SRS — Monitor de Preços Agro (Atacado/Varejo)

Versão: 1.0
Data: 07/10/2025
Equipe: (preencher)
Contato: (preencher)

---

1. Resumo executivo

O **Monitor de Preços Agro** é uma aplicação em Python (CLI) que registra, consolida e analisa preços de **hortifrutis** tanto no **atacado (CEASAs)** quanto no **varejo (hortifrutis/mercados)**. O sistema normaliza unidades (ex.: caixa 23 kg → kg), calcula **média móvel, variação percentual e alertas** de outliers, e persiste dados em **Oracle** (com espelho em arquivos **JSON**). O foco do MVP é **atacado**, com estrutura pronta para ativar **varejo**.

---

2. Justificativa (Contexto e Problema)

- **Volatilidade de preços** em hortifruti dificulta negociação, planejamento de compra/venda e gestão de margem, especialmente para pequenos produtores e comerciantes.
- Dados de atacado são relativamente padronizados (CEASAs), mas ainda **pouco acessíveis** numa visão histórica e analítica simples.
- No **varejo**, faltam ferramentas de coleta e comparação localizadas (bairro/loja) para apoiar **precificação** e **promoções**.
- Uma ferramenta **leve, validada e auditarável** ajuda a profissionalizar decisões com base em **séries históricas**, reduzindo percepção subjetiva e empírica.

Benefícios esperados: - Visão rápida de **tendências** (média móvel 7d, variações diárias/semanais). - **Alertas** de desvios relevantes para negociar melhor e programar compras. - **Padronização** (normalização de unidade e tipagem) e **auditoria** (logs) dos registros. - Base técnica que reforça aprendizado em **Python + Oracle + arquivos** (escopo da disciplina).

---

3. Objetivo & Escopo

Objetivo: disponibilizar um monitor de preços simples, confiável e extensível que consolide preços de hortifrutis (atacado e, opcionalmente, varejo), normalizando unidades e produzindo indicadores e alertas.

Escopo do MVP (Fase 1): - Cadastro de produtos e mercados (CEASAs e/ou lojas). - Registro de preços diários no atacado (mín/med/máx) com normalização por kg. - Cálculo de média móvel 7 dias, variação diária e alerta de outliers. - Consultas históricas filtradas por produto, mercado, intervalo. - Exportação JSON e relatório texto. - Persistência no **Oracle** (fonte da verdade) e espelho em **JSON** (backup local).

Escopo Futuro (Fase 2 – Varejo): - Registro de preço ao consumidor por loja/bairro. - Comparativo atacado × varejo (margem aproximada). - Módulo simples de promoções e detecção de quedas >5% em varejo.

Fora de escopo (por ora): - Integrações automáticas com APIs externas. - Interface gráfica (GUI/Web). O front será via CLI. - Modelos de previsão de séries temporais (ARIMA/Prophet).

---

4. Stakeholders & Personas

- **Produtor/Atacadista (Operador)**: registra preços observados/negociados, consulta tendência para definir preço diário.
- **Comerciante varejista (Operador)**: consulta histórico para precificação e promoções; futuramente registra preço de gôndola.
- **Analista (Leitor/Power User)**: cruza séries, exporta relatórios e consolida visões para diretoria.
- **Administrador**: gerencia cadastros, backup e parâmetros (janelas, limites de alerta).

---

5. Visão geral da solução (Como funciona)

1. **Cadastro**: produtos (nome, código, categoria) e mercados (nome, tipo: ATACADO/VAREJO).
2. **Registro**: usuário informa (data, produto, mercado, tipo_preco, unidade_original, preço_original, fonte). O sistema **normaliza** para **kg**.
3. **Cálculo**: para cada (produto, mercado, tipo_preco), calcula **média móvel 7d**, **desvio** e **variação diária**; dispara **alerta** quando preço atual excede limites (z-score > 2) ou queda >5%.
4. **Persistência**: grava em **Oracle** (tabelas MARKETS, PRODUCTS, PRICES) e mantém espelho em JSON + logs em TXT.
5. **Consulta**: o usuário filtra e visualiza tabelas alinhadas, exporta JSON/relatório .txt e observa alertas.

---

6. Definições e Glossário

- **tipo_preco**: ATACADO_MIN, ATACADO_MED, ATACADO_MAX, VAREJO.
- **unidade_original**: forma reportada (ex.: "cx23kg", "kg", "dúzia").
- **preco_kg**: preço normalizado por kg.
- **média móvel 7d**: média simples das últimas 7 observações válidas do par (produto, mercado, tipo).
- **alerta**: indicação de outlier (alta/baixa) conforme regra.

---

7. Regras de negócio (RN)

- **RN1 — Normalização de unidade**: converter preço para **preco_kg** usando tabela de fatores (`unidades.json`). Ex.: `cx23kg → fator 23`.
- **RN2 — Deduplicação**: não aceitar registro duplicado para a mesma **data_ref, produto, mercado, tipo_preco**.
- **RN3 — Variação diária**: `(preco_hoje - preco_ontem) / preco_ontem * 100`, se houver preço no dia anterior.
- **RN4 — Média móvel 7d**: média simples dos últimos **até 7** preços válidos (ignorar nulos/zeros).
- **RN5 — Alerta (outlier)**: alerta **ALTA** se `preco_hoje > media7 + 2*desvio7`; **BAIXA** se `preco_hoje < media7 - 2*desvio7`; ou **QUEDA** se `variacao <= -5%`.

- **RN6 — Validação de entrada**: preço > 0, unidade reconhecida, data válida, produto/mercado existentes.
- **RN7 — Fonte**: cada registro deve armazenar a fonte (CEASA, coleta manual, CSV loja, etc.).

---

8. Requisitos funcionais (RF)

RF01 — Cadastrar produto
Descrição: Permitir incluir/editar/listar produtos com código único e categoria.
Critérios de aceite: - Dado um nome e código ainda não utilizado, quando o usuário salvar, então o produto fica disponível para seleção. - Não permitir duplicar código.

RF02 — Cadastrar mercado
Descrição: Incluir/editar/listar mercados (nome, tipo: ATACADO/VAREJO).
Critérios de aceite: - Apenas valores válidos para tipo.

RF03 — Registrar preço
Descrição: Registrar preço diário com normalização e validações.
Entradas mínimas: data, produto, mercado, tipo_preco, unidade_original, preco_original, fonte.
Saídas: preco_kg calculado e registro persistido.
Critérios de aceite: - Bloquear duplicidade por (data_ref, produto, mercado, tipo_preco). - Recusar preço ≤ 0 e unidade desconhecida.

RF04 — Consultar histórico
Descrição: Filtrar por produto, mercado, tipo_preco e intervalo de datas.
Saída: tabela com Data | Produto | Mercado | Tipo | Preço/kg | Média7 | Var% | Alerta.
Critérios de aceite: - Ordenação por data ascendente e colunas alinhadas.

RF05 — Cálculo de indicadores
Descrição: Calcular média móvel 7d, desvio e variação diária automaticamente.
Critérios de aceite: - Exibir `—` quando não houver histórico suficiente.

RF06 — Alertas
Descrição: Indicar alertas conforme RN5 nas consultas e em relatório textual do dia.
Critérios de aceite: - Alerta mostrado ao lado da linha e em um sumário.

RF07 — Exportação
Descrição: Exportar resultado da consulta em **JSON** e relatório **.txt**.
Critérios de aceite: - Arquivos gerados em `/exports` com timestamp.

RF08 — Logs e auditoria
Descrição: Registrar ações (cadastro, inserção, erro de validação) em `logs/operacoes.txt`.
Critérios de aceite: - Log com data/hora, usuário (quando aplicável), ação e status.

RF09 — Backup/espelho local
Descrição: Manter espelho dos registros em `data/observacoes.json`.
Critérios de aceite: - Falha de Oracle não impede gravação local; sincronizar quando reconectar.

RF10 — Parâmetros
Descrição: Permitir configurar janela (N dias), limites de alerta e fatores de unidade via arquivos JSON.
Critérios de aceite: - Alterações surtindo efeito após reinício da aplicação.

---

9. Requisitos não funcionais (RNF)

RNF01 — Usabilidade (CLI): menus numerados, mensagens claras; colunas alinhadas; exemplos de entrada.

RNF02 — Desempenho: consultas dos últimos 30 dias devem retornar em < 2s com até 10k registros.

RNF03 — Confiabilidade: em caso de falha do Oracle, gravar no espelho JSON e reprocessar inserção na próxima execução.

RNF04 — Segurança: uso de *queries parametrizadas*; credenciais do Oracle via variáveis de ambiente (`.env` não versionado); validação estrita de tipos e faixas; sanitização de texto.

RNF05 — Portabilidade: Python 3.10+; dependências mínimas (oracledb/cx_Oracle, python-dotenv). Não requer GUI.

RNF06 — Manutenibilidade: código modular (camadas: serviços, persistência, validação); testes unitários para funções de regra (≥80% cobertura dessas funções).

RNF07 — Observabilidade: logs com níveis (INFO/WARN/ERROR) e códigos de erro padronizados.

RNF08 — Internacionalização (opcional): mensagens em PT-BR; strings centralizadas para futura i18n.

---

10. Dados & Modelo (Oracle + JSON)

10.1 Modelo lógico (simplificado)

- **MARKETS(market_id, nome, tipo)**
- **PRODUCTS(product_id, codigo, nome, categoria)**
- **PRICES(price_id, data_ref, product_id, market_id, tipo_preco, unidade_orig, preco_orig, preco_kg, fonte)**

Restrições: - `MARKETS.tipo ∈ {ATACADO, VAREJO}` - `PRICES.tipo_preco ∈ {ATACADO_MIN, ATACADO_MED, ATACADO_MAX, VAREJO}` - `preco_orig > 0`, `preco_kg > 0` - **Unique**: `(data_ref, product_id, market_id, tipo_preco)`

10.2 DDL sugerido (Oracle)

```
CREATE TABLE MARKETS (
  market_id     NUMBER GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
  nome          VARCHAR2(100) NOT NULL,
  tipo          VARCHAR2(10)  CHECK (tipo IN ('ATACADO','VAREJO'))
);
```

```
CREATE TABLE PRODUCTS (
  product_id    NUMBER GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
  codigo        VARCHAR2(20) UNIQUE NOT NULL,
  nome          VARCHAR2(100) NOT NULL,
  categoria     VARCHAR2(50)
);

CREATE TABLE PRICES (
  price_id      NUMBER GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
  data_ref      DATE NOT NULL,
  product_id    NUMBER NOT NULL REFERENCES PRODUCTS(product_id),
  market_id     NUMBER NOT NULL REFERENCES MARKETS(market_id),
  tipo_preco    VARCHAR2(15) CHECK (tipo_preco IN
('ATACADO_MIN','ATACADO_MED','ATACADO_MAX','VAREJO')),
  unidade_orig  VARCHAR2(20),
  preco_orig    NUMBER(12,2) CHECK (preco_orig > 0),
  preco_kg      NUMBER(12,4) CHECK (preco_kg  > 0),
  fonte         VARCHAR2(40),
  CONSTRAINT uq_price UNIQUE (data_ref, product_id, market_id, tipo_preco)
);
```

10.3 Arquivos JSON

- `data/produtos.json` — catálogo de produtos (código, nome, categoria).
- `data/mercados.json` — mercados cadastrados (nome, tipo).
- `data/unidades.json` — fatores de conversão (ex.: {"cx23kg": 23, "kg": 1, "duzia": 12}).
- `data/observacoes.json` — espelho de registros PRICES.

---

11. Interface (CLI) & Fluxos

11.1 Menu principal

1. Cadastrar produto
2. Cadastrar mercado
3. Registrar preço
4. Consultar histórico
5. Exportar resultados
6. Parâmetros
7. Sair

11.2 Fluxo — Registrar preço (caso de uso principal)

Ator: Operador
Pré-condição: Produto e mercado cadastrados.
Passos: 1. Informar data (YYYY-MM-DD). 2. Selecionar produto (código) e mercado. 3. Selecionar `tipo_preco`. 4. Informar `unidade_original` e `preco_original`. 5. Informar fonte. 6. Sistema valida, normaliza `preco_kg`, persiste Oracle/JSON e loga ação. 7. Exibe resumo e indicadores (média7 atual, variação vs. ontem, alerta se houver).

Exceções: - E1: Unidade desconhecida → sugerir opções e abortar. - E2: Duplicidade por chave única → perguntar se deseja editar registro existente (Futuro) ou abortar. - E3: Falha Oracle → registrar em `observacoes.json`, logar ERROR e avisar usuário.

---

12. Projeto de Software (Arquitetura)

12.1 Estilo arquitetural

- **Camadas**:
- **CLI (Interface)** → **Serviços (Regras/Calcular/Validar)** → **Persistência (Oracle/Arquivos)**.
- **Módulos Python**:
- `main.py` — orquestra o menu.
- `services/regras.py` — normalização, média móvel, variação, alertas.
- `services/validacao.py` — validações e sanitização.
- `services/arquivos.py` — leitura/escrita JSON e TXT (logs, exports).
- `services/db.py` — conexão e operações Oracle (parametrizadas).
- `models/entidades.py` — dataclasses/tuplas.
- `config/params.py` — carregar parâmetros (janela, limites) de JSON/ENV.

12.2 Diagrama (PlantUML — componente)

```
@startuml
package CLI {
  [main.py]
}
package Services {
  [regras.py]
  [validacao.py]
}
package Persistencia {
  [db.py]
  [arquivos.py]
}
[main.py] --> [regras.py]
[main.py] --> [validacao.py]
[main.py] --> [db.py]
[main.py] --> [arquivos.py]
[regras.py] --> [arquivos.py]
[db.py] --> (Oracle)
[arquivos.py] --> (JSON/TXT)
@enduml
```

12.3 Sequência (Registrar preço)

```
@startuml
actor Operador
Operador -> main.py: inputs (dados do preço)
main.py -> validacao.py: validar_campos()
```

```
validacao.py --> main.py: ok/erro
main.py -> regras.py: normalizar_unidade()
regras.py --> main.py: preco_kg
main.py -> db.py: inserir_preco()
db.py --> main.py: sucesso/erro
alt erro Oracle
  main.py -> arquivos.py: append_observacao_json()
end
main.py -> regras.py: calcular_indicadores()
regras.py --> main.py: media7, var%, alerta
main.py -> arquivos.py: log_operacao()
main.py --> Operador: resumo + indicadores
@enduml
```

12.4 Subalgoritmos (assinaturas)

```
def normalizar_unidade(preco: float, unidade: str, fatores: dict) ->
float: ...

def media_movel(valores: list[float], n: int = 7) -> float | None: ...

def desvio_padrao(valores: list[float], n: int = 7) -> float | None: ...

def variacao_percentual(hoje: float, ontem: float) -> float | None: ...

def detectar_alerta(preco_hoje: float, serie: list[float]) -> dict: ...

def ler_json(caminho: str) -> dict | list: ...

def escrever_json(caminho: str, dados: dict | list) -> None: ...

def log_txt(msg: str, nivel: str = "INFO") -> None: ...
```

12.5 Estruturas de dados (exemplos)

```
Produto = {"codigo": str, "nome": str, "categoria": str}
Mercado = {"nome": str, "tipo": "ATACADO"|"VAREJO"}
Observacao = {
  "data": "YYYY-MM-DD",
  "produto": "TOM_LV",
  "mercado": "CEASA-GO",
  "tipo_preco": "ATACADO_MED",
  "unidade_original": "cx23kg",
  "preco_original": 120.00,
  "preco_kg": 5.22,
  "fonte": "CEASA"
}
```

12.6 Conexão Oracle (boas práticas)

- Driver `oracledb` (ou `cx_Oracle`).
- Credenciais via variáveis de ambiente (`DB_USER`, `DB_PASS`, `DB_DSN`).
- *Queries* parametrizadas; *commit* explícito; tratamento de exceções; *retry* simples.

---

13. Testes & Qualidade

- **Teste de unidade** (pytest) para: `normalizar_unidade`, `media_movel`, `desvio_padrao`, `variacao_percentual`, `detectar_alerta`, `validar_campos`.
- **Teste de integração**: inserção e consulta Oracle (com base de teste).
- **Teste de aceitação (UAT)**: roteiros "cadastrar produto/mercado → registrar preço → consultar histórico → exportar".

Critérios de sucesso (exemplos): - 100% das validações rejeitam entradas inválidas simuladas. - Consultas dos últimos 30 dias retornam < 2s em 10k linhas. - Alertas coerentes com cenários sintéticos (picos/quedas).

---

14. Métricas de sucesso (negócio)

- Redução de erros de entrada (medida por logs de validação) $\geq$ 80% após 2 semanas.
- Aumento de uso de histórico e exportações (indicando adoção) semana a semana.
- Decisões de preço documentadas em relatório diário (qualitativo).

---

15. Riscos & Mitigações

- **Dados incompletos/inconsistentes** → validações, fontes declaradas e logs.
- **Mudanças de unidade** → tabela de fatores externa (`unidades.json`).
- **Indisponibilidade do Oracle** → espelho JSON e reprocessamento posterior.
- **Entrada duplicada** → chave única + checagem prévia.

---

16. Roadmap

- **Fase 1 (MVP)**: Atacado completo (cadastros, registro, indicadores, alertas, exportação, Oracle/JSON).
- **Fase 2**: Varejo + comparativos e alertas de promoções (>5% queda).
- **Fase 3**: Importações CSV, previsão simples (opcional), interface web.

---

17. Anexos

17.1 Exemplo de entrada (registro)

```
Data: 2025-10-07
Produto (código): TOM_LV
Mercado: CEASA-GO
Tipo: ATACADO_MED
Unidade: cx23kg
Preço: 120.00
Fonte: CEASA
```

17.2 Exemplo de saída (consulta)

```
Data       Produto  Mercado   Tipo          Preço/kg  Média7  Var%    Alerta
2025-10-01 TOM_LV   CEASA-GO  ATACADO_MED   5.10      5.05    —       —
2025-10-02 TOM_LV   CEASA-GO  ATACADO_MED   5.08      5.06    -0.39%  —
...
2025-10-07 TOM_LV   CEASA-GO  ATACADO_MED   5.22      5.12    +1.16%
ALTA(z=2.1)
```

17.3 Pseudocódigo — Registrar preço

```
ler entradas
validar_campos()
preco_kg = normalizar_unidade(preco_original, unidade)
try:
  inserir_preco_oracle(..., preco_kg)
except OracleError:
  append_observacao_json(..., preco_kg)
serie = consultar_ultimos_n_precos(..., n=7)
media, desvio = calc_media_desvio(serie)
var = variacao(hoje, ontem)
alerta = detectar_alerta(hoje, serie)
mostrar_resumo(media, var, alerta)
log_txt("preco registrado", INFO)
```

— FIM —