UNIVERSIDADE FEDERAL DE MINAS GERAIS
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO
SISTEMAS DE INFORMAÇÃO

Padrão de Sagas em Sistemas Distribuídos: Implementação e Análise de Desempenho

DANIEL HENRIQUE TOLEDO SANTOS

BELO HORIZONTE
JANEIRO DE 2025

Padrão de Sagas em Sistemas Distribuídos: Implementação e Análise de Desempenho

Relatório final da disciplina Monografia em Sistemas de Informação II do Bacharelado em Sistemas de Informação da Universidade Federal de Minas Gerais
Orientador: Marco Tulio Valente

BELO HORIZONTE
JANEIRO DE 2025

1. RESUMO/ABSTRACT

Este trabalho apresenta um estudo do padrão de Sagas, uma solução para o problema de atomicidade em transações distribuídas, que se tornou prevalente com a ampla adoção da arquitetura de microsserviços. O estudo inclui uma revisão bibliográfica abrangente, destacando a importância do padrão Saga e sua aplicação em empresas de tecnologia renomadas. O trabalho também inclui um estudo de caso prático, no qual uma saga de reserva de pacote de viagem é implementada e avaliada na plataforma de orquestração Conductor, usando três abordagens distintas: Push, Pull e Pull Decomposto. Os resultados revelam que a abordagem Push supera as outras em termos de desempenho, enquanto a abordagem Pull Decomposto prioriza a manutenibilidade. O trabalho conclui que a escolha da abordagem ideal depende dos requisitos específicos do projeto, enfatizando a necessidade de pesquisas futuras para otimizar a implementação e o desempenho do padrão Saga em arquiteturas de microsserviços.

2. INTRODUÇÃO

O crescente uso de sistemas distribuídos na indústria é uma consequência direta da ampla adoção do paradigma arquitetural de microsserviços, aplicado nos mais diversos domínios. Essa arquitetura oferece vantagens organizacionais significativas, como separação de contextos, autonomia de equipes e responsabilidades independentes. No entanto, também apresenta desafios técnicos complexos, sendo possível destacar dentre estes desafios o da atomicidade de transações que envolvem múltiplos sistemas.

Para solucionar o problema específico de transações que abrangem múltiplos sistemas, as transações distribuídas emergem como uma abordagem eficaz, mitigando as limitações nos processos de persistência de dados. Na literatura, muitos autores atribuem esses desafios à incapacidade dos bancos de dados tradicionais de acompanhar a evolução das aplicações distribuídas. Independentemente disso, a responsabilidade por garantir a integridade das operações de leitura e escrita, antes a cargo dos SGBDs (Sistemas de Gerenciamento de Banco de Dados), agora recai sobre as próprias aplicações.

Apesar da importância da consistência em sistemas distribuídos para inúmeras aplicações críticas na indústria, a pesquisa acadêmica sobre o tema ainda é incipiente. Entre as diversas técnicas para implementar transações distribuídas, este trabalho se concentra em um padrão específico, nomeado Saga. Este é um padrão amplamente adotado na indústria para solucionar o problema de transações atômicas em sistemas distribuídos. Grandes empresas como Uber, Spotify, Netflix, Amazon e Airbnb utilizam esse padrão como base de seus sistemas. Um exemplo amplamente conhecido da aplicação desse padrão é o fluxo de pareamento entre passageiro e motorista para a criação de corridas da Uber, que envolve diversos sistemas e agentes coordenados por uma saga. Contudo, apesar de sua relevância prática, o padrão Saga ainda carece de investigação acadêmica aprofundada.

Este trabalho busca contribuir para essa área por meio de um estudo teórico-experimental, realizando uma revisão bibliográfica dos principais recursos disponíveis, junto de uma etapa prática de estudo e implementando o padrão Saga em uma ferramenta relevante. O objetivo é obter um conhecimento profundo do funcionamento da ferramenta e identificar pontos de melhoria que possam impulsionar a evolução de práticas e ferramentas para transações distribuídas em

larga escala. Isso será alcançado por meio de uma avaliação de desempenho da ferramenta em diferentes cenários de implementação de uma mesma saga.

3. REFERENCIAL TEÓRICO

3.1. Arquitetura de sistemas distribuídos e microsserviços

Com o avanço das tecnologias e o aumento da necessidade de escalabilidade e flexibilidade, a arquitetura de sistemas distribuídos tornou-se padrão para aplicações de grande porte. Nesse modelo, a arquitetura de microsserviços destaca-se por dividir grandes sistemas monolíticos (aplicação única responsável por todo um produto/negócio) em serviços menores e independentes, cada um dedicado a uma parte específica da funcionalidade. Cada serviço é autônomo e se comunica com outros por meio de mensagens (filas e tópicos) ou chamadas (HTTP e gRPC), o que permite uma evolução independente e facilita a divisão do trabalho entre equipes. Esse tipo de arquitetura possibilita um desenvolvimento mais ágil e facilita a implementação de novas funcionalidades. No entanto, a fragmentação dos serviços traz consigo uma grande carga adicional de complexidade técnica, com novos desafios como a consistência de dados e a integridade de operações que antes eram gerenciadas de forma centralizada.

3.2. Problemas de consistência e atomicidade em sistemas distribuídos

Em sistemas centralizados (monolíticos), a consistência de dados e a atomicidade de transações podem ser asseguradas facilmente por meio de um banco de dados transacional, que lida com operações como um conjunto único e indivisível. Em contrapartida, sistemas distribuídos, devido à sua natureza descentralizada e interdependente, requerem abordagens diferenciadas para manter a integridade dos dados e a atomicidade das operações. Quando uma transação se estende por vários serviços ou bancos de dados, a probabilidade de falhas aumenta, e técnicas tradicionais tornam-se insuficientes. Como resultado, assegurar a consistência e a atomicidade em ambientes distribuídos exige técnicas especializadas, como mecanismos de coordenação e protocolos de compensação, para mitigar o impacto de falhas em transações multi-serviços.

3.3. Transações distribuídas

Transações distribuídas são um conjunto de técnicas desenvolvidas para coordenar a execução de transações que envolvem múltiplos serviços ou bancos de dados em sistemas distribuídos. O modelo ACID (Atomicity, Consistency, Isolation,

Durability), tradicionalmente aplicado em bancos de dados centralizados, oferece garantias de atomicidade, consistência, isolamento e durabilidade. No entanto, em um ambiente distribuído, o cumprimento de todos os requisitos ACID se torna desafiador devido à latência da rede, possibilidade de falhas nos serviços e à dificuldade de isolar as operações. Surgem, então, abordagens alternativas, como o modelo BASE (Basically Available, Soft State, Eventually Consistent), que sacrifica a consistência imediata em prol de maior disponibilidade e escalabilidade. Essas soluções viabilizam operações que, embora não sigam rigorosamente o padrão ACID, permitem que sistemas distribuídos ofereçam garantias adequadas para muitas aplicações.

3.4. Padrão de Sagas

O padrão de Sagas é uma estratégia para lidar com transações distribuídas em sistemas de microsserviços, buscando resolver a complexidade de manter consistência e atomicidade em operações que envolvem múltiplos serviços. Diferentemente de uma transação monolítica, que é tratada como um único bloco indivisível, uma saga é composta por uma sequência de transações locais (ou sub-transações) independentes que, juntas, formam uma operação lógica maior. Esse modelo é flexível, pois, em caso de falha em uma das etapas, é possível realizar operações compensatórias nas etapas anteriores para reverter suas ações, restaurando um estado consistente.

No contexto das sagas, existem duas abordagens principais para a coordenação das transações:

- Coreografia: Nesta abordagem, cada serviço que participa da saga é responsável por escutar eventos de outros serviços e reagir a eles conforme necessário. Por exemplo, em um sistema de reserva de viagens, o serviço de reserva de voo pode emitir um evento "reserva realizada" que o serviço de hotel escuta, acionando seu processo de reserva correspondente. Se uma falha ocorrer em algum ponto da sequência, os serviços também executam suas operações de compensação de maneira autônoma. Essa abordagem é descentralizada, favorecendo a independência entre serviços, mas pode levar a uma complexidade maior na manutenção e no rastreamento de falhas.
- Orquestração**:** Diferente da coreografia, a orquestração envolve um serviço centralizado (orquestrador) que coordena a execução das transações na

sequência desejada. O orquestrador gerencia o fluxo de trabalho da saga, chamando cada serviço em uma ordem específica e acionando as operações compensatórias em caso de falhas. Esse modelo torna o gerenciamento mais simples e previsível, pois toda a lógica da saga está contida em um único serviço. No entanto, essa abordagem centralizada pode se tornar um ponto de falha, exigindo cuidados para garantir a robustez e disponibilidade do orquestrador.

Considerações sobre a Implementação de Sagas

A escolha entre Sagas Orquestradas e Coreografadas depende das características do sistema e dos requisitos de negócio. Sagas Orquestradas simplificam a coordenação e o tratamento de falhas, mas podem introduzir um ponto único de falha no orquestrador. Sagas Coreografadas promovem baixo acoplamento, mas exigem maior esforço na gestão da comunicação e na detecção de falhas.

Independentemente da abordagem escolhida, a implementação de Sagas exige atenção a alguns aspectos:

- Idempotência das operações: As transações de compensação devem ser idempotentes, ou seja, sua execução repetida não deve gerar efeitos colaterais indesejados.
- Gerenciamento do estado da transação: É crucial manter o estado da transação para garantir a correta execução das etapas e das transações de compensação.
- Monitoramento e tratamento de falhas: Mecanismos de monitoramento e tratamento de falhas são essenciais para garantir a consistência do sistema em caso de falhas.

Operações compensatórias

O uso de operações compensatórias é um conceito central das sagas, uma vez que permite lidar com falhas de maneira controlada. Uma operação compensatória é, essencialmente, uma ação que reverte ou mitiga os efeitos de uma operação anterior, retornando o sistema a um estado consistente. Por exemplo, se uma transação de débito for realizada como parte de uma saga, uma operação compensatória correspondente pode realizar um crédito reverso em caso de falha em etapas subsequentes. Essas operações requerem planejamento cuidadoso, pois

nem toda transação pode ser compensada facilmente, especialmente em casos que envolvem mudanças de estado irreversíveis ou interações com sistemas externos. É importante ressaltar que o conceito de “estado consistente” pode variar dependendo dos requisitos do sistema e da natureza da transação. Em alguns casos, pode ser aceitável um estado que não seja idêntico ao estado inicial, desde que as informações relevantes sejam preservadas e as dependências entre os serviços sejam respeitadas.

3.5. Exemplos de aplicação do padrão de Sagas

Empresas como Uber, Netflix e Amazon implementam o padrão de Sagas para assegurar a consistência em operações distribuídas de grande escala. No caso da Uber, o padrão de Sagas ajuda a coordenar processos complexos como reserva e confirmação de viagens, onde vários serviços, como pagamento, geolocalização e notificação, precisam trabalhar em harmonia. Já na Netflix, o padrão auxilia em fluxos críticos como provisionamento de infraestrutura e gestão de conteúdo, mantendo a consistência sem comprometer a escalabilidade. A Amazon também utiliza Sagas para orquestrar transações que abrangem diferentes etapas do processo de compra, como gerenciamento de estoque, processamento de pagamentos e logística de entrega.

A adoção do padrão de Sagas em sistemas de microsserviços oferece flexibilidade e resiliência, mas exige uma implementação cuidadosa e uma compreensão profunda do funcionamento e das implicações das operações compensatórias. É fundamental que os desenvolvedores considerem cuidadosamente os cenários de falha e as dependências entre os serviços para projetar operações compensatórias eficazes e garantir a consistência do sistema em caso de erros.

4. REVISÃO BIBLIOGRÁFICA

4.1. Análise do Artigo Seminal e da Tese de Mestrado

Dois trabalhos, em particular, fornecem uma base fundamental para a compreensão e aplicação do Padrão Saga nesta pesquisa: o artigo seminal que o introduziu e uma tese de mestrado que aprofundou a análise do padrão.

Garcia-Molina e Salem (1987), em seu artigo "Sagas", introduziram o conceito de sagas como uma solução para transações distribuídas de longa duração. Os autores propuseram a divisão dessas transações em sub-transações menores, que podem ser intercaladas com outras transações, aumentando a flexibilidade e a eficiência do sistema. A principal contribuição do artigo foi a definição do padrão Saga e a introdução de transações de compensação para reverter os efeitos de sub-transações em caso de falhas, garantindo a consistência do sistema. Este trabalho serve como base conceitual para a pesquisa, fornecendo o fundamento teórico para a aplicação do padrão Saga.

Ioannidis (2023), em sua tese de mestrado "Distributed Transactions using the SAGA pattern", aprofunda a discussão sobre o padrão Saga no contexto de microsserviços, com foco em seus desafios e oportunidades em arquiteturas distribuídas. O autor desenvolveu piSaga, uma biblioteca Java para Spring Boot que facilita a implementação de sagas em microsserviços. Uma contribuição crucial da tese é a exploração detalhada das seis possíveis formas de implementação de sagas, classificadas de acordo com a comunicação (síncrona ou assíncrona), consistência (atômica ou eventual) e coordenação (orquestrada ou coreografada). Essa análise fornece um guia abrangente para desenvolvedores que buscam implementar sagas em diferentes cenários e com diferentes requisitos, e contribui para a compreensão das diferentes nuances e possibilidades de implementação do Padrão Saga.

4.2. Artigos Relacionados Diretamente ao Trabalho

Diversos trabalhos se conectam diretamente ao tema desta pesquisa, fornecendo um panorama abrangente do estado da arte no estudo do Padrão Saga, e contribuindo para a escolha do Conductor como ferramenta de implementação como veremos na seção seguinte.

- Stefanko et al. (2019) apresentaram uma pesquisa de frameworks Saga para transações distribuídas em microsserviços orientados a eventos, comparando diferentes ferramentas em termos de recursos e escalabilidade. O foco principal do estudo foi analisar as funcionalidades e capacidades de cada framework, com ênfase na comparação qualitativa das ferramentas. É importante destacar que este trabalho se diferencia ao avaliar o desempenho do Conductor com base em métricas de recursos computacionais, enquanto o estudo de Stefanko et al. (2019) não aborda o Conductor e se concentra em métricas de tempo de execução.
- Nylund (2023) demonstrou a implementação do padrão Saga em um sistema de e-commerce, com o objetivo de comparar o desempenho da Saga com o protocolo 2PC. O autor avaliou o desempenho da implementação da Saga em um contexto real, fornecendo insights sobre os desafios e benefícios da adoção do padrão. É importante salientar que este trabalho se diferencia ao avaliar o desempenho do Conductor em um sistema real, enquanto Nylund (2023) se concentra na comparação entre Saga e 2PC em um sistema de e-commerce.
- Dürr et al. (2022) forneceram uma avaliação abrangente de diferentes tecnologias para implementação do padrão Saga, com base em critérios como desempenho, escalabilidade e flexibilidade. O estudo realizou uma análise comparativa de diferentes ferramentas, com foco em avaliar suas funcionalidades e capacidades, e serviu como um dos recursos utilizados para a escolha do Conductor neste trabalho. É importante destacar que este trabalho se diferencia ao avaliar o desempenho do Conductor em um sistema real, enquanto Dürr et al. (2022) se concentra em uma comparação mais abrangente de funcionalidades e capacidades entre as ferramentas.
- Koyya (2022) realizou uma pesquisa abrangente de frameworks Saga para transações distribuídas em microsserviços orientados a eventos, incluindo o Conductor, e serviu como outro recurso utilizado na escolha da ferramenta neste trabalho. O autor explorou os desafios de implementar transações distribuídas em microsserviços e forneceu uma análise detalhada das características, vantagens e desvantagens de cada framework, argumentando a favor do Conductor como uma solução robusta e escalável para orquestrar Sagas.

4.3. Artigos Relacionados Indiretamente

Outros trabalhos também se conectam ao tema, de forma mais indireta, explorando diferentes aspectos do padrão Saga e de arquiteturas de microsserviços.

- Tang et al. (2022) investigaram transações ad hoc em aplicações web, destacando a importância da coordenação para garantir a correção em ambientes concorrentes. O estudo destaca os desafios de coordenar operações em diferentes partes do sistema e propõe mecanismos para garantir a consistência dos dados, mesmo em cenários complexos.
- De Heus et al. (2022) exploraram transações em funções serverless, demonstrando a viabilidade de implementar o padrão Saga em arquiteturas serverless. O trabalho destaca os desafios de garantir a atomicidade e a consistência em ambientes serverless, onde a execução das funções é efêmera e a comunicação entre os componentes é limitada.
- Daraghmi et al. (2022) propuseram uma melhoria no padrão Saga para lidar com a falta de isolamento em transações distribuídas, um dos desafios inerentes ao uso de sagas que frequentemente resulta em consistência eventual. Os autores introduziram um mecanismo de cache de quota e um serviço de sincronização de commit para garantir a consistência atômica (ACID), buscando superar as limitações da consistência eventual e garantir a atomicidade das operações distribuídas.
- Malyuga et al. (2022) discutiram a implementação de um orquestrador central de sagas tolerante a falhas em arquitetura RESTful. O trabalho apresenta um modelo para garantir a alta disponibilidade do orquestrador, com foco em otimizar o tempo e os requisitos de memória, e destaca a importância de garantir a tolerância a falhas em arquiteturas distribuídas, onde a falha de um componente pode comprometer todo o sistema.
- Laigner et al. (2021) analisaram o estado da prática, desafios e direções de pesquisa em gerenciamento de dados em microsserviços. O estudo destaca a necessidade de soluções eficientes para garantir a consistência de dados em arquiteturas distribuídas, considerando os desafios de escalabilidade, concorrência e heterogeneidade, e fornece um panorama abrangente dos desafios e das oportunidades na área de gerenciamento de dados em microsserviços.

4.5. Conclusão da revisão bibliográfica

As referências exploradas nesta revisão da literatura fornecem um panorama abrangente do estado da arte em relação ao Padrão Saga, abordando desde seus fundamentos teóricos até suas aplicações práticas em diferentes contextos. Os trabalhos analisados contribuem para a compreensão dos desafios e oportunidades na implementação de Sagas, e fornecem insights valiosos para a escolha de ferramentas e estratégias de implementação.

5. ATIVIDADES DESENVOLVIDAS

Nesta seção aprofundaremos um pouco mais no processo de desenvolvimento deste trabalho, passando por todas as atividades desenvolvidas.

5.1. Definição e estudo

A etapa inicial do projeto consistiu na seleção do tema "Sagas" e na definição do escopo. Optou-se por uma abordagem híbrida, combinando uma revisão bibliográfica abrangente com um estudo de caso, implementando e avaliando o desempenho de uma saga em uma ferramenta de orquestração.

A revisão bibliográfica, resumida na seção anterior, permitiu aprofundar o conhecimento sobre os conceitos de sistemas distribuídos, o padrão Saga e as diversas ferramentas disponíveis para sua implementação. A partir dessa análise, selecionou-se a ferramenta Conductor para a implementação da saga. Os critérios utilizados para sua escolha foram:

- Capacidade de Orquestração: O Conductor permite a definição de fluxos de trabalho complexos, com diferentes tipos de tarefas, como chamadas HTTP, execução de código e interação com filas de mensagens, atendendo às necessidades do projeto.
- Interface Amigável: A interface visual intuitiva do Conductor facilita o desenvolvimento e a monitoração das sagas, agilizando o processo de implementação.
- Escalabilidade e Tolerância a Falhas: O Conductor suporta um grande volume de fluxos de trabalho concorrentes, com mecanismos de tolerância a falhas que garantem a continuidade da execução, mesmo em caso de falhas de hardware ou software.
- Extensibilidade: O Conductor permite a criação de tarefas personalizadas e integrações com diferentes sistemas, tornando-se uma plataforma flexível para orquestrar microsserviços.
- Facilidade de Aprendizado: A curva de aprendizado do Conductor é relativamente baixa, o que facilita sua adoção e o desenvolvimento da saga dentro do prazo do projeto.
- Comunidade Ativa: O Conductor possui uma comunidade ativa que oferece suporte, documentação e recursos, o que facilita a resolução de problemas e dúvidas durante o desenvolvimento.

O Conductor é uma plataforma de orquestração de microsserviços de código aberto que viabiliza a implementação do padrão Sagas seguindo o paradigma orquestrado. Ele foi desenvolvido pela Netflix para lidar com a crescente complexidade de seus fluxos de trabalho de microsserviços e, posteriormente, disponibilizado como um projeto de código aberto.Com o Conductor, é possível definir fluxos de trabalho (workflows), monitorar o progresso da execução e gerenciar a comunicação entre os diferentes serviços envolvidos em uma transação distribuída, tudo isso por meio de uma interface simples. Os workflows são definidos utilizando o formato JSON, que oferece uma sintaxe clara e concisa para especificar as etapas, as dependências e as regras de execução.

Os workflows no Conductor são compostos por uma sequência de tarefas, que podem ser de diferentes tipos, como chamadas HTTP, execução de código, interação com filas de mensagens e integração com outros sistemas. Cada tarefa representa uma unidade de trabalho que precisa ser executada dentro do fluxo de trabalho.Além das tarefas pré-definidas, o Conductor permite a criação de tarefas personalizadas, que podem ser implementadas em diferentes linguagens de programação e integradas ao workflow. Essa flexibilidade permite que os desenvolvedores adaptem o Conductor às suas necessidades específicas.

A orquestração das tarefas no workflow é definida por meio de uma DSL (Linguagem de Domínio Específico) JSON, que permite especificar a ordem de execução das tarefas, as condições de transição entre as etapas, e as regras de tratamento de erros. A DSL JSON oferece uma sintaxe declarativa, que facilita a leitura e a compreensão dos workflows, mesmo para fluxos de trabalho complexos.

O Conductor oferece uma série de vantagens para a implementação de sagas, incluindo simplicidade, robustez, escalabilidade e flexibilidade. Sua interface amigável e recursos de monitoramento por meio de métricas coletadas pela própria ferramenta facilitam o desenvolvimento e a gestão de sagas, enquanto sua arquitetura robusta e escalável garante a confiabilidade e o desempenho, mesmo em aplicações complexas.

Após a definição da ferramenta foi definido o fluxo de negócio a ser simulado: a reserva de um pacote de viagem (hotel + voo). A simplicidade desse fluxo viabiliza a implementação dentro do prazo do trabalho e, ao mesmo tempo, permite aplicar os conceitos-chave do padrão Saga.

Considerando as capacidades e limitações da ferramenta, foram definidos três paradigmas de implementação: um modelo ativo (a ferramenta notifica a aplicação sobre as operações), um modelo passivo (a aplicação busca a operação na ferramenta) e um segundo modelo passivo com abordagem diferenciada que visa melhorar a experiência do desenvolvedor responsável pela manutenção do workflow, dividindo o fluxo de sucesso e falha em dois para melhorar a organização. A representação abstrata da saga é ilustrada no diagrama da figura 1.

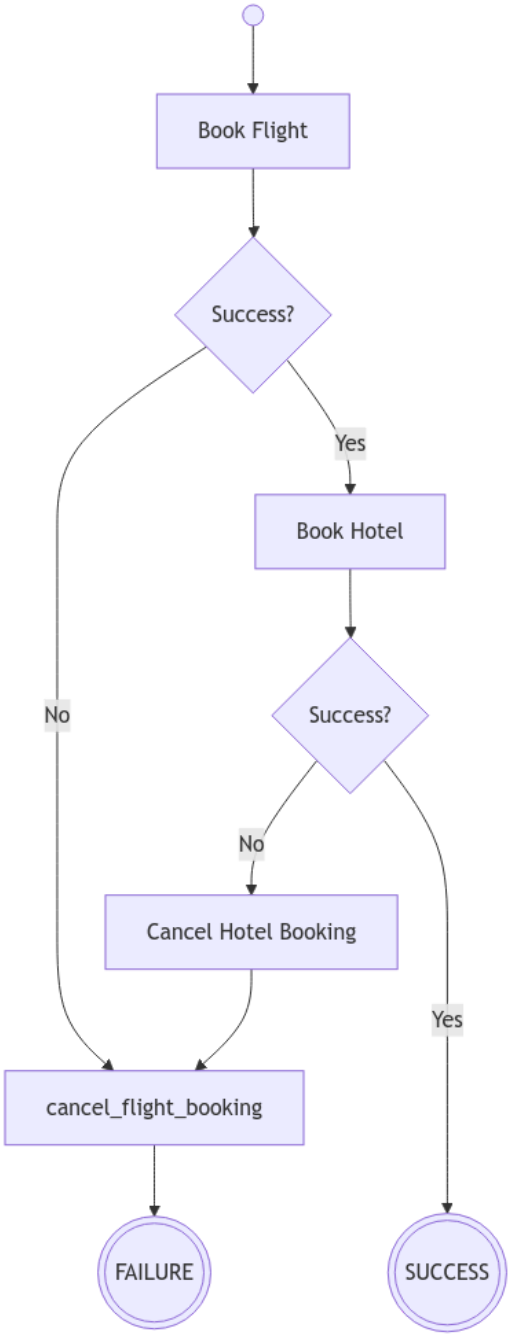

Figura 1

5.2. Implementação

A etapa de implementação teve início após o estudo detalhado do padrão Sagas e da ferramenta Conductor, descrito na subseção anterior. O primeiro passo consistiu na implementação das aplicações que emulam os serviços reais de alocação de voo e hotel.

Duas aplicações foram desenvolvidas em Go, simulando os serviços de reserva de voo e hotel, cada uma com sua própria instância do banco de dados PostgreSQL. Um serviço de configuração inicial, em Python, garante a correta inicialização dos bancos de dados. Todo o sistema foi containerizado usando Docker para facilitar a execução. O código fonte está disponível no repositório GitHub conductor-sagas. A Figura 2 ilustra a arquitetura do sistema.

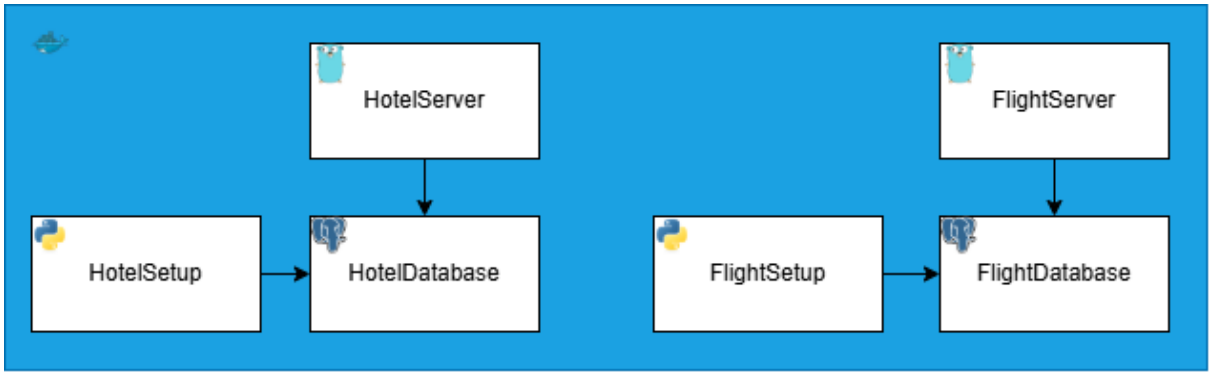

Figura 2

Com os serviços implementados, iniciou-se a etapa de implementação da saga dentro da ferramenta. A mesma saga foi implementada em três abordagens diferentes:

1. Push: Nesta abordagem (Figura 3), a ferramenta notifica o serviço sobre a operação a ser executada por meio de uma chamada HTTP. Em caso de falha, a ferramenta realiza chamadas HTTP para os endpoints de compensação para compensar as operações anteriores. Essa abordagem apresenta desafios técnicos como os citados a seguir que tornaram essa solução menos atraente. Foram os desafios:

- Tratamento de erros HTTP: Para a implementação dessa abordagem foram utilizadas as tarefas disponibilizadas pela própria ferramenta para realização de chamadas HTTP. Porém foi averiguado que em caso de falha na resposta (qualquer status diferente de 2XX) a tarefa responsável pela requisição falha,

o que causa a falha do workflow, que por sua vez faz com que as tarefas subsequentes que seriam responsáveis por fazer a compensação neste cenário não sejam executadas. Para contornar isso, o resultado da tarefa foi marcado como opcional, e na tarefa subsequente, o status da resposta HTTP era verificado para determinar o sucesso ou a falha.

- Conexão com serviços locais: O trabalho foi desenvolvido no ambiente local, e neste cenário foi necessário o uso de artifícios, considerados más práticas, para viabilizar a conexão do servidor da ferramenta aos serviços para execução da lógica desejada. Foi necessário tanto expor os serviços na rede local, fora do padrão de redes virtualizadas do docker, quanto fazer uma alteração na interface de rede do container da própria ferramenta.Reuso de tarefas: Em uma saga é comum que uma operação de compensação seja reutilizada. Pela definição do padrão a compensação da primeira operação faz parte do fluxo de compensação de todas as operações subsequentes, i.e., compensar a operação 2 envolve compensar a 1. Compensar a 3 envolve compensar a 2 que por sua vez transitivamente envolve compensar a 1, e assim por diante.

  No Conductor não existe maneira fácil de fazer esse reuso de definições de operações especificadas no próprio workflow. Com isso se tornou necessário realizar um improviso. A ferramenta disponibiliza dentre suas tarefas pré-prontas uma tarefa de execução de sub-workflow. Usando essa tarefa para chamar sub-workflows que definimos isolando as tarefas que desejamos reutilizar, torna-se possível o reuso. Mas perceba que essa é uma solução improvisada, claramente esse não é o objetivo dessa tarefa específica.

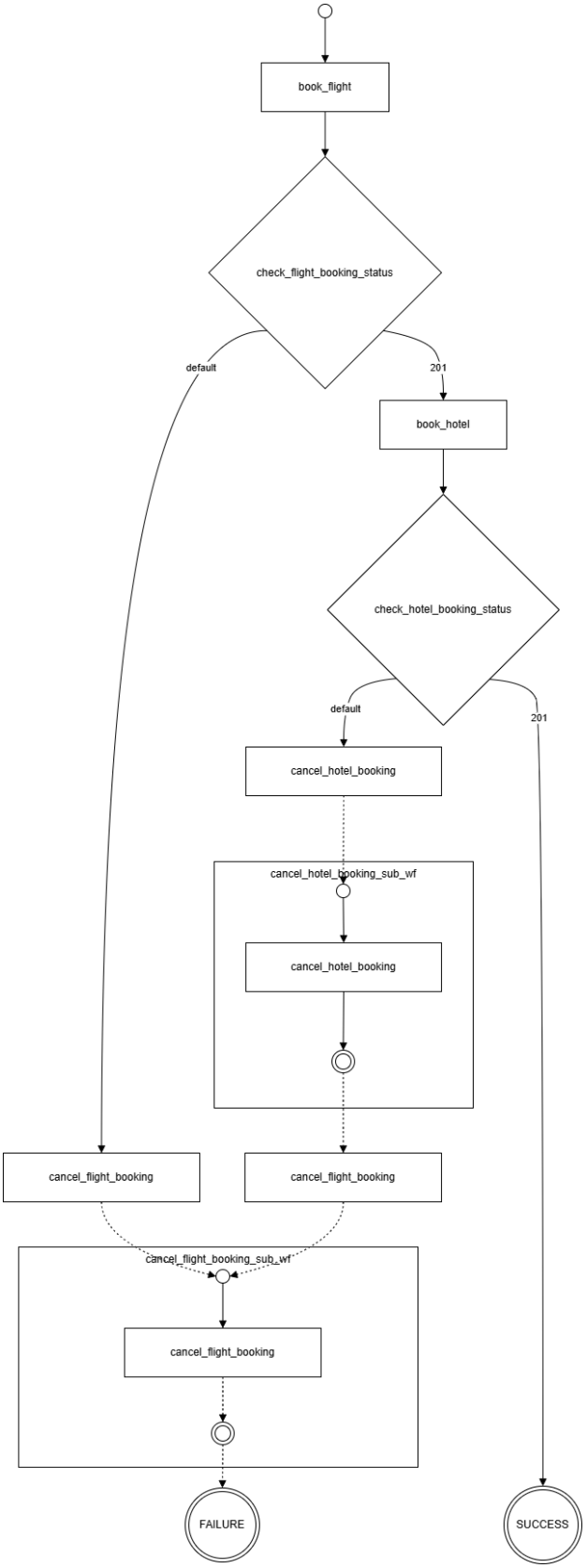
book_flight
check_flight_booking_status
default
201
book_hotel
check_hotel_booking_status
default
201
cancel_hotel_booking
cancel_hotel_booking_sub_wf
cancel_hotel_booking
cancel_flight_booking
cancel_flight_booking
cancel_flight_booking_sub_wf
cancel_flight_booking
FAILURE
SUCCESS

Figura 3

2. Pull: Nesta abordagem (Figura 4), a dinâmica das operações é invertida, o serviço notifica a ferramenta que está disponível para processar um determinado tipo de operação. Para suportar essa nova abordagem foi necessário realizar alguns pequenos ajustes no código dos serviços para passarem a suportar esse novo método de integração. Essa abordagem simplifica a configuração do ambiente e a integração da ferramenta com os serviços, eliminando a necessidade de expor os serviços na rede local e de contornar as limitações de reuso de tarefas. No entanto, introduz um mecanismo de polling por parte do serviço, o que dependendo do cenário pode trazer um aumento no custo de rede relevante.

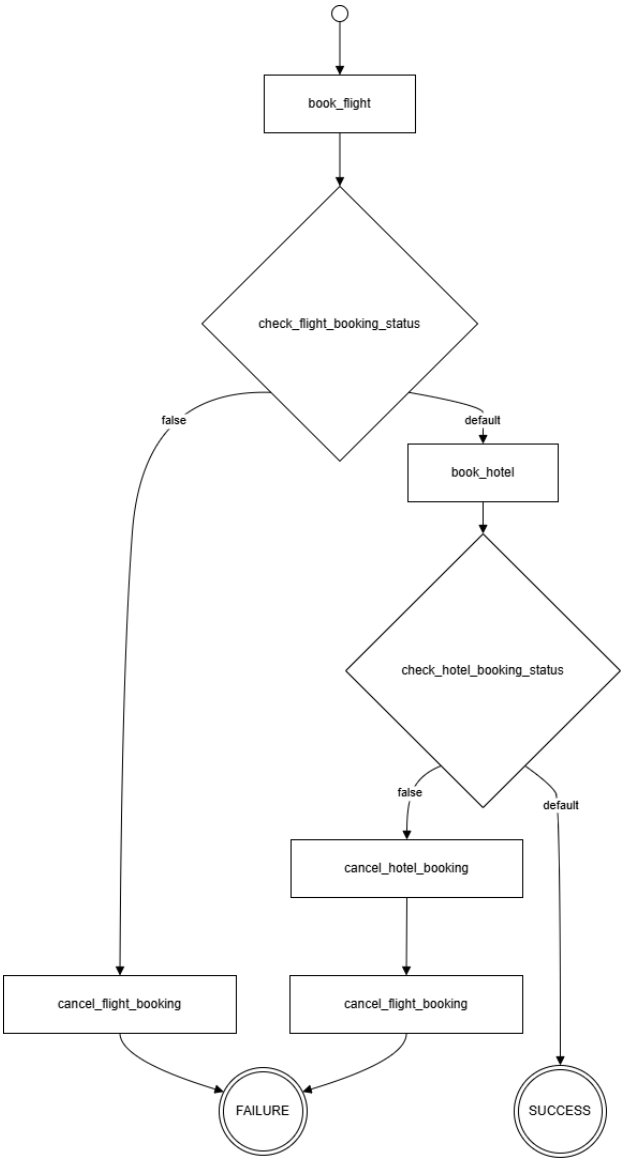

Figura 4

Apesar dos fatores citados anteriormente essa se provou uma abordagem muito boa. A configuração do ambiente e fluxo de trabalho, assim como a integração com os serviços foram incomparavelmente mais fáceis.

Outro ponto muito importante é o fato de que nessa modalidade de comunicação, a decisão por qual atividade desempenhar fica vinculado ao nome da tarefa, que é uma propriedade que pode ser repetida entre diferentes tarefas de um mesmo workflow. Isso em termos práticos nos permite “reutilizar” as operações. “Reutilizar” pois na prática continuam sendo tarefas definidas separadamente, mas desde que possuam o mesmo nome, a ferramenta as trata da mesma forma e assim consequentemente o serviço responsável pela computação da lógica de negócio.

Além de todos esses ganhos, essa abordagem também nos dá controle total sobre o comportamento da tarefa, o que se traduz no controle sobre os dados retornados nessa tarefa. O que por sua vez resolve o problema observado na abordagem anterior de tratamento de erros. Com o controle dos dados retornados nos tornamos capazes de estipular um formato claro de comunicação em casos de falha ou sucesso, facilitando as tomadas de decisão sem necessidade de improviso.

3. Pull decomposto: Essa abordagem faz uso do aprendizado da última abordagem, trazendo um incremento do ponto de vista de manutenibilidade dos workflows dentro da ferramenta. À medida que uma saga cresce no número de operações, o número de tarefas no workflow aumenta exponencialmente, tornando-se cada vez mais complexo, dificultando a extensão e manutenção.

Pensando neste problema foi desenvolvida uma forma de organização das tarefas em workflows separados (Figura 5).

O problema de se estender uma saga está atrelado a composição dos fluxos de compensação, que encadeiam cada vez mais tarefas de compensação à medida que novas operações vão sendo adicionadas, dificultando a manutenção. Esse é um problema incontornável no contexto deste trabalho devido a limitações da própria ferramenta, que por não possuir suporte específico para saga, demanda que está os fluxos de compensação sejam implementados como workflows (sequência de tarefas).

Desse modo, a melhoria encontrada consiste na separação do caminho de sucesso do caminho de falha que dá origem ao termo decomposto. Essa separação nada mais é que a separação do fluxo de sucesso do fluxo de compensações. Com

o isolamento não se resolve o problema da quantidade de tarefas, mas se resolve o problema de todas essas tarefas estarem em um único lugar.

Por fim, com a separação em diferentes workflows, para preservar a definição da saga de ser uma transação atômica, tendo assim um único ponto de entrada, criamos um novo workflow responsável por chamar o fluxo de sucesso, e em caso de falha chamar o de compensação com qual tarefa falhou, para que esse saiba como proceder com as operações corretas.

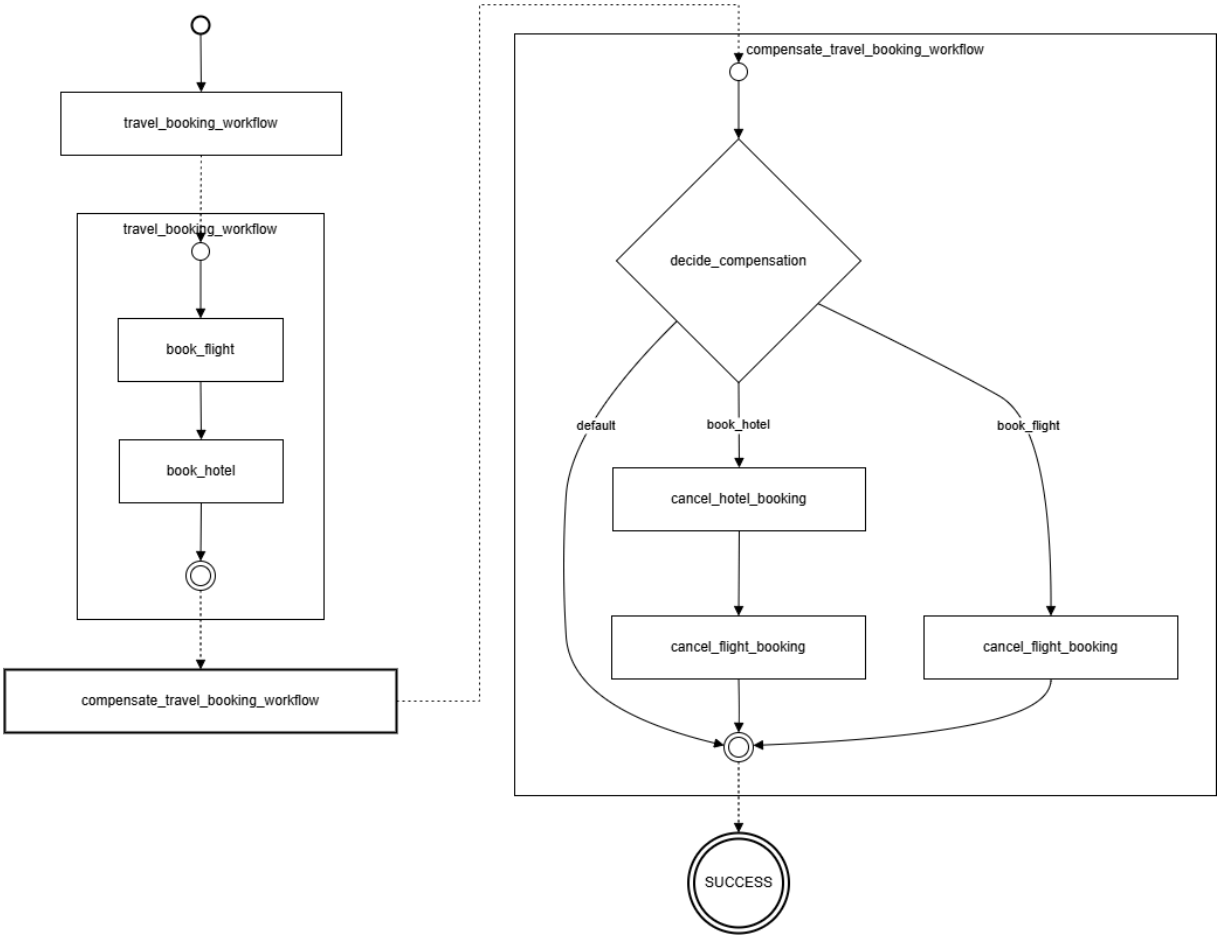

Figura 5

Após a implementação da saga nas 3 abordagens citadas acima foram efetuados alguns testes funcionais, garantindo o comportamento correto das 3 diferentes abordagens, o que permitiu com que o trabalho avançasse para a próxima etapa. Antes porém de seguir para essa próxima etapa, visando facilitar a familiarização do leitor com o projeto, a seguir estão dois exemplos de execução.

- HTTP (Figura 6)

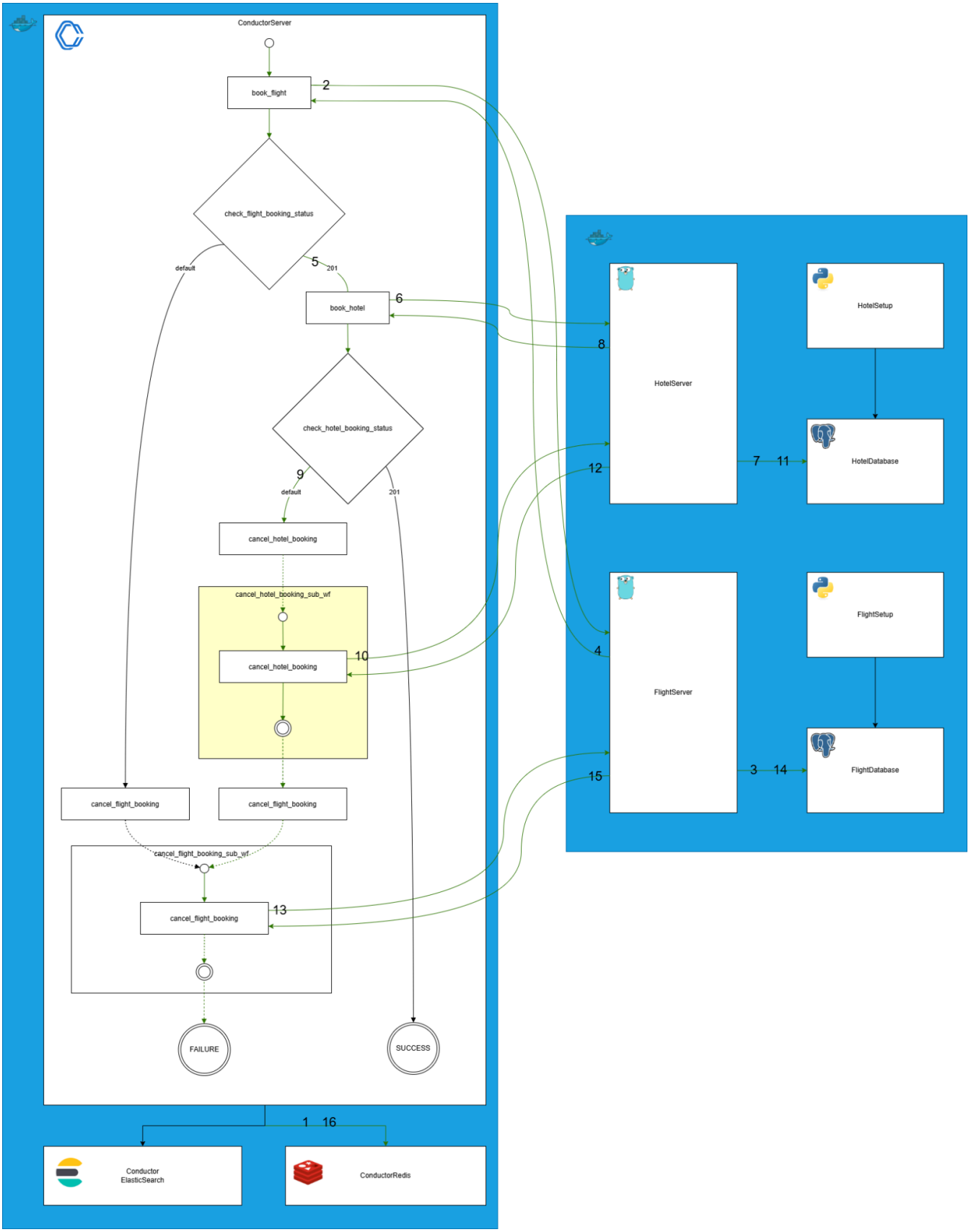

Figura 6

1. Registra o início da execução do fluxo de trabalho no banco de dados (Redis)
2. Requisita a operação de compra de passagem no serviço de voo via HTTP
3. Registra a reserva no banco de dados de passagens
4. Responde com sucesso a requisição de reserva de voo
5. Ao averiguar que a compra de passagem foi um sucesso, segue pelo caminho de sucesso para a reserva de hotel
6. Requisita a operação de reserva de hotel no serviço de hotel via HTTP
7. Gera registros intermediários no banco, i.e., registros do pedido de reserva, mas não da reserva em si pois falha
8. Responde com sucesso a requisição de reserva hotel
9. Ao averiguar que a reserva de hotel falhou, segue pelo caminho de compensação, iniciando pelo fluxo de trabalho de compensação de reservas de hotel para garantir que todo e qualquer estado intermediário possivelmente gerado seja limpo/compensado
10. Requisita a compensação da tentativa de reserva de hotel via HTTP
11. Remove/invalida quaisquer registros referentes a tentativa de reserva de hotel
12. Responde com sucesso a requisição de compensação de reserva de hotel
13. Dentro da execução do fluxo de trabalho de compensação da compra de passagem requisita a compensação da compra de passagem efetuada via HTTP
14. Compensa a compra efetuada anteriormente no banco de dados
15. Responde com sucesso a requisição de compensação de compra de passagem
16. Persiste no banco de dados da ferramenta o resultado final, assim como o caminho percorrido e outras diversas métricas

- Worker (Figura 7)

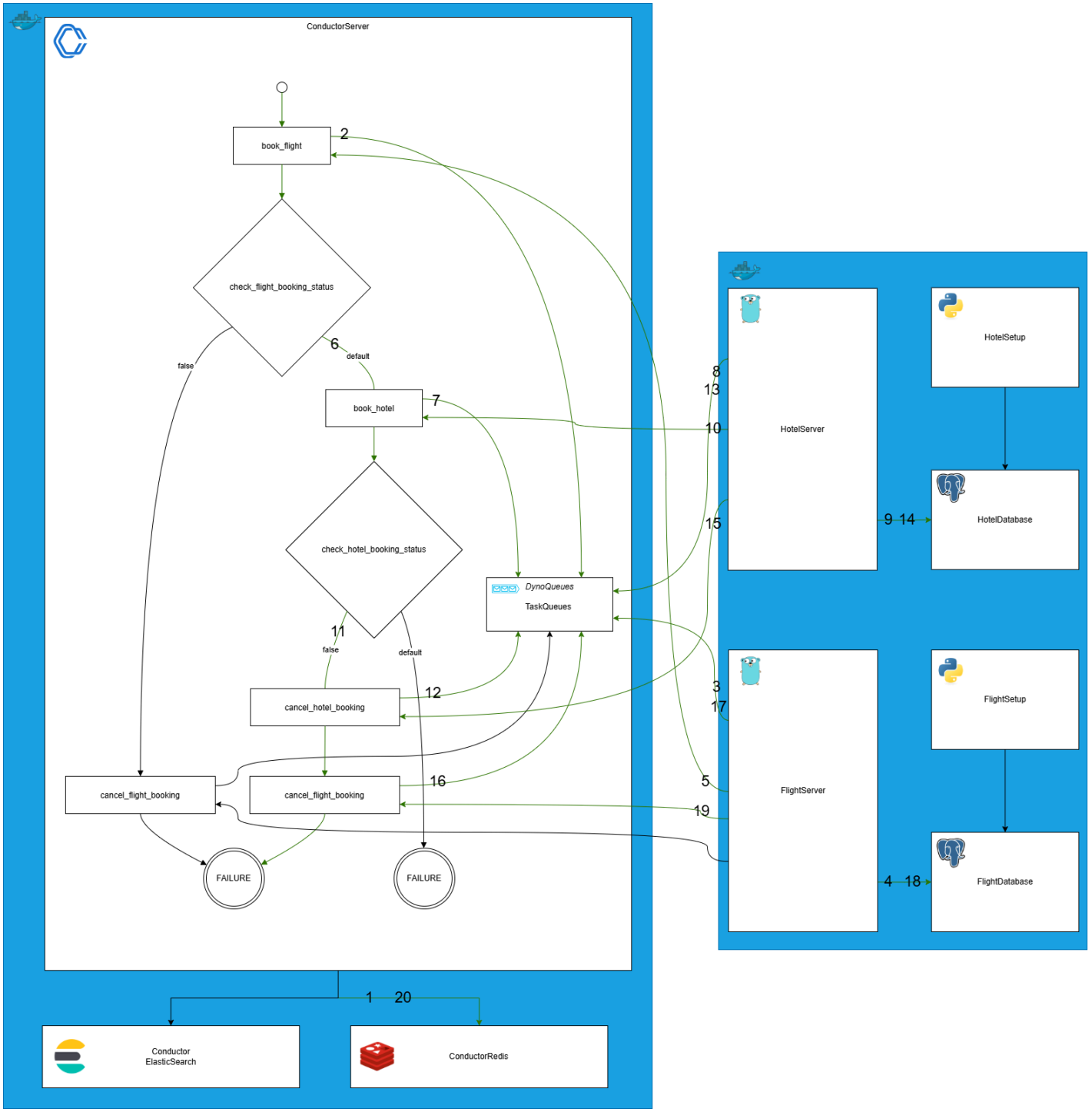

Figura 7

1. Registra o início da execução do fluxo de trabalho no banco de dados (Redis)
2. Adiciona a tarefa de compra de passagem à fila de tarefas. Essa é uma fila interna, específica da aplicação que é persistida no Redis
3. O serviço de voos e passagens quando disponível chama o servidor da ferramenta via HTTP checando por tarefas disponíveis para serem processadas, e recebe a tarefa de compra de passagem
4. Registra a reserva no banco de dados de passagens
5. Notifica a ferramenta do sucesso na execução remota da tarefa, dando prosseguimento a execução do fluxo de trabalho
6. Ao averiguar que a compra de passagem foi um sucesso, segue pelo caminho de sucesso para a reserva de hotel
7. Adiciona a tarefa de reserva de hotel à fila de tarefas
8. O serviço de hotel quando disponível chama o servidor da ferramenta via HTTP checando por tarefas disponíveis para serem processadas, e recebe a tarefa de reserva de hotel
9. Gera registros intermediários no banco, i.e., registros do pedido de reserva, mas não da reserva em si pois falha
10. Notifica a ferramenta da falha na execução remota da tarefa, dando prosseguimento a execução do fluxo de trabalho
11. Ao averiguar que a reserva de hotel falhou, segue pelo caminho de compensação
12. Adiciona a tarefa de compensação de reserva de hotel à fila de tarefas
13. O serviço de hotel chama o servidor da ferramenta via HTTP checando por tarefas para serem processadas, e encontra a tarefa de compensação de reserva de hotel
14. Remove/invalida quaisquer registros referentes a tentativa de reserva de hotel
15. Notifica a ferramenta do sucesso na execução remota da tarefa, dando prosseguimento a execução do fluxo de trabalho
16. Adiciona a tarefa de compensação de compra de passagem à fila de tarefas
17. O serviço de voos e passagens chama o servidor da ferramenta via HTTP checando por tarefas para serem processadas, e encontra a tarefa de compensação de compra de passagem
18. Compensa a compra efetuada anteriormente no banco de dados

19. Notifica a ferramenta do sucesso na execução remota da tarefa, dando prosseguimento a execução do fluxo de trabalho
20. Persiste no banco de dados da ferramenta o resultado final, assim como o caminho percorrido e outras diversas métricas

5.3. Testes de desempenho

Planejamento & Coleta

Com o sistema implementado, restou para este projeto a última etapa do estudo de caso, um teste de desempenho para avaliar as três implementações do padrão Sagas no Conductor, analisando diversas métricas, levantando do ponto de vista de desempenho a melhor solução, assim como possíveis pontos de discussão.

Os testes foram executados em um servidor pessoal com as seguintes especificações:

- Processador: Intel Core i7
- Memória RAM: 32 GB
- Armazenamento: NVMe SK Hynix 512 GB
- Sistema Operacional: Ubuntu 20.04

O script K6 foi executado na mesma máquina, simulando 50 e 500 usuários virtuais (clientes), respectivamente, para os cenários Base e Stress. Cada usuário virtual realizou 10 requisições sequenciais de compra de pacote de viagem, i.e., inicialização de uma saga, sendo 1/5 dessas requisições inválidas para testar a compensação da saga. As ferramentas utilizadas foram:

- Conductor (versão 3.8.3): plataforma de orquestração de workflows, executado em um container Docker.
- K6 (versão 0.40.1): ferramenta de teste de carga para gerar tráfego e simular usuários, executado na mesma máquina do servidor.
- DataDog: plataforma de monitoramento para coletar e visualizar as métricas de desempenho, configurado para coletar métricas do Conductor e serviços.
- Docker: para virtualizar as aplicações e garantir a consistência do ambiente de testes.

As métricas foram coletadas e centralizadas na ferramenta DataDog, utilizando a integração nativa do Conductor e um processo de monitoramento da própria ferramenta na modalidade side-car, i.e., um processo executado separadamente do processo da aplicação que observa o processo da aplicação. No caso deste trabalho, observa os containers e o uso de recursos em cada um. As métricas coletadas foram as seguintes:

- Uso de memória e CPU: utilização da memória e CPU de todos os containers.
- Uso de rede do Conductor.

- Tempo de espera nas filas: tempo que as tarefas esperam nas filas do Conductor.
- Número de workflows iniciados.
- Tempo total de processamento: tempo decorrido entre o início e o fim da execução do workflow.
- Taxa de requisições por segundo (RPS): requisições por segundo recebidas pelas aplicações (Relevante no cenário da abordagem de push).
- Tempo de resposta: tempo de resposta das requisições recebidas pelas aplicações (Relevante no cenário da abordagem de push)

Resultados

Todas as métricas coletadas e os gráficos gerados para sua visualização se encontram disponíveis no repositório do trabalho no GitHub conductor-sagas.

A Tabela 1 apresenta o tempo total de processamento, o uso médio de CPU e o tempo de espera médio nas filas para cada abordagem nos cenários com 500 e 5000 requisições.

Tabela 1: Resultados dos testes de desempenho.

	Push		Pull		Pull Decomposto	
Volume de requisições	500	5000	500	5000	500	5000
Tempo para concluir processamento	2m 30s	22m 15s	2m 00s	29m 35s	5m 00s	45m 05s
Uso médio CPU nos serviços (mcores)	11.1	12.9	25.3	36.0	25.1	35.6
Recebimento médio por rede (KiB)	439	1149	301	918	332	422
Envio médio por rede (KiB)	447	702	434	625	467	573
Tempo de espera médio nas filas	34s	4m 57s	35s	12m 12s	1m 31s	14m 30s

Como podemos observar, a abordagem Push apresentou o melhor desempenho em termos de tempo total de processamento em ambos os cenários, seguida pela Pull e, por último, a Pull Decomposto.

Os resultados indicam que a abordagem Push é a mais eficiente em termos de tempo de execução e uso de CPU, o que contradiz a hipótese inicial de que a abordagem Pull seria mais eficiente por reduzir o overhead de comunicação. Essa diferença pode ser explicada por:

- Overhead de complexidade: O overhead adicionado pela complexidade extra exigida na implementação da abordagem Push pode não ser tão significativo quanto inicialmente previsto, especialmente em comparação com o overhead do mecanismo de polling na abordagem Pull.
- Implementação da abordagem Pull: A implementação da abordagem Pull pode ter alguma ineficiência que impacta o desempenho, como a forma como o polling é realizado. É importante mencionar que existem opções específicas da ferramenta para otimizar essa integração do mecanismo de polling, que podem possivelmente melhorar a performance ao ponto de tornar a comparação viável.
- Overhead da abordagem Pull Decomposto: A abordagem Pull Decomposto, apesar de ter vantagens em termos de organização e manutenabilidade, pode ter um overhead adicional devido à divisão do workflow em dois fluxos distintos, o que pode impactar o tempo total de processamento.

  Além disso, outros fatores podem ter influenciado os resultados:
- Implementação das tarefas: O método Push utiliza a task HTTP nativa da ferramenta, enquanto no Pull a tarefa é implementada pelo usuário, o que pode introduzir diferenças expressivas de desempenho devido a falta de uma otimização extensa da solução do usuário.
- Escalabilidade dos serviços: O fator de escala pode ter pesado contra a abordagem Pull, que pode ter tido desvantagem devido ao fato de servidores HTTP aceitarem mais de uma requisição ao mesmo tempo, enquanto a função do worker é executada em uma única thread de forma sequencial, limitando o throughput da aplicação na abordagem Pull. Esse argumento se fortalece quando observamos as comparações das métricas coletadas no cenário de baixa demanda (500 requisições), onde as métricas da abordagem

Pull são muito semelhantes às da abordagem Push. Nestes casos chegando a ser marginalmente melhores do ponto de vista de uso de rede.

- Uso de CPU: Nos Workers, há um aumento no uso de CPU, o que se deve ao fato de que, nessa abordagem, o serviço é responsável por coordenar a execução, enquanto no HTTP essa responsabilidade é do Conductor.
- A abordagem Pull Decomposto se mostrou a menos eficiente das três, performance muito abaixo de todas as outras. Isso possivelmente é devido à decomposição do processamento em sub workflows, independentemente de sucesso ou falha. Isso adiciona uma camada extra de tarefas entre a requisição do usuário e a computação, que junto a forma do Conductor organizar suas execuções fazendo o uso de filas adiciona um delay em toda computação, uma vez que a chamada de sub workflow é uma tarefa, que é direcionada portanto a uma fila, muito provavelmente ao seu final.

Os resultados obtidos, onde a abordagem HTTP (Push) se mostrou mais eficiente que a Worker (Pull), difere da hipótese inicial de que a comunicação assíncrona da abordagem Pull seria mais performática. Essa divergência levanta questões interessantes sobre a otimização da task HTTP nativa do Conductor e a influência do mecanismo de polling no desempenho da abordagem Pull.

Essa observação corrobora a pesquisa de Štefanko et al. (2019), que, ao analisar diferentes frameworks Saga, concluiu que a escolha da melhor abordagem depende de diversos fatores, incluindo a complexidade do workflow, a frequência de falhas e as características do ambiente de execução. No seu caso, a otimização da task HTTP nativa do Conductor pode ter sido um fator determinante para o melhor desempenho da abordagem Push.

6. CONCLUSÃO

Este trabalho investigou a implementação do padrão Sagas em microsserviços utilizando a plataforma Conductor. Através de um estudo de caso prático, foram implementadas e avaliadas três abordagens: HTTP (Push), Worker (Pull) e Worker Segregado (Pull Decomposto), com o objetivo de analisar o desempenho e a viabilidade de cada uma, tanto do ponto de vista da manutenibilidade quanto do ponto de vista do desempenho.

Os resultados dos testes de desempenho demonstraram que a abordagem HTTP (Push) apresentou o melhor desempenho em termos de tempo de execução e uso de CPU, contradizendo a hipótese inicial de que a abordagem Worker (Pull), com sua comunicação assíncrona, seria mais eficiente. Essa observação levanta questões importantes sobre a influência do mecanismo de polling no desempenho da abordagem Worker e abre espaço para pesquisas futuras que investiguem formas de otimizar as worker tasks no Conductor.

Embora a abordagem Worker Segregado (Pull Decomposto) tenha se mostrado a menos eficiente em termos de desempenho, ela oferece vantagens em termos de organização e manutenibilidade, especialmente em workflows complexos. A divisão do workflow em sub workflows facilita a compreensão e a gestão das etapas da saga, o que pode ser crucial em cenários com um grande número de etapas e compensações.

Com base nesses resultados, a escolha da melhor abordagem para a implementação de sagas no Conductor depende das necessidades e prioridades de cada projeto. Se o foco é o desempenho, a abordagem HTTP (Push) se mostra a mais adequada até o momento. Se a prioridade é a organização e a manutenibilidade, a abordagem Worker Segregado (Pull Decomposto) pode ser a melhor opção, apesar do seu impacto no desempenho. No entanto, com mais pesquisas e otimizações na implementação da abordagem Worker (Pull), ela pode se tornar um ponto de equilíbrio ideal entre desempenho e manutenibilidade, especialmente tendo em vista a performance dessa abordagem nos testes sob baixa carga, onde a perda de performance é muito baixa,o que compensa o grande ganho de manutenibilidade.

É importante destacar que o Conductor, apesar de ser uma plataforma robusta e escalável para orquestração de workflows, não oferece suporte nativo para

o padrão Sagas. Isso exige que o desenvolvedor implemente o padrão "manualmente", o que aumenta a complexidade do desenvolvimento e da manutenção, especialmente em sagas complexas. No entanto, o Conductor possui uma comunidade ativa e uma ampla gama de recursos, como interface amigável, monitoramento detalhado e integração com diversas ferramentas, o que facilita o desenvolvimento e a gestão de workflows.

A revisão bibliográfica realizada neste trabalho evidenciou a importância do padrão Sagas para garantir a consistência de dados em sistemas distribuídos e forneceu um panorama das diferentes abordagens e ferramentas disponíveis para sua implementação. Os resultados da pesquisa contribuem para o debate sobre as melhores práticas na implementação de sagas, demonstrando que a escolha da abordagem ideal depende de fatores como o desempenho, a escalabilidade e a manutenibilidade do sistema.

Este trabalho contribui para a compreensão dos desafios e das oportunidades na implementação do padrão Sagas em microsserviços, fornecendo insights valiosos para a escolha da melhor abordagem e da ferramenta de orquestração.

6.1. Trabalhos futuros

Dando continuidade a esta pesquisa, sugere-se que trabalhos futuros explorem diferentes aspectos da implementação de sagas no Conductor. Seria interessante investigar formas de otimizar as worker tasks, buscando reduzir o overhead do mecanismo de polling e melhorar o desempenho da abordagem Worker (Pull), à tornando mais viável. É importante analisar o impacto do mecanismo de polling no desempenho da abordagem Worker, avaliando diferentes estratégias e suas implicações no consumo de recursos e no tempo de resposta.

A comparação do Conductor com outras ferramentas de orquestração, como Cadence e Eventuate, pode fornecer insights valiosos sobre as vantagens e desvantagens de cada ferramenta. Essa comparação poderia ser feita implementando a mesma saga em diferentes ferramentas e avaliando o desempenho, a escalabilidade e a facilidade de uso de cada uma.

Outro possível ponto de expansão consiste na realização de testes de desempenho no Conductor, porém em um ambiente de cloud, com cenários mais complexos e carga ainda maior. Esses testes permitiriam explorar a arquitetura

distribuída do Conductor e sua capacidade de escalar horizontalmente trabalhando distribuídamente.

7. REFERÊNCIAS BIBLIOGRÁFICAS

1. DARAGHMI, E.; ZHANG, C.-P.; YUAN, S.-M. **Enhancing Saga Pattern for Distributed Transactions within a Microservices Architecture**. Applied Sciences, v. 12, n. 12, p. 6242, jun. 2022.
2. DE HEUS, P.; ROSSI, D.; TORDSSON, J.; VAN HOOF, H. **Transactions in Serverless Functions: a Saga Implementation**. Proceedings of the 17th International Conference on Web Information Systems and Technologies, WEBIST 2021, v. 2, p. 134–141, abr. 2021.
3. DÜRR, M.; LAIGNER, R.; REITER, M.; SCHODER, S.; WIMMER, M. **Saga Technology Evaluation for Distributed Transactions in Microservice Architectures**. 2022 IEEE 16th International Conference on Software Testing, Verification and Validation (ICST), p. 215–226, abr. 2022.
4. GARCIA-MOLINA, H.; SALEM, K. **Sagas**. ACM SIGMOD Record, v. 16, n. 3, p. 249–259, dez. 1987.
5. HAAPAKOSKI, J. **Implementing Asynchronous Sagas in a Microservice Architecture**. set. 2023.
6. IOANNIDIS, P. I. **Distributed Transactions using the SAGA pattern**. Master Thesis – National and Kapodistrian University of Athens, School of Sciences, Department of Informatics and Telecommunications, Athens, mar. 2023.
7. KOYYA, V. S. **A Survey of Saga Frameworks for Distributed Transactions in Event-driven Microservices**. 2022 5th International Conference on Intelligent Computing and Control Systems (ICICCS), p. 1538–1543, maio 2022.
8. LAIGNER, R. et al. **Data management in microservices**. Proceedings of the VLDB Endowment, v. 14, n. 13, p. 3348–3361, set. 2021.
9. MALYUGA, K.; PERL, O.; SLAPOGUZOV, A.; PERL, I. **Fault Tolerant Central Saga Orchestrator in RESTful Architecture**. St. Petersburg: ITMO University, abr. 2020.
10. NYLUND, W. **Distributed Transactions in an E-Commerce System: A Performance Comparison between Saga and Two-Phase Commit (2PC)**. William Nylund, Thesis for the Degree of Master of Engineering, 2023.
11. STEFANKO, P.; VUKOLIĆ, M.; MILOŠEVIĆ, D. **The Saga Pattern in a Reactive Microservices Environment**. 2019 42nd International Convention

on Information and Communication Technology, Electronics and Microelectronics (MIPRO), p. 1387–1392, maio 2019.

12. TANG, W.; CAI, K.; LIAO, Z.; ZHANG, W.; CHEN, F. **Ad hoc Transactional Coordination for Web Applications**. Proceedings of the 43rd International Conference on Software Engineering, ICSE 2021, p. 1573–1584, maio 2021.
13. BILGIN IBRYAM. **Distributed transaction patterns for microservices compared** | Red Hat Developer. Disponível em: https://developers.redhat.com/articles/2021/09/21/distributed-transaction-patterns-microservices-compared#choreography. Acesso em: 28 out. 2024.
14. BLOG, N. T. Netflix Conductor: **A microservices orchestrator**. Disponível em: https://netflixtechblog.com/netflix-conductor-a-microservices-orchestrator-2e8d4771bf40. Acesso em: 28 out. 2024.
15. KONEY, K. K. **Uber Cadence**. Disponível em: https://kktechkaizen.blogspot.com/2021/04/uber-cadence.html. Acesso em: 28 out. 2024.
16. UBER ENGINEERING BLOG. **Conducting Better Business with Uber's Open Source Orchestration Tool, Cadence**, Disponível em: https://www.uber.com/en-BR/blog/open-source-orchestration-tool-cadence-overview/. Acesso em: 28 out. 2024.
17. THOUGHTWORKS. **How to Orchestrate Microservices with DSLs**. Disponível em: https://www.thoughtworks.com/insights/blog/how-orchestrating-microservices-dsls. Acesso em: 28 out. 2024.
18. GITHUB. **conductor-sagas [Repositório do projeto]**. Disponível em: https://github.com/danhenrik/conductor-sagas. Acesso em: 20 jan. 2025.