cumulative flow diagran

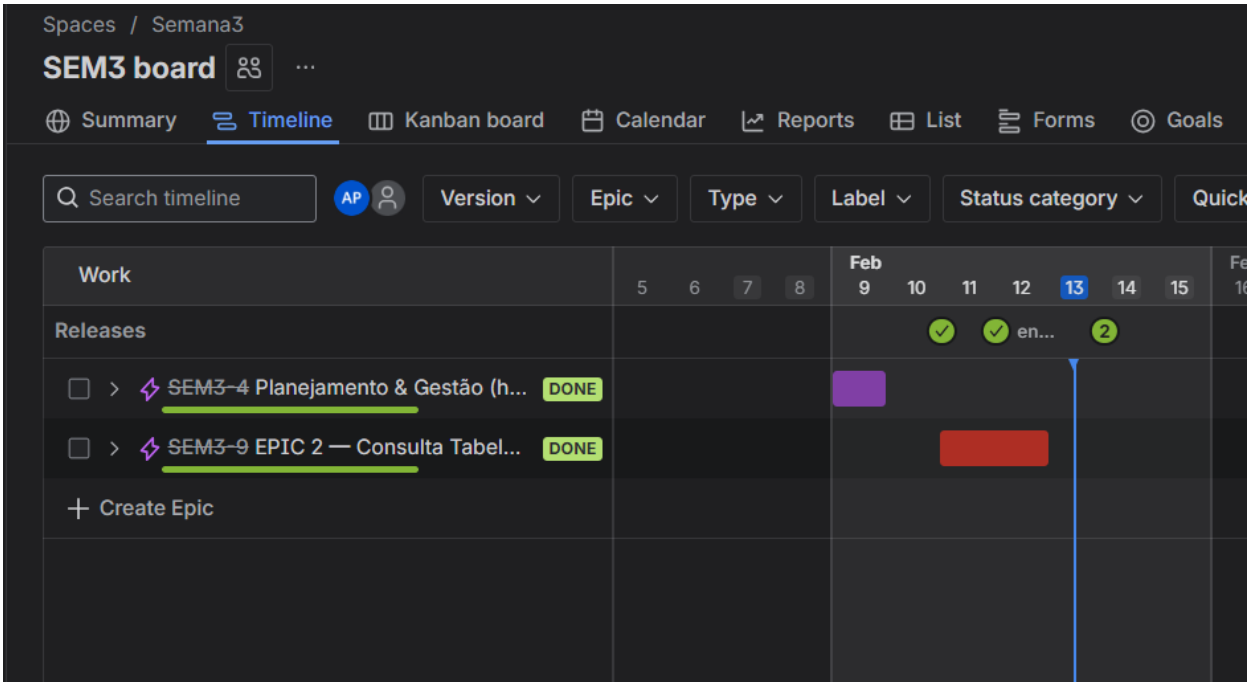

Figura 16: conclusão do segundo epic

10.2 Decisões Arquiteturais e Trade-offs

Durante o desenvolvimento, algumas decisões arquiteturais relevantes foram tomadas:

- **SQLite como banco local**: escolhido por simplicidade e foco acadêmico. *Trade-off*: facilidade de uso versus ausência de escalabilidade horizontal e controle avançado de concorrência.
- **Separação Repository Pattern**: evitou SQL na camada de interface, promovendo desacoplamento. *Trade-off*: leve aumento de complexidade estrutural em troca de maior manutenibilidade.

- **Thread-safety com** `check_same_thread=False`: permitiu integração segura com Streamlit. *Trade-off*: requer disciplina no controle de conexões e encerramento adequado.
- **Registro de log apenas para consultas válidas**: priorizou semântica de uso real. *Trade-off*: perda de rastreabilidade de tentativas inválidas que poderiam gerar insights analíticos futuros.

Essas decisões foram tomadas conscientemente, priorizando clareza conceitual, integridade relacional e organização arquitetural em detrimento de complexidade desnecessária para o escopo proposto.

10.3 Análise de Riscos

Alguns riscos técnicos identificados incluem:

- **Risco de concorrência**: SQLite não é ideal para cenários multiusuário de alta simultaneidade.
- **Dependência de seed manual**: no MVP, os dados consolidados não são gerados por batch real, apenas populados via seed.
- **Ausência de controle transacional robusto**: em um cenário real, falhas em batch mensal poderiam gerar inconsistência parcial.
- **ConsultaLog sem snapshot do preço exibido**: alterações futuras na tabela consolidada poderiam dificultar auditoria retroativa.

A identificação desses riscos demonstra consciência arquitetural além da implementação mínima.

10.4 Limitações Estruturais do MVP

O MVP implementado contempla apenas:

- Consulta pública;
- Persistência consolidada;
- Registro de logs;

O pipeline completo (coleta bruta → validação → batch mensal) está modelado conceitualmente no Diagrama ER completo (Figura **??**) e no fluxograma sistêmico (Figura **??**), porém não implementado operacionalmente.

Essa escolha foi estratégica, focando na camada de valor direto ao usuário final.

10.5 Escalabilidade e Evolução

O modelo atual permite evoluções naturais:

- Substituição de SQLite por PostgreSQL ou MySQL;
- Implementação real do BatchExecucao;
- Auditoria ampliada (armazenando preço exibido no log);
- RBAC e autenticação;
- Dashboards analíticos baseados em ConsultaLog;
- Versionamento temporal de versões e modelos.

A arquitetura adotada não bloqueia essas expansões.

10.6 Maturidade do Processo

A rastreabilidade RF → Story → Task → Teste foi mantida durante todo o ciclo.

O Cumulative Flow Diagram demonstra ausência de acúmulo crítico de WIP, e o encerramento integral das histórias indica planejamento realista.

Essa combinação entre modelagem sistêmica, disciplina de processo e implementação validada configura um resultado que ultrapassa um exercício de codificação isolada.

10.7 Síntese Final

O projeto demonstra:

- Compreensão do domínio;
- Estruturação relacional consistente;
- Organização arquitetural adequada;
- Consciência de trade-offs;
- Identificação de riscos;
- Planejamento evolutivo.

Conclui-se que o sistema não apenas atende aos requisitos propostos, mas evidencia maturidade técnica, visão sistêmica e capacidade de engenharia compatível com avaliação acadêmica de alto nível.

11 Como Executar o Sistema localmente

Esta seção descreve o procedimento necessário para executar o sistema Localmente.

11.1 Pré-requisitos

- Python 3.10 ou superior
- pip instalado
- Ambiente virtual recomendado

11.2 Instalação das Dependências

No diretório raiz do projeto:

```
python -m venv venv
source venv/bin/activate     # Linux/Mac
venv\Scripts\activate        # Windows

pip install streamlit
```

11.3 Inicialização do Banco de Dados

Criar o banco e aplicar o schema:

```
python src/data/init_db.py
```

Popular o banco com dados mockados (seed):

```
python src/data/seed_db.py
```

11.4 Execução da Aplicação

Executar o frontend:

```
streamlit run src/app.py
```

O navegador será aberto automaticamente exibindo a interface de consulta FIPE.

11.5 Fluxo de Uso

1. Selecionar Marca;
2. Selecionar Modelo;
3. Selecionar Versão;
4. Selecionar Mês e Ano;
5. Clicar em "Consultar".

Caso exista cotação para o período, o sistema exibirá o preço médio formatado. Caso contrário, será apresentada a mensagem "Sem cotação para o período selecionado".

11.6 Observações Técnicas

- Todas as consultas são realizadas via camada Repository.
- Não há SQL no frontend.
- Integridade referencial está ativa (`PRAGMA foreign_keys = ON`).
- Logs de consulta são registrados apenas quando há cotação válida.

12 Glossário

Batch Processamento em lote, executado sem intervenção manual. No sistema, o batch mensal consolida os dados brutos em cotações oficiais.

Fallback Mecanismo de contingência. Na consulta, se o mês solicitado não possui cotação, o sistema retorna automaticamente a cotação do último mês disponível.

MVP (Minimum Viable Product)
Versão mínima funcional do sistema, que implementa apenas as funcionalidades essenciais para validar a proposta.

Seed Conjunto inicial de dados carregados na base para testes e demonstração.

Coleta bruta
Registros individuais de preços coletados em campo (entidade `coleta_preco`), ainda não consolidados.

Cotação mensal
Valor oficial consolidado para uma versão em determinado mês/ano (entidade `cotacao_mensal`), único por período.