UNIVERSIDADE FEDERAL DO CEARÁ

CENTRO DE CIÊNCIAS

DEPARTAMENTO DE COMPUTAÇÃO

CURSO DE GRADUAÇÃO EM CIÊNCIA DA COMPUTACÃO

LUAN CARVALHO DE ARAÚJO COELHO

UMA ARQUITETURA DE CÓDIGO ABERTO PARA A APLICAÇÃO DO STRANGLER PATTERN COM CHANGE DATA CAPTURE PARA SETORES COM ALTO VOLUME DE DADOS

FORTALEZA

2023

LUAN CARVALHO DE ARAÚJO COELHO

UMA ARQUITETURA DE CÓDIGO ABERTO PARA A APLICAÇÃO DO STRANGLER PATTERN COM CHANGE DATA CAPTURE PARA SETORES COM ALTO VOLUME DE DADOS

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Ciência da Computacão do Centro de Ciências da Universidade Federal do Ceará, como requisito parcial à obtenção do grau de bacharel em Ciência da Computacão.

Orientador: Prof. Dr. José Maria da Silva Monteiro Filho.

FORTALEZA

2023

Dados Internacionais de Catalogação na Publicação
Universidade Federal do Ceará
Sistema de Bibliotecas
Gerada automaticamente pelo módulo Catalog, mediante os dados fornecidos pelo(a) autor(a)

C617a Coelho, Luan Carvalho de Araújo.
Uma arquitetura de código aberto para a aplicação do Strangler Pattern com Change Data Capture para setores com alto volume de dados / Luan Carvalho de Araújo Coelho. – 2023.
79 f. : il. color.

Trabalho de Conclusão de Curso (graduação) – Universidade Federal do Ceará, Centro de Ciências, Curso de Computação, Fortaleza, 2023.
Orientação: Prof. Dr. José Maria da Silva Monteiro Filho.

1. Strangler Pattern. 2. Arquitetura de Microserviços. 3. Alto Volume de Dados. 4. Change Data Capture. I. Título.

CDD 005

LUAN CARVALHO DE ARAÚJO COELHO

UMA ARQUITETURA DE CÓDIGO ABERTO PARA A APLICAÇÃO DO STRANGLER PATTERN COM CHANGE DATA CAPTURE PARA SETORES COM ALTO VOLUME DE DADOS

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Ciência da Computacão do Centro de Ciências da Universidade Federal do Ceará, como requisito parcial à obtenção do grau de bacharel em Ciência da Computacão.

Aprovada em: xx/xx/xxxx.

BANCA EXAMINADORA

Prof. Dr. José Maria da Silva Monteiro Filho (Orientador)
Universidade Federal do Ceará (UFC)

Prof. Dr. Fernando Antonio Mota Trinta
Universidade Federal do Ceará (UFC)

Prof. Dr. José Wellington Franco da Silva
Universidade Federal do Ceará (UFC)

Prof. Me. Ivandro Claudino de Sá
Serviço Federal de Processamento de Dados
(SERPRO)

Dedico este trabalho à minha família e namorada, meus alicerces inabaláveis. Aos meus pais, padrasto e irmã, pela fé e apoio contínuos. À minha namorada, por seu encorajamento constante. À minha avó e tia, que com firmeza e determinação, impediram-me de desistir do curso nos momentos de dúvida. A cada um, minha profunda gratidão por todo seu amor.

AGRADECIMENTOS

Ao Prof. Dr. José Maria da Silva Monteiro Filho, pela excelente orientação e suporte contínuo junto ao laboratório ARIDA.

Aos professores Dr. José Antônio Fernandes de Macêdo, Dr. João Fernando Lima Alcântara e Prof. Dr. Fernando Antonio Mota Trinta, pelas mentorias valiosas e pelo apoio em diversos projetos durante minha trajetória acadêmica.

Aos meus pais, padrasto e irmã, pela força, apoio e amor incondicionais que foram alicerce em minha formação. A minha tia e avó, pela criação e educação incansável e por serem as colunas que sustentaram minha permanência no curso em momentos desafiadores.

A minha namorada e sua família, pelo amor profundo, apoio e compreensão que iluminaram meu processo acadêmico.

Aos integrantes e amigos dos projetos de extensão Ceos JR e Sociedade de Debates, pelas experiências inestimáveis e pelo apoio que se tornaram pilares de minha permanência no curso.

A todos os amigos da época de colégio do grupo "Bom Dia Flopados", minha sincera gratidão. Cada momento compartilhado e apoio dado foram essenciais, fortalecendo nossos laços e enriquecendo minha jornada acadêmica. Vocês são inestimáveis.

Aos meus colegas da computação, pelas lições valiosas, auxílio e amizade que transformaram os desafios em experiências memoráveis.

À Greenmile Inc, e em especial a André Campos, Claudemir Woche, Ivaldo Almeida, Kaynan Coelho, Rafael Rêgo e Regis Melo, pelo impacto significativo em minha carreira e desenvolvimento profissional.

Ao Banco BTG Pactual, e de forma destacada, à equipe de PFM, pela oportunidade única de imersão na integração entre a academia e o mercado. Sou grato pela ampliação da minha perspectiva profissional, pelo aprofundamento do meu entendimento sobre a complexidade e dinâmica do setor financeiro, e pela compreensão e apoio inestimáveis durante todo o desenvolvimento deste trabalho.

A todos os membros do Departamento de Computação, pela orientação, suporte e inspiração que foram decisivos em cada etapa da minha graduação. Agradeço especialmente ao corpo docente pelo conhecimento e inspiração impartidos.

RESUMO

Este artigo propõe uma arquitetura de código aberto para a implementação do *Strangler Pattern* em contextos que gerenciam grandes volumes de dados, usando tecnologias como *Apache Spark*, *Debezium* e *Apache Kafka*.

O *Strangler Pattern* é uma estratégia de modernização de sistemas legados que facilita a transição gradual para novas funcionalidades e serviços, reduzindo a interrupção de serviços essenciais. É especialmente útil em cenários com grandes volumes de dados, onde a continuidade e a evolução dos serviços necessitam ser equilibradas cuidadosamente.

Nesta arquitetura, o *Apache Spark* e o *Debezium* são empregados para implementar o *Change Data Capture* (CDC), uma técnica que identifica e captura alterações em um banco de dados, incluindo operações de inserção (*INSERT*), atualização (*UPDATE*) e exclusão (*DELETE*). Isso permite o processamento e transferência apenas dos dados modificados para o *Data Warehouse*, *Data Lake* ou outro destino, otimizando o tempo de processamento e o uso de recursos.

A arquitetura também utiliza o Apache Kafka para gerenciar fluxos de dados em tempo real, recurso essencial para a inteligência de negócios em ambientes de alto volume de dados. Outras ferramentas como *Kafka Connect* e *Alluxio* desempenham papéis centrais na arquitetura e serão discutidos posteriormente.

Resultados iniciais sugerem que a arquitetura proposta pode diminuir substancialmente os custos computacionais, aprimorar a eficiência do processamento de dados e minimizar a interrupção dos serviços existentes. O artigo apresenta detalhadamente a implementação da arquitetura, abordando os desafios, as soluções e um estudo de caso no mercado de *supply chain* em que a arquitetura foi primeiramente aplicada.

Palavras-chave: Strangler Pattern; Arquitetura de Microserviços; Alluxio; Apache Spark; Debezium; Apache Kafka; Change Data Capture; Alto Volume de Dados.

ABSTRACT

This paper proposes an open-source architecture for implementing the Strangler Pattern in contexts managing large volumes of data, using technologies such as Apache Spark, Debezium, and Apache Kafka.

The Strangler Pattern is a legacy system modernization strategy that facilitates the gradual transition to new functionalities and services, minimizing the disruption of essential services. It is particularly beneficial in scenarios dealing with large data volumes, where continuity and evolution of services need to be carefully balanced.

In this architecture, Apache Spark and Debezium are employed to implement the Changed Data Capture (CDC), a technique that identifies and captures changes in a database, including INSERT, UPDATE, and DELETE operations. This allows for processing and transfer of only modified data to the Data Warehouse, Data Lake, or other destinations, optimizing processing time and resource use.

The architecture also utilizes Apache Kafka to manage real-time data flows, a critical feature for business intelligence in high data volume environments. Other tools such as Kafka Connect and Alluxio play central roles in the architecture and will be discussed later.

Initial results suggest that the proposed architecture can significantly decrease computational costs, improve data processing efficiency, and minimize the disruption of existing services. The paper thoroughly presents the architecture implementation, addressing challenges, solutions, and a case study in the supply chain market where the architecture was initially applied.

Keywords: Strangler Pattern; Microservices Architecture; Alluxio; Apache Spark; Debezium; Apache Kafka; Change Data Capture; High Volume of Data.

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

ACID	Atomicidade, Consistência, Isolamento e Durabilidade
API	Application Programming Interface
AWS	Amazon Web Services
CDC	*Change Data Capture*
CQRS	Command Query Responsibility Segregation
CRM	Customer Relationship Management
CTO	Chief Technology Officer
ECS	Amazon Elastic Container Service
EMR	Amazon Elastic MapReduce
ERP	Enterprise Resource Planning
ETL	Extract, Transform, Load
GCS	Google Cloud Storage
gRPC	gRPC Remote Procedure Calls
HDFS	Hadoop Distributed File System
HTTP	Hypertext Transfer Protocol
I/O	Input/Output
IOPS	Input/Output Operations Per Second
JDBC	Java Database Connectivity
JSON	JavaScript Object Notation
OLAP	Online analytical processing
PDI	Pentaho Data Integration
POC	Prova de Conceito
S3	Amazon S3
SMT	Single Message Transformation
SQL	Structured Query Language
VDFS	Virtual Distributed File System

SUMÁRIO

1 INTRODUÇÃO

Este trabalho explora uma arquitetura de microsserviços, focando na aplicação do *Strangler Pattern* em sinergia com o CDC. Utiliza-se um conjunto de tecnologias de código aberto, incluindo *Apache Spark*, *Debezium* e *Apache Kafka*, para gerenciar eficientemente os fluxos de dados em tempo real e otimizar o processamento e a análise de informações.

Com o aumento da demanda por processamento ágil de grandes volumes de dados, especialmente em setores de alto tráfego de dados como o *supply chain*, surge a necessidade de sistemas mais eficientes e escaláveis. A arquitetura de microsserviços, com sua abordagem modular, oferece uma solução promissora para atingir essa escalabilidade e eficiência.

A migração de sistemas legados para arquiteturas baseadas em microsserviços, no entanto, apresenta desafios consideráveis. O *Strangler Pattern* se destaca como uma metodologia eficaz para essa transição, facilitando a substituição gradual de componentes de sistemas antigos por serviços mais modernos. Combinado com o CDC, uma técnica essencial para detectar e capturar alterações em dados, essa abordagem promove uma migração suave e uma integração contínua entre sistemas monolíticos e microsserviços.

Este estudo concentra-se em um estudo de caso, aplicando a arquitetura proposta para aprimorar uma solução analítica em uma empresa. Desafios significativos eram enfrentados anteriormente ao novo processo, especialmente nos processos de Extract, Transform, Load (ETL) em bancos de dados *Oracle* e nas intensas consultas analíticas realizadas em *dashboards*. Estes processos, críticos para a funcionalidade do produto, frequentemente causavam interrupções em toda a operação.

A aplicação da arquitetura proposta neste caso específico demonstrou sua eficácia na gestão de grandes volumes de dados. Os resultados iniciais indicam uma redução significativa nos custos computacionais e um aumento na eficiência do processamento de dados, minimizando as interrupções nos serviços existentes e permitindo uma extração de dados mais eficiente e rápida.

O documento está organizado da seguinte forma: Capítulos 2 a 7 discutem fundamentação teórica, trabalhos relacionados, metodologia, arquitetura proposta, estudo de caso, e conclusões O Anexo 1 fornece uma visão aprofundada sobre o CDC, explorando seu funcionamento e peculiaridades.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo apresenta os principais conceitos que foram utilizados durante o desenvolvimento dessa pesquisa. Inicialmente iremos discutir os fundamentos teóricos que nortearam a concepção da arquitetura proposta neste trabalho. Em seguida, discutiremos as ferramentas que foram utilizadas na construção da arquitetura proposta.

2.1 Monólitos

Os sistemas monolíticos, uma abordagem tradicional na arquitetura de software, caracterizam-se pela integração de todas as funcionalidades do sistema em uma única aplicação e base de código. Esta arquitetura oferece vantagens iniciais como a simplicidade no desenvolvimento, implantação e gerenciamento. No entanto, os desafios se tornam aparentes à medida que a aplicação se expande em tamanho e complexidade, afetando negativamente a manutenção, escalabilidade e flexibilidade do sistema (NEWMAN, 2019).

Newman categoriza os monólitos em três tipos principais, cada um com suas peculiaridades e complexidades:

Monólitos de Processo Único: Caracterizam-se por ter todo o código do sistema operando em um único processo. Esta estrutura pode apresentar variações como o monólito modular, onde módulos independentes são implantados como uma unidade única, mantendo o código uniforme em todas as instâncias.

Monólitos Distribuídos: Esses monólitos consistem em vários serviços altamente interconectados que são implantados conjuntamente. Esta abordagem acumula as desvantagens dos sistemas monolíticos e distribuídos, limitando a flexibilidade e a eficiência do sistema.

Sistemas de Terceiros: Estes são sistemas externos à organização, funcionando como caixas pretas. A falta de acesso ao código-fonte e à arquitetura interna desses sistemas impõe desafios únicos em termos de integração e adaptação.

A migração de sistemas monolíticos para arquiteturas baseadas em microsserviços é um processo complexo e um tópico de grande interesse nas discussões sobre modernização de sistemas de software. Compreender as nuances dos diferentes tipos de monólitos é fundamental para planejar e executar essa transição com sucesso.

Cada tipo de monólito apresenta desafios específicos na migração para uma arquitetura de microsserviços. Uma estratégia adaptativa e bem planejada pode oferecer soluções

Figura 1 – Diagrama simples de como funciona uma arquitetura monolítica. Fonte: Imagem feita pelo próprio autor

personalizadas para assegurar uma transição suave. No entanto, neste artigo, generalizaremos esses desafios e focaremos no processo global de transição para a arquitetura de microsserviços.

2.2 Microsserviços

Microsserviços representam um paradigma avançado na engenharia de dados, caracterizados por sua modularidade e adaptabilidade. Segundo (NEWMAN, 2015), microsserviços são pequenos serviços autônomos que trabalham juntos, modelados em torno de domínios de negócios. Fowler e Lewis (FOWLER; LEWIS, 2014) também definem microsserviços como uma

abordagem para desenvolver uma única aplicação como um conjunto de pequenos serviços, cada um executando em seu próprio processo e se comunicando com mecanismos leves, muitas vezes uma *Application Programming Interface (API)* executada sobre o Hypertext Transfer Protocol (HTTP). Essas definições enfatizam a independência de cada serviço, permitindo que sistemas sejam altamente escaláveis e flexíveis, essenciais para organizações que buscam constante inovação e adaptação rápida às mudanças do mercado.

2.2.1 Características dos Microsserviços

Os microsserviços, enquanto entidades de software, são caracterizados pela sua granularidade, especialização e independência (FOWLER; LEWIS, 2014; NEWMAN, 2015). Cada microsserviço é projetado para realizar uma função ou conjunto de funções bem definidas, frequentemente alinhadas a um aspecto específico do negócio, permitindo que operem de forma eficiente e eficaz.

Sua independência é crucial; operam autonomamente e se comunicam com outros serviços apenas através de interfaces bem definidas, como APIs, o que promove a resiliência do sistema.

Os microsserviços também facilitam a escalabilidade, permitindo o dimensionamento independente com base nas necessidades de recursos e demanda de cada serviço. Isso contrasta com sistemas monolíticos, onde a escalabilidade geralmente requer replicação do sistema inteiro.

Além disso, suportam a diversidade tecnológica, permitindo o uso de diferentes linguagens ou *frameworks* para cada serviço, escolhidos pelo seu adequado à tarefa específica (NEWMAN, 2015).

2.2.2 Vantagens

A arquitetura de microsserviços se destaca pela sua capacidade de isolar inovações e atualizações em unidades individuais, reduzindo significativamente o risco associado a mudanças no sistema (NEWMAN, 2015; RICHARDS, 2015). Essa estrutura modular não só acelera o ciclo de inovação, mas também proporciona uma escalabilidade sem precedentes (FOWLER; LEWIS, 2014; WOLFF, 2016). Os processos de testes e implantações são simplificados e mais eficientes (FOWLER; LEWIS, 2014), enquanto a modularidade promove maior flexibilidade de desenvolvimento, coesão interna dos serviços e autonomia das equipes de desenvolvimento. Esta abordagem descentralizada permite uma melhor alocação de recursos e uma gestão mais eficaz,

maximizando a eficiência operacional.

2.2.3 Desafios

No entanto, os microsserviços apresentam desafios intrínsecos, como uma complexidade significativa no gerenciamento de múltiplos serviços independentes (PARSONS, 2018). A consistência dos dados e a comunicação eficaz entre os serviços distribuídos são aspectos importantes que requerem atenção cuidadosa (RICHARDSON, 2018). Problemas de segurança e monitoramento são mais desafiadores devido à natureza distribuída e aos múltiplos pontos de entrada e saída (WOLFF, 2016). Além disso, a orquestração eficaz de serviços, a manutenção da consistência em transações complexas e o desafio de testar e depurar um ecossistema extenso e interconectado apresentam dificuldades adicionais. Para enfrentar esses desafios, é fundamental adotar práticas eficientes de monitoramento, automação de implantação, escalabilidade dinâmica, segurança aprimorada, design tolerante a falhas e gerenciamento eficiente de tráfego de rede. A adoção dessas práticas assegurar a integridade e a manutenção da eficiência operacional e a resiliência dos sistemas em um ambiente distribuído.

2.2.4 Arquitetura de Microsserviços

A arquitetura de microsserviços emerge como uma técnica inovadora no *design de software*, caracterizada por dividir aplicações complexas em serviços mais granulares e independentes. Cada microsserviço é dedicado a um processo de negócio específico e se comunica com outros serviços via protocolos leves, frequentemente HTTP ou gRPC Remote Procedure Calls (gRPC)(NEWMAN, 2021).

A autonomia dos microsserviços facilita uma série de vantagens operacionais e de desenvolvimento. Estes componentes independentes podem ser escritos em diferentes linguagens de programação, aproveitando diversos ecossistemas de software. A modularidade inerente promove a manutenção, o teste e a implementação contínua, permitindo que as organizações respondam de maneira ágil às mudanças no mercado e às demandas dos usuários.

Entretanto, a complexidade da gestão e operacionalização desses serviços distribuídos não pode ser subestimada. A comunicação inter-serviço, a coordenação, a descoberta de serviços, e a segurança tornam-se questões preponderantes. Cada serviço, enquanto isolado em funcionalidade, é parte integral de um ecossistema mais amplo, demandando estratégias coesas de integração, monitoramento, e gestão de falhas.

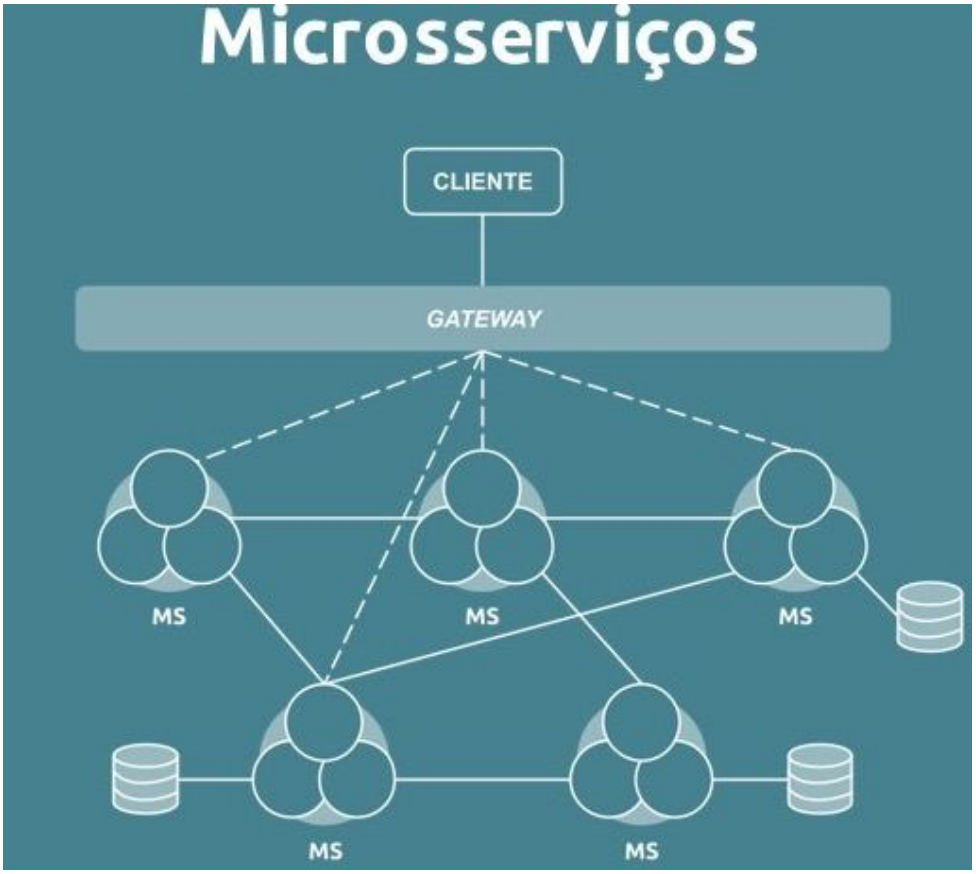

Figura 2 – Representação esquemática da arquitetura de microsserviços, destacando a independência e comunicação entre os serviços. Fonte: Imagem feita pelo próprio autor

A sustentabilidade dos microsserviços é outro ponto crítico. Cada alteração em um serviço pode ter efeitos cascata em toda a arquitetura. Portanto, práticas robustas de versionamento, testes automatizados e integração contínua/*devops* são imperativas para garantir a estabilidade, segurança e a evolução progressiva do sistema.

Neste contexto, tecemos uma análise aprofundada de como a arquitetura de microsserviços se entrelaça com estratégias contemporâneas como o *Strangler Pattern* e o CDC. O *Strangler Pattern* oferece um caminho metodológico para a transição incremental de sistemas monolíticos para microsserviços, mitigando riscos e assegurando a continuidade operacional. O CDC, por sua vez, amplifica a integração de dados em tempo real, vital para a dinâmica de operações baseadas em microsserviços, garantindo a consistência, a integridade e a disponibilidade dos dados através dos serviços distribuídos.

2.3 Strangler Pattern

Como citado em (YODER; MERSON, 2020), o *Strangler Pattern* é uma estratégia para a migração de sistemas legados para arquiteturas de microsserviços. Ele envolve a criação

de um novo sistema ao redor do sistema existente, com o novo sistema gradualmente assumindo a funcionalidade do sistema antigo até que o sistema antigo possa ser "estrangulado"(FOWLER, 2004). Este padrão permite uma transição suave e controlada, minimizando o risco de interrupções ou perda de dados. No contexto da arquitetura proposta, o *Strangler Pattern* é utilizado para migrar de um sistema de processamento de dados em lote para um sistema de processamento de dados em tempo real.

No entanto, existem desvantagens e desafios associados à implementação do *Strangler Pattern*. Um deles é a sustentação, pois é necessário manter duas tecnologias em paralelo durante o período de transição. Isso pode levar a um aumento nos custos operacionais e à necessidade de habilidades e conhecimentos técnicos adicionais para gerenciar ambas as arquiteturas eficientemente. Existem desafios inerentes à coordenação e integração de sistemas legados e novos, o que pode complicar o processo de migração e aumentar o risco de erros e incompatibilidades.

2.4 Estratégias de Integração

A transição de monólitos para microsserviços é um processo complexo que vai muito além da mera replicação de tabelas de dados, conforme ilustrado por (YANAGA, 2017). As estratégias para integrar estas arquiteturas podem variar de acordo com as necessidades de cada aplicação. Nas seções a seguir, são apresentadas algumas das estratégias possíveis para lidar com a integração entre monólitos e microsserviços.

2.4.1 Event Sourcing

Event Sourcing é uma estratégia vital para a transição entre sistemas monolíticos e microsserviços. A estratégia consiste em persistir o estado do sistema através do registro contínuo de eventos, permitindo a recriação do estado do sistema a qualquer momento. No contexto de integração, oferece uma trilha detalhada de mudanças, facilitando a sincronização, rastreamento de *bugs* e auditorias. Cada microsserviço pode se inscrever e reagir aos eventos de interesse, garantindo a consistência e a coesão entre os serviços durante a fase de transição.

2.4.2 Virtualização de Dados

A virtualização de dados surge como uma técnica essencial para unificar e simplificar o acesso aos dados distribuídos em sistemas monolíticos e microsserviços. Ela age como uma interface unificada, oferecendo uma visão coesa dos dados dispersos, sem necessidade de migrá-los ou transformá-los. Esse aspecto é crucial para garantir a integridade, consistência e acessibilidade dos dados durante o processo de integração, promovendo a flexibilidade e a agilidade na evolução da arquitetura do sistema.

2.4.3 CQRS

CQRS, é um padrão de design que propõe a separação das operações de leitura (*queries*) e escrita (*commands*) em um sistema de software, permitindo que sejam otimizadas de maneira independente. Em um sistema tradicional, os modelos de leitura e escrita são tipicamente o mesmo, o que pode levar a uma complexidade desnecessária quando as operações de leitura e escrita têm requisitos diferentes. Por exemplo, as operações de leitura podem exigir uma visão agregada e complexa dos dados, enquanto as operações de escrita podem ser mais simples e focadas no domínio. A segregação dessas responsabilidades em diferentes modelos permite a otimização de cada um deles de acordo com seus próprios requisitos, o que pode levar a um aumento no desempenho e na escalabilidade do sistema.

2.4.4 Change Data Capture

CDC é uma técnica que permite a captura de alterações feitas em um banco de dados em tempo real. Isso é particularmente útil em arquiteturas de microsserviços, onde os dados podem ser distribuídos entre vários serviços (SHI *et al.*, 2008). Com o CDC, as alterações feitas em um serviço podem ser imediatamente refletidas em outros serviços, garantindo a consistência dos dados. Além disso, o CDC pode ser utilizado para alimentar sistemas de análise de dados em tempo real, permitindo *insights* mais oportunos e precisos. Neste trabalho, discutimos como o CDC é aplicado na arquitetura proposta, e como ele pode ser combinado com outras tecnologias, como o *Apache Kafka* e o *Debezium*, para criar uma solução de processamento de dados robusta e eficiente em tempo real.

2.5 Data Lake

Data Lakes surgiram como soluções abrangentes para os desafios apresentados pelo crescente volume, variedade e velocidade dos dados. Ao contrário dos armazéns de dados tradicionais, que exigem que os dados sejam estruturados e processados antes de serem ingeridos, *Data Lakes* permitem que as organizações armazenem dados não estruturados, semi-estruturados e estruturados em sua forma bruta (LAPLANTE, 2016). Essa flexibilidade facilita a análise e o processamento de dados em grande escala, oferecendo uma plataforma unificada para *insights* de negócios. O Hadoop Distributed File System (HDFS) é uma das tecnologias fundamentais frequentemente associadas a *Data Lakes*, proporcionando um sistema de arquivos distribuídos que suporta armazenamento e processamento de grandes volumes de dados (SHVACHKO *et al.*, 2010). Paralelamente, soluções de armazenamento em nuvem, como Amazon S3 (S3) e Google Cloud Storage (GCS), tornaram-se populares para a implementação de *Data Lakes* devido à sua escalabilidade, durabilidade e capacidade de integração com uma variedade de serviços e ferramentas de análise (Amazon Web Services, Inc., 2023; Google Cloud, 2023).

Combinando armazenamento flexível com ferramentas avançadas de processamento e análise, *Data Lakes* habilitam as organizações a extrair valor de seus dados de maneiras antes consideradas inviáveis ou muito caras.

2.6 Delta Lake

Delta Lake é uma camada de armazenamento de código aberto que traz confiabilidade para os *Data Lakes*. Desenvolvido pela *Databricks*, ele se posiciona como uma solução essencial para resolver muitos dos desafios enfrentados pelos *Data Lakes* tradicionais, como problemas de consistência de dados, eficiência de leitura e controle de transações.

Ele oferece transações Atomicidade, Consistência, Isolamento e Durabilidade (ACID), escalabilidade e desempenho de leitura otimizado para processos de *big data* (ARMBRUST *et al.*, 2020). As transações ACID garantem que as operações comumente de banco de dados sejam processadas de forma confiável e sem erros em *data lakes*. Isso é crucial para ambientes onde a confiabilidade e a integridade dos dados são primordiais.

Delta Lake permite que você faça operações de leitura e gravação em conjuntos de dados de grande escala, com controle de transações e consistência de dados. Ele também fornece esquemas que ajudam a garantir a qualidade dos dados, evitando problemas comuns como dados

corrompidos ou faltantes.

No contexto da arquitetura proposta, o *Delta Lake* foi usado para armazenar e gerenciar os dados processados pelo *Apache Spark*, fornecendo uma fonte única de verdade para os dados analíticos.

Uma das principais vantagens do *Delta Lake* em relação aos *Data Lakes* baseados apenas em arquivos *Parquet* é sua capacidade de gerenciar e manter um histórico de todas as transações. Isso permite operações como *"time-travel"* (viajar no tempo), onde os usuários podem acessar versões anteriores dos dados. Além disso, enquanto os arquivos Parquet são excelentes para armazenar dados em colunas, eles não têm mecanismos nativos para lidar com transações ou controle de versão. O *Delta Lake*, por outro lado, oferece esses recursos, tornando-o uma escolha superior para organizações que buscam confiabilidade e flexibilidade em seus *Data Lakes*.

2.7 Docker

O *Docker* é uma ferramenta de containerização *open-source* que permite aos desenvolvedores encapsular aplicações em contêineres, promovendo a independência entre software e hardware (DOCKER, 2023). Diferentemente de máquinas virtuais, que isolam todo um sistema operacional, o *Docker* trabalha isolando apenas o aplicativo e suas dependências, tornando-o mais leve e eficiente. O principal benefício desta abordagem é a portabilidade. Uma vez que um aplicativo é empacotado como um contêiner *Docker*, ele pode ser executado em qualquer ambiente que suporte o *Docker*, garantindo consistência em desenvolvimento, testes e produção.

Além da portabilidade, o *Docker* também promove a modularidade. Os serviços de uma aplicação podem ser contêinerizados individualmente, permitindo que sejam desenvolvidos, atualizados e escalonados de forma independente. No entanto, essa abordagem também apresenta desafios, como a necessidade de gerenciar e orquestrar múltiplos contêineres para garantir a integridade da aplicação.

A gestão de recursos pode se tornar particularmente desafiadora em ambientes de larga escala ou que manipulam grandes volumes de dados e aplicações. Neste contexto, é essencial contar com ferramentas e estratégias específicas para o monitoramento eficiente, alocação e ajuste de recursos para assegurar a performance e a estabilidade do sistema.

Exemplo de Código Docker: Um exemplo prático é a criação de um contêiner *Docker* para uma aplicação web simples. O arquivo ‘Dockerfile‘ pode parecer com o seguinte:

```
# Use uma imagem base oficial do Python
FROM python:3.8-slim

# Define o diretório de trabalho dentro do contêiner
WORKDIR /app

# Copia os arquivos da aplicação para o contêiner
COPY . /app

# Instala as dependências da aplicação
RUN pip install -r requirements.txt

# Define o comando para iniciar a aplicação
CMD ["python", "app.py"]
```

Este ‘Dockerfile‘ define um ambiente para uma aplicação *Python*, copiando os arquivos necessários para dentro do contêiner, instalando as dependências e configurando o comando para iniciar a aplicação.

2.7.1 Docker Compose

O *Docker Compose* é uma ferramenta associada ao *Docker*, projetada para definir e gerenciar aplicações multi-contêiner (Docker Inc., 2023). Ele permite que os desenvolvedores descrevam uma aplicação composta por múltiplos contêineres em um único arquivo de configuração, conhecido como ‘*docker-compose.yml*‘. Ao utilizar o comando ‘*docker-compose up*‘, todos os contêineres descritos são instanciados em sequência, mantendo as relações e dependências definidas.

Esta abordagem simplifica significativamente a implantação e o gerenciamento de aplicações complexas que dependem de vários serviços inter-relacionados. Além disso, o *Docker Compose* suporta variáveis de ambiente, volumes e redes personalizadas, permitindo flexibilidade na configuração e integração de serviços.

No contexto da arquitetura de microsserviços, o *Docker Compose* é particularmente útil durante o desenvolvimento e testes, pois permite que os desenvolvedores instanciem rapida-

mente todas as partes de uma aplicação em um ambiente local. No entanto, também pode ser uma ferramenta valiosa em ambientes de produção, especialmente para aplicações que exigem uma implantação rápida e coordenada de múltiplos contêineres com dependências interconectadas.

Exemplo de Docker Compose para Kafka:

```
version: '3'
services:
  zookeeper:
    image: zookeeper:3.4.9
    ports:
      - "2181:2181"
  kafka:
    image: wurstmeister/kafka
    ports:
      - "9092:9092"
    environment:
      KAFKA_ADVERTISED_HOST_NAME: localhost
      KAFKA_ZOOKEEPER_CONNECT: zookeeper:2181
    volumes:
      - /var/run/docker.sock:/var/run/docker.sock
```

Este exemplo de ‘docker-compose.yml‘ cria um ambiente *Kafka* com *Zookeeper*, facilitando a configuração e o *deploy* de um sistema de mensageria complexo. Após o exemplo de *Docker Compose* para Kafka, é importante destacar a diferença entre o *deploy* usando apenas o *Docker* e o *deploy* utilizando o *Docker Compose*. Sem o *Docker Compose*, a implantação de cada contêiner seria feita individualmente através de comandos *Docker* separados. Isso exigiria a configuração manual das redes, volumes e variáveis de ambiente para cada contêiner. Além disso, iniciar ou parar toda a aplicação demandaria a execução de múltiplos comandos em sequência, aumentando a complexidade e a chance de erros. Em contraste, o *Docker Compose* permite definir e gerenciar todos esses aspectos em um único arquivo, simplificando a implantação e gestão da aplicação como um todo.

2.8 Apache Spark

Apache Spark é um sistema de computação em *cluster* de código aberto que fornece API de alto nível em *Java*, *Scala*, *Python* e *R*, e um mecanismo otimizado de suporte geral que suporta gráficos de computação de dados em geral (ZAHARIA *et al.*, 2016). O *Spark* é conhecido por sua capacidade de processar grandes conjuntos de dados de forma rápida e eficiente, graças à sua arquitetura distribuída e à sua capacidade de realizar cálculos em memória. Além disso, o *Spark* suporta uma variedade de operações de processamento de dados, incluindo transformações de dados, agregações e operações de *machine learning*, tornando-o uma ferramenta versátil para o processamento de dados em grande escala. No contexto da arquitetura proposta, o *Apache Spark* é utilizado para processar e analisar os dados capturados pelo *Debezium* e transmitidos pelo *Apache Kafka*.

2.9 Debezium

Debezium é uma plataforma de código aberto para a realizacao de CDC(DEBEZIUM, n.d.a). Ele permite capturar e transmitir alterações feitas em um banco de dados em tempo real, tornando-o uma ferramenta valiosa para manter a consistência dos dados em arquiteturas de microserviços. O *Debezium* é projetado para ser usado com uma variedade de sistemas de banco de dados, incluindo *MySQL*, *MongoDB* e *PostgreSQL*, e pode transmitir alterações de dados para uma variedade de destinos, incluindo *Apache Kafka*. No contexto da arquitetura proposta, o *Debezium* é utilizado para capturar as alterações nos dados de origem, utilizando a técnica de *Log-based CDC*, que será mencionada com mais detalhes na seção 5.3.0.1, e transmiti-las para o *Apache Kafka* para processamento posterior.

2.10 Alluxio

Alluxio, anteriormente conhecido como Tachyon, é um sistema de armazenamento de dados em memória, de código aberto, orientado ao desempenho e com tolerância a falhas. Ele é projetado para melhorar a velocidade de aplicações de *big data* e se integra de maneira transparente com várias estruturas de cálculo distribuídas, aproveitando a velocidade do armazenamento em memória, independentemente da infraestrutura de armazenamento subjacente.

A arquitetura do *Alluxio* consiste em um *master* e múltiplos *workers*. O *master* é responsável por manter o estado global do sistema, como metadados, enquanto os *workers*

gerenciam o armazenamento e o acesso aos dados. Contrariamente ao mencionado anteriormente, cada instância do *worker* e do *master* são geralmente implantadas em nós separados para isolamento de recursos e falhas.

Alluxio não é estritamente um sistema de arquivos; ele é mais bem descrito como um sistema de gerenciamento de dados virtualizado que oferece uma visão unificada e global dos dados para aplicações. Ele virtualiza diferentes fontes de dados, permitindo que sejam acessadas através de uma API comum.

Alluxio é compatível com uma série de sistemas de armazenamento distribuídos, como HDFS, S3 e GCS. Ele serve como uma camada de aceleração de dados, possibilitando que os dados sejam acessados e manipulados em alta velocidade, beneficiando-se do armazenamento em memória.

Alluxio é utilizado em conjunção com várias estruturas de processamento de dados populares, como *Apache Spark*, *Apache Hadoop*, e *TensorFlow*. Ele facilita o compartilhamento eficiente de dados em memória entre *jobs* e *tasks*, reduzindo a latência e acelerando o processamento de dados.

A capacidade do *Alluxio* de se integrar com diversos sistemas de armazenamento e computação, juntamente com sua escalabilidade e eficiência, o torna uma peça crucial na otimização de pipelines de dados de *big data* e ambientes de computação distribuída. (LI *et al.*, 2014).

2.11 Apache Kafka

Apache Kafka é uma plataforma de *streaming* de eventos distribuída de código aberto (KREPS *et al.*, 2011). Ele é projetado para lidar com fluxos de dados em tempo real em grande escala, tornando-o uma escolha popular para a construção de *pipelines* de dados em tempo real e aplicações de *streaming*. O *Kafka* permite a publicação e a subscrição de fluxos de registros, semelhante a uma fila de mensagens, mas com melhor taxa de transferência, particionamento integrado, replicação e tolerância a falhas. No contexto da arquitetura proposta, o *Apache Kafka* é utilizado para facilitar a comunicação entre os serviços, transmitindo as alterações de dados capturadas pelo *Debezium* para o *Apache Spark* processar.

2.12 Confluent Schema Registry

O *Confluent Schema Registry* é uma ferramenta que fornece um serviço de armazenamento centralizado para os esquemas de dados utilizados em aplicações Kafka e Debezium. Ele permite que os produtores e consumidores de dados verifiquem que os dados estão sendo escritos e lidos usando um esquema compatível, o que pode ajudar a prevenir erros de incompatibilidade de dados. Além disso, o *Confluent Schema Registry* permite a evolução segura dos esquemas ao longo do tempo, o que é crucial para a manutenção de sistemas de dados em grande escala. No contexto da arquitetura proposta, o *Confluent Schema Registry* é utilizado para gerenciar os esquemas de dados usados pelos serviços de microsserviços, garantindo a consistência e a compatibilidade dos dados à medida que eles são transmitidos entre os serviços. (Confluent Inc., 2023)

2.13 Apache Avro

Apache Avro é um *framework* de serialização de dados que produz dados compactos, rápidos e binários. O *Avro* usa um esquema para estruturar os dados que estão sendo codificados. Ele armazena os esquemas em JavaScript Object Notation (JSON), o que torna os esquemas fáceis de ler e interpretar por humanos (VOHRA; VOHRA, 2016). No contexto da arquitetura proposta, o *Apache Avro* é usado para serializar os dados antes de serem transmitidos através do *Apache Kafka*. Isso pode ajudar a garantir a consistência dos dados e a compatibilidade entre diferentes partes da arquitetura, assim como diminuir o custo de *storage*.

3 TRABALHOS RELACIONADOS

3.1 Trabalhos Relacionados

Nesta seção, discutiremos trabalhos relacionados que abordam a migração de aplicações monolíticas para microserviços e a otimização de consultas em banco de dados, ambos relevantes para a aplicação do *Strangler Pattern*.

3.1.1 Migrating from Monolithic Application to Microservices (TRAN, 2020)

O trabalho de (TRAN, 2020) aborda a migração de uma aplicação monolítica para microserviços utilizando o *Strangler Pattern*. A estratégia proposta é semelhante à adotada no artigo base, com ambos os trabalhos defendendo a utilização do *Strangler Pattern* para a migração de sistemas legados para uma arquitetura de microserviços.

No entanto, (TRAN, 2020) destaca a importância de um *sistema proxy* para redirecionar o tráfego durante o processo de migração, um aspecto não explicitamente mencionado no artigo base. Além disso, o autor discute a resiliência e a mitigação de riscos durante o processo de migração, destacando a eficácia do *Strangler Pattern* em manter a disponibilidade do sistema durante a migração.

Em comparação com o artigo base, o trabalho de (TRAN, 2020) não aborda a utilização de tecnologias específicas como *Apache Spark*, *Debezium* e *Apache Kafka* para gerenciar grandes volumes de dados. No entanto, ressalta a importância da comunicação e do consenso entre os membros da equipe durante o processo de migração, um aspecto que não é abordado no artigo base.

Em suma, ambos os trabalhos oferecem perspectivas valiosas sobre a aplicação do *Strangler Pattern* na migração de sistemas legados para uma arquitetura de microserviços. No entanto, eles diferem em seus focos específicos, com o artigo base concentrando-se mais na implementação técnica e no gerenciamento de grandes volumes de dados, enquanto (TRAN, 2020) enfatiza mais a gestão do processo de migração e a mitigação de riscos.

3.1.2 Strategies to Mitigate Anti-patterns in Microservices Before Migrating from a Monolithic System to Microservices (VILLACA et al., 2022)

O trabalho de (VILLACA *et al.*, 2022) apresenta uma análise profunda sobre a migração de sistemas monolíticos para microserviços, com foco na mitigação de anti-padrões. O autor destaca o uso do *Strangler Pattern* como uma estratégia popular para transformar gradualmente uma aplicação monolítica em microserviços, substituindo uma funcionalidade específica por um novo serviço. Uma vez que a nova funcionalidade está pronta, o antigo componente é "estrangulado". O autor também menciona o uso do *Strangler Pattern* em conjunto com o padrão anti-corrupção. A principal vantagem do *Strangler Pattern*, segundo o autor, é que ele oferece um processo de migração incremental sem quebrar a funcionalidade completa da aplicação.

Comparando com o trabalho base, ambos os trabalhos concordam sobre a eficácia do *Strangler Pattern* na modernização de sistemas legados. No entanto, o trabalho de (VILLACA *et al.*, 2022) vai além ao discutir a mitigação de anti-padrões e a aplicação do padrão anti-corrupção, enquanto o trabalho base se concentra mais na implementação do *Strangler Pattern* em contextos que gerenciam grandes volumes de dados, usando tecnologias como *Apache Spark*, *Debezium* e *Apache Kafka*.

Em relação à aplicação do *Strangler Pattern*, o trabalho base propõe uma arquitetura específica para sua implementação em contextos de alto volume de dados, enquanto o trabalho de (VILLACA *et al.*, 2022) discute o *Strangler Pattern* como uma estratégia geral para a migração de sistemas monolíticos para microserviços, sem se aprofundar em um contexto específico de alto volume de dados. No entanto, ambos os trabalhos concordam que o *Strangler Pattern* utilizando o CDC como estratégia de integração é uma valiosa combinação.

3.1.3 Relational Database Query Optimization Strategy Based on Industrial Internet Situation Awareness System (YAO et al., 2022)

No trabalho de (YAO *et al.*, 2022), é apresentada uma estratégia que utiliza o mecanismo de CDC para otimizar consultas em bancos de dados relacionais. Esta estratégia é particularmente útil para bancos de dados que possuem mais de um milhão de registros em uma única tabela e múltiplos campos. A abordagem emprega a tecnologia CDC baseada em *log* para monitorar alterações de dados no banco de dados relacional. As informações capturadas

são enviadas para a fila de mensagens *Kafka* e, posteriormente, consumidas no projeto, com todas as condições de recuperação sendo escritas em *Elasticsearch*. A pesquisa é realizada inicialmente em *Elasticsearch* através das condições de busca, seguida de uma busca no banco de dados relacional usando o ID de chave primária obtido do *Elasticsearch*, resultando em dados completos. Experimentos demonstram que a estratégia baseada em CDC pode efetivamente acelerar a recuperação de dados em bancos de dados relacionais.

Em comparação com o trabalho base, (YAO *et al.*, 2022) também foca na otimização de bancos de dados, porém com uma abordagem distinta. Enquanto o trabalho base propõe uma arquitetura para a aplicação do *Strangler Pattern* com CDC em setores com alto volume de dados, (YAO *et al.*, 2022) apresenta uma estratégia de otimização de consulta para bancos de dados relacionais que utiliza CDC. Ambos os trabalhos reconhecem a importância do CDC para o gerenciamento de grandes volumes de dados, mas aplicam essa técnica de maneiras diferentes para otimizar o desempenho dos bancos de dados.

4 METODOLOGIA

Este capítulo tem por finalidade detalhar a metodologia utilizada para a construção da arquitetura proposta neste trabalho, bem como para sua aplicação em um estudo de caso. Cada etapa que compõe a referida metodologia será discutida em detalhes.

4.1 Etapa 1: Definição de Pré-requisitos

Inicialmente, os pré-requisitos arquiteturais foram delineados por uma equipe multidisciplinar, integrando a visão estratégica do Chief Technology Officer (CTO) e do diretor de produto com *insights* operacionais do time de *analytics*, expertise técnica da equipe de *DevOps* e *feedback* direto dos consultores de *customer success*. Esse processo colaborativo garantiu que as necessidades do cliente fossem refletidas nos *dashboards* a serem desenvolvidos.

Ao longo de duas semanas, reuniões diárias foram realizadas para estabelecer um conjunto abrangente de requisitos e o escopo completo do projeto. Estas sessões intensivas asseguraram um entendimento compartilhado e alinhado dos objetivos do projeto, resultando em pré-requisitos bem definidos que serviram como alicerce para o desenvolvimento subsequente da arquitetura proposta. A seguir, descrevemos esses pré-requisitos:

1. O projeto deverá desacoplar as consultas analíticas do banco de dados principal.
2. Não poderia haver nenhuma conexão que impactasse os bancos de dados de produção.
3. O máximo de tempo que o dado precisaria estar disponível era de 15 segundos, visto que era um requisito do cliente já que ele queria ter o mesmo comportamento se ele estivesse fazendo a consulta no seu próprio banco de dados de produção.
4. Os dados têm que estar em uma outra maneira a qual seja fácil para outros microsserviços consumirem que não seja fazer uma consulta no banco de dados de produção.
5. Todo o processo deveria ser paralelo, não devendo haver relação de sincronia entre múltiplos contextos.
6. O projeto deveria ser responsável por montar o *data lake* da empresa, desacoplando os dados dos bancos de dados existentes e colocando em um ambiente onde possibilitasse análises mais fáceis e interação melhor com esses dados.
7. O objetivo final era que todos os dados processados e transformados estivessem disponíveis em um banco de dados *Aurora Postgres*(VERBITSKI *et al.*, 2017), uma solução já estabelecida na empresa para alimentar *dashboards* empresariais com alta performance de

Figura 3 – Infográfico Metodologia e definição - Visão do Cliente. Este infográfico foi especificamente elaborado e utilizado para apresentação aos clientes, visando ilustrar claramente a metodologia e definições relevantes. Fonte: Elaborado pelo autor.

consulta.

Com base nisso, foram realizados estudos para a pesquisa de ferramentas e processos nas seguintes áreas:

1. Estratégia de integração e ferramenta para replicação.
2. Ferramenta de processamento paralelo e geração dos arquivos *raw* do data lake.
3. Ferramenta de distribuição dos dados para os microsserviços.

4.2 Etapa 2: Definição da Estratégia de Integração e da Ferramenta para Replicação

Conforme delineado no capítulo de fundamentação teórica, a integração eficaz de dados é um pilar fundamental para o sucesso de sistemas modernos, sobretudo em arquiteturas orientadas a eventos. A escolha de CDC como estratégia de integração, em específico o uso do *Debezium*, foi fundamentada na necessidade de uma solução robusta que pudesse capturar mudanças de dados em tempo real sem impactar a performance do banco de dados.

Esta decisão foi reforçada pelos requisitos críticos do projeto, onde a consistência e disponibilidade dos dados são essenciais. O CDC oferece a vantagem de replicar dados entre sistemas de maneira confiável e eficiente, características imprescindíveis para a integridade e resiliência do projeto. O *Debezium*, sendo a única opção *open source* viável na época que realizava CDC de bancos de dados *Oracle*, destacou-se não apenas pela sua capacidade de replicação sem intrusão, mas também por sua ampla comunidade e documentação robusta, como citado em (DEBEZIUM, n.d.b) e corroborado pelas experiências compartilhadas em (REORCHESTRATE, n.d.).

Esta escolha estratégica visou também o alinhamento com a filosofia de código aberto da organização, proporcionando a flexibilidade de customizações. A integração do *Debezium* com práticas de *DevOps*, incluindo a conteinerização como característica padrão, enfatiza a abordagem proativa no gerenciamento de dados, facilitando a manutenção e a escalabilidade futura do sistema.

4.3 Etapa 3: Seleção das Ferramentas para Processamento Paralelo a serem Avaliadas e Construção do Data Lake

A seleção de uma ferramenta adequada para processamento paralelo é crítica para a eficiente integração de dados ao nosso *data lake*, permitindo a execução de consultas analíticas

avançadas e extrair *insights* conforme estipulado pelo requisito 5. A ferramenta ideal precisa suportar Structured Query Language (SQL) para facilitar a análise e a extração de informação.

Com o objetivo de satisfazer essa demanda crítica e aderir ao requisito 4, nossa equipe avaliou três ferramentas de ponta: *Apache Spark*, *Apache Flink*, e *KSQLDB*. Cada ferramenta foi analisada por seu potencial de processamento paralelo e capacidades analíticas, elementos essenciais para o funcionamento do *data lake*.

O processo de seleção foi meticuloso, liderado pelo time de *analytics* sob a orientação do CTO, envolvendo uma semana de pesquisa intensiva em literatura acadêmica e *benchmarks* de mercado, além da consulta de soluções internas já aplicadas em desafios semelhantes. A experiência dos desenvolvedores seniores de outros times também foi uma fonte valiosa de conhecimento, verificando ferramentas existentes e recomendando soluções adequadas para atender aos nossos requisitos específicos.

4.3.1 Etapa 4: Desenvolvimento de uma Prova de Conceito

A Prova de Conceito (POC) foi um passo crítico para validar a arquitetura proposta. O objetivo era processar dados capturados pelo *Debezium* em formato *Avro* e convertê-los para o formato *Delta Lake* no *data lake* hospedado no S3. A POC exigiu a replicação de eventos de *insert*, *delete* e *update* de múltiplas tabelas, armazenando-os no *data lake* de maneira a refletir a estrutura do banco de dados de origem no *Oracle*.

O desafio consistia em garantir que o processamento fosse paralelo e assíncrono, capaz de lidar com a replicação de vários bancos de dados simultaneamente, e que os dados replicados fossem enviados de volta para um banco de dados *PostgreSQL* em até 15 segundos, como definido no pré requisito 3. Este rápido ciclo de replicação era essencial para assegurar a integridade e a atualidade dos dados no sistema de replicação, refletindo fielmente o pipeline da arquitetura proposta.

Para isso cada desenvolvedor do time de *analytics* foi responsável por replicar todo o processo com uma ferramenta diferente, avaliar, metrificar e apresentar os resultados do seu desenvolvimento. Diariamente os desenvolvedores tinham um fórum para trocar suas experiências e ajudar entre si no desenvolvimento para, assim, compartilhar o conhecimento adquirido, experiencias, e permitir com que a equipe tivesse um nível de conhecimento parelho nas ferramentas.

4.3.2 Etapa 5: Seleção da Ferramenta de Replicação

Os critérios considerados para a tomada de decisão incluíram a curva de aprendizagem, documentação, comunidade e adequação ao problema em questão. Na tabela abaixo temos quatro critérios principais: curva de aprendizagem, documentação, comunidade e adequação ao problema. Cada critério foi avaliado numa escala de 0 a 5, onde 5 indica uma alta qualidade ou adequação, e 0 uma qualidade ou adequação muito baixa.

Em particular, para a curva de aprendizagem, uma pontuação mais baixa indica uma curva de aprendizagem mais íngreme, significando que é mais difícil aprender a usar a ferramenta.

Como supracitado, o processo de metrificação e avaliação de cada uma delas ficou a par de cada desenvolvedor individualmente e ao final de todo processo houve uma discussão geral para refinamento das notas com base em toda a experiencia prévia compartilhada entre os desenvolvedores do processo e também inferências individuais de acordo com todo o processo.

Por fim as subseções a seguir discutem a avaliação de cada uma das ferramentas.

Tabela 1 – Critérios para seleção da ferramenta

Ferramenta	Curva de Aprendizagem	Documentação	Comunidade	Adequação ao Problema	Aprovado
Apache Spark	3	4	5	5	Aprovado
Apache Flink	1	3	2	5	Não Aprovado
KSQL DB	5	5	3	0	Não Aprovado

4.3.3 Etapa 5: Seleção da Ferramenta de Replicação

A escolha da ferramenta de replicação foi um processo meticuloso, baseado em quatro critérios principais: curva de aprendizagem, qualidade da documentação, suporte da comunidade e adequação ao problema específico do projeto. A avaliação foi feita em uma escala de 0 a 5, com 5 representando alta qualidade ou adequação e 0 o oposto.

- **Curva de Aprendizagem:** Esse critério avaliou a facilidade de adquirir habilidades para utilizar a ferramenta. Uma pontuação baixa aqui indicou uma curva de aprendizagem

íngreme, o que significaria mais tempo e recursos investidos em treinamento.

- **Qualidade da Documentação:** Avaliou a clareza, completude e atualização dos materiais de apoio da ferramenta, um aspecto crucial para a resolução ágil de problemas e implementação eficiente.

- **Suporte da Comunidade:** Considerou a atuação e disponibilidade da comunidade de usuários e desenvolvedores da ferramenta, um recurso valioso para suporte técnico e compartilhamento de melhores práticas.

- **Adequação ao Problema:** Analisou o quão bem a ferramenta atendia às necessidades específicas do projeto, incluindo compatibilidade com a infraestrutura existente e requisitos de desempenho.

Cada membro da equipe de desenvolvimento realizou avaliações individuais, seguidas de discussões em grupo para refinar as notas. Esta abordagem colaborativa permitiu uma visão ampla e equilibrada, combinando experiências individuais com perspectivas coletivas.

4.3.3.1 Avaliação do Apache Spark

A análise detalhada do *Apache Spark* revelou que ele cumpria eficazmente a maioria dos pré-requisitos. Um aspecto notável foi a robusta documentação, apoiada por uma comunidade ativa, que facilitou o desenvolvimento da POC. Sua API unificada, que suporta *Python*, *Java* e *Scala*, em conjunto com o *Spark SQL*, proporcionou uma experiência de desenvolvimento intuitiva e eficiente, tornando-o a escolha preferencial para a ingestão de dados.

No entanto, ao explorar o módulo de *Structured Streaming* do *Spark*, foi identificado desafios relacionados à otimização do processo para usuários iniciantes. Isso indicou a necessidade de um investimento adicional em treinamento e aprendizado para aproveitar plenamente suas capacidades avançadas. A curva de aprendizado mais acentuada para esses recursos específicos foi um ponto de atenção, requerendo uma avaliação cuidadosa sobre o equilíbrio entre a eficácia da ferramenta e o esforço de capacitação da equipe.

4.3.3.2 Avaliação do Apache Flink

Nossa avaliação inicial do *Apache Flink* para integração de dados via *streaming* de tópicos *Kafka* apresentou-se promissora, particularmente porque a pesquisa inicial indicava o *Flink* como uma solução líder de mercado para *streaming* de dados em comparação com seus concorrentes (GARCÍA-GIL *et al.*, 2017)(VERBITSKIY *et al.*, 2016)(PERERA *et al.*,

2016). Contudo, durante a POC, múltiplos desafios foram encontrados. Um ponto crítico foi a falta de documentação aprofundada e acessível, com várias seções marcadas como "TODO"e conteúdo ainda não traduzido do mandarim para o inglês. Além disso, a comunidade ocidental do *Flink* mostrou-se menos ativa em comparação com a oriental, limitando o acesso a recursos, comunidades e artigos relevantes.

Esses fatores, somados à curva de aprendizado naturalmente elevada do *Flink* especialmente quando comparada ao *Apache Spark*, o que já atua quase como um consenso quando procurado na comunidade e pode ser visto em (WHAT..., 2023b)(THE..., 2023) (WHAT..., 2023a)levaram à decisão de não adotar esta ferramenta para o nosso projeto. Por fim, é válido salientar que a falta de suporte robusto e documentação detalhada em inglês mostrou-se um obstáculo significativo, impactando a eficiência e progresso da POC.

4.3.3.3 Avaliação do KSQLDB

Ao trabalhar com dados do *Debezium* no Kafka e visando integrá-los ao nosso *data lake*, o *ksqlDB* inicialmente parecia uma escolha natural, especialmente por sua integração nativa com o *Kafka*. No entanto, um desafio significativo surgiu: o *ksqlDB* não tratava de maneira eficiente os *tombstone records*. Estes são registros especiais em sistemas de *streaming* que indicam a deleção de dados. Eles são caracterizados por terem uma chave que corresponde ao dado deletado e um valor ‘null‘, sinalizando a remoção do registro.

Considere o seguinte cenário com o Debezium:

Exemplo de Registro Normal:

```
{
  "key": {
    "delivery_id": 2001
  },
  "value": {
    "delivery_id": 2001,
    "package_id": "PKG123456",
    "destination": "123 Main St, Anytown",
    "status": "Dispatched",
    "last_updated": "2023-03-15T12:00:00Z"
  }
```

```
}
```

Exemplo de Tombstone Record Após Deleção:

```
{
  "key": {
    "delivery_id": 2001
  },
  "value": null
}
```

Neste exemplo, um registro normal no *Kafka* indica uma entrega programada. No entanto, se essa entrega for cancelada e o registro correspondente deletado do banco de dados, o *Debezium* gera um *tombstone record* com a mesma chave (`"delivery_id": 2001`) e um valor `null`. A incapacidade do *ksqlDB* de processar corretamente esses registros significava que tais deleções não seriam refletidas no nosso *data lake*, levando a dados desatualizados ou inconsistentes. Essa limitação foi um fator crucial na decisão de descartar o uso do *ksqlDB* para nosso projeto.

5 ARQUITETURA PROPOSTA

Neste capítulo, apresentaremos a arquitetura proposta, detalhando o fluxo de dados e as especificidades de cada um de seus componentes. A arquitetura é estruturada em três camadas principais: ingestão, processamento e distribuição, e exibição.

Um importante *disclaimer* a ser feito é que, embora os componentes desta arquitetura tenham sido implantados na Amazon Web Services (AWS), não há impedimentos para que ela seja replicada em outros tipos de ambientes. Isso inclui ambientes locais ou em outras plataformas de nuvem, demonstrando a versatilidade e adaptabilidade da arquitetura em diferentes contextos tecnológicos.

5.1 Fluxo dos Dados

A Figura 4 ilustra os componentes da arquitetura proposta e o fluxo de dados. O processo inicia-se com a ingestão de dados do banco de origem para o *Apache Kafka*, utilizando o *Debezium*. Em seguida, o *Kafka S3 Sink Connect* lê os tópicos do *Debezium* e os armazena no S3 convertendo-os em *parquet*. O *Apache Spark* então lê esses *buckets* do *S3* via *Structured Streaming* e recria as tabelas baseadas nos logs de replicação gerados pelo *Debezium*, utilizando o *Delta Lake*. Simultaneamente, o *Spark* envia os mesmos dados para o *Kafka*.

Na etapa final, os dados são consumidos pelo *Kafka Postgres Sink Connector* e inseridos no *Postgres*, que serve como o banco de dados do microsserviço.

Esta arquitetura permite uma implementação eficiente do *Strangler Pattern*, uti-

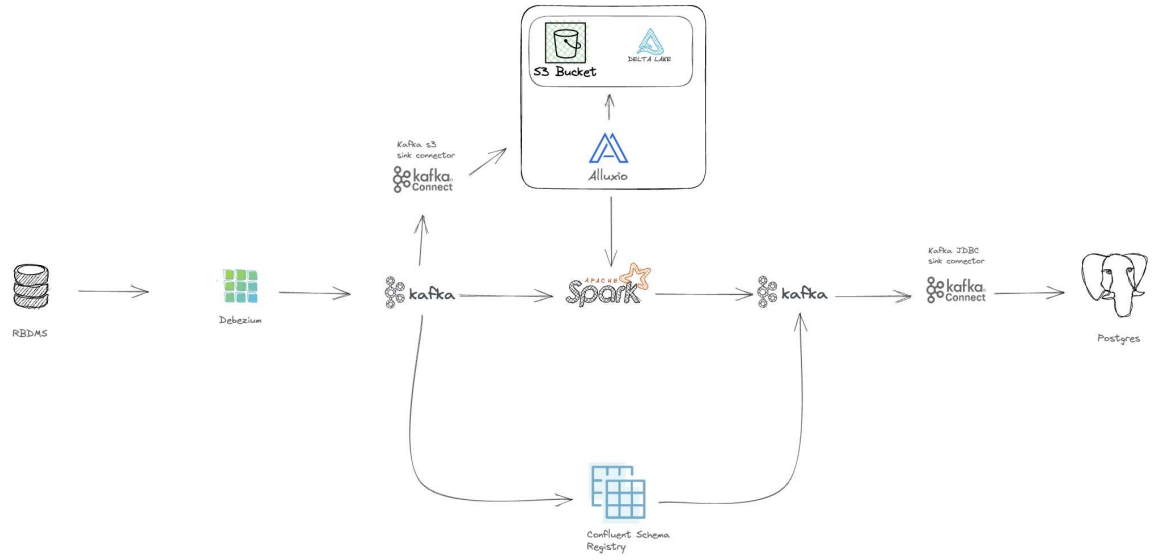

Figura 4 – Proposta de arquitetura. Fonte: Elaborado pelo autor.

lizando o CDC como estratégia de integração. A sua natureza modular e a utilização de componentes amplamente adotados e suportados pela comunidade, como o *Apache Kafka* e o *Apache Spark*, tornam a arquitetura altamente escalável e flexível. Ela pode ser adaptada para diferentes cargas de trabalho, volumes de dados e requisitos de latência, atendendo às necessidades de ambientes de desenvolvimento variados. Além disso, a flexibilidade na escolha dos componentes de armazenamento e processamento de dados permite que a arquitetura seja ajustada para maximizar o desempenho e a eficiência em diferentes infraestruturas, sejam elas na nuvem ou locais, proporcionando uma solução robusta e adaptável para diversos cenários de uso.

5.2 Debezium

Vamos analisar como o *Debezium* funciona com um exemplo do seu *payload*. Suponha que temos uma mudança de dados no banco de dados *Oracle*, que o *Debezium* está monitorando. O *payload* gerado pelo *Debezium* pode se parecer com isto:

```
{
  "before": {
    "id": "1001",
    "first_name": "John",
    "last_name": "Doe",
    "email": "john.doe@email.com"
  },
  "after": {
    "id": "1001",
    "first_name": "John",
    "last_name": "Doe",
    "email": "johnny.doe@email.com"
  },
  "source": {...},
  "op": "u",
  "ts_ms": 1589362330904
}
```

Neste ponto, gostaria de introduzir o conceito de *Single Message Transformation*

(Single Message Transformation (SMT))(SINGLE..., 2023). SMT são usadas no *Kafka Connect* para modificar mensagens que fluem do produtor para o consumidor. Elas podem alterar tanto os dados quanto o esquema da mensagem.

Em nosso caso de uso, estamos interessados apenas no estado final dos dados modificados. Para essa finalidade, podemos aplicar uma SMT específica chamada *UnwrapFromEnvelope*. Esta SMT remove a estrutura de envelope do *payload Debezium*, proporcionando diretamente o estado final dos dados.

Aplicando a SMT *UnwrapFromEnvelope*, o *payload* se transforma em:

```
{
  "id": "1001",
  "first_name": "John",
  "last_name": "Doe",
  "email": "johnny.doe@email.com",
  "op": "u",
  "ts_ms": 1589362330904
}
```

Se a arquitetura exigir o rastreamento do estado anterior dos dados, pode-se simplesmente omitir a transformação *UnwrapFromEnvelope*.

A configuração do conector *Oracle Debezium* com a SMT *UnwrapFromEnvelope* ativada seria:

```
{
  "name": "inventory-connector",
  "config": {
    "connector.class": "io.debezium.connector.oracle.OracleConnector",
    "database.hostname": "oracle-db-server",
    "database.port": "1521",
    "database.user": "c##asgard",
    "database.password": "asgard",
    "database.dbname": "ORCLCDB",
    "database.pdb.name": "ORCLPDB1",
    "database.server.name": "oracle-server-1",
```

```
    "database.history.kafka.bootstrap.servers": "kafka:9092",
    "database.history.kafka.topic": "schema-changes.inventory",
    "table.include.list": "inventory.customers",
    "transforms": "unwrap",
    "transforms.unwrap.type": "io.debezium.transforms.UnwrapFromEnvelope"
  }
}
```

Essa configuração permite monitorar e transmitir eficientemente as alterações de dados para um sistema de mensagens como o *Kafka*.

Por fim, destaco que o *Debezium* é implantado via *docker compose* e gerenciado no Amazon Elastic Container Service (ECS). Utiliza-se o *Kafka S3 Sink Connector* para capturar dados do *Debezium*, transformá-los em *parquet* e, assim, criar a camada *raw* do nosso *data lake*. Detalhes adicionais sobre o funcionamento do *kafka connect* serão explorados na seção 5.3.4.

5.3 *Apache Spark*, *Structured Streaming* e *Delta Lake*

Revisitando nossa discussão sobre o *Apache Spark*, é válido salientar novamente que ele é um *framework* de computação em *cluster* de alta performance, oferecendo rapidez, facilidade de uso e uma variedade de ferramentas para processamento de grandes volumes de dados. Em nossa proposta de arquitetura, vamos focar em uma funcionalidade específica do *Spark*: o *Structured Streaming*.

5.3.1 Structured Streaming

O *Structured Streaming* é uma extensão da API *Spark SQL* que permite processamento de fluxos de dados de maneira contínua. Ele provê uma interface de alto nível para expressar operações de *streaming*, permitindo ao mecanismo de execução do *Spark* realizar estas operações de maneira incremental e intercalada em fluxos de dados ao vivo.

Este recurso torna o *Apache Spark* uma solução ideal para operações que exigem análise de dados em tempo real. Ele oferece vantagens significativas, incluindo facilidade de uso, tolerância a falhas e integração robusta com todas as funcionalidades do *Spark*, como *MLlib*, *GraphX*, *Spark SQL* e também com o *Delta Lake*.

É o *Structured streaming*, o responsável em nossa arquitetura por processar os dados gerados pelo *Debezium* e postos no S3 pelo *Kafka S3 Sink Connector*.

Por fim, nosso deploy do *Apache Spark* era gerenciado pelo Amazon Elastic MapReduce (EMR) (EMR, 2023).

5.3.2 Delta Lake

O *Delta Lake* emerge como um componente crítico na nossa arquitetura proposta, servindo como um formato de armazenamento de dados *open-source* que eleva a confiabilidade dos *Data Lakes*. Projetado para proporcionar transações ACID, a *Delta Lake* assegura a integridade dos dados, mesmo em ambientes complexos e de alta concorrência, onde os dados são continuamente modificados.

Com uma capacidade aprimorada para escalar metadados de forma eficiente, o *Delta Lake* se destaca por permitir operações de modificação de dados como *INSERT*, *UPDATE*, e *DELETE* em grandes volumes de dados com eficiência e precisão. O comando `MERGE INTO` é particularmente notável por sua capacidade de combinar e sincronizar conjuntos de dados, garantindo que as modificações sejam aplicadas de maneira ordenada e consistente.

Considerando um cenário prático no contexto de *Supply Chain*, o *Delta Lake* demonstra sua utilidade em manter a integridade e atualidade dos dados. Suponha uma tabela *Delta* que contém informações sobre as rotas de entrega, e esta necessita ser atualizada regularmente para refletir as mudanças em tempo real. A operação pode ser ilustrada como:

```
1  val updatesDF = spark.sql("SELECT * FROM updates_routes")
2
3  val deltaTable = DeltaTable.forPath(spark, "/path/to/delta/
       routes")
4
5  deltaTable.alias("routes")
6  .merge(
7  updatesDF.alias("updates"),
8  "routes.id = updates.id")
9  .whenMatched("updates.op = 'u'")
10 .update(Map("value" -> "updates.value"))
11 .whenMatched("updates.op = 'd'")
```

```
12 .delete()
13 .whenNotMatched("updates.op = 'i'")
14 .insert(Map("id" -> "updates.id", "value" -> "updates.value
      "))
15 .execute()
```

Neste exemplo, cada registro no *DataFrame* de atualização `updatesDF` contém uma operação de modificação especificada pela coluna `op`: 'i' indica uma inserção, 'u' uma atualização, e 'd' uma deleção. A tabela *Delta* de rotas, por sua vez, é modificada conforme a operação especificada, garantindo que os dados sejam atualizados ou excluídos conforme necessário, ou que novos registros sejam inseridos com precisão e eficiência.

Esta funcionalidade ressalta a flexibilidade e a potência do *Delta Lake* em suportar um ecossistema de dados dinâmico e em evolução, garantindo que os *Data Lakes* mantenham a integridade, confiabilidade e relevância dos dados em aplicações do mundo real.

5.3.3 Alluxio

Como mencionado na Seção 2.1, o *Alluxio* é um Sistema de Arquivos Virtual Distribuído (Virtual Distributed File System (VDFS)) que facilita o compartilhamento e a gestão de dados. Ele cria um barramento de dados que oferece flexibilidade e eficiência no desenvolvimento. Nesta arquitetura, destacam-se duas funcionalidades principais do *Alluxio*:

Namespace Unificado:

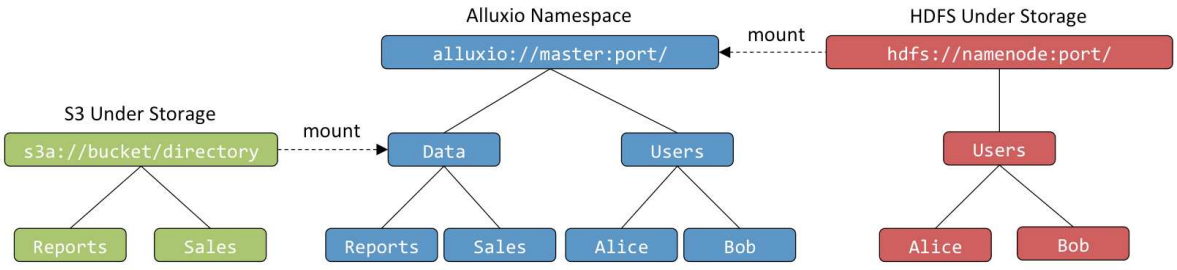

Figura 5 – Figura mostrando como funcionam os endpoints unificados na prática. Fonte: https://docs.alluxio.io/os/user/stable/img/screenshot_unified.png.

O *Alluxio* estabelece um *endpoint unificado* (ALLUXIO..., 2023b) para todos os sistemas de arquivos montados, simplificando a integração. Por exemplo, ao montar um caminho S3 no *Alluxio*, ele sincroniza continuamente os metadados do *bucket* S3 para manter a

atualização constante. A integração com o *Spark* potencializa essa funcionalidade, permitindo que o *Alluxio* armazene *checkpoints*, desacoplando essa tarefa do S3 e reduzindo custos, conforme será detalhado na seção 5.6.

Cache Multi-Nível:

O *Alluxio* otimiza a leitura de dados através de sua função de cache (ALLUXIO. . . , 2023a). Ele lê arquivos uma única vez, armazenando-os em *cache* para acessos futuros. Esse cache é escalável, podendo ser armazenado tanto na memória quanto em disco (*HD* ou *SSD*). Assim, após a primeira solicitação ao S3, todas as subsequentes, incluindo as de *checkpoint*, são direcionadas ao *Alluxio*.

5.3.4 Kafka e Kafka Connect

Após o processamento dos dados pelo *Spark* e sua passagem pelo *Alluxio*, os dados são novamente enviados para o *Kafka*. O *Kafka* é uma plataforma de *streaming* de eventos distribuída que é usada para construir *pipelines* de dados em tempo real e aplicativos de *streaming*. Ele é capaz de lidar com trilhões de eventos por dia e fornece uma solução robusta e resiliente para o processamento de fluxos de dados.

Em nossa arquitetura, o *Kafka* funciona em conjunto com o *Kafka Connect*, uma plataforma escalável e *stand-alone* que simplifica a integração de dados com o *Kafka*. O *Kafka Connect* possui uma série de conectores que permitem mover dados entre o *Kafka* e uma variedade de sistemas de armazenamento, incluindo bancos de dados *SQL*, *NoSQL*, *caches*, sistemas de busca e muito mais.

Uma parte crucial desse processo é o *Kafka Sink Connector*, que é responsável por transmitir os dados do *Kafka* para o sistema de armazenamento de destino, realizando as operações de inserção, atualização e exclusão no banco de dados. O conector *Sink* lê os registros de dados do *Kafka* e, em seguida, traduz esses registros em operações que são aplicadas ao sistema de armazenamento de destino.

Para representar os dados de maneira eficiente e consistente, optamos pelo uso de *Avro*, um formato de serialização de dados que permite a compactação e a verificação de esquemas. O *Avro* é particularmente útil quando usado em conjunto com dados tabulares, pois permite representar esses dados de maneira estruturada e compacta.

O *Kafka* também se integra ao *Schema Registry*, um serviço que permite que os esquemas *Avro* sejam armazenados de forma centralizada e recuperados. Isso garante que os

dados sejam sempre lidos e escritos de maneira consistente, independentemente de onde estejam sendo processados.

Aqui está um exemplo de uma configuração de *Kafka Sink Connector* para o *PostgreSQL*:

```
{
    "name": "postgres-sink",
    "config": {
        "connector.class": "io.confluent.connect.jdbc.
           JdbcSinkConnector",
        "tasks.max": "1",
        "topics": "my_topic",
        "key.converter": "org.apache.kafka.connect.storage.
           StringConverter",
        "value.converter": "io.confluent.connect.avro.
           AvroConverter",
        "value.converter.schema.registry.url": "http://
           localhost:8081",
        "connection.url": "jdbc:postgresql://localhost
           :5432/my_database",
        "connection.user": "my_user",
        "connection.password": "my_password",
        "auto.create": "true"
    }
}
```

Esta configuração define um *sink connector* que lê do tópico `my_topic` no *Kafka* e grava os dados no banco de dados *PostgreSQL* `my_database` usando a conexão Java Database Connectivity (JDBC) fornecida. O conector usa o `AvroConverter` para converter os dados do formato *Avro* e também se conecta ao *Schema Registry* para garantir a consistência dos esquemas.

Por fim, é válido ressaltar que para o desenvolvimento desse trabalho, o *Kafka* teve seu *deploy* baseado em um *docker compose* e orquestrado pelo ECS (AMAZON..., 2023).

5.4 PostgreSQL

Finalmente, volto a falar do *PostgreSQL*, que será o banco de dados escolhido para armazenar os dados que serão consumidos pelos microsserviços.

O *PostgreSQL* é um sistema de gerenciamento de banco de dados relacional de código aberto robusto, com mais de 30 anos de desenvolvimento ativo que o tornou uma solução confiável para muitas organizações.

Optar pelo *PostgreSQL* apresenta várias vantagens:

1. **Compatibilidade SQL**: Uma consideração importante na escolha do *PostgreSQL* foi sua linguagem SQL compatível com várias *engines* de consulta, incluindo *Trino*, *AWS Athena* e *Spark SQL*. Isso facilita a interação entre esses serviços e o *PostgreSQL*, proporcionando um ambiente de dados unificado e flexível.
2. **Desempenho**: O *PostgreSQL* oferece alto desempenho através de uma variedade de técnicas de otimização, incluindo indexação avançada e otimização de consultas.
3. **Confiabilidade**: O *PostgreSQL* é conhecido por sua confiabilidade e integridade de dados. Ele inclui uma série de características para garantir a segurança dos dados, incluindo transações atômicas, suporte para replicação síncrona e assíncrona e uma variedade de mecanismos de *backup*.
4. **Recursos Avançados**: Além disso, o *PostgreSQL* oferece uma série de recursos avançados, incluindo suporte para JSON, *arrays* e outros tipos de dados não convencionais, bem como extensões para funcionalidades geoespaciais, *full-text search*, entre outras.
5. **Suporte à Geolocalização**: O suporte à geolocalização é um recurso importante em vários contextos, por exemplo, em *Supply Chain*, onde a localização geográfica dos ativos é crucial. O *PostgreSQL* apresenta um módulo chamado *PostGIS*, que permite o armazenamento e a manipulação eficiente de dados geolocalizados. Isso torna o *PostgreSQL* uma escolha ainda mais adequada para cenários que necessitam do tratamento de dados de localização.

Por esses motivos, o *PostgreSQL* se estabelece como uma escolha sólida para o armazenamento de dados em nossa arquitetura de microsserviços.

5.5 Integração Spark + Kafka

A sinergia entre *Apache Spark* e *Apache Kafka* é fundamental para a eficiência na ingestão de dados em paralelo. Conforme delineado na documentação *Spark-Kafka* (STRUCTURED..., 2023a), um modelo de paralelismo 1:1 é adotado, onde cada partição no *Kafka* é correspondida por uma partição no *Spark*.

A correlação 1:1 entre as partições facilita um processamento eficiente dos dados. No entanto, esta estrutura impõe limitações na flexibilidade do escalonamento, uma vez que a capacidade de escalonamento é estritamente definida pelo número de partições no *Kafka*. Se o *Kafka* estiver sub-particionado, isso poderá resultar em recursos de processamento subutilizados no *Spark*.

Para contornar esta limitação, foi adotada uma estratégia alternativa, consumir dados a partir de arquivos *parquet*, em vez de diretamente do *Kafka*. O *Spark* é notavelmente eficiente no manejo de arquivos *parquet*, possibilitando a leitura paralela de múltiplas partições de um único arquivo. Este método oferece uma flexibilidade superior no controle do paralelismo da ingestão e otimização do desempenho.

Por exemplo, em um conjunto de dados de 1 TB segmentado em 1000 partições *parquet*, cada partição contendo cerca de 1 GB, é possível ajustar o número de partições lidas para se alinhar com a especificidade dos filtros de consulta. Isso minimiza a quantidade de dados lidos, economizando tempo e recursos.

Apesar das vantagens de eficiência e flexibilidade oferecidas pela ingestão de arquivos *parquet*, esta abordagem não está isenta de desafios, os quais serão explorados em detalhes na próxima seção.

5.6 Integração Alluxio + Spark

Este último tópico aborda a integração do *Alluxio* com o *Apache Spark* e a consequente redução expressiva de custos para projetos de *streaming* em tempo real empregando o *Apache Spark*.

No *Spark Structured Streaming*, a replicação contínua de dados do *Debezium*, presentes na camada *raw*, é efetuada por meio da API *Read Stream*, utilizando a opção de *File Source* (STRUCTURED..., 2023b). Este recurso permite que o *Spark* leia dados diretamente de arquivos no formato especificado (como *Parquet*, *CSV*, JSON) em um fluxo contínuo de dados.

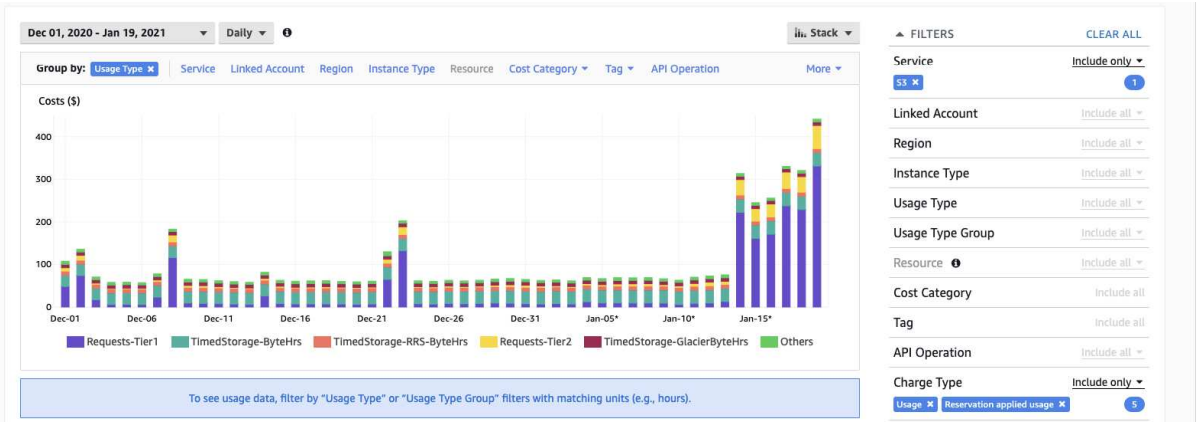

Figura 6 – Custo do S3 sem Alluxio. Fonte: Elaborado pelo autor.

A cada minuto, o *Spark* invoca a API *ListBucket* (LIST..., 2023) do S3 para cada um dos 50 *buckets específicos* (representando as 50 tabelas do banco de dados do cliente).

Além disso, o *Spark Structured Streaming* recorre ao mecanismo de *checkpoint* (ARMBRUST *et al.*, 2018) para registrar o estado atual do sistema de leitura. Essa característica é vital para manter a integridade dos dados ao prevenir a duplicação da leitura de um mesmo arquivo. Entretanto, esse *checkpoint*, armazenado no S3 e verificado sempre que um novo arquivo é detectado, acarreta um custo adicional. Tal comportamento levou a um aumento considerável no custo diário do S3, de $60/dia para cerca de $460/dia.

Como solução, o *Alluxio* foi incorporado à arquitetura em substituição ao S3. Este sistema de arquivos virtual permite ao *Spark* interagir com os dados como se estivessem em um único sistema unificado. Semelhante ao *Spark*, o *Alluxio* verifica a presença de novos arquivos ao invocar a API *HEAD* do S3 (S3, 2023), que consulta os metadados do Bucket. Consequentemente, o *Alluxio* é atualizado sempre que um novo arquivo é inserido no S3.

A otimização de custos é viabilizada pela diferença nos preços das chamadas à API do S3. Enquanto a chamada à API *ListBucketS3* custa 0.005$/1000 requisições, a chamada à API *HeadS3* custa apenas 0.0004$/1000 requisições. Para um único banco de dados, supondo que as requisições do *Spark* e do *Alluxio* ocorram a cada minuto, os custos diários estão descritos na tabela 2.

Tabela 2 – Tabela comparativa de custo de chamada ao S3

Custo List/1 cliente	Custo Head/ 1 cliente
0,36$	0,0288$

Esses números evidenciam as significativas economias proporcionadas pelo Alluxio. Além disso, vale lembrar que esta comparação considera apenas as chamadas à API de *ListBucket*.

No entanto, o *Spark* faz várias outras requisições de leitura durante o processamento. O *Alluxio*, com sua funcionalidade de *caching*, reduz essas múltiplas chamadas para apenas uma por arquivo, gerando economias adicionais.

É relevante abordar alternativas para mitigar os custos decorrentes das frequentes operações de *list bucket* realizadas pelos fluxos em tempo real no *Spark* para além do *Alluxio*. Uma opção notável é o *Autoloader* (AUTO. . . , 2023), uma funcionalidade exclusiva da *Databricks*. Esta plataforma integrada para análises de dados e *machine learning* otimiza o carregamento de grandes conjuntos de dados no *Spark*, automatizando o rastreamento e a ingestão de novos dados à medida que são gerados.

Contudo, a aplicação do *Autoloader* é limitada em nosso contexto, uma vez que nosso projeto se compromete a utilizar estritamente tecnologias de código aberto, e o *Autoloader* é uma solução proprietária. Em cenários onde essa restrição não é aplicável, especialmente quando a arquitetura está hospedada na nuvem *Databricks*, o *Autoloader* emerge como uma alternativa competente ao *Alluxio*, oferecendo uma solução eficaz para os desafios econômicos ligados ao manuseio de extensos volumes de dados.

6 ESTUDO DE CASO

Neste capítulo, discute-se o estudo de caso desenvolvido, conforme a Figura 4, com a finalidade de avaliar a viabilidade da arquitetura proposta. Inicialmente, uma visão detalhada da arquitetura monolítica utilizada antes da migração para microsserviços é apresentada. Em seguida, discute-se o processo de migração. Por fim, os resultados obtidos após a migração para a nova arquitetura são apresentados.

6.1 Descrição do Problema

A crítica necessidade de replicar dados dos bancos de dados Oracle de produção para ambientes analíticos, especialmente aqueles com alta demanda de consultas simultâneas, foi abordada no capítulo 1. A abordagem anterior empregava o Pentaho Data Integration (PDI) (Hitachi Vantara, 2023) e consistia em um processo de ETL em lotes. Contudo, essa metodologia enfrentava vários desafios:

- **Custos Elevados:** A infraestrutura e os recursos necessários para manter o processo de ETL em lotes implicavam em custos operacionais significativos.
- **Interrupções Operacionais:** A realização de consultas analíticas e processos de ETL diretamente no banco de produção frequentemente levava a interrupções e afetava negativamente a performance do sistema.
- **Lentidão e Falta de Paralelismo:** O modelo síncrono do processo resultava em atrasos consideráveis, pois os dados de um cliente eram processados somente após a conclusão do processamento do cliente anterior.
- **Desafios de Desenvolvimento e Depuração:** A complexidade do sistema e a falta de ferramentas adequadas tornavam o desenvolvimento e a depuração de processos um desafio.
- **Problemas com Logs e Orquestração:** A gestão de *logs* e a orquestração de processos ETL eram complicadas, o que dificultava a monitorização e a resolução de problemas.

Esses desafios sublinhavam a necessidade de uma solução mais eficiente, capaz de lidar com as demandas de um ambiente de produção e análise de dados em constante evolução.

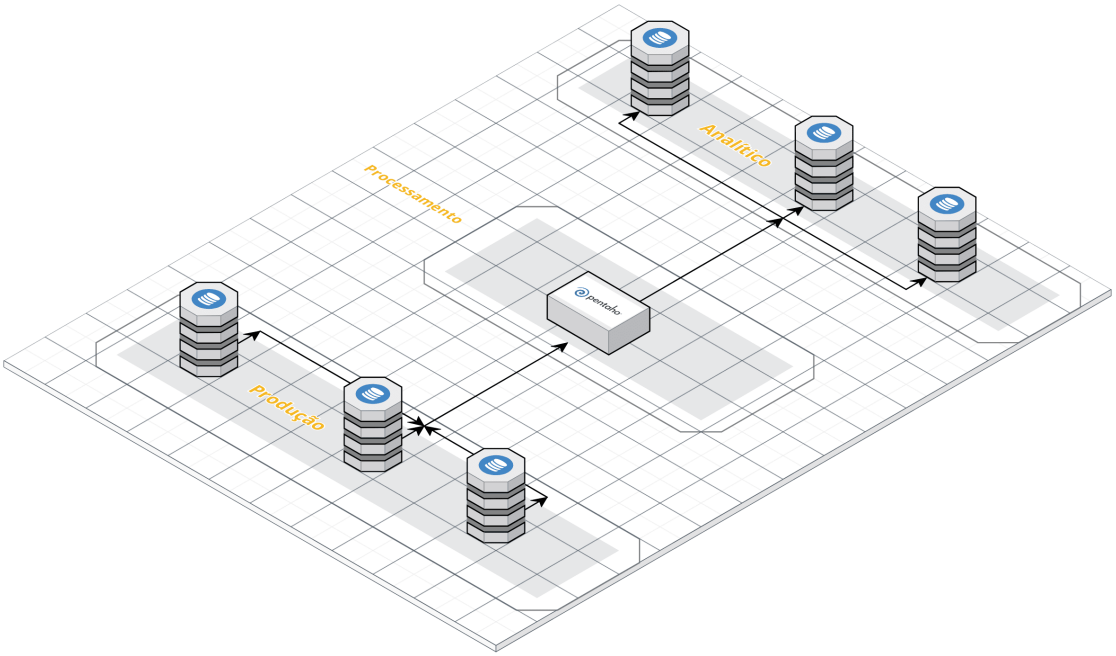

Figura 7 – Arquitetura *Batch* enfatizando a divisão em camadas do trabalho de replicação de dados. Fonte: Elaborado pelo Autor.

6.2 Arquitetura Anterior

A arquitetura monolítica que era utilizada anteriormente é ilustrada na Figura 7. Essa arquitetura se baseava no uso do PDI para a replicação dos dados de produção, transferindo-os para a camada analítica. O processo de replicação era realizado em lotes, uma característica central dessa arquitetura.

Nas próximas seções, cada componente da arquitetura monolítica é descrito em detalhes, com ênfase em suas funções específicas e na maneira como estes se integravam ao fluxo de dados geral. Esta análise é fundamental para entender as limitações inerentes a essa abordagem monolítica e para destacar as melhorias implementadas na nova arquitetura proposta.

6.2.1 Bancos de Dados de Produção

Esta camada era formada pelos bancos de dados relacionais *AWS RDS Oracle*, versão 12C (ORACLE, 2023), que eram essenciais nas operações de produção, armazenando dados operacionais críticos para o sistema de gerenciamento da cadeia de suprimentos de mais de 50 clientes.

Os dados armazenados nesses bancos eram vitais para o eficiente monitoramento e gerenciamento da cadeia de suprimentos. Eles incluíam informações detalhadas de telemetria,

rotas e paradas, que eram fundamentais para a gestão proativa e eficiente das operações logísticas. A telemetria fornecia *insights* valiosos sobre o desempenho e a localização dos veículos, permitindo um acompanhamento preciso e em tempo real. As informações sobre rotas e paradas eram cruciais para o planejamento e execução eficazes das atividades de transporte, contribuindo para a otimização dos itinerários e a redução dos tempos de entrega.

Além disso, a análise desses dados era essencial para a tomada de decisões estratégicas, oferecendo uma base sólida para análises operacionais em tempo real. Isso possibilitava uma otimização contínua e dinâmica das operações de *supply chain*, resultando em processos logísticos mais eficientes e adaptáveis às necessidades variáveis dos clientes.

6.2.2 Camada de Processamento - ETL

A camada de processamento é o núcleo da replicação e transformação dos dados. Realizada em uma única instância *EC2 M4.2xlarge* (EC2, 2023) da AWS, o processamento ocorria a cada hora, utilizando a ferramenta PDI, uma solução *open-source* para integração de dados que oferece recursos de ETL. Com sua interface visual intuitiva, o PDI simplifica a manipulação de dados, reduzindo a necessidade de codificação complexa.

A Figura 8 demonstra a construção de um processo ETL no PDI, destacando sua capacidade de executar leituras e escritas de tabelas, validações, mapeamentos, verificações de qualidade de dados, entre outras funcionalidades. Estas características, aliadas à facilidade de implementação, contribuíram para a popularidade da ferramenta no mercado.

No nosso contexto, o PDI executava uma análise completa das tabelas no banco de produção, sincronizando-as com o banco analítico e inserindo novos registros. Para tabelas de importância crítica, o processo incluía a replicação de deleções e atualizações, além de transformações de dados, aumentando a complexidade e o tempo de processamento.

A orquestração dos processos do PDI para todos os clientes era realizada através de um único arquivo *Cron* na instância. Além disso, não havia processamento em paralelo entre clientes; o processamento de um tinha que ser concluído antes de iniciar o de outro, caracterizando um método altamente síncrono e menos eficiente.

Também é válido salientar que os *logs* eram concentrados em um único arquivo e para depurar esses *logs* era-se necessário acessar diretamente a instancia para checar o arquivo, visto que não havia conexão diretamente com o sistema de *logs* da empresa.

Embora o PDI seja uma ferramenta *low-code*, com uma interface de arrastar e soltar

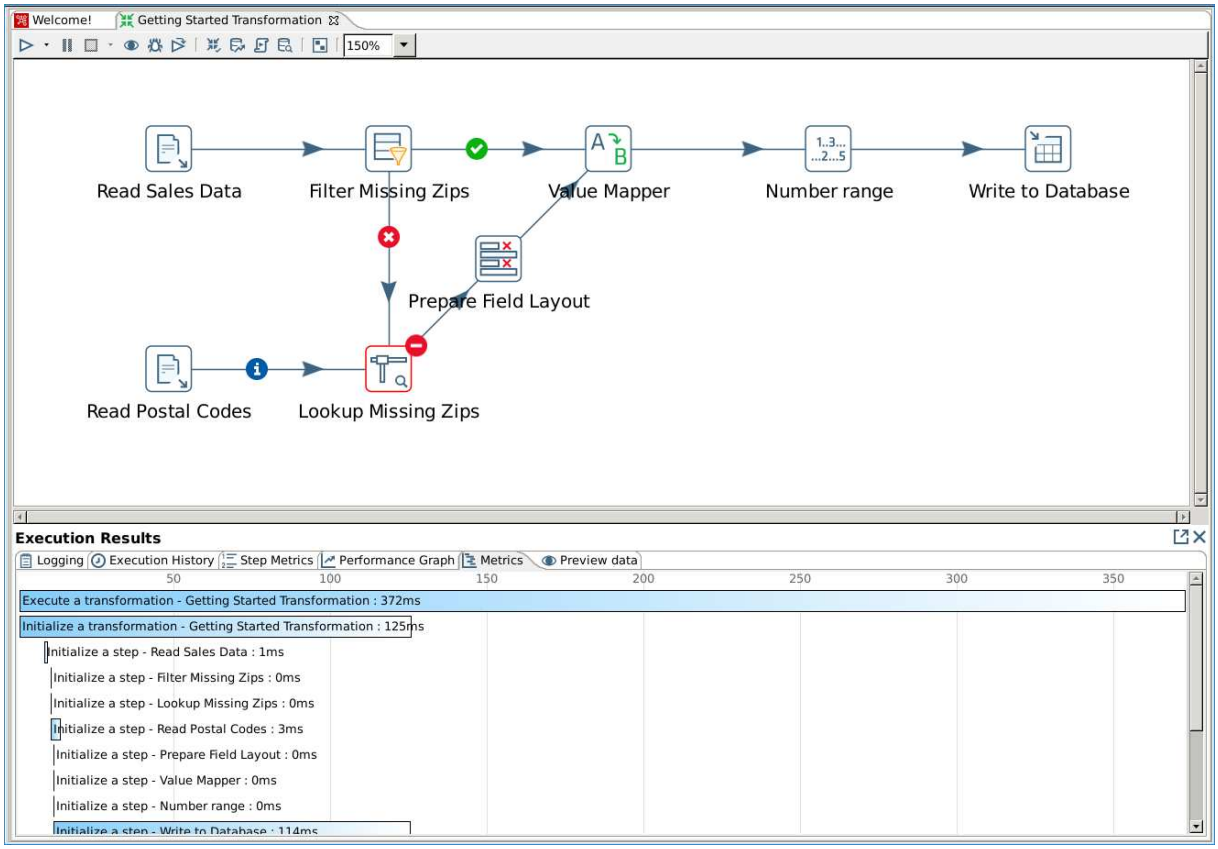

Figura 8 – Exemplo de um ETL feito com PDI. Fonte: Elaborado pelo Autor

que facilita o desenvolvimento, existiam preocupações sobre sua capacidade de se adaptar a necessidades futuras mais complexas ou personalizadas, dada a potencial inflexibilidade da ferramenta diante de demandas evolutivas do projeto.

6.2.3 Camada Analítica

Esta camada abriga os bancos de dados analíticos *Postgres*, hospedados no *Aurora RDS* da AWS. O custo de manutenção destes bancos é influenciado pelos Input/Output Operations Per Second (IOPS), conforme detalhado em (SERVICES, 2023) e (VERBITSKI *et al.*, 2017). O Aurora, otimizado para aumentar a eficiência do Input/Output (I/O), utiliza um mecanismo conhecido como *"I/O de banco de dados em burst"*, o qual permite ultrapassar os limites normais de I/O em situações de demanda elevada (VERBITSKI *et al.*, 2017).

Os bancos de dados nesta camada estão interligados a *dashboards* de alta concorrência de *queries*, utilizados para análises operacionais tanto pelos clientes quanto pela equipe de *customer success* da empresa. Esta estrutura facilita o monitoramento da qualidade das operações de cada cliente, permitindo a extração de valor agregado dos dados produzidos.

Para as tabelas com grande volume de dados e atualizações frequentes, o sistema

analítico era configurado para estabelecer conexão direta com o banco de dados de produção. Essa abordagem, apesar de reduzir a latência no acesso às informações, levava a interrupções significativas no desempenho do sistema devido à complexidade das consultas analíticas.

6.2.4 Vantagens e Desvantagens da Arquitetura Monolítica

A arquitetura monolítica, apesar de suas limitações, oferecia vantagens em determinados contextos:

- **Simplicidade na Gerência de Replicação:** A capacidade de gerenciar a replicação de múltiplos bancos de dados através de um único processo era uma vantagem notável. Isso simplificava a administração, pois apenas um único *cron job* era necessário para o agendamento das tarefas de replicação.
- **Intuitividade das Ferramentas *Low-Code*:** A ferramenta PDI, por ser *low-code*, facilitava a compreensão dos processos de transformação e a lógica de negócios envolvida nos ETL. Isso tornava o desenvolvimento e a manutenção mais acessíveis, mesmo para usuários com menos experiência em codificação. Conforme Martinez e Pfister (MARTINEZ; PFISTER, 2023) discutem, as ferramentas *low-code* permitem que indivíduos mais próximos das operações desenvolvam abordagens digitais para enfrentar desafios operacionais, potencializando a digitalização e inovação em vários níveis de uma indústria.
- **Consistência Operacional:** A uniformidade dos processos de ETL, alinhados em uma única plataforma, assegurava uma consistência operacional, facilitando a padronização para casos de uso comuns.
- **Baixa Barreira de Entrada para Correções Pontuais:** A arquitetura monolítica, juntamente com o uso de ferramentas *low-code*, facilitava a integração de novos desenvolvedores, particularmente aqueles com experiência limitada ou sem conhecimento aprofundado do negócio. Essa abordagem permitia a realização intuitiva e ágil de ajustes menores e correções pontuais. Contudo, é importante ressaltar que, apesar das ferramentas *low-code* empoderarem os usuários próximos às operações para desenvolverem suas próprias abordagens digitais para desafios operacionais, como destacado por (MARTINEZ; PFISTER, 2023), elas podem enfrentar limitações ao lidar com demandas mais complexas e personalizadas devido à possível inflexibilidade em projetos evolutivos.

Essas vantagens, no entanto, precisavam ser equilibradas com as limitações inerentes à arquitetura monolítica, especialmente no que diz respeito à escalabilidade e flexibilidade.

Já como desvantagens da arquitetura monolítica, podemos enumerar:

- **Complexidade em Transformações Personalizadas:** Embora a gerência centralizada da replicação fosse vantajosa, ela se tornava um desafio quando necessárias transformações específicas para clientes individuais. Isso aumentava a complexidade e o número de *pipelines*, complicando o agendamento e a gestão.
- **Limitação na Flexibilidade de Replicação:** A arquitetura impunha uma sequência fixa de replicação, restringindo a flexibilidade. Acelerar um processo exigia ou o aumento dos recursos da máquina (escalabilidade vertical) ou a criação de um processo de replicação dedicado.
- **Ineficiência Temporal:** Mesmo com volumes moderados de dados, o tempo de processamento era excessivo. Por exemplo, sincronizar 50.000 registros de tabelas críticas podia levar uma hora, tempo este que se ampliava em períodos de alta demanda.
- **Impacto nos Custos e Performance devido ao IOPS:** O processo de ETL executado pelo PDI elevava os custos com IOPS e afetava negativamente a performance do banco, especialmente durante picos de I/O, apesar das otimizações do Aurora.
- **Tratamento Limitado de Exclusões e Atualizações:** Inicialmente, exclusões e atualizações eram tratadas apenas para tabelas consideradas críticas. Com o aumento na demanda e no número de tabelas críticas, prolongava-se o tempo de processamento, afetando a eficiência para todos os clientes.
- **Gestão de Logs Ineficiente:** O sistema de logs, centralizado em um único arquivo, complicava a verificação das operações. Era frequente a necessidade de apagar e recriar esse arquivo devido ao esgotamento do espaço de armazenamento disponível na máquina.

6.2.5 Metodologia e Estratégia de Migração

A migração foi planejada com um enfoque na seleção criteriosa dos clientes, visando minimizar interrupções e garantir a integridade dos dados. A estratégia, detalhada no infográfico apresentado na Figura 9, envolveu a avaliação das necessidades atuais e futuras dos clientes, o uso das ferramentas analíticas e a criticidade das operações, optando-se por uma migração incremental.

Inicialmente, selecionou-se três clientes que utilizavam o produto analítico, mas não de maneira crítica, podendo assim tolerar possíveis interrupções no serviço. Esta seleção também levou em conta a volumetria das operações, priorizando clientes com menos de 100 *gigabytes*

Figura 9 – Infográfico utilizado na apresentação do projeto de migração ao cliente. Fonte: Elaborado pelo autor.

no total de suas tabelas, classificado internamente como um volume pequeno. O objetivo dessa primeira fase era verificar a consistência do processo de migração, a adequação das operações e a coleta de métricas, conforme ilustrado na primeira etapa do infográfico.

Após a migração bem-sucedida desses clientes iniciais, que ocorreu sem intercorrências, procedeu-se à validação do processo. Esta etapa, evidenciada no infográfico, envolveu a validação dos *logs* de operação e a comparação dos dados dos *dashboards* analíticos com os dados de produção. Com a qualidade do processo e dos dados confirmada, elaborou-se uma nova lista de clientes prioritários baseada na urgência de replicação de dados em tempo real na plataforma analítica, como mostrado na subsequente fase do infográfico.

A migração desses clientes subsequentes ocorreu de forma incremental ao longo de dois meses, com o número de clientes migrados variando conforme as demandas internas da empresa e alternando períodos de maior e menor intensidade de migração. Nesse mapeamento, as equipes de *analytics* e *customer success* trabalharam conjuntamente, um aspecto destacado na fase final do infográfico.

Por fim, é válido salientar que o procedimento técnico para a migração de cada cliente seguiu a seguinte metodologia:

1. Desativação do processo de replicação do cliente orquestrado pelo PDI).
2. Replicação do banco de dados utilizando a nova arquitetura.
3. Validação do encaminhamento dos *logs* do processo para a plataforma de *logs* interna.
4. Verificação dos *dashboards* analíticos pelo time de *customer success* e pelo próprio cliente.

6.2.6 Sucessos e Lições Aprendidas

A migração foi um processo enriquecedor, trazendo várias lições importantes que contribuíram para a melhoria contínua do projeto. Os principais sucessos e aprendizados incluem:

- **Preparação e Planejamento Estratégico:** A colaboração alinhada entre os times técnico (analytics) e de negócios (customer success) provou ser crucial para o sucesso da migração. A definição conjunta de estratégias e objetivos assegurou que todas as partes estivessem sincronizadas, garantindo um processo migratório eficiente e transparente.
- **Testes Extensivos:** A condução de testes abrangentes em cada segmento da nova arquitetura e do processo de migração como um todo foi vital. Isso não apenas assegurou a confiabilidade e eficácia da nova arquitetura, mas também contribuiu para a otimização do processo migratório.
- **Comunicação Eficaz com Clientes:** Estabelecer uma comunicação clara e eficaz com os clientes foi fundamental. Informá-los sobre a necessidade da migração, os procedimentos envolvidos, as janelas de manutenção e os benefícios esperados ajudou a criar uma atmosfera de confiança e colaboração. Esse diálogo transparente foi essencial para garantir o apoio e a compreensão dos clientes durante todo o processo.

6.2.7 Desafios e Áreas de Melhoria

A migração, embora tenha alcançado seu objetivo, enfrentou desafios notáveis, revelando pontos de melhoria:

- **Gerenciamento de Logs:** Um dos desafios mais críticos foi o gerenciamento eficaz dos *logs*, particularmente desafiador quando múltiplos clientes eram migrados ao mesmo tempo. A complexidade na leitura e interpretação dos *logs* de vários microsserviços tornou-se um ponto de atenção. Para abordar essa questão, desenvolvemos e implementamos filtros aprimorados na nossa aplicação interna de *logs*, o que facilitou significativamente a

identificação e a análise dos registros.

- **Gerenciamento de Configuração e Pipeline:** Outro desafio enfrentado foi na gestão de configuração dos processos de migração. A diversidade dos clientes, especialmente em termos de volume de dados, tornou as ferramentas internas de gerenciamento de configuração um gargalo. Gerenciar a mudança de recursos de um cliente para outro durante o processo não era intuitivo devido a como a ferramenta corporativa foi desenvolvida. Isso ressaltou a necessidade de uma estratégia mais flexível e robusta. Por conseguinte, a adoção de ferramentas de infraestrutura como código, como o Terraform (**??**), foi valorizada como uma solução potencial para melhorar a eficiência e escalabilidade nos processos de migração, adaptando-se às diferentes exigências dos clientes.

Esses desafios, embora tenham apresentado obstáculos significativos, ofereceram oportunidades valiosas para aprimorar continuamente a infraestrutura e as práticas de migração.

6.2.8 Resultados da Migração para a Nova Arquitetura

A transição para a arquitetura proposta resultou em diversas melhorias, dentre elas destacam-se:

- **Replicação de Dados em Tempo Real:** A atualização possibilitou a replicação de dados em tempo real para todos os clientes, melhorando a eficiência e a resposta operacional. Assim, clientes que tinham *delay* de até 2 horas em seus dados, dependendo da fila do PDI e do tráfego, agora têm seus dados replicados em até 15 segundos.
- **Cobertura Completa das Operações de Banco de Dados para todas as tabelas:** Todas as operações de inserção, exclusão e atualização são agora replicadas para todas as tabelas. Na arquitetura anterior, isso resultava em maior duração do processo e, devido a ser síncrono, o aumento em um cliente fazia aumentar o tempo de replicação de todos os outros, garantindo que os dados analíticos sejam um reflexo fiel dos dados de produção.
- **Desativação do Serviço de Análise nos Bancos de Produção:** A capacidade de desativar completamente o serviço de análise dos bancos de produção resultou em aumento de performance e redução de custos operacionais e de riscos de interrupção em produção.
- **Implementação de um *Data Lake*:** A introdução de um *data lake* baseado em *Delta Tables* facilitou a manipulação dos dados, agilizando e aprofundando as análises.
- **Redução do Tempo de Replicação completa das tabelas:** O tempo necessário para a replicação completa das tabelas de um cliente foi drasticamente reduzido. Na arquitetura

anterior, o processo que poderia durar, em média, 6 dias, diminuiu para aproximadamente 5 horas.

- **Latência Máxima na Replicação de Dados:** A latência na replicação de novos dados foi minimizada, mantendo o sistema ágil mesmo durante períodos de alta demanda ou de reprocessamentos. Assim, o tempo que poderia ser até mais de 2 horas foi para no máximo 1 minuto.
- **Redução Significativa no Armazenamento com Formato Avro:** A adoção do formato *Avro* para armazenamento de registros no *Kafka* resultou em uma redução substancial do espaço de armazenamento necessário. Com 56 milhões de registros, o espaço ocupado foi de apenas 15.6 *gigabytes*. Uma análise comparativa revelou que, se os mesmos dados fossem armazenados em formato JSON, o armazenamento necessário seria aproximadamente 39 *gigabytes*. Portanto, a escolha do formato *Avro* proporcionou uma taxa de compressão de cerca de 2.5 vezes, demonstrando uma economia significativa de armazenamento.
- **Confiabilidade Aprimorada do Produto de Análise:** A confiabilidade incrementada transformou o produto de análise de um extra em um recurso central, aumentando seu valor e integração ao portfólio de produtos da empresa. Assim, um produto que antes era somente uma cortesia para os clientes virou uma fonte de receita para a companhia.

7 CONCLUSÕES E TRABALHOS FUTUROS

A arquitetura proposta foi validada em um ambiente de alta criticidade e intensidade de fluxo de dados. No entanto, devido ao avanço contínuo das tecnologias de engenharia de dados, existe espaço para futuras otimizações.

Conforme discutido na seção de arquitetura, *Debezium* e *Apache Kafka* foram implantados apenas em *containers Docker*, orquestrados no ECS, enquanto *Apache Spark* e *Alluxio* foram implantados no EMR. A transição para uma orquestração baseada em *Kubernetes* (BURNS *et al.*, 2016) ofereceria inúmeras vantagens, incluindo economia de custos, melhor orquestração e escalabilidade. Ademais, com o *Apache Spark* e *Alluxio* sendo implantados no *Kubernetes*, poderíamos acessar os dados ainda mais rapidamente devido ao recurso de *'short circuit'* (FOUNDATION, 2023). Esta funcionalidade permite que ambas as aplicações compartilhem seus volumes, acelerando assim o acesso aos dados.

Outra melhoria potencial envolveria a simplificação da arquitetura, removendo o *Apache Spark* e *Alluxio* e substituindo-os por ferramentas que executam tarefas semelhantes com menor demanda de infraestrutura. Por exemplo, ferramentas baseadas em *Rust*, com interface *Python*, como o *ByteWax* (BYTEWAX, 2023), poderiam oferecer a alta performance do *Rust* sem comprometer a facilidade de uso proporcionada pelo *Python*.

Para além disso, a substituição do *Delta Lake* por outros formatos de dados, como o *Apache Iceberg*, também deveria ser considerada. Como o *Iceberg* não requer um *cluster Spark* para interagir com suas tabelas, ao contrário do *Delta Lake*, esta substituição poderia diminuir nossa dependência do *Spark*. Esta mudança poderia resultar em uma possível redução dos custos de *cloud* da arquitetura, como sugerido em (BELOV; NIKULCHEV, 2021).

Por fim, ponderar a substituição do *Postgres* na última camada arquitetural por soluções mais adequadas ao contexto Online analytical processing (OLAP), como o *Apache Doris*, *Apache Pinot* ou *Clickhouse*, é fundamental. Estas alternativas não apenas potencializariam a eficiência das *queries*, mas também robusteceriam a execução de múltiplas *queries* simultâneas por segundo, possibilitando, por exemplo, a operação concomitante de diversos *dashboards*. (DORIS, 2023; PINOT, 2023; CLICKHOUSE, 2023)

Com base na análise do artigo "Uma Proposta de uma Arquitetura de Código Aberto para a Aplicação do Strangler Pattern com Change data capture para Setores com Alto Volume de Dados", conclui-se que a arquitetura proposta é uma solução robusta para a modernização de sistemas legados em setores com alto volume de dados. A utilização de tecnologias de código

aberto, como *Apache Spark*, *Debezium* e *Apache Kafka*, permite o gerenciamento eficiente de fluxos de dados em tempo real.

A implementação do *Strangler Pattern*, que permite a substituição gradual de partes do sistema por novas funcionalidades e serviços, demonstrou ser uma estratégia eficaz para a modernização de sistemas legados. A arquitetura também se beneficia do uso do CDC, uma estratégia de integração que carrega apenas registros novos, atualizados e excluídos no banco banco de dados legal, no *data lake* e no banco de dados do novo microsserviço, otimizando tempo e recursos.

Apesar da eficácia da arquitetura atual, há espaço para otimizações futuras. A evolução constante das tecnologias de engenharia de dados abre novas possibilidades para melhorar a eficiência do processamento de dados e a escalabilidade da arquitetura. Entre as melhorias sugeridas estão a transição para uma orquestração baseada em *Kubernetes*, a substituição do *Apache Spark* e do *Alluxio* por ferramentas mais eficientes, e a troca do *Delta Lake* por outra API de tabela, como o *Apache Iceberg*.

Os trabalhos futuros visam não apenas manter ou aumentar a performance da arquitetura, mas também facilitar a sua manutenção e reduzir a curva de aprendizado e implantação. Isso é crucial para garantir a adaptabilidade da arquitetura a diferentes setores.

Por fim, é importante destacar que, apesar das possibilidades de otimização, a arquitetura já demonstrou sucesso em sua implementação atual. A eficácia e robustez da arquitetura são validadas pelos resultados obtidos até agora e pelo tempo de funcionamento de mais de dois anos sem interrupções e com pequenas manutenções pontuais. No entanto, a pesquisa contínua e a implementação de novas tecnologias são essenciais para garantir que a arquitetura continue a evoluir e a atender às demandas crescentes de processamento de dados em tempo real.

REFERÊNCIAS

ALLUXIO Caching Namespace docs. 2023. Last accessed 20 Aug 2023. Disponível em: https://docs.alluxio.io/os/user/stable/en/core-services/Caching.html.

ALLUXIO Unified Namespace docs. 2023. Last accessed 20 aug 2023. Disponível em: https://docs.alluxio.io/os/user/stable/en/core-services/Unified-Namespace.html.

AMAZON Elastic Container Service. 2023. Disponível em: https://aws.amazon.com/ecs/.

Amazon Web Services, Inc. **Amazon Simple Storage Service (S3) Documentation**. [*S. l.*]: Amazon Web Services, Inc., 2023. https://aws.amazon.com/s3/. Último acesso em 2023.

ARMBRUST, M.; DAS, T.; SUN, L.; YAVUZ, B.; ZHU, S.; MURTHY, M.; TORRES, J.; HOVELL, H. van; IONESCU, A.; ŁUSZCZAK, A. *et al.* Delta lake: high-performance acid table storage over cloud object stores. **Proceedings of the VLDB Endowment**, VLDB Endowment, v. 13, n. 12, p. 3411–3424, 2020.

ARMBRUST, M.; DAS, T.; TORRES, J.; YAVUZ, B.; ZHU, S.; XIN, R.; GHODSI, A.; STOICA, I.; ZAHARIA, M. Structured streaming: A declarative api for real-time applications in apache spark. In: **Proceedings of the 2018 International Conference on Management of Data**. [*S. l.*: *s. n.*], 2018. p. 601–613.

AUTO Loader. Databricks, 2023. Disponível em: https://docs.databricks.com/ingestion/auto-loader/index.html.

BELOV, V.; NIKULCHEV, E. Analysis of big data storage tools for data lakes based on apache hadoop platform. **International Journal of Advanced Computer Science and Applications**, Science and Information (SAI) Organization Limited, v. 12, n. 8, 2021.

BURNS, B.; GRANT, B.; OPPENHEIMER, D.; BREWER, E.; WILKES, J. Borg, omega, and kubernetes. **Communications of the ACM**, ACM New York, NY, USA, v. 59, n. 5, p. 50–57, 2016.

BYTEWAX. bytewax, 2023. Disponível em: https://bytewax.io/.

CHANDRA, H. Analysis of change data capture method in heterogeneous data sources to support rtdw. In: IEEE. **2018 4th International Conference on Computer and Information Sciences (ICCOINS)**. [*S. l.*], 2018. p. 1–6.

CLICKHOUSE. **ClickHouse**. Yandex, 2023. ClickHouse. Disponível em: https://clickhouse.com/.

Confluent Inc. **Schema Registry Documentation**. [*S. l.*]: Confluent, 2023. https://docs.confluent.io/platform/current/schema-registry/index.html. Último acesso em agosto de 2023.

DEBEZIUM. **Debezium**. n.d. Disponível em: https://debezium.io/.

DEBEZIUM. **FAQ**. n.d. Disponível em: https://debezium.io/documentation/faq/#how_does_debezium_affect_source_databases.

DOCKER, I. **Documentação do Docker**. 2023. Último acesso em Agosto de 2023. Disponível em: https://docs.docker.com/.

Docker Inc. **Docker Compose Documentation**. 2023. https://docs.docker.com/compose/. Último acesso em agosto de 2023.

DORIS. **Apache Doris**. The Apache Software Foundation, 2023. Apache Doris. Disponível em: https://doris.apache.org/.

EC2. **Aws EC2 Homepage**. Amazon Web Services, Inc., 2023. Last accessed 20 Aug 2023. Disponível em: https://aws.amazon.com/pt/ec2/instance-types/.

EMR. **Amazon EMR**. Amazon Web Services, Inc., 2023. Disponível em: https://aws.amazon.com/emr/.

FOUNDATION, C. N. C. **Improving Data Locality for Analytics Jobs on Kubernetes Using Alluxio**. Cloud Native Computing Foundation, 2023. Disponível em: https://www.cncf.io/online-programs/improving-data-locality-for-analytics-jobs-on-kubernetes-using-alluxio/.

FOWLER, M. **StranglerFigApplication**. martinfowler.com, 2004. Disponível em: https://martinfowler.com/bliki/StranglerFigApplication.html.

FOWLER, M.; LEWIS, J. **Microservices: a definition of this new architectural term**. martinfowler.com, 2014. Disponível em: https://martinfowler.com/articles/microservices.html.

GARCÍA-GIL, D.; RAMÍREZ-GALLEGO, S.; GARCÍA, S.; HERRERA, F. A comparison on scalability for batch big data processing on apache spark and apache flink. **Big Data Analytics**, BioMed Central, v. 2, n. 1, p. 1–11, 2017.

Google Cloud. **Google Cloud Storage Documentation**. [*S. l.*]: Google Cloud, 2023. https://cloud.google.com/storage. Último acesso em 2023.

Hitachi Vantara. **Pentaho Data Integration 9.3 Documentation**. [*S. l.*]: Hitachi Vantara, 2023. https://help.hitachivantara.com/Documentation/Pentaho/9.3/Products/Pentaho_Data_Integration. Último acesso em 2023.

KREPS, J.; NARKHEDE, N.; RAO, J. *et al.* Kafka: A distributed messaging system for log processing. In: ATHENS, GREECE. **Proceedings of the NetDB**. [*S. l.*], 2011. v. 11, n. 2011, p. 1–7.

LAPLANTE, A. **Architecting data lakes**. [*S. l.*]: O'Reilly Media, 2016.

LI, H.; GHODSI, A.; ZAHARIA, M.; SHENKER, S.; STOICA, I. Tachyon: Reliable, memory speed storage for cluster computing frameworks. In: **Proceedings of the ACM Symposium on Cloud Computing**. [*S. l.*: *s. n.*], 2014.

LIST Bucket S3 docs. Amazon Web Services, Inc., 2023. Disponível em: https://docs.aws.amazon.com/AmazonS3/latest/API/API_ListBuckets.html.

MARTINEZ, E.; PFISTER, L. Benefits and limitations of using low-code development to support digitalization in the construction industry. **Automation in Construction**, Elsevier, v. 152, p. 104909, 2023.

NEWMAN, S. **Building Microservices: Designing Fine-Grained Systems**. [*S. l.*]: O'Reilly Media, 2015. ISBN 978-1-491-95035-7.

NEWMAN, S. **Monolith to microservices: evolutionary patterns to transform your monolith**. [*S. l.*]: O'Reilly Media, 2019.

NEWMAN, S. **Building microservices**. [*S. l.*]: "O'Reilly Media, Inc.", 2021.

ORACLE, R. **Amazon RDS for Oracle**. Amazon Web Services, Inc., 2023. Last accessed 20 Aug 2023. Disponível em: https://aws.amazon.com/pt/rds/oracle/.

PARSONS, R. **Effective Microservices: Strategies for Managing Complex Systems**. [*S. l.*]: ThoughtWorks, 2018.

PERERA, S.; PERERA, A.; HAKIMZADEH, K. Reproducible experiments for comparing apache flink and apache spark on public clouds. **arXiv preprint arXiv:1610.04493**, 2016.

PINOT. **Apache Pinot**. The Apache Software Foundation, 2023. Apache Pinot. Disponível em: https://pinot.apache.org/.

REORCHESTRATE. **Debezium Performance Impact**. n.d. Disponível em: https://reorchestrate.com/posts/debezium-performance-impact/.

RICHARDS, M. **Microservices vs. Service-Oriented Architecture**. [*S. l.*]: O'Reilly Media, 2015.

RICHARDSON, C. **Microservices Patterns: With Examples in Java**. [*S. l.*]: Manning Publications, 2018.

S3, H. **S3 Head Api docs**. Amazon Web Services, Inc., 2023. Disponível em: https://docs.aws.amazon.com/AmazonS3/latest/API/API_HeadObject.html.

SERVICES, I. A. W. **Preços do Amazon Aurora**. 2023. Último acesso em Agosto de 2023. Disponível em: https://aws.amazon.com/rds/aurora/pricing/.

SHI, J.; BAO, Y.; LENG, F.; YU, G. Study on log-based change data capture and handling mechanism in real-time data warehouse. In: IEEE. **2008 international conference on computer science and software engineering**. [*S. l.*], 2008. v. 4, p. 478–481.

SHVACHKO, K.; KUANG, H.; RADIA, S.; CHANSLER, R. The hadoop distributed file system. In: IEEE. **2010 IEEE 26th symposium on mass storage systems and technologies (MSST)**. [*S. l.*], 2010. p. 1–10.

SINGLE Message Transforms for Confluent Platform. Confluent, Inc., 2023. Last accessed 26 Aug 2023. Disponível em: https://docs.confluent.io/platform/current/connect/transforms/overview.html.

STRUCTURED Streaming + Kafka Integration Guide (Kafka broker version 0.10.0 or higher). Apache Spark, 2023. Disponível em: https://spark.apache.org/docs/latest/structured-streaming-kafka-integration.html.

STRUCTURED Streaming Programming Guide - Input Sources. Apache Spark, 2023. Disponível em: https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html#input-sources.

TANK, D. M.; GANATRA, A.; KOSTA, Y.; BHENSDADIA, C. Speeding etl processing in data warehouses using high-performance joins for changed data capture (cdc). In: IEEE. **2010 International Conference on Advances in Recent Technologies in Communication and Computing**. [*S. l.*], 2010. p. 365–368.

THE Essential Guide to Apache Flink for Stream Processing. 2023. Last accessed on [data de acesso]. Disponível em: https://risingwave.com/blog/the-essential-guide-to-apache-flink-for-stream-processing/#:~:text=Learning\%20Curve\%3A\%20Apache\%20Flink\%20has,new\%20users\%20to\%20get\%20started.

TRAN, T. T. Migrating from monolithic application to microservices. 2020.

VERBITSKI, A.; SIVASUBRAMANIAN, S.; HANDY, D.; SHUTE, J.; WHANG, K.; LERNER, A. Amazon aurora: Design considerations for high throughput cloud-native relational databases. **Proceedings of the 2017 ACM International Conference on Management of Data**, 2017.

VERBITSKIY, I.; THAMSEN, L.; KAO, O. When to use a distributed dataflow engine: evaluating the performance of apache flink. In: IEEE. **2016 Intl IEEE Conferences on Ubiquitous Intelligence & Computing, Advanced and Trusted Computing, Scalable Computing and Communications, Cloud and Big Data Computing, Internet of People, and Smart World Congress (UIC/ATC/ScalCom/CBDCom/IoP/SmartWorld)**. [*S. l.*], 2016. p. 698–705.

VILLACA, G. L. D. *et al.* Strategies to mitigate anti-patterns in microservices before migrating from a monolithic system to microservices. Universidade Estadual do Oeste do Paraná, 2022.

VOHRA, D.; VOHRA, D. Apache avro. **Practical Hadoop Ecosystem: A Definitive Guide to Hadoop-Related Frameworks and Tools**, Springer, p. 303–323, 2016.

WHAT advantages/disadvantages using Apache Flink? 2023. Last accessed on [data de acesso]. Disponível em: https://www.linkedin.com/advice/0/what-advantages-disadvantages-using-apache-flink-cshec.

WHAT is Apache Flink? 2023. Last accessed on [data de acesso]. Disponível em: https://nexocode.com/blog/posts/what-is-apache-flink/.

WHAT is Change Data Capture? Confluent, Inc., 2023. Disponível em: https://www.confluent.io/learn/change-data-capture/#:~:text=Change%20data%20capture%20(CDC)%20refers,all%20systems%20and%20deployment%20environments.

WOLFF, E. **Microservices: Security Challenges and Best Practices**. [*S. l.*]: InfoQ, 2016.

YANAGA, E. **Migrating to microservice databases: From relational monolith to distributed data**. [*S. l.*]: O'Reilly Media, 2017.

YAO, X.; LI, J.; TAO, Y.; JI, S. Relational database query optimization strategy based on industrial internet situation awareness system. In: **2022 7th International Conference on Computer and Communication Systems (ICCCS)**. [*S. l.*: *s. n.*], 2022. p. 152–155.

YODER, J. W.; MERSON, P. Strangler patterns. In: **Proceedings of the 27th Conference on Pattern Languages of Programs**. [*S. l.*: *s. n.*], 2020. p. 1–25.

ZAHARIA, M.; XIN, R.; WENDELL, P.; DAS, T.; ARMBRUST, M.; DAVE, A.; MENG, X.; ROSEN, J.; VENKATARAMAN, S.; FRANKLIN, M.; GHODSI, A.; GONZALEZ, J.; SHENKER, S.; STOICA, I. Apache spark: A unified engine for big data processing. **Communications of the ACM**, v. 59, p. 56–65, 11 2016.

ANEXO A – CHANGE DATA CAPTURE (CDC) COMO ESTRATÉGIA DE INTEGRAÇÃO: UMA ANÁLISE DETALHADA

A.1 Introdução

A técnica de CDC emergiu como a abordagem de integração primordial para estabelecer uma conexão eficiente entre sistemas monolíticos e subsequente implementação de microsserviços. Este capítulo aprofunda-se nos mecanismos e particularidades do CDC, elucidando seu funcionamento com base em estudos e pesquisas citados (TANK *et al.*, 2010) (SHI *et al.*, 2008) (CHANDRA, 2018) (WHAT. . . , 2023c)

A.2 Técnicas de CDC

O CDC adota várias técnicas para monitorar e capturar alterações em fontes de dados. A escolha entre essas técnicas depende das necessidades específicas do sistema, dos requisitos de latência e do impacto no desempenho.

A.2.1 Log-based CDC

Tabela de Origem

ID	Produto	Quantidade	Localização
1	Caixa A	20	Armazém 1
2	Pallet B	10	Armazém 2

Tabela de Destino (Registro de Transações)

Timestamp	Operação	ID	Dados Antigos	Dados Novos
2023-10-05 14:02	UPDATE	1	{Qtd: 20, Local: Arm 1}	{Qtd: 25, Local: Arm 1}
2023-10-05 14:03	INSERT	4	N/A	{Qtd: 15, Local: Arm 3}

Figura 10 – Tabelas com a técnica Log-based. Fonte: Elaborado pelo autor.

O *Log-based CDC* é uma técnica sofisticada que se baseia na leitura e interpretação dos logs do sistema para capturar mudanças nos dados em tempo real. Esta abordagem é particularmente eficaz em sistemas de banco de dados complexos e ambientes de alta transacionalidade, onde a precisão e a temporalidade das informações são cruciais.

Em sua essência, o *Log-based CDC* envolve o conceito de "journaling", um processo contínuo de registrar e monitorar todas as transações e modificações de dados no sistema. Cada operação que altera os dados, incluindo inserções, atualizações e exclusões, é meticulosamente registrada nos logs de transações. Estes registos documentam detalhes cruciais, como o momento da operação, os dados antes e após a mudança, e outros metadados relevantes, criando uma trilha auditável e rastreável de todas as atividades.

A riqueza de informações contidas nos logs de transações torna o *Log-based CDC* uma escolha inestimável para organizações que buscam uma visão detalhada e insights aprofundados de suas operações de dados. A técnica é frequentemente empregada em cenários de Big Data e analytics, onde a capacidade de acessar e analisar dados em tempo real é essencial para a tomada de decisões informada e ágil.

Na integração de sistemas heterogêneos, especialmente na transição e coexistência entre arquiteturas monolíticas e microsserviços, o *Log-based CDC* garante a sincronização e consistência dos dados entre plataformas. Minimiza a latência e assegura que os dados sejam capturados, processados e entregues com integridade e precisão.

A implementação eficaz desta técnica exige consideração das especificidades do sistema, a estrutura e a natureza dos dados e os requisitos específicos de negócios e técnicos. Ao integrar o *Log-based CDC*, as organizações podem otimizar a captura, o processamento e a entrega de dados modificados, potencializando a análise e a inteligência de negócios em tempo real.

A.2.2 Audit Columns

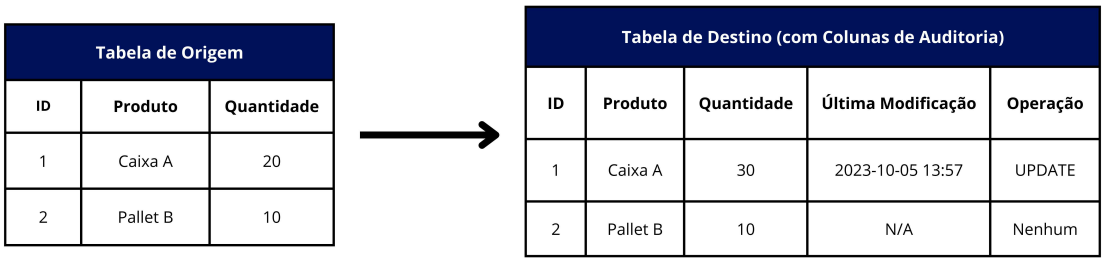

Figura 11 – Tabelas com a técnica Audit Columns. Fonte: Elaborado pelo autor.

As colunas de auditoria são campos específicos anexados a registros em um banco de dados para rastrear as últimas modificações. Elas registram automaticamente informações, como timestamps, cada vez que um registro é alterado, proporcionando uma maneira eficiente de identificar dados recentemente modificados.

A principal vantagem desse método é sua simplicidade e eficiência operacional. É comum em sistemas onde a identificação rápida de alterações de dados é necessária, mas um registro detalhado de cada modificação individual não é crítico. Por exemplo, sistemas Customer Relationship Management (CRM) e Enterprise Resource Planning (ERP) muitas vezes empregam colunas de auditoria para manter um rastreamento incremental dos registros alterados.

No entanto, há limitações a considerar. As colunas de auditoria não detectam exclusões, pois quando um registro é excluído, sua coluna de auditoria correspondente também é perdida. Além disso, quando usado um único campo para rastrear as modificações, distinguir entre inserções e atualizações pode ser desafiador.

Este método é uma opção atraente para sistemas que buscam uma solução de baixo impacto e de fácil implementação para o rastreamento de mudanças, oferecendo uma visão rápida das modificações recentes sem a necessidade de análise ou processamento intensivo.

A.2.3 Snapshot Differentials

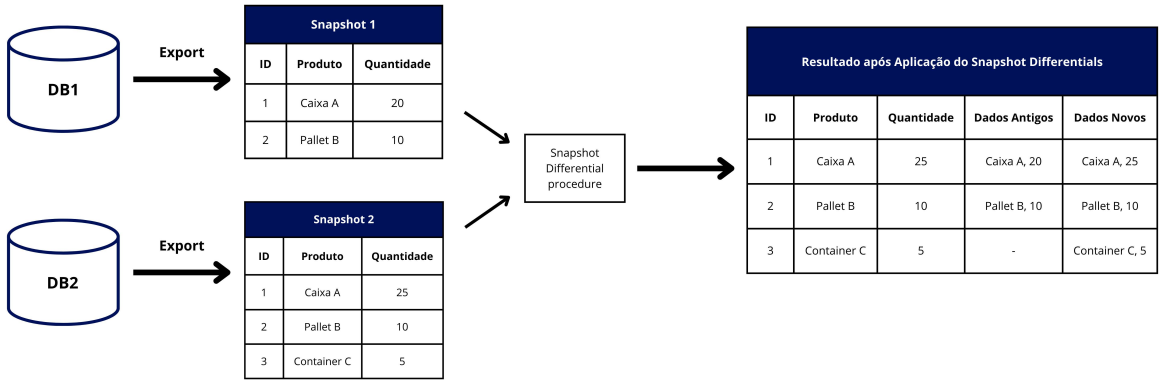

Figura 12 – Tabelas com a técnica Snapshot Differentials. Fonte: Elaborado pelo autor.

Os diferenciais de *snapshots* são uma técnica de CDC que se destaca pela sua aplicabilidade a sistemas que não possuem mecanismos intrínsecos para rastrear mudanças nos dados, como arquivos planos ou aplicativos legados. Esta abordagem se baseia na captura e comparação de estados inteiros de conjuntos de dados em momentos distintos para identificar modificações.

Em uma implementação típica, um *snapshot* completo dos dados é tirado em um ponto específico no tempo e preservado. Subsequentemente, um novo snapshot é capturado e comparado com o anterior, permitindo a identificação de registros adicionados, modificados ou excluídos. Esta técnica, embora simples e universalmente aplicável, vem com o custo de exigir a extração e o armazenamento de grandes volumes de dados em cada instância.

Embora possa ser considerado ineficiente, especialmente em cenários com grandes conjuntos de dados, os diferenciais de snapshots oferecem uma solução robusta para ambientes que não podem acomodar métodos de CDC mais complexos ou invasivos. É uma estratégia comum em sistemas que não suportam funcionalidades de log ou auditoria avançadas, garantindo que as mudanças nos dados ainda possam ser monitoradas e analisadas, apesar das limitações tecnológicas.

Um ponto a considerar é a latência na detecção de mudanças. Dado que esta técnica

depende da captura periódica de snapshots, as mudanças são identificadas em intervalos, em vez de em tempo real. Assim, é mais adequada para aplicativos e sistemas onde as atualizações em tempo real não são críticas e onde a análise periódica é suficiente para atender aos requisitos operacionais e analíticos.

A.2.4 Trigger-Based CDC

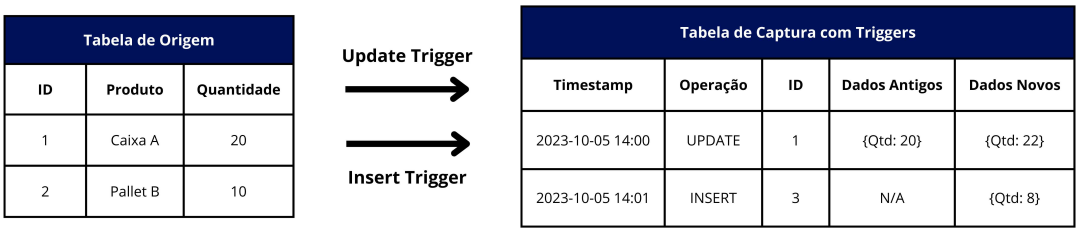

Figura 13 – Tabelas com a técnica Trigger-Based. Fonte: Elaborado pelo autor.

O *Trigger-Based CDC* é uma técnica dinâmica que emprega gatilhos no banco de dados para monitorar e registrar mudanças de dados em tempo real. Esses gatilhos são procedimentos automatizados acionados por eventos específicos, como inserções, atualizações ou exclusões de registros. Cada vez que um desses eventos ocorre, o gatilho é ativado e a informação sobre a mudança é instantaneamente capturada e armazenada em uma tabela separada. Esta abordagem assegura que cada modificação seja registrada conforme ocorre, permitindo uma resposta imediata e a garantia de que nenhuma alteração seja perdida ou não detectada.

Este método é altamente eficaz em ambientes que demandam atualizações de dados em tempo real e uma trilha auditável precisa das alterações. Em aplicações como plataformas de comércio eletrônico, sistemas de gestão de clientes e outras interfaces interativas, onde a visão instantânea e precisa dos dados é essencial para a funcionalidade e a experiência do usuário, o *Trigger-Based CDC* se mostra uma opção valiosa.

No entanto, também é essencial considerar o impacto no desempenho que os gatilhos podem ter. A ativação constante de gatilhos pode, em alguns casos, levar a um overhead significativo, especialmente em sistemas com um volume alto de transações. Portanto, uma avaliação cuidadosa do impacto potencial no desempenho e da capacidade do sistema em gerenciar essa carga adicional é vital.

A robustez e a reatividade do *Trigger-Based CDC* tornam-no uma escolha popular em muitas aplicações modernas. A habilidade de capturar, registrar e responder a mudanças de dados em tempo real não apenas potencializa a funcionalidade do sistema e a entrega de serviços, mas também facilita a conformidade regulatória e as iniciativas de governança de dados, assegurando que todas as alterações possam ser rastreadas, auditadas e gerenciadas eficazmente.

A.2.5 Timestamp-Based CDC

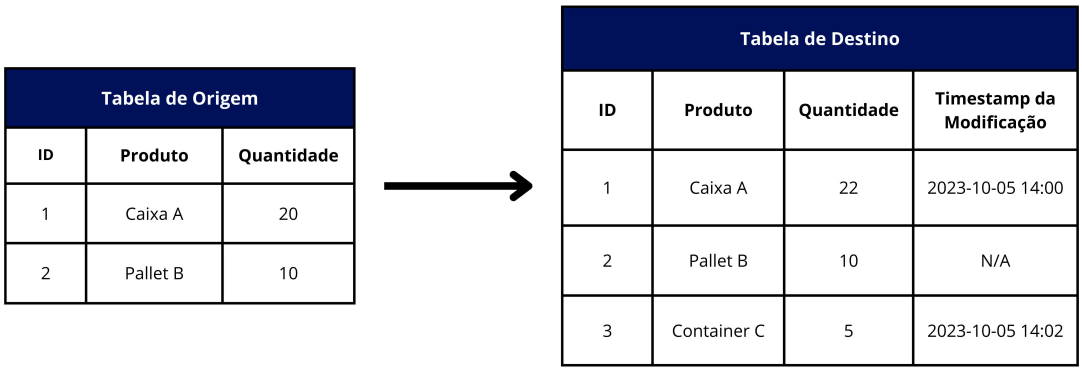

Figura 14 – Tabelas com a técnica *Timestamp-Based*. Fonte: Elaborado pelo autor.

O *Timestamp-Based CDC* é uma estratégia focada na eficiência, que se caracteriza por rastrear mudanças nos dados através da monitorização de colunas de *timestamp*. Neste método, as mudanças são identificadas e capturadas com base nos *timestamps* dos registros, sendo especialmente útil em cenários onde a minimização do impacto no desempenho e a eficiência operacional são prioritários.

A técnica se baseia em verificar regularmente os registros que possuem *timestamps*

mais recentes do que a última verificação. Assim, é possível identificar de maneira eficiente os dados que foram modificados, inseridos ou excluídos, sem a necessidade de inspecionar toda a base de dados ou depender de processos mais complexos e potencialmente intrusivos.

A vantagem distintiva do *Timestamp-Based CDC* reside em sua natureza não invasiva. A capacidade de identificar e capturar mudanças sem interagir extensivamente com o banco de dados ou afetar seu desempenho o torna adequado para sistemas sensíveis ou para ambientes onde a otimização dos recursos é crítica.

No entanto, é essencial garantir que os *timestamps* sejam gerenciados e sincronizados com precisão para evitar a perda de dados ou a captura imprecisa de mudanças. Em aplicações onde a consistência e a acurácia dos dados são imperativas, uma atenção especial deve ser dada à gestão dos *timestamps* para garantir que reflitam com precisão as operações de dados.

A.2.6 Polling-Based CDC

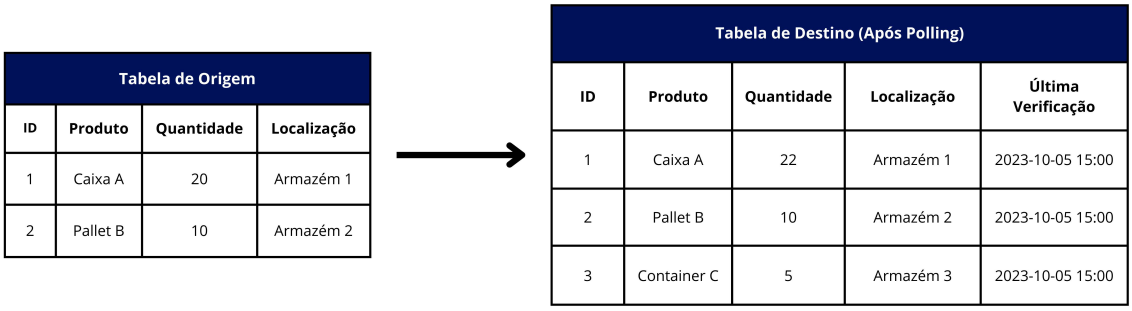

Figura 15 – Tabelas com a técnica Polling-Based CDC. Fonte: Elaborado pelo autor.

O *Polling-Based CDC* é um método que se caracteriza pela verificação periódica de conjuntos de dados para identificar e capturar alterações. Embora não seja tão eficiente em termos de tempo real quando comparado a outros métodos de CDC, ele se destaca pela sua flexibilidade e aplicabilidade em uma variedade de ambientes, especialmente onde métodos mais sofisticados de CDC não são praticáveis ou onde a infraestrutura de dados é limitada.

Nesta abordagem, a detecção de mudanças é realizada através de pesquisas regulares nos conjuntos de dados. O sistema é configurado para verificar em intervalos específicos e identificar alterações com base em critérios predeterminados, como timestamps ou identificadores de versão. É um método que, apesar de sua latência na detecção de mudanças, oferece a vantagem de ser menos invasivo e mais fácil de implementar.

O *Polling-Based CDC* é comumente encontrado em sistemas legados e ambientes onde a complexidade técnica e os recursos são limitados. Sua implementação não requer modificações significativas no sistema existente e pode ser adaptada para acomodar uma variedade de estruturas de dados e requisitos operacionais.

No entanto, é importante considerar as limitações associadas a este método. A natureza periódica das pesquisas implica que as mudanças são capturadas com um certo atraso, tornando este método menos adequado para aplicações que requerem detecção e resposta em tempo real às alterações dos dados. Além disso, a eficiência do *Polling-Based CDC* pode ser afetada pelo volume de dados e pela frequência das pesquisas, sendo essencial um balanceamento cuidadoso para otimizar o desempenho e a eficácia.

A.3 Comparação de Estratégias de CDC

A tabela abaixo compara as várias técnicas de CDC, destacando suas vantagens e limitações.

Tabela 3 – Comparação de estratégias de CDC

Critérios	Log-Based	Audit Columns	Snapshot Differentials	Trigger-Based	Timestamp-Based	Polling-Based
Atualização em Tempo Real	Sim	Não	Não	Sim	Não	Não
Impacto no Desempenho	Médio	Baixo	Alto	Alto	Baixo	Médio
Complexidade	Alta	Baixa	Média	Alta	Baixa	Baixa
Detecção de Exclusões	Sim	Não	Sim	Sim	Desafiador	Desafiador
Granularidade	Alta	Média	Baixa	Alta	Média	Baixa

Após a análise detalhada da tabela comparativa das técnicas de CDC, torna-se evidente que cada técnica se alinha distintamente a cenários específicos

A.3.0.1 Log-based CDC

Essa técnica se mostra excepcionalmente eficaz em ambientes que requerem uma captura detalhada de todas as modificações de dados. É uma escolha popular em sistemas financeiros e bancários, bem como em aplicações que envolvem a construção e manutenção de *Data Lakes*.

Os *Data Lakes* beneficiam-se da capacidade do *Log-based CDC* de capturar dados com granularidade fina, permitindo a incorporação de uma variedade diversificada de dados - estruturados e não estruturados - em um repositório centralizado para análises mais profundas. Isso facilita o acesso e a análise de grandes volumes de dados, habilitando insights mais ricos e informados.

Além disso, no contexto de análises de dados em tempo real, o *Log-based CDC* é incomparável. Ele permite que as organizações monitorem as mudanças nos dados conforme ocorrem, oferecendo insights instantâneos que são críticos para a tomada de decisões ágeis e informadas. Em cenários onde as condições de mercado, padrões de comportamento do consumidor ou outras variáveis dinâmicas estão em jogo, a capacidade de acessar e reagir aos dados em tempo real é um diferencial competitivo significativo.

A.3.0.2 Audit Columns

Ideal para cenários que requerem uma abordagem menos invasiva e onde a detecção de exclusões não é crítica. Pode ser bem aplicada em sistemas CRM ou ERP, onde as atualizações incrementais dos registros são frequentes, mas uma rastreabilidade detalhada de cada modificação não é necessária.

A.3.0.3 Snapshot Differentials

Esta técnica se encaixa bem em ambientes legados ou sistemas que não suportam técnicas de CDC mais avançadas. É frequentemente utilizada em aplicações que não requerem atualizações em tempo real e podem tolerar um processamento em lote, como análise de tendências ou relatórios que não são sensíveis ao tempo.

A.3.0.4 Trigger-Based CDC

Ótima para sistemas que necessitam de atualizações em tempo real e podem arcar com o impacto no desempenho associado ao uso de *triggers*. É comum em aplicativos de *e-commerce* e plataformas *online* interativas, onde a visão em tempo real dos dados é crucial para a experiência do usuário e operações de negócios.

A.3.0.5 Timestamp-Based CDC

Esta técnica é vantajosa em cenários onde é desejável minimizar o impacto no desempenho e ainda assim rastrear as mudanças de forma eficaz. É adequada para aplicações de análise de dados e BI, onde os dados precisam ser consistentes, mas não necessariamente em tempo real.

A.3.0.6 Polling-Based CDC

É muitas vezes uma escolha em sistemas legados ou em situações onde outras técnicas de CDC não são aplicáveis ou eficientes. É típica em ambientes que podem tolerar latência na atualização dos dados, como em sistemas de *backup* ou arquivamento.

A.4 Conclusões

O CDC (Change Data Capture) é essencial para orquestrar a integração eficiente entre monólitos e microsserviços, abordando os desafios inerentes à consistência e atualização de dados em ambientes heterogêneos. A personalização e adaptação meticulosas das estratégias de CDC são imperativas, garantindo que sejam alinhadas com as peculiaridades e requisitos únicos de cada sistema específico, promovendo, assim, uma integração otimizada.

A diversidade de métodos de CDC disponíveis amplia as opções para os engenheiros, permitindo seleções estratégicas baseadas em considerações precisas das necessidades funcionais e técnicas. No contexto contemporâneo, marcado pelo advento do big data e pela demanda crescente por informações em tempo real, a *Log-based CDC* ganha destaque, sendo amplamente adotada nos domínios da engenharia de dados.

No entanto, é crucial reconhecer que outras técnicas de CDC mantêm sua relevância e aplicabilidade em cenários específicos. Cada abordagem apresenta atributos distintos, e a

seleção apropriada pode ser instrumental para alcançar os objetivos desejados de integração e consistência de dados.