Product Requirements Document (PRD) – AgroSolutions 8NETT

Versão: 3.0 (Code First & Phased Approach) **Contexto:** Hackathon 8NETT – MVP Local **Arquitetura:** Clean Architecture + DDD + CQRS **Infraestrutura:** Docker Local (Dev) -> Kubernetes (Final)

---

1. Visão Geral

Construir uma plataforma MVP de Agricultura 4.0 focada na ingestão de telemetria e análise de dados para prevenção de seca. **Filosofia do Projeto:** "Simples, mas Profissional".

- **Simples:** Execução 100% local, foco no MVP.
- **Profissional:** Uso de padrões de mercado (Clean Arch, CQRS com MediatR, Code First, Testes e Containers).

---

2. Stack Tecnológica

- **Linguagem:** C# .NET 8.
- **Estilo Arquitetural:** Clean Architecture com DDD (Domain-Driven Design).
- **Comunicação Interna:** CQRS (Command Query Responsibility Segregation) utilizando biblioteca **MediatR**.
- **Banco de Dados:** PostgreSQL (Gerenciado via **EF Core Code First**).
- **Mensageria:** RabbitMQ.
- **Workers:** Azure Functions (Core Tools).
- **Observabilidade:** Prometheus & Grafana.

---

3. Estrutura do Repositório

Para garantir organização e facilidade no setup do Docker:

Plaintext
/AgroSolutions
│

```
├── /src                    # Código Fonte (.NET)
│   ├── /Agro.Domain            # Entidades e Regras de Negócio (Puro)
│   ├── /Agro.Application       # Casos de Uso, CQRS (Commands/Queries), DTOs
│   ├── /Agro.Infra             # EF Core Context, Repositories, RabbitMQ Service
│   ├── /Agro.API               # Controllers (Entrada HTTP)
│   └── /Agro.Worker            # Azure Functions (Triggers)
│
├── /infra                  # "A Pasta Mágica" do Docker
│   ├── /docker-compose         # Arquivos para subir dependências rápido (Dev)
│   │   └── dependencies.yaml   # Sobe Postgres + RabbitMQ + Grafana
│   └── /k8s                    # Manifestos finais para o Minikube (Prod Simulado)
│
└── README.md                   # Deve conter a tabela de "Comandos Rápidos"
```

---

4. Plano de Execução em Fases (Incremental)

Cada fase deve ser "testável" e agregar valor, garantindo que a base não quebre a próxima etapa.

FASE 1: O Core do Domínio (Identity & Propriedades)

Objetivo: Estabelecer a arquitetura, o banco de dados (Code First) e os cadastros base.
Features:

- Setup da Solution (Clean Arch).
- Configuração do EF Core + PostgreSQL.
- Auth: Login (JWT).
- Cadastro de Fazenda e Talhões.
- **Testes:** Unitários de Domínio + Teste Manual via Swagger.

Critério de Aceite (Definition of Done):

1. Rodar `docker compose -f infra/docker-compose/dependencies.yaml up -d` sobe o banco.
2. A API roda e aplica a *Migration* automaticamente na inicialização.
3. Consigo criar um usuário e usar o token para cadastrar uma fazenda.

FASE 2: Ingestão de Alta Performance

Objetivo: Receber dados sem gargalo. Implementar o padrão "Fire and Forget" com RabbitMQ.
Features:

- Endpoint `POST /telemetry`.
- Configuração do RabbitMQ na Infra.
- Implementação do *Publisher* na API.
- **Regra:** Este endpoint **NÃO** salva no banco. Apenas valida o JSON e publica na fila `telemetry-queue`.

Critério de Aceite:

1. Enviar um POST simulando um sensor.
2. O request retorna `202 Accepted` em milissegundos.
3. Acessar o painel do RabbitMQ (`localhost:15672`) e ver a mensagem parada na fila aguardando.

FASE 3: Workers & Inteligência (Azure Functions)

Objetivo: Processar os dados assincronamente e aplicar as regras de negócio. **Features:**

- Projeto Azure Functions (integrado ao DI do .NET).
- **Function 1 (QueueTrigger):** Consome a fila e grava no PostgreSQL (Tabela `Telemetry`).
- **Function 2 (TimerTrigger - 1h):** Analisa as últimas 24h e gera `Alerts` se (Umidade < 30%).
- **Function 3 (TimerTrigger - 00:00):** Limpa/Inativa alertas do dia anterior.

Critério de Aceite:

1. Ao ligar a Function, a fila do RabbitMQ esvazia.
2. O dado aparece na tabela do PostgreSQL.
3. Forçar o Timer (via HTTP request local) gera um alerta na tabela `Alerts` se a condição for atendida.

FASE 4: Observabilidade & Entrega Final

Objetivo: Visualização (Dashboard) e Empacotamento (Kubernetes). **Features:**

- Dashboards no Grafana lendo do PostgreSQL.
- Dockerfiles para API e Worker.
- Manifestos Kubernetes (Deployment/Service).

Critério de Aceite:

1. Dashboard mostra gráfico de linha (histórico) e cards de alerta.
2. Vídeo de demonstração gravado com tudo rodando no Minikube.

---

5. Requisitos Funcionais (Consolidado)

5.1. Módulo de Acesso (API)

- [RF-01] Autenticação de Produtor (Email/Senha) retornando JWT.
- [RF-02] Cadastro de Propriedade (Nome, Tamanho).
- [RF-03] Cadastro de Talhão (Nome, Cultura, Área).

5.2. Módulo de Sensores (API + Rabbit)

- [RF-04] Endpoint de recebimento de telemetria (Umidade, Temp, Chuva).
- [RF-05] Enfileiramento seguro em Message Broker.

5.3. Processamento (Functions)

- [RF-06] Persistência assíncrona da telemetria.
- [RF-07] Motor de Regras: Validar regra de seca (30% umidade/24h).
- [RF-08] Rotina de Limpeza Diária (Reset de alertas).

5.4. Visualização (Grafana)

- [RF-09] Gráfico histórico de leituras.
- [RF-10] Indicador visual de "Alerta Ativo".

---

6. Modelo de Dados (Entidades de Domínio)

Como usaremos **Code First**, estas são as classes de Domínio esperadas:

- **User:** `Guid Id`, `string Name`, `string Email`, `string PasswordHash`.
- **Farm:** `Guid Id`, `Guid UserId`, `string Name`.
- **FieldPlot (Talhão):** `Guid Id`, `Guid FarmId`, `string Name`, `string CropType` (Enum ou String).
- **Telemetry:** `Guid Id`, `Guid FieldPlotId`, `DateTime Timestamp`, `decimal SoilMoisture`, `decimal Temperature`, `decimal Precipitation`.
- **Alert:** `Guid Id`, `Guid FieldPlotId`, `AlertType Type` (Seca/Praga), `DateTime CreatedAt`, `bool IsActive`.