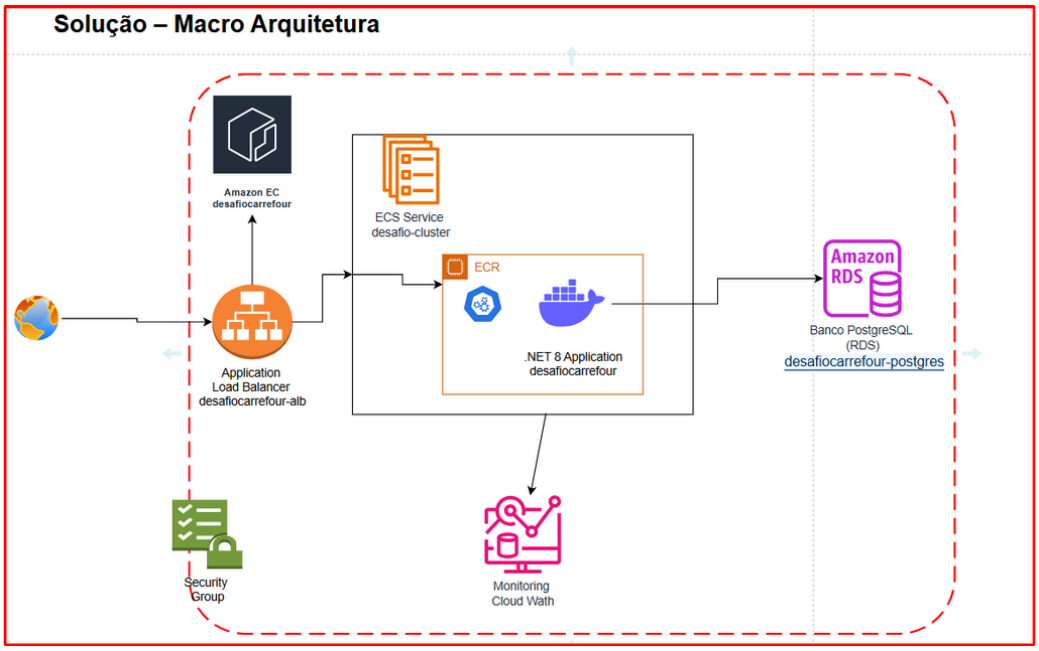
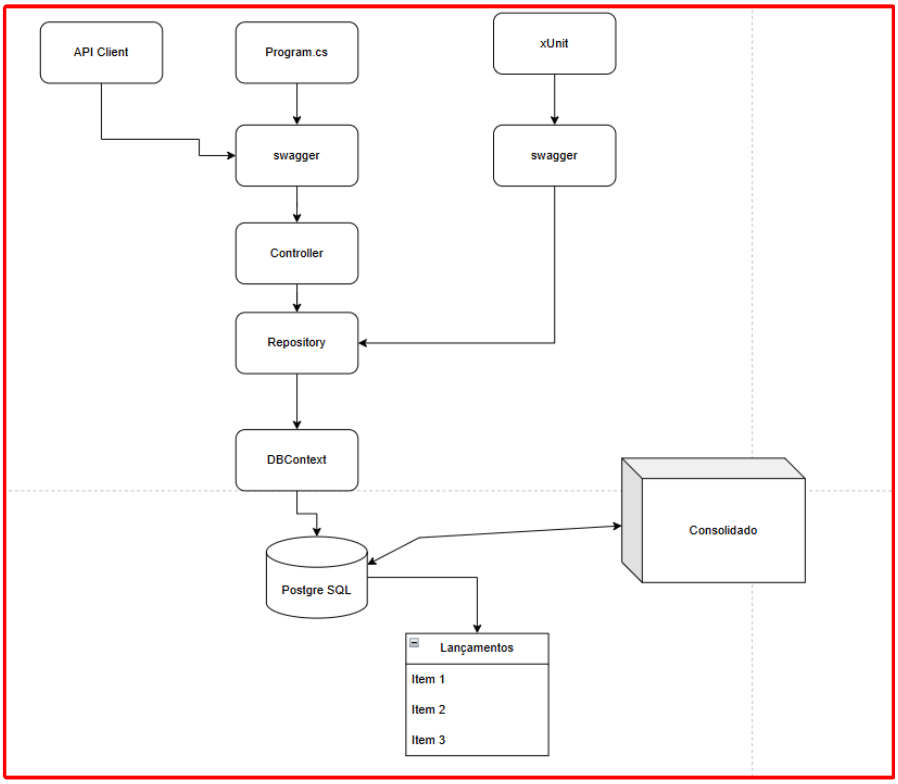
Funcionalidades da Aplicação

Arquitetura



Compentes Internos



Funcionalidades da Aplicação

- `POST /Lancamentos`: Criação de lançamentos de valor
- `GET /Consolidado?data=YYYY-MM-DD`: Consulta de saldo em uma data

Infraestrutura Criada com Terraform

- Repositório ECR para imagens Docker
- Cluster ECS + Fargate + Task Definitions
- Load Balancer com listener público (porta 80)
- Serviço ECS com acesso ao banco
- Instância RDS PostgreSQL
- Security Groups interligados (ECS ↔ RDS)

Entregáveis

- `application/`: Código-fonte da aplicação
- `terraform/`: Provisionamento completo da AWS
- `Dockerfile`, `docker-compose.yml`: Empacotamento e execução local
- `README.md`: Instruções de build, deploy e acesso
- `README_EXECUCAO_LOCAL.md`: Guia para execução local
- `checklist.pdf`: Validação de requisitos implementados
- `entrega_desafio_carrefour_v2.zip`: Pacote final organizado

Requisitos Cumpridos

✅ Aplicação ASP.NET Core, an open-source web development framework | .NET Core (.NET 8)
✅ Banco de Dados PostgreSQL na AWS (RDS)
✅ Dockerfile e Docker Compose
✅ Imagem publicada no ECR
✅ Infraestrutura automatizada com Terraform
✅ Execução via ECS Fargate
✅ Load Balancer público com acesso ao Swagger
✅ Scripts SQL de criação da tabela
✅ API funcional testada via Swagger e cURL
✅ README com instruções completas
✅ Task ECS configurada com string de conexão do RDS

Proposta de Melhoria Arquitetural (Evolução)

Embora o desafio tenha sido entregue com sucesso em uma arquitetura monolítica orientada a serviços, para uma futura evolução em produção ou escala, consideramos a seguinte proposta:

1. Evoluir para Arquitetura de Microserviços

- Separar o domínio de **Lançamentos** e **Consolidado** em serviços independentes
- Cada serviço com seu banco, lógica e API isolada

- Melhorar escalabilidade, deploys independentes e segurança por domínio

2. Aplicar Arquitetura Hexagonal (Ports & Adapters)

- Permitir que a lógica de negócio fique desacoplada de frameworks e infraestrutura
- Facilitar testes unitários, mocks e substituição de tecnologias com baixo acoplamento
- Exemplo: `LancamentoService` isolado de `DbContext`, `Http`, etc.

3. Utilizar Mensageria com RabbitMQ + MassTransit

- Comunicação assíncrona entre serviços
- Exemplo: serviço de lançamento emite evento `LancamentoCriado`, serviço de consolidação reage
- Garante resiliência, escalabilidade e desacoplamento