UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE INFORMÁTICA
GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Hitallo Cavalcanti da Silva

Aplicação de Change Data Capture em Arquiteturas de Microsserviços para Sincronização de Dados em Tempo Real

Recife

2025

Hitallo Cavalcanti da Silva

Aplicação de Change Data Capture em Arquiteturas de Microsserviços para Sincronização de Dados em Tempo Real

Trabalho apresentado ao Programa de Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Bacharel em Ciência da Computação

Área de Concentração: Sistemas Distribuídos

Orientador (a): Carlos Andre Guimarães Ferraz

Recife

2025

Ficha de identificação da obra elaborada pelo autor,
através do programa de geração automática do SIB/UFPE

Silva, Hitallo Cavalcanti da.
Aplicação de Change Data Capture em Arquiteturas de Microsserviços para Sincronização de Dados em Tempo Real / Hitallo Cavalcanti da Silva. - Recife, 2025.
43p : il., tab.

Orientador(a): Carlos Andre Guimaraes Ferraz
Trabalho de Conclusão de Curso (Graduação) - Universidade Federal de Pernambuco, Centro de Informática, Ciências da Computação - Bacharelado, 2025.
Inclui referências.

1. Sistemas Distribuídos. I. Guimaraes Ferraz, Carlos Andre. (Orientação). II. Título.

000 CDD (22.ed.)

HITALLO CAVALCANTI DA SILVA

APLICAÇÃO DE CHANGE DATA CAPTURE EM ARQUITETURAS DE MICROSSERVIÇOS PARA SINCRONIZAÇÃO DE DADOS EM TEMPO REAL

Trabalho apresentado ao Programa de Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Bacharel em Ciência da Computação.

Aprovado em: 05/08/2025

BANCA EXAMINADORA

_____

Prof. Dr. Carlos André Guimarães Ferraz (Orientador)

Universidade Federal de Pernambuco

_____

Prof. Dr. David Júnio Mota Cavalcanti (Examinador Interno)

Universidade Federal de Pernambuco

Dedico a Deus, por ser a razão pela qual tudo existe e todas as coisas são possíveis; à minha família — meus pais, Ezequiel e Niedja, e minha irmã, Hellen — que foram meu porto seguro em toda a jornada estudantil, desde o ensino infantil até a graduação; e à minha esposa, que também é minha família, Ana Beatriz, que me motivou e me incentivou nesta fase final.

AGRADECIMENTOS

Gostaria de agradecer a Deus pela graça de me conduzir até aqui, fortalecendo-me quando já não havia mais forças e sendo o motivo pelo qual realizei tudo até agora — a razão do meu existir.

Agradeço aos meus pais, Ezequiel Luiz e Niedja Cavalcanti, por todo o esforço, dedicação e renúncia dos próprios sonhos em favor da minha educação e bem-estar. Esta dedicatória é para vocês — ainda que seja o mínimo — pois estarão para sempre marcados na minha história.

À minha irmã, Hellen Darly, minha parceira de sonhos, lutas e lágrimas ao longo de toda a minha jornada estudantil, minha sincera gratidão.

Agradeço à minha esposa, Ana Beatriz, por ter caminhado comigo nessa reta final, me incentivado diariamente e trazido a motivação que eu já não tinha para concluir o curso.

Deixar um agradecimento inusitado ao meu cachorrinho Gucci, que sempre esteve disposto a brincar e me animar, mesmo em momentos difíceis.

Agradeço também ao professor Carlos Ferraz por ter aceitado caminhar comigo neste processo como orientador, e a todos os professores do Centro de Informática (CIn), que contribuíram para tornar minha experiência acadêmica a melhor possível — reafirmando o CIn como uma referência nacional e internacional em educação.

RESUMO

A crescente adoção da arquitetura de microsserviços impôs novos desafios à sincronização de dados entre sistemas independentes, especialmente em contextos com bancos de dados heterogêneos. Este trabalho de graduação propõe e valida a aplicação da técnica de *Change Data Capture* (CDC) como solução eficiente e desacoplada para replicação de dados em tempo real entre microsserviços. Para isso, foi implementada uma arquitetura composta por dois serviços independentes: um utilizando PostgreSQL como banco de dados de origem, e outro baseado em MongoDB como banco de destino. A sincronização foi viabilizada por meio do Debezium, que captura alterações diretamente dos logs do PostgreSQL, e do Apache Kafka, responsável pelo transporte dos eventos entre os serviços. Foram realizados testes práticos com diferentes níveis de carga utilizando a ferramenta k6, avaliando dois aspectos principais: a consistência dos dados replicados e a latência de sincronização. Os resultados indicaram uma taxa de consistência de 100% em todos os cenários analisados, com latência média inferior a 600ms, demonstrando a viabilidade da abordagem adotada. Conclui-se que o uso de CDC com ferramentas modernas é uma alternativa robusta, escalável e eficaz para sincronização de dados em arquiteturas de microsserviços.

Palavras-chave: microsserviços. sincronização de dados. change data capture. apache kafka. debezium.

ABSTRACT

The widespread adoption of microservices architecture has introduced new challenges for data synchronization between independent systems, especially when dealing with heterogeneous databases. This undergraduate thesis proposes and evaluates the application of *Change Data Capture* (CDC) as an efficient and decoupled solution for real-time data replication in microservice environments. A functional architecture was implemented using two independent services: one with PostgreSQL as the source database and another with MongoDB as the destination. Synchronization was achieved through Debezium, which captures changes directly from PostgreSQL transaction logs, and Apache Kafka, responsible for event transport between the services. Practical tests were conducted under varying load conditions using the k6 tool, focusing on two main aspects: data consistency and synchronization latency. The results showed a 100% consistency rate across all test scenarios, with average latency below 600ms, demonstrating the feasibility of the proposed solution. It is concluded that CDC combined with modern tools offers a robust, scalable, and effective alternative for real-time data synchronization in distributed microservice architectures.

Keywords: microservices. data synchronization. change data capture. apache kafka. debezium.

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

A crescente adoção da arquitetura de microsserviços tem transformado a forma como sistemas modernos são projetados, desenvolvidos e mantidos. Essa abordagem permite dividir aplicações complexas em serviços menores, independentes e focados em responsabilidades específicas, promovendo maior flexibilidade, escalabilidade e autonomia das equipes de desenvolvimento (FOWLER; LEWIS, 2014). Empresas como Amazon, Netflix e Spotify têm utilizado esse modelo para lidar com desafios de alta escalabilidade e evolução contínua de seus sistemas (Amazon Web Services, 2024).

Um dos princípios fundamentais das arquiteturas de microsserviços é o gerenciamento descentralizado de dados. Cada serviço deve ser responsável por sua própria base de dados, o que, embora promova o desacoplamento e facilite a manutenção, também traz desafios significativos relacionados à consistência e sincronização das informações entre os diversos serviços (VALENTE, 2024). A ausência de um banco de dados compartilhado exige soluções específicas para garantir que diferentes partes do sistema mantenham uma visão coerente dos dados em tempo real.

Diversas estratégias vêm sendo propostas para lidar com esses desafios. Entre elas, destaca-se o padrão Sagas, que utiliza transações distribuídas coordenadas por meio de eventos e compensações. Embora eficaz, esse padrão pode se tornar complexo e custoso em cenários de grande escala, especialmente quando há a necessidade de tratar múltiplas exceções e garantir a reversão de operações em caso de falha (ORACLE, 2024). Em contextos onde há alto volume de alterações e requisitos de latência reduzida, soluções mais diretas e eficientes são necessárias.

Nesse cenário, a técnica de *Change Data Capture* (CDC) surge como uma abordagem promissora para sincronização de dados em tempo real. O CDC permite capturar e propagar alterações feitas em bancos de dados relacionais de forma contínua, assíncrona e com baixo impacto, utilizando mecanismos como leitura de logs transacionais (CONFLUENT, 2024). Essa técnica viabiliza a replicação de dados entre serviços distintos sem a necessidade de acoplamento direto entre eles, contribuindo para arquiteturas mais resilientes e reativas.

Este trabalho de graduação tem como objetivo explorar a aplicação prática do CDC em arquiteturas de microsserviços, implementando um protótipo funcional para sincronização de dados entre dois sistemas independentes. O estudo busca avaliar a viabilidade, os benefícios e os desafios dessa abordagem, utilizando ferramentas como Debezium e Apache Kafka, am-

plamente reconhecidas no ecossistema de dados distribuídos. Além disso, serão conduzidos testes práticos para analisar o comportamento do sistema quanto à consistência dos dados e à latência de replicação, fornecendo uma base concreta para discussões e possíveis melhorias futuras.

A estrutura deste trabalho está organizada da seguinte forma: o Capítulo 2 apresenta a fundamentação teórica sobre os principais conceitos envolvidos, incluindo arquitetura de microsserviços, sincronização de dados e técnicas de CDC. O Capítulo 3 descreve a metodologia adotada, detalhando as etapas de planejamento, implementação e validação. O Capítulo 4 trata do desenvolvimento da solução, abordando a arquitetura proposta, as tecnologias utilizadas e os desafios enfrentados. Em seguida, o Capítulo 5 apresenta a análise dos resultados obtidos nos testes realizados. Por fim, o Capítulo 6 traz as conclusões do trabalho e sugestões para trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

2.1 ARQUITETURA DE MICROSSERVIÇOS

2.1.1 Definição

Nos últimos anos, a arquitetura de microsserviços tem se destacado e se consolidado como uma abordagem amplamente adotada por organizações em todo o mundo. Esse modelo arquitetural surgiu como uma alternativa às arquiteturas monolíticas tradicionais, proporcionando maior flexibilidade, escalabilidade e independência no desenvolvimento e manutenção de sistemas.

A arquitetura de microsserviços pode ser definida como um estilo de desenvolvimento de software no qual uma aplicação é estruturada como um conjunto de pequenos serviços independentes, cada um responsável por uma funcionalidade específica do sistema (NEWMAN, 2022).

Uma das principais características dessa abordagem é a autonomia dos serviços, que são projetados para operar de maneira isolada, possuindo seus próprios processos e gerenciando seus estados. Isso permite que cada microsserviço seja desenvolvido, testado, implantado e escalado de forma independente, sem impactar diretamente os demais componentes da aplicação. Além disso, a comunicação entre esses serviços ocorre por meio de protocolos leves, sendo o HTTP um dos mais utilizados. Outras tecnologias, como chamadas de procedimentos remotos (RPCs) e mensagens assíncronas via filas e eventos, também são empregadas para garantir maior eficiência e desacoplamento (FOWLER, 2017).

A adoção da arquitetura de microsserviços traz diversos benefícios, tais como a possibilidade de utilização de diferentes tecnologias em um mesmo sistema (heterogeneidade tecnológica), maior robustez, escalabilidade e facilidade na implantação, manutenção e evolução das aplicações. Também favorece o alinhamento organizacional e a composição modular do software. Contudo, apresenta desafios, como a complexidade na coordenação das equipes, aumento da sobrecarga tecnológica, custos operacionais e dificuldades relacionadas ao monitoramento e garantia de consistência dos dados distribuídos (NEWMAN, 2022).

2.1.2 Comparação com Arquiteturas Monolíticas

Arquiteturas monolíticas caracterizam-se por aplicações desenvolvidas como uma única unidade de implantação, em que todas as funcionalidades são implantadas e executadas em conjunto (NEWMAN, 2022). Geralmente, essas aplicações incluem uma interface de usuário (executada no cliente, como páginas HTML e JavaScript ou componentes móveis), um banco de dados gerido por um Sistema de Gerenciamento de Banco de Dados (SGBD) e uma aplicação servidor responsável pela lógica de negócios e manipulação dos dados. Essa configuração forma um monólito, com funcionalidades fortemente acopladas e interdependentes (FOWLER, 2017).

De acordo com (AMAZON, 2025) e (NEWMAN, 2022), aplicações monolíticas facilitam o desenvolvimento inicial, pois seus módulos são integrados e reutilizados de maneira simples. Entretanto, a falta de boas práticas pode acarretar dificuldades nas entregas, já que múltiplos times trabalhando nas mesmas funcionalidades podem afetar a estabilidade do sistema. Cada nova funcionalidade impacta o sistema como um todo, exigindo retestes completos para garantir o funcionamento adequado (FOWLER, 2017).

Por outro lado, a arquitetura de microsserviços facilita a escalabilidade, permitindo que apenas partes específicas do sistema sejam expandidas. Nas arquiteturas monolíticas, toda a aplicação deve ser escalada, o que pode gerar custos desnecessários. Monólitos ainda são vantajosos em sistemas pequenos, com poucos usuários e sem grandes perspectivas de crescimento. Para cenários que exigem escalabilidade complexa, múltiplos times de desenvolvimento e evolução a longo prazo, recomenda-se a adoção de microsserviços (NEWMAN, 2022).

2.1.3 Comunicação entre Microsserviços

A definição adequada da comunicação entre microsserviços é fundamental para garantir o correto funcionamento do sistema e atingir os objetivos de negócio. A escolha da abordagem influencia desempenho, escalabilidade, desacoplamento e confiabilidade.

Entre as principais estratégias de comunicação destacam-se: chamadas de procedimentos remotos (RPCs), comunicação baseada em REST e uso de sistemas de mensagens. Os RPCs permitem a invocação de métodos em serviços remotos como se fossem locais, abstraindo a complexidade da comunicação subjacente. Tecnologias como gRPC e SOAP implementam esse conceito, exigindo contratos bem definidos entre serviços (NEWMAN, 2022). O REST

(Representational State Transfer) é um estilo arquitetural amplamente utilizado, baseado em troca de mensagens via HTTP, em que recursos do sistema são expostos através de endpoints acessíveis por verbos padrão (GET, POST, PUT, DELETE). As mensagens geralmente usam JSON, pela simplicidade e ampla aceitação. Já os sistemas de mensageria implementam comunicação assíncrona, utilizando filas (modelo ponto a ponto) ou tópicos (publicação e assinatura), permitindo múltiplos consumidores de uma mesma mensagem.

Segundo (NEWMAN, 2022), destacam-se cinco padrões principais: comunicação síncrona bloqueante, em que o microsserviço faz uma chamada e aguarda resposta antes de continuar o processamento, típica em REST; comunicação assíncrona não bloqueante, em que o microsserviço continua o processamento sem aguardar resposta, aumentando eficiência; requisição-resposta, em que há envio de requisição com espera por resposta, podendo ser síncrono ou assíncrono; comunicação orientada a eventos, em que microsserviços publicam eventos consumidos por outros sem conhecer os consumidores, promovendo alto desacoplamento; e, por fim, compartilhamento de dados em comum, via fonte centralizada, menos comum e com desafios de concorrência e consistência. A escolha do padrão depende dos requisitos de latência, acoplamento, escalabilidade e confiabilidade, e muitas vezes combinações são utilizadas para otimizar o sistema.

2.1.4 Vantagens e Desafios

Em *Microservices in Practice: A Survey Study* (VIGGIATO et al., 2018), foram identificadas vantagens e desafios baseados em comentários de Stack Overflow, GitHub e Reddit, conforme as definições de (NEWMAN, 2022). Entre as vantagens destacam-se implantações independentes, escalabilidade facilitada, maior manutenibilidade e possibilidade de uso de tecnologias variadas. Os principais desafios citados foram a complexidade nas transações distribuídas, a dificuldade em testes integrados, o risco de falhas nos serviços e o custo elevado das chamadas remotas, o que indica a necessidade de planejamento cuidadoso para que os benefícios superem os desafios.

2.2 SINCRONIZAÇÃO DE DADOS

Ambientes digitais atuais são cada vez mais distribuídos, com grande volume de informações geradas por diversos dispositivos que compartilham dados continuamente. Usuários

utilizam serviços que demandam atualização constante, com dados trafegando globalmente pela rede (CHINA; GOODWIN, 2024). Nesse contexto, a sincronização de dados é o processo que garante a atualização e a consistência entre dispositivos e sistemas (CHEN, 2024). Para operar com eficácia e integridade, é essencial que os dados sejam precisos, atualizados e uniformes (CHINA; GOODWIN, 2024). Ferramentas especializadas automatizam esse processo, assegurando reconciliação e integridade contínua.

As estratégias de sincronização variam conforme o modelo de dados adotado e podem ser classificadas em dois grandes grupos: modelos direcionais e modelos temporais (CHINA; GOODWIN, 2024). Nos modelos direcionais, a sincronização unidirecional ocorre quando os dados são propagados de um sistema de origem para destino sem alterações no sistema de origem, funcionando como uma cópia. Já a sincronização bidirecional permite que modificações feitas tanto no sistema de origem quanto no de destino sejam propagadas entre ambos, exigindo monitoramento contínuo. A sincronização multidirecional acontece quando todos os sistemas da rede atuam como fontes de dados, refletindo alterações entre si para manter atualizações consistentes. Por fim, a sincronização híbrida é aplicada em ambientes de computação híbrida, permitindo o intercâmbio de dados entre diferentes contextos, como nuvens públicas e privadas (CHINA; GOODWIN, 2024). Nos modelos temporais, a sincronização pode ser realizada em tempo real ou em lote. Na primeira, as atualizações são enviadas simultaneamente para todos os dispositivos e para o armazenamento primário, garantindo consistência imediata (CHEN, 2024). Já a sincronização em lote concentra as alterações e as envia em blocos ao longo do tempo, resultando em um custo-benefício superior em determinados cenários, especialmente quando não há necessidade de atualização contínua (CHEN, 2024).

Entre as principais técnicas de sincronização estão o *Full Sync*, o *Incremental Sync*, o *Change Data Capture* (CDC), o *Pull-based Sync*, o *Push-based Sync* e o *Event-based Sync* (AIRBYTE, 2023). Cada técnica apresenta vantagens e desafios conforme o volume de dados, a frequência de atualização e os requisitos de consistência. Este trabalho aborda o *Change Data Capture* (CDC), técnica amplamente utilizada para rastrear modificações em bancos de dados de forma eficiente (KLEPPMANN, 2017).

2.3 CHANGE DATA CAPTURE (CDC)

O Change Data Capture (CDC) é uma solução para sincronização de dados em tempo real, capturando e extraindo alterações realizadas em bancos de dados para replicação em outros

sistemas (KLEPPMANN, 2017). Modificações podem ser propagadas a índices de busca ou a sistemas que demandam dados atualizados, garantindo consistência e integridade distribuída. Ferramentas de CDC monitoram continuamente os sistemas de origem para capturar e registrar alterações, que podem ser armazenadas em bancos auxiliares, arquivos de log estruturados ou encaminhadas a sistemas de middleware, assegurando propagação eficiente (SHARMA, 2023).

As abordagens para implementação de CDC podem ser classificadas em três tipos principais. A primeira é a *timestamp-based*, que utiliza campos de data e hora para identificar alterações, consultando periodicamente registros modificados após determinado momento. Embora seja simples, pode gerar sobrecarga em cenários de alta concorrência (TOBIN, 2021). A segunda é a *trigger-based*, que emprega gatilhos no banco de dados, acionados automaticamente em eventos como inserções ou atualizações. Essa técnica captura mudanças em tempo real, mas pode impactar o desempenho e aumentar a complexidade em bases de grande porte (SHARMA, 2023; TOBIN, 2021). Por fim, a abordagem *log-based* baseia-se na leitura dos logs nativos do banco, que registram todas as operações executadas. Trata-se de uma solução eficiente, assíncrona e com baixo impacto no desempenho transacional, embora dependa do suporte do banco e da compatibilidade dos logs (SHARMA, 2023; TOBIN, 2021).

Os principais benefícios do CDC incluem a captura e propagação quase instantânea das alterações, permitindo decisões ágeis em sistemas distribuídos (SHARMA, 2023). Além disso, possibilita a integração eficiente entre sistemas de origem e destino, com conversão dos dados para formatos compatíveis, e contribui para processos de migração, mantendo a consistência sem exigir mudanças na estrutura original (SHARMA, 2023). Entre as ferramentas mais utilizadas destacam-se o Debezium e o Apache Kafka. O Debezium é uma plataforma distribuída que realiza CDC em tempo real, utilizando logs de transação para identificar alterações, eliminando consultas frequentes e reduzindo sobrecarga. Ele oferece suporte a diversos bancos de dados, como MySQL, PostgreSQL, MongoDB e SQL Server (DEBEZIUM, 2025). O Apache Kafka, por sua vez, é uma plataforma distribuída para processamento de fluxos de eventos em tempo real, permitindo publicação, consumo, armazenamento e processamento por meio de tópicos (KAFKA, 2025). Trabalhando em conjunto, Kafka e Debezium garantem propagação eficiente e confiável das alterações (DEBEZIUM, 2025).

3 METODOLOGIA

Para solucionar o problema de sincronização de dados entre microsserviços distintos, foi conduzida uma pesquisa de natureza aplicada e abordagem experimental. O trabalho consistiu na implementação de uma arquitetura orientada a microsserviços, de forma simples e objetiva, incorporando a técnica de Change Data Capture (CDC). A abordagem utilizada para captura das alterações foi baseada em logs (*log-based*), possibilitando a sincronização em tempo real entre dois sistemas distintos que utilizam bases de dados heterogêneas.

3.1 TIPO DE PESQUISA

A pesquisa pode ser classificada da seguinte forma: aplicada, por buscar solucionar problemas específicos por meio da aplicação prática do conhecimento (GIL, 2002), e experimental, por envolver a manipulação de variáveis em ambientes controlados para observação dos efeitos e resultados (GIL, 2002).

3.2 ETAPAS DO TRABALHO

O desenvolvimento do trabalho foi dividido em etapas práticas e objetivas, sempre com foco em validar uma solução funcional para o problema de sincronização de dados entre microsserviços. As etapas realizadas foram: estudo teórico sobre conceitos fundamentais como microsserviços, sincronização de dados e técnicas de CDC, com o objetivo de embasar tecnicamente as decisões de implementação; definição da arquitetura da solução, planejada com dois microsserviços independentes, mensageria via Apache Kafka e utilização do Debezium para captura de alterações em banco de dados relacional; escolha das ferramentas, baseada em critérios de simplicidade, compatibilidade com CDC e familiaridade do autor com tecnologias como *Micronaut*, *Docker*, *Kafka*, bancos relacionais e NoSQL, garantindo agilidade no desenvolvimento e menor curva de aprendizado; configuração da infraestrutura utilizando Docker e Docker Compose para orquestrar os serviços de banco de dados (PostgreSQL e MongoDB), mensageria (Kafka e Zookeeper) e captura de dados (Debezium); desenvolvimento dos microsserviços com Micronaut, framework leve e eficiente que facilita a construção de APIs REST e o consumo de eventos de forma reativa; integração com Debezium, que

monitora o log de transações do PostgreSQL e publica automaticamente os eventos no Kafka sempre que uma alteração ocorre na base de dados; consumo dos eventos por um segundo microsserviço, responsável por ler os eventos do Kafka e inserir os dados transformados no banco de destino (MongoDB); e, por fim, validação da sincronização dos dados, realizada em duas etapas: (i) extração manual dos dados das bases PostgreSQL e MongoDB, seguida de comparação automatizada no Google Colab para verificar a replicação; e (ii) testes de carga utilizando a ferramenta *k6*, simulando inserções em massa na API para avaliar a performance e a integridade da replicação.

3.3 FERRAMENTAS E TECNOLOGIAS UTILIZADAS

Para o desenvolvimento da solução foram escolhidas ferramentas que combinam eficiência, facilidade de uso e familiaridade do autor, garantindo agilidade e qualidade na implementação. As principais tecnologias utilizadas foram: Micronaut, framework leve para desenvolvimento dos microsserviços, que facilita a criação de APIs REST e o consumo reativo de eventos; PostgreSQL, banco de dados relacional utilizado como sistema de origem dos dados; MongoDB, banco de dados NoSQL escolhido como sistema de destino para armazenar os dados sincronizados; Apache Kafka, plataforma de mensageria usada para transporte dos eventos capturados; Debezium, ferramenta de CDC baseada em logs, que monitora o PostgreSQL e publica eventos no Kafka; Docker e Docker Compose, para containerização e orquestração dos serviços, facilitando a montagem e o gerenciamento do ambiente; k6, ferramenta de teste de carga para validar a performance e a robustez da arquitetura; e Google Colab, utilizado para análise e comparação dos dados exportados das bases.

3.4 CRITÉRIOS DE VALIDAÇÃO

A validação da solução foi feita para garantir que a sincronização dos dados entre os microsserviços ocorresse de forma correta e eficiente. Foram adotados os seguintes critérios: os dados inseridos ou atualizados no banco PostgreSQL deveriam ser refletidos corretamente no MongoDB, mantendo a integridade e consistência; a comparação entre os dados exportados das duas bases deveria apresentar alta consistência, com divergências mínimas ou inexistentes.

4 DESENVOLVIMENTO DA SOLUÇÃO

Este capítulo apresenta a solução prática para permitir a sincronização de dados entre microsserviços distintos, de forma automatizada e em tempo quase real. A proposta busca garantir que os dados permaneçam consistentes entre diferentes serviços sem a necessidade de acoplamento direto entre eles.

4.1 VISÃO GERAL DA ARQUITETURA

Para atingir o objetivo principal deste trabalho, foi planejada uma arquitetura simples composta por dois microsserviços distintos, cada um operando com sua própria base de dados, mas manipulando uma mesma fonte de informação: registros de transações bancárias.

O primeiro microsserviço, denominado `transactions-api`, é responsável por interagir com o banco de dados PostgreSQL chamado `transactions-db`. A esse banco, é conectado um conector do Debezium (DEBEZIUM, 2025), que monitora as alterações registradas nos logs de transação. Sempre que ocorre uma modificação nos dados, o Debezium converte essa alteração em uma mensagem e a publica em um tópico do Apache Kafka (KAFKA, 2025). A Figura 1 ilustra esse fluxo de comunicação entre os componentes descritos.

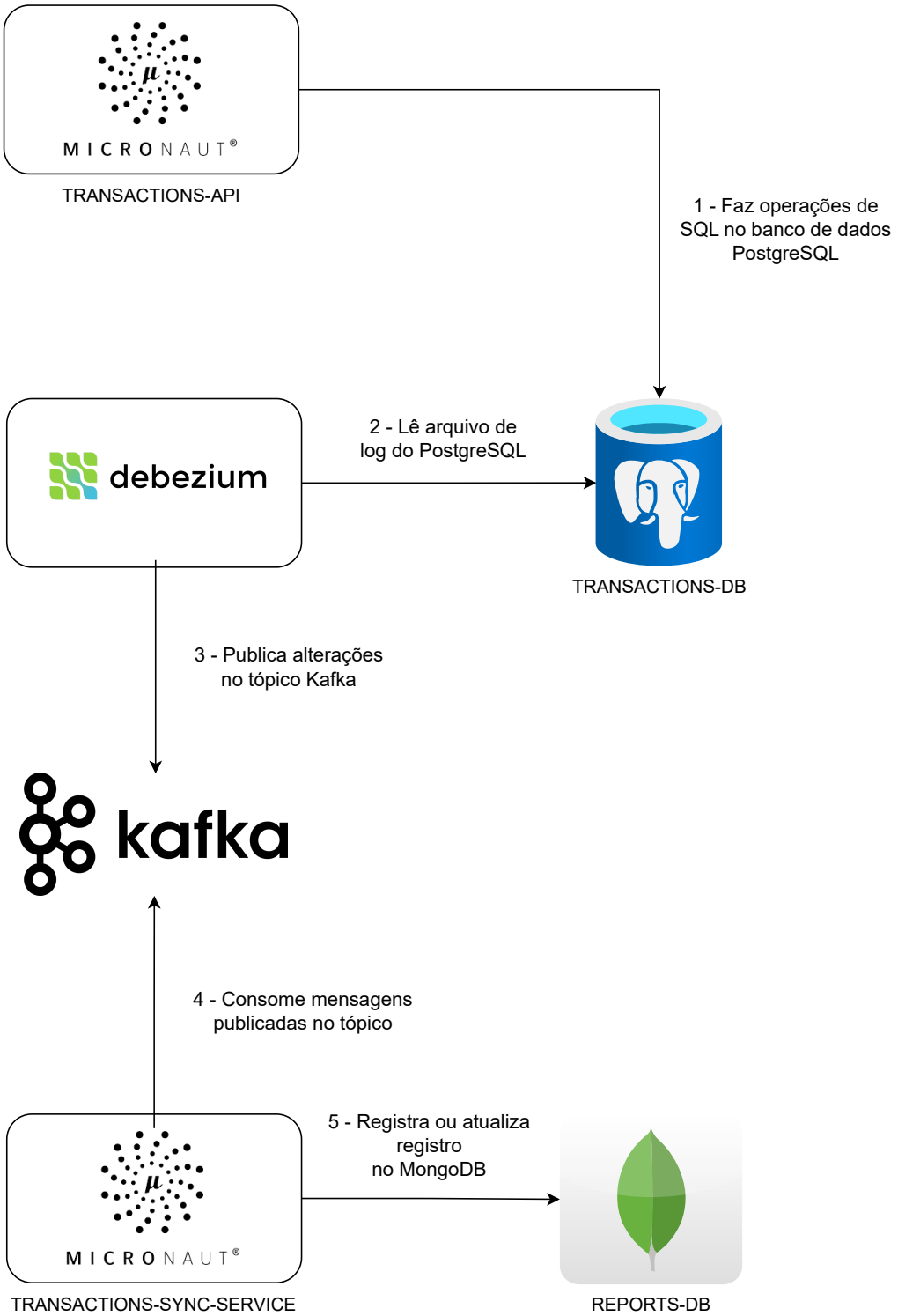

Figura 1 – Visão geral da solução proposta, utilizando CDC com Debezium e Kafka.

O segundo microsserviço, chamado `transactions-sync-service`, é consumidor desse tópico Kafka. Ele processa as mensagens publicadas e persiste os dados transformados em uma base de dados MongoDB voltada para relatórios, denominada `reports-db`.

Essa arquitetura demonstra, de forma prática, como é possível sincronizar dados entre microsserviços heterogêneos utilizando a abordagem de Change Data Capture (CDC) com o Debezium e o Kafka.

4.2 IMPLEMENTAÇÃO

Para dar suporte à solução proposta de sincronização de dados entre microsserviços, foi montada uma infraestrutura com o uso do **Docker** e do **Docker Compose**, que facilitou a criação e o gerenciamento dos serviços necessários para a comunicação e replicação dos dados.

Os containers definidos nessa infraestrutura foram responsáveis por subir os seguintes serviços principais:

- **PostgreSQL (transactions-db)**: banco relacional onde ocorrem as alterações monitoradas;
- **MongoDB (reports-db)**: banco NoSQL que recebe os dados sincronizados;
- **Apache Kafka e Zookeeper**: usados como camada de mensageria entre os microsserviços;
- **Debezium**: ferramenta que captura alterações no PostgreSQL e envia os eventos para o Kafka;
- **Mongo Express**: interface para visualizar os dados presentes no MongoDB.

A configuração de cada um desses serviços foi feita de forma modular, com arquivos separados e organizados por propósito. Esses arquivos estão disponíveis em um repositório público criado pelo autor[1]. Os principais são:

- `transactions-db-docker-compose.yaml`: configuração do banco de origem (PostgreSQL);
- `reports-db-docker-compose.yaml`: definição do banco de destino (MongoDB);

[1] <https://github.com/hitallocavas/cdc-environment>

- `transactions-kafka-zookeper-docker-compose.yaml`: configuração dos serviços de mensageria;
- `transactions-debezium-docker-compose.yaml`: inicialização do Debezium;
- `mongo-db-express-docker-compose.yaml`: definição do Mongo Express para visualização dos dados;
- `register-connector.sh`: script utilizado para registrar o conector Debezium no banco PostgreSQL.

Já os microsserviços implementados foram executados diretamente no ambiente local por meio do comando nohup, permitindo que rodassem em segundo plano de forma simples e independente da orquestração por containers.

4.3 DESENVOLVIMENTO DOS MICROSSERVIÇOS E INTEGRAÇÃO

Os microsserviços foram implementados utilizando o framework **Micronaut**, que oferece integração nativa com o Apache Kafka por meio do seu cliente oficial. Neste projeto, foram utilizadas as configurações *default* fornecidas pelo cliente Kafka do Micronaut, o que simplificou a implementação e garantiu uma comunicação eficiente entre os serviços.

O serviço `transactions-api` expõe uma API simples para manipulação dos registros no banco de dados PostgreSQL, enquanto o `transactions-sync-service` atua como consumidor dos eventos publicados no Kafka.

Essa abordagem permitiu a sincronização dos dados entre os serviços de forma desacoplada, escalável e próxima do tempo real, com mínimo esforço de configuração adicional.

4.4 AVALIAÇÃO DA SOLUÇÃO

Para verificar o correto funcionamento e o desempenho da solução desenvolvida, foram realizados testes tanto funcionais quanto de carga.

Nos testes funcionais, foram feitas operações de inserção, atualização e exclusão de registros por meio do microsserviço `transactions-api`. As alterações foram validadas manualmente na base de dados MongoDB, acessada via Mongo Express, garantindo que os dados fossem sincronizados corretamente e refletidos na base de relatórios.

Além disso, foi utilizada a ferramenta **k6** para simular requisições simultâneas à `transactions-api`, avaliando a performance e o comportamento do sistema sob carga. O script de teste simulava múltiplos usuários realizando inserções em paralelo, permitindo observar a capacidade da arquitetura em processar eventos e manter a sincronização dos dados em tempo próximo do real.

Os códigos dos cenários de teste, bem como as imagens dos resultados obtidos para cada um deles, estão disponíveis no repositório GitHub: <https://github.com/hitallocavas/cdc-tests-k6>. Este repositório contém o detalhamento dos scripts utilizados no k6 e os gráficos que ilustram o desempenho e a latência do sistema durante os testes, facilitando a reprodução e análise dos resultados.

Para validar especificamente a consistência e a latência da replicação dos dados entre as bases, foi utilizado um notebook no Google Colab, cujos códigos e dados extraídos estão disponíveis no repositório: <https://github.com/hitallocavas/cdc-results>. Este repositório permite a reprodução dos testes de validação da consistência e análise detalhada dos tempos de replicação observados.

4.5 CONSIDERAÇÕES FINAIS

Este capítulo apresentou a solução proposta, cobrindo desde a arquitetura, infraestrutura e implementação dos microsserviços até os testes realizados. As tecnologias escolhidas mostraram-se adequadas para o objetivo de sincronização de dados em tempo quase real, e a validação prática confirmou a eficácia da abordagem adotada. A experiência demonstrou que é possível aplicar Change Data Capture com ferramentas modernas de forma eficiente em um cenário realista.

5 AVALIAÇÃO

Este capítulo apresenta os resultados obtidos por meio da execução dos testes realizados sobre a arquitetura proposta. O objetivo foi validar a eficácia da técnica de Change Data Capture (CDC) aplicada, com foco em Dois objetivos: (1) A replicação em ação e (2) o impacto da replicação do desempenho do sistema.

Para isso, duas métricas: (1) a consistência que é manter os mesmos dados nos dois banco de dados e (2) a latência do processo de sincronização que é o tempo decorrido para que o dados sejam sincronizados.

Os testes foram organizados em quatro cenários distintos, com variações na carga de requisições simuladas. Os dados replicados entre o banco de origem (`transactions-db`, em PostgreSQL) e o banco de destino (`reports-db`, em MongoDB) foram extraídos e analisados com scripts em `Python`, executados no Google Colab.

5.1 METODOLOGIA DOS TESTES

A análise dos dados replicados foi realizada por meio da extração manual dos registros após a execução dos testes de carga.

Para o banco PostgreSQL, foi disponibilizado um endpoint do tipo `GET` na aplicação `transactions-api`, responsável por retornar todas as transações registradas no banco `transactions-db`. Esse endpoint foi utilizado para obter os dados completos do banco relacional após os testes.

Já no caso do MongoDB, os dados foram extraídos por meio da interface web do Mongo Express, conectada ao container do banco de destino `reports-db`.

As duas extrações foram salvas como arquivos no formato `JSON` e utilizadas como base para as comparações realizadas em scripts Python executados no Google Colab.

A partir desses arquivos JSON, foram analisados os seguintes aspectos:

- **Consistência dos dados**: comparação direta dos campos `clientId`, `type`, `amount` e `description` entre os dois bancos.
- **Latência de replicação**: medida pela diferença entre os timestamps de criação dos registros (`createdAt`) nas duas fontes.

5.2 CENÁRIOS DE TESTE

Foram definidos quatro cenários com diferentes configurações de carga, executados com a ferramenta k6. Cada cenário envolveu a simulação de requisições `POST` e `PATCH` com dados variados, incluindo um identificador único na descrição (ex: `cenario1`) para posterior filtragem e análise dos registros correspondentes.

- **Cenário 1 - Baixa carga**: 5 usuários virtuais por 30 segundos.
- **Cenário 2 - Carga moderada**: 20 usuários virtuais por 1 minuto.
- **Cenário 3 - Pico de carga**: de 0 a 50 usuários em 10 segundos.
- **Cenário 4 - Carga sustentada**: 50 usuários virtuais por 5 minutos.

5.3 RESULTADOS OBTIDOS

Os resultados foram divididos por cenário, apresentando as métricas de execução do k6, a taxa de consistência por campo e a latência de replicação. Gráficos e tabelas foram incluídos para facilitar a visualização das informações.

5.3.1 Cenário 1: Baixa carga

O primeiro cenário teve como objetivo avaliar o comportamento da arquitetura sob uma carga leve, simulando 5 usuários virtuais (`VUs`) enviando requisições simultâneas durante 30 segundos. O script de teste foi implementado na ferramenta k6, utilizando operações de `POST` e `PATCH` sobre o endpoint `/transactions` da aplicação transactions-api.

Análise das requisições

A Tabela 1 apresenta as principais métricas coletadas durante a execução do teste.

Tabela 1 – Métricas do cenário 1 - k6

Métrica	**Valor**	**Unidade**
Requisições totais (sucesso)	129	
Latência média	79.34	ms
Latência mínima	35.54	ms
Latência máxima	644.22	ms
Latência (p90)	84.63	ms
Latência (p95)	337.21	ms
Falhas	0	
VUs em execução	5	
Duração do teste	30	segundos

Durante o teste, nenhuma requisição falhou, evidenciando estabilidade funcional na aplicação transactions-api. Além disso, todas as respostas `POST` e `PATCH` retornaram com o status `200 OK`, conforme validado nas verificações internas do script `k6`.

Avaliação de consistência

A consistência dos dados entre o banco de origem (PostgreSQL) e o banco de destino (MongoDB) foi avaliada com base em atributos equivalentes (`clientId`, `type`, `description`, `amount`). A comparação considerou tanto inserções (`POST`) quanto atualizações (`PATCH`).

Tabela 2 – Resumo da consistência dos dados entre PostgreSQL e MongoDB

Indicador	**Valor**	**Unidade**
Total de registros comparados	68	registros
Registros consistentes	68	registros
Registros inconsistentes	0	registros
Taxa de consistência	100.00	%

Observa-se que todos os registros analisados apresentaram consistência entre os dois bancos, demonstrando a eficácia da arquitetura de sincronização baseada em `Debezium` e `Kafka`.

A quantidade de registros comparados é inferior ao total de requisições realizadas, pois para um mesmo recurso podem ocorrer múltiplas operações — como inserções seguidas de atualizações — que resultam em um único registro final. Ainda assim, os dados mantiveram-se íntegros em todas as verificações.

5.3.1.1 Análise de latência

A latência foi avaliada separadamente para os dois tipos de operação: criação e atualização de registros. Essa análise considera o tempo entre a inserção ou modificação de dados no banco de origem (PostgreSQL) e a persistência correspondente no banco de destino (MongoDB).

A Tabela 3 apresenta os valores estatísticos da latência de criação, calculada como a diferença entre os timestamps `createdAt_pg` e `createdAt_mongo`.

Tabela 3 – Latência de criação - Cenário 1

Métrica	**Valor**	**Unidade**
Latência média	479.97	ms
Latência mínima	91.13	ms
Latência máxima	924.12	ms
Latência (p90)	657.65	ms
Latência (p95)	711.79	ms

Já a Tabela 4 apresenta os valores relacionados à latência de atualização, obtida pela diferença entre os campos `updatedAt_pg` e `updatedAt_mongo`.

Tabela 4 – Latência de atualização - Cenário 1

Métrica	**Valor**	**Unidade**
Latência média	579.89	ms
Latência mínima	209.78	ms
Latência máxima	919.88	ms
Latência (p90)	840.38	ms
Latência (p95)	856.26	ms

Os resultados indicam que, mesmo em um cenário com baixa carga, o tempo de replicação apresenta variações perceptíveis, especialmente nas operações de atualização, que tiveram latência média superior às de criação. Isso pode estar associado à complexidade interna do fluxo de atualização, que pode envolver reprocessamento ou revalidação de mensagens.

Ainda assim, ambas as operações mantiveram latências médias inferiores a 600ms, o que é aceitável para sistemas que utilizam comunicação assíncrona baseada em eventos. Os percentis p90 e p95 também indicam uma distribuição relativamente estável, sem grandes picos ou outliers relevantes no cenário avaliado.

5.3.2 Cenário 2: Carga intermediária

O segundo cenário teve como objetivo avaliar o comportamento da arquitetura sob uma carga moderada, simulando até 20 usuários virtuais (VUs) realizando operações simultâneas durante 60 segundos. O script de teste foi desenvolvido na ferramenta k6, aplicando requisições do tipo POST e PATCH ao endpoint `/transactions` da aplicação.

Análise das requisições

A Tabela 5 apresenta as principais métricas coletadas durante a execução do teste.

Tabela 5 – Métricas do cenário 2 - k6

Métrica	**Valor**	**Unidade**
Requisições totais realizadas	600	
Requisições bem-sucedidas	570	
Falhas	30	
Latência média	53.18	ms
Latência mínima	0.00	ms
Latência máxima	210.82	ms
Latência (p90)	64.71	ms
Latência (p95)	74.25	ms
VUs em execução	até 20	
Duração do teste	60	segundos

Durante o teste, 95% das requisições foram bem-sucedidas. Apesar das falhas, a aplicação manteve estabilidade, com latência média inferior a 55ms e percentis de até 75ms, demonstrando bom desempenho sob carga intermediária.

Avaliação de consistência

A consistência dos dados entre PostgreSQL e MongoDB foi verificada com base nos campos `clientId`, `type`, `description` e `amount`. Foram comparados 290 registros sincronizados — número inferior ao total de requisições emitidas, devido à presença de múltiplas operações sobre os mesmos dados (como inserção e posterior atualização).

A Tabela 6 resume os resultados da análise de consistência.

Tabela 6 – Resumo da consistência dos dados entre PostgreSQL e MongoDB

Indicador	**Valor**	**Unidade**
Total de registros comparados	290	registros
Registros consistentes	290	registros
Registros inconsistentes	0	registros
Taxa de consistência	100.00	%

Todos os registros analisados foram considerados consistentes, com 100% de correspondência em todos os campos verificados. Isso evidencia a robustez do mecanismo de replicação baseado em `Kafka` e `Debezium`, mesmo com aumento no volume de transações.

5.3.2.1 Análise de latência

A latência foi medida considerando a diferença de tempo entre os registros nos bancos de origem (PostgreSQL) e destino (MongoDB), tanto para inserções quanto para atualizações.

A Tabela 7 apresenta os resultados estatísticos da latência de criação.

Tabela 7 – Latência de criação - Cenário 2

Métrica	**Valor**	**Unidade**
Latência média	542.25	ms
Latência mínima	184.92	ms
Latência máxima	967.31	ms
Latência (p90)	787.02	ms
Latência (p95)	864.16	ms

A seguir, a Tabela 8 mostra os dados referentes à latência de atualização.

Tabela 8 – Latência de atualização - Cenário 2

Métrica	**Valor**	**Unidade**
Latência média	546.10	ms
Latência mínima	88.39	ms
Latência máxima	965.98	ms
Latência (p90)	793.28	ms
Latência (p95)	846.56	ms

Observa-se que a latência média de criação e de atualização foram bastante próximas, em torno de 540ms, o que demonstra consistência no tempo de replicação entre diferentes tipos de operação. Os percentis p90 e p95 também mantiveram-se abaixo de 900ms, indicando que a maior parte das transações foi processada com atraso inferior a 1 segundo.

Mesmo sob carga intermediária e presença de falhas pontuais nas requisições, a arquitetura se mostrou eficaz na entrega de mensagens e replicação dos dados, sem comprometer a integridade nem o desempenho geral do sistema.

5.3.3 Cenário 3: Pico de carga

O terceiro cenário teve como objetivo avaliar o comportamento da arquitetura sob uma situação de pico de carga. Para isso, foi utilizada a estratégia de escalonamento progressivo de usuários virtuais (VUs) por meio do executor `ramping-vus` da ferramenta k6. O teste foi configurado da seguinte forma:

- Início com 10 VUs durante 10 segundos.
- Aumento para 50 VUs por mais 10 segundos.
- Pico de carga com 100 VUs durante 20 segundos.
- Redução gradual para 50 VUs por 10 segundos.
- Finalização com 10 VUs por 10 segundos.

Durante a execução, foram realizadas operações de POST e PATCH sobre o endpoint `/transactions`, simulando simultaneamente a criação e atualização de registros transacionais.

Análise das requisições

A Tabela 9 apresenta as principais métricas extraídas do relatório do k6 para este cenário.

Tabela 9 – Métricas do cenário 3 - k6

Métrica	**Valor**	**Unidade**
Requisições totais	680	
Requisições com sucesso (checks)	525	
Requisições com falha	155	
Latência média	36.44	ms
Latência mínima	0.00	ms
Latência máxima	67.64	ms
Latência (p90)	53.22	ms
Latência (p95)	54.44	ms
VUs máximos simultâneos	100	
Duração total do teste	60	segundos

A presença de falhas nas requisições (~23%) indica que, sob alta concorrência, o sistema pode enfrentar dificuldades para manter estabilidade. Apesar disso, as requisições que foram bem-sucedidas apresentaram latência média aceitável (36,44ms), com percentis 90 e 95 próximos, indicando consistência no tempo de resposta.

Avaliação de consistência

A consistência dos dados entre os bancos PostgreSQL (origem) e MongoDB (destino) foi verificada por meio da comparação dos campos `clientId`, `type`, `description` e `amount`.

Tabela 10 – Resumo da consistência dos dados entre PostgreSQL e MongoDB

Indicador	**Valor**	**Unidade**
Total de registros comparados	273	registros
Registros consistentes	273	registros
Registros inconsistentes	0	registros
Taxa de consistência	100.00	%

Mesmo com um número considerável de requisições falhas, os registros que foram efetivamente persistidos se mantiveram consistentes entre os dois sistemas. Isso demonstra robustez na arquitetura de sincronização, mesmo sob pressão. Ressalta-se que o número de registros comparados pode ser inferior ao número de requisições totais devido à coexistência de múltiplas operações (inserção e atualização) sobre o mesmo recurso.

5.3.3.1 Análise de latência

A Tabela 11 apresenta as métricas de latência de criação dos registros (tempo entre `createdAt_pg` e `createdAt_mongo`).

Tabela 11 – Latência de criação - Cenário 3

Métrica	**Valor**	**Unidade**
Latência média	486.79	ms
Latência mínima	35.72	ms
Latência máxima	941.69	ms
Latência (p90)	744.14	ms
Latência (p95)	835.27	ms

A seguir, a Tabela 12 apresenta os dados de latência de atualização (diferença entre `updatedAt_pg` e `updatedAt_mongo`).

Tabela 12 – Latência de atualização - Cenário 3

Métrica	**Valor**	**Unidade**
Latência média	463.03	ms
Latência mínima	20.47	ms
Latência máxima	963.43	ms
Latência (p90)	752.91	ms
Latência (p95)	822.80	ms

Mesmo sob uma carga extrema, os tempos médios de replicação ficaram em torno de 460–490ms, e os percentis p90 e p95 mantiveram-se abaixo de 850ms. Esses resultados mostram que o sistema continua operando dentro de limites aceitáveis de latência para um ambiente assíncrono, mesmo com aumento abrupto e elevado de concorrência.

Contudo, a presença de falhas HTTP durante o pico reforça a necessidade de ajustes de resiliência e escalabilidade para cenários reais de produção.

5.3.4 Cenário 4: Carga sustentada

O quarto cenário teve como objetivo avaliar o comportamento da arquitetura sob uma carga sustentada, com 50 usuários virtuais (`VUs`) ativos durante 5 minutos. Esse padrão simula um

ambiente com demanda constante e contínua. O teste foi implementado na ferramenta k6, com foco em operações `POST` e `PATCH` no endpoint `/transactions` da aplicação.

Análise das requisições

A Tabela 13 apresenta as principais métricas coletadas durante o teste.

Tabela 13 – Métricas do cenário 4 - k6

Métrica	**Valor**	**Unidade**
Requisições totais (sucesso)	2817	
Requisições totais (falha)	560	
Latência média	54.78	ms
Latência mínima	31.35	ms
Latência máxima	47.31	ms
Latência (p90)	52.96	ms
Latência (p95)	55.79	ms
VUs em execução	50	
Duração do teste	300	segundos

Apesar de 16,59% das requisições terem falhado (560 de 3377), a taxa geral de verificações de status HTTP foi de 83,41%, e todas as requisições `PATCH` retornaram com `200 OK`, demonstrando resiliência da aplicação mesmo sob carga contínua. O tempo médio de resposta permaneceu estável, com percentis p90 e p95 abaixo de 60ms.

Avaliação de consistência

A comparação dos dados entre o banco de origem (PostgreSQL) e o banco de destino (MongoDB) mostrou resultados totalmente consistentes, conforme demonstrado na Tabela 14.

Tabela 14 – Resumo da consistência dos dados entre PostgreSQL e MongoDB

Indicador	**Valor**	**Unidade**
Total de registros comparados	1442	registros
Registros consistentes	1442	registros
Registros inconsistentes	0	registros
Taxa de consistência	100.00	%

Todos os registros analisados apresentaram consistência total, confirmando que a arquitetura baseada em eventos manteve a integridade dos dados mesmo com tráfego constante. A diferença entre o número de requisições e de registros comparados se deve à ocorrência de múltiplas operações (como POST e PATCH) sobre os mesmos dados, o que resulta em um único registro final.

5.3.4.1 Análise de latência

A análise de latência foi realizada separadamente para as operações de criação e atualização. A Tabela 15 apresenta os valores estatísticos referentes à latência de criação.

Tabela 15 – Latência de criação - Cenário 4

Métrica	**Valor**	**Unidade**
Latência média	496.41	ms
Latência mínima	17.91	ms
Latência máxima	999.70	ms
Latência (p90)	765.21	ms
Latência (p95)	844.67	ms

Já a Tabela 16 mostra os valores de latência para atualizações.

Tabela 16 – Latência de atualização - Cenário 4

Métrica	**Valor**	**Unidade**
Latência média	497.76	ms
Latência mínima	17.12	ms
Latência máxima	993.08	ms
Latência (p90)	793.46	ms
Latência (p95)	843.92	ms

Mesmo sob carga sustentada, a latência média de replicação permaneceu em torno de 500ms para ambas as operações. Os percentis p90 e p95 indicam que a maioria das operações é replicada em menos de 850ms, o que está dentro dos limites aceitáveis para sistemas assíncronos baseados em eventos. A consistência e estabilidade desses tempos reafirmam a robustez da arquitetura.

5.3.4.2 Análise Comparativa entre os Cenários

A análise comparativa entre os quatro cenários permite observar como a arquitetura se comporta sob diferentes níveis de carga e tipos de estímulo. A Tabela 17 resume os principais indicadores de consistência, enquanto a Tabela 18 apresenta os resultados de latência para criação e atualização de registros.

Tabela 17 – Resumo comparativo de consistência entre os cenários

Cenário	**Registros Comparados**	**Consistentes**	**Taxa de Consistência**
Cenário 1 (baixa carga)	68	68	100.00%
Cenário 2 (carga moderada)	290	290	100.00%
Cenário 3 (pico de carga)	273	273	100.00%
Cenário 4 (carga sustentada)	1442	1442	100.00%

Os resultados demonstram que, independentemente da intensidade da carga, a arquitetura manteve consistência total entre os dados processados no PostgreSQL e os replicados no MongoDB. A utilização de `Debezium` e `Kafka` mostrou-se robusta, mesmo em situações de maior estresse, como os cenários 3 e 4.

Tabela 18 – Resumo comparativo de latência (média) entre os cenários

Cenário	**Criação (ms)**	**Atualização (ms)**
Cenário 1	479.97	579.89
Cenário 2	542.25	546.10
Cenário 3	486.79	463.03
Cenário 4	496.41	497.76

Embora o cenário 2 tenha apresentado as maiores latências médias, é importante notar que essas variações são aceitáveis dentro do contexto de arquiteturas baseadas em eventos. O cenário 3, que simulou um pico repentino de carga, apresentou desempenho satisfatório, com latências menores que as do cenário 2. Já o cenário 4, com carga sustentada, manteve latências estáveis, demonstrando boa escalabilidade da solução.

5.3.5 Considerações Finais

A avaliação prática da arquitetura demonstrou que a integração entre PostgreSQL, Kafka, Debezium e MongoDB é capaz de suportar diferentes cargas de trabalho mantendo alta consistência e desempenho aceitável. Mesmo sob picos de carga ou operações sustentadas, os dados foram sincronizados corretamente e dentro de um tempo de latência inferior a 600ms na média.

A ausência de inconsistências nos quatro cenários reforça a confiabilidade do uso de *Change Data Capture* (CDC) para sincronização em tempo real em arquiteturas de microsserviços. Além disso, a utilização de mensageria assíncrona contribuiu para o desacoplamento entre os sistemas e a resiliência da comunicação.

Os resultados obtidos indicam que a abordagem testada é viável para sistemas distribuídos que exigem replicação de dados em tempo real com confiabilidade e tolerância a falhas, servindo como base sólida para aplicações modernas baseadas em microsserviços.

6 CONCLUSÃO E TRABALHOS FUTUROS

A crescente demanda por sistemas distribuídos robustos, escaláveis e responsivos tem impulsionado a adoção de arquiteturas baseadas em microsserviços. No entanto, essa abordagem impõe desafios significativos relacionados à sincronização de dados entre serviços independentes, especialmente em ambientes com bancos de dados heterogêneos. Neste contexto, este trabalho explorou a aplicação prática da técnica *Change Data Capture* (CDC), com o objetivo de garantir a consistência de dados em tempo real entre microsserviços desacoplados.

Foi implementada uma arquitetura simples, composta por dois microsserviços independentes, integrando PostgreSQL e MongoDB como sistemas de origem e destino, respectivamente. A sincronização foi viabilizada por meio do Debezium, que atua como conector CDC baseado em logs, e do Apache Kafka, que garantiu a comunicação assíncrona e resiliente entre os serviços. A solução foi desenvolvida com suporte de tecnologias modernas como Micronaut, Docker e k6.

Através de testes controlados, foram analisados quatro cenários distintos de carga, avaliando dois aspectos centrais: a consistência dos dados replicados e a latência no processo de sincronização. Os resultados demonstraram que, mesmo sob picos de carga ou demandas sustentadas, o sistema manteve uma taxa de consistência de 100% entre os bancos, com latência média de replicação inferior a 600ms. Esses indicadores reforçam a viabilidade do uso de CDC como estratégia eficiente e escalável para sincronização de dados em arquiteturas distribuídas.

Além da efetividade técnica, o trabalho também evidenciou boas práticas de desenvolvimento e orquestração de ambientes com containers, uso de mensageria desacoplada, e aplicação de ferramentas de teste para avaliação de performance. A infraestrutura baseada em Docker e a modularização dos serviços contribuíram para a simplicidade da implantação e facilitaram a análise dos resultados.

Entretanto, algumas limitações foram identificadas. A principal delas está relacionada à escalabilidade da solução em ambientes reais de alta concorrência, uma vez que parte dos testes apresentou falhas de requisição HTTP sob carga extrema. Outro ponto a considerar é a necessidade de ajustes finos na configuração dos componentes (Kafka, Debezium, bancos de dados) para ambientes de produção com maior volume de dados e múltiplos serviços simultâneos.

Como trabalhos futuros, sugere-se:

- Ampliar o número de microsserviços e fontes de dados na arquitetura, testando sincronização com múltiplas tabelas e modelos mais complexos;
- Avaliar estratégias complementares de tolerância a falhas e reprocessamento de eventos;
- Investigar o uso de CDC em ambientes cloud-native, como AWS ou Kubernetes, integrando com ferramentas como Kafka Connect e Apache Flink;
- Comparar o desempenho do Debezium com outras abordagens CDC, como trigger-based ou timestamp-based, em diferentes contextos;
- Explorar métricas adicionais, como throughput e impacto na performance transacional do banco de origem.

REFERÊNCIAS

AIRBYTE. *Data Synchronization: Everything You Need to Know*. 2023. Acesso em: 07 mar. 2025. Disponível em: <https://airbyte.com/data-engineering-resources/data-synchronization>.

AMAZON. *Qual é a diferença entre arquitetura monolítica e de microsserviços?* 2025. Disponível em: <https://aws.amazon.com/pt/compare/the-difference-between-monolithic-and-microservices-architecture>. Acesso em: Acesso em: 03 mar. 2025.

Amazon Web Services. *Microsserviços*. 2024. <https://aws.amazon.com/pt/microservices/>. Acesso em: 11 dez. 2024.

CHEN, M. *What is data synchronization*. 2024. Disponível em: <https://www.oracle.com/pt/integration/data-synchronization/>. Acesso em: Accessed: 2025-03-07.

CHINA, C. R.; GOODWIN, M. *What is data synchronization*. 2024. Disponível em: <https://www.ibm.com/think/topics/data-synchronization>. Acesso em: Accessed: 2025-03-07.

CONFLUENT. *Change Data Capture (CDC)*. 2024. <https://www.confluent.io/learn/change-data-capture/>. Acesso em: 11 dez. 2024.

DEBEZIUM. *Debezium Documentation*. 2025. Acesso em: 07 mar. 2025. Disponível em: <https://debezium.io/>.

FOWLER, M.; LEWIS, J. *Microservices*. 2014. <https://martinfowler.com/articles/microservices.html>. Acesso em: 11 dez. 2024.

FOWLER, S. J. *Microsserviços Prontos para Produção: Construindo sistemas escaláveis e confiáveis*. [S.l.]: Novatec Editora, 2017.

GIL, A. C. *Como elaborar projetos de pesquisa*. [S.l.]: Atlas São Paulo, 2002. v. 4.

KAFKA, A. *Apache Kafka*. 2025. Acesso em: 07 mar. 2025. Disponível em: <https://kafka.apache.org/documentation/#gettingStarted>.

KLEPPMANN, M. *Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems*. [S.l.]: O'Reilly Media, 2017.

NEWMAN, S. *Criando Microsserviços: Projetando sistemas pequenos e independentes*. 2. ed. [S.l.]: O'Reilly Brasil, 2022.

ORACLE. *Sagas Are Great: What's the Problem?* 2024. <https://blogs.oracle.com/database/post/sagas-are-great-whats-the-problem>. Acesso em: 11 dez. 2024.

SHARMA, N. *Change-data capture: What, why and how to make the most of it*. 2023. Disponível em: <https://www.thoughtworks.com/en-br/insights/blog/architecture/change_data_capture>. Acesso em: Accessed: 2025-03-05.

TOBIN, D. *How to Implement Change Data Capture (CDC)*. 2021. Disponível em: <https://www.integrate.io/blog/how-to-implement-change-data-capture-cdc/>. Acesso em: Accessed: 2025-03-05.

VALENTE, M. T. *Engenharia de Software Moderna: Capítulo 7 - Arquiteturas de Microsserviços*. 2024. <https://engsoftmoderna.info/cap7.html>. Acesso em: 11 dez. 2024.

VIGGIATO, M.; TERRA, R.; ROCHA, H.; VALENTE, M. T.; FIGUEIREDO, E. Microservices in practice: A survey study. *Brazilian Workshop on Software Visualization, Evolution and Maintenance (VEM)*, 2018. Disponível em: <https://arxiv.org/abs/1808.04836>.