Engenharia de Softwares Escaláveis

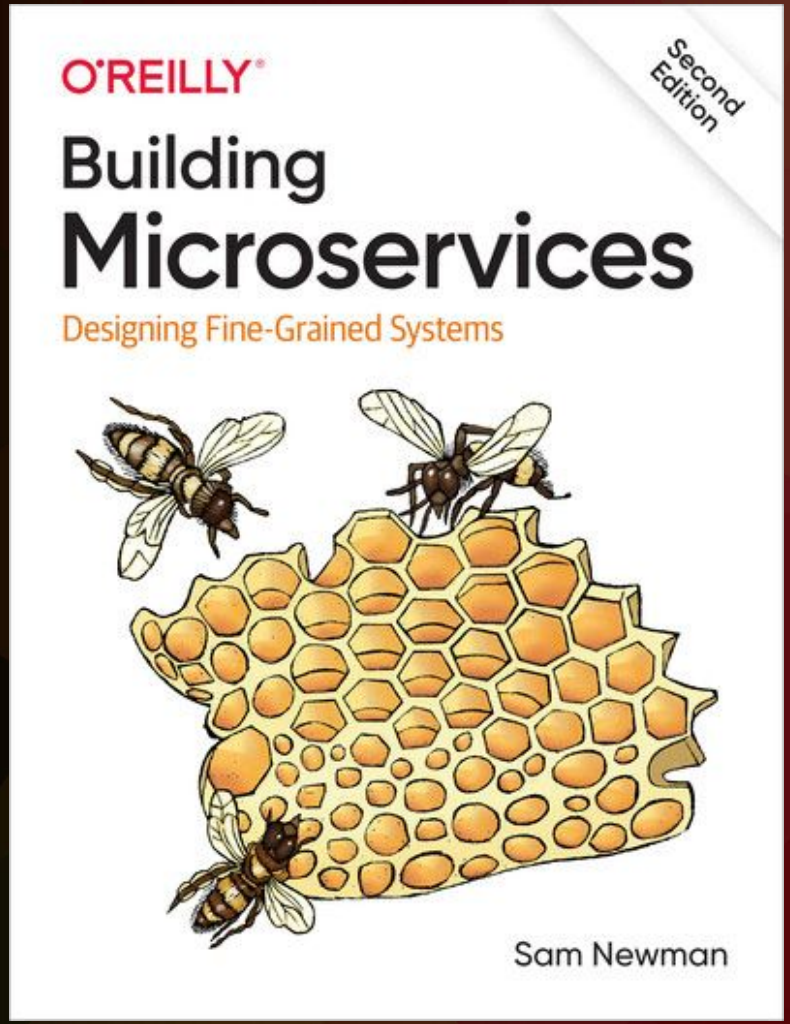
Domain-Driven Design (DDD) e Arquitetura de Softwares Escaláveis com Java

Agenda

Etapa 7: Consistência de Dados em Microsserviços com Java.

- Transações e Transações Compensatórias.
- Saga Design Pattern.
- Implementação de Saga.

Transações e Transações Compensatórias



Capítulo 6: Workflows

https://learning.oreilly.com/library/view/building-microservices-2nd/9781492034018/ch06.html

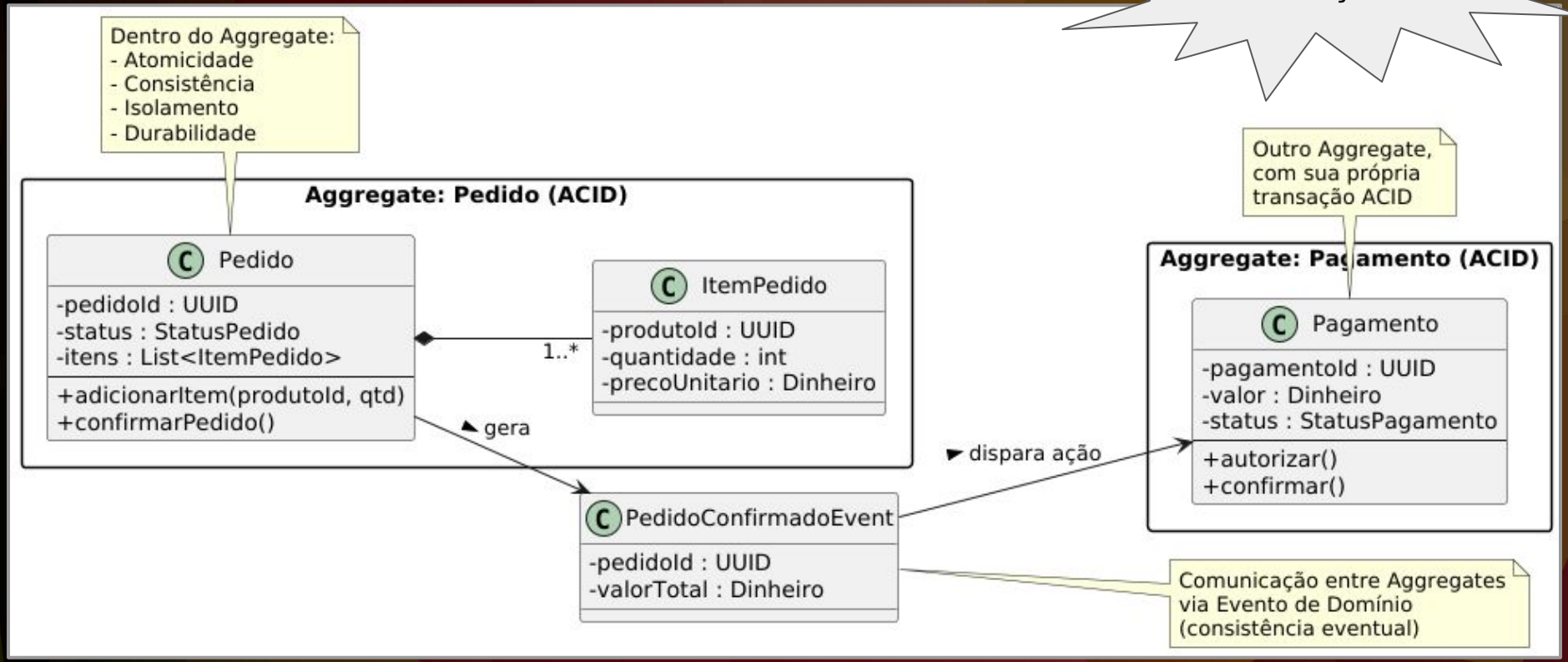
O que acontece quando queremos que vários microsserviços colaborem, talvez para implementar um processo de negócio?

Modelar e implementar esses tipos de fluxos de trabalho em sistemas distribuídos pode ser algo complexo.

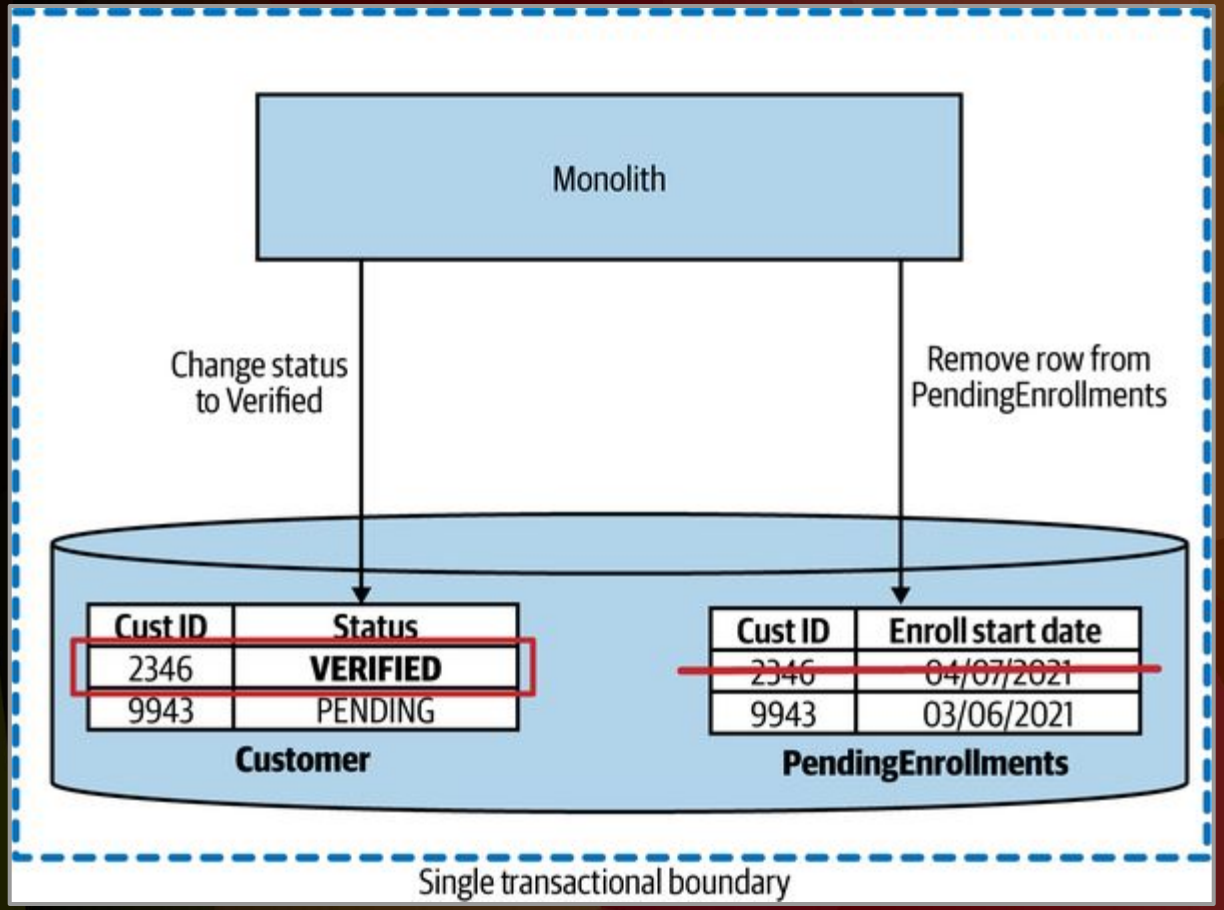
Transações

ACID	Relação com Aggregates no DDD
Atomicidade	**Aggregate** é a **unidade atômica de modificação**. Todas as mudanças em suas **entidades** e **value objects** devem ser aplicadas juntas em uma única transação.
Consistência	**Aggregate Root** garante as **invariantes de negócio** (ex.: Pedido não pode ter Item com quantidade ≤ 0). Assim, qualquer transação que termina deixa o **Aggregate** consistente.
Isolamento	É recomendável **manter Aggregates pequenos**, de modo que as transações concorrentes tenham menos chance de colisão. Dois **Aggregates** diferentes podem ser atualizados em paralelo.
Durabilidade	A persistência normalmente é feita via **repositórios** que salvam o estado do **Aggregate Root**. Uma vez commitado, o estado é durável.

Transações


Dentro do Aggregate:
- Atomicidade
- Consistência
- Isolamento
- Durabilidade
Aggregate: Pedido (ACID)
Pedido
-pedidoId : UUID
-status : StatusPedido
-itens : List<ItemPedido>
+adicionarItem(produtoId, qtd)
+confirmarPedido()
ItemPedido
-produtoId : UUID
-quantidade : int
-precoUnitario : Dinheiro
1..*
gera
PedidoConfirmadoEvent
-pedidoId : UUID
-valorTotal : Dinheiro
dispara ação
Outro Aggregate,
com sua própria
transação ACID
Aggregate: Pagamento (ACID)
Pagamento
-pagamentoId : UUID
-valor : Dinheiro
-status : StatusPagamento
+autorizar()
+confirmar()
Comunicação entre Aggregates
via Evento de Domínio
(consistência eventual)

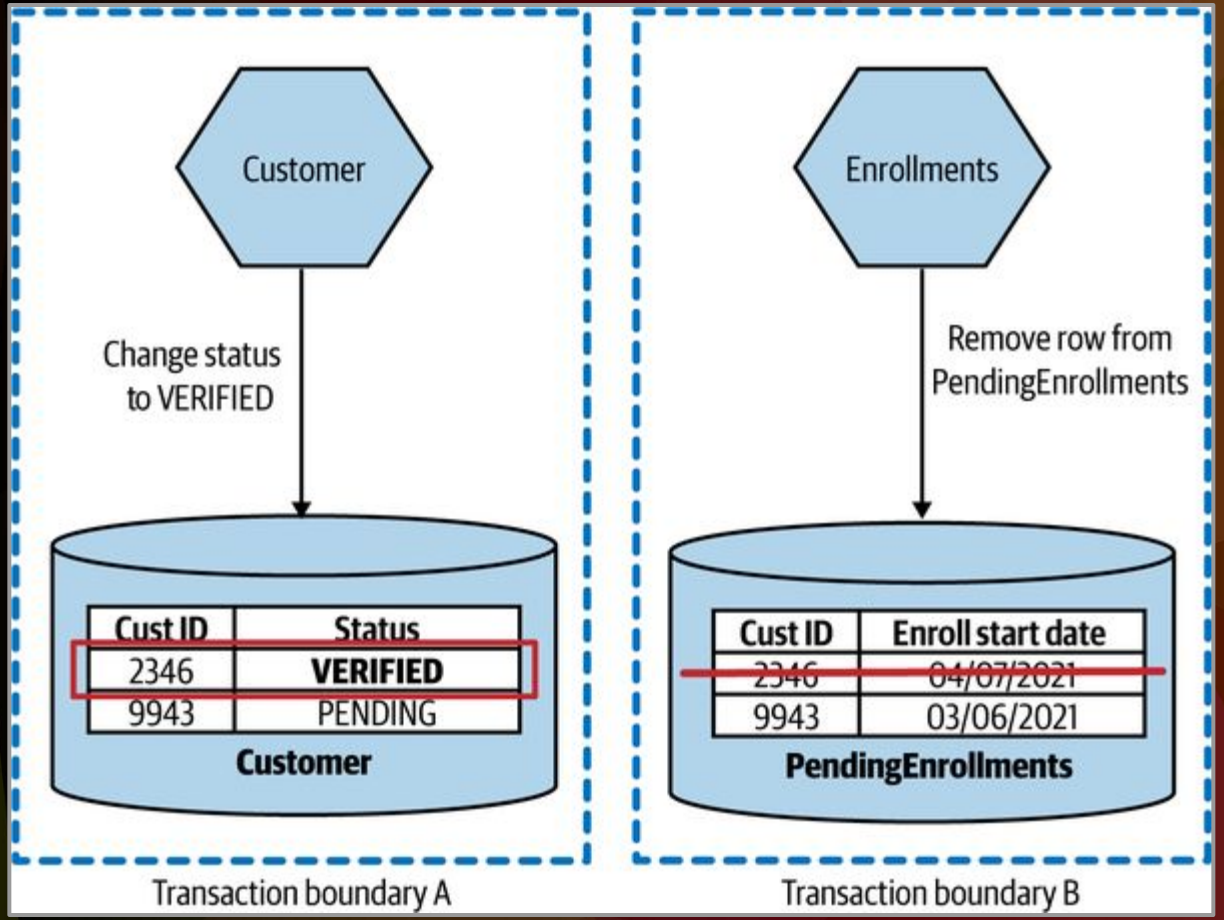
Aspecto	Dentro de um Aggregate (ACID)	Entre Aggregates (Consistência Eventual)
Unidade de transação	**Aggregate** é a **fronteira atômica**. Todas as mudanças em suas entidades e VOs ocorrem juntas.	Cada **Aggregate** tem sua própria transação independente.
Atomicidade	Ou todas as mudanças no **Aggregate** são aplicadas, ou nenhuma.	Não é garantida entre **Aggregates**, pois pode haver falhas parciais.
Consistência	Invariantes sempre mantidas pelo **Aggregate Root**.	Regras de negócio entre **Aggregates** podem ficar **temporariamente inconsistentes** até o processamento dos eventos.
Isolamento	Conflitos de concorrência são gerenciados dentro do **Aggregate**.	Concorrência entre **Aggregates** é **independente**, reduzindo contenção.
Durabilidade	Mudanças persistem após o commit do **Aggregate**.	Cada **Aggregate** persiste sua parte. Sincronização acontece via eventos.
Comunicação	Métodos e invariantes dentro do próprio **Aggregate**.	**Eventos de domínio** ou mensagens assíncronas publicadas.
Exemplo no E-Commerce	Adicionar `ItemPedido` a um `Pedido`: tudo ou nada na mesma transação.	`PedidoConfirmadoEvent` dispara processo no `Pagamento`, que pode ocorrer em outro momento.



Aqui, estamos monitorando o processo envolvido na integração de um novo cliente da empresa.

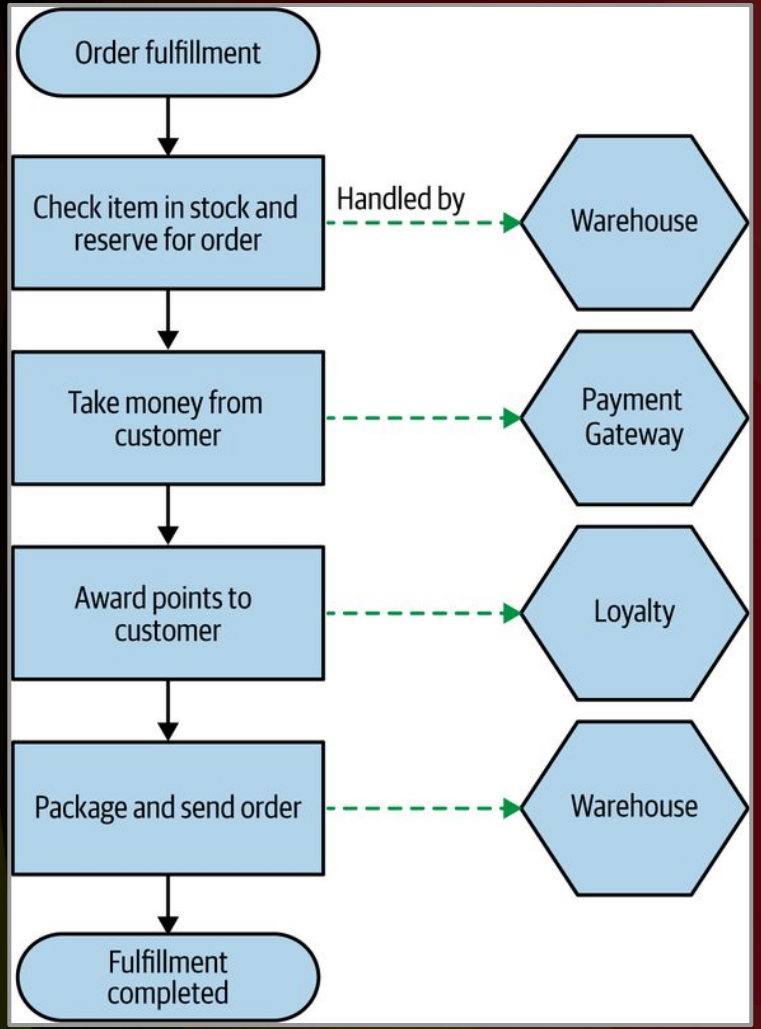
Chegamos ao fim do processo, que envolve a alteração do cliente 2346 de PENDING para VERIFIED.

Todas as solicitações de verificação de pendências devem ser removidas no processo.



Aqui estamos fazendo exatamente a mesma alteração, mas cada alteração é feita em um banco de dados diferente.

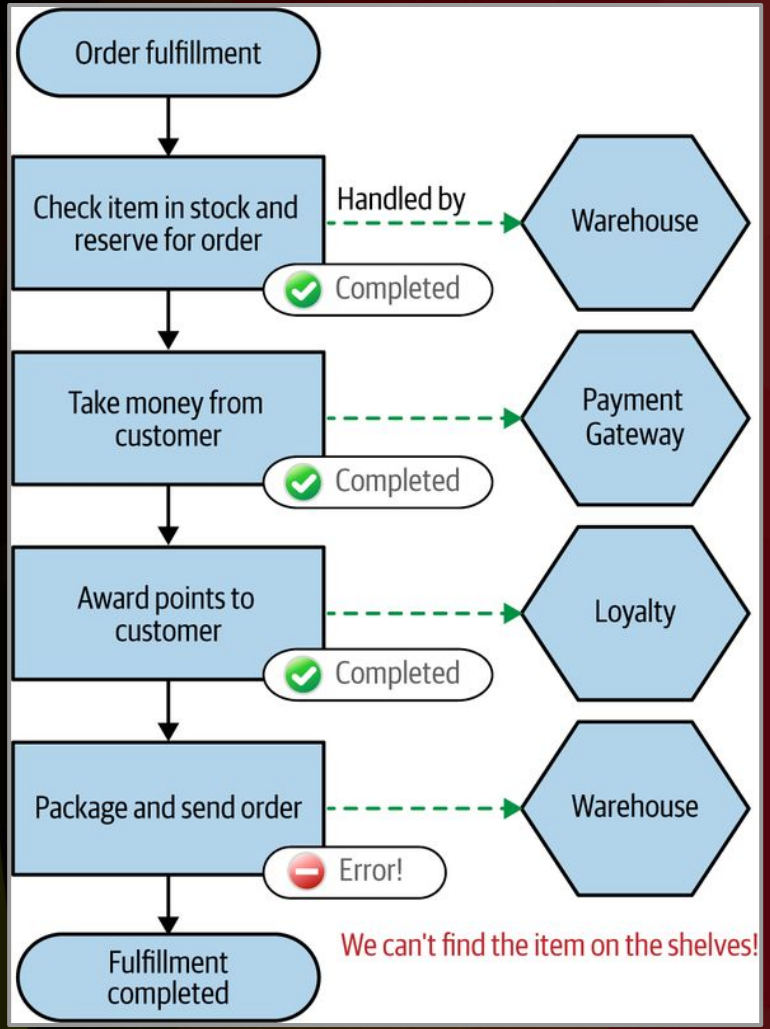
Isso significa que **há duas transações a serem consideradas, cada uma das quais pode funcionar ou falhar independentemente da outra.**



Aqui o processo de atendimento de pedidos é representado, com cada etapa neste fluxo representando uma operação que pode ser realizada por um serviço diferente.

Dentro de cada serviço, qualquer mudança de estado pode ser tratada dentro de uma transação ACID local.

Por exemplo, quando verificamos e reservamos estoque usando o serviço Warehouse, que internamente pode criar uma linha em sua tabela local Reservation registrando a reserva; essa mudança seria tratada dentro de uma transação normal de banco de dados.



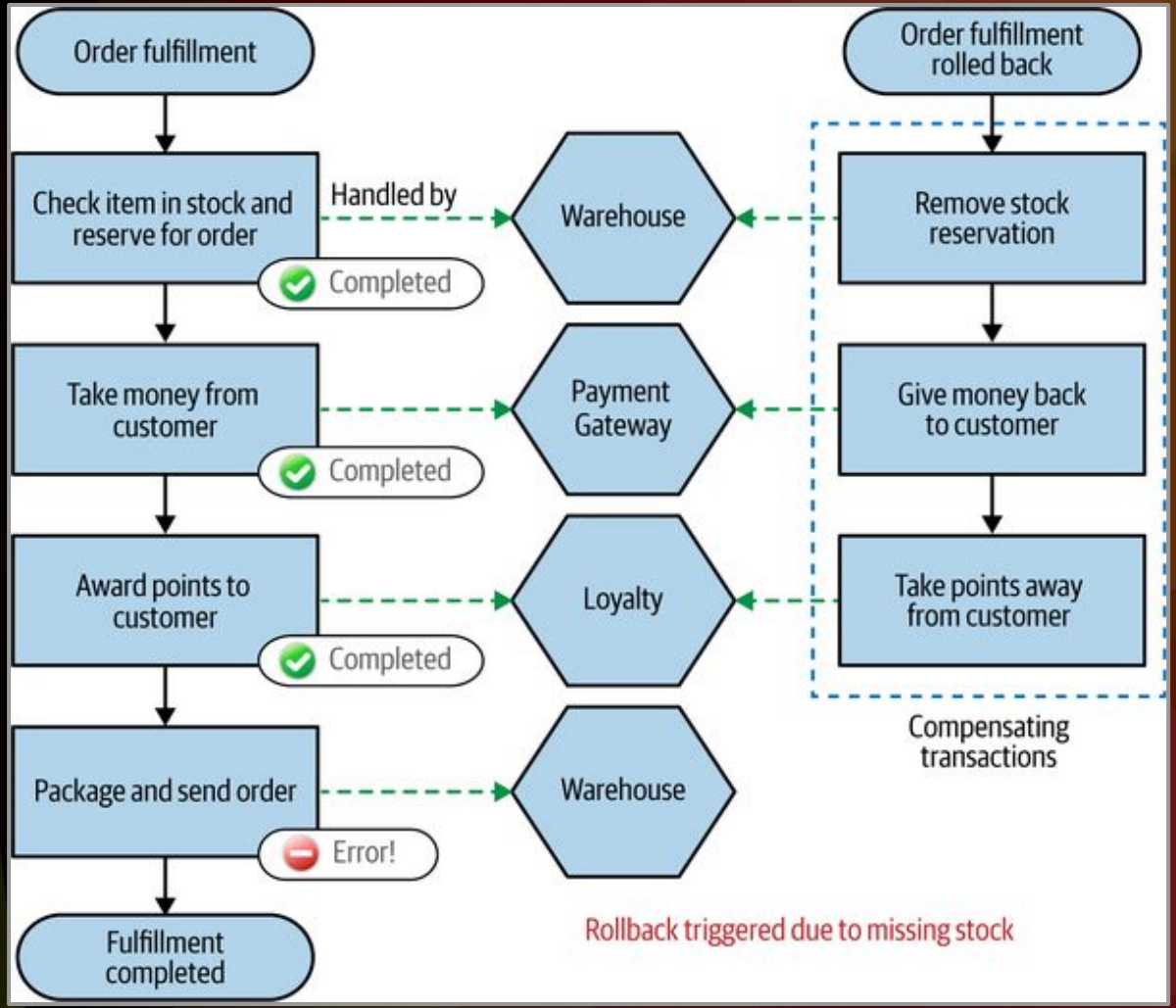
Considere um modo de falha potencial.

Chegamos ao ponto de tentar embalar o item, apenas para descobrir que o item não pode ser encontrado no depósito. Nosso sistema acha que o item existe, mas ele simplesmente não está na prateleira!

Agora, vamos supor que decidimos que queremos apenas reverter o pedido inteiro, em vez de dar ao cliente a opção de colocar o item em back order.

O problema é que já recebemos o pagamento e concedemos pontos de fidelidade para o pedido.

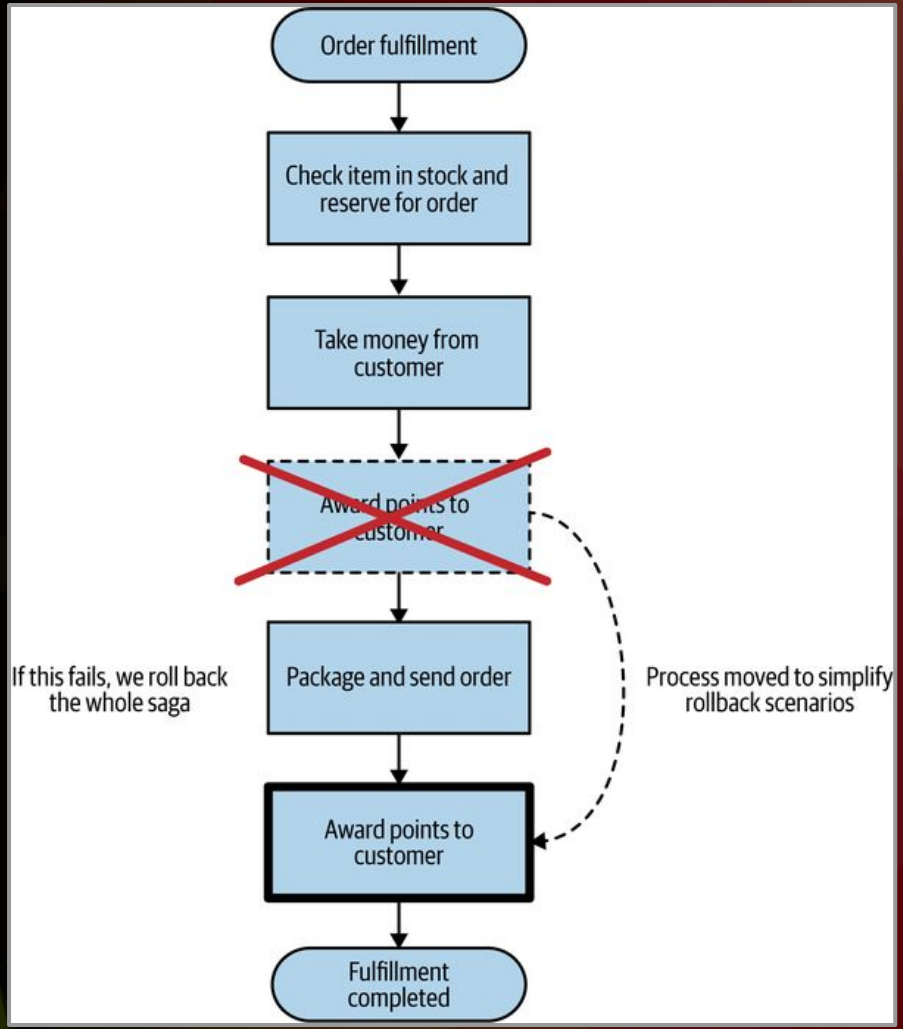
Não há um “rollback” simples para toda a operação.



Se quiser implementar um rollback, você precisa implementar uma **transação compensatória**.

A **transação compensatória** é uma operação que desfaz uma transação previamente comprometida.

Para reverter nosso processo de atendimento de pedidos, será preciso disparar a **transação compensatória** para cada etapa que já foi comprometida.



Nós poderíamos ter tornado nossos prováveis cenários de rollback um pouco mais simples ao reordenar as etapas em nosso fluxo de trabalho original.

Uma mudança simples seria conceder pontos somente quando o pedido for realmente despachado.

Às vezes é possível simplificar as operações de reversão apenas ajustando como o fluxo de trabalho é realizado.

Ao adiantar as etapas com maior probabilidade de falhar e falhar o processo mais cedo, evitamos ter que acionar transações compensatórias posteriores.

A **transação compensatória** é uma operação que desfaz uma transação previamente comprometida.

As **transações compensatórias** não se comportam exatamente como aquelas de um rollback normal de banco de dados.

Porque nem sempre podemos reverter uma transação de forma limpa, **dizemos que essas transações compensatórias são rollbacks semânticos**.

Algumas falhas podem exigir um **rollback** e outras podem ser **fail forward**.

No envio do pedido se por alguma razão a transportadora falhar, pode ser necessário a intervenção humana para redirecionar para uma outra transportadora.

Saga Design Pattern

Saga Design Pattern é uma maneira de gerenciar a consistência de dados entre microsserviços em cenários de transação distribuída.

Uma **Saga** é uma sequência de transações que atualiza cada serviço e publica uma mensagem ou evento para disparar a próxima etapa de transação.

Se uma etapa falhar, a **Saga** executará **transações compensatórias** que contrariam as transações anteriores.

O termo **Saga** se refere a **Long Lived Transactions**: vem do conceito de uma longa história com muitas partes, assim como uma transação distribuída.

Em uma **Saga**, cada parte da história é uma transação local e juntas elas formam a história completa.

Saga Design Pattern é um algoritmo que pode coordenar múltiplas mudanças de estado, mas evita a necessidade de bloquear recursos por longos períodos de tempo.

Uma **Saga** faz isso modelando as etapas envolvidas como atividades discretas que podem ser executadas independentemente.

Usar **Sagas** vem com o benefício adicional de nos forçar a modelar explicitamente nossos processos de negócios, o que pode ter benefícios significativos.

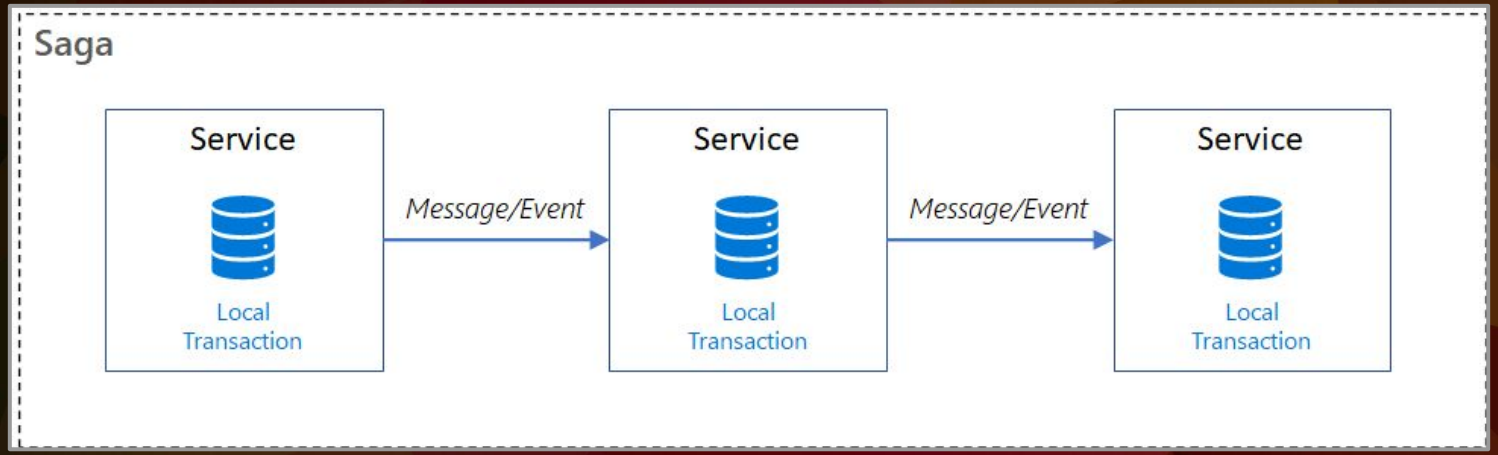
A ideia central delineada em "**Sagas**" por Hector Garcia-Molina e Kenneth Salem, aborda como lidar melhor com operações conhecidas como transações de longa duração. Essas transações podem levar muito tempo (minutos, horas ou talvez até dias) e como parte desse processo, exigem que sejam feitas alterações em um banco de dados.

Saga Design Pattern fornece gerenciamento de transações usando uma sequência de transações locais.

Uma transação local é o esforço de trabalho atômico realizado por um participante da **Saga**.

Cada transação local atualiza o banco de dados e publica uma mensagem ou evento para disparar a próxima transação local na Saga.

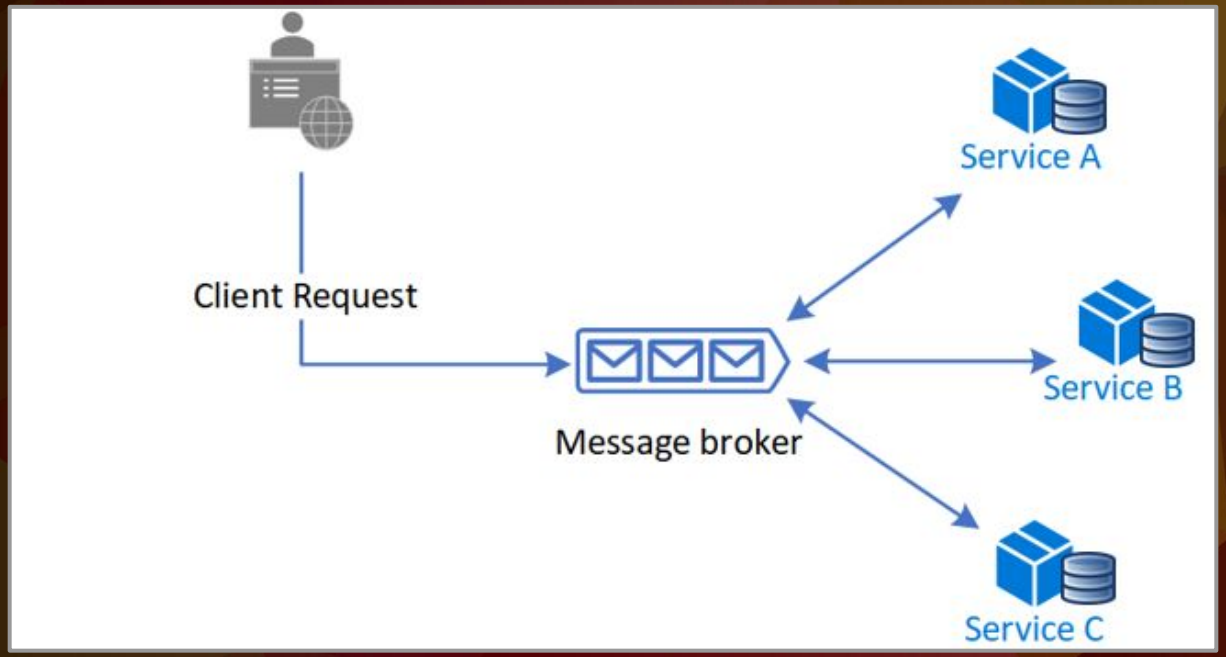
Se uma transação local falhar, a **Saga** executará uma série de **transações compensatórias** que desfazem as alterações feitas pelas transações locais anteriores.

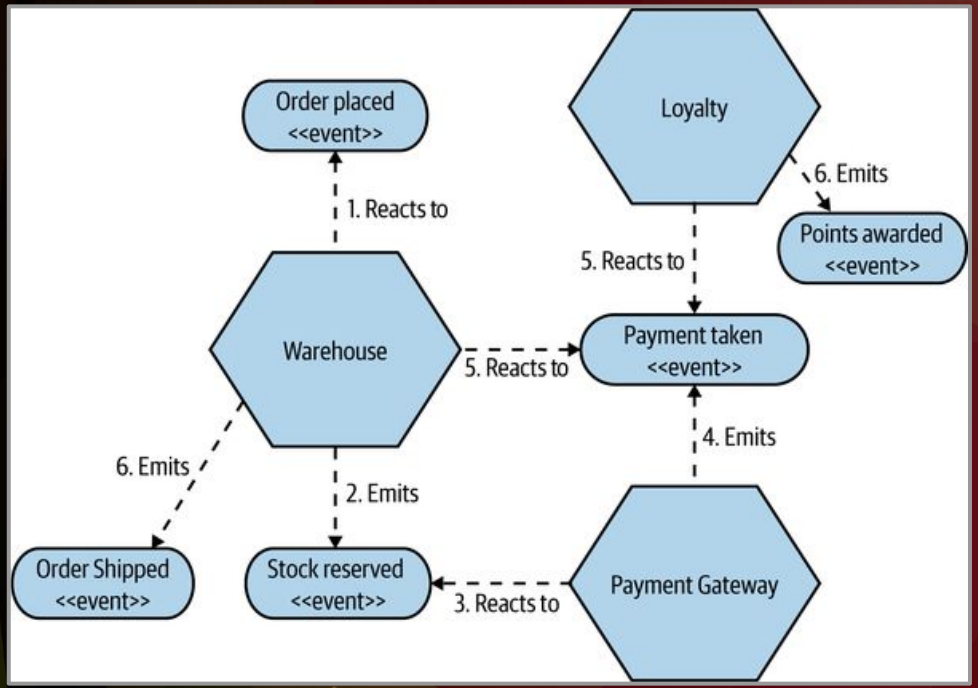


Implementação de Saga

Coreografia é uma maneira de coordenar **Sagas** em que os participantes trocam eventos sem um ponto centralizado de controle.

Com a **Coreografia**, cada transação local publica eventos de domínio que disparam transações locais em outros serviços.





Assíncrono

Uma **Saga Coreografada** visa distribuir a responsabilidade pela operação entre vários serviços de colaboração.

Esses microsserviços estão reagindo a eventos recebidos. Conceitualmente, os eventos são transmitidos no sistema, e as partes interessadas podem recebê-los.

Quando o serviço **Warehouse** recebe o primeiro evento **Order Placed**, ele sabe que sua função é reservar o estoque apropriado e disparar um evento quando isso for feito.

A falta de um lugar central para interrogar sobre o status de uma **Saga** é um grande problema.

Coreografia

Prós

Bom para fluxos de trabalho simples que exigem poucos participantes e não precisam de uma lógica de coordenação.

Não requer implementação e manutenção de serviço adicionais.

Não introduz um único ponto de falha, pois as responsabilidades são distribuídas entre os participantes da **Saga**.

Contras

O fluxo de trabalho pode se tornar confuso ao adicionar novas etapas, pois é difícil rastrear quais participantes da **Saga** escutam quais comandos.

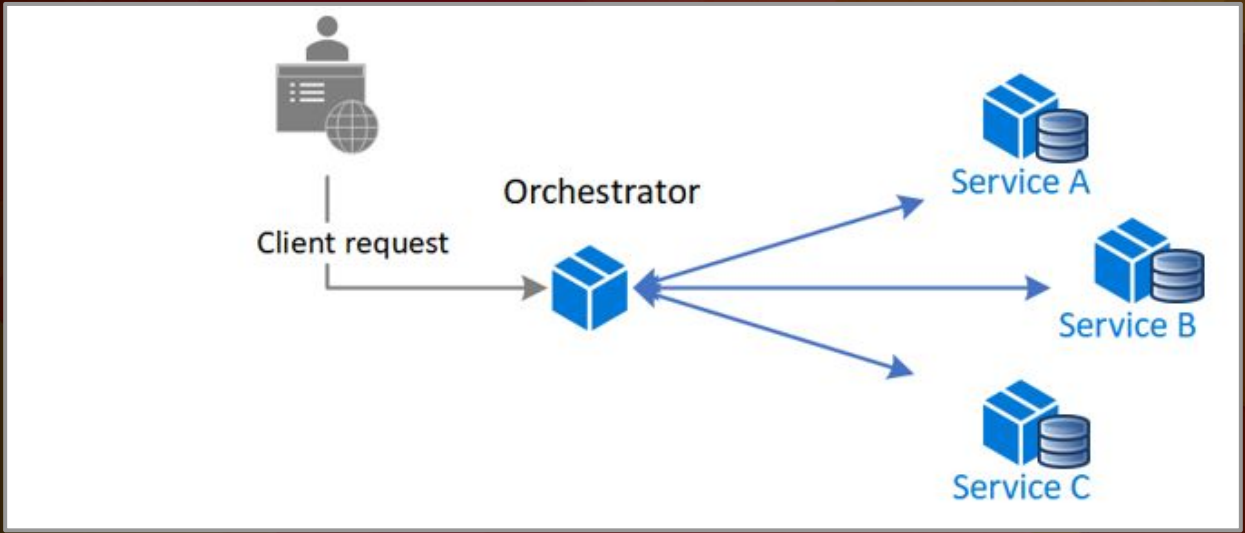
Há um risco de dependência cíclica entre os participantes da **Saga** porque eles têm que consumir os comandos uns dos outros.

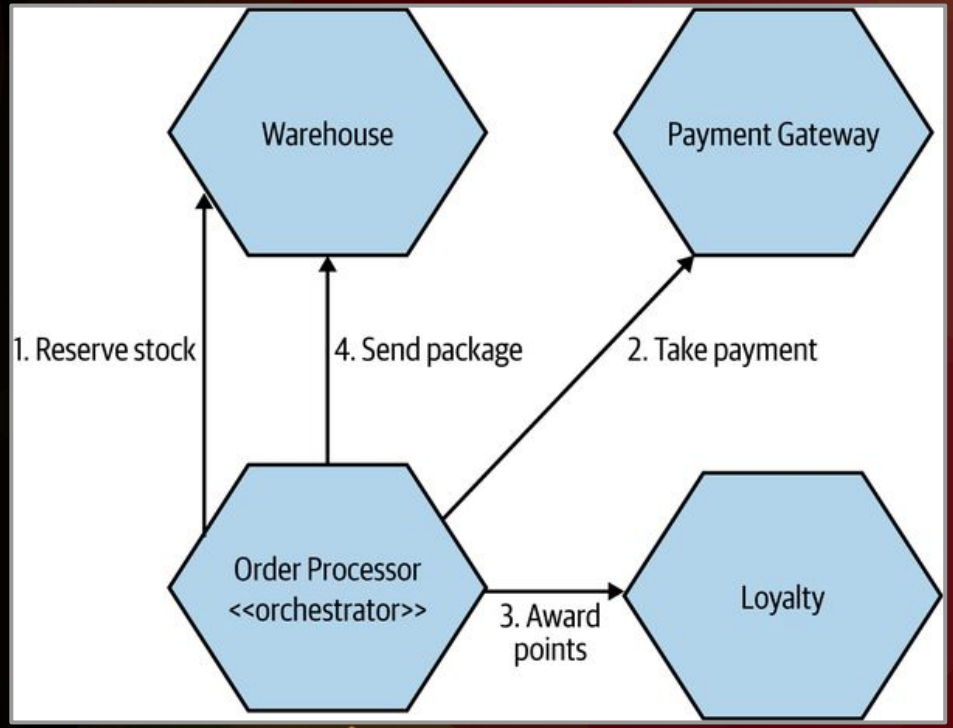
O teste de integração é difícil porque todos os serviços devem estar em execução para simular uma transação.

Orquestração é uma maneira de coordenar **Sagas** em que um controlador centralizado informa aos participantes da **Saga** quais transações locais executar.

O **Orquestrador de Saga** manipula todas as transações e informa aos participantes qual operação executar com base em eventos.

O **Orquestrador** executa solicitações de **Saga**, armazena e interpreta os estados de cada tarefa e lida com a recuperação de falhas com a compensação de transações.





Síncrono

Order Processor desempenha o papel de **Orquestrador**, coordenando nosso processo de atendimento.

Ele sabe quais serviços são necessários para executar a operação e decide quando fazer chamadas para esses serviços.

Se as chamadas falharem, ele pode decidir o que fazer como resultado.

Em geral, **Sagas Orquestradas** tendem a fazer uso pesado de interações de solicitação-resposta entre serviços: o **Order Processor** envia uma solicitação para serviços e espera uma resposta para informá-lo se a solicitação foi bem-sucedida e fornecer os resultados da solicitação.

Orquestração

Prós

Bom para fluxos de trabalho complexos envolvendo muitos participantes ou novos participantes adicionados ao longo do tempo.

Adequado quando há controle sobre cada participante no processo e controle sobre o fluxo de atividades.

Não introduz dependências cíclicas, pois o **Orquestrador** depende unilateralmente dos participantes da **Saga**.

Os participantes da **Saga** não precisam saber sobre comandos para outros participantes. A separação clara de preocupações simplifica a lógica de negócios.

Contras

A complexidade de design externa requer uma implementação de uma lógica de coordenação.

Há um ponto de falha externo, pois o **Orquestrador** gerencia o fluxo de trabalho completo.