






SAGA Pattern além do
carrinho de compras

-  Paulistano, Casado, pai do Théo e da Helena
-  Análise e Desenvolvimento de Sistemas/Engenharia de Software
-  Arquiteto de Soluções no MB | Mercado Bitcoin



2TM

MB

Maior plataforma de **Criptoeconomia** da América Latina com mais de 3,6 milhões de clientes

Ecossistema de soluções

MB Private

Assessoria especializada para empresas, instituições e clientes de alta renda

MB Pay

Instituição de pagamentos

MB One

Canal exclusivo de atendimento, soluções e produtos

MB Custody

Custódia digital para contas institucionais

MB Traders

Plataforma, educação e time de especialistas traders

Blockchain Academy

Plataforma de educação B2B e B2C

MB NFT

Um marketplace para comprar e colecionar NFTs exclusivos

MB Cloud

Solução para implementação de infraestrutura de compra, venda e hold de criptoativos

MB Asset

Gestão e Planejamento de investimentos e patrimônio

Portal do Bitcoin

Plataforma de conteúdo

MB Tokens

Tokenização de ativos reais em ativos digitais

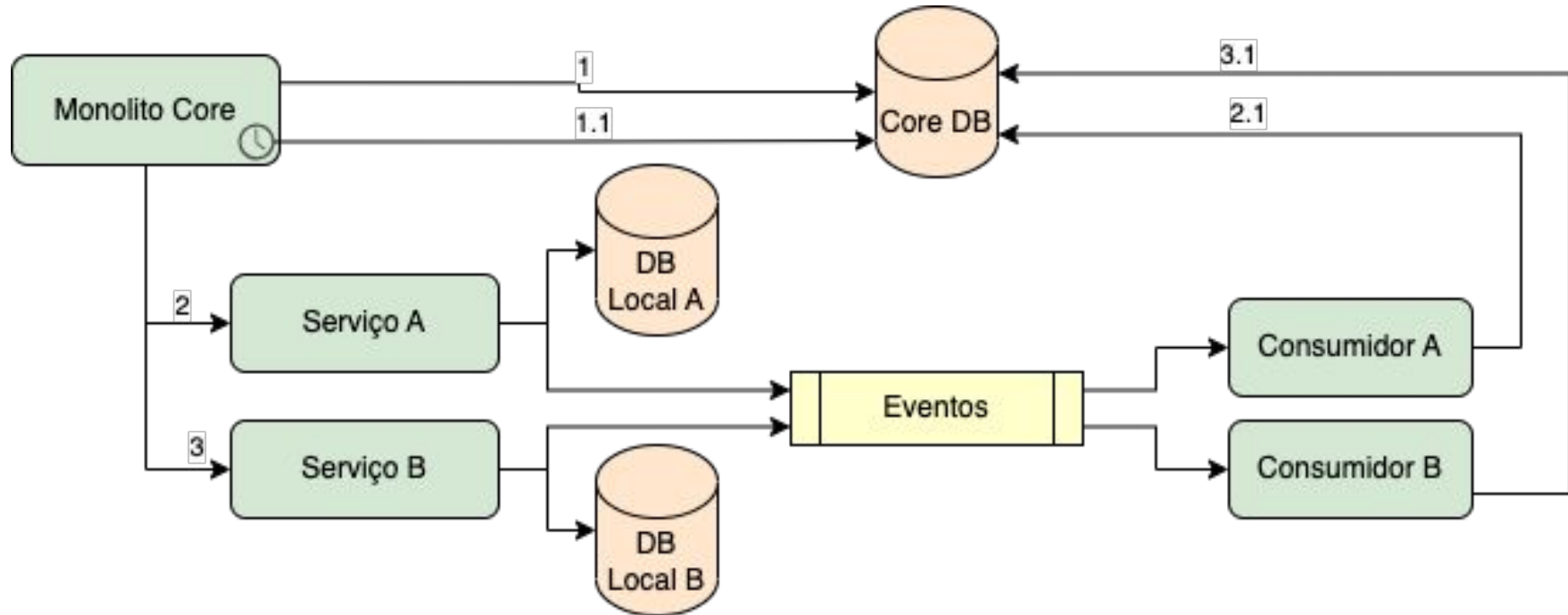
Canais e soluções Cripto, Trader & Tokens

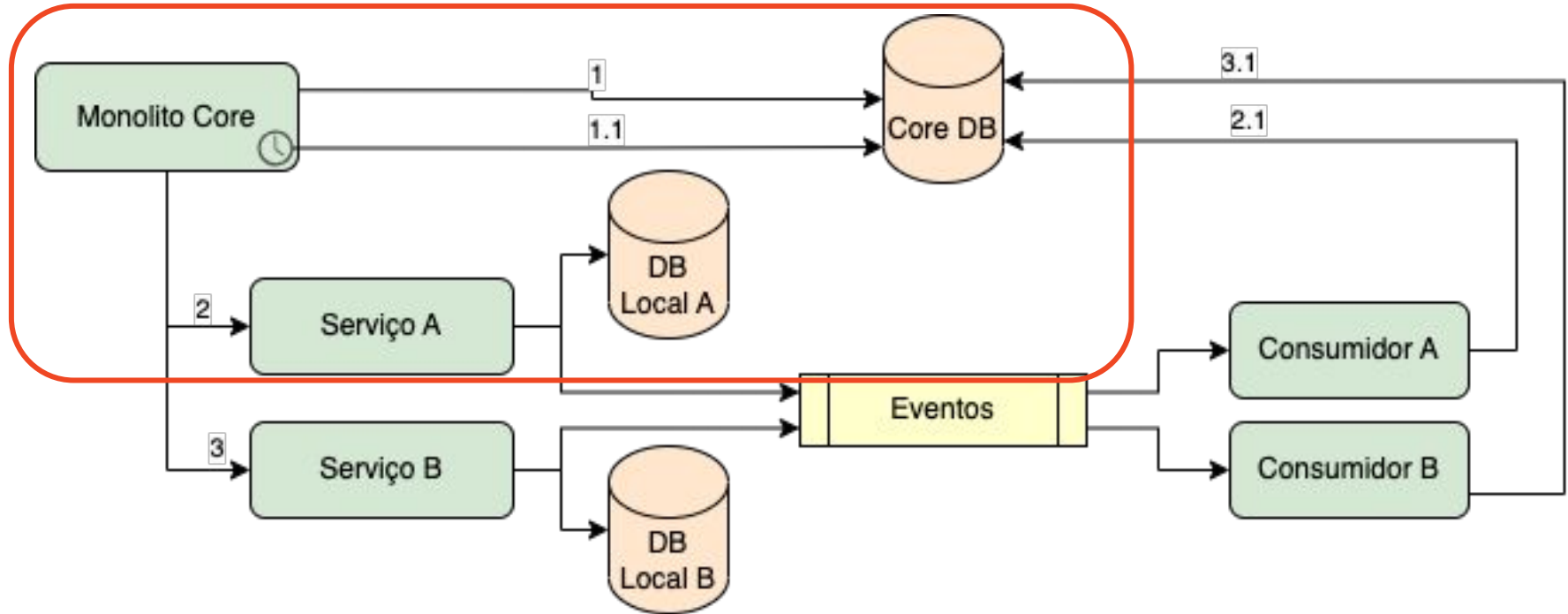
Infraestrutura Cripto e Educacional

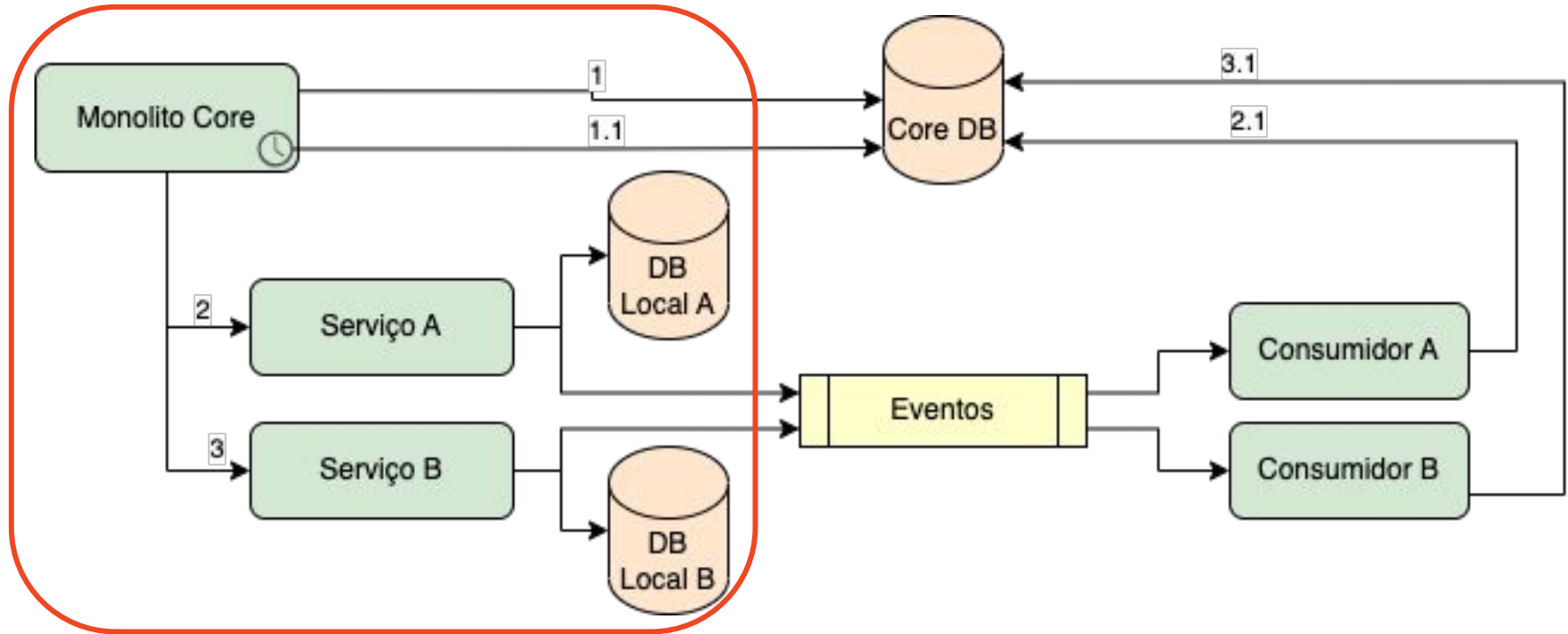


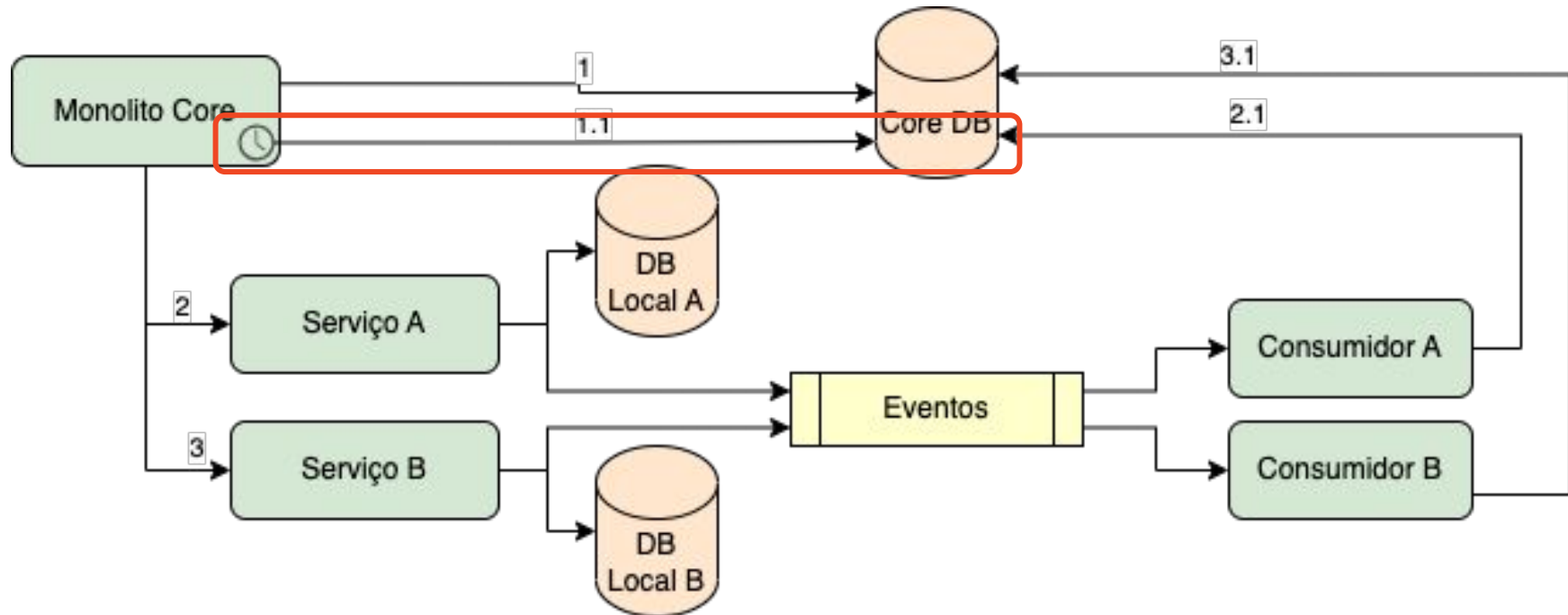
- Primeira empresa do grupo
- Sistema Core Monolítico
- Sobrecarga de funções de negócio: Corretora + Bolsa
- Hiper crescimento em 2021

Temos ótimos locais que não se comunicam globalmente.



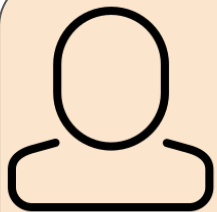








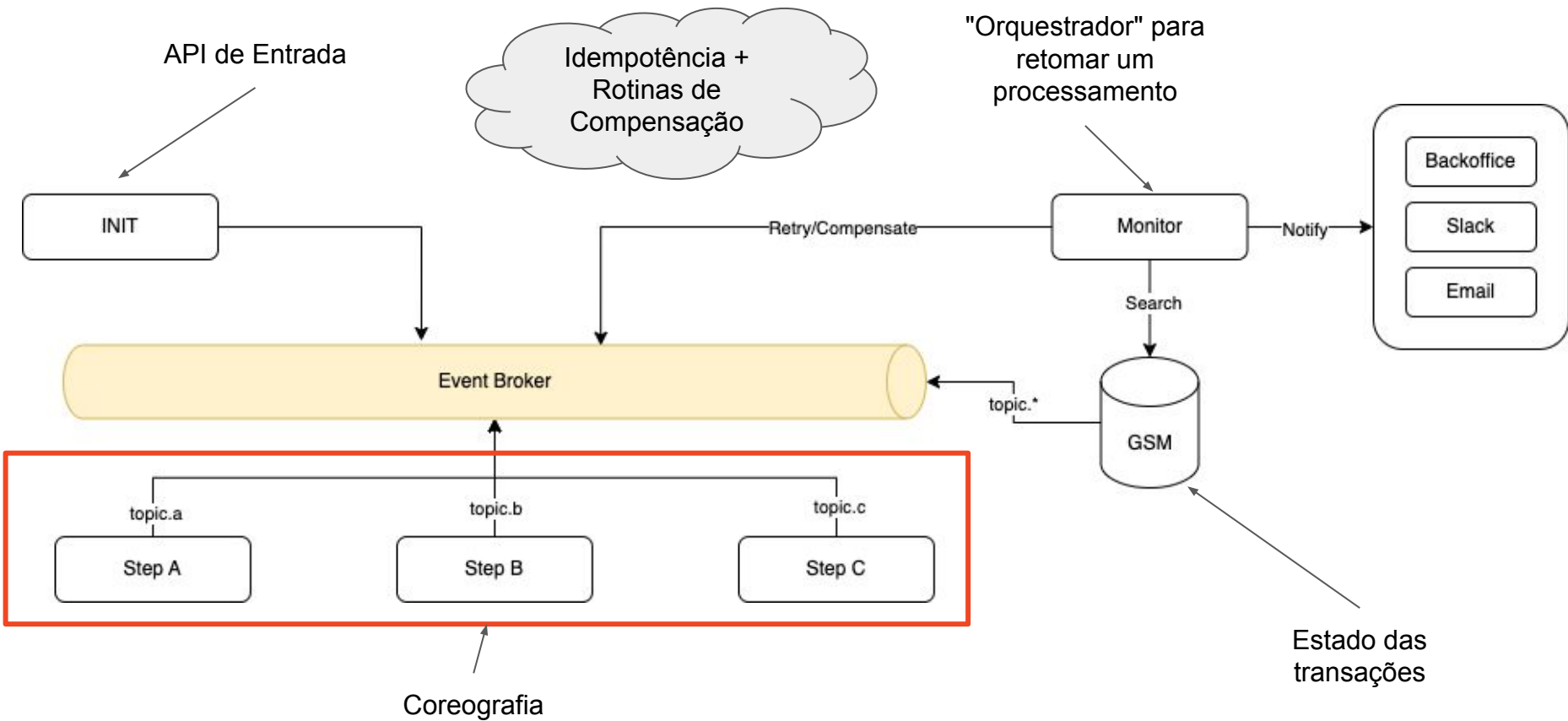
CTO: Temos um problema de gestão de estado global, deveríamos usar SAGA para tratar disso.



Eu: Cara, todo material de vídeo, apresentação ou livro desse negócio usa carrinho de compras de exemplo, aqui temos clientes esperando comportamento síncrono com menos de 100ms de latência, carrinho de compras é assíncrono com TTL alto pra resposta de cartão ou boleto.



CTO: Eu sugeri um padrão de mercado para gerenciar transações distribuídas; vocês são especialistas no nosso negócio e devem dizer se ele atende ou qual a solução equivalente.

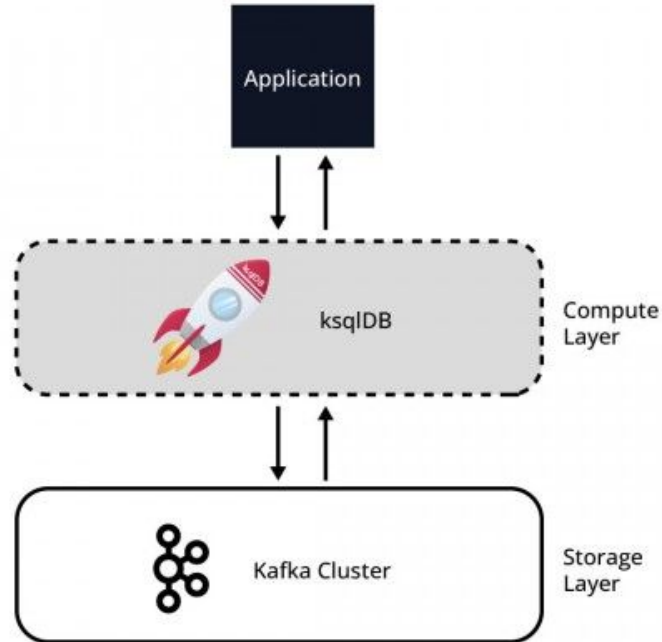


UC 1.1

1. Validar dados de entrada;
2. Persistir base core;
3. **Executar lógica Serviço A;**
4. Execução lógica Serviço B;

UC 1.2

1. Validar dados de entrada;
2. Persistir base core;
3. Execução lógica Serviço B;



<https://blog.ordix.de/ksqldb-the-superpower-in-the-kafka-universe-part-2>

- **in:** Progressão do fluxo;
- **compensate:** Fluxo de rollback;
- **error:** Dead Letter Queue;

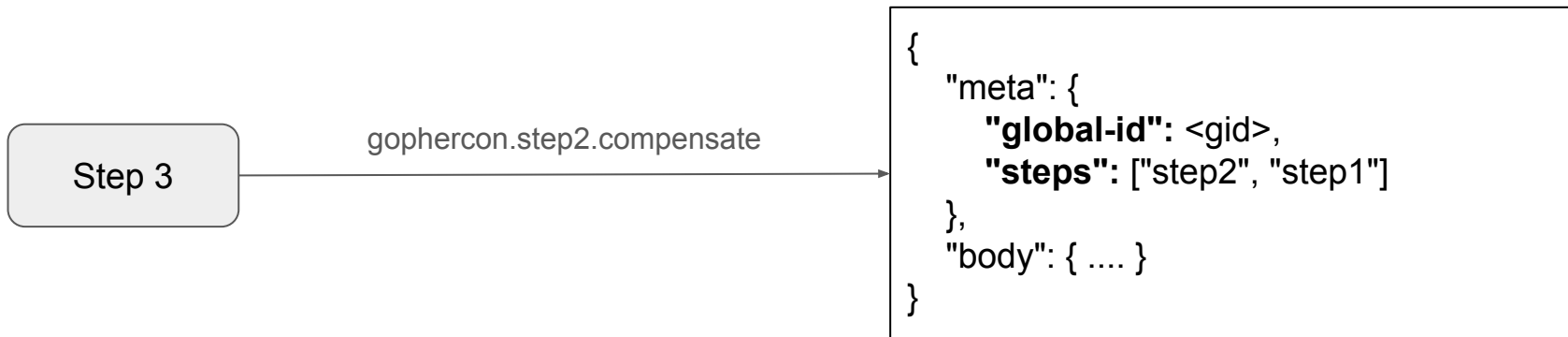
dominio.subdominio.microservico.tipo

INIT

gophercon.step1.in

```
{
  "meta": {
    "global-id": <gid>,
    "steps": ["step1", "step2", "step3"]
  },
  "body": { .... }
}
```

Imagine que o Step 3 teve um erro de negócio irreversível, neste caso ele mesmo pode inverter a lista de passos e publicar a mensagem no tópico **.compensate** do passo anterior.

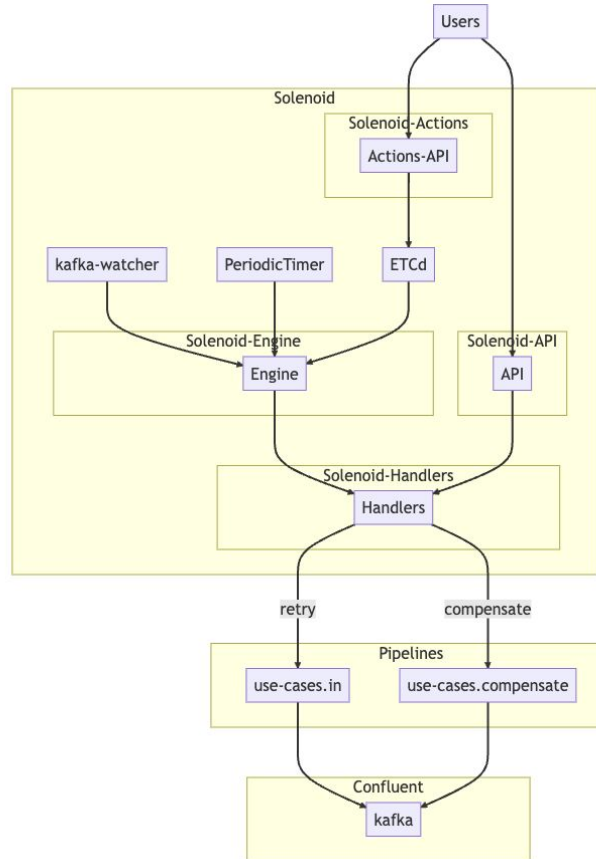


Muito bonito tudo isso mas, cadê o Golang?

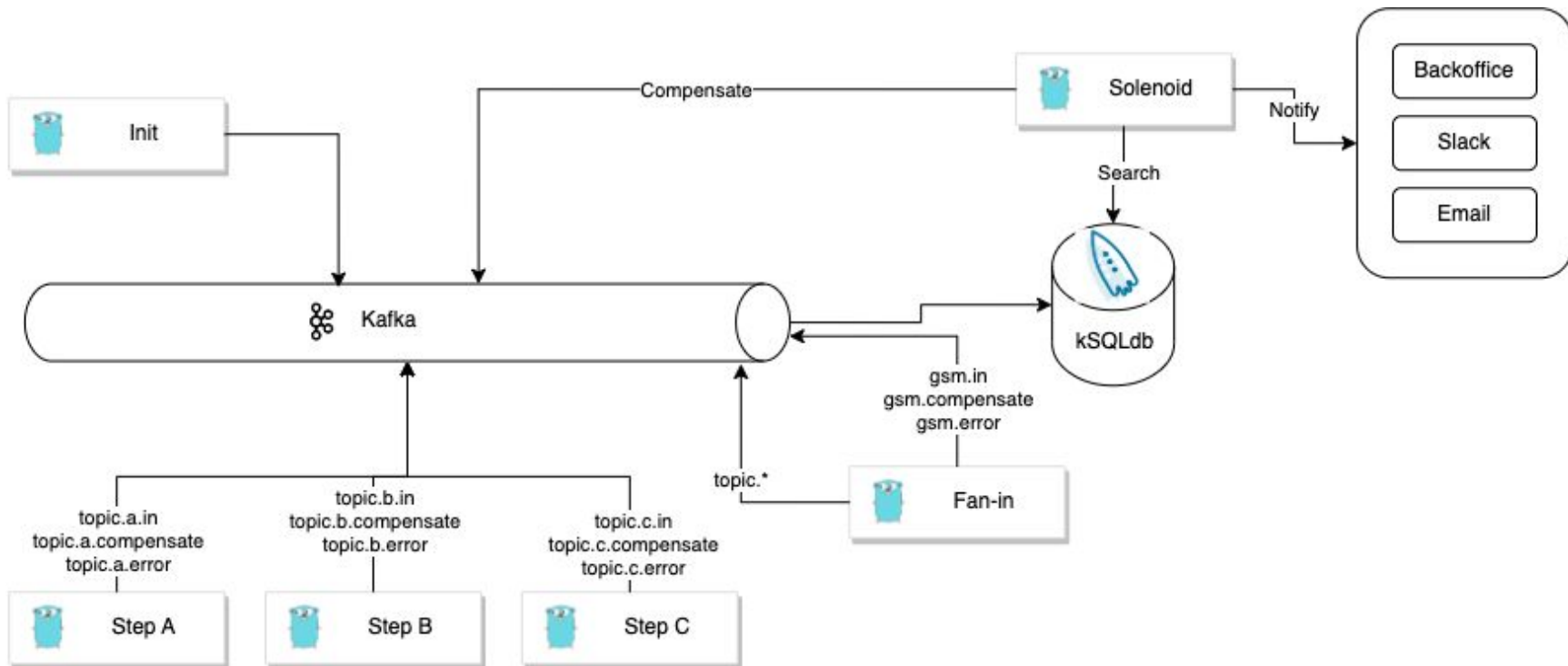
- Uso de FX para ingestão de dependência
- Configuração padronizada
- Consumo dos tópicos *.in* e *.compensate* do passo corrente
- Gerenciamento das mensagens:
 - input | compensate -> error | output
 - parser
 - global-id
- Publicação no tópico seguinte da cadeia

```
12 type FirstStep struct {
13     fx.In
14 }
15
16 func (f *FirstStep) Handle(ctx context.Context, msg broker.Message) (any, error) {
17     var d share.Data
18     if err := msg.Unmarshal(&d); err != nil {
19         return nil, err
20     }
21     return d, nil
22 }
23
24 func (f *FirstStep) Compensate(ctx context.Context, msg broker.Message) (any, error) {
25     var d share.Data
26     if err := msg.Unmarshal(&d); err != nil {
27         return nil, err
28     }
29     return d, nil
30 }
31
32 func main() {
33     app := fx.New(
34         use.CompensatableStep(share.FirstName, &FirstStep{}), // define compensatable step
35     )
36
37     app.Run() // run application
38 }
```

- Interface para responder o estado atual de uma transação
- Agregador de erros para triagem manual
- Dispara rotinas de compensação ou retentativas caso os próprios microsserviços não o façam



Roda pra ver o erro!

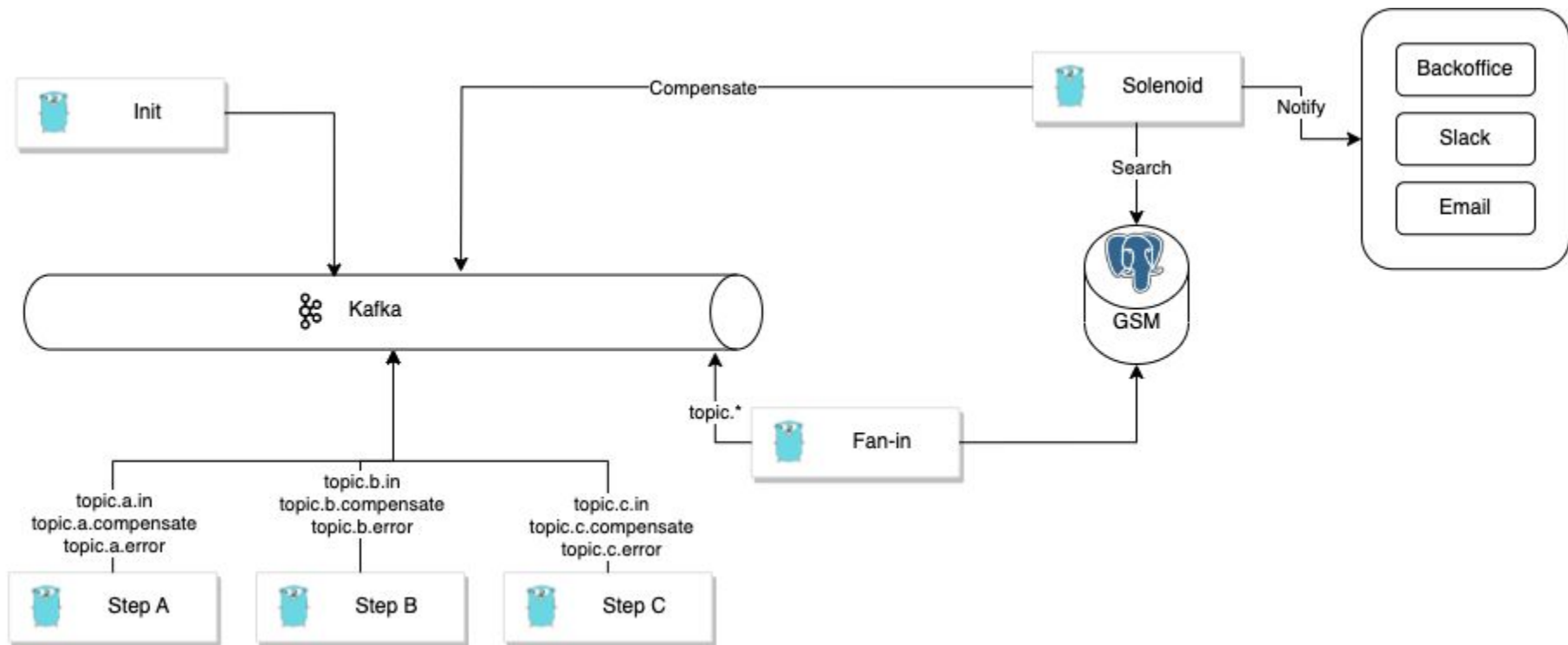


Criamos um Fan-In para contornar o limite de query tables no Confluent Cloud

- Degradação para fazer as consultas que precisávamos desde o início
 - Me dê todos os itens com erro no caso de uso ABC;
 - Me dê o estado da transação XYZ;
- Oportunidade para discutir gestão da base do GSM e retenção das mensagens

"O principal motivador para usar o kSQL era não precisar criar um microsserviço que ouvisse todos os tópicos, e foi justamente o que fizemos no final"

O que realmente foi pro ar



- Zero erros de parada de inconsistência entre as bases envolvidas nos fluxos críticos de compra e venda
- Inversão de dependência com o monolito (código) ao criar um passo específico para manter a compatibilidade dos dados na base core
- Redução do tempo médio das operações em ~50% nas operações com sucesso comparado com a arquitetura legada
- Campanha para que clientes críticos migrem suas integrações para nova API
- Padrão para gerenciar transações distribuídas em novos produtos

1. Governança dos Casos de Uso;
2. Retry automático nos steps;
3. Disparar rotinas de compensação por stream do kafka;
4. Suporte à RabbitMQ e/ou outros brokers;
5. Pipes Framework para Python e Java;
6. Migração de transações distribuídas legadas para o modelo GSM.

Muito
obrigado.

