Implementação e análise comparativa dos padrões saga: solução aplicada para sistema de gestão de vendas

Francisco Antonio da Silva Neto[1]*; Ugo Henrique Pereira da Silva[2]

[1] Universidade de São Paulo. Analista de sistemas. Avenida Franklin de Campos Sobral – Grageru; 49027-000 Aracaju, Sergipe, Brasil
[2] Instituto Federal de São Paulo, IFSP - Campus São Paulo, DTI-SCI. Mestre em Sistemas complexos. Rua Pedro Vicente – Luz; 01109010 - São Paulo, SP - Brasil
*autor correspondente: fasneto2@gmail.com

Implementação e análise comparativa dos padrões saga: solução aplicada para sistema de gestão de vendas

Introdução

A arquitetura de microsserviços tem se destacado como uma abordagem inovadora e eficiente para a criação de sistemas distribuídos, abordando questões como escalabilidade, modularidade e autonomia no processo de desenvolvimento (Newman, 2021). Grandes empresas como Amazon, Netflix, Twitter, PayPal já migraram parte dos seus sistemas para uma arquitetura de microsserviços (Liu et al., 2020).

Esta arquitetura permite a separação do sistema em serviços menores, autônomos e especializados, o que não acontece no monolito, aonde todos os componentes do projeto estão centralizados em um único ponto. Essa metodologia oferece benefícios como a escalabilidade direcionada, uma manutenção mais simplificada e a possibilidade de realizar atualizações e implantações de maneira independente. Contudo, a divisão do sistema em diversos serviços também apresenta desafios, especialmente no que tange à gestão da consistência dos dados e ao controle das transações distribuídas (Taibi et al., 2017; Newman, 2021; Liu et al., 2020).

Diferentemente de uma arquitetura monolítica aonde as transações ocorrem em um único banco de dados, no ambiente de microsserviços essas transações se dividem em bancos separados para cada serviço (Reza e Rahman, 2020). Esse contexto traz à tona a questão das transações distribuídas, que precisam gerenciar a complexidade de manter a consistência dos dados entre os serviços independentes. Caso haja uma falha entre a troca desses dados, uma inconsistência pode prejudicar o sistema como um todo. (Newman, 2021; Liu et al., 2020). Esta situação cria a necessidade de adotar padrões e soluções para garantir a integridade dos dados num sistema distribuído.

Uma forma de lidar com os desafios relacionados as transações distribuídas, é implementando o padrão Saga proposto por Garcia-Molina e Salem (1987). Esta abordagem divide a transação distribuída em uma sequência de transações locais, realizando a troca de dados dentro de único serviço. Cada transação pode ser cancelada por uma ação de compensação em caso de falhas, ou seja, o Saga assegura que todas as operações sejam executadas com sucesso ou reverte o processamento parcial das operações. O Saga possui duas abordagens de aplicações mais utilizadas: Saga Orquestrado e Saga Coreografado.

No Saga orquestrado, um serviço central interage de forma direta com os microsserviços, este serviço é chamado de orquestrador. Os microsserviços são independentes, a sua única comunicação é com o orquestrador. No Saga coreografado, não há esse orquestrador central, os microsserviços possuem uma lógica que determina qual será

o próximo serviço a ser executado no fluxo, neste caso os microsserviços não são independentes, eles se comunicam entre si. (Reza e Rahman, 2020; Stefanko et al., 2019; Garcia-Molina e Salem, 1987; Aydin e Çebi, 2022).

Mas, de acordo com Reza e Rahman (2020), o Saga apesar de ser útil para fazer o gerenciamento de transações distribuídas, ele apresenta uma fragilidade referente a problemas de infraestrutura, ou seja, caso haja uma interrupção como um serviço indisponível temporariamente ou não fornecer uma resposta em um tempo adequado, em qualquer um dos serviços, a execução da saga ficará comprometida. Com isso, complementando o padrão Saga, utiliza-se praticas modernas para melhorar a robustez e a confiabilidade das transações. Dentre essas práticas está o padrão *Outbox* (Laigner et al., 2020).

O padrão *Outbox* surge para resolver este problema referente a infraestrutura. Integrado ao Saga, ele é implementado a partir de uma tabela intermediaria, que armazena os dados de saída que serão enviados para outro microssserviço, eliminando o risco de perda de dados entre as transações caso haja alguma falha no processo, pois os dados só serão enviados ao próximo serviço se a operação for realizada com sucesso (Laigner et al., 2020; Reza e Rahman, 2020).

Segundo Newman (2021), Microsserviços emitem eventos, que são mensagens assíncronas, que podem ou não serem consumidos por outros microsserviços. Neste cenário, o serviço que emitiu o evento não possui conhecimento se há alguém consumindo essas mensagens.

A implementação esse tipo de comunicação baseada em eventos utiliza ferramentas como o Apache Kafka (John e Liu, 2017). Kreps et al. (2011) define o Kafka como uma plataforma de streaming de dados distribuída com grande capacidade de processamento de eventos. Este sistema trabalha em cima do conceito de publicação de mensagens em tópicos, aonde um *producer* publica uma mensagem e um *consumer* pode se inscrever nesse tópico e consumir este evento. O Kafka garante que cada envio seja entregue no mínimo uma vez garantindo a eficiência e preservando a ordem dos eventos na comunicação entre os microsserviços.

Em vista disso, este trabalho propõe uma implementação e comparação de um sistema de gestão de vendas utilizando os padrões Saga orquestrado e coreografado. Através de casos de uso práticos como fluxos de vendas e simulações de falhas, será possível validar o projeto, com o objetivo de realizar uma análise comparativa identificando as vantagens e complexidade de implementação e manutenção.

Metodologia ou Material e Métodos

A abordagem adotada para este trabalho é a de pesquisa experimental, devido a necessidade de analisar o comportamento dos padrões Saga em um ambiente controlado. Este tipo de pesquisa permite explorar relações de causa e efeito entre as variáveis envolvidas, como os padrões implementados entre os microsserviços e as condições de falha simuladas, buscando entender a influência sobre a consistência dos dados envolvidos na transação e a resiliência do sistema.

Neste projeto, serão implementados os padrões orquestrado e coreografado, trabalhando em cenários que simulam casos práticos, como transações de fluxo de venda padrão: validando produtos em estoque, processamento do pagamento, atualização do estoque; E simulações de falha controladas durante o fluxo de venda. Esta abordagem experimental permite que a recuperação das transações em um cenário de falha e a consistência dos dados entre os serviços sejam observados de forma direta, pois, permite a manipulação das variáveis de implementação e condições operacionais, gerando uma análise precisa dos resultados obtidos, podendo diferenciar as abordagens implementadas.

Os sistemas de gerenciamento dos microsserviços serão implementados utilizando tecnologias modernas que são adequadas ao cenário de sistemas distribuídos. É necessário garantir que o sistema seja altamente escalonável, resiliente e que as operações sejam consistentes. As seguintes são as ferramentas/tecnologias escolhidas para a implementação do sistema e a metodologia escolhida para finalizar o projeto:

- **Java 17 e Spring Boot 3:** Trata-se de uma versão estável do Java amplamente usada para a produção de aplicativos devido aos seus recursos avançados, responsivos e seguros. O Spring Boot 3 é um dos frameworks mais populares para o desenvolvimento de aplicações REST em uma arquitetura de microsserviços. Ele simplifica a criação de serviços com inicialização rápida, com um modelo de programação produtivo, além de fornecer um conjunto robusto de módulos e ferramentas prontos para produção.
- **Apache Kafka:** Plataforma de streaming de eventos distribuída, que proporciona uma comunicação assíncrona e desacoplada entre microsserviços. O Kafka irá coordenar os eventos no sistema, atuando como um intermediador de mensageria entre os serviços.
- **API REST:** A comunicação com os serviços necessários para a efetivação da venda será feita por meio de uma API REST exposta em dos serviços, oferecendo os endpoints para executar o processo e recuperar os dados do evento.

- **PostgreSQL e MongoDB:** O sistema adotará uma abordagem de banco de dados híbrida. Para o serviço de entrada será utilizado um banco NoSQL, o MongoDB, ideal para armazenar dados semiestruturados e permite facilmente a evolução da estrutura de dados da venda. Para os demais serviços será utilizado o PostgreSQL, um banco de dados relacional adequado para armazenar informações estruturadas e consistentes, como validação de produtos, pagamentos e estoque.
- **Docker e docker-compose:** Docker será utilizado para conteinerizar cada microsserviço e seu ambiente, permitindo facilmente a replicação e implantação do sistema. Com o uso de Docker Compose, podemos gerenciar a execução do contêiner garantindo que todos os serviços como Kafka, MongoDB, PostgreSQL sejam iniciados e devidamente configurados, faciltando a criação e a execução do sistema.
- **Redpanda Console:** Plataforma de gerenciamento e monitoramento das mensagens trocadas entre os microsserviços a partir do Kafka, permitindo a visualização dos eventos em tempo real, permitindo a análise dos dados de comunicação.

Saga orquestrado

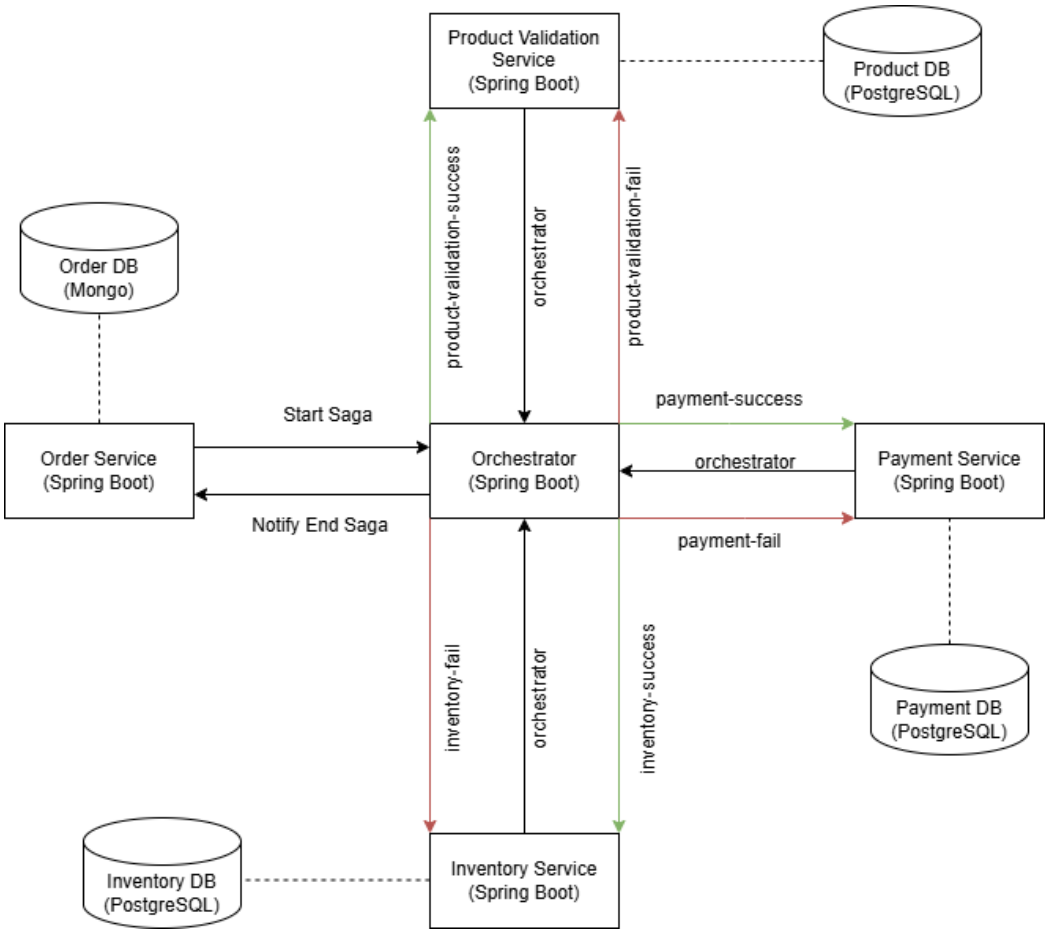

Figura 1. Representação da saga orquestrada

Fonte: Dados originais da pesquisa

A figura 1 representa a arquitetura proposta para a saga orquestrada. Composta por cinco microsserviços que possuem responsabilidades específicas no fluxo de execução da venda. A seguir será descrito de forma detalhada a funcionalidade de cada serviço.

1. **Order-Service:** Ponto inicial e final do fluxo, este serviço tem como responsabilidade a criação inicial de um pedido. Disponibilizando endpoints REST para iniciar o processo e consultar os dados relacionados aos eventos. Para armazenamento das informações dos eventos, será utilizado o banco de dados MongoDB
2. **Orchestrator-Service:** Serviço responsável por coordenar o fluxo de execução da saga. Gerenciando a sequência de execução dos demais serviços, controlando o estado atual de cada um dos serviços e determinando o próximo passo a ser executado. Diferente dos demais, este serviço não utiliza um banco de dados, uma vez que o seu dever é apenas de coordenação.
3. **Product-Validation-Service:** Sua função é validar a existência de um produto no pedido e também se o mesmo existe na base de dados. Ele armazena a validação do produto associado ao ID do pedido. O banco de dados utilizado neste e nos próximos serviços será o PostgreSQL.
4. **Payment-Service**: Serviço encarregado de realizar o processamento de pagamentos com base nos valores unitários e quantidades de cada item especificado no pedido. Ele armazena as informações relacionadas a pagamentos de cada um dos pedidos executados.
5. **Inventory-Service:** Serviço responsável por dar a baixa no estoque dos produtos associados a um pedido, armazenando as informações necessárias vinculadas a cada ID do pedido.

Definido os serviços que compõe a arquitetura, o próximo passo será o mapeamento dos tópicos. Esta etapa é crucial, pois a partir dela é possível garantir uma comunicação eficiente e consistente entre os microsserviços. Os tópicos desempenham um papel de um canal de mensagens, permitindo a troca de informações entre os serviços, garantindo que cada etapa do fluxo seja efetuada de forma coordenada e confiável, realizando o tratamento apropriado em situações de falhas e permitindo a rastreabilidade das operações. A tabela 1 detalha a definição desses tópicos para cada serviço e o seu tipo.

Tabela 1. Tópicos da arquitetura saga orquestrada

Serviço	Tópico	Tipo
order-service	notify-ending	consumer
order-service	start-saga	producer
orchestrator	orchestrator	consumer
orchestrator	finish-success	producer/consumer
orchestrator	finish-fail	producer/consumer
orchestrator	notify-ending	producer
product-validation-service	product-validation-success	consumer
product-validation-service	product-validation-fail	consumer
product-validation-service	orchestrator	producer
payment-service	payment-success	consumer
payment-service	payment-fail	consumer
payment-service	orchestrator	producer
inventory-service	inventory-success	consumer
inventory-service	inventory-fail	consumer
inventory-service	orchestrator	producer

Fonte: Dados originais da pesquisa

Após o mapeamento dos tópicos, foi definido a infraestrutura de containers utilizando Docker. Estruturado a partir docker-compose.yml para criar e gerenciar os contêineres necessários para a execução da aplicação. Foram configurados contêineres para os bancos de dados, microsserviços, Kafka e o Redpanda console, garantindo um ambiente integrado e funcional para o sistema.

Em cada microsserviço foi preciso definir algumas propriedades do Kafka a partir do arquivo application.yml, definindo algumas informações como o endereço do servidor do Kafka, os tópicos específicos de cada serviço e as propriedades dos *consumers*, o *group-id*, responsável por garantir que os eventos sejam distribuídos de forma correta entre as instâncias do mesmo serviço e o *auto-offset-reset* definido como *latest,* que certifica que os *consumers* iniciem a leitura dos eventos a partir do *offset* mais recente, se adequando ao nosso cenário aonde não há a necessidade de reprocessamento de eventos mais antigos.

Estas informações foram utilizadas em outro arquivo chamado KafkaConfig.java. A função deste arquivo é implementar as funções que serão responsáveis por fazer as configurações dos *consumers* e *producers* e criar os tópicos necessários para o funcionamento da aplicação.

Concluída a etapa de configuração do Kafka para cada serviço e com os tópicos configurados, o próximo passo foi a implementação das lógicas dos microsserviços garantindo que cada serviço execute usa respectiva tarefa e a sua comunicação de forma eficiente. Estes serviços foram projetados para consumir e produzir eventos em tópicos específicos, seguindo o fluxo da orquestração. Cada serviço possui uma estrutura de pacotes padronizada, adotando boas práticas de desenvolvimento de software. Esses pacotes possuem arquivos

contendo classes, interfaces e enums dividindo as responsabilidades e facilitando a compreensão, manutenção e escalabilidade do código. A seguir, está descrito a estruturação desses pacotes:

- **Consumer**: Possui classes que são responsáveis por consumir os eventos publicados nos tópicos do Kafka, contendo métodos que processam eventos com status de sucesso ou falha;
- **DTO**: Pacote que contém os objetos de transferência de dados, essas classes são utilizadas para determinar os dados que serão exibidos e trocados entre os microsserviços;
- **Enums**: Contém classes do tipo enum que armazenam constantes que serão utilizadas durante a execução dos serviços;
- **Model**: Pacote com as entidades são mapeadas para as tabelas do banco de dados;
- **Producer**: Possui a classe responsável por produzir eventos de sucesso ou falha nos tópicos do Kafka.
- **Repository**: Configura as interfaces que acessam o banco de dados e métodos específicos para a busca de informações;
- **Service**: Camada que implementa a lógica com as regras de negócio de cada microsserviço;
- **Controller** (exclusivo do serviço order-service): Abriga as classes que definem os enpoints REST que serão utilizados para a criação e busca dos pedidos;
- **Saga** (exclusivo do serviço orchestrator-service): Contém classes que tem como responsabilidade orquestrar o fluxo de execução dos eventos. Essas classes que vão definir qual será o próximo serviço a ser executado.

O pacote Saga desempenha um papel fundamental no gerenciamento centralizado do fluxo da transação distribuída. Neste pacote está contida a classe SagaExecutionController que determina quais serviços serão executados com base nas informações presentes no evento publicado em seu tópico.

Esta classe utiliza uma matriz de mapeamento que faz a associação do status do evento, a sua origem e o tópico a ser publicado. Os status podem ser *SUCCESS, ROLLBACK_PENDING* ou *FAIL* e a origem é o serviço que publicou o evento no tópico orquestrador. A tabela 2 apresenta de forma detalhada a matriz com as combinações possíveis.

As informações contidas na matriz são utilizadas como um guia para os métodos implementados na classe fazer o direcionamento do fluxo. Ao receber um evento, o método

responsável por esse tratamento verifica qual é a origem e status, comparando com os valores possíveis da matriz e determinando o próximo tópico e fazendo a produção de um novo evento. Dessa forma a arquitetura orquestrada foi implementada com sucesso, pronta para realizar e coordenar as transações de forma centralizada.

Tabela 2. Matriz de mapeamento

Serviço Origem	Status	Tópico
orchestrator	SUCCESS	product-validation-success
orchestrator	FAIL	finish-fail
product-validation-service	ROLLBACK_PENDING	product-validation-fail
product-validation-service	FAIL	finish-fail
product-validation-service	SUCCESS	payment-success
payment-service	ROLLBACK_PENDING	payment-fail
payment-service	FAIL	product-validation-fail
payment-service	SUCCESS	inventory-success
inventory-service	ROLLBACK_PENDING	inventory-fail
inventory-service	FAIL	payment-fail
inventory-service	SUCCESS	finish-success

Fonte: Dados originais da pesquisa

Saga coreografado

A figura 2 descreve o funcionamento da aplicação utilizando o padrão coreografado. Neste cenário, os serviços possuem uma ligação direta entre si, comunicando por meio dos eventos publicados no Kafka.

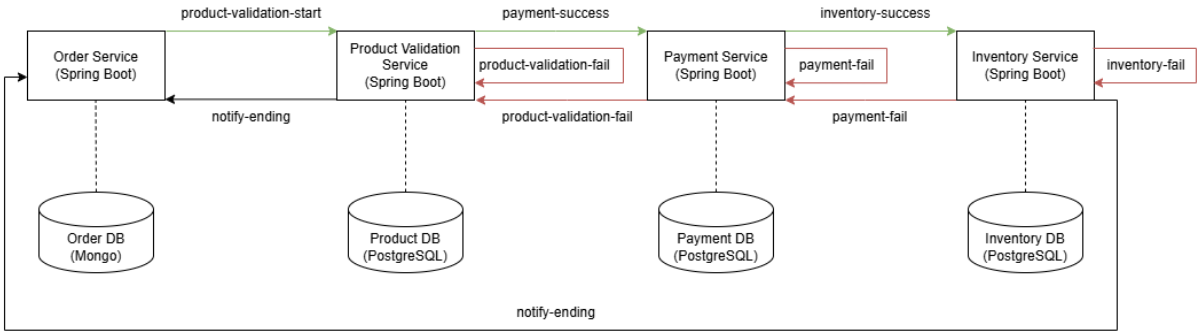

Figura 2. Representação da saga coreografada

Fonte: Dados originais da pesquisa

Diferente padrão do orquestrado, no coreografado não há mais o serviço central orchestrator-service que tinha a responsabilidade de coordenar o fluxo da operação, determinando qual seria a próxima ação. Agora, cada microsserviço possui a obrigação de fazer o gerenciar os seus eventos. Esta autonomia é implementada a partir do pacote Saga, presente em todos os serviços, contendo a classe SagaExecutionController. Esta classe atua

como o processador de cada microsserviço, implementando métodos que tratam os eventos recebidos a partir dos consumers e decidem para qual tópico produzir o evento tratado, eliminando a necessidade de um serviço especifico para realizar essa lógica

Os serviços utilizados nesse modelo foram os seguintes:

1. **Order-Service;**
2. **Product-Validation-Service;**
3. **Payment-Service**;
4. **Inventory-Service;**

Para implementar o padrão coreografado foram utilizadas as configurações para o processo de criação dos contêineres, do funcionamento do Kafka e a organização dos pacotes, que seguem mesmos padrões descritos na seção anterior. A tabela 3 detalha os tópicos mapeados para a arquitetura coreografada que serão utilizados para as configurações de produção e consumo dos eventos.

Tabela 3. Tópicos da arquitetura saga coreografado

Serviço	Tópico	Tipo
order-service	product-validation-start	producer
order-service	notify-ending	consumer
product-validation-service	payment-success	producer
product-validation-service	product-validation-fail	consumer/producer
product-validation-service	notify-ending	producer
product-validation-service	product-validation-start	consumer
payment-service	inventory-success	producer
payment-service	payment-fail	consumer/producer
payment-service	payment-success	consumer
payment-service	product-validation-fail	producer
inventory-service	inventory-success	consumer
inventory-service	inventory-fail	consumer/producer
inventory-service	notify-ending	producer
inventory-service	payment-fail	producer

Fonte: Dados originais da pesquisa

Neste padrão, foi implementado em cada serviço pacote Saga com a classe SagaExecutionController, necessário para configurar os cenários da transação a depender do seu estado (*SUCCESS, ROLLBACK_PENDING* ou *FAIL),* para decidir qual tópico do fluxo publicar o evento.

Resultados Preliminares

Finalizada as implementações para os padrões orquestrado e coreografado, foram realizados testes buscando avaliar o sistema através do seu comportamento em situações de sucesso e falha. No cenário de sucesso o teste é realizado com o usuário realizando uma compra, na falha, o usuário solicita uma quantidade de item que não está disponível no estoque.

Saga Orquestrado

No cenário de sucesso, o sistema executou todos os serviços coordenado pelo orchestrator-service, conforme o esperado. Demonstrando a eficácia na orquestração das transações.

Após a publicação do primeiro evento partindo do order-service, o orquestrador garantiu uma execução sequencial, publicando e consumindo os eventos nos tópicos necessários, seguindo a lógica configurada no SagaExecutionController com a matriz de mapeamento. A figura 4 mostra a criação do pedido, aonde é possível observar o status “SUCCESS” que indica que o fluxo foi realizado de forma. Vale observar que esta arquitetura permite uma rastreabilidade precisa da transação, registrando o histórico de execução de cada etapa, trazendo informações do serviço de origem, o status da execução e uma mensagem informativa (Figura 5)

```
"id": "67a3fe567547db058b93a38a",
"transactionId": "1738800726413_662a2454-7d1e-450f-b5db-a9020c12bf4a",
"orderId": "67a3fe567547db058b93a389",
"payload": {
    "id": "67a3fe567547db058b93a389",
    "products": [
        {
            "product": {
                "code": "SMARTPHONE",
                "unitValue": 1015.5
            },
            "quantity": 3
        },
        {
            "product": {
                "code": "NOTEBOOK",
                "unitValue": 2999.9
            },
            "quantity": 1
        }
    ],
    "createdAt": "2025-02-05T21:12:06.413",
    "transactionId": "1738800726413_662a2454-7d1e-450f-b5db-a9020c12bf4a",
    "totalAmount": 6046.4,
    "totalItems": 4
},
"source": "ORCHESTRATOR",
"status": "SUCCESS",
```


Figura 4. Criação com sucesso do pedido

Fonte: Dados originais da pesquisa

```
"eventHistory": [
    {
        "source": "ORCHESTRATOR",
        "status": "SUCCESS",
        "message": "Saga started!",
        "createdAt": "2025-02-05T21:12:06.43"
    },
    {
        "source": "PRODUCT_VALIDATION_SERVICE",
        "status": "SUCCESS",
        "message": "Products are validated successfully!",
        "createdAt": "2025-02-05T21:12:06.454"
    },
    {
        "source": "PAYMENT_SERVICE",
        "status": "SUCCESS",
        "message": "Payment realized successfully!",
        "createdAt": "2025-02-05T21:12:06.727"
    },
    {
        "source": "INVENTORY_SERVICE",
        "status": "SUCCESS",
        "message": "Inventory updated successfully!",
        "createdAt": "2025-02-05T21:12:07.051"
    },
    {
        "source": "ORCHESTRATOR",
        "status": "SUCCESS",
        "message": "Saga finished successfully!",
        "createdAt": "2025-02-05T21:12:07.114"
    }
],
```


Figura 5. Histórico de execução dos serviços

Fonte: Dados originais da pesquisa

Em relação a falhas, o sistema demonstrou robustez ao realizar as ações de compensação. Ao simular a falha no serviço de inventário (responsável por validar a quantidade do produto no estoque), o orquestrador teve a capacidade de coordenar com eficiência o processo de retorno ao estado inicial garantindo a consistência dos dados. Primeiro, realizando o *rollback* na quantidade disponível no inventário, compensando o pagamento e revertendo a validação dos produtos. A figura 6 mostra o histórico da execução dos serviços com a compensação após a falha no serviço do estoque.

```
"source": "ORCHESTRATOR",
"status": "FAIL",
"eventHistory": [
    {
        "source": "ORCHESTRATOR",
        "status": "SUCCESS",
        "message": "Saga started!",
        "createdAt": "2025-02-05T21:28:40.294"
    },
    {
        "source": "PRODUCT_VALIDATION_SERVICE",
        "status": "SUCCESS",
        "message": "Products are validated successfully!",
        "createdAt": "2025-02-05T21:28:40.312"
    },
    {
        "source": "PAYMENT_SERVICE",
        "status": "SUCCESS",
        "message": "Payment realized successfully!",
        "createdAt": "2025-02-05T21:28:40.34"
    },
    {
        "source": "INVENTORY_SERVICE",
        "status": "ROLLBACK_PENDING",
        "message": "Fail to update inventory: Product is out of stock!",
        "createdAt": "2025-02-05T21:28:40.374"
    },
    {
        "source": "INVENTORY_SERVICE",
        "status": "FAIL",
        "message": "Rollback executed for inventory!",
        "createdAt": "2025-02-05T21:28:40.399"
    },
    {
        "source": "PAYMENT_SERVICE",
        "status": "FAIL",
        "message": "Rollback executed for payment!",
        "createdAt": "2025-02-05T21:28:40.418"
    },
    {
        "source": "PRODUCT_VALIDATION_SERVICE",
        "status": "FAIL",
        "message": "Rollback executed on product validation!",
        "createdAt": "2025-02-05T21:28:40.439"
    },
    {
        "source": "ORCHESTRATOR",
        "status": "FAIL",
        "message": "Saga finished with errors!",
        "createdAt": "2025-02-05T21:28:40.448"
    }
```


Figura 6. Histórico de execução dos serviços com a execução do *rollback*

Fonte: Dados originais da pesquisa

Saga Coreografado

Na arquitetura coreografada, foi possível observar que o controle da aplicação foi distribuído entre os serviços participantes do fluxo, com a comunicação direta por meio dos eventos publicados no Kafka. Essa distribuição fez com que cada microsserviço tivesse autonomia para gerenciar a comunicação e as transações.

O fluxo iniciou-se no order-service e em seguida já publica o evento no tópico do próximo serviço e assim por diante, após a conclusão do pedido o fluxo retorna para o serviço inicial com o status “SUCCESS”. A sequência lógica para o cenário é semelhante ao modelo anterior, diferindo apenas na eliminação do serviço central.

A figura 7 mostra o histórico de execução dos serviços no padrão coreografado


```
"source": "ORDER_SERVICE",
"status": "SUCCESS",
"eventHistory": [
    {
        "source": "ORDER_SERVICE",
        "status": "SUCCESS",
        "message": "Saga started!",
        "createdAt": "2025-02-05T23:17:09.772"
    },
    {
        "source": "PRODUCT_VALIDATION_SERVICE",
        "status": "SUCCESS",
        "message": "Products are validated successfully!",
        "createdAt": "2025-02-05T23:17:09.79"
    },
    {
        "source": "PAYMENT_SERVICE",
        "status": "SUCCESS",
        "message": "Payment realized successfully!",
        "createdAt": "2025-02-05T23:17:09.807"
    },
    {
        "source": "INVENTORY_SERVICE",
        "status": "SUCCESS",
        "message": "Inventory updated successfully!",
        "createdAt": "2025-02-05T23:17:09.848"
    },
    {
        "source": "ORDER_SERVICE",
        "status": "SUCCESS",
        "message": "Saga finished successfully!",
        "createdAt": "2025-02-05T23:17:09.854"
    }
```


Figura 7. Histórico de execução dos serviços

Fonte: Dados originais da pesquisa

No cenário de falha, o serviço responsável pelo estoque identificou a inconsistência entre os dados e emitiu o evento de falha, mas dessa vez com o conhecimento de qual serviço iniciaria o processo de reversão dos dados para o estado inicial.

```
"source": "ORDER_SERVICE",
"status": "FAIL",
"eventHistory": [
    {
        "source": "ORDER_SERVICE",
        "status": "SUCCESS",
        "message": "Saga started!",
        "createdAt": "2025-02-05T23:16:27.867"
    },
    {
        "source": "PRODUCT_VALIDATION_SERVICE",
        "status": "SUCCESS",
        "message": "Products are validated successfully!",
        "createdAt": "2025-02-05T23:16:28.149"
    },
    {
        "source": "PAYMENT_SERVICE",
        "status": "SUCCESS",
        "message": "Payment realized successfully!",
        "createdAt": "2025-02-05T23:16:28.431"
    },
    {
        "source": "INVENTORY_SERVICE",
        "status": "ROLLBACK_PENDING",
        "message": "Fail to update inventory: Product is out of stock!",
        "createdAt": "2025-02-05T23:16:28.694"
    },
    {
        "source": "INVENTORY_SERVICE",
        "status": "FAIL",
        "message": "Rollback executed for inventory!",
        "createdAt": "2025-02-05T23:16:28.764"
    },
    {
        "source": "PAYMENT_SERVICE",
        "status": "FAIL",
        "message": "Rollback executed for payment!",
        "createdAt": "2025-02-05T23:16:28.78"
    },
    {
        "source": "PRODUCT_VALIDATION_SERVICE",
        "status": "FAIL",
        "message": "Rollback executed on product validation:",
        "createdAt": "2025-02-05T23:16:28.818"
    },
    {
        "source": "ORDER_SERVICE",
        "status": "FAIL",
        "message": "Saga finished with errors!",
        "createdAt": "2025-02-05T23:16:28.839"
    }
],
```


Figura 8. Fluxo de execução de rollback

Fonte: Dados originais da pesquisa

Análise comparativa

Em relação a implementação, o padrão orquestrado apresentou uma complexidade maior devido ao fato da criação do serviço central e toda a lógica de controle de execução a partir dos dados recebidos nos eventos, aumentando a dependência da aplicação em um único ponto de coordenação. Mas, essa centralização facilita a compreensão e manutenibilidade da aplicação.

O padrão coreografado apresentou incialmente uma implementação mais fácil, diminuindo a quantidade de código centralizado, porém, houve um aumento na complexidade de manutenção com a inserção da lógica de controle distribuído entre os serviços e dificuldades para compreender o fluxo da aplicação por completo.

Os dois padrões se mostraram capaz em manter a consistência entre os dados trafegados nos dois cenários. Cada um com a sua particularidade, mas, indicando a partir dos resultados a sua viabilidade de aplicação em um ambiente de gestão de transações distribuídas como o sistema de vendas apresentado neste trabalho

Referências

Newman, S. 2021. Building Microservices: Designing Fine-Grained Systems. O'Reilly Media. Sebastopol.

G. Liu, B. Huang, Z. Liang, M. Qin, H. Zhou and Z. Li. 2020. Microservices: architecture, container, and challenges, IEEE 20th International Conference on Software Quality, Reliability and Security Companion (QRS-C), Macau, China, 2020, pp. 629-635.

Taibi, D., Lenarduzzi, V., Pahl, C., & Janes, A. 2017. Microservices in agile software development: a workshop-based study into issues, advantages, and disadvantages. Proceedings of the XP2017 Scientific Workshops. 2017.

Reza, K., Rahman, N. 2023. Explication and Extension of Saga and Microservice Patterns to Enable Resilient Distributed Transaction. In: Ranganathan, G., Fernando, X., Rocha, Á. (eds) Inventive Communication and Computational Technologies. Lecture Notes in Networks and Systems, vol 383. Springer, Singapore.

Garcia-Molina, H., & Salem, K. 1987. Sagas. ACM Sigmod Record, 16(3), 249-259.

Štefanko, M., Chaloupka, O., Rossi, B., van Sinderen, M., & Maciaszek, L. 2019. The saga pattern in a reactive microservices environment. In Proc. 14th Int. Conf. Softw. Technologies (ICSOFT 2019) (pp. 483-490). Prague, Czech Republic: SciTePress.

Aydin, S., & Çebi, C. B. 2022. Comparison of choreography vs orchestration based Saga patterns in microservices. In 2022 International Conference on Electrical, Computer and Energy Technologies (ICECET) (pp. 1-6). IEEE.

Laigner, R., Kalinowski, M., Diniz, P., Barros, L., Cassino, C., Lemos, M., ... & Zhou, Y. 2020. From a monolithic big data system to a microservices event-driven architecture. In 2020 46th Euromicro conference on software engineering and advanced applications (SEAA) (pp. 213-220). IEEE.

Kreps, J., Narkhede, N., & Rao, J. 2011. Kafka: A distributed messaging system for log processing. In Proceedings of the NetDB (Vol. 11, No. 2011, pp. 1-7).