Pontifícia Universidade Católica
do Rio de Janeiro

João Gabriel Cavalcanti Dini Nielsen

Avaliação de Arquiteturas de Streaming: Experimentos de Desempenho com Apache Kafka.

Relatório de Projeto Final

Relatório de Projeto Final, apresentado ao Programa de Engenharia da Computação, do Departamento de Informática da PUC-Rio como requisito parcial para a obtenção do titulo de Bacharel em Engenharia da Computação.

Orientador: Prof. Marcos Vianna Villas

Rio de Janeiro
Junho de 2025

Pontifícia Universidade Católica
do Rio de Janeiro

João Gabriel Cavalcanti Dini Nielsen

Avaliação de Arquiteturas de Streaming: Experimentos de Desempenho com Apache Kafka.

Relatório de Projeto Final, apresentado ao Programa de Engenharia da Computação, do Departamento de Informática da PUC-Rio como requisito parcial para a obtenção do titulo de Bacharel em Engenharia da Computação.

Prof. Marcos Vianna Villas
Orientador
Departamento de Informática – PUC-Rio

Rio de Janeiro, 29 de Junho de 2025

João Gabriel Cavalcanti Dini Nielsen

Graduando em Engenharia da Computação na PUC-Rio

Ficha Catalográfica

João Gabriel Cavalcanti Dini Nielsen

Avaliação de Arquiteturas de Streaming: Experimentos de Desempenho com Apache Kafka. / João Gabriel Cavalcanti Dini Nielsen; orientador: Marcos Vianna Villas. – 2025.

57 f: il. ; 30 cm

Projeto Final - Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática, 2025.

Inclui bibliografia

1. Informática – Teses. 2. Dados em fluxo contínuo. 3. Apache Kafka. 4. Processamento de dados em tempo quase real. 5. Mensageria distribuída. I. Villas, Marcos. II. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. III. Título.

CDD: 004

Agradecimentos

Ao meu orientador Professor Marcos Villas por todo o apoio e orientação durante a realização deste trabalho.

Aos meus pais e familiares pelo apoio e a oportunidade de realizar essa graduação.

Aos meus colegas da PUC-Rio.

Aos professores que participaram da Comissão examinadora.

A todos os professores e funcionários do Departamento pelo ótimo trabalho durante esses anos.

A todos os amigos que de alguma forma tornaram a jornada mais leve, em especial àqueles cuja presença constante, mesmo de outro país, fez toda a diferença.

Resumo

João Gabriel Cavalcanti Dini Nielsen; Villas, Marcos. **Avaliação de Arquiteturas de Streaming: Experimentos de Desempenho com Apache Kafka.**. Rio de Janeiro, 2025. 57p. Relatório de Projeto Final – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Este trabalho investigou a aplicação de sistemas de streaming de dados, com foco na ingestão contínua de eventos e na influência das configurações do sistema sobre o desempenho. Foi apresentado um modelo de referência composto por três etapas — ingestão, processamento e armazenamento — mediadas por um componente de *message broker*. A partir desse modelo, escolheu-se o Apache Kafka como ferramenta para implementação prática e avaliação experimental. Foram realizados testes controlados variando o tamanho das mensagens e os parâmetros de envio do produtor, com o objetivo de medir a latência e o *throughput* do sistema em cenários com rajadas de carga. Os resultados mostraram que ajustes no tamanho do lote e na memória de *buffer* influenciam significativamente a estabilidade e o desempenho. O trabalho combinou fundamentação teórica com experimentação prática, documentando os efeitos de diferentes configurações e fornecendo subsídios para o dimensionamento de pipelines em aplicações reais.

Palavras-chave

Dados em fluxo contínuo; Apache Kafka; Processamento de dados em tempo quase real; Mensageria distribuída.

Abstract

João Gabriel Cavalcanti Dini Nielsen; Villas, Marcos (Advisor). **Assessing Streaming Architectures: Performance Testing with Apache Kafka**. Rio de Janeiro, 2025. 57p. Final Project Report — Department of Informatics, Pontifical Catholic University of Rio de Janeiro (PUC-Rio).

This study investigated the use of data streaming systems, focusing on continuous event ingestion and the influence of system configuration on performance. A reference model was presented, comprising three main stages — ingestion, processing, and storage — mediated by a message broker component. Based on this model, Apache Kafka was selected as the practical tool for implementation and experimental evaluation. Controlled tests were conducted by varying message sizes and producer parameters to measure system latency and throughput under burst-load scenarios. The results showed that tuning batch size and buffer memory significantly impacts system stability and performance. The work combines theoretical foundations with practical experimentation, documenting the effects of different configurations and providing guidance for designing pipelines in real-world applications.

Keywords

Streaming Data; Apache Kafka; Stream Processing in near real-time; Message Broker.

Sumário

Lista de figuras

Lista de tabelas

Lista de Códigos

Lista de Abreviaturas

LZ4 – Algoritmo de compressão de dados com baixa latência.
P50, P95, P99 – Percentis 50, 95 e 99, respectivamente, usados para análise de latência.

1
Introdução

Atualmente, o tema de streaming de dados se torna cada vez mais relevante, devido ao crescimento do volume de informações que precisam ser tratadas, característica marcante do Big Data (Amazon Web Services, 2024a), definido pelos chamados "três Vs": volume, variedade e velocidade. Hoje os dados são disponibilizados por múltiplas fontes com formatos variados, como redes sociais e sensores IoT [1], que precisam ser processados em alta velocidade para suportar tamanha demanda. Com isso, novas ferramentas foram desenvolvidas para conseguirem tratar, processar e analisar esses dados em tempo quase real, ou seja, com latência suficientemente baixa para que a resposta ocorra quase simultaneamente à geração dos eventos.

As soluções tradicionais, como o processamento em lotes, abordagem em que os dados são acumulados por um período e processados em blocos periódicos, conforme definido no dicionário 4.2 , não são adequadas para lidar com esse fluxo contínuo de dados, oriundos de diversas fontes e heterogêneos. Um dos maiores problemas em se tratando desse tópico é a latência, intervalo de tempo entre a entrada de um dado no sistema e a obtenção de uma resposta processada, (conforme definido no dicionário na seção 2.1) elevada que é gerada devido ao armazenamento feito pré processamento, limitando a capacidade de responder a eventos críticos e sensíveis ao tempo. São necessárias soluções que sejam capazes de lidar com a alta velocidade, de maneira contínua e com baixa latência.

Para as empresas, a capacidade de analisar os dados conforme são gerados contribui para uma maior velocidade de decisão e definição de estratégias, aprimorando a eficiência das operações. Em setores como finanças e e-commerce, por exemplo, o processamento em tempo quase real permite uma rápida detecção de fraudes e monitoramento de eventos críticos. Em contrapartida, para a sociedade, podemos ter benefícios para a segurança e qualidade de vida, já que é possível o monitoramento de desastres e de saúde.

[1] IoT, ou Internet das Coisas, representa uma rede de dispositivos interconectados que utilizam tecnologia para se comunicar, seja com a nuvem ou diretamente entre si (Amazon Web Services, 2024b).

2
Objetivos

Este trabalho teve como objetivo analisar o conceito de streaming de dados, abordando desde os fundamentos da tecnologia até sua aplicação prática em um sistema real. Foram descritos os principais conceitos relacionados ao processamento contínuo de dados, com ênfase nos desafios técnicos enfrentados nesse tipo de arquitetura, como latência, escalabilidade e tolerância a falhas.

Em seguida, foi apresentado um modelo de referência que organiza as etapas de ingestão, processamento, armazenamento e *message brokers*. A partir desse modelo, foram analisadas as funcionalidades do Apache Kafka, ferramenta escolhida para representar uma solução de mercado voltada ao streaming de dados.

Além disso, o trabalho buscou avaliar, por meio de testes controlados, como diferentes configurações de produtor e variações de tamanho de mensagem impactam o desempenho do Kafka. Os testes foram conduzidos com foco em *throughput* e latência, simulando cenários de carga com rajadas e medindo o comportamento do sistema diante dessas condições.

3
Metodologia

3.1
Etapas

A seguir, vamos detalhar o diagrama de plano de ação, explicando de forma concisa cada um dos itens que compõem o diagrama. Abordaremos as etapas realizadas, indicando como serão desenvolvidas e implementadas. O objetivo é entregar uma visão mais organizada e direta do processo, visando facilitar o desenvolvimento do estudo.

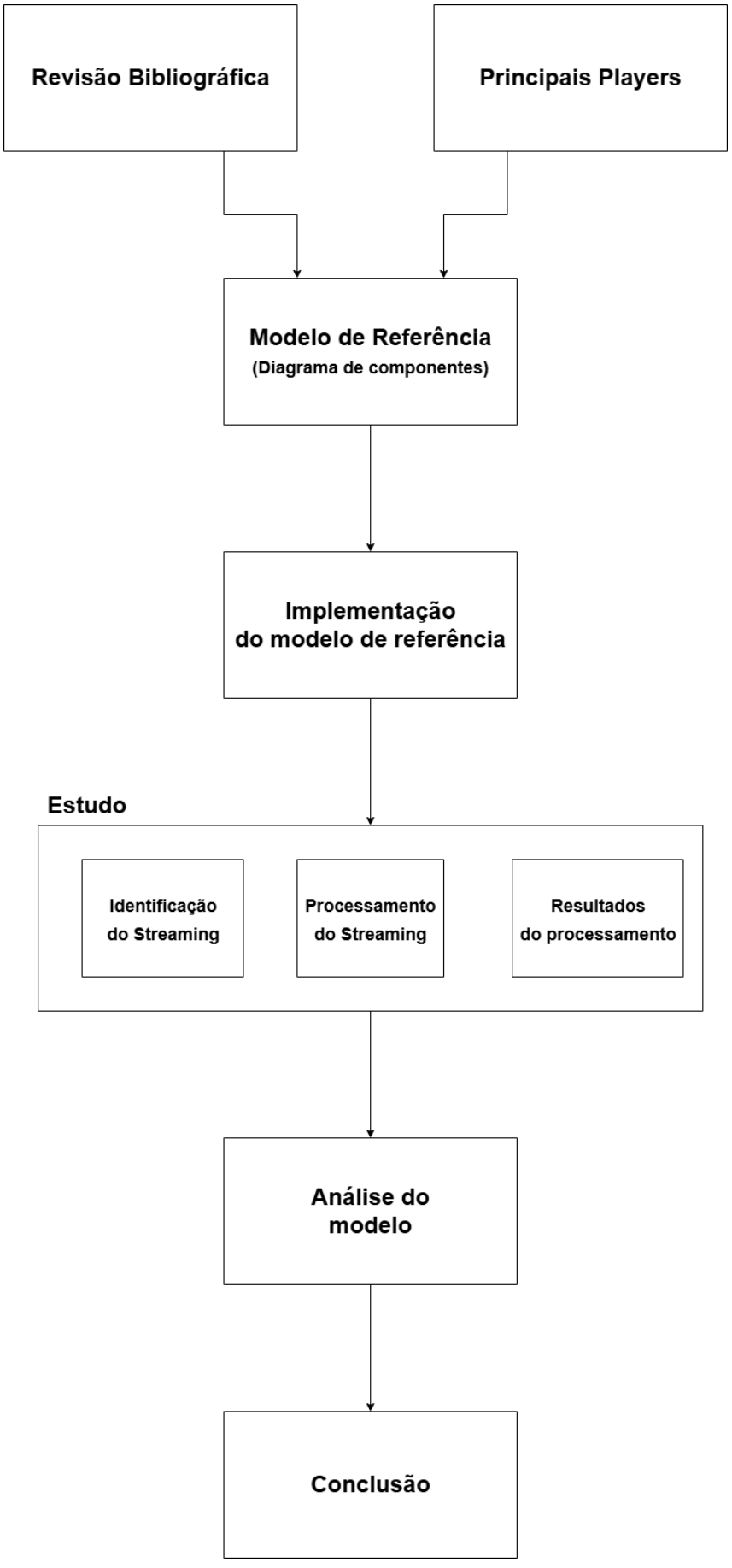

Figura 3.1: Plano de Ação. Fonte: elaborado pelo autor.

3.1.1
Revisão Bibliográfica

Apresenta os principais estudos realizados sobre fluxo de dados contínuos, dando destaque para os principais conceitos e abordagens adotadas em relação ao processamento de dados em tempo quase real.

3.1.2
Principais Players

Análise das principais ferramentas do mercado, com a apresentação de diagramas de fluxo e a discussão das arquiteturas e ferramentas abordadas, explicando como cada uma lida com o streaming de dados.

3.1.3
Modelo de Referência

Apresentação de um modelo de referência para o trabalho, que serve como uma estrutura conceitual para orientar o desenvolvimento e a implementação dos estudos. Além disso, é apresentado um diagrama dos componentes escolhidos que foram trabalhados neste projeto. Os componentes foram escolhidos com base nos estudos realizados e visaram uma implementação que melhor se adapte ao nosso problema.

3.1.4
Implementação do modelo de referência

Foram escolhidas as ferramentas utilizadas no estudo e criado um ambiente apropriado para o desenvolvimento. Os testes foram realizados no ambiente `kafka_perf_test`, configurado especificamente para avaliar o desempenho do Apache Kafka em diferentes cenários, com controle sobre parâmetros como taxa de envio, tamanho de mensagens e configuração dos produtores.

3.1.5
Revisão Bibliográfica

Apresenta os principais estudos realizados sobre fluxo de dados contínuos, dando destaque para os principais conceitos e abordagens adotadas em relação ao processamento de dados em tempo quase real

3.1.6 Estudos

Esta seção apresenta os estudos realizados para a condução do projeto, contemplando desde a definição da carga de teste utilizada até a análise dos resultados obtidos.

3.1.6.1 Definição da Carga de Teste

Inicialmente, o estudo previa a utilização de dados oriundos de redes sociais como fluxo contínuo de informações. No entanto, optou-se por utilizar o *script* `perf_test`, que gera dados sintéticos com controle preciso sobre parâmetros como tamanho das mensagens e taxa de envio. Essa escolha permitiu maior reprodutibilidade dos testes e controle sobre a carga aplicada ao sistema.

3.1.6.2 Processamento do Streaming

Esta etapa consiste no detalhamento do processamento do fluxo de dados utilizando o Apache Kafka como ferramenta principal. Foram analisadas diferentes configurações do produtor Kafka, variando parâmetros como `batch.size`, `throughput`, com o objetivo de observar os impactos no desempenho do sistema.

3.1.6.3 Resultado do Processamento

Nesta etapa, discutem-se as aplicações dos resultados obtidos por meio dos testes realizados. A análise considerou o impacto das diferentes configurações sobre métricas como latência média, throughput e percentis de latência. Avaliou-se como as ferramentas e parâmetros empregados influenciaram a performance observada em cada cenário.

3.1.7 Análise do Modelo

A análise dos resultados obtidos permitiu avaliar a eficácia das soluções implementadas e dos testes realizados. Foram identificados pontos fortes das configurações padrão, bem como desafios relacionados ao controle de latência em cenários com rajadas de mensagens. Discutem-se ainda possíveis melhorias e ajustes nos parâmetros para contextos específicos.

3.1.8 Conclusão

Esta seção apresenta as principais conclusões do estudo, destacando os resultados alcançados, os objetivos não atendidos e as soluções implementadas ao longo do projeto. Também são sugeridas direções para trabalhos futuros.

3.2 Cronogramas

As Tabelas abaixo representam um planejamento semanal para a realização das atividades de projeto final I e II.

Tabela 3.1: Cronograma Projeto Final I. Fonte: elaborado pelo autor.

Atividades	**Agosto**	**Setembro**	**Outubro**	**Novembro**
Revisão Bibliográfica				
Principais Players				
Desenvolvimento de um modelo de referência				

Tabela 3.2: Cronograma Projeto Final II. Fonte: elaborado pelo autor.

Atividades	**Março**	**Abril**	**Maio**	**Junho**	**Julho**
Implementação do modelo de referência					
Definição da Carga de Teste					
Processamento do Streaming					
Resultados do Processamento					
Análise do modelo					
Conclusão					

4
Estudos Preliminares

Nesta seção, apresentamos os principais estudos e definições relacionados ao fluxo contínuo de dados. O objetivo é entender como o tema vem sendo tratado na literatura, destacando os conceitos mais relevantes para o trabalho e os desafios enfrentados por sistemas que lidam com grandes volumes de dados em tempo quase real.

4.1
Revisão Bibliográfica

Heinze et al. (2014) descrevem um sistema de fluxo de dados como consultas contínuas sobre um fluxo potencialmente infinito, produzindo, assim, consultas em tempo quase real. Evidenciando, assim, a natureza contínua e dinâmica dos fluxos de dados. Já Gama e Rodrigues (2007) tratam o fluxo como uma sequência ordenada de instâncias, que podem ser lidas apenas um pequeno número de vezes, ou até mesmo apenas uma vez, destacando as limitações de computação e armazenamento, porém, enfatizando o fluxo contínuo com sua alta velocidade em ambientes dinâmicos.

Por sua vez, Isah, Zulkernine e Mcheick (2019) traz uma definição mais focada no framework do Apache Flink, que cria fluxos através de fila de mensagens e fontes de arquivos. Tratando o fluxo contínuo como um registro de dados e uma transformação como uma operação que recebe, como entrada, fluxos e devolve, como saída, fluxos processados. Todavia, Cugola e Margara (2012), abordam o problema de diversas fontes de dados e como produzir novos fluxos como saída, além de ver o problema como uma evolução do processamento de dados tradicional.

Além disso, para começar a entender sobre o tema de fluxo de dados contínuos, compreender os desafios e perspectivas, o artigo de Ounacer et al. (2017) fornece uma visão ampla sobre o tema, abordando questões práticas e teóricas, além de desenvolvimento e aplicação de soluções. Complementando, o trabalho de Wang et al. (2022) traz uma análise sobre Big Data para sistemas de manufatura inteligente, integrando com o processamento de dados em tempo quase real.

4.2 Conceitos Relacionados

Com base nas documentações técnicas e materiais de referência de plataformas amplamente utilizadas no mercado, como Amazon Web Services (AWS), Google Cloud e Confluent, apresenta-se a seguir uma breve relação dos principais conceitos relacionados ao processamento e análise de dados em fluxo (Amazon Web Services, 2024e; Amazon Web Services, 2024d; Confluent, 2024; Google Cloud, 2024).

At-least-once / At-most-once / Exactly-once: Modelos de entrega de mensagens em sistemas distribuídos. At-most-once significa que cada mensagem será entregue no máximo uma vez (podendo perder em falhas); at-least-once garante que todas sejam entregues ao menos uma vez (podendo duplicar em casos de repetição após falha); exactly-once assegura entrega única e sem perdas – obtido tipicamente combinando idempotência de produtor e transações de consumidor.

Lote (*Batch*): Conjunto de mensagens agrupadas para envio ou processamento conjunto.

Buffer de produtor: Área de memória reservada para armazenar mensagens que ainda não foram enviadas. Serve como uma espécie de fila temporária, absorvendo picos de geração de dados. Quando o buffer está cheio, novos envios podem ser bloqueados até que haja espaço disponível. Um buffer maior ajuda a evitar bloqueios, mas consome mais memória. Um buffer menor reduz o uso de recursos, mas pode esgotar rapidamente sob alta carga, aumentando o risco de atraso ou perda de dados.

Grupo de Consumidores: Conjunto de processos ou instâncias responsáveis por ler dados de forma compartilhada. Cada grupo tem um identificador em comum, e os dados são divididos entre os membros para que cada parte seja processada por apenas um deles. Isso permite que o sistema escale horizontalmente, aumentando a capacidade de leitura. Quando algum membro entra ou sai do grupo, as divisões são redistribuídas para manter o equilíbrio.

Consumidores de *Stream*: Componentes de software responsáveis por processar e analisar fluxos de dados que estão temporariamente armazenados. Eles podem realizar operações como agregações, filtragem e machine learning.

Data Lake: Armazenam todos os dados em seu formato bruto, permitindo uma flexibilidade maior para análises.

Data Warehouse: Um conjunto de dados orientado por assunto, integrado, não-volátil e variável com o tempo. Então são ótimos para consultas e relatórios específicos, já que armazenam os dados estruturados e processados.

Escalabilidade: Capacidade do sistema de lidar com o aumento de volume de dados, mantendo a eficiência e a consistência, mesmo durante picos de demanda, como em grandes eventos em redes sociais.

Evento: Um evento captura as mudanças e o estado de algo que acontece em um momento específico no tempo, sinalizando que algo aconteceu e contendo as informações sobre o ocorrido. Um evento pode ser gerado quando um usuário adiciona um produto no carrinho de um shopping virtual ou quando finaliza uma compra.

A Listagem 1 mostra um exemplo de evento em formato JSON. O campo speed representa a velocidade do pedestre em metros por segundo, enquanto direction indica a direção em graus (0 = Norte, 90 = Leste, etc.).

Código 1: Exemplo de evento representado em JSON

```
{
  "event_id": "12345",
  "timestamp": "2024-11-26T14:30:00Z",
  "pedestrian_id": "P001",
  "coordinate": "-22.9793, -43.2336",
  "speed": 1.5,
  "direction": 90
}
```

Latência: É o tempo entre o envio de uma solicitação e o recebimento da resposta. Reflete o atraso presente em qualquer sistema de comunicação ou processamento, e é um dos principais fatores que afetam a responsividade de aplicações em tempo quase real.

Latência de Cauda: É o tempo de resposta observado nos casos mais lentos de um sistema. Ela representa a parte final da distribuição de latência, onde estão os percentis mais altos, como P99 ou P99.9. Mesmo que a maioria das requisições seja rápida, uma cauda longa indica que uma pequena parte delas sofre atrasos significativos. Isso pode ser um problema em sistemas que exigem consistência de desempenho. Particionamento de Stream: Divisão de um fluxo de dados em várias partições, permitindo que diferentes consumidores processem partes do fluxo em paralelo, aumentando a eficiência e a velocidade do sistema.

Percentil (Px): Medida estatística indicando que x% das observações foram atendidas em um tempo menor ou igual ao valor de PX%. No contexto de desempenho, percentil 50 (mediana) é o tempo abaixo do qual metade das requisições foram respondidas; percentil 95 (P95) abrange a grande maioria (95%) das requisições, ignorando as 5% mais lentas; percentil 99 foca nas 1% mais lentas. Avaliar percentis é útil para entender a variabilidade de latência e

garantir SLAs – por exemplo, um sistema pode ter média de 100 ms mas P99 de 1 s, o que pode ser inadequado.

Processamento em Lote: Método de processamento de grandes volumes de dados em intervalos periódicos, geralmente fora dos horários de pico. Ideal para tarefas repetitivas e que não exigem respostas em tempo quase real.

Produtores de Stream: Componentes de software que coletam e transmitem dados em tempo quase real para o processador de stream. São responsáveis por enviar os registros com informações como nome do fluxo, valor dos dados e número de sequência.

Streaming: Dados de streaming são dados gerados por várias fontes, emitidos em alto volume de maneira contínua e incremental, em tempo quase real, com o objetivo de alcançar um processamento de baixa latência.

Throughput: Taxa de processamento ou transferência de dados, geralmente medida em número de mensagens por segundo ou volume em bytes/segundo. No Kafka, throughput de produção refere-se a quantos eventos o cluster consegue aceitar por unidade de tempo, enquanto throughput de consumo é quantos eventos podem ser lidos/processados por segundo.

4.3 Produtores de *Stream*

Os produtores de stream são responsáveis por coletar e transmitir dados em tempo quase real, alimentando o fluxo contínuo que será processado. Esses dados são particionados, permitindo que diferentes consumidores de stream os analisem em paralelo, aumentando a eficiência do sistema. A escalabilidade é um aspecto crucial, pois o sistema precisa lidar com volumes crescentes de dados, mantendo a consistência e o desempenho mesmo durante picos de demanda. A latência deve ser minimizada para garantir tempos de resposta rápidos em aplicações sensíveis, como jogos online ou análise de dados de redes sociais, enquanto o throughput mede a quantidade de dados transmitida com sucesso em um determinado período, indicando a eficiência da rede. Além disso, o processamento em lote é utilizado para lidar com grandes volumes de dados em intervalos definidos, ideal para tarefas que não requerem resposta imediata, complementando o streaming em um sistema completo e eficiente.

Essas definições fornecem uma visão concisa dos conceitos essenciais relacionados ao streaming e ao processamento de dados.

5
Soluções existentes para Streaming

Para este trabalho analisamos três dos principais nomes referentes ao tema de fluxo de dados, streaming, onde buscamos suas definições, abordagens e soluções: Google Cloud, AWS e Microsoft Azure. Foram analisados dados sobre suas estratégias e principais ferramentas, como cada uma das três abordagens diferem uma das outras.

A arquitetura do Google Cloud Pipelines (2024) possui as etapas de Data Ingestion, Processing e Storage and Analysis, conforme ilustrado na Figura 5.1.

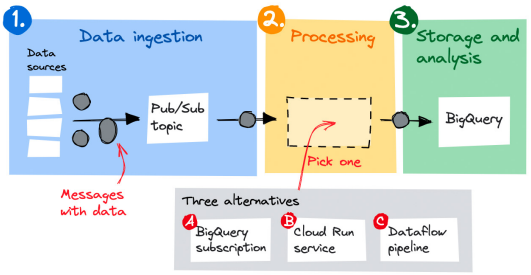

Figura 5.1: Diagrama da arquitetura de dados do Google Cloud, com etapas de ingestão, processamento e análise

Fonte: <https://cloud.google.com/blog/products/data-analytics/building-streaming-data-pipelines>

O responsável pela ingestão de dados é o Pub/Sub (2024) que recebe dados em forma de mensagens de múltiplas fontes e mantém em fluxo contínuo. O processamento é feito pelo Dataflow (2024) ("C" na figura 5.1), ferramenta que organiza os dados em intervalos de tempos específicos para análise em tempo quase real. É no dataflow em que os dados serão transformados, agregados e enriquecidos, esses dados podem ser enviados para múltiplos destinos, como o Storage (2024) ou BigQuery (2024), sendo flexível e escalável.

Na AWS (Amazon Web Services), a arquitetura é dividida em cinco camadas ,(Amazon Web Services, 2024c), segundo Figura 5.2.

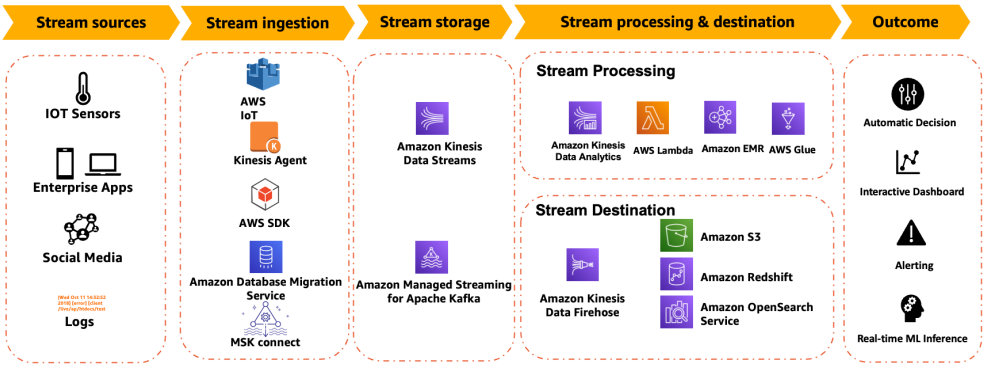

Figura 5.2: Arquitetura de dados em streaming da AWS, com etapas de ingestão, armazenamento, processamento e destinos

Fonte: <https://docs.aws.amazon.com/whitepapers/latest/build-modern-data-streaming-analytics-architectures/what-is-a-modern-streaming-data-architecture.html>

A primeira delas são as *Stream Sources*, as fontes de dados, como sensores IoT, redes sociais e logs. Os dados são coletados pela camada Stream Ingestion, responsável por coletar dados de milhares de fontes ao mesmo tempo em tempo quase real, onde temos componentes como o AWS IoT e *Kinesis Agent*. Esses dados são armazenados na etapa de *Stream Storage*, com as ferramentas (MSK) (2024) ou Kinesis (2024), garantindo sua escalabilidade, continuidade e ordenação. A penúltima etapa é a de *Stream Processing & Destination*, onde o processamento é feito através de ferramentas como o *Data Analytics*, e a destinação é feita por ferramentas como o *Amazon Data Firehose*. Por fim, a última etapa é a de Outcome, onde temos a saída desse processamento que pode ser enviado para dashboards interativos, por exemplo.

Por fim, a Microsoft Azure utiliza também uma arquitetura muito semelhante, dividida em algumas etapas (Microsoft Azure, 2024), de acordo com Figura 5.3.

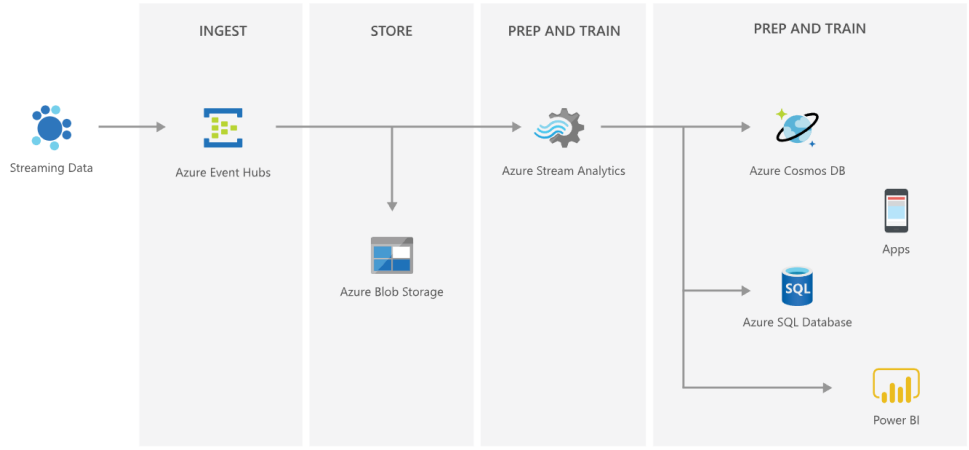

Figura 5.3: Arquitetura serverless de streaming de dados com Azure Event Hubs

Fonte: <https://azure.microsoft.com/pt-br/products/event-hubs/>

A primeira delas é a *Ingest*, onde temos o *Azure Event Hubs*, responsável pela ingestão de dados, ele utiliza partições para segmentar os dados. As partições permitem um paralelismo, onde cada partição pode ser lida em paralelo. A etapa de armazenamento, Store, é feita pelo Blob (2024). Em seguida temos a etapa de *Prep and Train*, dividida em duas partes, onde o Analytics (2024), um mecanismo de processamento de eventos, lê os fluxos de dados e faz o processamento dos dados, na primeira parte. Por fim, na segunda parte da etapa de *Prep and Train*, esses dados que são a saída do *Stream Analytics* vão para diversas fontes, como o PowerBI (2024) ou o Cosmos DB, que recebe os dados em formato JSON.

6
Modelo de Referência

Para o nosso projeto vamos trabalhar com o seguinte modelo de arquitetura para streaming de dados.

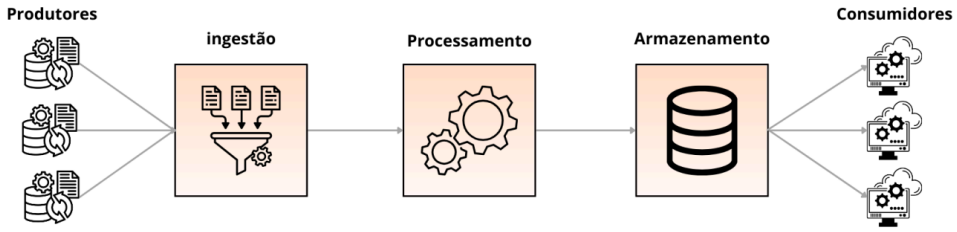

Figura 6.1: Modelo de referência com etapas de ingestão, processamento e armazenamento.

Fonte: elaborado pelo autor.

Essas são as três etapas principais. A cada uma delas, associam-se processos complementares que sustentam o modelo de streaming. A seguir, define-se de forma objetiva e concisa a responsabilidade de cada etapa:

1. **Ingestão:** Etapa responsável pela coleta e armazenamento dos dados. Um modelo de streaming trabalha recebendo dados de milhares de fontes ao mesmo tempo, logo precisamos de uma maneira eficaz de capturar todos esses dados que estão sendo gerados a todo momento, em diversos formatos e vindo de diversas fontes diferentes.

2. **Processamento:** Etapa designada para processar os dados coletados. Em diversos casos esses dados precisam ser tratados e reorganizados antes de serem enviados, essa é a etapa responsável por esse papel.

3. **Armazenamento:** A etapa final, onde os dados são enviados para diversos tipos de armazenamentos, possibilitando diversos tipos de análises a serem feitas em cima dos dados.

6.1 Ingestão

Agora que definimos cada uma das etapas de forma mais objetiva, podemos expandir um pouco mais cada uma delas. Em um modelo de streaming de dados, a ingestão começa com os *eventos*, ocorrências registradas de forma pontual no tempo, como a temperatura medida por um sensor ou a localização de um pedestre em determinado instante. Esses eventos representam capturas individuais, mas, na prática, o sistema trabalha com fluxos contínuos dessas ocorrências, formando o que chamamos de *streams*.

Uma stream pode representar, por exemplo, todas as leituras de um termômetro ao longo do dia ou o trajeto completo de um pedestre durante uma caminhada. A etapa de ingestão é responsável por capturar esses dados em tempo quase real, de múltiplas fontes simultaneamente, e encaminhá-los de forma eficiente ao restante do sistema, garantindo que o fluxo seja constante, ordenado e escalável. Essa entrada contínua é o que sustenta toda a estrutura de processamento subsequente.

6.2 Processamento

Agora vamos entrar na etapa de processamento dos dados, onde um *stream processor* realiza operações como filtrar, agregar, transformar e enriquecer os dados. O mecanismo de processamento recebe os dados do *message broker* e consegue identificar padrões ou anomalias nos dados em tempo quase real, acionando ações específicas com base nessas detecções.

Além disso, o mecanismo de processamento de *streams* também é capaz de suportar o *Processamento de Eventos Complexos* (*Complex Event Processing* – CEP), que consiste basicamente na capacidade de detectar padrões ou anomalias nos dados em *streaming*, além de acionar soluções correspondentes. Ao analisar sequências de eventos em conjunto, o sistema consegue identificar fraudes, falhas em equipamentos ou tendências emergentes no mercado. O CEP permite que possamos reagir de forma proativa a situações em tempo quase real, permitindo uma tomada de decisão mais ágil e precisa.

6.3 Armazenamento

Nesta etapa, os dados resultantes do processamento são armazenados conforme as necessidades da aplicação, podendo assumir formas estruturadas ou não estruturadas, e serem mantidos de maneira temporária ou persistente. Dependendo do tipo de análise a ser feita, exploratória, histórica ou em

tempo quase real, os dados podem ser enviados para diferentes destinos, como *data lakes*, *data warehouses* ou *lakehouses*, otimizando o custo, performance e flexibilidade do ambiente.

6.4 Message Broker

Agora que entendemos o formato dos dados utilizados em um fluxo de dados, além do funcionamento de cada etapa, podemos refletir sobre como o mecanismo de processamento lida com o grande volume de dados sendo continuamente enviado. Para isso, é necessário um componente intermediário: o *message broker*.

O *message broker* é uma ferramenta responsável por intermediar a comunicação entre os produtores de dados e os consumidores, recebendo, organizando e disponibilizando as mensagens geradas em tempo quase real. Seu papel é desacoplar os produtores dos consumidores, permitindo que os dados sejam armazenados temporariamente e entregues de forma controlada aos processadores. O principal padrão utilizado nesse tipo de arquitetura é o modelo de publicação/assinatura (*publish/subscribe*, ou *pub/sub*), em que os produtores publicam mensagens em tópicos, e os consumidores assinam esses tópicos para receber os dados.

Dessa forma, o *message broker* atua em conjunto com as etapas de ingestão e processamento, garantindo que os dados oriundos de múltiplas fontes possam ser encaminhados de forma eficiente e confiável para o mecanismo de processamento.

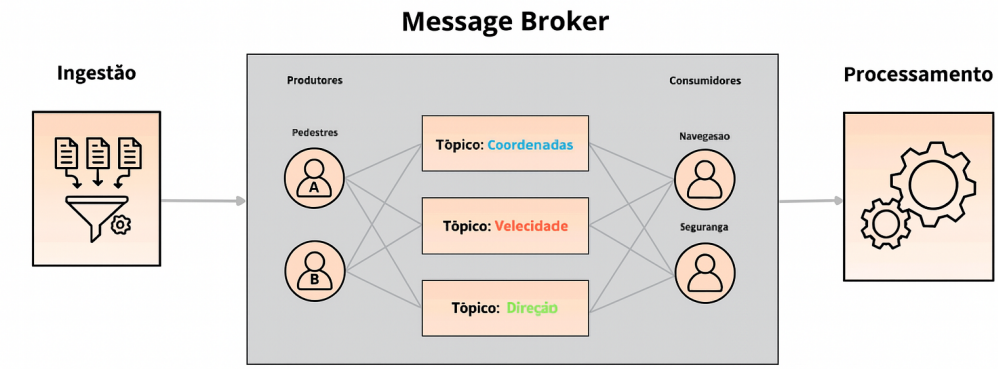

Figura 6.2: Exemplo de intermediação entre produtores e consumidores por meio de um message broker, com divisão de tópicos e múltiplos assinantes.

Fonte: Elaborado pelo autor.

O modelo de publicação e assinatura funciona através de tópicos onde o produtor publica a mensagem em um tópico e os consumidores se inscrevem nos tópicos que estão interessados em receber. Vale ressaltar que as mensagens que estão sendo enviadas pelos produtores são eventos.

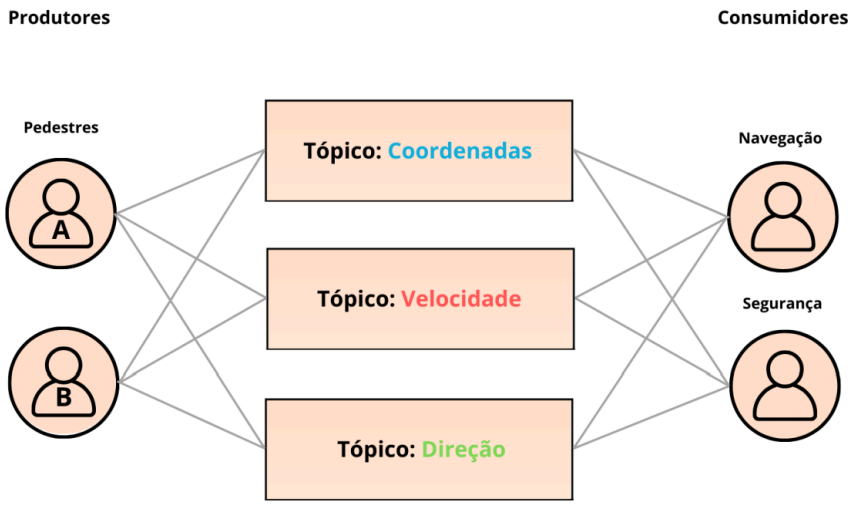

Figura 6.3: Exemplo de *pub/sub* com tópicos

Fonte: Elaborado pelo autor.

Isso permite a criação de um sistema **desacoplado**, no qual produtores

e consumidores não interagem diretamente, mas se comunicam exclusivamente por meio do *message broker*. Dessa forma, cada componente pode ser desenvolvido, atualizado ou substituído de forma independente, sem afetar os demais, o que facilita tanto a manutenção quanto a escalabilidade da arquitetura. Para garantir desempenho e paralelismo, o sistema divide os tópicos em **partições**, que funcionam como segmentos independentes de dados. Cada produtor pode enviar mensagens para diferentes partições de um mesmo tópico, geralmente associadas por uma chave de particionamento. Essa estratégia permite que múltiplos produtores enviem dados simultaneamente sobre o mesmo tema (tópico), mesmo que sejam fontes distintas, otimizando a distribuição e o consumo dos dados em tempo quase real pelo *stream processor*.

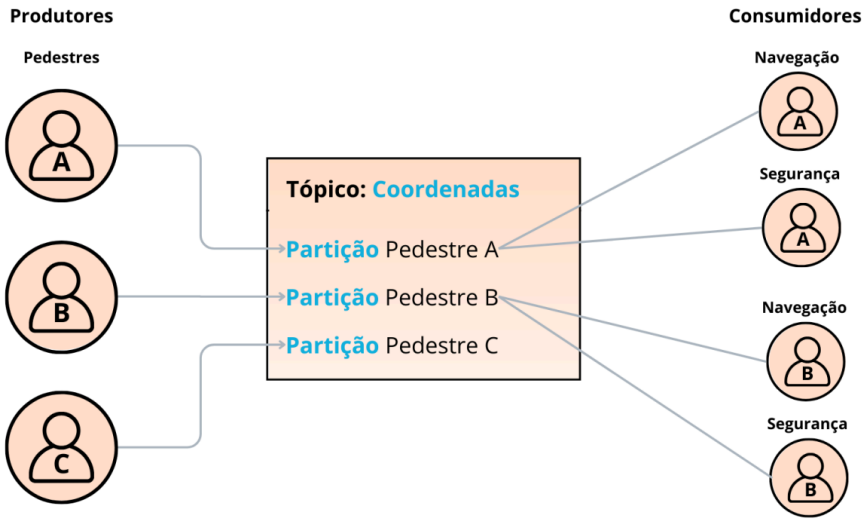

Figura 6.4: Exemplo de *pub/sub* com tópicos e partições

Fonte: Elaborado pelo autor.

Portanto, os serviços consumidores podem se inscrever nos tópicos de interesse e receber dados apenas das partições atribuídas a eles. Caso um consumidor falhe, outro pode assumir sua partição, garantindo continuidade no fluxo de dados. Em cenários distribuídos, como operações internacionais, o sistema pode ser escalado horizontalmente por meio de múltiplos *clusters* de servidores. Isso permite que diferentes partições de um mesmo tópico estejam fisicamente distribuídas em servidores localizados em diferentes regiões, o que melhora a disponibilidade e reduz a latência para aplicações geograficamente dispersas.

Como o *message broker* atua como elo entre os dados ingeridos e o mecanismo de processamento, a Figura 6.5 ilustra essa relação. Nela, observa-se que os dados passam pela camada de processamento antes de seguirem para

o armazenamento, e só então são acessados por aplicações consumidoras que dependem dos resultados já tratados. É importante destacar, no entanto, que essa arquitetura pode variar de acordo com o sistema implementado, e existem modelos nos quais o consumo ocorre diretamente do *message broker*, com ou sem processamento intermediário, dependendo do caso de uso.

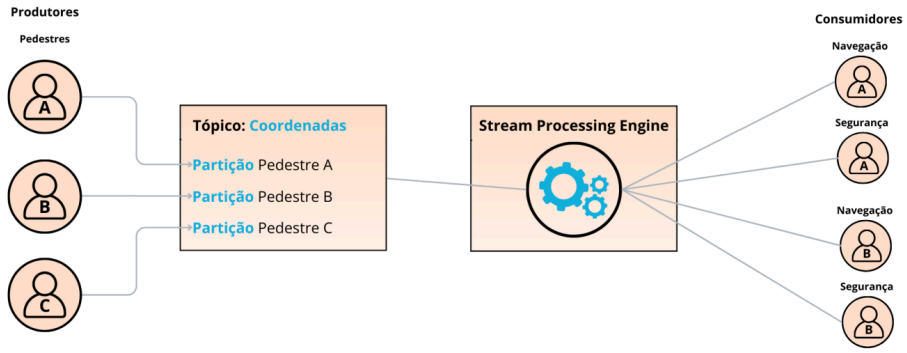

Figura 6.5: Exemplo de *Message Broker* com mecanismo de processamento

Fonte: Elaborado pelo autor.

Além de mediar a comunicação, o *message broker* também pode exercer um papel de armazenamento intermediário, retendo as mensagens por um tempo configurável antes de disponibilizá-las para os consumidores. Esse tipo de retenção, no entanto, é geralmente limitado e voltado para uso temporário, já que o armazenamento prolongado de dados deve ser delegado a sistemas especializados, como *data lakes* ou *warehouses*, de acordo com a finalidade analítica ou operacional dos dados.

7
Kafka

O Apache Kafka é adotado neste trabalho como a solução de referência para implementar o modelo de arquitetura de streaming de dados descrito na seção anterior. Com base nesse modelo, o Kafka é utilizado para desempenhar, de forma integrada, os papéis de ingestão de eventos, armazenamento transitório, intermediação entre produtores e consumidores e, em parte, processamento de dados. Esta seção detalha como o Kafka se insere nas quatro etapas do modelo (ingestão, processamento, armazenamento e mediação por message broker), além de apresentar dois componentes adicionais que garantem o funcionamento e a avaliação do sistema: o Apache ZooKeeper (Apache Software Foundation, 2025a), responsável pela coordenação do cluster Kafka, e a ferramenta de benchmark kafka-producer-perf-test (perf_test), disponibilizada pela Confluent (TESTING, 2024) para medições de desempenho.

Tabela 1 - Parâmetros de configuração do produtor Kafka

Tabela 7.1: Parâmetros de configuração do produtor Kafka. Fonte: elaborado pelo autor.

Parâmetro	**Descrição**
`batch.size`	Define o tamanho máximo (em bytes) de um lote de mensagens antes do envio. Lotes maiores podem aumentar o throughput, pois permitem o envio mais eficiente de grandes volumes de dados. No entanto, isso pode introduzir mais latência se o lote demorar a ser preenchido.
`linger.ms`	Define o tempo máximo (em milissegundos) que o produtor aguarda antes de enviar um lote incompleto. Valores maiores favorecem o agrupamento de mensagens, otimizando o throughput. Valores baixos reduzem a latência.
`acks`	Determina quantas réplicas devem confirmar o recebimento da mensagem. Pode ser 0, 1 ou `all`. Valores maiores aumentam a durabilidade e confiabilidade dos dados, mas reduzem o throughput.
`throughput`	Mede a taxa de envio de mensagens por segundo pelo produtor Kafka. Pode ser controlado durante os testes com ferramentas como o `perf_test`, e afeta diretamente a carga sobre o broker e o tempo total de ingestão.

7.1 Ingestão com Kafka

No modelo de referência implementado, o Kafka atua como ponto de entrada dos dados. Aplicativos ou serviços que produzem mensagens (os chamados produtores Kafka) publicam dados em tópicos. Essa publicação pode ser otimizada por técnicas como envio assíncrono em lote (batching) e compressão de mensagens. Em nossos testes, configuramos alguns dos principais parâmetros do produtor com o objetivo de investigar seus impactos sobre latência e throughput de ingestão. Os três parâmetros mais relevantes utilizados foram:

– **batch.size**: define o tamanho máximo (em bytes) de um lote de mensagens antes do envio. Lotes maiores podem melhorar o throughput ao custo de mais latência.

- **linger.ms**: define o tempo máximo (em milissegundos) que o produtor espera antes de enviar um lote incompleto. Um valor igual a zero reduz a latência, enquanto valores maiores favorecem a eficiência do envio.
- **acks**: determina quantos servidores precisam confirmar o recebimento da mensagem para que ela seja considerada entregue. Utilizamos `acks=1`, o que garante confirmação do líder da partição.

Também mencionamos, por completude, o Kafka Connect (CONECT, 2024), ferramenta oficial para integração de fontes externas de dados ao Kafka. Embora não tenha sido utilizada em nossos experimentos, é importante destacar que, em ambientes produtivos, ela é amplamente adotada para capturar dados de bancos de dados, filas e APIs, viabilizando ingestão em tempo quase real.

7.2 Processamento com Kafka

No modelo de referência adotado, a etapa de processamento é responsável por aplicar transformações nos dados em tempo quase real, como filtragens, agregações, enriquecimentos e, em casos mais elaborados, a identificação de padrões ao longo do tempo. O Apache Kafka oferece suporte a essas operações principalmente por meio de duas ferramentas: o *Kafka Streams* e o *ksqlDB*.

O *Kafka Streams* é uma biblioteca oficial da plataforma Kafka que permite construir aplicações de processamento de fluxos utilizando código em Java (Apache Software Foundation, 2025b). Ela possibilita que o desenvolvedor leia dados diretamente de tópicos Kafka, aplique transformações nesses dados e, em seguida, envie os resultados processados para novos tópicos. Entre as operações disponíveis estão mapeamento, filtragem e agrupamentos simples. Esses recursos viabilizam a criação de aplicações capazes de detectar, por exemplo, comportamentos fora do padrão em sensores ou alterações inesperadas em sistemas de monitoramento.

Já o *ksqlDB* fornece uma interface baseada em linguagem SQL que facilita o trabalho de quem não deseja escrever código em linguagens de programação. Por meio de comandos semelhantes aos utilizados em bancos de dados, é possível definir transformações contínuas sobre os dados em fluxo, permitindo a criação de pipelines de processamento de forma mais intuitiva (Confluent, 2025).

Além dessas ferramentas nativas, o Kafka também permite a integração com soluções externas voltadas a processamento de fluxo de dados, como o Apache Flink ou o Apache Spark Streaming. Essas plataformas são capazes

de consumir dados dos tópicos Kafka, realizar operações mais complexas de detecção de padrões e, posteriormente, encaminhar os resultados processados para novos tópicos ou sistemas de armazenamento.

Portanto, no contexto deste trabalho, o Kafka não se limita à função de transporte de eventos: ele também fornece os mecanismos necessários para processar os dados em tempo quase real, garantindo que as informações ingeridas possam ser transformadas antes de avançarem para as etapas de armazenamento ou análise.

7.3 Armazenamento

No modelo de referência implementado, os dados processados podem ser encaminhados para sistemas de armazenamento com diferentes finalidades, dependendo do tipo de análise a ser realizada. Para análises em tempo quase real ou de curta duração, os dados podem ser mantidos temporariamente em cache ou armazenados em sistemas intermediários. Já para análises históricas, consolidações e relatórios, eles são direcionados a sistemas permanentes, como bancos relacionais, bancos NoSQL, Data Lakes, Data Warehouses ou diretórios de arquivos.

O Apache Kafka atende a essa demanda de duas formas complementares. Primeiramente, ele oferece uma estrutura interna de armazenamento temporário: os dados são gravados em disco dentro de cada partição de tópico, permitindo que os consumidores acessem as mensagens com base em um número sequencial chamado *offset*. Essa organização garante que os dados fiquem disponíveis mesmo que o consumidor ainda não tenha lido a mensagem. Os administradores do sistema podem configurar por quanto tempo as mensagens devem ser mantidas,por exemplo, sete dias, ou até que tamanho máximo de armazenamento seja atingido. Após esses limites, os dados antigos são removidos automaticamente. Essa abordagem faz com que o Kafka funcione como um "buffer" confiável, capaz de reter os dados por tempo suficiente para lidar com falhas temporárias ou reprocessamentos.

Além disso, o Kafka também se integra com sistemas externos de armazenamento por meio de uma ferramenta chamada Kafka Connect (CONECT, 2024). Com ela, é possível enviar os dados de um tópico diretamente para plataformas como Hadoop HDFS, Amazon S3, Snowflake, BigQuery, bancos relacionais ou Elasticsearch. Esses chamados *sink connectors* automatizam o envio dos eventos, garantindo que os dados cheguem a destinos permanentes e fiquem disponíveis para análises mais detalhadas e de longo prazo.

Dessa forma, o Kafka não apenas transporta e processa dados em tempo

quase real, mas também oferece suporte à etapa de armazenamento, tanto temporário quanto permanente, cumprindo assim todas as etapas previstas no modelo de referência adotado.

7.4 Message Broker

A camada de message broker do modelo de referência é responsável por intermediar a comunicação entre produtores e consumidores, seguindo o padrão de publicação/assinatura (pub/sub). No Kafka, essa funcionalidade é realizada por uma arquitetura distribuída baseada em brokers, tópicos e partições.

Cada broker Kafka gerencia partições de diferentes tópicos. Quando um produtor publica uma mensagem em um tópico, essa mensagem é atribuída a uma das partições disponíveis, podendo ser feita com base em uma chave fornecida ou por balanceamento automático. Cada partição possui um broker líder, que recebe as gravações, e réplicas em outros brokers para garantir tolerância a falhas.

Os consumidores, por sua vez, são organizados em grupos de consumo. Cada partição de um tópico é atribuída a apenas um consumidor por grupo, permitindo que o trabalho de leitura seja distribuído entre diferentes consumidores. Caso um consumidor falhe, o Kafka redistribui automaticamente as partições, garantindo continuidade na leitura.

Esse mecanismo de particionamento, replicação e reequilíbrio de consumidores proporciona ao Kafka escalabilidade horizontal e alta disponibilidade, características fundamentais para sistemas de transmissão de dados em larga escala. O parâmetro `acks`, utilizado na configuração do produtor, permite controlar o nível de garantia exigido para considerar uma mensagem como entregue, ajustando o equilíbrio entre latência e durabilidade. Dessa forma, o Kafka oferece uma comunicação eficiente, robusta e desacoplada entre componentes do sistema, como previsto no modelo de referência.

7.5 Kafka perf_test

A ferramenta `kafka-producer-perf-test`, também chamada de `perf_test`, foi utilizada neste trabalho para mensurar, de forma controlada, o desempenho de produção de eventos no Kafka (TESTING, 2024). Trata-se de uma ferramenta oficial que simula a atividade de um produtor Kafka, permitindo definir parâmetros como a quantidade total de mensagens a serem enviadas, o tamanho de cada mensagem, a taxa de envio (throughput)

e diversas propriedades de configuração do produtor, como `acks`, `linger.ms`, `batch.size` e `compression.type`.

Durante a execução, a ferramenta se conecta ao cluster Kafka por meio do parâmetro `bootstrap.servers` e inicia o envio de mensagens para o tópico especificado. As mensagens são agrupadas em lotes, conforme os parâmetros definidos, para simular uma carga realista. A cada segundo, o `perf_test` exibe métricas parciais, como a taxa de mensagens enviadas por segundo (em registros por segundo), o throughput de envio em megabytes por segundo (MB/s), e indicadores de latência, como média, máximo e percentis (50º, 95º, 99º e 99,9º).

Essas medições nos permitiram observar o comportamento do cluster Kafka em diferentes configurações e identificar, por exemplo, até que ponto o sistema conseguia manter baixas latências antes de começar a apresentar sinais de sobrecarga. Também foi possível analisar o impacto de diferentes estratégias de confirmação de entrega, como `acks=1` (confirmação apenas do broker líder) em comparação com `acks=all` (confirmação por todas as réplicas), assim como os efeitos do uso de compressão e do tamanho dos lotes no desempenho do sistema.

Em resumo, o uso do `perf_test` foi essencial para testar o limite de ingestão do Kafka com segurança, permitindo ajustes finos nas configurações antes da simulação de cenários reais de carga.

7.6 ZooKeeper e Coordenação do Cluster

O Apache ZooKeeper é utilizado pelo Kafka para coordenar o funcionamento do cluster, garantindo que todos os componentes tenham uma visão atualizada da estrutura do sistema. Ele mantém informações essenciais, como quais brokers estão ativos, quais tópicos estão registrados e quem é o líder de cada partição (Apache Software Foundation, 2025a).

Essas informações são organizadas em uma estrutura hierárquica chamada de znodes, mantida em memória pelos servidores ZooKeeper. Sempre que um produtor ou consumidor precisa interagir com um tópico Kafka, ele consulta essas informações para saber qual broker está responsável por cada partição. Isso evita tentativas de leitura ou escrita em nós que possam estar indisponíveis ou fora de sincronia.

Além disso, o ZooKeeper monitora continuamente o estado dos brokers. Caso algum deles falhe, ele promove automaticamente outra réplica da partição afetada para o papel de líder. Essa transição ocorre de forma rápida e transparente, permitindo que a operação do sistema continue sem necessidade

de reinicializações.

É importante destacar que o ZooKeeper não participa do armazenamento ou do processamento de mensagens. Sua função principal é fornecer as informações de controle e manter o sistema coordenado e tolerante a falhas. Em outras palavras, ele funciona como uma "central de controle"que assegura a estabilidade da comunicação entre produtores, brokers e consumidores no Kafka.

7.7 Especificidades Técnicas do Kafka

Apesar da existência de outros sistemas de mensageria no mercado, como RabbitMQ, ActiveMQ e Amazon MQ, o Apache Kafka se destaca por sua arquitetura voltada ao tratamento de grandes volumes de dados em tempo quase real. Nesta seção, discutimos duas características técnicas que ajudam a explicar esse desempenho: o modelo de armazenamento e o controle de fluxo do consumidor.

7.7.1 Armazenamento Baseado em Log

O Kafka organiza os dados de cada tópico como um log sequencial — ou seja, uma lista ordenada de mensagens, armazenadas uma após a outra, sem alterações retroativas. Cada mensagem recebe um identificador único chamado offset, que representa sua posição dentro da partição. Esse modelo oferece duas vantagens principais: (i) permite que diferentes consumidores leiam os dados no seu próprio ritmo, sem interferir uns nos outros, e (ii) possibilita a releitura de mensagens antigas (replay), algo útil em casos de reprocessamento ou recuperação após falhas.

Além disso, o Kafka permite configurar como os dados serão mantidos em disco. É possível, por exemplo, definir que as mensagens fiquem disponíveis por um tempo determinado (como sete dias), ou até que o log atinja um certo tamanho (como 500 GB). Após esse limite, os dados mais antigos são descartados automaticamente, evitando que o sistema ocupe espaço indefinidamente.

Outra funcionalidade importante é a compaction, ou compactação de log: nesse modo, o Kafka mantém apenas a última mensagem registrada para cada chave de dado. Isso é útil em situações em que o estado atual de um elemento (como o perfil de um usuário) é mais importante do que o histórico completo de mudanças.

Por fim, o Kafka permite configurar o nível de garantia na entrega das mensagens. Três modos estão disponíveis:

At-most-once: a mensagem é enviada uma única vez, sem confirmação do broker. É mais rápido, mas pode haver perda em caso de falha.

At-least-once: a mensagem é reenviada até o broker confirmar o recebimento. Garante que nada se perca, mas pode haver duplicações.

Exactly-once: combinação de recursos para garantir que cada mensagem seja processada uma única vez, sem perdas nem repetições. Esse modo é mais complexo e exige configurações específicas.

7.7.2 Controle de Fluxo do Consumidor

No Kafka, os consumidores não recebem os dados automaticamente. Em vez disso, cada aplicação chama periodicamente o método poll() para buscar novas mensagens (FOUNDATION, 2020). Isso significa que o consumidor tem controle direto sobre o ritmo em que processa os dados. Se ele estiver lento ou desatento (por exemplo, chamando poll() com grandes intervalos), pode acabar acumulando atraso ou até sendo desconectado do grupo, gerando reconfigurações indesejadas.

Para lidar com isso, o Kafka oferece parâmetros que permitem ajustar esse comportamento. Por exemplo:

max.poll.interval.ms define o tempo máximo entre chamadas de poll(). Se esse tempo for excedido, o Kafka assume que o consumidor está inativo.

max.poll.records limita a quantidade de mensagens retornadas por chamada, ajudando a evitar sobrecargas.

Além do ritmo de leitura, outro ponto importante é como o consumidor informa ao Kafka que já processou as mensagens — isso é feito por meio do commit de offsets. O offset é como um marcador que indica até qual mensagem o consumidor já leu em cada partição. Por padrão, o Kafka realiza esses commits automaticamente a cada 5 segundos, o que facilita o uso, mas pode levar a situações em que mensagens já processadas sejam lidas novamente após uma falha (duplicação), ou em que mensagens lidas, mas ainda não processadas, sejam consideradas como entregues (perda) (FOUNDATION, 2020).

Para evitar esses problemas, o desenvolvedor pode desativar o commit automático e realizar os commits manualmente, escolhendo o momento mais seguro para registrar o avanço no consumo. Isso pode ser feito de forma síncrona, aguardando a confirmação do broker (mais confiável, porém mais lento), ou assíncrona, onde o commit é enviado em segundo plano (mais rápido, com pequeno risco de perda em caso de falhas súbitas). Essa estratégia

manual oferece maior controle sobre a confiabilidade, mas exige mais cuidado na implementação para evitar inconsistências.

Por fim, o Kafka também permite configurar o volume de dados que o broker entrega por requisição. Parâmetros como fetch.min.bytes e fetch.max.wait.ms definem quanto o broker deve aguardar e acumular antes de enviar os dados ao consumidor. Ajustar esses valores permite equilibrar entre throughput e latência. Por exemplo, em sistemas sensíveis ao tempo, pode-se preferir receber poucos dados rapidamente; já em pipelines que priorizam eficiência, é melhor acumular lotes maiores.

Em resumo, o Kafka fornece um controle fino sobre o fluxo de dados, permitindo ao consumidor definir tanto o ritmo de leitura quanto a forma de confirmação e volume de dados por requisição. Isso evita sobrecargas, garante robustez e contribui para a eficiência do sistema.

7.7.3 Armazenamento em Bancos Relacionais

Quando um sistema consome dados em fluxo contínuo, é comum que esses dados precisem ser salvos em um banco relacional, como o PostgreSQL ou MySQL. Nesse contexto, cada evento do fluxo pode representar uma nova linha na tabela, ou uma atualização de algum registro já existente. Para isso funcionar bem, é importante que o consumidor controle a frequência e o volume dessas inserções, evitando enviar uma requisição para o banco a cada nova mensagem. Muitas vezes, os dados são agrupados temporariamente antes de serem inseridos, o que ajuda a reduzir o uso de recursos e melhora o desempenho. Além disso, como o fluxo de eventos não tem fim definido, o processo de escrita precisa ser pensado como algo contínuo e resiliente, com mecanismos para lidar com quedas ou falhas inesperadas.

No caso do Kafka, esse tipo de integração pode ser feito diretamente pelo consumidor, como descrito anteriormente, ou por ferramentas que já oferecem esse suporte embutido. Um exemplo é o ksqlDB, uma extensão do próprio Kafka que permite consultar os dados dos tópicos como se fossem tabelas. Com ele, é possível criar regras que transformam os eventos em registros estruturados e escrevem os resultados em bancos relacionais de forma automática. Embora o projeto tenha focado na etapa de ingestão, esse tipo de solução mostra como o Kafka pode se conectar a sistemas tradicionais, permitindo que dados em fluxo contínuo também sejam armazenados em estruturas mais estáveis e conhecidas.

8
Testes

Nesta seção, descrevemos o ambiente de testes, a metodologia de execução dos experimentos de performance e analisamos os resultados obtidos em diferentes cenários. O objetivo é avaliar como o Apache Kafka se comporta ao lidar com rajadas de dados, variando o tamanho das mensagens e alguns parâmetros de configuração, e verificar se a arquitetura montada atinge os requisitos de baixa latência e alta vazão propostos.

8.1
Modelo Simples com Um Único Produtor

Os testes realizados neste trabalho utilizaram apenas um produtor Kafka, como forma de manter o modelo experimental mais simples e controlado. Essa escolha foi intencional: por se tratar de uma primeira exploração prática da tecnologia, buscou-se começar com uma estrutura mínima que permitisse observar com clareza o comportamento do sistema frente a variações de carga. Em vez de simular múltiplos produtores concorrentes, concentrou-se em um único fluxo contínuo, com rajadas periódicas de dados configuradas ao longo do tempo. Isso possibilitou avaliar como o Kafka reagia a picos de throughput em um cenário isolado, sem interferência de paralelismo ou coordenação entre instâncias produtoras. A abordagem favoreceu a identificação de padrões de latência e de uso de buffer com mais precisão, criando uma base confiável para futuras extensões do experimento.

8.2
Objetivos dos Testes

Os testes foram realizados com o objetivo de entender como o Kafka se comporta diante de picos de carga — situações em que há um grande volume de mensagens sendo enviadas em um curto intervalo de tempo. O foco principal foi verificar se a arquitetura escolhida consegue manter o desempenho e a estabilidade, mesmo em cenários mais exigentes.

Especificamente, buscamos observar:

– A latência média de entrega das mensagens, assim como os percentis de latência (P50, P95 e P99), que indicam se existem atrasos pontuais ou se o tempo de resposta permanece estável ao longo do teste.

- O *throughput* (taxa média de envio, em registros por segundo) alcançado em cada cenário, considerando diferentes tamanhos de mensagem e configurações de envio.
- O efeito de parâmetros como `batch.size` e `buffer.memory` no desempenho do produtor Kafka, especialmente em cenários otimizados e não otimizados para mensagens de 1 kB.
- Se o Kafka consegue manter a ordem e consistência das mensagens mesmo durante picos de envio, sem perdas ou reordenações inesperadas.

Esses objetivos estão conectados à proposta geral do trabalho, que é avaliar se o uso do Kafka, da forma como foi configurado, é adequado para lidar com dados em fluxo contínuo e com baixa latência.

8.3 Ambiente

Os testes foram executados em um ambiente controlado, criado com contêineres Docker. O cenário incluiu:

- Um contêiner com o ZooKeeper (`confluentinc/cp-zookeeper:latest`), responsável pela coordenação do cluster Kafka.
- Um contêiner com o broker Kafka (`confluentinc/cp-kafka:latest`), configurado com apenas uma partição e fator de replicação igual a 1, o que simplifica a análise e permite isolar o comportamento do sistema.
- Scripts de teste executados diretamente no contêiner do broker, utilizando a ferramenta `kafka-producer-perf-test`. Isso garantiu que a geração de mensagens e a coleta das métricas ocorressem no mesmo ambiente, evitando interferências externas.

Nos testes, foram utilizadas mensagens com dois tamanhos distintos: 100 bytes e 1 kilobyte. Para cada um, foram ajustados parâmetros como `batch.size` e `buffer.memory` de forma proporcional, com o objetivo de avaliar como essas configurações impactam no desempenho.

A compressão escolhida foi a LZ4 (`compression.type=lz4`), por ser leve e eficiente, contribuindo para reduzir o tráfego de rede sem adicionar muita sobrecarga.

Além disso, o parâmetro `acks` foi mantido em 1, o que significa que o produtor considera a mensagem entregue assim que o broker líder confirma a gravação local. Isso oferece um bom equilíbrio entre desempenho e confiabilidade básica.

O ambiente foi configurado com o objetivo de minimizar interferências externas, buscando observar com mais clareza o comportamento do Kafka nas condições definidas.

8.4 Execução dos Testes

Para executar os testes, foram criados dois scripts em Bash, um para mensagens de 100 B e outro para mensagens de 1 kB, executados dentro do contêiner Kafka com o comando `docker exec`. Cada script definiu uma taxa de produção base: 250 mil mensagens por segundo para 100 B e 80 mil mensagens por segundo para 1 kB. Essas taxas foram ajustadas de forma que, em condições estáveis, a latência média observada ficasse próxima de 1 ms — valor de referência utilizado como patamar desejado de desempenho.

Logo no início de cada script, um trecho em Python gerava dez rajadas pseudo-aleatórias. Cada rajada era composta por um instante de início, uma duração (entre 5 e 15 segundos) e um fator de multiplicação de throughput (entre 2× e 6×). O uso da função `random.seed(42)` garantiu que as rajadas se repetissem exatamente nas três execuções de cada cenário, permitindo comparabilidade direta dos resultados.

Durante a simulação, cada segundo de teste representava uma iteração no loop principal. O script determinava o throughput desejado para aquele instante (multiplicado em caso de rajada) e o aplicava via chamada ao utilitário `kafka-producer-perf-test`, conforme abaixo:

Código 2: Chamada ao utilitário kafka-producer-perf-test

```
kafka-producer-perf-test \
  --topic <TOPIC> \
  --num-records <throughput> \
  --record-size <MSG_SIZE> \
  --throughput <throughput> \
  --producer-props \
    bootstrap.servers=broker:9092 \
    acks=1 \
    linger.ms=0 \
    batch.size=... \
    compression.type=lz4 \
    buffer.memory=... \
    max.in.flight.requests.per.connection=1
```

Para garantir repetibilidade, o tópico era esvaziado e recriado a cada execução. Os parâmetros específicos para cada cenário foram:

– **Cenário 1 (100 B)**: `batch.size = 32 kB`, `buffer.memory = 32 MB`.

- **Cenário 2 (1 kB - padrão)**: `batch.size = 64 kB`, `buffer.memory = 64 MB`.
- **Cenário 3 (1 kB - reduzido)**: `batch.size = 32 kB`, `buffer.memory = 32 MB`.

Durante os testes, o utilitário `kafka-producer-perf-test` reportava, a cada segundo, métricas como throughput instantâneo (em registros/s e MB/s), latência média, latência máxima e percentis de latência (P50, P95 e P99). Esses dados foram redirecionados para arquivos de log com 600 amostras cada (uma por segundo), totalizando 1800 amostras por cenário após três execuções consecutivas.

Nota sobre visualização: os gráficos gerados com base nesses logs estão apresentados em uma seção específica adiante, visando facilitar a comparação entre os testes. Ao longo desta seção, os gráficos serão referenciados pontualmente conforme a análise de cada cenário.

Antes de avançarmos com a análise, vale esclarecer os conceitos utilizados:

- **P50**: indica a mediana da latência — metade das mensagens são processadas abaixo desse valor.
- **P95 e P99**: representam os valores de latência abaixo dos quais 95% e 99% das mensagens foram processadas, respectivamente. Esses percentis ajudam a avaliar a robustez do sistema em relação a atrasos esporádicos (latência de cauda).

A Tabela 8.1 resume os principais resultados médios obtidos em cada cenário:

Tabela 8.1: Resumo dos testes de desempenho por métrica e cenário. Fonte: elaborado pelo autor.

Métrica	**100 B**	**1 kB (padrão)**	**1 kB (reduzido)**
Latência média (ms)	163,9	311,7	449,2
P50 (ms)	154,7	309,5	456,6
P95 (ms)	316,3	499,9	538,0
P99 (ms)	431,0	552,0	580,6
Throughput (msg/s)	196113	49858	40518

8.4.1 Evidência de Entrega Bem-Sucedida

Apesar do foco principal estar na ingestão, foi importante garantir que todas as mensagens geradas durante os testes chegaram corretamente ao broker. Para isso, utilizou-se o utilitário kafka-producer-perf-test, que reporta apenas as mensagens efetivamente entregues — ou seja, cada linha de log só é registrada se a quantidade completa de mensagens especificada for aceita pelo Kafka. Nos testes com mensagens de 100 B, por exemplo, cada linha de execução apresenta exatamente "250000 records sent", valor que se repete ao longo de todas as rajadas e períodos de carga base. Da mesma forma, no cenário com mensagens de 1 KB (configuração reduzida), cada linha dos logs mostra "80000 records sent", correspondendo exatamente ao volume configurado para aquele segundo. Em nenhum momento os logs apresentaram falhas, mensagens perdidas ou variações nos totais esperados.

Além disso, os testes foram executados em ambiente controlado: o broker era reiniciado e o tópico recriado antes de cada rodada, garantindo um início limpo e previsível. O parâmetro acks=1 assegurava que o produtor só considerava a mensagem enviada após receber confirmação do broker, e o perf-test encerrava a execução de cada segundo apenas após completar o envio do total planejado. Com isso, mesmo sem um consumidor ativo para validar a leitura, foi possível comprovar que todas as mensagens foram aceitas e armazenadas no Kafka, sem perdas na etapa de envio.

8.4.2 Repetições para Redução de Ruído

Cada cenário de teste foi executado três vezes com as mesmas configurações, justamente para reduzir o impacto de variações pontuais no comportamento do sistema. Como o ambiente de teste envolve chamadas de sistema, acesso ao disco e outras operações assíncronas, pode haver pequenas flutuações no desempenho a cada execução, mesmo com carga idêntica. Ao repetir os testes, foi possível observar a consistência dos resultados e identificar possíveis desvios fora do padrão esperado. Além disso, as repetições permitiram calcular médias mais representativas de latência e throughput, desconsiderando efeitos isolados causados por eventuais interferências do sistema operacional ou do Docker durante uma execução específica.

8.5 Análise dos Resultados

8.5.1 Comparação: 100 B vs. 1 kB (configuração padrão)

Nos dois primeiros cenários, comparamos mensagens pequenas (100 B) com mensagens maiores (1 kB), mantendo todas as outras configurações iguais: *acks=1*, *linger.ms=0*, compressão LZ4, mesmo padrão de rajadas e mesmo ambiente de execução. A única diferença foi o uso de lote e buffer com o dobro da capacidade para o caso de 1 kB.

Após reunir os dados de latência, observamos uma média de cerca de 164 ms para mensagens de 100 B e 312 ms para mensagens de 1 kB. O gráfico de sobreposição das execuções com mensagens de 100 B (Figura 8.4) mostra curvas muito semelhantes entre si, com picos durante as rajadas e recuperação rápida em seguida. Já para 1 kB, a Figura 8.8 mostra latência base mais alta, picos mais acentuados e tempo maior para que o sistema se estabilize após as rajadas.

As latências máximas foram de aproximadamente 1118 ms para 100 B e 1380 ms para 1 kB. A variação entre as execuções também foi diferente: enquanto as três execuções de 100 B (Figuras 8.1, 8.2 e 8.3) apresentaram curvas quase idênticas, no caso de 1 kB (Figuras 8.5, 8.6 e 8.7) houve diferenças mais visíveis entre elas, especialmente na altura e na duração dos picos.

Quanto ao throughput, o cenário de 100 B sustentou cerca de 196 mil mensagens por segundo, enquanto o de 1 kB ficou em torno de 50 mil mensagens por segundo. Isso está de acordo com o fato de que mensagens maiores ocupam mais espaço e exigem mais tempo de processamento por parte do sistema.

Em resumo, o cenário com mensagens pequenas foi mais estável e respondeu melhor às rajadas. Já as mensagens maiores aumentaram a latência e deixaram o sistema mais sensível a variações.

8.5.2 Comparação: 1 kB padrão vs. 1 kB reduzido

Neste caso, comparamos duas configurações diferentes para mensagens de 1 kB. A primeira é a configuração padrão, com tamanho de lote de 65.536 bytes (64 kB) e memória de buffer de 67.108.864 bytes (64 MB), e a segunda é uma versão reduzida, com lote e buffer pela metade: 32 kB e 32 MB, respectivamente. Mantivemos todo o resto igual, incluindo o padrão de rajadas e o ambiente.

Com essa mudança, a latência média subiu de 312 ms para 449 ms, e o throughput caiu de 50 mil para cerca de 40,5 mil mensagens por segundo. No

gráfico de sobreposição (Figura 8.12), vemos que as curvas do cenário reduzido são mais altas durante quase toda a execução, com picos mais frequentes.

As três execuções com configuração reduzida (Figuras 8.9, 8.10 e 8.11) mantiveram um padrão parecido, mas sempre com latências maiores do que na configuração padrão. Mesmo em rajadas menos intensas, os picos de latência já ultrapassaram 500 ms.

Um comportamento curioso observado nos gráficos reduzidos é que, em certos momentos, a latência parece diminuir justamente durante as rajadas — o oposto do esperado. Isso ocorre porque, fora das rajadas, o produtor não consegue preencher os lotes com rapidez suficiente, e acaba enviando lotes menores ou incompletos, o que aumenta o tempo médio de envio. Durante as rajadas, o volume maior de mensagens facilita o preenchimento dos lotes, que são enviados mais rapidamente, reduzindo temporariamente a latência. Essa diferença pode ser vista com clareza na Figura 8.11, onde alguns picos ocorrem fora das áreas de carga intensa.

Apesar de a configuração reduzida ainda funcionar, ela tornou o sistema mais lento e menos estável. As diferenças entre as execuções também foram mais visíveis, o que mostra que um sistema com menos recursos pode se comportar de forma mais irregular.

Em resumo, reduzir o tamanho do lote e do buffer piorou tanto a latência quanto o throughput. O sistema se tornou mais sensível às rajadas e apresentou mais variações de uma execução para outra.

8.6 Gráficos

A seguir, apresentamos os gráficos de latência ao longo do tempo para cada um dos três cenários de payload testados – 100 B, 1 kB (configuração padrão) e 1 kB (configuração reduzida). Para cada cenário, exibimos três gráficos individuais (um para cada execução de 30 minutos) e um gráfico de *overlay* que sobrepõe as três *runs*, permitindo visualizar as diferenças de comportamento e validar visualmente as conclusões extraídas na análise anterior.

Antes de apresentar os gráficos, é importante explicar que, ao calcular os valores de latência (como média, mínima, máxima e percentis), escolhemos ignorar o trecho inicial de cada execução, ou seja, o intervalo que ocorre antes do envio da primeira rajada de mensagens. Essa parte inicial costuma ter comportamentos diferentes, com picos altos de latência, que não estão diretamente relacionados aos testes que queríamos observar. Esses picos acontecem porque, no começo da execução, o Kafka ainda está se organizando internamente

— por exemplo, abrindo conexões, carregando os processos e se preparando para enviar e receber os dados. Como o objetivo era analisar o comportamento do Kafka durante as rajadas de mensagens, deixamos esse início de fora para que os resultados refletissem apenas o que acontece quando o sistema já está funcionando normalmente.

8.6.1 Latência vs. Tempo – 100 B

Na primeira execução (veja a Figura 8.1), a latência média foi de 161,6 ms. O valor mínimo registrado foi de 26,9 ms e o máximo de 514,2 ms. O P95 ficou em 261,9 ms e o P99 em 270,3 ms.

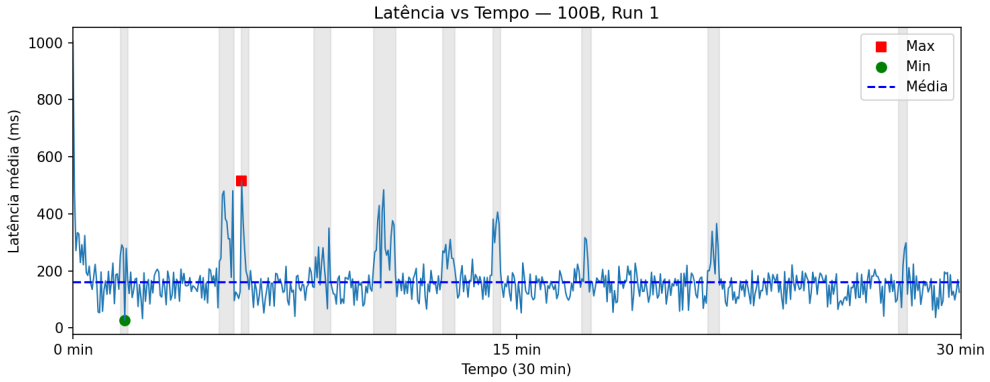

Figura 8.1: Latência vs. tempo – 100 B (Execução 1)

Fonte: Elaborado pelo autor.

A Figura 8.2 mostra a segunda execução, com latência média de 163,0 ms. Os valores extremos foram 29,5 ms (mínimo) e 460,9 ms (máximo). O P95 foi de 264,7 ms e o P99 de 273,1 ms.

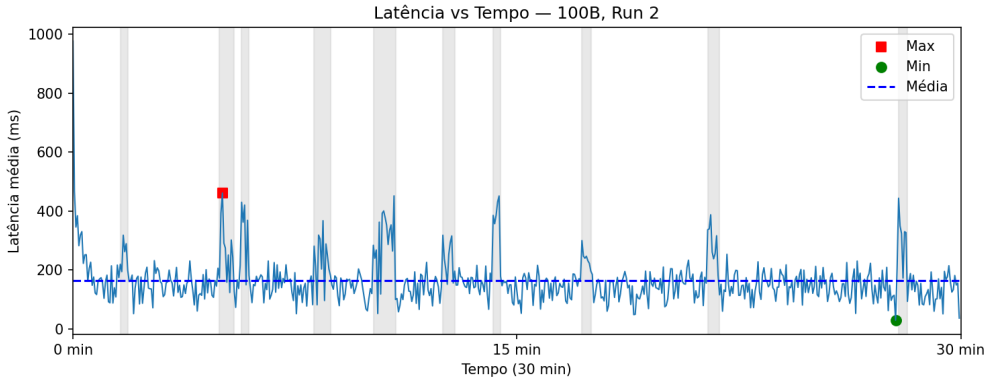

Figura 8.2: Latência vs. tempo – 100 B (Execução 2)

Fonte: Elaborado pelo autor.

Na terceira execução (Figura 8.3), a latência média foi de 156,3 ms, com mínima de 33,6 ms e máxima de 474,3 ms. O P95 foi de 255,7 ms e o P99, de 264,4 ms.

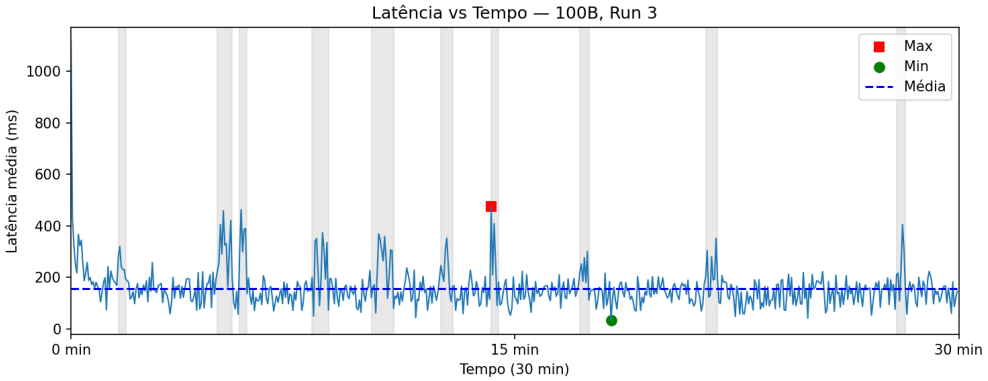

Figura 8.3: Latência vs. tempo – 100 B (Execução 3)

Fonte: Elaborado pelo autor.

A Figura 8.4 apresenta a sobreposição das três execuções. As curvas são bastante próximas, com picos coincidentes e baixa dispersão durante as rajadas.

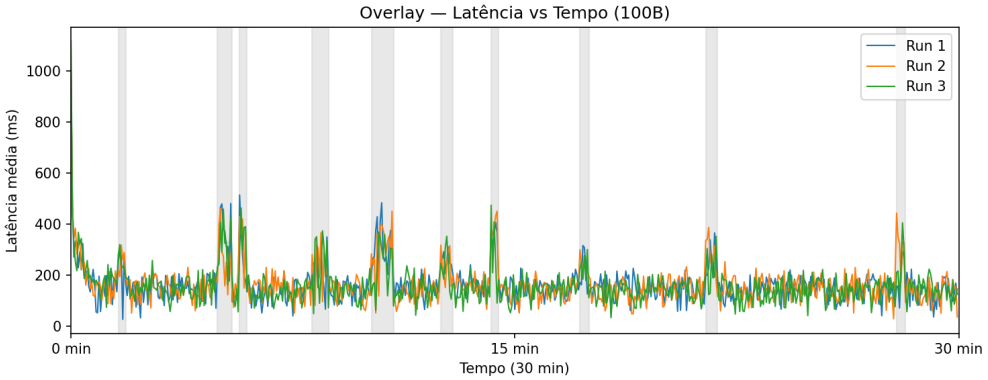

Figura 8.4: Latência vs. tempo – 100 B (Overlay das 3 execuções)

Fonte: Elaborado pelo autor.

8.6.2 Latência vs. Tempo – 1 kB (config. padrão)

Na primeira execução (Figura 8.5), a latência média foi de 310 ms, com picos ao final das rajadas. O menor valor foi de 49 ms e o maior de 552 ms. P95 e P99 ficaram em 466 ms e 477 ms, respectivamente.

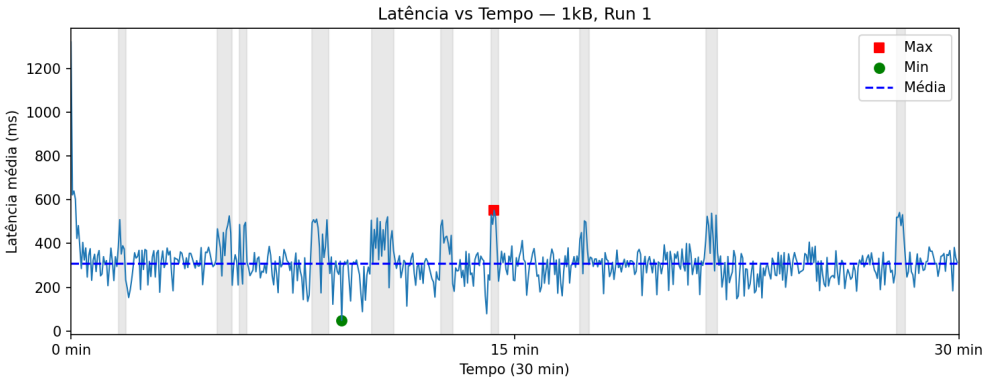

Figura 8.5: Latência vs. tempo – 1 kB (Execução 1, config. padrão)

Fonte: Elaborado pelo autor.

A Figura 8.6 apresenta a segunda execução, com latência média de 301 ms. Os valores extremos foram 102 ms e 580 ms. P95 e P99 ficaram em 448 ms e 458 ms.

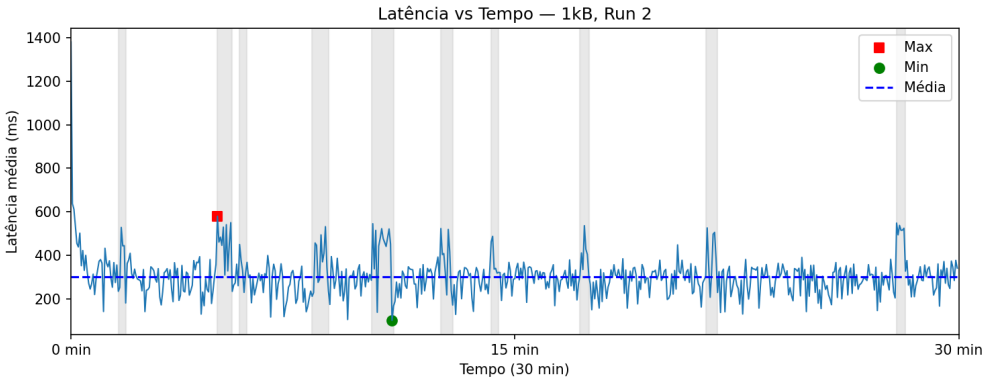

Figura 8.6: Latência vs. tempo – 1 kB (Execução 2, config. padrão)

Fonte: Elaborado pelo autor.

A terceira execução (Figura 8.7) registrou média de 305 ms, com extremos de 59 ms e 566 ms. O P95 foi de 458 ms e o P99 de 469 ms.

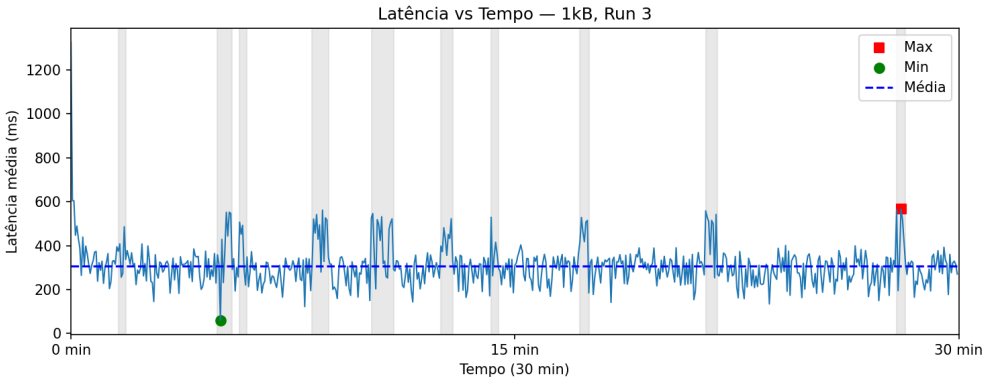

Figura 8.7: Latência vs. tempo – 1 kB (Execução 3, config. padrão)

Fonte: Elaborado pelo autor.

A Figura 8.8 mostra a sobreposição das três execuções. As curvas são semelhantes, mas há variação na amplitude dos picos entre as execuções.

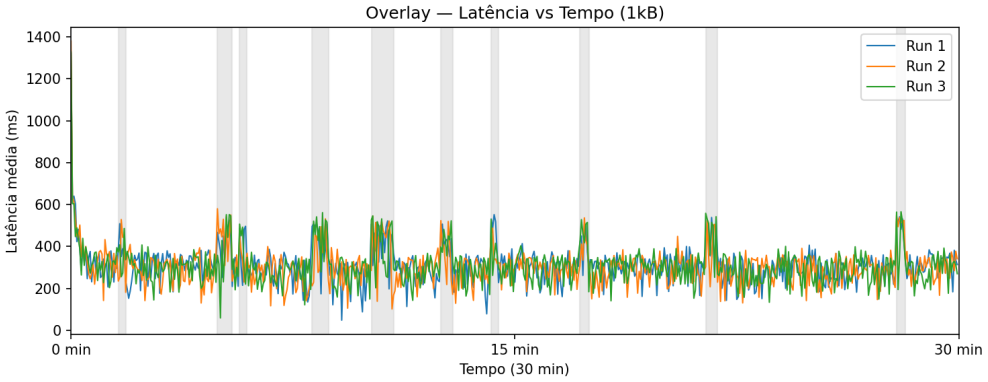

Figura 8.8: Latência vs. tempo – 1 kB (Overlay das 3 execuções, config. padrão)

Fonte: Elaborado pelo autor.

8.6.3 Latência vs. Tempo – 1 kB (config. reduzida)

A Figura 8.9 mostra a primeira execução, com média de 431 ms. O mínimo foi de 152,8 ms e o máximo, de 603,9 ms. O P95 foi de 594,0 ms e o P99 de 610,3 ms.

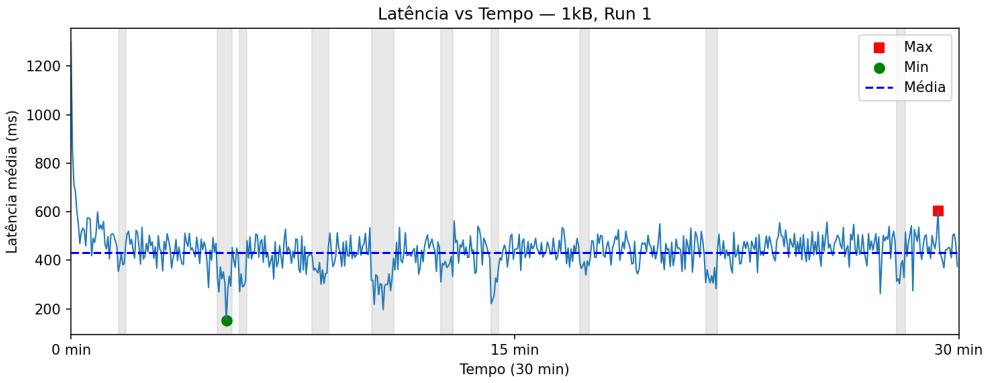

Figura 8.9: Latência vs. tempo – 1 kB (Execução 1, config. reduzida)

Fonte: Elaborado pelo autor.

Na Figura 8.10, a segunda execução apresentou média de 445,5 ms, com mínimos e máximos de 187,4 ms e 617,0 ms. P95 e P99 foram de 612,2 ms e 628,6 ms.

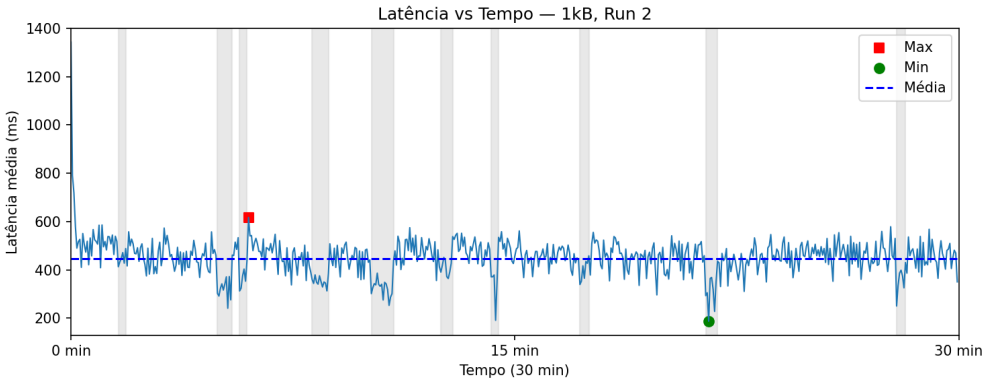

Figura 8.10: Latência vs. tempo – 1 kB (Execução 2, config. reduzida)

Fonte: Elaborado pelo autor.

A terceira execução (Figura 8.11) manteve uma média mais alta, de 469,9 ms. O menor valor foi 306,0 ms e o maior 573,8 ms. P95 e P99 ficaram em 634,6 ms e 649,4 ms.

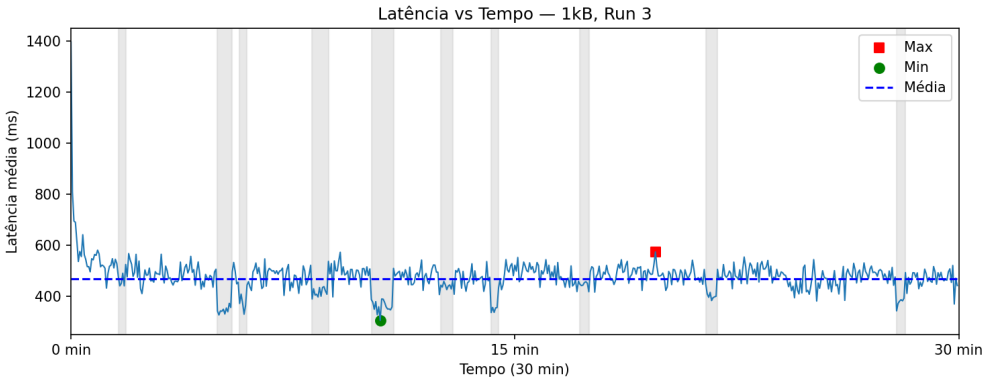

Figura 8.11: Latência vs. tempo – 1 kB (Execução 3, config. reduzida)

Fonte: Elaborado pelo autor.

A Figura 8.12 mostra a sobreposição das três execuções. As curvas apresentam maior dispersão nos picos finais e valores médios mais elevados do que nos cenários anteriores.

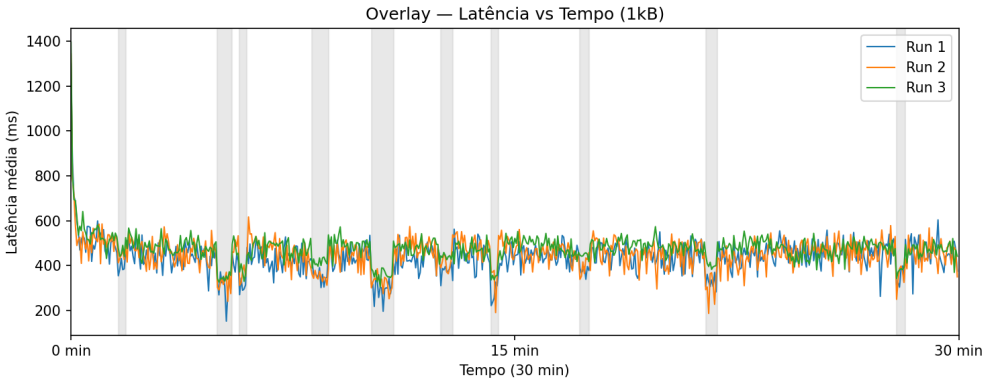

Figura 8.12: Latência vs. tempo – 1 kB (Overlay das 3 execuções, config. reduzida)

Fonte: Elaborado pelo autor.

8.6.4 Repositório do Projeto

Todo o código-fonte utilizado para os experimentos descritos neste trabalho está disponível em:

`<https://github.com/JoaoGabrielNielsen/Projeto-Final>`

O repositório contém os scripts de simulação, arquivos de configuração, logs gerados e os gráficos utilizados na análise de desempenho.

9
Conclusão

Retomando os objetivos apresentados no início do trabalho, podemos dizer que eles foram atingidos de forma satisfatória. O primeiro deles era entender como funcionam as arquiteturas de streaming de dados em tempo quase real, com foco nos principais componentes envolvidos e nos desafios que surgem na prática. Esse objetivo foi alcançado por meio do estudo do Apache Kafka, que foi analisado dentro de um modelo de referência dividido em quatro etapas: ingestão, processamento, armazenamento e mediação por message broker (Capítulos **??**, 5 e 6). Ao longo do trabalho, explicamos como o Kafka atua nessas etapas e quais são suas principais características internas.

O segundo objetivo era avaliar o comportamento do Kafka em situações de carga controlada. Para isso, construímos um ambiente de testes com três cenários diferentes de configuração do produtor e variamos o tamanho das mensagens, a capacidade de lote e de buffer. A partir desses testes, apresentados no Capítulos 7, conseguimos observar como essas mudanças impactam o tempo de resposta do sistema (latência), o volume de dados que ele consegue processar por segundo (throughput) e a estabilidade da operação. Os testes foram conduzidos com cuidado, repetindo as execuções, automatizando o ambiente e mantendo os mesmos parâmetros em todas as rodadas, o que nos ajudou a reduzir ao máximo interferências externas, como picos de uso da máquina ou variações nos recursos do container.

A principal contribuição do trabalho foi justamente mostrar, na prática, como pequenas mudanças nas configurações podem ter efeitos visíveis no desempenho do sistema. Por exemplo, vimos que reduzir o tamanho de lote e a memória do buffer pode deixar o sistema mais lento e mais instável, mesmo que os outros parâmetros continuem os mesmos. Também observamos que o uso de compressão e o tamanho das mensagens afetam diretamente a latência e o número de mensagens que o sistema consegue processar. Os gráficos e tabelas ajudam a visualizar esses efeitos de forma clara e são úteis para quem precisa configurar o Kafka em um cenário quase real.

Durante a realização do projeto, aprendemos várias coisas importantes. Uma delas foi que o Kafka é um sistema bastante robusto, mas que exige atenção na hora de configurar. Parâmetros como *linger.ms*, *batch.size* e *buffer.memory* precisam ser ajustados de acordo com o tipo de carga que será enviada, pois escolhas erradas podem piorar o desempenho. Também ficou claro que é importante planejar bem os testes, garantindo que eles sejam repetíveis,

organizados e com o mínimo de interferência. Para isso, usamos ferramentas como Docker, bash e Python, que foram essenciais para automatizar o processo e manter os testes organizados.

Para trabalhos futuros, há várias possibilidades de expansão. Uma delas seria testar o Kafka em um ambiente com mais de um broker, simulando um cluster real e analisando como a replicação e a divisão das partições afetam o desempenho. Outra seria adicionar consumidores com diferentes velocidades de leitura, para entender o impacto disso na fila de mensagens. Também seria interessante conectar o Kafka com ferramentas de processamento em tempo quase real, como Kafka Streams ou Apache Flink, e testar o sistema de ponta a ponta. Por fim, explorar diferentes formatos de compressão e serialização de mensagens, ou estudar mais a fundo o lado do consumidor (por exemplo, como o *commit* de mensagens afeta o sistema), são caminhos possíveis para ampliar os resultados encontrados aqui.

Em resumo, este trabalho conseguiu explorar de forma prática e teórica como o Apache Kafka se comporta em diferentes situações de carga. Através de testes bem definidos, mostramos como as decisões de configuração impactam o desempenho e documentamos tudo de maneira clara. Esperamos que os resultados sirvam de base para outras pessoas que precisem construir sistemas de streaming de dados que sejam eficientes e confiáveis.

10
Referências bibliográficas

Amazon Web Services. **O que é Big Data?** 2024. Acesso em: 31 out. 2024. Disponível em: <https://aws.amazon.com/pt/what-is/big-data/?nc1=h_ls>.

Amazon Web Services. **O que é IoT?** 2024. Acesso em: 31 out. 2024. Disponível em: <https://aws.amazon.com/pt/what-is/iot/>.

Amazon Web Services. **What is a modern streaming data architecture?** 2024. Acesso em: 22 set. 2024. Disponível em: <https://docs.aws.amazon.com/whitepapers/latest/build-modern-data-streaming-analytics-architectures/what-is-a-modern-streaming-data-architecture.html>.

Amazon Web Services. **What is Streaming Data?** 2024. Acesso em: 27 jun. 2025. Disponível em: <https://aws.amazon.com/what-is/streaming-data/>.

Amazon Web Services. **What's the Difference Between Throughput and Latency?** 2024. Acesso em: 27 jun. 2025. Disponível em: <https://aws.amazon.com/compare/the-difference-between-throughput-and-latency/>.

ANALYTICS, A. S. **Azure Stream Analytics**. 2024. Acesso em: 29 set. 2024. Disponível em: <https://azure.microsoft.com/pt-br/products/stream-analytics>.

Apache Software Foundation. **Apache ZooKeeper**. 2025. Acesso em: 15 jun. 2025. Disponível em: <https://zookeeper.apache.org/>.

Apache Software Foundation. **Kafka Streams Developer Guide**. 2025. Acesso em: 15 jun. 2025. Disponível em: <https://kafka.apache.org/documentation/streams/>.

BIGQUERY, C. **BigQuery Overview**. 2024. Acesso em: 23 set. 2024. Disponível em: <https://cloud.google.com/bigquery?hl=en>.

BLOB, A. **Azure Blob Storage**. 2024. Acesso em: 29 set. 2024. Disponível em: <https://azure.microsoft.com/en-us/products/storage/blobs>.

CONECT, C. K. **Kafka Connect Overview**. 2024. Acesso em: 28 jun. 2025. Disponível em: <https://docs.confluent.io/platform/current/connect/index.html>.

Confluent. **Kafka Consumer Groups**. 2024. Acesso em: 27 jun. 2025. Disponível em: <https://docs.confluent.io/platform/current/clients/consumer.html>.

Confluent. **ksqlDB Overview**. 2025. Acesso em: 15 jun. 2025. Disponível em: <https://docs.confluent.io/platform/current/ksqldb/overview.html>.

CUGOLA, G.; MARGARA, A. Processing flows of information: From data stream to complex event processing. **ACM Computing Surveys (CSUR)**, v. 44, n. 3, p. 1–62, 2012.

DATAFLOW, C. **Cloud Dataflow Overview**. 2024. Acesso em: 23 set. 2024. Disponível em: <https://cloud.google.com/dataflow/docs/overview>.

FOUNDATION, A. S. **KafkaConsumer (Apache Kafka 2.5.0 API)**. 2020. <https://kafka.apache.org/25/javadoc/org/apache/kafka/clients/consumer/KafkaConsumer.html>. Acesso em: 10 jun. 2025.

GAMA, J.; RODRIGUES, P. P. Data stream processing. In: **Learning from Data Streams: Processing Techniques in Sensor Networks**. [S.l.: s.n.], 2007. p. 25–39.

Google Cloud. **What is a data lakehouse?** 2024. Acesso em: 27 jun. 2025. Disponível em: <https://cloud.google.com/blog/products/data-analytics/open-data-lakehouse-on-google-cloud>.

HEINZE, T. et al. Cloud-based data stream processing. In: **Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems**. [S.l.: s.n.], 2014. p. 238–245.

ISAH, H.; ZULKERNINE, F.; MCHEICK, H. A survey of distributed data stream processing frameworks. **IEEE Access**, v. 7, p. 154300–154316, 2019.

KINESIS, A. **Amazon Kinesis**. 2024. Acesso em: 25 set. 2024. Disponível em: <https://aws.amazon.com/pt/pm/kinesis/>.

Microsoft Azure. **Azure Event Hubs**. 2024. Acesso em: 22 set. 2024. Disponível em: <https://azure.microsoft.com/pt-br/products/event-hubs/>.

(MSK), A. M. S. for A. K. **Amazon Managed Streaming for Apache Kafka (MSK)**. 2024. Acesso em: 25 set. 2024. Disponível em: <https://aws.amazon.com/pt/msk/>.

OUNACER, S. et al. Real-time data stream processing challenges and perspectives. **International Journal of Computer Science Issues (IJCSI)**, v. 14, n. 5, p. 6–12, 2017.

PIPELINES, C. S. **Building streaming data pipelines**. 2024. Acesso em: 22 set. 2024. Disponível em: <https://cloud.google.com/blog/products/data-analytics/building-streaming-data-pipelines>.

POWERBI, M. **Power BI**. 2024. Acesso em: 29 set. 2024. Disponível em: <https://www.microsoft.com/pt-br/power-platform/products/power-bi>.

PUB/SUB, C. **Cloud Pub/Sub Overview**. 2024. Acesso em: 23 set. 2024. Disponível em: <https://cloud.google.com/pubsub/docs/overview>.

STORAGE, C. **Google Cloud Storage**. 2024. Acesso em: 23 set. 2024. Disponível em: <https://cloud.google.com/storage/?hl=en>.

TESTING, C. P. **Kafka Performance Testing**. 2024. <https://www.confluent.io/learn/kafka-performance-testing/>. Acesso em: 16 jun. 2025.

WANG, J. et al. Big data analytics for intelligent manufacturing systems: A review. **Journal of Manufacturing Systems**, v. 62, p. 738–752, 2022.