FluxoCaixa – Microservices
Trade-offs das tecnologias e padrões utilizados

A solução adotada envolve decisões sobre arquitetura, mensageria, persistência, comunicação assíncrona, tolerância a falhas e padrões de projeto.

A seguir, apresento cada escolha, seus benefícios e seus trade-offs, i. e., as vantagens e os custos oulimitações assumidos.

---

1. Arquitetura em Microsserviços

Benefícios

- Desacoplamento real entre Lançamentos e Consolidação.
- Cada serviço pode evoluir, escalar e ser deployado independentemente.
- Maior confiabilidade: se um serviço cai, o outro continua funcional.
- Organização clara por domínios funcionais (DDD básico).

Trade-offs

- Aumenta a complexidade operacional.
- Demanda observabilidade .
- Necessário orquestrar múltiplas aplicações.
- Exige mensageria para comunicação.

Justificativa: A separação entre escrita (TransactionsService) e leitura/relatório (ConsolidatedService) é uma divisão que reduz acoplamento e aumenta resiliência.

2. Comunicação Assíncrona com RabbitMQ

Benefícios

- Desacopla completamente o fluxo de lançamento do consolidado.
- Garante que o ConsolidatedService continue funcionando mesmo que o TransactionsService esteja sobrecarregado.
- Evita sobrecarga de requisições.

Trade-offs

- Introduz uma infraestrutura adicional (RabbitMQ).
- Exige entendimento de conceitos como específicos.
- Necessário lidar com cenários de reprocessamento e mensagens duplicadas, se for o caso.

Justificativa:

RabbitMQ é leve, estável, simples de configurar e fornece garantias fortes para fila de eventos. É perfeito para tráfego médio (cerca de 50 eventos por segundo, no pico).

3. Banco de Dados SQLite no ConsolidatedService

Benefícios

- Extremamente leve.
- Para um consolidado diário, o volume de dados é pequeno.
- Não depende de servidor externo e evita complexidade adicional.
- Fácil de portar e versionar (arquivo .db).

Trade-offs

- Não ideal para volume alto ou concorrência pesada.
- Não é adequado para cargas de trabalho distribuídas e simultâneas.

Justificativa:

O serviço de consolidado tem foco em consultas diárias, baixo volume e persistência simples, então o SQLite acaba sendo simples e eficiente.

4. IHostedService e BackgroundService para consumo assíncrono

Benefícios

- Permite consumo contínuo sem bloquear requisições.
- Mantém o consumer separado do pipeline HTTP.
- Simples e nativo do .NET, por isso é fácil de manter.
- Evita acoplamento entre API e mensageria.

Trade-offs

- BackgroundService é Singleton, exige cuidado ao usar serviços “scoped”.
- Sem isolamento por processo, tudo é executado na mesma aplicação.

Justificativa:

Para consumir eventos de RabbitMQ de forma simples e resiliente, o padrão BackgroundService eficaz.

5. Uso de Circuit Breaker (via Polly) para Resiliência

Benefícios

- Protege o TransactionsService caso o ConsolidatedService falhe.
- Evita excesso de requisições para um serviço fora do ar.
- Melhora experiência do usuário com respostas mais previsíveis.

Trade-offs

- Torna o fluxo mais complexo e exige monitoramento dos estados (Open, Half-Open, Closed).
- Pode ocultar erros reais se configurado incorretamente.
- Requer tuning fino de thresholds.

Justificativa:

O requisito exige que o TransactionsService **não fique disponível** se o ConsolidatedService cair.

O Circuit Breaker cumpre esse papel.

6. DDD para organização do código

Benefícios

- Maior clareza entre domínio, aplicação e infraestrutura.
- Facilita manutenção a longo prazo.
- Evita que controllers fiquem extensos.
- Incentiva baixo acoplamento entre camadas.

Trade-offs

- Sobrecarrega projetos pequenos com estrutura extra.
- Exige disciplina de equipe para seguir padrões.

Justificativa:

Mesmo em soluções simples, separar responsabilidades evita que a aplicação cresça de forma desorganizada.

7. Swagger para documentação e validação das APIs

Benefícios

- Rápido de implementar.
- Útil para testar serviços individualmente.
- Permite visualização rápida dos endpoints.

Trade-offs

- Não cobre casos assíncronos (eventos RabbitMQ não aparecem).
- Pode ser confuso quando há múltiplas APIs em portas diferentes.

Justificativa:

Para um desafio técnico ou ambiente local, Swagger atende perfeitamente.

8. HTTP + JSON como interface de entrada

Benefícios

- Padrão universal e fácil de testar (Postman, Swagger, curl).
- Permite desacoplamento entre clientes.

Trade-offs

- Nem sempre é o mais performático.

Justificativa:

Simplicidade acarreta em melhor performance para este tipo de serviço.

9. Testes Unitários para Domínio e Serviços

Benefícios

- Garante consistência das regras de negócio.
- Evita regressões em cálculos do consolidado.
- Aumenta confiabilidade do domínio.

Trade-offs

- Adiciona manutenção ao projeto.
- Não cobre fluxo assíncrono sem testes de integração.

Justificativa:

O requisito exige testes, e testes no domínio são a entrega mais estável com melhor custo/benefício.