Universidade Federal de Juiz de Fora
Instituto de Ciências Exatas
Bacharelado em Sistemas de Informação

Atualização *Near Real-Time* em Arquiteturas Inspiradas em *Data Lakehouse*: Utilizando *Change Data Capture* e *Streaming ETL*

João Pedro Ferreira Pedreira

JUIZ DE FORA
JULHO, 2026

Atualização *Near Real-Time* em Arquiteturas Inspiradas em *Data Lakehouse*: Utilizando *Change Data Capture* e *Streaming ETL*

João Pedro Ferreira Pedreira

Universidade Federal de Juiz de Fora
Instituto de Ciências Exatas
Departamento de Ciência da Computação
Bacharelado em Sistemas de Informação

Orientador: Victor Stroele de Andrade Menezes

JUIZ DE FORA
JULHO, 2026

Atualização *Near Real-Time* em Arquiteturas Inspiradas em *Data Lakehouse*: Utilizando *Change Data Capture* e *Streaming ETL*

João Pedro Ferreira Pedreira

MONOGRAFIA SUBMETIDA AO CORPO DOCENTE DO INSTITUTO DE CIÊNCIAS EXATAS DA UNIVERSIDADE FEDERAL DE JUIZ DE FORA, COMO PARTE INTEGRANTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE BACHAREL EM SISTEMAS DE INFORMAÇÃO.

Aprovada por:

Victor Stroele de Andrade Menezes
Doutor em Engenharia de Sistemas e Computação - COPPE/UFRJ

Ciro de Barros Barbosa
Doutor em Computação - University of Twente

Edelberto Franco Silva
Doutor em Computação - Universidade Federal Fluminense (UFF)

JUIZ DE FORA

09 DE JULHO, 2026

Resumo

O modelo tradicional de carga de dados em *batch* entre sistemas transacionais e ambientes analíticos tem se mostrado insuficiente diante da crescente necessidade de tomada de decisão com base em dados recentes. Neste cenário, o paradigma *data lakehouse* propõe unir a flexibilidade dos *data lakes* às garantias de governança típicas dos *data warehouses*. O presente trabalho tem como objetivo desenvolver e avaliar um protótipo de atualização *near real-time* entre um banco relacional e um ambiente analítico organizado em camadas bronze/silver/gold, inspirado na arquitetura *lakehouse*. Para isso, foram utilizados MySQL, Debezium, Apache Kafka e um consumidor escrito em PHP como motor de *Streaming ETL*, operando sobre dados reais do Cadastro Nacional de Estabelecimentos de Saúde (CNES/DATASUS). Um ponto importante do protótipo é que o dado analítico é construído inteiramente a partir do fluxo de *streaming*, sem consultas ao banco de dados de origem. O ambiente foi todo montado com Docker e ferramentas *open source*. Os testes, executados sobre uma grade de oito cenários de carga (1 a 100 op/s), com cinco repetições por cenário, mostraram que o *pipeline* sustenta o processamento em regime *near real-time* para cargas de até ∼25 op/s. O joelho da curva de desempenho foi localizado entre 25 e 40 op/s, e os cenários de saturação (60 a 100 op/s) evidenciaram o principal fator limitante, a natureza *single-threaded* do consumidor, com degradação progressiva por acúmulo de *backlog*, atingindo latências médias de dezenas a centenas de segundos. No regime sustentável, a latência média ficou entre 232 e 361 milissegundos, confirmando a viabilidade da abordagem, e a decomposição instrumentada indicou que a quase totalidade do tempo ponta a ponta se concentra no intervalo entre o *commit* no MySQL e a chegada do evento ao consumidor (compatível com o `poll.interval.ms` padrão de 500 ms do Debezium, cuja metade, ∼250 ms, corresponde à latência observada), enquanto o processamento em PHP contribui com poucos milissegundos.

Palavras-chave: *Data Lakehouse*. *Change Data Capture*. *Streaming ETL*. Apache Kafka. Atualização *near real-time*.

Abstract

The traditional batch data-loading model between transactional systems and analytical environments has proven insufficient given the growing need for decision-making based on fresh data. In this context, the *data lakehouse* paradigm proposes to combine the flexibility of *data lakes* with the governance guarantees typical of *data warehouses*. This work aims to develop and evaluate a *near real-time* update prototype between a relational database and an analytical environment organized into bronze/silver/gold layers, inspired by the *lakehouse* architecture. To this end, MySQL, Debezium, Apache Kafka, and a PHP-based consumer were used as the *Streaming ETL* engine, operating over real data from the Brazilian National Registry of Health Establishments (CNES/DATASUS). A key aspect of the prototype is that the analytical data is built entirely from the streaming flow, without any queries to the source database. The whole environment was assembled with Docker and *open source* tools. The tests, executed over a grid of eight load scenarios (1 to 100 op/s) with five repetitions each, showed that the *pipeline* sustains *near real-time* processing for loads of up to ∼25 op/s. The knee of the performance curve was located between 25 and 40 op/s, and the saturation scenarios (60 to 100 op/s) revealed the main limiting factor, the *single-threaded* nature of the consumer, with progressive degradation due to backlog accumulation, reaching average latencies of tens to hundreds of seconds. In the sustainable regime, the average latency stayed between 232 and 361 milliseconds, confirming the feasibility of the approach, and the instrumented decomposition indicated that almost all of the end-to-end time is concentrated in the interval between the commit in MySQL and the arrival of the event at the consumer (consistent with Debezium's default `poll.interval.ms` of 500 ms, whose half, ∼250 ms, matches the observed latency), while PHP processing contributes only a few milliseconds.

Keywords: *Data Lakehouse*. *Change Data Capture*. *Streaming ETL*. Apache Kafka. *Near real-time* update.

Agradecimentos

Ao professor Victor Stroele de Andrade Menezes, pela orientação, paciência e disponibilidade ao longo do desenvolvimento deste trabalho.

Aos meus familiares e amigos, pelo apoio e incentivo constantes.

Aos professores e funcionários do Departamento de Ciência da Computação da UFJF, que contribuíram para a minha formação.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas e Siglas

ACID	*Atomicity, Consistency, Isolation, Durability*
API	*Application Programming Interface*
CBO	Classificação Brasileira de Ocupações
CDC	*Change Data Capture*
CNES	Cadastro Nacional de Estabelecimentos de Saúde
CSV	*Comma-Separated Values*
DATASUS	Departamento de Informática do Sistema Único de Saúde
ETL	*Extract, Transform, Load*
ESF	Estratégia Saúde da Família
IBGE	Instituto Brasileiro de Geografia e Estatística
JSON	*JavaScript Object Notation*
JSONL	*JSON Lines*
OLTP	*Online Transaction Processing*
PK	*Primary Key*
SQL	*Structured Query Language*
SUS	Sistema Único de Saúde
UBS	Unidade Básica de Saúde
UPA	Unidade de Pronto Atendimento
UFJF	Universidade Federal de Juiz de Fora

1 Introdução

Nos últimos anos, o volume de dados gerados por aplicações web, dispositivos móveis e sistemas empresariais cresceu de forma expressiva. Esse crescimento trouxe desafios que vão além do que os *data warehouses* tradicionais conseguem atender de forma satisfatória, tanto em termos de variedade de dados quanto de velocidade de atualização (MAZUMDAR; HUGHES; ONOFRé, 2023). Diante desse cenário, surgiram os *data lakes*, que oferecem maior flexibilidade para armazenar dados em diversos formatos, mas, por outro lado, sofrem com problemas de governança e qualidade (HARBY; ZULKERNINE, 2025).

Mais recentemente, surgiu o conceito de *data lakehouse*, que busca unir o melhor dos dois mundos: a flexibilidade do *data lake* às garantias transacionais e de governança do *data warehouse*. Tecnologias como Delta Lake, Apache Iceberg e Apache Hudi são as principais representantes desse paradigma (Databricks, 2024; ARMBRUST et al., 2021). Paralelamente, há uma demanda crescente por parte das organizações para que os dados analíticos estejam o mais próximo possível do tempo real. Muitas empresas e instituições ainda dependem de cargas em *batch*, diárias ou horárias para alimentar seus ambientes analíticos, o que gera uma defasagem entre o que ocorreu no sistema operacional e o que está disponível para análise.

Técnicas como *Change Data Capture* (CDC) permitem capturar alterações em bancos de dados transacionais e propagá-las continuamente para outros sistemas (Red Hat, 2026). Quando combinadas com um motor de *Streaming ETL* e uma organização em camadas, como a arquitetura medalhão (bronze/silver/gold), essas técnicas possibilitam a construção de ambientes analíticos que refletem o estado operacional com atraso de poucos segundos.

É nesse contexto que este trabalho se insere. A questão de pesquisa que o norteia é a seguinte: Como uma arquitetura inspirada em *lakehouse* alimentada exclusivamente por *Change Data Capture* e *Streaming ETL*, implementada com ferramentas *open source* em um ambiente conteinerizado em uma única máquina, se comporta em termos de latência ponta a ponta e de *throughput* sustentado à medida que a carga aumenta, e quais são os

fatores que limitam o seu desempenho?

Para responder a essa pergunta, definiu-se como objetivo geral implementar e analisar uma arquitetura de atualização *near real-time* entre um banco de dados relacional e um ambiente analítico organizado como *data lakehouse* simplificado, usando *Change Data Capture* e *Streaming ETL* sobre dados reais do CNES/DATASUS, avaliando principalmente a capacidade de processamento (*throughput* sustentado) e os fatores que limitam o desempenho do *pipeline* e, de forma complementar, a latência ponta a ponta, a consistência da propagação e a complexidade de implantação. Como toda a avaliação ocorre em uma única máquina, sem latência de rede, os valores absolutos de latência devem ser lidos como próprios desse cenário e úteis sobretudo para decompor onde o tempo é gasto, e não como métrica generalizável de desempenho. Ressalva-se, ainda, que a consistência é avaliada somente em condições nominais de operação, sem injeção de falhas (queda do consumidor, partição de rede ou interrupção de escrita), de modo que os resultados atestam a fidelidade da propagação, e não a tolerância a falhas do protótipo. Esse objetivo geral se desdobra nos seguintes objetivos específicos:

- Revisar na literatura os conceitos de *data warehouse*, *data lake* e *data lakehouse*, com foco em ingestão incremental.
- Estudar o funcionamento do CDC baseado em *logs* de transação, em particular o Debezium com Kafka Connect.
- Propor uma arquitetura de *Streaming ETL* que consuma eventos do Kafka e atualize o repositório analítico sem acessar o banco de origem.
- Implementar o protótipo em Docker, integrando MySQL, Kafka, Debezium e o consumidor PHP, operando sobre dados do CNES.
- Coletar e analisar métricas de processamento (*throughput* sustentado e fatores limitantes) e, de forma complementar, de latência entre a escrita no banco transacional e a disponibilidade do dado na camada analítica.
- Discutir limitações e possibilidades de evolução do protótipo.

Do ponto de vista metodológico, optou-se por conduzir a avaliação com base em dados reais do Cadastro Nacional de Estabelecimentos de Saúde (CNES), mantido pelo DATASUS/Ministério da Saúde (Ministério da Saúde, 2026). O CNES registra todos os

estabelecimentos de saúde do Brasil, públicos e privados, com informações sobre profissionais, equipes, equipamentos e serviços oferecidos. Essa escolha traz duas vantagens. O domínio é publicamente acessível e relevante do ponto de vista social, e o volume da base, com mais de 22 milhões de registros, representa um cenário realista para avaliar o comportamento do *pipeline*.

Sobre essa base, foi implementado um protótipo que integra o MySQL como banco de dados transacional, o Debezium para a captura de mudanças, o Apache Kafka como barramento de eventos e um consumidor, escrito em PHP, no papel de *Streaming ETL*. Os dados são persistidos em um repositório organizado na arquitetura medalhão. Vale destacar que toda a camada analítica, incluindo as agregações *gold*, é construída a partir dos dados que passaram pelo *pipeline*, sem nenhuma consulta direta ao banco de origem. Dessa forma, o CDC e o *Streaming ETL* são, de fato, responsáveis por levar a informação da origem ao destino.

O restante deste trabalho está organizado da seguinte forma. O Capítulo 2 apresenta a fundamentação teórica. O Capítulo 3 discute os trabalhos relacionados. O Capítulo 4 descreve a arquitetura proposta. O Capítulo 5 detalha a implementação. O Capítulo 6 reúne os experimentos e os resultados. Por fim, o Capítulo 7 encerra o texto com as conclusões.

2 Fundamentação Teórica

Esta seção apresenta os principais conceitos necessários para compreender o trabalho desenvolvido. A discussão começa pelos modelos de armazenamento analítico, passa pelas técnicas de captura e propagação de dados, e termina com as ferramentas de infraestrutura utilizadas.

2.1 *Data Warehouse*

O *Data Warehouse* é um tipo de repositório voltado para análise e inteligência de negócio. A ideia, formalizada por Inmon no início dos anos 1990, é reunir dados de diversas fontes operacionais em um local centralizado, onde possam ser consultados de forma eficiente para apoiar decisões estratégicas (INMON, 2005). Nele, os dados são organizados em esquemas bem definidos, como o modelo estrela ou floco de neve, priorizando consistência e desempenho em consultas analíticas (KIMBALL; ROSS, 2013).

A forma mais comum de alimentar um *data warehouse* é através de processos ETL (*Extract, Transform, Load*) que rodam em janelas periódicas, geralmente uma vez por dia ou a cada hora. Esse modelo funciona razoavelmente bem quando não há necessidade de dados muito recentes. Porém, quando a organização precisa de informações mais atualizadas para tomar decisões, essa defasagem inerente ao *batch* passa a ser um problema (HARBY; ZULKERNINE, 2025).

Outra limitação dos *data warehouses* tradicionais é a dificuldade de lidar com dados não estruturados ou semiestruturados, como *logs*, JSONs e dados de sensores, que têm se tornado cada vez mais comuns. Nesses casos, o custo de transformar tudo em um esquema relacional rígido pode ser proibitivo, tanto em tempo quanto em complexidade. Isso motivou o surgimento de abordagens mais flexíveis, como os *data lakes*.

Do ponto de vista operacional, o processo ETL que alimenta o DW possui três etapas bem definidas (KIMBALL; ROSS, 2013). A extração (*Extract*) lê os dados das diversas fontes operacionais, que podem ser bancos relacionais, arquivos CSV, APIs ou

sistemas legados. A transformação (*Transform*) aplica regras de limpeza, padronização (como converter formatos de data, unificar codificações de caracteres e resolver conflitos entre fontes diferentes), agregação e enriquecimento. A carga (*Load*) insere os dados transformados nas tabelas dimensionais do DW. Esse processo costuma ser orquestrado por ferramentas como Apache Airflow, Informatica ou Talend, e sua execução pode levar de minutos a horas dependendo do volume de dados.

Apesar de suas limitações, o *data warehouse* continua sendo amplamente utilizado na indústria. Muitas organizações operam com uma arquitetura híbrida onde o DW coexiste com outras soluções. Neste trabalho, o *data warehouse* aparece como ponto de partida conceitual: o objetivo final do *pipeline* é alimentar uma camada analítica que cumpra função semelhante à de um DW, mas com atualização contínua em vez de cargas periódicas.

2.2 *Data Lake*

O *Data Lake* é um repositório que armazena dados em grande escala, no formato original, sem exigir um esquema predefinido. A ideia é aplicar a estrutura apenas na hora de consumir os dados, no que se chama de *schema-on-read* (HARBY; ZULKERNINE, 2025). Essa abordagem contrasta com o *schema-on-write* dos *data warehouses*, onde os dados precisam ser transformados antes de serem armazenados.

Na prática, isso traz bastante flexibilidade, já que dados estruturados, semiestruturados e não estruturados podem conviver no mesmo repositório, normalmente em cima de um *object storage* como Amazon S3, HDFS ou MinIO. Organizações podem armazenar dados brutos sem saber de antemão como serão usados, o que agiliza a ingestão e permite explorar os dados de formas não previstas inicialmente.

O problema é que, sem controles adequados, o *data lake* pode se tornar desorganizado, o que a literatura chama de *data swamp* (pântano de dados). Sem transações ACID, sem controle de versão e sem otimizações para consultas, o *data lake* acaba sendo limitado para uso em *business intelligence* (MAZUMDAR; HUGHES; ONOFRé, 2023). A governança dos dados se torna difícil: não há como garantir que um dado não foi corrompido, que uma leitura reflete um estado consistente, ou que uma atualização não

deixou o repositório em estado inconsistente.

Apesar dessas limitações, o *data lake* trouxe um avanço importante ao desacoplar o armazenamento do processamento. Num DW tradicional, o motor de consulta e o armazenamento estão fortemente acoplados, o Oracle, por exemplo, gerencia tanto o armazenamento físico quanto a execução das consultas. No *data lake*, os dados ficam em um *object storage* genérico e podem ser processados por qualquer ferramenta: Spark, Presto, Trino, Hive, entre outros. Essa separação é uma das bases do paradigma *lakehouse*, que busca manter a flexibilidade de armazenamento enquanto adiciona as garantias que faltam ao *data lake* puro.

No protótipo deste trabalho, o ambiente Docker provisiona um contêiner MinIO compatível com a API S3 como infraestrutura preparada para evolução futura. Na implementação atual, contudo, os dados do *lakehouse* são gravados diretamente no *filesystem* local (via `file_put_contents()` sobre o volume `lakehouse-data/`), por simplicidade do protótipo. A migração para *object storage* real (MinIO localmente ou S3 em nuvem) é apontada como evolução natural no Capítulo 7, exigindo apenas a substituição da camada de I/O do `LakehouseWriter` sem alterar a lógica do *pipeline*.

2.3 *Data Lakehouse*

O *Data Lakehouse* é um conceito que tenta unir o melhor dos dois modelos anteriores: manter a flexibilidade de armazenamento do *data lake* e adicionar as garantias de governança e capacidade analítica do *data warehouse* (Databricks, 2024). Os dados continuam em formatos abertos sobre um *storage* distribuído, mas ganham uma camada de metadados que permite operações transacionais e consultas mais eficientes.

Armbrust et al. (ARMBRUST et al., 2021) propuseram formalmente a arquitetura *lakehouse* como uma nova geração de plataformas que unificam *data warehousing* e análise avançada. A proposta parte de uma constatação prática: muitas organizações mantêm tanto um *data lake* quanto um *data warehouse*, com pipelines ETL complexos movendo dados entre eles. Isso gera custo, duplicação e risco de inconsistência. O *lakehouse* elimina essa redundância ao adicionar uma camada transacional diretamente sobre o *data lake*.

As tecnologias mais conhecidas nesse espaço são Delta Lake, Apache Hudi e Apache Iceberg (Apache Software Foundation, 2024c; Apache Software Foundation, 2024a). Cada uma oferece transações ACID, versionamento de dados (*time travel*), evolução de esquema e otimizações de leitura. Embora **nenhum deles seja utilizado no protótipo** (a camada de armazenamento adotada é JSONL sobre *filesystem* local — ver Seção 4.6), eles representam a evolução natural mais direta do protótipo e são retomados no Capítulo 7 como trabalho futuro. Apresentá-los aqui contextualiza as limitações arquiteturais discutidas adiante.

Cabe ressaltar que existe uma distinção importante entre o conceito arquitetural do *lakehouse* e suas implementações completas com formatos transacionais. Elementos como a organização em camadas (bronze/silver/gold), a ingestão contínua de dados e a separação entre armazenamento e processamento já representam uma adoção parcial do paradigma, mesmo sem um formato como Delta ou Iceberg. Neste trabalho, adota-se essa visão incremental: o protótipo implementa o padrão arquitetural do *lakehouse* sem recorrer a formatos transacionais proprietários, e essa limitação é discutida com transparência.

2.4 Arquitetura Medalhão

A arquitetura medalhão é um padrão bastante utilizado em projetos de *lakehouse*. Ela organiza os dados em três camadas com níveis crescentes de qualidade e refinamento (Databricks, 2024):

A camada **bronze** armazena os dados no formato mais bruto possível, incluindo metadados do sistema de captura. A ideia é manter tudo que veio da origem, sem transformação, como um registro histórico completo. Se algo der errado nas camadas seguintes, os dados brutos podem ser reprocessados.

A camada **silver** contém os dados já processados: campos limpos, tipos normalizados, metadados internos removidos. É nessa camada que acontecem as transformações de qualidade, remoção de duplicatas, padronização de formatos de data, conversão de tipos. Os dados na *silver* representam uma visão confiável e consistente do que veio da origem.

A camada **gold** contém os dados prontos para consumo, na forma de agregações e visões desnormalizadas. Essa camada é construída a partir da *silver*, e não diretamente da origem, o que é um princípio importante da arquitetura. As agregações da *gold* respondem perguntas de negócio de forma direta, sem exigir que o analista conheça o modelo relacional original.

Essa separação em camadas traz benefícios práticos. Primeiro, facilita o rastreamento da origem dos dados (o chamado *data lineage*): para qualquer dado na camada *gold*, é possível rastrear de onde veio na *silver*, e consequentemente na *bronze*. Segundo, permite reprocessar os dados em caso de problemas, se uma regra de transformação precisa mudar, basta reaplicá-la sobre a *bronze* sem precisar refazer a ingestão. Terceiro, organiza as responsabilidades: cada camada tem um propósito claro, evitando a desorganização que levaria ao *data swamp*.

No contexto deste trabalho, a arquitetura medalhão é implementada sobre o *filesystem* local, usando arquivos JSONL (*JSON Lines*) em cada camada. A escolha do JSONL, em vez de formatos colunares como Parquet, é discutida no Capítulo 4. Para o CNES, as transformações da camada silver incluem operações como `trim()` em campos de texto (necessário porque os dados do DATASUS frequentemente contêm espaços em excesso), conversão de *timestamps* do formato Debezium (milissegundos desde a *epoch* Unix) para formato legível, e *casting* de tipos numéricos. Cada registro silver recebe metadados de processamento (`_processed_at`, `_operation`, `_is_deleted`) que são essenciais para a reconstrução de estado na camada gold.

2.5 *Change Data Capture* (CDC)

O CDC (*Change Data Capture*) é uma técnica que permite detectar e capturar as alterações feitas em um banco de dados: inserções, atualizações e exclusões, e transformá-las em eventos que outros sistemas podem consumir (Red Hat, 2026). Em vez de copiar tabelas inteiras periodicamente (como faz o ETL *batch* convencional), o CDC propaga apenas o que mudou, o chamado "delta" de alterações. Isso reduz drasticamente o volume de dados transferidos e viabiliza atualizações muito mais frequentes, em tese, tão rápidas quanto as próprias transações no banco de origem.

Para entender a magnitude dessa diferença, considere um cenário com a base do CNES: a tabela de estabelecimentos de saúde possui mais de 600 mil registros, e num dia típico talvez 50 desses registros sejam atualizados (mudança de telefone, atualização de coordenadas, cadastro de um novo estabelecimento). No modelo *batch*, seria necessário copiar os 600 mil registros para garantir que os 50 alterados estejam no destino, um desperdício de 99,99% da transferência. Com CDC, apenas os 50 eventos de alteração são propagados, permitindo que a propagação aconteça em tempo real.

Existem diferentes abordagens para implementar CDC, cada uma com seus *trade-offs* (JAIN et al., 2023):

A abordagem **baseada em *log* de transação** (*log-based CDC*) é considerada a mais eficiente e menos intrusiva. Ela lê diretamente o registro de mudanças que o banco de dados já mantém para fins de recuperação e replicação, como o *binlog* do MySQL, o WAL (*Write-Ahead Log*) do PostgreSQL ou o *redo log* do Oracle. Como esse *log* já existe independentemente do CDC, lê-lo não gera carga adicional significativa no banco. Essa abordagem captura todas as operações na ordem exata em que ocorreram, incluindo operações que não atualizam nenhuma coluna de *timestamp*, o que é uma limitação das abordagens baseadas em *polling*.

A abordagem **por *triggers*** insere gatilhos (*triggers*) nas tabelas monitoradas. A cada inserção, atualização ou exclusão na tabela original, o *trigger* dispara automaticamente uma gravação em uma tabela auxiliar de auditoria. Embora seja precisa e funcione em qualquer SGBD que suporte *triggers*, essa abordagem tem impacto direto no desempenho: cada transação na tabela original passa a executar lógica adicional. Em tabelas com alto volume de escritas, isso pode se tornar um gargalo significativo. Além disso, requer modificações no esquema do banco, o que pode ser inviável em sistemas legados ou compartilhados entre equipes.

A abordagem **por *polling*** consulta periodicamente as tabelas em busca de registros com *timestamp* de modificação (como `updated_at`) mais recente que a última consulta. É a mais simples de implementar e não requer configurações especiais no banco. Porém, possui limitações significativas: depende da existência de colunas de *timestamp* em todas as tabelas monitoradas (nem sempre garantido em sistemas legados), não de-

tecta exclusões físicas (apenas exclusões lógicas com *soft delete*), e tem latência mínima igual ao intervalo de *polling*, se o *polling* acontece a cada minuto, a latência mínima é de um minuto.

2.5.1 Debezium

O Debezium é a principal ferramenta *open source* para CDC *log-based* (Red Hat, 2026). Desenvolvido pela Red Hat e mantido pela comunidade, ele funciona como um conector do Kafka Connect, um *framework* para integração de dados do ecossistema Kafka e publica cada alteração detectada no *binlog* como um evento JSON estruturado em um tópico do Apache Kafka.

Cada evento publicado pelo Debezium contém campos bem definidos: `before` (estado anterior do registro, nulo para inserções), `after` (estado posterior, nulo para exclusões), `op` (tipo da operação: `c` para *create*, `u` para *update*, `d` para *delete*, `r` para *read*/*snapshot*), `ts_ms` (*timestamp* do evento no conector) e `source` (metadados sobre o servidor de origem, incluindo arquivo do *binlog* e posição exata da transação). Essa estrutura permite que o consumidor saiba exatamente o que mudou, quando mudou, e reconstrua tanto o estado anterior quanto o posterior de cada registro afetado.

Na primeira execução do conector, o Debezium realiza um *snapshot* completo de todas as tabelas monitoradas. Todos os registros existentes são publicados como eventos com operação `r`, com *timestamps* do momento da leitura. É essencial distinguir esses eventos de *snapshot* dos eventos de CDC "real": eles representam o estado inicial do banco e não alterações transacionais. Após o *snapshot*, o Debezium passa a ler o *binlog* em tempo real, capturando apenas as mudanças incrementais. Essa distinção tem impacto direto nos experimentos deste trabalho, pois eventos de *snapshot* podem gerar latências artificialmente altas quando medidos contra o relógio do sistema.

Diversas configurações do Debezium merecem destaque por terem impactado diretamente o desenvolvimento deste protótipo. Campos do tipo `DECIMAL` no MySQL são, por padrão, serializados pelo Debezium como bytes codificados em Base64. Na prática, isso significa que um valor como "3499.90" chega ao consumidor como a *string* "BVcm", sem nenhum erro ou aviso, apenas valores incorretos e silenciosos. A solução é configu-

rar o parâmetro `decimal.handling.mode` como `string`. De forma similar, o parâmetro `time.precision.mode` precisa ser configurado como `connect` para compatibilidade de *timestamps* com o PHP, e a propriedade `database.connectionProperties` precisa incluir `useUnicode=true;characterEncoding=UTF-8` para que nomes de estabelecimentos e profissionais com acentos sejam lidos corretamente nos dados do CNES.

O Listing 2.1 mostra um exemplo de evento CDC publicado pelo Debezium para a criação de um novo estabelecimento de saúde.

```
{
  "before": null,
  "after": {
    "co_unidade": "3550302000001",
    "co_cnes": "2070316",
    "no_fantasia": "UBS SANTANA",
    "co_municipio_gestor": "355030",
    "co_tipo_estabelecimento": "002",
    "nu_latitude": "-23.510500",
    "nu_longitude": "-46.625800",
    "created_at": 1710722034000,
    "updated_at": 1710722034000
  },
  "source": {
    "version": "2.5.0.Final",
    "connector": "mysql",
    "name": "cdc",
    "ts_ms": 1710722034000,
    "db": "cnes_db",
    "table": "estabelecimentos",
    "pos": 1234
  },
  "op": "c",
  "ts_ms": 1710722034123
}
```

Listing 2.1: Exemplo de evento CDC do Debezium para um INSERT em estabelecimentos.

O campo `before` é nulo porque se trata de uma inserção (não havia registro anterior). O campo `after` contém todos os valores do novo registro. Note que os *timestamps* `created_at` e `updated_at` chegam como milissegundos desde a *epoch* Unix (1710722034000 corresponde a 2024-03-18 02:13:54 UTC), exigindo conversão no `EventTransformer`.

O campo `ts_ms` no nível raiz (1710722034123) é o *timestamp* do Debezium, e a diferença entre ele e o `source.ts_ms` corresponde à latência interna do conector. Os valores deste exemplo são ilustrativos, em que a diferença de 123ms mostrada aqui serve apenas para evidenciar a existência dos dois timestamps. Na prática, conforme medido nos experimentos, essa parcela interna do conector (`latency_dbz_ms`) ficou entre 2 e 5 ms em todos os cenários.

2.6 Apache Kafka

O Apache Kafka é uma plataforma de *streaming* de eventos bastante utilizada na indústria (KREPS; NARKHEDE; RAO, 2011). Ele funciona como um barramento de mensagens de alto desempenho, desacoplando quem produz dados de quem consome. Na sua essência, o Kafka é um *log* distribuído e persistente onde os eventos são gravados de forma sequencial e podem ser lidos por múltiplos consumidores de forma independente.

Os conceitos fundamentais do Kafka são: **tópicos**, que funcionam como canais lógicos de eventos onde cada tópico agrupa mensagens de um mesmo tipo, **partições**, que subdividem os tópicos para permitir paralelismo tanto na escrita quanto na leitura, **produtores**, que publicam eventos nos tópicos, **consumidores**, que leem eventos dos tópicos, e **offsets**, que controlam a posição de leitura de cada consumidor em cada partição (Apache Software Foundation, 2024b).

Um aspecto relevante do Kafka para este trabalho é a persistência das mensagens. Diferente de filas tradicionais onde a mensagem é removida após o consumo, o Kafka mantém as mensagens por um período configurável (por padrão, 7 dias). Isso significa que um consumidor pode reler eventos passados, o que permite, por exemplo, reprocessar dados caso alguma transformação precise ser corrigida. Kleppmann (KLEPPMANN, 2017) destaca que essa propriedade de *log* imutável é fundamental para arquiteturas orientadas

a eventos.

No protótipo desenvolvido, o Kafka recebe os eventos do Debezium e os disponibiliza para o consumidor PHP. Cada tabela monitorada no MySQL gera um tópico correspondente no Kafka, seguindo o padrão de nomenclatura `cdc.cnes_db.{tabela}`. O Kafka é executado junto com o Zookeeper (para coordenação) em contêineres Docker dentro da mesma rede interna.

2.7 *Streaming ETL*

O *Streaming ETL* se diferencia do ETL tradicional por processar os dados de forma contínua, à medida que eles chegam, em vez de esperar acumular um lote grande. Isso reduz bastante a latência entre a geração do dado e sua disponibilidade para análise (ARMBRUST et al., 2018).

O fluxo básico é o mesmo do ETL convencional, extração, transformação e carga, mas acontece evento por evento ou em micro-lotes. A extração corresponde à leitura de eventos de uma fonte (no caso, o Kafka). A transformação aplica regras de limpeza, normalização e enriquecimento. A carga grava o resultado no destino (no caso, as camadas do *lakehouse* e o PostgreSQL).

As ferramentas de referência no mercado para *Streaming ETL* são o Apache Flink (CARBONE et al., 2015) e o Spark Structured Streaming (ARMBRUST et al., 2018; ZAHARIA et al., 2016). Ambas oferecem processamento paralelo, tolerância a falhas com *checkpointing* e garantias de entrega (*exactly-once* ou *at-least-once*). O Flink adota um modelo de processamento evento a evento com gerenciamento de estado nativo, enquanto o Spark trata *streams* como tabelas incrementais que recebem micro-lotes de dados.

Neste trabalho, optou-se por utilizar um consumidor PHP como motor de *Streaming ETL*. Essa escolha pode parecer incomum, mas faz sentido no contexto do protótipo. O objetivo não é avaliar uma ferramenta de *streaming* específica, e sim demonstrar que o fluxo completo (CDC até *lakehouse*) funciona e medir sua latência. O PHP, por ser uma linguagem de amplo conhecimento e fácil de rodar, torna o protótipo mais acessível e reproduzível. A possibilidade de substituir o PHP por Flink ou Spark é discutida nas limitações e trabalhos futuros.

2.8 Docker e Conteinerização

O Docker permite empacotar aplicações e suas dependências em contêineres isolados, garantindo que o software funcione da mesma forma em qualquer ambiente, da máquina do desenvolvedor até um servidor de produção (MERKEL, 2014). Com o Docker Compose, é possível orquestrar múltiplos contêineres que formam um sistema distribuído, tudo a partir de um único arquivo de configuração (o `docker-compose.yaml`).

A escolha do Docker neste trabalho se deu por três motivos. Primeiro, permitir que o ambiente seja reproduzido facilmente: qualquer pessoa com Docker instalado pode subir toda a infraestrutura com um único comando. Segundo, possibilitar a execução de toda a infraestrutura (Kafka, MySQL, PostgreSQL, Debezium, MinIO, PHP) numa única máquina, sem precisar de múltiplos servidores. Terceiro, não ter custo, já que todas as ferramentas utilizadas são *open source*.

No protótipo, o Docker Compose orquestra nove serviços que compõem o *pipeline*. Cada serviço roda em seu próprio contêiner com rede, volumes e variáveis de ambiente configurados de forma declarativa. Os bancos de dados usam o mecanismo de inicialização automática do Docker: scripts SQL colocados no diretório `/docker-entrypoint-initdb.d/` são executados na primeira vez que os contêineres sobem, criando os esquemas e populando os dados iniciais sem intervenção manual.

2.9 O CNES e o contexto dos dados

O Cadastro Nacional de Estabelecimentos de Saúde (CNES) é o sistema oficial do Ministério da Saúde que registra todos os estabelecimentos de saúde do Brasil, sejam públicos ou privados (Ministério da Saúde, 2026). Implantado em 2000, o sistema tem como objetivo cadastrar e manter atualizadas as informações sobre a infraestrutura de saúde do país, servindo como referência para planejamento, regulação, avaliação e controle dos serviços de saúde.

A base de dados do CNES é atualizada mensalmente e exportada pelo DATASUS em arquivos CSV para download público (DATASUS, 2026). A exportação de fevereiro de 2026, utilizada como referência neste trabalho, contém mais de 100 arquivos CSV que

abrangem desde os dados cadastrais dos estabelecimentos até a composição de equipes, inventário de equipamentos, serviços oferecidos e vínculos de profissionais. Os arquivos seguem convenções do sistema Oracle de onde são extraídos, com delimitador ponto-e-vírgula, *encoding* Latin-1 e nomes de coluna no padrão Oracle (prefixos como `CO_` para código, `NO_` para nome, `QT_` para quantidade).

Em termos de volume, os números são expressivos: entre as maiores tabelas estão 605.447 estabelecimentos de saúde, 7.614.703 profissionais, 6.507.974 registros de carga horária, 2.883.938 registros de horários de atendimento e 124.273 equipes de saúde; somando essas a outras tabelas da base, o total ultrapassa 22 milhões de registros. Essa escala torna o CNES um domínio realista para avaliar o comportamento de um *pipeline* de dados.

A escolha do CNES como domínio para este trabalho se justifica por três razões principais. Primeiro, trata-se de uma base real, pública e governamental, o que dá legitimidade ao protótipo e permite que outros pesquisadores reproduzam os experimentos com os mesmos dados. Segundo, o domínio de saúde pública tem relevância social direta: um *pipeline* capaz de manter dados do CNES atualizados em *near real-time* teria aplicações concretas em gestão hospitalar, planejamento de cobertura assistencial e monitoramento de equipamentos médicos. Terceiro, a estrutura relacional da base é rica e complexa, com chaves compostas, relacionamentos N:N, múltiplas dimensões e códigos que precisam de decodificação, o que exercita o *pipeline* de forma muito mais representativa do que um modelo fictício de e-commerce.

No protótipo, foi selecionado um subconjunto representativo da base para compor o modelo transacional: sete tabelas transacionais com dados de estabelecimentos, profissionais, vínculos, equipes e serviços, apoiadas por dezesseis tabelas dimensionais de referência (estados, municípios, tipos de estabelecimento, tipos de equipe, ocupações profissionais, entre outras). O banco é inicializado com um *seed* de dados que inclui 10 estabelecimentos reais em capitais brasileiras, 20 profissionais com nomes realistas, 10 equipes (majoritariamente ESF, Estratégia Saúde da Família), e registros associados de carga horária, equipamentos e serviços.

3 Trabalhos Relacionados

Nesta seção são apresentados os trabalhos identificados na literatura que possuem maior relação com o tema deste TCC. A análise foca em como cada trabalho aborda os conceitos de *data lakehouse*, CDC, *streaming* e experimentação, identificando lacunas que este trabalho busca preencher.

Harby e Zulkernine (HARBY; ZULKERNINE, 2025) fizeram um *survey* bastante completo sobre *Data Lakehouse*, analisando as plataformas existentes (Delta Lake, Hudi, Iceberg) e avaliando experimentalmente o desempenho em operações de leitura e escrita. O trabalho é valioso por mapear o estado da arte, mas não aborda a integração com CDC nem avalia a latência de ingestão em cenários *near real-time*, que é justamente o foco do presente TCC.

Mazumdar, Hughes e Onofré (MAZUMDAR; HUGHES; ONOFRé, 2023) fazem uma análise conceitual do *lakehouse* como evolução do *data warehousing*, comparando formatos de tabela e capacidades transacionais de Delta Lake, Hudi e Iceberg. Os autores discutem as limitações de cada formato e identificam áreas de convergência. O trabalho não inclui implementação prática, o que torna o presente TCC complementar ao oferecer um protótipo funcional com medições reais.

Armbrust et al. (ARMBRUST et al., 2021) foram os primeiros a formalizar o conceito de *lakehouse* como uma nova geração de plataformas. O artigo descreve os princípios arquiteturais e demonstra como o Delta Lake implementa transações ACID sobre um *data lake*. Embora o trabalho seja fundacional para o conceito, ele não aborda CDC como mecanismo de ingestão e foca na plataforma Databricks, que é comercial.

Armbrust et al. (ARMBRUST et al., 2018) descrevem o Spark Structured Streaming e como ele permite tratar computações de *streaming* como consultas incrementais sobre tabelas que crescem continuamente. Esse trabalho serve de base conceitual para a arquitetura de *Streaming ETL* adotada aqui, ainda que na prática o protótipo utilize PHP. Os conceitos de micro-lote e processamento contínuo são diretamente aplicáveis.

Jain et al. (JAIN et al., 2023) apresentam um estudo comparativo de aborda-

gens de CDC para integração de dados em tempo real, comparando ferramentas como Debezium, Maxwell e Oracle GoldenGate. O trabalho é relevante por contextualizar as vantagens do CDC *log-based* sobre alternativas como *polling* e *triggers*, mas não integra o CDC com uma arquitetura *lakehouse* nem realiza medições de latência ponta a ponta.

A documentação do Debezium (Red Hat, 2026) foi a referência técnica principal para a parte de CDC. Dela foram extraídos os conceitos de *snapshot* inicial, formato do envelope de eventos e configurações como o `decimal.handling.mode`. Embora seja documentação técnica e não um trabalho acadêmico, ela fornece detalhes de implementação indispensáveis.

Kreps, Narkhede e Rao (KREPS; NARKHEDE; RAO, 2011) descrevem a arquitetura original do Apache Kafka, explicando como um sistema de mensagens distribuído baseado em *log* pode servir como barramento central para integração de dados. Os conceitos de tópicos, partições, produtores e consumidores apresentados nesse trabalho são a base para a comunicação entre Debezium e o consumidor PHP no protótipo.

A Tabela 3.1 faz um comparativo entre os trabalhos citados e este TCC, destacando quais aspectos cada um aborda.

Tabela 3.1: Comparação entre trabalhos relacionados e este TCC.

Trabalho	**Lakehouse**	**CDC**	**Streaming**	**Experimentos**
Harby e Zulk. (2025)	Sim	Não	Não	Sim (leit./escr.)
Mazumdar et al. (2023)	Sim	Não	Não	Não
Armbrust et al. (2021)	Sim	Não	Não	Parcial
Armbrust et al. (2018)	Não	Não	Sim (Spark)	Sim
Jain et al. (2023)	Não	Sim	Parcial	Comparativo
Kreps et al. (2011)	Não	Não	Sim (Kafka)	Sim
Debezium Docs	Não	Sim	Não	Não
Este TCC	**Sim**	**Sim**	**Sim (PHP)**	**Sim (latência)**

O diferencial deste trabalho está em integrar CDC, *streaming* e organização em camadas *lakehouse* num único protótipo que funciona de ponta a ponta, com medições de latência em diferentes cenários de carga. Nenhum dos trabalhos identificados combina todos esses elementos simultaneamente: Harby e Zulkernine focam no *lakehouse* sem CDC, Jain et al. analisam CDC sem *lakehouse*, Armbrust et al. (2018) tratam *streaming* sem CDC, e Mazumdar et al. oferecem análise conceitual sem implementação prática.

Além disso, nenhum dos trabalhos identificados utiliza dados reais do domínio de saúde pública. O uso de dados do CNES/DATASUS adiciona uma dimensão de aplicabilidade prática e relevância social que os demais trabalhos, baseados em *benchmarks* genéricos ou plataformas comerciais, não contemplam.

Cabe ainda mencionar que a maioria dos trabalhos analisados utiliza ferramentas de *streaming* industriais como Flink e Spark, o que é adequado para avaliações de desempenho mas pode dificultar a reprodução em ambientes acadêmicos com recursos limitados. A escolha do PHP neste trabalho, embora traga limitações de desempenho documentadas, torna o protótipo mais acessível e permite que os limites do *pipeline* apareçam de forma clara e mensurável.

4 Arquitetura Proposta

Esta seção descreve como o *pipeline* de atualização *near real-time* foi projetado, incluindo seus componentes, modelo de dados, fluxo de processamento e justificativas técnicas.

4.1 Visão geral

A arquitetura é composta por cinco estágios:

1. **Origem transacional**: um banco MySQL que armazena dados do CNES, com tabelas de estabelecimentos, profissionais, carga horária, equipes, vínculos equipe-profissional, equipamentos e serviços.
2. **CDC**: o Debezium monitora o *binlog* do MySQL e publica as alterações em tópicos do Kafka.
3. **Barramento de eventos**: o Kafka armazena os eventos e os disponibiliza para consumo.
4. **Streaming ETL**: um consumidor PHP lê os eventos continuamente, transforma os dados e grava nas camadas bronze e silver.
5. **Camada analítica**: as agregações *gold* são construídas a partir dos dados da *silver* e materializadas no PostgreSQL.

A Figura 4.1 sintetiza essa organização. O caminho analítico caracteriza-se por ser estritamente linear e unidirecional. As alterações realizadas no MySQL são capturadas pelo Debezium por meio da leitura do binlog e, em seguida, publicadas nos tópicos do Kafka. O motor de Streaming ETL, desenvolvido em PHP, consome esses dados e os materializa nas camadas bronze, silver e gold do lakehouse, sendo a camada gold replicada no PostgreSQL para fins de consulta.

Neste esquema, as setas sólidas representam o fluxo de dados do pipeline, enquanto as setas tracejadas indicam acessos externos ao fluxo analítico. O princípio central do trabalho é garantir que o único componente com permissão de escrita no MySQL seja

o gerador de carga, responsável por simular a operação cotidiana. De forma análoga, o banco é acessado apenas pelo script de verificação de consistência, cuja função se limita à comparação entre a origem e o destino. É importante notar que nenhum estágio do percurso CDC → Kafka → Streaming ETL → bronze → silver → gold realiza consultas diretamente no banco de dados transacional. Todo o conjunto de informações que compõe a camada analítica trafegou, obrigatoriamente, pelo fluxo de streaming estabelecido.

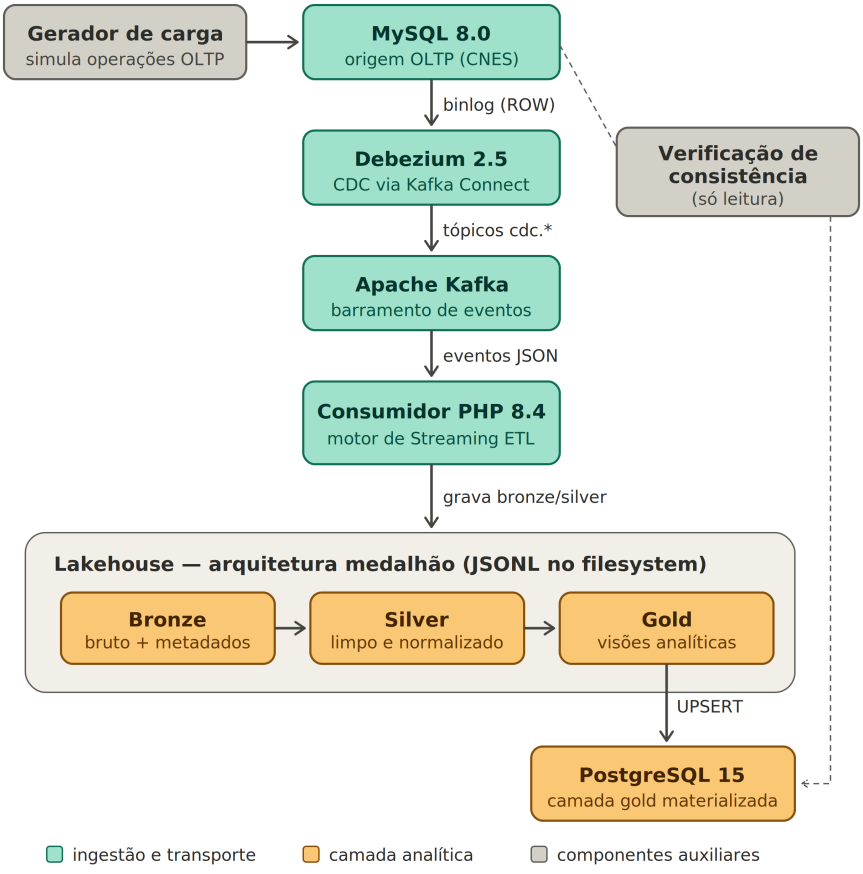

Figura 4.1: Arquitetura do pipeline de atualização *near real-time*. O fluxo analítico (setas sólidas) vai do MySQL ao PostgreSQL, passando por Debezium, Kafka, o consumidor PHP e pelas camadas bronze/silver/gold da arquitetura medalhão. O gerador de carga e a verificação de consistência (em cinza, conexões tracejadas) são os únicos componentes que acessam o MySQL diretamente e estão fora do pipeline analítico.

A Tabela 4.1 lista os componentes e tecnologias utilizados no protótipo.

Tabela 4.1: Componentes da arquitetura.

Componente	**Tecnologia**	**Função**
Banco transacional	MySQL 8.0	Origem OLTP com dados CNES
CDC	Debezium 2.5	Captura via *binlog*
Mensageria	Apache Kafka 7.5.0	Barramento de eventos
Orquestração CDC	Kafka Connect	Gerenciamento do conector
Streaming ETL	PHP 8.4 + rdkafka	Processamento contínuo
Lakehouse	Diretório local (JSONL)	Camadas bronze/silver/gold
Object storage	MinIO (provisionado)	Infraestrutura para evolução futura
Banco analítico	PostgreSQL 15	Camada *gold* materializada
Conteinerização	Docker Compose	Orquestração de todos os serviços

De forma complementar à Figura 4.1, que detalha o fluxo completo com os componentes auxiliares, a Figura 4.2 apresenta uma visão simplificada dos componentes do protótipo, destacando o fluxo linear principal: a cadeia de ingestão e transporte (MySQL, Debezium, Kafka e o consumidor PHP) e as camadas analíticas do *lakehouse* (bronze, silver e gold) até a materialização no PostgreSQL.

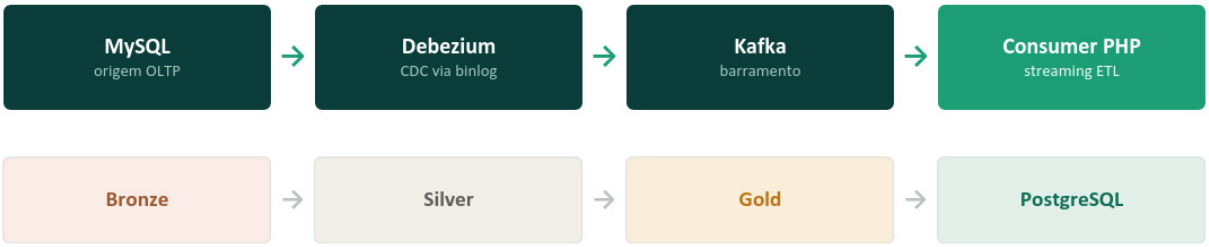

Figura 4.2: Visão simplificada dos componentes do protótipo: o fluxo de ingestão e transporte (MySQL → Debezium → Kafka → consumidor PHP) e as camadas analíticas (bronze → silver → gold) materializadas no PostgreSQL.

4.2 Modelo de dados

O banco transacional armazena dados do CNES organizados em sete tabelas transacionais monitoradas pelo Debezium e dezesseis tabelas dimensionais de referência. As tabelas transacionais são aquelas que sofrem inserções, atualizações e exclusões no dia a dia, é sobre elas que o CDC atua.

A tabela `estabelecimentos` é a entidade central, representando cada unidade de saúde do Brasil (UBSs, hospitais, clínicas, UPAs). Ela possui 24 colunas com dados como

código CNES, razão social, endereço, coordenadas geográficas, tipo de estabelecimento e natureza jurídica. A tabela `profissionais` armazena os dados cadastrais dos profissionais de saúde, com identificadores anonimizados (*hash* hexadecimal) por questões de privacidade. Cabe ressalvar que um *hash* simples, sem *salt*, oferece anonimização limitada: quando o domínio de entrada é restrito (como CPFs ou códigos com formato conhecido), o identificador original pode ser recuperado por ataques de dicionário ou *rainbow tables*. Uma anonimização mais robusta empregaria um *salt* por registro, ou um *HMAC* com chave secreta, o que fica registrado como ponto de melhoria de privacidade.

O vínculo entre profissional e estabelecimento é registrado na tabela `carga_horaria`, que funciona como tabela transacional de relacionamento, registrando a carga horária ambulatorial, hospitalar e outras de cada profissional em cada unidade. As tabelas `equipes` e `equipe_profissionais` registram as equipes de saúde (como ESF — Estratégia Saúde da Família) e sua composição.

Além dessas, as tabelas `estab_equipamentos` e `estab_servicos` registram o inventário de equipamentos e os serviços oferecidos por cada estabelecimento. A Tabela 4.2 resume as tabelas monitoradas pelo CDC.

Tabela 4.2: Tabelas transacionais monitoradas pelo Debezium.

Tabela	**Descrição**	**Cenário CDC**
`estabelecimentos`	Unidades de saúde	INSERTs, UPDATEs
`profissionais`	Profissionais do SUS	UPDATEs, INSERTs
`carga_horaria`	Vínculos prof.–estab.	INSERTs, UPDATEs, DELETEs
`equipes`	Equipes de saúde	INSERTs, UPDATEs
`equipe_profissionais`	Composição das equipes	INSERTs, DELETEs
`estab_equipamentos`	Equipamentos por estab.	INSERTs, UPDATEs
`estab_servicos`	Serviços oferecidos	INSERTs, UPDATEs

As tabelas dimensionais fornecem as descrições textuais para os códigos presentes nas tabelas transacionais. Entre as mais importantes estão: `municipios` (30 municípios representativos), `estados` (27 UFs brasileiras), `tipo_estabelecimento` (20 tipos, como posto de saúde, hospital geral, UPA, CAPS), `tipo_equipe` (15 tipos, incluindo ESF, NASF, eMAESM, consultório na rua), `atividade_profissional` (30 códigos CBO representativos, de médicos generalistas a psicólogos), `equipamentos_catalogo` (20 equipamentos, de raio-X a incubadora), `servico_especializado` (20 serviços, de urgência a

saúde mental) e `classificacao_servico` (20 classificações). Essas tabelas são carregadas via *seed* no `init.sql` e não são monitoradas pelo Debezium, pois representam dados de referência que mudam raramente.

O *seed* inicial dos dados transacionais foi projetado para ser representativo do cenário real. Contém 10 estabelecimentos em capitais brasileiras (incluindo UBSs em São Paulo e Belo Horizonte, um hospital em Brasília, e uma UBS em Juiz de Fora), 20 profissionais com nomes e códigos CBO realistas, 10 equipes (majoritariamente ESF — Estratégia Saúde da Família, que é o principal modelo de atenção primária do SUS), 20 vínculos de carga horária com distribuição variada de horas, 15 registros de equipamento em diferentes estabelecimentos e 15 registros de serviço. Esse volume é suficiente para demonstrar o funcionamento completo do *pipeline* e gerar agregações *gold* significativas, embora seja pequeno comparado à base real do CNES.

Além do *seed* estático, o protótipo inclui um *script* de importação (<import_cnes_csv.sh>) preparado para carregar a base completa do CNES a partir dos CSVs do DATASUS. Esse *script* trata os desafios de importação específicos da base, como a conversão de *encoding* Latin-1 para UTF-8, a renomeação de *headers* no padrão Oracle, e a conversão de datas do formato DD/MM/YYYY para YYYY-MM-DD via `STR_TO_DATE()`. A importação completa não foi utilizada nos experimentos deste trabalho, mas fica disponível como trabalho futuro.

4.3 Fluxo de dados

O fluxo de dados no protótipo pode ser dividido em duas fases com características distintas: o processamento evento a evento (que alimenta as camadas bronze e silver) e o processamento em micro-*batch* (que constrói a camada gold).

4.3.1 Processamento evento a evento

Quando algo acontece no MySQL, por exemplo, um novo vínculo profissional é criado com um INSERT em `carga_horaria`, ou uma equipe de saúde tem sua data de desativação preenchida com um UPDATE em `equipes`, o Debezium detecta a mudança no *binlog* e

publica um evento JSON no tópico Kafka correspondente. Esse evento traz os campos `before`, `after`, `op` e `ts_ms`.

O consumidor PHP, inscrito nos sete tópicos, recebe o evento e executa três ações em sequência: primeiro grava o evento completo (com todos os metadados do Debezium) na camada bronze, depois extrai os dados relevantes, aplica as transformações de limpeza e normalização através do `EventTransformer`, e grava o registro limpo na camada silver via `LakehouseWriter`, por fim, registra os *timestamps* de cada etapa no `MetricsCollector` para cálculo posterior de latência.

4.3.2 Processamento *gold* (micro-*batch*)

Para a camada *gold*, o processamento funciona de forma fundamentalmente diferente. Em vez de processar evento por evento, o `GoldProcessor` opera em micro-*batches*: ele lê todos os arquivos JSONL das sete tabelas na camada *silver*, reconstrói em memória o estado mais atual de cada registro (considerando a ordem temporal via `_processed_at` e tratando exclusões via `_is_deleted`), e a partir desse estado completo gera as três agregações analíticas.

Esse processamento é disparado em dois momentos: a cada 50 eventos processados pelo consumidor (garantindo atualização periódica durante cargas sustentadas), e quando o consumidor fica 5 segundos sem receber nenhum evento novo (período de inatividade).

O processamento *gold* inclui também o *flush* do `MetricsCollector`, que grava as métricas acumuladas no *buffer* para o PostgreSQL. Isso garante que, mesmo para lotes pequenos de eventos (menores que o *buffer* de 50 registros), as métricas de latência sejam persistidas.

As agregações geradas são gravadas em dois destinos simultaneamente: no *filesystem* (como arquivos JSONL na camada gold) e no PostgreSQL (via UPSERT, usando `INSERT ... ON CONFLICT DO UPDATE`). O PostgreSQL serve como camada de consulta final, onde ferramentas de *business intelligence* poderiam se conectar para gerar *dashboards* e relatórios.

4.4 Agregações da camada *gold*

O protótipo produz três agregações que representam visões analíticas distintas dos dados do CNES. Cada uma é materializada tanto no *filesystem* (como arquivo JSONL na camada gold) quanto no PostgreSQL (via UPSERT, usando `INSERT ... ON CONFLICT DO UPDATE`).

O `gold_estab_summary` é uma visão desnormalizada de cada estabelecimento de saúde, reunindo dados cadastrais (razão social, nome fantasia, tipo de estabelecimento, natureza jurídica, município, UF, turno de atendimento) e totais calculados a partir de outras tabelas: quantidade de profissionais vinculados (da `carga_horaria`), quantidade de equipes alocadas (da `equipes`), total de equipamentos existentes (da `estab_equipamentos`) e total de serviços oferecidos (da `estab_servicos`). Essa agregação permite responder perguntas como "quantos profissionais atuam neste hospital?" ou "quais serviços esta UBS oferece?" sem precisar fazer junções manuais entre tabelas.

O `gold_municipal_health` agrega indicadores de saúde por município, oferecendo um panorama da infraestrutura de saúde municipal. Para cada município, calcula: total de estabelecimentos (classificados por tipo: UBS, hospital, UPA, outros), total de profissionais distintos vinculados, quantidade de equipes ESF ativas versus outros tipos de equipe, total de equipamentos disponíveis para o SUS, e somatório de horas ambulatoriais e hospitalares. A classificação dos estabelecimentos por tipo usa os códigos do CNES (por exemplo, códigos 001 e 002 para postos e centros de saúde, 005 e 007 para hospitais gerais e especializados). Essa visão seria diretamente útil para gestores municipais de saúde que precisam de um retrato rápido da infraestrutura disponível.

O `gold_workforce` cruza dados de profissionais com seus vínculos de carga horária e participação em equipes, produzindo um perfil consolidado de cada profissional de saúde. Para cada profissional, registra: nome, código CBO (Classificação Brasileira de Ocupações), conselho de classe (CRM, COREN, etc.), número total de vínculos, horas ambulatoriais, hospitalares e outras, quantidade de estabelecimentos em que atua e quantidade de equipes de que participa. Essa agregação permite identificar, por exemplo, profissionais com múltiplos vínculos ou com carga horária total acima do permitido, informação relevante para fiscalização e planejamento de recursos humanos em saúde.

4.5 Sobre o uso do termo *lakehouse*

Antes de prosseguir, é importante fazer uma distinção que afeta a interpretação de todo o trabalho. O protótipo desenvolvido não implementa um *lakehouse* completo no sentido estrito, conforme definido por Armbrust et al. (2021). Não são utilizados formatos de tabela transacionais como Delta Lake ou Iceberg, os dados são armazenados em arquivos JSONL, sem transações ACID sobre os arquivos e sem recursos como *time travel*.

O que o protótipo implementa é o padrão arquitetural associado ao *lakehouse*: organização em camadas bronze/silver/gold, ingestão contínua via CDC e *streaming*, e separação entre armazenamento e processamento. A adoção de formatos transacionais seria o próximo passo natural, mas essas tecnologias não possuem bibliotecas para PHP, seria necessário um componente em Python ou Java.

4.6 Por que PHP?

A escolha do PHP como motor de *Streaming ETL* pode parecer incomum, mas faz sentido no contexto deste trabalho. O objetivo não é avaliar uma ferramenta de *streaming* específica, e sim demonstrar que o fluxo completo, do CDC até o *lakehouse*, funciona e medir sua latência. O PHP, por ser uma linguagem de amplo conhecimento e fácil de rodar, torna o protótipo mais acessível e reproduzível.

A extensão `php-rdkafka` fornece uma interface nativa para comunicação com o Kafka, sem necessidade de ferramentas intermediárias. O PHP suporta conexões PDO com MySQL e PostgreSQL, o que simplifica tanto a leitura de configurações quanto a escrita das agregações *gold*. Naturalmente, para um cenário de produção com alta carga, seria necessário migrar para algo como Flink ou Spark e essa é uma das principais limitações documentadas do protótipo.

5 Implementação

Esta seção descreve como o protótipo foi implementado na prática, desde a configuração do ambiente até os detalhes das classes PHP que processam os eventos. Todo o código-fonte, os arquivos de configuração e as instruções de execução estão disponíveis publicamente no repositório de código aberto github.com/Jo1oPedro/TCC.

5.1 Ambiente Docker

Todo o ambiente roda em Docker Compose, com nove serviços: Zookeeper (coordenação do Kafka), Kafka (barramento de eventos), Kafka Connect com Debezium (CDC), Kafka UI (para monitoramento visual dos tópicos), MySQL (origem transacional), PostgreSQL (destino analítico), MinIO (*object storage* provisionado para evolução futura, atualmente não utilizado em escrita), o inicializador de *buckets* do MinIO e a aplicação PHP.

O MySQL foi configurado com *binlog* no formato ROW e GTID ativo, que são pré-requisitos para o Debezium funcionar. O Listing 5.1 mostra as configurações relevantes.

```
--server-id=1
--log-bin=mysql-bin
--binlog-format=ROW
--binlog-row-image=FULL
--gtid-mode=ON
--enforce-gtid-consistency=ON
```

Listing 5.1: Configuração do MySQL para CDC.

Os esquemas dos bancos são criados automaticamente na primeira vez que os contêineres sobem. Tanto o MySQL quanto o PostgreSQL usam o mecanismo `/docker-entrypoint-initdb.d/`, onde scripts SQL colocados nesse diretório são executados na inicialização. Para forçar uma reinicialização limpa, basta derrubar os contêineres com `docker compose down -v`, que remove os volumes persistentes.

O Listing 5.2 mostra um trecho do `docker-compose.yaml` com a configuração

dos serviços principais.

```
mysql:
  image: mysql:8.0
  container_name: mysql
  environment:
    MYSQL_ROOT_PASSWORD: root123
    MYSQL_DATABASE: cnes_db
    MYSQL_USER: appuser
    MYSQL_PASSWORD: appuser123
  command: >
    --server-id=1
    --log-bin=mysql-bin
    --binlog-format=ROW
    --binlog-row-image=FULL
    --gtid-mode=ON
    --enforce-gtid-consistency=ON
  volumes:
    - ./sql/init.sql:/docker-entrypoint-initdb.d/init.sql
    - mysql_data:/var/lib/mysql
  healthcheck:
    test: mysqladmin ping -h localhost -u root -proot123
    interval: 10s
    timeout: 5s
    retries: 10
kafka-connect:
  image: debezium/connect:2.5
  container_name: kafka-connect
  depends_on:
    kafka:
      condition: service_healthy
    mysql:
      condition: service_healthy
  environment:
    BOOTSTRAP_SERVERS: kafka:29092
    GROUP_ID: connect-cluster
    KEY_CONVERTER_SCHEMAS_ENABLE: "false"
```

```
36      VALUE_CONVERTER_SCHEMAS_ENABLE: "false"
```

Listing 5.2: Trecho do docker-compose.yaml — serviços MySQL e Kafka Connect.

O MySQL é configurado com `command` para ativar o *binlog* no formato ROW, necessário para o Debezium. O Kafka Connect utiliza a imagem oficial do Debezium (`debezium/connect:2.5`), que já inclui os conectores MySQL pré-instalados. A dependência com `condition: service_healthy` garante que o MySQL e o Kafka estejam saudáveis antes do Kafka Connect iniciar, evitando falhas de conexão durante a inicialização.

Uma questão prática que surgiu durante o desenvolvimento foi a permissão de arquivos. Os arquivos do *lakehouse* (JSONL em `lakehouse-data/`) são criados pelo contêiner PHP, mas precisam ser legíveis no *host*. Para resolver, o serviço `php-app` foi configurado com `user: "${UID:-1000}:${GID:-1000}"`, garantindo que os arquivos criados tenham as permissões corretas.

5.2 Configuração do CDC

O Debezium foi configurado para monitorar as sete tabelas transacionais do banco `cnes_db`, gerando tópicos no formato `cdc.cnes_db.{tabela}`. Duas configurações merecem destaque:

- O `decimal.handling.mode` foi definido como `string`. Sem isso, campos do tipo `DECIMAL` chegam como bytes codificados em Base64, o que dificulta o processamento.
- Os conversores JSON foram configurados com `schemas.enable: false`, para simplificar a estrutura dos eventos e facilitar o *parsing* no PHP.

O Listing 5.3 mostra a definição da tabela principal do modelo `estabelecimentos` no MySQL.

```
1 CREATE TABLE estabelecimentos (
2     co_unidade VARCHAR(20) PRIMARY KEY,
3     co_cnes VARCHAR(7) NOT NULL,
4     no_razao_social VARCHAR(200),
```

```
    no_fantasia VARCHAR(200),
    no_logradouro VARCHAR(200),
    nu_endereco VARCHAR(10),
    no_bairro VARCHAR(100),
    co_cep VARCHAR(8),
    nu_telefone VARCHAR(20),
    no_email VARCHAR(100),
    co_turno_atendimento VARCHAR(5),
    co_estado_gestor VARCHAR(2),
    co_municipio_gestor VARCHAR(6),
    nu_latitude DECIMAL(12,8),
    nu_longitude DECIMAL(12,8),
    co_natureza_jur VARCHAR(10),
    co_tipo_unidade VARCHAR(5),
    tp_gestao VARCHAR(5),
    co_tipo_estabelecimento VARCHAR(5),
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
        ON UPDATE CURRENT_TIMESTAMP,
    INDEX idx_cnes (co_cnes),
    INDEX idx_municipio (co_municipio_gestor),
    INDEX idx_tipo_estab (co_tipo_estabelecimento)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;
```

Listing 5.3: Definição da tabela estabelecimentos no MySQL.

Alguns detalhes dessa definição merecem comentário. A chave primária <co_unidade> é um código alfanumérico que combina o código IBGE do município com um sequencial, seguindo o padrão do CNES real. Os campos `created_at` e `updated_at` são gerenciados automaticamente pelo MySQL: o `ON UPDATE CURRENT_TIMESTAMP` faz o `updated_at` ser atualizado a cada modificação no registro, o que permite ao Debezium capturar a data exata da alteração. Os índices em `co_cnes`, `co_municipio_gestor` e `co_tipo_estabelecimento` otimizam as consultas do gerador de carga, que precisa buscar registros aleatórios para atualização.

O Listing 5.4 mostra a definição da tabela `gold_municipal_health` no Post-

greSQL, que recebe as agregações municipais.

```
CREATE TABLE gold_municipal_health (
    co_municipio VARCHAR(6) PRIMARY KEY,
    nome_municipio VARCHAR(100),
    uf VARCHAR(2),
    total_estabelecimentos INT DEFAULT 0,
    total_profissionais INT DEFAULT 0,
    total_equipes_esf INT DEFAULT 0,
    total_equipes_outras INT DEFAULT 0,
    total_equipamentos_sus INT DEFAULT 0,
    qtd_ubs INT DEFAULT 0,
    qtd_hospitais INT DEFAULT 0,
    qtd_upas INT DEFAULT 0,
    qtd_outros INT DEFAULT 0,
    horas_ambulatoriais_total BIGINT DEFAULT 0,
    horas_hospitalares_total BIGINT DEFAULT 0,
    dt_processamento TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

Listing 5.4: Tabela de agregação gold_municipal_health no PostgreSQL.

O uso de `PRIMARY KEY` em `co_municipio` permite que o `GoldProcessor` utilize UPSERT (`INSERT ... ON CONFLICT DO UPDATE`) para atualizar os indicadores a cada processamento *gold* sem gerar duplicatas. O campo `dt_processamento` registra quando os dados foram atualizados pela última vez, permitindo verificar a "frescura" das agregações.

Além disso, foi configurado o `time.precision.mode` como `connect` para compatibilidade de *timestamps*, e o `database.connectionProperties` com `useUnicode=true; characterEncoding=UTF-8` para evitar problemas de *encoding* com acentos nos nomes de estabelecimentos e profissionais.

5.3 O consumidor PHP

O consumidor foi desenvolvido em PHP 8.4, usando a extensão `php-rdkafka` para a comunicação com o Kafka. Um detalhe importante: o consumidor não tem nenhuma

conexão com o MySQL. Ele se conecta apenas ao Kafka (para ler eventos) e ao PostgreSQL (para gravar métricas e agregações *gold*).

O código é organizado em cinco classes principais, cada uma com uma responsabilidade bem definida.

5.3.1 EventTransformer

A classe `EventTransformer` cuida da transformação dos eventos brutos do Debezium em registros limpos para a camada *silver*. Cada tabela tem uma função de transformação específica que ajusta tipos, aplica `trim()` em campos de texto, converte *timestamps* (o Debezium pode enviar em milissegundos desde a *epoch* ou como *string*), e padroniza campos.

O Listing 5.5 mostra a estrutura geral da transformação e um exemplo para a tabela de estabelecimentos.

```
public function transform(string $table, array $event): ?array
{
    $operation = self::OP_MAP[$event['op'] ?? ''] ?? 'UNKNOWN';

    // Para DELETE usa 'before'; para o resto, 'after'
    $data = ($operation === 'DELETE')
        ? ($event['before'] ?? null)
        : ($event['after'] ?? null);

    if ($data === null) return null;

    $cleaned = match ($table) {
        'estabelecimentos'     => $this->transformEstabelecimento($data),
        'profissionais'        => $this->transformProfissional($data),
        'carga_horaria'        => $this->transformCargaHoraria($data),
        'equipes'              => $this->transformEquipe($data),
        'equipe_profissionais' => $this->transformEquipeProfissional($data),
        'estab_equipamentos'   => $this->transformEstabEquipamento($data
```

```
    ↪ ),
19          'estab_servicos'        => $this->transformEstabServico($data),
20          default                 => $data,
21      };
22
23      // Metadados de processamento
24      $cleaned['_operation'] = $operation;
25      $cleaned['_source_table'] = $table;
26      $cleaned['_source_ts_ms'] = $event['ts_ms'] ?? null;
27      $cleaned['_is_deleted'] = ($operation === 'DELETE');
28
29      return $cleaned;
30  }
```

Listing 5.5: Método de transformação do EventTransformer.

A linha 4 identifica o tipo de operação (INSERT, UPDATE, DELETE, SNAPSHOT). As linhas 7–9 determinam de qual campo do evento extrair os dados: para exclusões, o estado anterior (`before`), para os demais casos, o estado posterior (`after`). O `match` da linha 13 direciona para a função de transformação específica de cada tabela. Ao final, os metadados de processamento são adicionados incluindo a *flag* `_is_deleted`, que é usada na camada *gold* para tratar registros excluídos.

5.3.2 LakehouseWriter

A classe `LakehouseWriter` cuida da escrita nas camadas bronze, silver e gold do *lakehouse*. Ela cria a estrutura de diretórios particionados por data (formato `YYYY-MM-DD`) e gera arquivos JSONL (*JSON Lines*), onde cada linha é um objeto JSON independente.

O formato JSONL foi escolhido por ser *append-friendly*: a adição de um novo registro consiste em abrir o arquivo em modo `FILE_APPEND` e escrever uma linha, sem precisar ler ou reescrever o conteúdo existente. Formatos colunares como Parquet seriam mais eficientes para leitura, mas exigiriam reescrever o arquivo inteiro a cada nova gravação. Cada registro recebe um campo `_processed_at` com o *timestamp* do momento da escrita, usado posteriormente para determinar a versão mais recente de cada registro.

5.3.3 GoldProcessor

O `GoldProcessor` é a classe responsável por construir as três agregações *gold*. Seu funcionamento merece uma explicação detalhada porque é onde se materializa o princípio de que a camada analítica não consulta o MySQL.

O método `loadLatestState()` lê todos os arquivos JSONL da camada *silver* para cada tabela e monta um mapa em memória onde a chave é o identificador do registro (como `co_unidade` para estabelecimentos) e o valor é o registro mais recente, usando o campo `_processed_at` para determinar a ordem. Se um registro foi marcado como deletado (`_is_deleted = true`), ele é removido do mapa. O Listing 5.6 mostra o trecho central desse método.

```
1  private function loadLatestState(string $table, string $keyField = 'id')
       ↪ : array
2  {
3      $dir = "{$this->silverPath}/{$table}";
4      if (!is_dir($dir)) return [];
5
6      $records = [];
7      $iterator = new \RecursiveIteratorIterator(
8          new \RecursiveDirectoryIterator($dir, \
       ↪ RecursiveDirectoryIterator::SKIP_DOTS)
9      );
10
11     foreach ($iterator as $file) {
12         if (!$file->isFile() || $file->getExtension() !== 'jsonl')
       ↪ continue;
13
14         $lines = file($file->getPathname(), FILE_IGNORE_NEW_LINES |
       ↪ FILE_SKIP_EMPTY_LINES);
15         if ($lines === false) continue;
16
17         foreach ($lines as $line) {
18             $record = json_decode($line, true);
19             if ($record === null || !isset($record[$keyField])) continue
       ↪ ;
```

```
20
21              $key = $record[$keyField];
22
23              if (!empty($record['_is_deleted'])) {
24                  unset($records[$key]);
25                  continue;
26              }
27
28              if (isset($records[$key])) {
29                  $existingTs = $records[$key]['_processed_at'] ?? '';
30                  $newTs = $record['_processed_at'] ?? '';
31                  if ($newTs <= $existingTs) continue;
32              }
33
34              $records[$key] = $record;
35          }
36      }
37      return $records;
38  }
```

Listing 5.6: Reconstrução de estado a partir dos arquivos silver.

As linhas 7–9 criam um iterador recursivo que percorre todos os subdiretórios e arquivos JSONL da tabela. As linhas 23–25 tratam registros deletados, removendo-os do mapa. As linhas 28–31 garantem que, quando existem múltiplas versões de um mesmo registro, apenas a mais recente é mantida. O resultado é uma fotografia do estado atual de cada tabela, reconstruída inteiramente a partir dos dados que passaram pelo *pipeline*.

Essa abordagem é simples e funciona bem para o volume do protótipo, mas tem uma limitação clara: para grandes volumes de dados, carregar todos os arquivos *silver* em memória se torna inviável. Uma alternativa seria manter um estado incremental persistente, mas isso exigiria um gerenciamento de estado com *checkpointing* que o PHP não oferece nativamente.

5.3.4 MetricsCollector

O `MetricsCollector` registra os *timestamps* de cada etapa do processamento e calcula as latências. Para cada evento, são extraídos do envelope do Debezium três *timestamps* de origem, `source.ts_ms` (*commit* no MySQL, lido do *binlog*), `ts_ms` (instante em que o conector processou o evento) e `ts_kafka_connect` (publicação no Kafka, quando disponível), e registrados os instantes de gravação no *bronze*, no *silver* e, quando aplicável, no *gold*. A partir desses valores são calculadas a latência de captura (`cdc`, do MySQL à *bronze*), suas sub-componentes, a latência de ETL (*bronze* a *silver*) e a latência total.

Os registros são acumulados num *buffer* de 50 e gravados no PostgreSQL via *flush*. O *flush* também acontece nos períodos de inatividade do consumidor (após 5 segundos sem novos eventos), para evitar que métricas de lotes pequenos fiquem presas na memória.

O Listing 5.7 mostra o método de registro de métricas, que captura os *timestamps* de cada etapa e calcula a decomposição da latência.

```
1  public function record(
2      string $table, string $operation, string|int|null $recordId,
3      ?int $tsSource,    // source.ts_ms -- commit no MySQL (binlog)
4      ?int $tsDbz,       // ts_ms -- processamento interno do Debezium
5      ?int $tsKc,        // ts_kafka_connect -- publicacao no Kafka (SMT)
6      string $tsBronze, ?string $tsSilver = null, ?string $tsGold = null,
   ↪ ): void {
7      if (!$this->enabled) return;
8
9      $nowMs = (int)(microtime(true) * 1000);
10     $bronzeMs = self::isoToMillis($tsBronze);
11     $silverMs = $tsSilver !== null ? self::isoToMillis($tsSilver):null;
12
13     // Decomposicao da latencia
14     $latencyDbz     = ($tsSource && $tsDbz) ? $tsDbz - $tsSource : null;
15     $latencyPublish = ($tsDbz && $tsKc) ? $tsKc - $tsDbz : null;
16     $latencyKafka   = ($tsKc && $bronzeMs) ? $bronzeMs - $tsKc : null;
17     $latencyCdc     = ($tsSource && $bronzeMs) ? $bronzeMs - $tsSource :
   ↪  null;
18     $latencyEtl     = ($bronzeMs && $silverMs) ? $silverMs - $bronzeMs :
```

```
    ↪  null;
19      $latencyTotal   = $tsSource ? $nowMs - $tsSource : null;
20
21      $this->buffer[] = [ /* ... campos + 6 latencias ... */ ];
22
23      if (count($this->buffer) >= $this->bufferLimit) {
24          $this->flush();
25      }
26  }
```

Listing 5.7: Método de registro de métricas no MetricsCollector, com decomposição da latência CDC.

A latência de captura `latency_cdc_ms` (linha 18) mede o intervalo entre o *commit* no MySQL e a primeira persistência na camada *bronze*, agregando: leitura do *binlog* pelo Debezium, processamento interno do conector, publicação no Kafka e consumo pelo PHP com escrita na *bronze*. Esse intervalo é decomposto em três sub-latências (linhas 15–17): `latency_dbz_ms` (*commit* → processamento do conector), `latency_publish_ms` (conector → publicação no Kafka) e `latency_kafka_ms` (Kafka → escrita *bronze*). A `latency_etl_ms` (linha 19) mede a transformação *bronze* → *silver*, e a `latency_total_ms` (linha 20) mede do *commit* no MySQL até o registro na *silver*, não inclui o processamento *gold*, assíncrono e em micro-*batch*. A decomposição aproveita o fato de o próprio envelope do Debezium já carregar os *timestamps* `source.ts_ms` e `ts_ms`, sem instrumentação adicional. A análise empírica dessa decomposição é feita na Seção 6.7.1.

O *buffer* acumula os registros de métricas e faz *flush* para o PostgreSQL quando atinge 50 registros, reduzindo o número de operações de I/O. Conforme descrito anteriormente, o *flush* também é disparado no período de inatividade do consumidor.

5.3.5 Loop principal do consumidor

O Listing 5.8 mostra o loop principal do consumidor, que integra todos os componentes.

```
1  while (true) {
2      $message = $consumer->consume(5000); // timeout 5 segundos
3      if ($message === null) continue;
```

```
4
5       switch ($message->err) {
6           case RD_KAFKA_RESP_ERR_NO_ERROR:
7               processEvent($message, $transformer, $writer, $metrics,
    ↪ $stats, $verbose);
8
9               if ($stats['total'] > 0 && $stats['total'] % $goldInterval
    ↪ === 0) {
10                  $results = $goldProcessor->processAll();
11              }
12              break;
13
14          case RD_KAFKA_RESP__TIMED_OUT:
15              // Nenhum evento novo: flush das metricas e dispara gold
16              if ($stats['total'] > 0) {
17                  if (!isset($stats["last_gold"]) || $stats["total"] >
    ↪ $stats["last_gold"]) {
18                      $metrics->flush();      // flush ANTES do gold
19                      $results = $goldProcessor->processAll();
20                      $stats['last_gold'] = $stats['total'];
21                  }
22              }
23              break;
24      }
25  }
```

Listing 5.8: Loop principal do consumidor Kafka.

A linha 2 consome do Kafka com *timeout* de 5 segundos. Se um evento é recebido com sucesso (linha 6), ele é processado e, a cada 50 eventos, a camada *gold* é atualizada. Se o *timeout* é atingido sem novos eventos (linha 14), as métricas acumuladas são persistidas e o processamento *gold* é disparado. A ordem importa: o *flush* de métricas é executado **antes** do processamento *gold*, de modo que as métricas de latência sejam persistidas mesmo que a materialização *gold* falhe, decisão tomada após observar que erros pontuais no *gold* impediam a gravação das métricas no lote correspondente.

5.4 Gerador de carga

Para simular um sistema em operação, foi criado um gerador de carga que executa operações no MySQL. Ele suporta oito cenários com diferentes intensidades, ordenados por taxa nominal crescente (de 1 a 100 operações por segundo). Os quatro primeiros têm semântica associada a situações realistas do dia a dia do CNES, os quatro cenários intermediários (*Rate25* a *Rate80*) foram introduzidos para densificar a região de transição entre o regime sustentável e o regime saturado, com o objetivo de traçar a curva de latência em função da carga (Seção 6.6):

- **Low**: 1 operação/segundo — cadastro de novos estabelecimentos e atualizações de endereço.
- **Mixed**: 5 operações/segundo — mix de operações em todas as tabelas, incluindo atualizações e transferências.
- **Moderate**: 10 operações/segundo — novos vínculos de carga horária e gestão de equipes (criação de equipes e alocação de profissionais existentes a equipes).
- **Rate25, Rate40, Rate60, Rate80**: 25, 40, 60 e 80 operações/segundo — intensidades intermediárias com perfil de operações semelhante ao *Moderate*, variando gradualmente a proporção de inserções de estabelecimentos e vínculos para sondar o ponto de saturação do consumidor.
- **Burst**: 100 operações/segundo — rajada simulando uma atualização mensal em lote do CNES.

As operações incluem inserção de novos estabelecimentos com dados gerados a partir de padrões reais (municípios brasileiros, tipos de estabelecimento do CNES, nomes fantasia típicos), criação e atualização de vínculos profissional-estabelecimento, e gestão de equipes de saúde. A distribuição probabilística das operações em cada cenário é configurada na classe `ScenarioConfig`, onde a taxa nominal é controlada pelo intervalo entre operações (`interval_ms` = 1000/taxa).

O Listing 5.9 mostra a operação de inserção de um novo estabelecimento, que ilustra como os dados sintéticos são gerados a partir de padrões reais do CNES.

```
1 public function insertEstabelecimento(): void
```

```php
{
    // Sorteia municipio e tipo de estabelecimento
    $municipio = $this->scenarioConfig->municipios[
        array_rand($this->scenarioConfig->municipios)];
    $coMunicipio = $municipio[0]; // codigo IBGE
    $uf = $municipio[2];          // sigla UF

    // Gera co_unidade: codigo municipio + 7 digitos
    $coUnidade = $coMunicipio . str_pad(
        (string) random_int(0, 9999999), 7, '0', STR_PAD_LEFT);

    // Dados realistas
    $tipos = $this->scenarioConfig->tiposEstabelecimento;
    $nomes = $this->scenarioConfig->nomesFantasia;
    $tipoEstab = $tipos[array_rand($tipos)];
    $nomeFantasia = $nomes[array_rand($nomes)];
    $latitude = round(-3.0 - (mt_rand()/mt_getrandmax()) * 30, 6);
    $longitude = round(-35.0 - (mt_rand()/mt_getrandmax()) * 20, 6);

    $stmt = $this->pdo->prepare("
        INSERT INTO estabelecimentos
            (co_unidade, co_cnes, no_razao_social, ...)
        VALUES (?, ?, ?, ...)");
    $stmt->execute([...]);
}
```

Listing 5.9: Inserção de novo estabelecimento no gerador de carga.

O código de unidade (`co_unidade`) segue o padrão real do CNES: é composto pelo código IBGE do município concatenado com um sequencial numérico. Os nomes fantasia são sorteados de uma lista de nomes típicos como "UBS CENTRO", "HOSPITAL MUNICIPAL" e "UPA 24H". As coordenadas geográficas são geradas dentro da faixa do território brasileiro (latitude entre -3° e -33°, longitude entre -35° e -55°).

A configuração dos cenários é centralizada na classe `ScenarioConfig`, que define para cada cenário o intervalo entre operações e a distribuição probabilística dos tipos de

operação. A Tabela 5.1 mostra essa distribuição.

Tabela 5.1: Distribuição de operações por cenário de carga.

Cenário	**Taxa**	**Intervalo**	**Ins. estab.**	**Upd. estab.**	**Vínculo**	**Equipe**
Low	1 op/s	1.000 ms	40%	30%	30%	–
Mixed	5 op/s	200 ms	10%	20%	45%	25%
Moderate	10 op/s	100 ms	15%	15%	50%	20%
Rate25	25 op/s	40 ms	15%	15%	50%	20%
Rate40	40 op/s	25 ms	15%	15%	50%	20%
Rate60	60 op/s	17 ms	20%	10%	50%	20%
Rate80	80 op/s	13 ms	25%	10%	50%	15%
Burst	100 op/s	10 ms	25%	10%	50%	15%

A coluna **Vínculo** agrega as operações de inserção e atualização de carga horária (vínculo profissional–estabelecimento), e a coluna **Equipe** corresponde à gestão de equipes de saúde (criação, ativação e desativação). Os percentuais definem a distribuição probabilística com que cada tipo de operação é sorteado a cada *tick* do gerador, o intervalo entre *ticks* é o que define a taxa nominal.

O cenário Burst, com intervalo de 10 ms entre operações, gera uma taxa nominal de 100 operações por segundo, muito acima da capacidade do consumidor PHP *single-threaded*. Isso é intencional: o objetivo é observar o comportamento do *pipeline* quando saturado e medir a degradação progressiva ao longo das repetições. Os cenários *Rate25* a *Rate80* preenchem a faixa entre o regime sustentável (até $\sim$10 op/s) e a rajada, permitindo localizar o ponto exato em que a latência começa a degradar (Seção 6.6).

5.5 Estrutura do *lakehouse* no *filesystem*

Os dados do *lakehouse* são organizados em três diretórios: `bronze/`, `silver/` e `gold/`, dentro do volume compartilhado `lakehouse-data/`. Cada camada é subdividida por tabela e depois por data:

```
lakehouse-data/
  bronze/
    estabelecimentos/2026-04-11/events.jsonl
    profissionais/2026-04-11/events.jsonl
    carga_horaria/2026-04-11/events.jsonl
```

```
    ...
  silver/
    estabelecimentos/2026-04-11/events.jsonl
    ...
  gold/
    estab_summary/events.jsonl
    municipal_health/events.jsonl
    workforce/events.jsonl
```

Listing 5.10: Estrutura de diretórios do lakehouse.

5.6 Problemas encontrados durante o desenvolvimento

O desenvolvimento do protótipo envolveu a integração de múltiplas tecnologias (MySQL, Debezium, Kafka, PHP, PostgreSQL) em um ambiente conteinerizado, o que naturalmente gerou problemas técnicos que merecem ser documentados. Essa documentação tem valor tanto para reprodutibilidade quanto para orientar futuros trabalhos que adotem arquitetura semelhante.

5.6.1 DECIMAL serializado como Base64

O primeiro problema significativo foi a serialização de campos `DECIMAL` pelo Debezium. O campo `nu_latitude` da tabela `estabelecimentos`, por exemplo, que deveria chegar como "-23.510500", chegava como "BVcm", uma representação em Base64 dos bytes do valor decimal. O problema não gerava nenhum erro, o valor simplesmente era gravado incorretamente na camada silver. A solução foi configurar `decimal.handling.mode=string` no conector Debezium, o que faz os valores decimais serem enviados como *strings* legíveis.

5.6.2 Zookeeper incompatível

A imagem `confluentinc/cp-zookeeper:7.5.0`, que seria a escolha natural por pertencer à mesma família do Kafka utilizado (`confluentinc/cp-kafka:7.5.0`), falhava sistema-

ticamente no *healthcheck* sem mensagem de erro clara. Após investigação, a solução foi substituí-la pela imagem `wurstmeister/zookeeper`, que funcionou sem problemas. Esse tipo de incompatibilidade entre versões de imagens Docker é comum em pilhas de *streaming* e pode consumir tempo considerável de depuração.

5.6.3 Permissões de arquivos em volumes Docker

Os arquivos JSONL criados pelo contêiner PHP no volume `lakehouse-data/` eram criados com permissões de *root*, impedindo sua leitura e exclusão no *host*. A solução foi configurar o serviço `php-app` com `user: "${UID:-1000}:${GID:-1000}"`, que faz o contêiner executar com o UID/GID do usuário do *host*.

5.6.4 GoldProcessor consultando MySQL diretamente

Este foi o problema conceitual mais importante encontrado durante o desenvolvimento. Na versão inicial do código, a classe `GoldProcessor` consultava diretamente o MySQL para gerar as agregações *gold*, por exemplo, contava profissionais por estabelecimento com um `SELECT COUNT(*) FROM carga_horaria GROUP BY co_unidade`. Embora funcionasse do ponto de vista técnico, essa abordagem invalidava completamente o propósito do CDC e do *Streaming ETL*: se a camada analítica simplesmente consulta o banco de origem, para que servem o Debezium e o Kafka?

A correção consistiu em reescrever o `GoldProcessor` para ler exclusivamente dos arquivos JSONL da camada silver, reconstruindo o estado de cada tabela em memória a partir dos eventos que passaram pelo *pipeline*. Essa correção aumentou a complexidade do código (o método `loadLatestState()` e a lógica de deduplicação por `_processed_at`) mas garantiu a integridade conceitual do trabalho.

5.6.5 *Encoding* de caracteres

Em tempo de execução, entre Debezium, Kafka e o consumidor PHP. Mesmo com os dados já em UTF-8 no MySQL, observou-se inicialmente corrupção de acentos no caminho até a camada *silver* ("Acessórios" virando "AcessÃ³rios", por exemplo). A solução envolveu duas configurações: no Debezium, a propriedade `database.connectionProperties` com `use`

`Unicode=true;characterEncoding=UTF-8` e no MySQL, o *charset* padrão das tabelas definido como `utf8mb4`.

5.6.6 Perda de precisão em `latency_cdc_ms` e `latency_etl_ms`

Numa primeira rodada de experimentos, observou-se que as latências reportadas no PostgreSQL apresentavam padrão suspeito: P95 e máximos colados em múltiplos de 1.000 ms (por exemplo, 472 ms ±0 entre repetições). A causa era uma falha sutil no `MetricsCollector`: o *timestamp* do bronze é gravado em ISO-8601 com microssegundos (`Y-m-d\TH:i:s.u`), mas a conversão para milissegundos usava `strtotime($tsBronze) * 1000`. A função `strtotime` do PHP descarta microssegundos e retorna apenas segundos inteiros, fazendo com que `latency_cdc_ms` e `latency_etl_ms` fossem calculadas com precisão de segundo, não de milissegundo. A `latency_total_ms`, calculada com `microtime(true)`, não era afetada, o que dificultou a identificação do problema. A correção substituiu `strtotime` por `DateTimeImmutable::createFromFormat` com sufixo de formato `Uv`, que preserva os milissegundos. Após a correção, os percentis passam a apresentar granularidade real (P95 do cenário low caiu de 472 para 367 ms, com variabilidade entre repetições), refletindo o comportamento efetivo do *pipeline*.

5.6.7 *Drift* de *timezone* entre componentes

Os *timestamps* do Debezium chegam em UTC, mas o consumidor PHP usava `date()` com o *timezone* padrão do contêiner, podendo introduzir deslocamentos de horas em ambientes com configuração regional. A solução foi padronizar todo o *pipeline* em UTC, definindo `date_default_timezone_set('UTC')` no `config.php` e substituindo `date()` por `gmdate()` na conversão de *timestamps* no `EventTransformer`. Essa padronização não afeta o cálculo de latências (que sempre operou em milissegundos desde *epoch*), mas evita que os campos `dt_atualizacao`, `ts_mysql` e `ts_bronze` apareçam com valores inconsistentes entre execuções em fusos diferentes.

6 Experimentos e Resultados

6.1 Metodologia

Este trabalho segue uma abordagem de pesquisa aplicada com caráter experimental. A metodologia pode ser dividida em quatro etapas:

Na primeira etapa, foi realizada uma **revisão bibliográfica** dos conceitos fundamentais relacionados ao trabalho: *data warehouse*, *data lake*, *data lakehouse*, CDC, *streaming* ETL e arquitetura medalhão. A revisão incluiu tanto artigos acadêmicos (periódicos e conferências) quanto documentação técnica oficial das ferramentas utilizadas. A busca por trabalhos relacionados priorizou publicações dos últimos 5 anos nas bases Google Scholar, IEEE Xplore e ACM Digital Library, usando termos como "data lakehouse", "change data capture", "streaming ETL" e "medallion architecture".

Na segunda etapa, foi feito o **projeto da arquitetura** do *pipeline*, definindo os componentes, o modelo de dados, o fluxo de processamento e as decisões técnicas. A escolha do CNES como domínio de dados foi feita nesta etapa, após análise da base pública disponibilizada pelo DATASUS. O modelo de dados foi adaptado da estrutura original dos CSVs do CNES para um esquema relacional normalizado no MySQL, com sete tabelas transacionais e dezesseis dimensionais.

Na terceira etapa, foi realizada a **implementação do protótipo**. O desenvolvimento seguiu uma abordagem iterativa: primeiro foi implementado o fluxo básico (Debezium → Kafka → Consumer → Bronze), depois adicionadas as transformações (Silver), e por fim as agregações (Gold). Problemas técnicos encontrados durante o desenvolvimento, documentados no Capítulo 5, foram resolvidos e suas soluções incorporadas ao código.

Na quarta etapa, foram realizados os **experimentos** para avaliação do protótipo. A grade final de cenários é composta por oito intensidades de carga: Low (1 op/s), Mixed (5 op/s), Moderate (10 op/s), Rate25 (25 op/s), Rate40 (40 op/s), Rate60 (60 op/s), Rate80 (80 op/s) e Burst (100 op/s), ordenadas em taxa nominal crescente. Os cenários intermediários (Rate25 a Rate80) foram introduzidos para densificar a região de

transição entre o regime sustentável e o regime saturado, onde se esperava localizar o "joelho" da curva latência × *throughput*. Cada cenário foi executado durante 30 segundos e repetido 5 vezes, totalizando 40 execuções na rodada de 28 de maio de 2026. As métricas coletadas incluem: latência total (intervalo entre o *commit* no MySQL e o registro do evento na camada *silver*), latência de captura/ingestão (intervalo entre o *commit* no MySQL e a primeira persistência na camada *bronze*), suas sub-componentes instrumentadas a partir de *timestamps* embutidos pelo Debezium (processamento interno do conector `latency_dbz_ms`, e publicação Kafka + trânsito + consumo no PHP `latency_cdc_ms` - `latency_dbz_ms`), latência de ETL (intervalo entre as escritas *bronze* e *silver*), *throughput* (eventos processados na janela de 30 s do gerador) e consistência (comparação do estado final entre MySQL e PostgreSQL).

Vale registrar que a latência de captura `latency_cdc_ms` é calculada entre dois relógios distintos, sendo o instante de commit source.ts_ms, proveniente do servidor MySQL/Debezium, e o instante de escrita na bronze, lido pelo relógio do contêiner PHP via microtime(). Como todos os contêineres executam no mesmo host Docker e, portanto, compartilham o relógio do kernel, o descompasso entre essas fontes é desprezível e os valores reportados são confiáveis. As parcelas `latency_dbz_ms` (intra-Debezium) e `latency_etl_ms` (intra-PHP) são, por construção, medidas dentro de um mesmo relógio e ficam imunes a esse efeito. Essa observação torna-se relevante caso o pipeline seja migrado para um ambiente distribuído ou na nuvem, em que os dois relógios deixam de ser sincronizados automaticamente e a medição de `latency_cdc_ms` passa a exigir sincronização via NTP para permanecer válida, podendo, sem essa sincronização, apresentar viés ou até valores negativos.

A avaliação utiliza percentis (P50, P95, P99) além da média e do desvio padrão amostral entre repetições, seguindo as boas práticas de medição de desempenho em sistemas distribuídos descritas por Kleppmann (KLEPPMANN, 2017). A análise por percentis é preferível à média simples porque latências têm distribuição assimétrica (*skewed*): poucos eventos com latência muito alta podem inflacionar a média sem representar o comportamento típico do sistema. As cinco repetições por cenário ($n = 5$, configuração padrão do *script* `run_experiment.php`) permitem reportar, para cada métrica, a média, o desvio

padrão amostral e a faixa [min, máx] entre execuções. Intervalos de confiança formais e testes estatísticos de comparação entre cenários ainda não são construídos, $n = 5$ é suficiente para caracterizar a dispersão, mas não para inferência formal e ficam apontados no Capítulo 7.

6.2 Ambiente de execução

Os testes foram realizados numa máquina com as seguintes especificações:

- **Processador**: Intel Core i7-1360P, 13ª geração (2,20 GHz, 12 núcleos / 16 *threads*)
- **Memória RAM**: 16 GB (15,7 GB utilizáveis)
- **Disco**: SSD NVMe
- **Sistema operacional**: Windows 11 com WSL2 (Ubuntu 24.04 LTS)
- **Docker**: Docker Desktop para Windows com *backend* WSL2

Todos os nove serviços foram executados simultaneamente na mesma máquina, sem nenhuma otimização específica de sistema operacional ou configuração avançada do Kafka.

6.3 Cenários

Foram definidos oito cenários, cada um executado durante 30 segundos e repetido 5 vezes (40 execuções no total). A Tabela 6.1 resume as características de cada um.

Tabela 6.1: Cenários de teste (ordenados por taxa nominal crescente).

Cenário	**Taxa nominal**	**Operações principais**	**Objetivo**
Low	1 op/s	INSERTs/UPDATEs em estab.	*Baseline*
Mixed	5 op/s	Mix de todas as operações	Carga leve variada
Moderate	10 op/s	Vínculos e gestão de equipes	Carga sustentada
Rate25	25 op/s	Mix variado	Pré-saturação
Rate40	40 op/s	Mix variado	Transição (joelho da curva)
Rate60	60 op/s	Predominância de vínculos	Saturação leve
Rate80	80 op/s	Predominância de vínculos	Saturação
Burst	100 op/s	Rajada em todas as tabelas	Limite superior

Cada cenário foi executado durante 30 segundos e repetido 5 vezes, conforme metodologia descrita na Seção 6.1.

Observação sobre o cenário Low: a primeira repetição (R1) pode incluir os eventos de *snapshot* do Debezium, que são a leitura inicial das tabelas. Como esses eventos têm *timestamps* do momento da leitura e não de alterações transacionais, a latência pode ficar artificialmente alta. Caso esse comportamento seja observado, os resultados de R1 serão apresentados separadamente.

6.4 Resultados de latência

Os resultados são apresentados em duas tabelas, separando o regime sustentável do regime saturado por uma diferença de três ordens de magnitude nas latências. A Tabela 6.2 reúne os cinco cenários de menor latência, reportáveis em milissegundos (Low a Rate40, sendo Rate40 já o joelho da curva), a Tabela 6.3 reúne os três cenários saturados (Rate60, Rate80 e Burst), com valores em segundos.

Tabela 6.2: Latência total — cenários em escala de milissegundos, Low a Rate40 (Rate40 = joelho da curva), média ± desvio padrão amostral [min, máx] entre 5 repetições. Execução de 28 de maio de 2026, 30 s por repetição, ambiente limpo com decomposição da latência CDC instrumentada.

Cenário	Média (ms)	P95 (ms)	P99 (ms)	Máximo (ms)
Low	232 ± 30 [188, 261]	455 ± 53 [363, 490]	484 ± 31 [434, 519]	493 ± 24 [462, 527]
Mixed	270 ± 4 [264, 275]	493 ± 6 [486, 498]	511 ± 7 [504, 522]	535 ± 33 [508, 587]
Moderate	278 ± 9 [268, 292]	498 ± 6 [492, 508]	632 ± 141 [511, 838]	798 ± 232 [546, 1.137]
Rate25	361 ± 31 [319, 402]	843 ± 134 [646, 973]	1.157 ± 163 [942, 1.341]	1.442 ± 248 [1.161, 1.761]
Rate40	1.360 ± 748 [696, 2.247]	2.546 ± 1.251 [1.469, 4.218]	2.871 ± 1.338 [1.660, 4.666]	3.175 ± 1.409 [1.854, 4.980]

Tabela 6.3: Latência total — regime saturado (**segundos**) — média ± desvio padrão amostral [min, máx] entre 5 repetições. Mesma execução de 28 de maio de 2026.

Cenário	Média (s)	P95 (s)	Máximo (s)
Rate60	24,9 ± 18,0 [9,4, 52,8]	36,7 ± 19,2 [18,3, 65,9]	38,9 ± 19,5 [20,4, 68,9]
Rate80	116,1 ± 43,7 [65,8, 177,4]	130,2 ± 44,8 [79,6, 193,1]	133,5 ± 45,0 [83,4, 196,8]
Burst	271,2 ± 58,6 [196,5, 349,3]	286,4 ± 59,9 [208,9, 365,0]	290,6 ± 60,1 [213,1, 369,2]

Nota metodológica: no regime saturado, as cinco execuções de cada cenário não constituem réplicas independentes. Como o *backlog* acumulado no Kafka não é zerado entre repetições, cada execução parte de uma fila maior do que a anterior, e a latência cresce de forma quase monotônica ao longo de R1–R5 (ver Seção 6.7.2 e a Tabela 6.5). Por esse

motivo, os valores de média e desvio padrão amostral da Tabela 6.3 não devem ser interpretados como dispersão de medições repetidas, mas como a faixa percorrida por uma única série de degradação progressiva. O desvio padrão mede a amplitude da degradação ao longo da rodada, não a variabilidade entre realizações equivalentes do mesmo cenário. Consequentemente, os valores absolutos do regime saturado dependem do comprimento total da rodada e não representam a latência do regime estacionário. O achado robusto e independente da duração é o qualitativo: acima de ∼30 op/s o consumidor não escoa a taxa de chegada e a fila cresce sem limite. A obtenção de réplicas verdadeiramente independentes nesse regime exigiria zerar o backlog (reset de offsets ou drenagem completa) antes de cada repetição.

Quatro observações sobre as Tabelas 6.2 e 6.3: (i) os cenários Low, Mixed e Moderate (1 a 10 op/s) permanecem com latência média abaixo de 280 ms e P99 abaixo de 640 ms, (ii) Rate25 mantém a média baixa (361 ms) mas já leva o P99 para ∼1,16 s, ou seja, a cauda da distribuição passa a cruzar 1 s, (iii) Rate40 marca o joelho da curva: a média salta para 1,36 s com dispersão larga ([696, 2.247] ms), pois as três primeiras repetições ficaram em ∼700–915 ms e as duas últimas pularam para ∼2,1–2,2 s à medida que o *backlog* se acumulou dentro do próprio cenário, (iv) de Rate60 em diante o sistema entra em saturação franca, com latências medianas de dezenas a centenas de segundos.

6.5 Resultados de *throughput*

Tabela 6.4: *Throughput* por cenário — média das 5 repetições.

Cenário	**Taxa nominal (op/s)**	**Eventos processados (média / 30 s)**	**Taxa efetiva (evt/s)**
Low	1	30	1,0
Mixed	5	140	4,7
Moderate	10	266	8,9
Rate25	25	595	19,8
Rate40	40	909	30,3
Rate60	60	880	29,3
Rate80	80	660	22,0
Burst	100	540	18,0

A coluna **Eventos processados** corresponde aos eventos efetivamente registrados na camada *silver* dentro da janela de 30 segundos do gerador de carga, e a **taxa efetiva** é obtida dividindo essa contagem por 30 s. Nos cenários Low, Mixed e Moderate (até 10 op/s), a taxa efetiva acompanha a taxa nominal, o consumidor absorve o ritmo do gerador sem acumular fila. A partir de Rate25, a taxa efetiva começa a divergir da nominal (25 nominal vs 19,8 efetivo), e o *gap* cresce conforme a carga aumenta.

Onde está o teto do consumidor? A taxa efetiva sobe até Rate40 (30,3 evt/s) e, a partir daí, *cai* de forma monotônica: 29,3 evt/s no Rate60, 22,0 no Rate80 e apenas 18,0 no Burst. Esse comportamento não-monotônico é a evidência empírica mais clara de que a janela de 30 s mistura dois regimes: nos primeiros segundos o consumidor escoa próximo do seu teto, e depois passa a competir com o crescimento da fila e quanto mais agressivo o gerador, mais rapidamente o *backlog* se acumula e rouba ciclos do processamento de eventos novos. Há ainda um segundo fator que agrava a queda nos cenários mais intensos: a cada disparo da camada *gold* (a cada 50 eventos), o `GoldProcessor` reconstrói o estado lendo *todos* os arquivos *silver* acumulados, como o número de arquivos cresce ao longo da rodada, esse custo síncrono passa a bloquear o *loop* de consumo por intervalos cada vez maiores, reduzindo ainda mais a taxa de escoamento nos cenários finais. Em outras palavras, **a vazão de regime estacionário do consumidor não pode ser lida diretamente desta tabela**: o pico de 30,3 evt/s em Rate40 é uma estimativa superior contaminada pelo *backlog* crescente e pelo custo da *gold*, não a capacidade real. A medida honesta exigiria um protocolo dedicado: deixar um *backlog* acumular, desligar o gerador, e medir o ritmo de drenagem com fila não-vazia e taxa de chegada zero, experimento ainda não realizado e apontado como trabalho futuro no Capítulo 7. Mesmo com essa ressalva, os dados são suficientes para situar a capacidade do consumidor PHP *single-threaded* entre ∼20 evt/s (taxa efetiva do Rate25, sustentada com latência média ainda estável em 361 ms, embora já com pequeno déficit ante a nominal de 25 op/s) e ∼30 evt/s (pico de escoamento observado no Rate40, já no joelho da curva). Reforçando, a faixa de ∼20–30 evt/s deve ser lida estritamente como estimativa superior da vazão de regime, e não como capacidade medida. Ela está contaminada por dois efeitos que crescem ao longo da rodada, a competição com o backlog acumulado e o custo O(n) da reconstrução

da camada gold, que relê todos os arquivos silver a cada 50 eventos. A medição limpa do teto de vazão exige o protocolo dedicado de drenagem (acumular backlog, desligar o gerador e medir o ritmo de escoamento com taxa de chegada zero), conforme indicado no Capítulo 7.

A Tabela 6.5 detalha a degradação do cenário Burst ao longo das cinco repetições (valores de latência em **segundos**).

Tabela 6.5: Degradação do cenário Burst entre repetições (latências em segundos).

Rep.	**Eventos proc.**	**Lat. média (s)**	**P95 (s)**	**Máximo (s)**
R1	550	196,5	208,9	213,1
R2	600	238,8	256,8	261,0
R3	500	268,7	279,9	282,9
R4	550	302,9	321,4	326,9
R5	500	349,3	365,0	369,2

6.6 Curva latência × *throughput*

A Figura 6.1 apresenta a latência média em função da taxa nominal de carga para os oito cenários da grade ampliada, em escala vertical logarítmica. As barras de erro representam o desvio padrão amostral entre as 5 repetições de cada cenário.

A curva mostra três achados empíricos importantes. Primeiro, a latência média é praticamente plana e bem comportada de 1 a 25 op/s (232 a 361 ms), e atravessa a marca de 1 segundo exatamente entre Rate25 (361 ms) e Rate40 (1,36 s na média), localizando o joelho da curva na faixa de 25 a 40 op/s. Segundo, ao contrário da rodada exploratória anterior, a curva é agora monotonicamente crescente: com cinco repetições por cenário, a média do Rate25 ficou estável em 361 ms (sem a contaminação pontual que, na rodada $n = 3$, havia inflado esse ponto), e Rate40 aparece claramente acima dele, o que torna o joelho visualmente nítido e elimina a "inversão" que confundia a leitura anterior. Terceiro, a dispersão entre repetições começa a crescer já no Rate40 (desvio padrão de 748 ms sobre média de 1.360 ms) e explode no regime saturado: as barras de erro de Rate60 a Burst abrangem fatores de 2 a 5×, sinalizando que, nessa faixa, o resultado de uma execução individual depende fortemente do estado de *backlog* herdado das repetições anteriores.

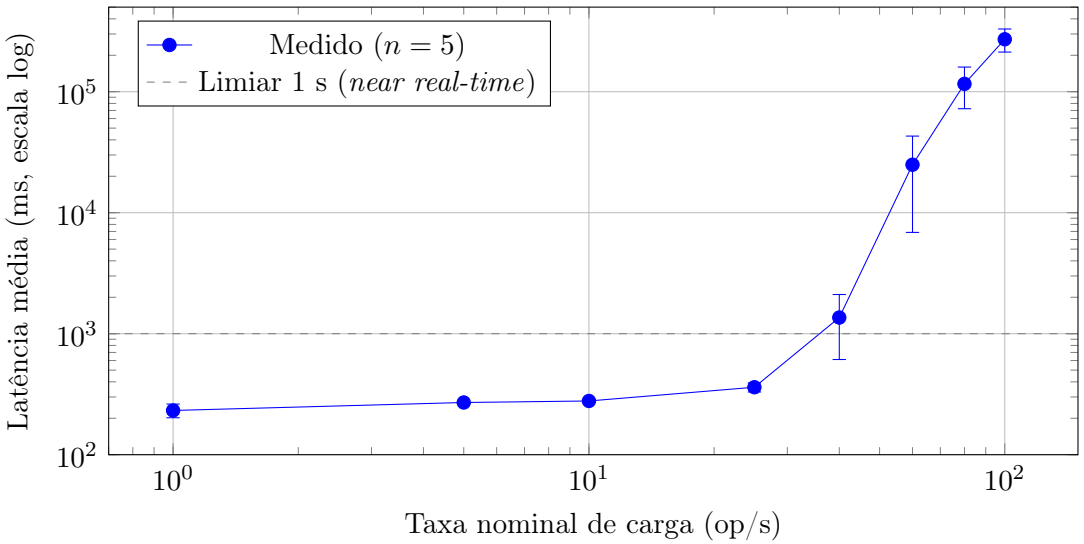

Figura 6.1: Latência média em função da taxa nominal de carga. Cada ponto representa a média de 5 repetições, as barras verticais indicam o desvio padrão amostral. A linha tracejada cinza marca 1 s como referência informal para latência *near real-time*. Observam-se três regimes: (i) regime estável até ∼25 op/s, com latência média abaixo de 400 ms e desvio padrão estreito, (ii) o joelho da curva na faixa de 25–40 op/s (entre Rate25 e Rate40), onde a média cruza 1 s e a dispersão entre repetições se alarga abruptamente, (iii) saturação consolidada a partir de 60 op/s, com latências de dezenas a centenas de segundos crescendo de forma monotônica com a carga.

6.7 Discussão

6.7.1 Cenários de carga normal

Os cenários Low (1 op/s), Mixed (5 op/s) e Moderate (10 op/s) apresentaram comportamento estável e similar entre si na rodada de 28 de maio, com latência média entre 232 e 278 ms e P95 entre 455 e 498 ms. Os desvios padrão amostrais de Mixed e Moderate são da ordem de 4–9 ms, indicando latência muito consistente dentro dessa faixa de carga. O cenário Low apresenta variabilidade um pouco maior (±30 ms na média, ±53 ms no P95), porque com apenas 30 eventos amostrados em 30 s a estatística é sensível a alguns poucos eventos de cauda longa (a r1 do Low teve um evento de 527 ms, suficiente para puxar o P95 daquela repetição). Não foi conduzido teste estatístico formal de igualdade de médias as $n = 5$ repetições atuais permitem caracterizar a dispersão mas não sustentam inferência formal, portanto a afirmação aqui é descritiva: a faixa de variação observada é compatível com a hipótese de que, dentro da faixa de 1 a 10 op/s, o tipo de operação (INSERT vs UPDATE) e a taxa nominal não são fatores dominantes da latência. O cenário Rate25

estende essa observação até ∼25 op/s (média ainda baixa, 361 ms), e Rate40 marca o joelho da curva, com a média já saltando para 1,36 s e dispersão larga entre repetições.

O P99 permanece abaixo de 520 ms nos cenários Low e Mixed e sobe para 632 ms no Moderate, o que ainda sustenta a caracterização *near real-time* dentro do ambiente avaliado. Já no Rate25 o P99 sobe para ∼1,16 s ou seja, a cauda da distribuição passa a cruzar 1 s mesmo com a média ainda baixa, e no Rate40 todos os percentis a partir do P95 ficam acima de 1 s, limiar informal a partir do qual a caracterização de "*near real-time*" deixa de se sustentar para a cauda da distribuição.

Decomposição da latência: onde estão os ∼230–360 ms? A rodada de 28 de maio de 2026 foi conduzida com a decomposição da latência CDC instrumentada (Seção 5.3.4), permitindo atribuir cada parcela do total a uma etapa concreta do *pipeline*. A Tabela 6.6 apresenta as médias por cenário.

Tabela 6.6: Decomposição da latência média por cenário (ms).

Cenário	`dbz` **(ms)** *Debezium interno*	`cdc` **(ms)** *MySQL → bronze*	`etl` **(ms)** *bronze → silver*	**Total (ms)** *MySQL → silver*
Low	5	231	0	232
Mixed	5	269	0	270
Moderate	4	277	1	278
Rate25	3	360	0	361
Rate40	3	1.360	0	1.360
Rate60	2	24.924	0	24.924
Rate80	2	116.095	0	116.095
Burst	2	271.231	0	271.231

É importante esclarecer a relação entre as colunas para a correta leitura da tabela. O **Total** (MySQL → *silver*) decompõe-se de forma aditiva em `cdc` (MySQL → *bronze*) mais `etl` (*bronze* → *silver*), a igualdade `cdc` + `etl` = Total é verificada em todos os cenários (a diferença residual, de no máximo ∼1 ms, corresponde ao instante de registro da métrica). Já a coluna `dbz` **não** é uma parcela paralela: ela é um *sub-intervalo* interno à `cdc`, medindo apenas o trecho entre o *commit* no *binlog* (`source.ts_ms`) e o processamento pelo conector (`ts_ms`).

Com essa leitura, os dados confirmam empiricamente o argumento central: **praticamente toda a latência ponta a ponta está concentrada na coluna** `cdc`, e mais

especificamente na parcela `cdc − dbz` (cerca de 226 ms no cenário Low), que corresponde a publicação no Kafka + trânsito + consumo pelo PHP + escrita na *bronze*. As duas etapas que conseguimos isolar mostram-se desprezíveis em todos os cenários: o processamento interno do Debezium (`dbz`) fica entre 2 e 5 ms, e a transformação *bronze* → *silver* em PHP (`etl`) fica em 0–1 ms. Em outras palavras, nem o conector nem o código PHP de transformação são gargalos, o tempo está no caminho entre o Debezium e a chegada do evento ao consumidor. *Caveat*: o sub-intervalo que separaria "publicação no Kafka" de "trânsito + consumo PHP" (colunas `latency_publish_ms` e `latency_kafka_ms`, baseadas em `ts_kafka_connect` adicionado pela SMT `InsertField`) ficou nulo na execução porque o campo não foi populado pelo conector, a SMT requer que o *ConnectRecord* carregue um *timestamp* do produtor, comportamento que não ocorreu nesta configuração. Essa separação fina fica como ajuste para a próxima rodada.

O resultado tem uma implicação clara: o componente que domina a latência é o que está entre o Debezium e a chegada do evento no PHP, ou seja, ou o ritmo de publicação do Debezium no Kafka, ou a latência de *polling* do consumidor PHP via `php-rdkafka`, ou uma combinação dos dois. O conector MySQL do Debezium utilizado está configurado em `config/debezium-mysql-source.json` sem definir explicitamente a propriedade `poll.interval.ms`, operando com o valor padrão de 500 ms (Red Hat, 2026). É importante precisar a semântica desse parâmetro, ele define o tempo máximo que o conector aguarda por novos eventos quando sua fila interna está vazia, antes de devolver um lote ao framework Kafka Connect (sendo, ainda, limitado a 5 s pela implementação e interagindo com `max.batch.size`). Um modelo de primeira ordem, supondo que uma operação ocorra em um instante arbitrário dentro de uma janela de *polling* de 500 ms, prevê um tempo médio de espera de metade desse intervalo, ou seja, ∼250 ms. Esse valor é da ordem de e compatível com a faixa de 232–278 ms observada nos cenários sustentáveis (Low/Mixed/Moderate). Em particular, a média do *Low* (232 ms) fica a menos de 20 ms em relação a essa referência. Trata-se de uma aproximação, e não de uma derivação exata, o *batching* real também depende de `max.batch.size` e da forma como o `poll()` drena a fila no meio do intervalo, mas o casamento numérico é forte o suficiente para indicar que o fator dominante está entre o Debezium e a chegada do evento ao consumidor, e não

no conector nem no código PHP. A confirmação direta dessa hipótese é o experimento sugerido no Capítulo 7, reduzir poll.interval.ms (por exemplo, para 100 ms) e verificar se a latência média cai proporcionalmente, até o teto imposto pelo batch do Kafka.

6.7.2 Cenário *Burst*

O Burst mostrou onde o *pipeline* atinge seu limite mais agressivo. Com 100 operações por segundo no gerador de carga, o consumidor PHP fica muito atrás: a latência média ao longo das cinco repetições ficou em ∼271 segundos e o pior caso individual chegou a ∼369 segundos (última repetição, ver Tabela 6.5). O fenômeno mais revelador foi a degradação progressiva e quase monotônica entre as repetições (Tabela 6.5): a latência média subiu de 196,5 s na R1 para 349,3 s na R5, enquanto a quantidade de eventos processados na janela de 30 s permaneceu travada em torno de 500–600. O gerador continua enviando 100 eventos/s, mas o consumidor escoa apenas ∼18 evt/s, e como o *backlog* no Kafka não é zerado entre repetições, cada repetição parte de uma fila ainda maior do que a anterior. Vale ressaltar que os valores absolutos do regime saturado são, em parte, um artefato do comprimento da rodada (cinco repetições de cada cenário, em ordem crescente de carga, sobre o mesmo *backlog* acumulado) e não devem ser lidos como latência de regime estacionário, a comparação com a rodada exploratória anterior ($n = 3$, latências de Burst na casa de dezenas de segundos) deixa claro que o número cresce com a duração total do experimento. O que é robusto e independente da duração é o achado qualitativo: acima de ∼30 op/s a fila cresce sem limite.

O joelho da curva está entre 25 e 40 op/s. Os cenários intermediários introduzidos nesta rodada permitem localizar a transição entre regime sustentável e regime saturado. Rate25 ainda mantém latência média estável (361 ± 31 ms), embora seu P99 já cruze 1 s, Rate40 é o primeiro cenário em que a própria *média* cruza 1 s (1,36 s), com dispersão larga entre repetições ([696, 2.247] ms) as três primeiras repetições ficaram em ∼700–915 ms e as duas últimas saltaram para ∼2,1–2,2 s à medida que o *backlog* se acumulou dentro do cenário. De Rate60 em diante a saturação é franca: a média sobe para ∼25 s no Rate60, ∼116 s no Rate80 e ∼271 s no Burst.

Sobre a interpretação da taxa efetiva: a Tabela 6.4 mostra que o pico de eventos

processados por segundo ocorre em Rate40 (30,3 evt/s) e *cai* de forma monotônica daí em diante (29,3 no Rate60, 22,0 no Rate80, 18,0 no Burst), comportamento que confirma que o número observado na janela de medição mistura processamento de eventos novos com competição contra um *backlog* crescente e contra o custo crescente da reconstrução da camada *gold*. A vazão de regime estacionário do consumidor, medida de forma isolada, deve ficar entre ∼20 evt/s (taxa efetiva do Rate25, ainda com latência estável embora já com pequeno déficit ante a nominal) e ∼30 evt/s (pico de escoamento observado no Rate40), a medida precisa exige protocolo dedicado de drenagem, apontado como trabalho futuro.

Esses resultados oferecem um argumento empírico concreto para a necessidade de motores distribuídos como Flink ou Spark, ou ao menos da paralelização do consumidor via partições Kafka adicionais e múltiplas instâncias no mesmo *consumer group*, em cenários de produção que precisem sustentar mais de ∼30 op/s. Para cargas até ∼25 op/s, o consumidor PHP *single-threaded* é viável.

6.7.3 Impacto do *snapshot* inicial

Conforme descrito na Seção 2.5.1, na primeira execução o Debezium realiza um *snapshot* de todas as tabelas monitoradas, publicando cada registro existente como um evento com operação `r`. Esses eventos têm `ts_ms` igual ao momento da leitura pelo conector não ao momento original da escrita do registro. Para evitar contaminação dos resultados, os experimentos foram conduzidos com a base já inicializada e o conector em modo de captura incremental: antes de cada repetição, o *script* `run_experiment.php` limpa apenas a tabela `pipeline_metrics` (não o estado do Debezium), e a primeira repetição (R1) já encontra o conector posicionado no fim do *binlog*. Como consequência, as latências reportadas refletem apenas eventos de CDC "real" (`c`, `u`, `d`), e não eventos de *snapshot*.

Se o ambiente fosse reiniciado entre repetições (`docker compose down -v`), a R1 do cenário low conteria os ∼100 eventos de *snapshot* do *seed*, que apareceriam com latência aparente da ordem de centenas de milissegundos a poucos segundos, não por limitação do *pipeline*, mas pela diferença entre o `ts_ms` de leitura e o instante em que o consumidor processou o evento. Essa distinção é importante para qualquer reprodução

do experimento.

6.8 Verificação de consistência

Além das métricas de latência e *throughput*, o protótipo inclui um mecanismo de verificação de consistência (`consistency_check.php`) que compara o estado final do MySQL com as tabelas *gold* do PostgreSQL. Essa verificação é essencial para validar que o *pipeline* propagou todos os dados corretamente, sem perdas, duplicações ou valores incorretos.

A verificação compara: o número total de registros em cada tabela do MySQL com os totais refletidos nas agregações *gold* do PostgreSQL, os valores agregados (como total de profissionais por estabelecimento) calculados diretamente no MySQL com os valores produzidos pelo `GoldProcessor` a partir dos arquivos *silver*, e a presença de registros que foram inseridos, atualizados ou deletados durante os experimentos.

Limitação da verificação adotada. A verificação implementada opera no nível de **contagens e somas agregadas**, e não no nível de *diff* por chave primária. Esse tipo de comparação detecta perdas grosseiras (registros que não chegaram ao destino) e divergências numéricas em métricas agregadas, mas não capta certos modos de falha mais sutis: por exemplo, dois registros distintos no MySQL que tenham sido mesclados em um único registro no destino por colisão de chave produzem a mesma contagem total, e seriam invisíveis para a comparação atual. Da mesma forma, uma corrupção de campo não-agregado (um nome de estabelecimento truncado, uma coordenada deslocada) não altera contagens nem somas, e portanto não seria detectada. Para um trabalho cujo tema central é a fidelidade da propagação MySQL → *lakehouse* → PostgreSQL, uma verificação *key-by-key* (*full outer join* entre origem e destino) seria mais rigorosa e é apontada como evolução natural no Capítulo 7.

A verificação foi conduzida em 28 de maio de 2026, em um cenário controlado, partindo de um ambiente limpo (`docker compose down -v`), em duas etapas. Na primeira, o *snapshot* inicial do Debezium propagou o *seed* pela cadeia CDC → Kafka → *Streaming ETL* → *bronze* → *silver* → *gold*, na segunda, aplicou-se uma carga incremental (cenário *Moderate*, 10 op/s por 30 s) e aguardou-se a drenagem completa do *backlog*

no Kafka (*lag* zero) antes de materializar a camada *gold* e comparar. Trata-se de uma execução distinta da rodada de latência/*throughput* (também de 28 de maio): como a rodada de desempenho satura deliberadamente o *pipeline*, ela é incompatível com a medição de consistência em *backlog* drenado, de modo que cada uma foi conduzida em seu próprio ambiente limpo. A Tabela 6.7 apresenta os resultados.

Tabela 6.7: Verificação de consistência MySQL (origem) × PostgreSQL *gold*, em cenário controlado com *backlog* drenado.

Verificação	**Após *snapshot* (seed)**		**Após carga *Moderate***	
	MySQL	**Gold**	**MySQL**	**Gold**
Estabelecimentos	10	10	55	55
Municípios	8	8	10	10
Profissionais	20	20	20	20
Σ equipamentos (`qt_existente`)	49	49	49	49
Σ horas ambulatoriais	662	662	3.001	3.001

Em ambas as etapas, todas as contagens e somas agregadas coincidiram exatamente entre origem e destino, e o *script* `consistency_check.php` reportou **consistência OK**. Esse resultado confirma que, com o consumidor operando de forma contínua e o *backlog* drenado, a propagação MySQL → *gold* preserva a integridade dos dados no nível de contagens e somas tanto para o *snapshot* inicial quanto para as alterações incrementais capturadas por CDC.

Nota sobre a materialização *gold*. Durante o desenvolvimento, observou-se que o `GoldProcessor` falhava ao gravar registros cujos campos `DECIMAL` (latitude e longitude) chegavam vazios, pois o PostgreSQL rejeita *string* vazia em colunas numéricas (`invalid input syntax for type numeric`). A correção consistiu em normalizar esses campos para `NULL` antes do *UPSERT*. Após a correção, a materialização *gold* e a verificação de consistência da Tabela 6.7 foram executadas sem erros. Cabe registrar que as métricas de latência e *throughput* reportadas nas seções anteriores são coletadas pelo `MetricsCollector` de forma independente do `GoldProcessor` (a partir dos *timestamps* do evento, antes da etapa *gold*) e, portanto, não eram afetadas por esse problema.

Cenários saturados. Nos cenários de alta carga (Rate60 em diante), a verificação imediatamente após a janela de 30 s exibiria divergência transitória, pois parte dos eventos

ainda estaria enfileirada no Kafka. Trata-se de manifestação da consistência *eventual*: como demonstrado no cenário controlado acima, uma vez drenado o *backlog*, as agregações convergem com o estado da origem.

Sobre a "ausência de perdas". O comportamento descrito é compatível com a configuração de entrega adotada: o consumidor opera com `enable.auto.commit=true` e `auto.commit.interval.ms=1000`, o que permite duplicatas ou perdas em caso de falha entre o consumo de um *batch* e a escrita na camada *silver*. Em outras palavras, o protótipo oferece *best-effort delivery*, e não *at-least-once* estrito. A obtenção de *at-least-once* exigiria alternar para *commit* manual de *offsets* após a confirmação da escrita *silver*.

É importante delimitar com cuidado o que os experimentos realmente sustentam sobre robustez. Em todas as execuções, nenhuma perda ou duplicação foi detectada pela verificação de consistência, **na ausência de falhas induzidas**: nenhuma das execuções derrubou o consumidor PHP no meio de um *batch*, simulou uma falha de rede entre o consumidor e o Kafka, ou matou o contêiner do PostgreSQL durante a escrita das métricas. O que se observou no cenário Burst foi apenas atraso na propagação (consistência eventual com fila não-vazia), o que é fenômeno distinto de perda em falha. Portanto, a afirmação "nenhuma perda foi observada" deve ser lida como "nenhuma perda foi observada nas condições nominais de operação testadas", e não como uma medida de tolerância a falhas. Um teste explícito de injeção de falha (e.g., `docker kill php-app` durante carga, seguido de `consistency_check.php`) é o experimento natural para fechar essa lacuna e fica como trabalho futuro.

6.9 Síntese

Os resultados indicam que, dentro do ambiente avaliado, o *pipeline* sustenta latência média entre 232 e 361 ms para cargas de até aproximadamente 20 eventos efetivos por segundo (cenário Rate25), configurando atualização *near real-time*. Uma alteração feita no banco transacional, como o cadastro de um novo estabelecimento, a atualização da carga horária de um profissional, ou a ativação de uma equipe de saúde fica disponível na camada *silver* em frações de segundo nesses cenários (média abaixo de 400 ms, ainda que

a cauda P99 já cruze 1 s no Rate25).

A decomposição instrumentada da latência (Seção 6.7.1, Tabela 6.6) confirma empiricamente que a quase totalidade desse tempo está concentrada no intervalo entre o *commit* no MySQL e a chegada do evento à camada *bronze* (coluna `cdc`). O processamento interno do Debezium contribui com 2–5 ms, e a transformação *bronze* → *silver* (totalmente em PHP) contribui com 0–1 ms, portanto nem o conector nem o código PHP são gargalos relevantes. O fator dominante é compatível com o intervalo de *polling* default do Debezium (`poll.interval.ms` = 500 ms): a latência média esperada de metade desse intervalo (∼250 ms) cai praticamente no centro da faixa observada nos cenários sustentáveis (232–278 ms), e a média do Low (232 ms) fica a menos de 20 ms dessa previsão. Reduzir esse intervalo é a primeira ação de *tuning* sugerida no Capítulo 7.

No que diz respeito à fidelidade da propagação, a verificação de consistência em cenário controlado (Seção 6.8, Tabela 6.7) confirmou a igualdade exata entre origem e destino em todas as contagens e somas agregadas, tanto após o *snapshot* inicial quanto após uma carga incremental com *backlog* drenado, evidência de que a propagação MySQL → *gold* preserva a integridade dos dados no nível avaliado. Nos cenários saturados, a divergência observada imediatamente após a janela de medição é transitória e desaparece após a drenagem do *backlog*, caracterizando consistência *eventual*. A utilização de dados do CNES demonstrou que a arquitetura opera sobre um modelo relacional não trivial, sete tabelas transacionais, chaves alfanuméricas e dezesseis dimensões.

O joelho da curva latência × carga foi localizado entre Rate25 (361 ms de média estável, embora com P99 já próximo de 1 s) e Rate40 (1,36 s de média, primeiro cenário em que a média cruza 1 s, com dispersão larga entre repetições). Acima desse ponto, o consumidor PHP *single-threaded* não sustenta a taxa de chegada e o *backlog* cresce, levando a degradação progressiva (latência média de ∼25 s no Rate60, escalando a centenas de segundos no Burst). A vazão de regime estacionário do consumidor, medida ainda contaminada pelo *backlog* dentro da janela, fica em algum ponto entre ∼20 e ∼30 evt/s, a medida precisa exige protocolo dedicado de drenagem, apontado como trabalho futuro. Para cargas sustentadas acima desse patamar, a migração para um motor distribuído como Apache Flink ou Spark Structured Streaming, ou ao menos a paralelização

do consumidor via partições Kafka adicionais e múltiplas instâncias no mesmo *consumer group*, é apontada como evolução natural no Capítulo 7.

7 Considerações Finais e Trabalhos Futuros

Este trabalho teve como objetivo desenvolver e avaliar um protótipo de atualização *near real-time* entre um banco de dados transacional e um ambiente analítico organizado em camadas bronze/silver/gold, utilizando dados reais do CNES/DATASUS. A solução foi construída integrando o MySQL como banco de dados transacional de origem, o Debezium para a captura de mudanças por meio da leitura do *binlog*, o Apache Kafka como barramento de eventos e um consumidor PHP no papel de motor de *Streaming ETL*. Toda a infraestrutura roda em Docker, com ferramentas *open source*, e pode ser reproduzida com um único comando.

Um ponto central da implementação final é que nenhum componente do *pipeline* analítico realiza consultas ao MySQL. O `GoldProcessor` reconstrói o estado de cada tabela a partir dos arquivos da camada *silver*, conforme o método `loadLatestState()` apresentado no Listing 5.6. Essa decisão garante que toda a propagação de dados ocorra, de fato, pelo fluxo de CDC e *Streaming ETL*, e não por consultas paralelas ao banco de origem, uma correção conceitual que se mostrou relevante ao longo do desenvolvimento.

A utilização de dados reais do CNES trouxe mais complexidade e realismo ao protótipo em comparação com o domínio de e-commerce fictício utilizado em versões anteriores. O modelo, com sete tabelas transacionais, chaves compostas e múltiplas dimensões, exercita o *pipeline* de forma mais representativa e demonstra que a arquitetura proposta não se limita a cenários simples.

Retomando a questão de pesquisa que orientou o trabalho, os resultados mostram que uma arquitetura *lakehouse* alimentada exclusivamente por CDC e *Streaming ETL*, montada com ferramentas *open source* em uma única máquina, sustenta atualizações *near real-time* enquanto a carga permanece moderada. Até cerca de 25 op/s, a latência média ficou entre 232 e 361 ms e o comportamento se manteve estável. A transição para o regime saturado, o joelho da curva, situa-se entre 25 e 40 op/s, faixa em que a latência média cruza a marca de 1 segundo. Acima de aproximadamente 30 op/s, o consumidor *single-threaded* deixa de escoar a taxa de chegada, a fila cresce sem limite e a latência se

degrada para dezenas ou centenas de segundos.

Quanto aos fatores que limitam o desempenho, a decomposição instrumentada da latência identificou dois gargalos distintos. Em cargas baixas, o tempo é dominado pelo intervalo de *poll* do Debezium, configurado no *default* de 500 ms, e não pelo conector nem pelo código de transformação em PHP, que juntos somam poucos milissegundos. Em cargas altas, o limite passa a ser a natureza *single-threaded* do consumidor, agravada pelo custo crescente de reconstrução da camada *gold*. Em condições nominais e com a fila drenada, a verificação de consistência não acusou perdas nem divergências entre a origem e o destino, o que indica que a propagação preserva a integridade dos dados no nível avaliado.

7.1 Contribuições

As principais contribuições deste trabalho são:

- A implementação de uma arquitetura completa de atualização *near real-time*, na qual toda a camada analítica é construída a partir do *pipeline* de *streaming*, sem nenhuma consulta ao banco de origem, disponibilizada como protótipo reproduzível em Docker e de código aberto.
- A aplicação do *pipeline* a dados reais do CNES/DATASUS, evidenciando sua viabilidade em um domínio de saúde pública de relevância social e em um modelo relacional não trivial.
- A avaliação experimental sob oito intensidades de carga, com cinco repetições por cenário, que localizou empiricamente o joelho da curva de latência × *throughput* entre 25 e 40 op/s.
- A decomposição instrumentada da latência a partir dos *timestamps* do próprio Debezium, que isolou o processamento do conector (entre 2 e 5 ms) e identificou o `poll.interval.ms` *default* de 500 ms como fator dominante da latência em cargas baixas.

7.2 Limitações

O protótipo também apresenta limitações que merecem registro. O consumidor é *single-threaded*, sem paralelismo, *checkpointing* ou garantia *exactly-once*. Como o *commit* de *offsets* no Kafka é automático e periódico (<enable.auto.commit=true> e <auto.commit.interval.ms=1000>), a entrega é, em rigor, *best-effort*, de modo que, em caso de falha entre o consumo e a escrita *silver*, eventos podem ser duplicados ou perdidos. A obtenção de *at-least-once* estrito exigiria o *commit* manual de *offsets* após a confirmação da escrita.

No armazenamento, o *lakehouse* usa arquivos JSONL, sem transações ACID, versionamento ou otimizações de leitura. Formatos como Delta Lake e Iceberg resolveriam essas lacunas, mas não dispõem de bibliotecas para PHP, o que exigiria um componente em Python ou Java. Além disso, a reconstrução do estado na camada *gold* carrega todos os arquivos *silver* em memória, o que se torna inviável para volumes grandes, e essa camada opera em micro-*batch* periódico, e não evento por evento. Uma agregação incremental pura exigiria gerenciamento de estado com persistência, algo que o PHP não oferece nativamente.

Quanto à avaliação, os testes foram conduzidos em ambiente local, com Docker em uma única máquina, sem simular latência de rede ou ambientes distribuídos, e utilizaram um *seed* reduzido, de cerca de uma centena de registros, em vez da base completa do CNES, com mais de 22 milhões de registros, o que limita a análise de escalabilidade. A verificação de consistência, por sua vez, compara contagens e somas agregadas, sem realizar *diff* por chave primária, de modo que modos de falha mais sutis, como colisão de chave ou corrupção de campo não-agregado, não seriam detectados. Uma comparação *key-by-key* é apontada como evolução natural.

7.3 Trabalhos futuros

Diversos desdobramentos podem dar continuidade a este trabalho. No plano de desempenho, o passo mais imediato é substituir o consumidor PHP por um motor distribuído como Flink ou Spark, comparando *throughput* e tolerância a falhas. Na mesma direção, é interessante reduzir o `poll.interval.ms` do Debezium, do *default* de 500 ms para algo

como 100 ms, e verificar se a latência média cai na proporção prevista pela análise da Seção 6.7.1. Também é necessário medir a vazão de regime saturado de forma isolada, gerando *backlog*, desligando o gerador e medindo o ritmo de drenagem, de modo a obter um teto de *throughput* livre da contaminação da fila e do custo crescente da reconstrução da *gold*, já que a estimativa atual, entre 20 e 30 evt/s, é apenas um limite superior.

No que diz respeito ao armazenamento, a adoção de Delta Lake ou Iceberg traria transações ACID e *time travel* ao *lakehouse*, enquanto uma agregação incremental na camada *gold* evitaria reconstruir o estado do zero a cada execução.

Para avaliar escalabilidade e realismo, seria importante importar a base completa do CNES, por meio do *script* `import_cnes_csv.sh`, e observar o comportamento do *pipeline* com volume realista, além de executar os experimentos em nuvem para medir a escalabilidade horizontal e o impacto da latência de rede. Nesse cenário distribuído, torna-se indispensável sincronizar os relógios dos hosts por NTP e revalidar a decomposição da latência, já que a `latency_cdc_ms` é medida entre o relógio da origem e o do consumidor.

Do ponto de vista do rigor estatístico, recomenda-se aumentar o número de repetições, de $n = 5$ para $n = 10$ ou mais, e conduzir testes formais de comparação entre cenários, em especial para confirmar a localização precisa do joelho da curva entre 25 e 40 op/s. Convém ainda ajustar a SMT `InsertField` para que o campo `ts_kafka_connect` seja efetivamente populado, o que ficou nulo na rodada atual e impediu separar empiricamente a parcela de publicação no Kafka da parcela de trânsito e consumo em PHP dentro da `latency_cdc_ms`.

No tocante à robustez, um teste de injeção de falha, como um `docker kill php-app` durante a carga, permitiria validar a verificação de consistência e evoluí-la para uma comparação *key-by-key*, via *full outer join* entre origem e destino, capaz de detectar a corrupção de campos individuais que a comparação atual, baseada em contagens e somas, não capta. Por fim, a integração com ferramentas de visualização como Grafana ou Metabase abriria espaço para *dashboards* em tempo real dos indicadores de saúde.

Referências Bibliográficas

Apache Software Foundation. *Apache Iceberg: High-Performance Table Format for Huge Analytic Tables*. 2024. <https://iceberg.apache.org/>. Acesso em: mar. 2026.

Apache Software Foundation. *Apache Kafka Documentation*. 2024. <https://kafka.apache.org/documentation/>. Acesso em: mar. 2026.

Apache Software Foundation. *What is Apache Hudi*. 2024. <https://hudi.apache.org/docs/overview/>. Acesso em: mar. 2026.

ARMBRUST, M. et al. Structured streaming: A declarative API for real-time applications in Apache Spark. In: *Proceedings of the 2018 International Conference on Management of Data (SIGMOD '18)*. [S.l.]: ACM, 2018. p. 601–613.

ARMBRUST, M. et al. Lakehouse: A new generation of open platforms that unify data warehousing and advanced analytics. In: *Proceedings of the 11th Annual Conference on Innovative Data Systems Research (CIDR '21)*. [S.l.: s.n.], 2021.

CARBONE, P. et al. Apache Flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, v. 36(4), 2015.

Databricks. *What is a Data Lakehouse?* 2024. <https://www.databricks.com/glossary/data-lakehouse>. Acesso em: mar. 2026.

DATASUS. *Arquivos Base de Dados CNES*. 2026. <https://cnes.datasus.gov.br/pages/downloads/arquivosBaseDados.jsp>. Acesso em: mar. 2026.

HARBY, A. A.; ZULKERNINE, F. Data lakehouse: A survey and experimental study. *Information Systems*, Elsevier, v. 127, p. 102460, 2025.

INMON, W. H. *Building the Data Warehouse*. 4th. ed. [S.l.]: John Wiley & Sons, 2005.

JAIN, P. et al. Change data capture for real-time data integration: A comparative study. In: *2023 International Conference on Communication, Security and Artificial Intelligence (ICCSAI)*. [S.l.]: IEEE, 2023.

KIMBALL, R.; ROSS, M. *The Data Warehouse Toolkit: The Definitive Guide to Dimensional Modeling*. 3rd. ed. [S.l.]: John Wiley & Sons, 2013.

KLEPPMANN, M. *Designing Data-Intensive Applications*. [S.l.]: O'Reilly Media, 2017.

KREPS, J.; NARKHEDE, N.; RAO, J. Kafka: A distributed messaging system for log processing. In: *Proceedings of the 6th International Workshop on Networking Meets Databases (NetDB)*. [S.l.: s.n.], 2011.

MAZUMDAR, D.; HUGHES, J.; ONOFRé, J.-B. The data lakehouse: Data warehousing and more. *arXiv preprint arXiv:2310.08697*, 2023.

MERKEL, D. Docker: Lightweight Linux containers for consistent development and deployment. *Linux Journal*, v. 2014, n. 239, 2014.

Ministério da Saúde. *CNES – Cadastro Nacional de Estabelecimentos de Saúde*. 2026. <https://cnes.datasus.gov.br/>. Acesso em: mar. 2026.

Red Hat. *Debezium Documentation*. 2026. <https://debezium.io/documentation/reference/stable/>. Acesso em: mar. 2026.

ZAHARIA, M. et al. Apache Spark: A unified engine for big data processing. *Communications of the ACM*, v. 59, n. 11, p. 56–65, 2016.