UNIVERSIDADE FEDERAL DE UBERLÂNDIA

José Lucas Ferreira de Lima

De voltas rápidas a decisões rápidas: uma arquitetura orientada a eventos para telemetria dos carros de Fórmula 1

Uberlândia, Brasil

2026

UNIVERSIDADE FEDERAL DE UBERLÂNDIA

José Lucas Ferreira de Lima

De voltas rápidas a decisões rápidas: uma arquitetura orientada a eventos para telemetria dos carros de Fórmula 1

Trabalho de conclusão de curso apresentado à Faculdade de Computação da Universidade Federal de Uberlândia, como parte dos requisitos exigidos para a obtenção título de Bacharel em Sistemas de Informação.

Orientador: Prof. Dr. Ronaldo Castro de Oliveira

Universidade Federal de Uberlândia – UFU

Faculdade de Computação

Bacharelado em Sistemas de Informação

Uberlândia, Brasil

2026

José Lucas Ferreira de Lima

De voltas rápidas a decisões rápidas: uma arquitetura orientada a eventos para telemetria dos carros de Fórmula 1

Trabalho de conclusão de curso apresentado à Faculdade de Computação da Universidade Federal de Uberlândia, como parte dos requisitos exigidos para a obtenção título de Bacharel em Sistemas de Informação.

Trabalho aprovado. Uberlândia, Brasil, 01 de novembro de 2016:

Prof. Dr. Ronaldo Castro de Oliveira
Orientador

Prof. Dr. rer. nat. Daniel Duarte Abdala

Profa. Dra. Fernanda Maria da Cunha Santos

Uberlândia, Brasil
2026

Dedico este trabalho aos meus pais, pelo apoio incondicional em toda a minha trajetória; aos meus amigos, que foram pilares essenciais durante minha jornada acadêmica; à minha namorada, que me incentiva diariamente em tudo; e à minha avó, que nos deixou enquanto eu construía este trabalho, mas cuja memória permanece viva em meu coração.

Agradecimentos

Agradeço primeiramente a **Jesus**, por ter me mantido firme e resiliente durante toda a minha jornada acadêmica.

Aos **meus pais**, por sempre me apoiarem em tudo e por não medirem esforços para que eu alcance meus objetivos.

À minha namorada e futura esposa, **Maria Eduarda Franco**, por me incentivar todos os dias a nunca desistir e por me lembrar do motivo pelo qual estamos constantemente lutando.

À *minha família*, por todo o apoio e carinho durante todo o curso.

Ao meu orientador, **Prof. Dr. Ronaldo Castro de Oliveira**, por todo o conhecimento passado e pelas nossas agendas, que permitiu o desenvolvimento e conclusão deste trabalho com excelência.

Aos membros da banca avaliadora, **Prof. Dr. rer. nat. Daniel Duarte Abdala** e **Profa. Dra. Fernanda Maria da Cunha Santos**, por dedicarem parte do seu tempo para contribuir com este trabalho.

Aos meus queridos colegas **Victor Ricarte Silva**, **Eduardo Santos Pires**, **Thayrony Thadeu Brum**, **José Luzia da Silva Neto** e **Amanda Estrela dos Santos**, pela parceria, risadas e amizade durante essa trajetória.

Resumo

A Fórmula 1 é um ambiente altamente tecnológico, caracterizado pela geração contínua de grandes volumes de dados de telemetria em tempo real, os quais são fundamentais para a tomada de decisões estratégicas e operacionais. Nesse contexto, a capacidade de coletar, processar e disponibilizar essas informações com baixa latência torna-se um diferencial competitivo essencial. Este trabalho propõe o desenvolvimento de uma arquitetura de dados baseada no paradigma orientado a eventos, com o objetivo de viabilizar o processamento simultâneo de dados em tempo real e em lote. A solução foi projetada para simular o fluxo completo de dados de telemetria de um carro de Fórmula 1, utilizando como fonte o simulador *F1 23*. A arquitetura implementada é composta por um módulo de ingestão de dados via protocolo UDP, responsável pela captura e pré-processamento das informações, seguido por um sistema de mensageria, que garante a transmissão escalável e tolerante a falhas dos eventos. A partir dessa camada, os dados são direcionados para duas trilhas: uma camada rápida, responsável pelo processamento em tempo real e visualização; e uma camada em lote, responsável pelo armazenamento em um *data lake* e posterior processamento em um *data warehouse* relacional. A metodologia adotada baseia-se na construção e validação experimental da *pipeline*, utilizando exclusivamente tecnologias *open source*, com foco em desempenho, escalabilidade e desacoplamento entre os componentes. Os resultados demonstram a viabilidade da arquitetura proposta para lidar com dados de alta velocidade e volume, além de evidenciar seu potencial para aplicações em cenários reais de engenharia de dados. Conclui-se que a utilização de uma arquitetura orientada a eventos é eficaz para o processamento de dados de telemetria em tempo real, permitindo não apenas a análise instantânea durante a corrida, mas também o armazenamento estruturado para análises posteriores e aplicações futuras, como modelos de aprendizado de máquina.

Palavras-chave: arquitetura orientada a eventos, engenharia de dados, *streaming* de dados, Fórmula 1, telemetria.

Lista de ilustrações

Lista de abreviaturas e siglas

UDP	*User Datagram Protocol*
TCP	*Transmission Control Protocol*
ETL	*Extract, Transform and Load*
F1	Fórmula 1
BI	*Business Intelligence*
IA	Inteligência Artificial
ML	*Machine Learning*
LA	*Lambda Architecture*
EDA	*Event-Driven Architecture*
JSON	*JavaScript Object Notation*
ACID	Atomicidade, Consistência, Integridade, Durabilidade
IoT	*Internet of Things*
SQL	*Structured Query Language*
DNS	*Domain Name Server*
IaC	*Infrastructure as Code*
API	*Application Programming Interface*
RPM	Rotações por minuto
CLI	*Command-Line Interface*
CAN	*Controller Area Network*
SAE	*Society of Automotive Engineers*

Sumário

1 Introdução

1.1 Contextualização e motivação

De acordo com um artigo publicado pela Mercedes (2022), cada carro de Fórmula 1 é equipado com mais de 250 sensores categorizados em controle, instrumentação e monitoramento, que coletam dados vitais de pressão, temperatura, inércia e deslocamento de todos os sistemas do veículo. Durante um fim de semana de corrida, o volume total de dados gerados por um único carro, incluindo telemetria ao vivo (cerca de 30 MB por volta), vídeo e informações auxiliares, excede 1 *terabyte*, podendo duplicar ou triplicar após o pós-processamento. Adicionalmente, as fábricas contribuem com vasta quantidade de dados provenientes de dinamômetros, simuladores e túneis de vento, criando um ecossistema de informação que exige a transferência de até 11 *terabytes* entre a pista e as sedes da equipe, com telemetria ao vivo chegando em 10 milissegundos em corridas europeias. Essa imensa e contínua geração de dados deixa claro a natureza intensiva em dados da Fórmula 1 moderna.

No cenário altamente competitivo da Fórmula 1, a premissa fundamental é que a análise de dados e, por extensão, a otimização do desempenho não pode existir sem um fluxo contínuo e confiável de informações. A limitação drástica do tempo de testes em pista impõe uma pressão imensa para a coleta de dados precisos e completos na primeira tentativa, pois não há oportunidade para repetições, como aponta a Mercedes (2022). Essa característica ressalta a importância crítica de uma infraestrutura de dados robusta, onde, como detalhado por Msakamali (2024), a análise de dados é fundamental para aprimorar o desempenho e embasar decisões estratégicas em tempo real. Uma vez coletados do carro, os dados são sincronizados, criptografados e transmitidos via telemetria para a garagem, garantindo que engenheiros e pilotos tenham acesso a informações consistentes e seguras. O uso de software especializado, como o ATLAS[1] da *McLaren Applied*, tanto em tempo real quanto para análises retrospectivas, demonstra a dedicação contínua à extração de *insights* valiosos, seja para comparar o desempenho dos pilotos, analisar dados de frenagem e velocidades de curva, ou para identificar oportunidades que tornarão o carro e o piloto ainda mais rápidos. A enorme quantidade de dados, embora desafiadora, é também uma oportunidade de engenharia para priorizar e analisar as informações corretas, acelerando o aprendizado e a evolução das equipes. Assim, a existência de uma *pipeline* (fluxo de dados) de coleta e processamento de dados em tempo real não é apenas uma conveniência, mas uma necessidade estratégica e operacional para sustentar a competitividade na Fórmula 1.

[1] https://atlas.motionapplied.com

A competitividade na Fórmula 1 moderna é diretamente proporcional à confiabilidade dos dados, pois a análise e a tomada de decisão dependem intrinsecamente da qualidade e integridade da informação. Dados imprecisos ou inconsistentes comprometem *insights* e decisões estratégicas, como enfatizado por Suman (2025). Em um ambiente de alta *performance* e tempo crítico, a alta latência (tempo que os dados levam para trafegar de um ponto ao outro) é inaceitável; a transformação de vastos volumes de telemetria em conhecimento acionável em milissegundos exige uma arquitetura de dados eficiente. O processamento em tempo real, segundo o artigo do Richman (2025), é crucial para tomada de decisões imediatas e ações ágeis, permitindo a detecção e correção de anomalias. Assim, torna-se imperativa uma *pipeline* robusta de coleta, transmissão e processamento em tempo real, capaz de assegurar a integridade e a prontidão dos dados, do sensor até o engenheiro, garantindo resultados reproduzíveis (SCISURE, 2025).

Diante da complexidade e da criticidade dos desafios impostos pela volumosa geração e pela necessidade de processamento em tempo real dos dados na Fórmula 1, este trabalho propõe uma solução prática para o problema. Nosso estudo se dedicará a explorar e aplicar conceitos modernos de engenharia de dados, com foco em processamento de dados contínuos e arquiteturas orientadas a eventos, a fim de construir uma *pipeline* robusta e que permita a disponibilização de dados em duas frentes simultaneamente. A arquitetura proposta promove o desacoplamento entre componentes por meio do paradigma orientado a eventos e considera a dualidade entre processamento em tempo real e em lote, permitindo tanto a análise instantânea durante a corrida quanto o armazenamento estruturado para análises posteriores. O diferencial reside na utilização de ferramentas gratuitas e de código aberto, demonstrando que é possível desenvolver soluções de alta performance e segurança sem depender de *softwares* proprietários ou de alto custo. Dessa forma, busca-se evidenciar como essas tecnologias podem efetivamente resolver os desafios contemporâneos da engenharia de dados, garantindo que as equipes de Fórmula 1 possam extrair *insights* valiosos e tomar decisões ainda mais assertivas em um ambiente onde cada milissegundo conta.

1.2 Problema de pesquisa

A Fórmula 1 contemporânea, impulsionada por uma colossal geração de dados – com mais de 250 sensores em cada carro produzindo *terabytes* de informações cruciais por fim de semana de corrida, de acordo com levantamento realizado pela Mercedes (2022) –, demanda uma capacidade sem precedentes de coleta, processamento e análise em tempo real para a otimização contínua do desempenho das equipes.

Entretanto, além da necessidade de análise imediata durante a corrida, esses mesmos dados possuem alto valor estratégico para análises posteriores, como o treinamento de

modelos de inteligência artificial, simulações e aprimoramento do desempenho do piloto. Esse cenário impõe um desafio adicional: como conciliar, em uma mesma arquitetura, o processamento de dados com baixa latência para tomada de decisão em tempo real e, simultaneamente, o armazenamento estruturado para análises históricas e aplicações futuras.

Diante desse contexto, surge o seguinte problema de pesquisa:

Como projetar e implementar uma arquitetura de dados que seja capaz de coletar dados de telemetria em tempo real, transmiti-los com baixa latência em um formato legível para consumo imediato durante a corrida e, ao mesmo tempo, persistir esses dados de forma estruturada para análises posteriores, como treinamento de modelos de *machine learning* e simulações de desempenho?

Este trabalho se propõe a responder a essa questão por meio da aplicação de conceitos modernos de engenharia de dados, com ênfase em arquiteturas orientadas a eventos e processamento de dados em *streaming*, visando demonstrar a viabilidade de uma solução que atenda simultaneamente às demandas de processamento em tempo real e análise em lote.

1.3 Objetivos

1.3.1 Objetivo geral

O objetivo geral deste trabalho é projetar e implementar uma arquitetura de dados baseada no paradigma orientado a eventos e no modelo de arquitetura lambda, utilizando exclusivamente tecnologias de código aberto, capaz de coletar, processar e disponibilizar dados de telemetria em tempo real no contexto da Fórmula 1, ao mesmo tempo em que permite o armazenamento estruturado e limpo desses dados para análises posteriores.

A solução proposta busca demonstrar a viabilidade de uma *pipeline* de dados que atenda simultaneamente às demandas de baixa latência para suporte à tomada de decisão durante a corrida e às necessidades de processamento em lote para aplicações futuras, como análise histórica, simulações e modelos de inteligência artificial.

1.3.2 Objetivos Específicos

Alguns objetivos específicos deste trabalho incluem:

- **Analisar os conceitos de ETL e processamento de dados em tempo real**: Investigar e compreender os principais conceitos de ETL, bem como as metodologias de processamento de dados em tempo real (*streaming*), focando em sua aplicabili-

dade e desafios em ambientes de alta velocidade e volumetria de dados, como a Fórmula 1.

- **Desenvolver um módulo de coleta e pré-processamento de telemetria**: Implementar um módulo capaz de estabelecer conexão com uma fonte simulada de dados de telemetria da Fórmula 1 (neste caso, o simulador **F1 23**), realizar a captura e o pré-processamento inicial desses dados (transformação de valores binários/brutos para formatos legíveis) e encaminhá-los para um sistema de mensageria.
- **Implementar uma arquitetura de mensageria e processamento de *streams***: Configurar e integrar um sistema de mensageria (*message broker*) para garantir a transmissão eficiente e tolerante a falhas dos dados e desenvolver módulos de processamento que consumam esses dados em tempo real, aplicando as transformações e enriquecimentos necessários.
- **Visualizar dados de telemetria em tempo real**: Construir um *dashboard* interativo que receba os dados processados e os exiba graficamente em tempo real, permitindo a visualização clara de métricas cruciais de desempenho veicular, validando a funcionalidade e a latência da *pipeline*.
- **Discutir implicações e aplicações futuras da análise de dados**: Apresentar como os dados coletados e processados podem subsidiar a tomada de decisões estratégicas por engenheiros na Fórmula 1 e propor futuras melhorias e extensões para a *pipeline* desenvolvida, incluindo a potencial integração com modelos de análise de dados e inteligência artificial.

1.4 Justificativa

A Fórmula 1, mais do que um esporte de alta velocidade, consolidou-se como um dos ecossistemas mais avançados para o estudo e aplicação da engenharia e análise de dados. Cada carro é, em essência, um laboratório de inovação em tempo real, equipado para gerar uma quantidade massiva de informações – com dados de telemetria que podem chegar a 1,5 *terabytes* por fim de semana de corrida (LUZICH, 2025). Conforme evidenciado por Msakamali (2024), a análise de dados na F1 é fundamental para aprimorar o desempenho na corrida, fornecendo uma base sólida para a tomada de decisões estratégicas em tempo real. Nesse ambiente, a tecnologia de dados não é apenas um suporte, mas um diferencial estratégico crucial, tornando a F1 um campo de estudo exemplar para desafios contemporâneos em processamento de *streaming* e engenharia de dados de alta performance.

Apesar do vasto potencial dos dados na Fórmula 1, sua exploração plena é barrada por desafios técnicos inerentes ao *big data*, especialmente a velocidade e o volume com

que são gerados. Cada milissegundo de atraso na aquisição ou processamento de dados pode comprometer uma decisão estratégica, tornando a baixa latência um requisito não negociável. Conforme a Mercedes (2022) destaca, a transmissão de telemetria em tempo real, embora impressionantemente rápida (10 milissegundos em corridas europeias), ainda lida com a complexidade de milhões de pontos de dados por segundo. Além da velocidade, a variedade dos dados provenientes de centenas de sensores exige sistemas que possam integrar e harmonizar diferentes formatos e estruturas. Tais complexidades impõem a necessidade crítica de *pipelines* de dados robustas, eficientes e seguras. Estes fluxos de dados não apenas garantem a coleta e o transporte ininterrupto das informações, mas também realizam as transformações necessárias para converter dados brutos em *insights* acionáveis, salvaguardando a integridade e a segurança da informação, desde o carro até a tomada de decisão final.

Para além do espetáculo e da competitividade da pista, os avanços tecnológicos e as soluções desenvolvidas na Fórmula 1, particularmente no que tange à engenharia de dados e processamento em tempo real, possuem um impacto e uma aplicabilidade notáveis em diversas outras áreas. Setores como a manufatura, por exemplo, demandam a capacidade de coletar e processar dados de sensores em tempo real para fins de manutenção preditiva, otimização de linhas de produção e controle de qualidade, onde cada milissegundo na detecção de anomalias pode significar uma economia substancial ou a prevenção de falhas críticas (WIJESINGHE; ABEYSINGHE; SAMARANAYAKE, 2024). Dessa forma, o estudo aprofundado dos processos de coleta e tratamento de dados em tempo real na Fórmula 1 não apenas revela as inovações de ponta da categoria, mas também serve como um modelo prático e um campo de testes para demonstrar como a engenharia de dados e o *big data* podem ser aliados estratégicos fundamentais na tomada de decisões em cenários de alta performance e complexidade industrial.

1.5 Estrutura do trabalho

Este trabalho está organizado em cinco capítulos principais, cada um abordando um aspecto fundamental da pesquisa sobre o processo de coleta e tratamento de dados em tempo real na Fórmula 1, desde a contextualização teórica e a metodologia de engenharia de dados até a análise dos resultados obtidos.

No **Capítulo 1 - Introdução**, são apresentados o tema da pesquisa, a contextualização e a motivação para o estudo, que destacam a evolução da Fórmula 1 como um esporte intensivo em dados, bem como a formulação do problema de pesquisa. Além disso, são definidos os objetivos geral e específicos, a justificativa para a escolha do tema e a relevância do estudo. Também são descritas a metodologia utilizada e a própria estrutura do trabalho.

O **Capítulo 2 - Fundamentação teórica** apresenta os conceitos fundamentais que embasam a arquitetura proposta neste trabalho, com foco em engenharia de dados moderna e processamento de dados em tempo real. São discutidos temas como *streaming* de dados, telemetria em ambientes de alta performance e o paradigma orientado a eventos, destacando sua importância na construção de sistemas distribuídos escaláveis e desacoplados. Também são apresentados os principais componentes tecnológicos utilizados na implementação da solução, incluindo ferramentas de mensageria, bancos de dados especializados e plataformas de visualização, contextualizando seu papel dentro da arquitetura proposta.

No **Capítulo 3 - Arquitetura proposta**, são apresentados os aspectos técnicos da solução, com foco na arquitetura de dados baseada no paradigma orientado a eventos e no modelo de arquitetura lambda. O capítulo descreve, de forma abstrata, o fluxo de ingestão, processamento e distribuição dos dados de telemetria, evidenciando a separação entre o processamento em tempo real e em lote, bem como os mecanismos de mensageria, armazenamento e visualização utilizados para viabilizar a análise imediata e o processamento posterior dos dados.

O **Capítulo 4 - Implementação da solução** apresenta a materialização prática da arquitetura proposta, detalhando a integração dos componentes responsáveis pela ingestão, processamento e armazenamento dos dados de telemetria. São descritas as configurações necessárias, os principais trechos de implementação e o funcionamento da *pipeline* em execução, evidenciando a comunicação entre os módulos e o comportamento do sistema em tempo real. O capítulo também aborda os desafios encontrados durante o desenvolvimento e as soluções adotadas, reforçando a viabilidade prática da arquitetura em um cenário próximo ao real.

Por fim, o **Capítulo 5 - Conclusão e trabalhos futuros** revisita os objetivos propostos no início do trabalho e discute as principais aplicações da área de engenharia de dados em tempo real, além de sugestões de aplicações futuras para os conceitos e tecnologias exploradas, considerando possíveis avanços na área e a expansão do trabalho.

1.6 Uso de inteligência artificial no trabalho

A IA foi utilizada neste trabalho, principalmente, nas seguintes etapas:

- **Texto**: correção ortográfica, sugestão de estrutura dos parágrafos, melhoria da abordagem apresentada, utilizando majoritariamente o *ChatGPT* e *Claude*.
- **Citações e referências**: recomendações de outros trabalhos e artigos acadêmicos relacionados ao tema principal para a base argumentativa do texto, utilizando *PerplexityAI* e *ChatGPT*.

- **Codificação**: auxílio na tradução de protótipos em *Python* para *Golang* na etapa de pré-processamento, utilizando exclusivamente o *GitHub Copilot* no modo Auto (escolha automática do modelo).

2 Revisão Bibliográfica

2.1 Engenharia de dados

A engenharia de dados consolidou-se como uma disciplina central na computação aplicada a grandes volumes de informação. Ela é frequentemente definida na literatura como o conjunto de práticas voltadas ao projeto, construção e manutenção de infraestruturas e sistemas para a coleta, processamento e armazenamento de dados de forma eficiente (REIS; HOUSLEY, 2022). Essa área abrange um espectro de atividades que engloba desde a extração e limpeza dos dados brutos até sua transformação e modelagem. O intuito dessas etapas é preparar os dados para análises subsequentes, viabilizando seu consumo por ferramentas de BI e algoritmos de aprendizado de máquina.

O objetivo fundamental desta disciplina é garantir propriedades como qualidade, integridade, segurança e acessibilidade da informação. Ao garantir essas características, a infraestrutura desenvolvida permite que as organizações automatizem processos decisórios e extraiam conhecimento de maneira confiável.

Complementando essa visão estrutural, Chai et al. (2023) destaca a etapa de preparação de dados como um dos pilares críticos dessa disciplina. Os autores conceituam essa fase como o ato de manipular dados brutos de múltiplas fontes, convertendo-os para um formato unificado que possa ser prontamente analisado. A relevância dessa transformação é tamanha que, segundo os autores, ela é o primeiro passo para tarefas de *machine learning* e chega a consumir cerca de 80% do tempo total de um *pipeline* de análise de dados avançada, impactando diretamente o desempenho dos modelos analíticos.

2.2 Análise de dados

Tukey (1962) propõe uma visão ampliada da análise de dados, indo além da estatística tradicional. Segundo o autor, a análise de dados compreende não apenas os procedimentos para análise em si, mas também as técnicas de interpretação dos resultados, o planejamento da coleta de dados de forma a torná-los mais adequados à análise e todo o arcabouço estatístico envolvido nesse processo.

Nesse contexto, a análise de dados não se limita à inferência estatística, mas constitui um campo mais abrangente, voltado à extração de padrões, à geração de insights e ao suporte à tomada de decisão.

Na Fórmula 1, a análise de dados é utilizada para monitorar o desempenho dos carros, simular cenários de corrida, prever resultados e otimizar estratégias, permitindo

que as equipes identifiquem padrões complexos e obtenham vantagens competitivas ao longo do campeonato.

Ao longo deste trabalho, será discutida a importância da análise de dados como etapa final do fluxo de dados, com o objetivo de agregar valor prático às informações coletadas e processadas. Segundo Fayyad, Piatetsky-Shapiro e Smyth (1996), o processo de transformar dados de baixo nível em representações mais compactas, abstratas e úteis caracteriza a descoberta de conhecimento.

2.3 *Streaming* de dados

No panorama atual da análise de dados, a capacidade de processar informações em tempo hábil tornou-se um diferencial competitivo crucial para organizações e sistemas complexos. Tradicionalmente, o processamento de dados era predominantemente realizado em modo de lote (*batch processing*), onde grandes volumes de dados são coletados e processados periodicamente. Embora eficiente para tarefas que não exigem imediatismo, essa abordagem apresenta limitações significativas em cenários dinâmicos e de alta volatilidade (HASHEM et al., 2015).

É nesse contexto que o processamento de dados em tempo real (*real-time data processing*) ou *streaming* de dados emerge como uma abordagem indispensável. O *streaming* de dados refere-se à metodologia de processamento de informações à medida que elas são geradas ou chegam, de forma contínua e com mínima latência (XU et al., 2018). Diferentemente do processamento em lote, que opera sobre conjuntos de dados estáticos e finitos, o processamento de *streaming* lida com fluxos de dados contínuos e potencialmente ilimitados.

A capacidade de processar e reagir a esses fluxos contínuos de dados rapidamente permite que as equipes otimizem o desempenho em frações de segundo, transformando a agilidade informacional em vantagem competitiva (MERCEDES, 2022).

2.4 Extração, transformação e carga (ETL)

O processo de extração, transformação e carga (ETL) consiste em um conjunto de operações fundamentais para a integração e consolidação de dados. Segundo Seenivasan (2023), essas etapas são responsáveis por coletar informações de diversas fontes, torná-las consistentes e úteis, e armazená-las em sistemas de destino, como bancos de dados ou *data warehouses*. Para elucidar a complexidade operacional desse fluxo, Sajida e Ramakrishna (2015) detalham os desafios inerentes a cada uma de suas três fases.

A etapa inicial de **Extração** lida com a aquisição de dados provenientes de sistemas locais ou externos. O desafio primordial dessa fase reside em acessar fontes diversificadas

e frequentemente instáveis, extraindo as informações relevantes e reformatando-as para um padrão inicial, sem comprometer a performance dos sistemas de origem (SAJIDA; RAMAKRISHNA, 2015).

Em seguida, a **Transformação** é descrita na literatura como a fase mais laboriosa e a que mais agrega valor ao processo (SAJIDA; RAMAKRISHNA, 2015). É neste momento que as regras de negócio são aplicadas e os conflitos semânticos e sintáticos são resolvidos. Essa etapa atua em duas frentes principais: a limpeza (*clean*), focada em tratar dados errôneos, corrompidos ou ausentes; e a conformidade (*conform*), que garante a compatibilidade estrutural das informações, verificando chaves e regras de integridade do sistema. Para isso, são utilizadas operações técnicas como filtragem, ordenação e junção de dados (*joins*).

Por fim, a etapa de **Carga** (*Loading*) consiste em gravar os dados previamente processados e integrados nos sistemas de destino. Sajida e Ramakrishna (2015) observa que essa fase pode ser computacionalmente custosa e demandar um tempo considerável, especialmente devido às técnicas de indexação e particionamento exigidas por repositórios analíticos robustos.

A Figura 1 ilustra o papel de cada etapa e como os componentes são orquestrados para atingir o objetivo principal da tarefa de dados.

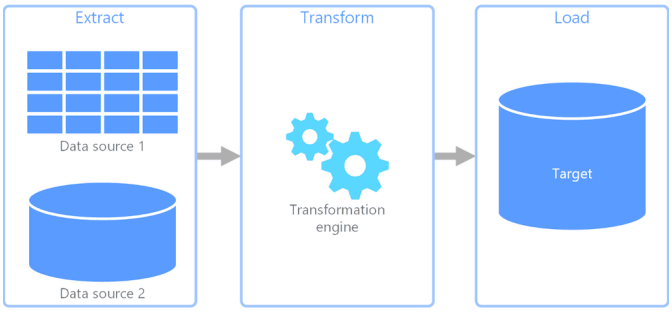

Figura 1 – Figura ilustrativa de um fluxo de ETL.

2.5 Telemetria de dados em tempo real

A telemetria consiste na tecnologia que viabiliza a mensuração remota e a transmissão contínua de dados. De acordo com Mendes (2011), os sistemas telemétricos são inerentemente compostos por múltiplos elementos integrados, que abrangem sensores de captação, unidades de processamento, meios de comunicação entre dispositivos e sistemas de armazenamento em banco de dados.

Aplicando esse conceito ao contexto veicular, Santos, Silva e Machado (2020) definem a telemetria automotiva como um sistema integrado de captura digital de informações. Essa arquitetura tem como finalidade rastrear, medir e comandar veículos de forma

remota. Os autores destacam que o processamento dessas informações garante maior assertividade no gerenciamento e atua ativamente na prevenção de riscos e sinistros, uma vez que os dados interpretados se tornam o alicerce para a tomada de decisões.

A captura ininterrupta das variáveis dinâmicas do carro exige que os componentes de comunicação e processamento - detalhados nas seções subsequentes - sejam capazes de lidar com a ingestão massiva de dados sem gargalos de latência. É exatamente essa demanda por respostas instantâneas que expõe as limitações dos fluxos de dados tradicionais e justifica a adoção de arquiteturas mais reativas.

2.6 Protocolo de comunicação UDP

De acordo com Tanenbaum e Wetherall (2011), o UDP é um dos vários protocolos da internet e oferece uma maneira para que as aplicações enviem datagramas IP encapsulados sem a necessidade de estabelecer uma conexão. Ele transmite segmentos compostos por um cabeçalho de 8 bytes seguido pela carga útil (*payload*). As duas portas servem para identificar os pontos de extremidade (*endpoints*) dentro das máquinas de origem e destino. Quando um pacote UDP chega, sua carga útil é entregue ao processo vinculado à porta de destino (TANENBAUM; WETHERALL, 2011).

Conforme apontado por Al-Dhief et al. (2018), o UDP é considerado um protocolo sem conexão e não garante uma entrega confiável dos dados entre os pontos de extremidade, uma vez que não cria uma conexão de ponta a ponta entre o remetente e o destinatário das informações, não havendo, inclusive, a verificação da identidade de quem está enviando ou recebendo os dados.

2.7 Bancos de dados de séries temporais

Como explicado por Hadjichristofi et al. (2024), os bancos de dados de séries temporais são criados com o propósito de lidar com dados sensíveis ao tempo de forma mais eficiente. Projetados para *streams* de dados de alta frequência, são otimizados para latência mínima e utilizam técnicas de indexação específicas para consultas baseadas em tempo mais rápidas.

Shah, Jat e Sashidhar (2022) explica o conceito de séries temporais como uma sequência ordenada de valores variáveis em um determinado intervalo de tempo. Esses bancos são utilizados para registrar e gerenciar essas mudanças nas variáveis ao longo do tempo. Shah, Jat e Sashidhar (2022) também esclarece sobre as principais propriedades desses bancos, as quais englobam:

1. **Escalabilidade:** O volume de dados de séries temporais cresce de forma acelerada,

exemplificado por veículos conectados que podem gerar até 25 GB de informações por hora. Diferente de sistemas convencionais, os TSDBs (bancos de dados de séries temporais) são projetados para suportar essa escala ao priorizar a dimensão temporal, resultando em maiores taxas de inserção, consultas rápidas em grandes volumes e compressão otimizada.

2. **Usabilidade:** Frequentemente, incorporam capacidades nativas de análise, como políticas de retenção, consultas contínuas e agregações temporais flexíveis, o que aprimora a experiência do usuário em fluxos de análise temporal.

3. **Compressão de dados:** Devido à alta frequência de registros — muitas vezes em intervalos de segundos ou inferiores —, a aplicação de técnicas robustas de compactação é mandatória para gerenciar a massiva volumetria de dados.

4. **Localização de dados:** Para evitar a degradação da performance em consultas onde informações relacionadas estão dispersas, os TSDBs organizam blocos (*chunks*) de dados de um mesmo dispositivo e período de forma contígua, garantindo um acesso físico mais eficiente.

5. **Consultas de intervalo rápidas e fáceis:** Graças à organização da localização dos dados, as consultas de intervalo (*range queries*) são significativamente mais rápidas se comparadas aos bancos tradicionais. Além disso, a utilização de linguagens similares ao SQL facilita a curva de aprendizado.

6. **Alto desempenho de escrita:** Em cenários de carga máxima, os TSDBs garantem alta disponibilidade e performance tanto para leitura quanto para escrita, sendo arquitetados para manter a resiliência operacional sob condições extremas.

2.8 Visualização de dados

Neri et al. (2025) define a visualização de dados como um mecanismo facilitador na apresentação, democratização e entendimento dos dados do domínio, por meio de ferramentas como gráficos de dispersão, colunas e *boxplots*. Para o autor, essa prática desempenha um papel crucial na transformação de dados complexos em formatos acessíveis e interpretáveis, configurando-se como um processo dinâmico que envolve a percepção, a interpretação e o raciocínio do usuário para a extração de significado (NERI et al., 2025).

Complementarmente, Hou, Li e Zhang (2026) afirma que, na era do *big data*, a visualização tornou-se uma ponte fundamental que conecta a informação à tomada de decisão, convertendo dados abstratos em expressões gráficas intuitivas que sustentam tanto pesquisas científicas quanto aplicações industriais de alta complexidade.

2.9 *Data lake*

Hai, Geisler e Quix (2021) descreve os *data lakes* como um sistema de gerenciamento e armazenamento de arquivos escalável e flexível, que tem a capacidade de ingerir dados brutos de múltiplas fontes no seu formato original, além de prover consultas nestes arquivos e *analytics*.

Hai, Geisler e Quix (2021) complementam que, embora o armazenamento seja central, um *data lake* deve atuar como um sistema de governança para evitar que a ausência de esquemas transforme o repositório em um "pântano de dados"(*data swamp*). Para isso, a gestão de metadados é essencial e deve ocorrer de forma incremental: durante a ingestão, extraem-se informações estruturais e, no momento do acesso, aplica-se o esquema sob demanda (*schema-on-the-fly*). Essa abordagem garante flexibilidade para consultas *ad-hoc*, permitindo que o conhecimento sobre os dados amadureça conforme as necessidades de processamento e integração surgem.

2.10 *Data warehouse*

Lamer et al. (2024) descreve o *data warehouse* como o componente mais prevalente do pipeline, atuando como um repositório centralizado de dados integrados de múltiplas fontes díspares. Ele armazena dados históricos e atuais em um formato otimizado, utilizando uma única tecnologia de armazenamento, convenções de nomenclatura consistentes e identificadores coerentes entre as fontes, o que o diferencia do *data lake*.

Lamer et al. (2024) complementa que o *data warehouse* funciona como um repositório unificado e normalizado para dados e metadados, mantendo as informações em formato orientado a linhas. Além disso, o sistema é definido como uma coleção de dados orientada por assunto, não volátil, integrada e variante no tempo, onde os dados são armazenados persistentemente, sem suposições sobre seu uso futuro, permanecendo abertos para diversas aplicações.

2.11 Paradigma orientado a eventos

No paradigma orientado a eventos, os sistemas comunicam-se produzindo e consumindo eventos que representam mudanças significativas de estado ou ocorrências dentro de um sistema. Eventos são registros imutáveis que descrevem o que aconteceu, quando aconteceu e, frequentemente, incluem metadados contextuais (ABDI et al., 2025).

Lazzari e Farias (2023) complementa apresentando que, neste paradigma, os componentes produtores publicam os dados sem necessariamente conhecer os demais componentes ou quais deles consumirão e reagirão aos dados publicados. Tal abordagem promove

a separação entre a computação, a publicação do evento e qualquer processamento subsequente.

Além disso, o processo de comunicação entre produtor e consumidor ocorre de forma assíncrona, e ambos permanecem independentes entre si. Consequentemente, essa dinâmica promove o baixo acoplamento (*loose coupling*) entre os componentes da aplicação, razão pela qual a arquitetura orientada a eventos tornou-se predominante em cenários de aplicações distribuídas de larga escala (LAZZARI; FARIAS, 2023).

Na Figura 2, é possível visualizar como este tipo de arquitetura opera e como os eventos (neste trabalho, os dados pré-processados) são distribuídos entre os consumidores ligados à mensageria.

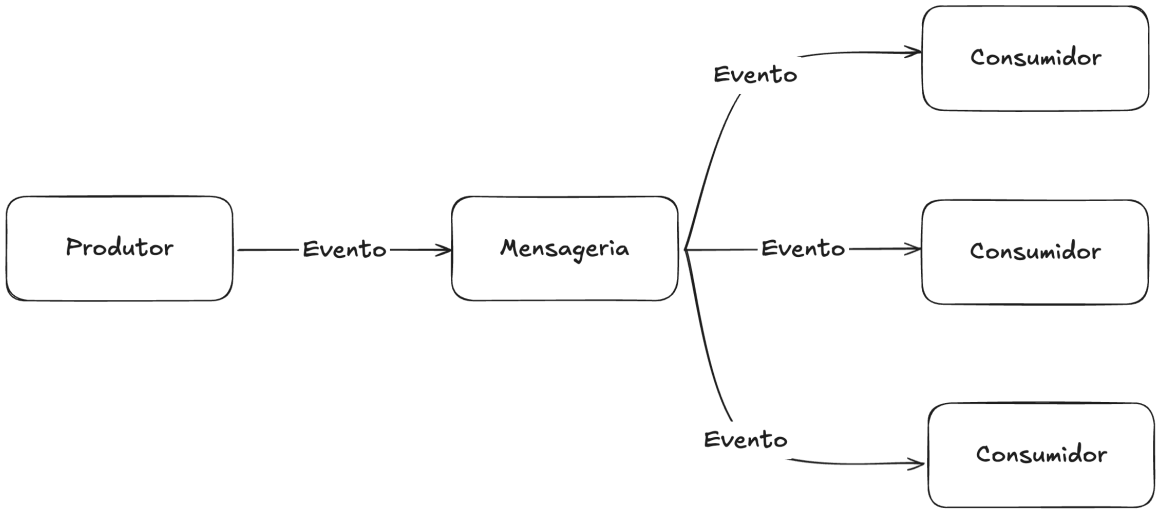

Figura 2 – Exemplo de uma arquitetura aplicando o paradigma orientado a eventos.

2.12 Tecnologias Utilizadas

2.12.1 *Python*

Python[1] é uma linguagem de programação de alto nível, interpretada, de *script*, imperativa, orientada a objetos, funcional, de tipagem dinâmica e forte. Foi lançada por Guido van Rossum em 1991 com o objetivo de ser uma linguagem de fácil leitura e aprendizado, com uma sintaxe limpa e clara, diferenciando-se de linguagens como C, que inspirou sua criação. Atualmente, é uma das linguagens mais populares do mundo, com uma vasta comunidade ativa e um ecossistema robusto de bibliotecas voltadas para análise de dados, desenvolvimento *web*, automação, aprendizado de máquina e inteligência artificial.

[1] <https://www.python.org/>

2.12.2 *Apache Kafka*

O *Apache Kafka*[2] é uma plataforma distribuída de transmissão de eventos (*event streaming*) de alto desempenho, que permite a publicação, subscrição, armazenamento e processamento de fluxos de dados em tempo real. Desenvolvido originalmente pelo *LinkedIn* e doado à *Apache Software Foundation* em 2011, o *Kafka* foi projetado para lidar com volumes massivos de dados com baixa latência e alta tolerância a falhas.

Sua arquitetura é fundamentada em um *log* de transações distribuído, organizado em tópicos e partições, o que permite a escalabilidade horizontal e a persistência dos dados de forma imutável.

2.12.3 *Docker*

O *Docker*[3] é uma plataforma de código aberto baseada no conceito de virtualização em nível de sistema operacional, que permite a criação, a implantação e a execução de aplicações dentro de contêineres isolados. Um contêiner permite que o desenvolvedor empacote uma aplicação juntamente com todas as suas dependências — como bibliotecas, arquivos de configuração e binários —, garantindo que o software seja distribuído como uma unidade padronizada, denominada imagem.

Diferente das máquinas virtuais tradicionais, que exigem um sistema operacional completo para cada instância, os contêineres *Docker* compartilham o núcleo (*kernel*) do sistema operacional hospedeiro. Isso resulta em uma infraestrutura significativamente mais leve, com inicialização rápida e menor consumo de recursos. O principal benefício desta tecnologia é a portabilidade, assegurando que a aplicação funcione de maneira idêntica e previsível em diferentes ambientes de computação, eliminando inconsistências entre os estágios de desenvolvimento, teste e produção.

2.12.4 *InfluxDB*

O *InfluxDB*[4] é um sistema de gerenciamento de banco de dados de séries temporais (*Time Series Database* — TSDB) de código aberto, projetado especificamente para lidar com altos volumes de dados rotulados com carimbos de tempo (*timestamps*). Desenvolvido pela *InfluxData* em 2013, o sistema é otimizado para cenários que exigem alta taxa de ingestão e consultas em tempo real, como monitoramento de infraestrutura, métricas de aplicações e telemetria de sensores de *IoT*.

A arquitetura do *InfluxDB* é estruturada em torno de pontos de dados, que consistem em um tempo, uma medição (*measurement*) e um conjunto de etiquetas (*tags*) e

[2] <https://kafka.apache.org/>
[3] <https://www.docker.com/>
[4] <https://www.influxdata.com/>

campos (*fields*). Essa organização permite uma indexação eficiente e uma compressão de dados superior em comparação com os bancos de dados relacionais tradicionais. Além de sua escalabilidade horizontal, a ferramenta oferece recursos nativos para o gerenciamento do ciclo de vida dos dados, como políticas de retenção (*retention policies*) e agregações contínuas, consolidando-se como uma solução robusta para o armazenamento de métricas que exigem baixa latência e alta disponibilidade.

2.12.5 *Golang*

Golang[5] é uma linguagem de programação de código aberto, compilada e estaticamente tipada, desenvolvida internamente pelo Google em 2007 e lançada para a comunidade em 2009. Criada por Robert Griesemer, Rob Pike e Ken Thompson, a linguagem foi projetada com o objetivo de unir a eficiência e a segurança de linguagens de baixo nível, como C++, à simplicidade e produtividade de linguagens de alto nível.

Um dos diferenciais fundamentais do *Golang* é o seu modelo de concorrência nativo, baseado em *goroutines* e canais (*channels*), que permite a execução de milhares de tarefas simultâneas com um consumo de memória extremamente reduzido. Devido à sua alta performance, gerenciamento automático de memória (*garbage collection*) e robustez no tratamento de protocolos de rede, a linguagem tornou-se o padrão para o desenvolvimento de infraestruturas de nuvem, sistemas distribuídos, microsserviços e aplicações de tempo real que exigem baixa latência e alta escalabilidade.

2.12.6 *Grafana*

O *Grafana*[6] é uma plataforma de código aberto voltada para a análise, monitoramento e visualização de dados de múltiplas fontes. Reconhecido por sua versatilidade em ambientes de observabilidade, o sistema permite a criação de painéis (*dashboards*) interativos e dinâmicos que consolidam métricas, logs e rastreamentos em uma interface unificada, facilitando a interpretação de grandes volumes de informações complexas.

A ferramenta destaca-se por sua ampla compatibilidade com diversos bancos de dados — especialmente os de séries temporais — e por sua capacidade de processamento de fluxos em tempo real. Além das funcionalidades de exibição gráfica, o Grafana oferece um motor de alertas robusto, capaz de identificar anomalias e notificar usuários sobre eventos críticos. Na engenharia de dados moderna, ele atua como a camada de *interface* final, convertendo dados brutos e processados em *insights* visuais que fundamentam a tomada de decisão estratégica em cenários de alta frequência e baixa latência.

[5] <https://go.dev/>
[6] <https://grafana.com/>

2.12.7 *MinIO*

O *MinIO*[7] é um servidor de armazenamento de objetos (*object storage*) de alto desempenho, distribuído sob licença de código aberto e totalmente compatível com a API do serviço *Amazon S3* (*Simple Storage Service*). Ele foi projetado para ser nativo da nuvem (*cloud-native*), permitindo o armazenamento e a gestão de grandes volumes de dados não estruturados, como arquivos de *log*, imagens, binários e *datasets* de telemetria.

Diferente dos sistemas de arquivos convencionais, o MinIO utiliza uma arquitetura baseada em objetos, o que o torna ideal para compor a camada de persistência dos *data lakes*. Sua principal característica é a alta vazão (*throughput*) e a eficiência em operações de leitura e escrita, sendo otimizado para lidar com cargas de trabalho intensivas em processamento de dados e inteligência artificial. Devido à sua portabilidade e leveza, a ferramenta é amplamente adotada em infraestruturas baseadas em contêineres, fornecendo uma solução de armazenamento escalável, resiliente e de fácil integração com ferramentas modernas de engenharia de dados.

2.12.8 *PostgreSQL*

O *PostgreSQL*[8] é um sistema de gerenciamento de banco de dados objeto-relacional de código aberto, amplamente reconhecido por sua confiabilidade, robustez de recursos e conformidade com os padrões SQL. Desenvolvido ao longo de mais de 35 anos, o sistema é focado na integridade dos dados e no suporte a cargas de trabalho complexas, permitindo o armazenamento de dados estruturados e semiestruturados.

Sua arquitetura é baseada no modelo de transações ACID, o que garante a precisão e a segurança das informações, mesmo em cenários de falha. Além de sua extensibilidade, o PostgreSQL destaca-se em ecossistemas de engenharia de dados como um repositório central para *data warehouses*, oferecendo suporte a consultas analíticas avançadas, indexação performática e integração facilitada com ferramentas de BI. Por ser altamente escalável e possuir uma comunidade ativa, consolidou-se como a principal escolha para persistência de dados que exigem consistência rigorosa e suporte a análises históricas multidimensionais.

2.13 Trabalhos relacionados

Esta seção apresenta trabalhos que abordam, em conjunto ou separadamente, os principais problemas tratados neste estudo: a coleta e transmissão de telemetria em ambientes de alta velocidade, o processamento de fluxos de dados em tempo real e a aplicação de arquiteturas orientadas a eventos para processamento dual — simultâneo em tempo

[7] <https://min.io/>
[8] <https://www.postgresql.org/>

real e em lote. A análise desses trabalhos evidencia lacunas que a presente proposta busca preencher, especialmente no que diz respeito à combinação de um domínio de aplicação específico — a telemetria da Fórmula 1 — com um conjunto de ferramentas moderno, de código aberto e orientado a eventos.

Tan (2011) propõe e implementa o sistema *Crystal Ball*, a camada de *software* de um sistema de telemetria para um carro de Fórmula SAE elétrico desenvolvido pela equipe REV da Universidade de *Western Australia*. O sistema é responsável por receber, interpretar e visualizar em tempo real os dados coletados do veículo, permitindo que engenheiros de corrida acompanhem o comportamento do carro durante os eventos. Tan descreve os desafios de lidar com diferentes tipos de sensores — temperatura, pressão, forças — e a necessidade de exibição contínua das métricas para suporte à decisão no *paddock*. A proximidade com o presente trabalho é direta: ambos tratam da camada de visualização em tempo real de dados de telemetria automotiva voltados ao ambiente de corrida. A diferença central está no escopo arquitetural — enquanto o *Crystal Ball* se concentra na *interface* de visualização como produto final, o presente trabalho trata essa camada como apenas um dos módulos de um fluxo distribuído e orientado a eventos, acrescentando uma camada de persistência histórica para análises *post-mortem*.

Banerjee et al. (2022) descrevem um sistema de aquisição de dados, registro e telemetria ao vivo para um carro de fórmula, utilizando o protocolo CAN como espinha dorsal da comunicação e o microcontrolador *myRIO* programado em *LabVIEW*. Os autores destacam que um carro de Fórmula 1 abriga mais de cem sensores durante as corridas e que o sistema de aquisição de dados, embora não afete diretamente o desempenho do carro, é indispensável na fase de testes e desenvolvimento, fornecendo ao piloto e à equipe informações sobre o comportamento do veículo em tempo real. O trabalho compartilha com esta dissertação o mesmo domínio-problema — a telemetria de carros de fórmula —, mas adota uma abordagem centrada em *hardware* embarcado e em protocolos de rede veicular. Em contraste, o presente trabalho opera sobre dados simulados via UDP e prioriza uma arquitetura de sistemas distribuída, demonstrando que soluções de telemetria de alto desempenho podem ser construídos sobre infraestruturas modernas, sem dependência de *hardware* proprietário ou especializado.

Lima et al. (2023) apresentam a concepção e implementação de uma rede de comunicação para compor um sistema de aquisição de dados e telemetria para um carro de Fórmula SAE, utilizando componentes de baixo custo. O trabalho aborda os desafios de integrar sensores, protocolos de comunicação e ferramentas de análise dentro das limitações orçamentárias típicas de equipes universitárias de competição. Assim como o presente trabalho, os autores enfrentam o problema de como tornar os dados do veículo acessíveis em tempo real para a equipe técnica, priorizando custo reduzido e abertura de tecnologias. A diferença fundamental reside na escala arquitetural: enquanto Lima et al. (2023)

focam na camada de transmissão e aquisição física dos dados, a presente proposta assume essa camada como resolvida — pelo simulador e pelo módulo de pré-processamento — e concentra sua contribuição na orquestração dos dados via mensageria e no processamento dual pelas camadas rápida e em lote.

Kiran et al. (2015) investigam a aplicação da arquitetura lambda em um contexto de segurança de fronteiras e monitoramento de redes, utilizando os serviços da AWS para implementar as três camadas do modelo. Os autores discutem as compensações de custo e desempenho entre o processamento em lote e o processamento em tempo real, concluindo que a arquitetura *lambda* permite otimizar custos ao identificar quais partes dos dados exigem análise imediata e quais podem aguardar o processamento em lote. A relevância deste trabalho para a presente proposta é dupla: primeiro, valida a escolha da arquitetura *lambda* como solução para cenários de processamento dual; segundo, demonstra a generalidade do modelo, aplicável a domínios distintos da Fórmula 1. A principal distinção está no contexto de aplicação e nas tecnologias empregadas — enquanto Kiran et al. (2015) utiliza infraestrutura de nuvem proprietária, este trabalho implementa a arquitetura *lambda* de ponta a ponta com ferramentas de código aberto em ambiente conteinerizado gratuitamente, demonstrando a viabilidade do modelo sem dependência de provedores externos.

Khattach, Moussaoui e Hassine (2025) propõem uma arquitetura abrangente de ponta a ponta para o processamento de fluxos de dados de sensores IoT, integrando *Apache Kafka* para ingestão de dados de alta frequência, *Apache Spark* para ETL em lote e em tempo real, e *InfluxDB* para armazenamento de séries temporais com análise histórica. O trabalho valida experimentalmente a capacidade dessa combinação tecnológica de processar dados de sensores com baixa latência, aplicar modelos de aprendizado de máquina para detecção de falhas e entregar visualizações em tempo real. A sobreposição com o presente trabalho é notável: a *stack* tecnológica central — *Kafka*, *InfluxDB* e a dicotomia entre camadas de tempo real e em lote — é compartilhada em sua essência. A diferença está no domínio de aplicação — manutenção preditiva industrial versus telemetria de Fórmula 1 — e na fonte de dados: sensores industriais genéricos versus pacotes UDP com contrato de dados proprietário do simulador de corridas, que exigem um módulo de pré-processamento especializado em *Golang* para decodificação binária em formato *Little Endian* (formato de dados binário) antes da publicação no barramento de eventos.

3 Arquitetura proposta

3.1 Visão da arquitetura conceitual da aplicação

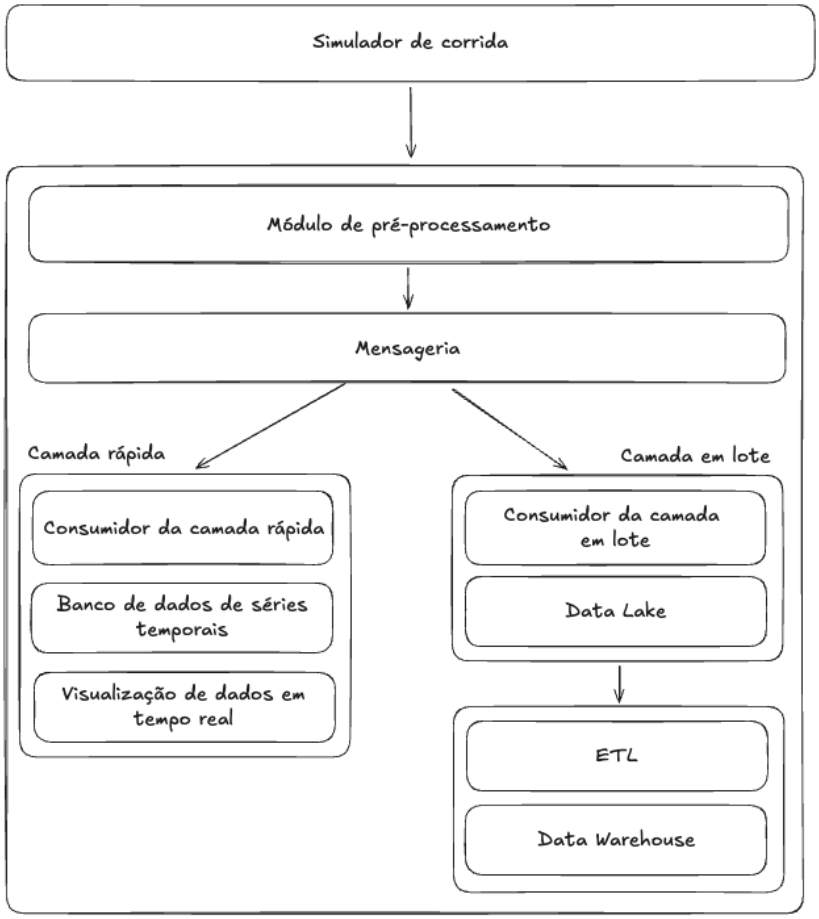

Figura 3 – Figura ilustrativa da arquitetura conceitual do fluxo de dados

A arquitetura proposta compreende os seguintes módulos e fluxo de dados:

- **Fonte de dados:** O simulador **F1 23** será utilizado como a fonte primária para simular a geração de dados de telemetria em tempo real, transmitindo-os via protocolo **UDP**.
- **Módulo de pré-processamento:** O primeiro módulo da aplicação será responsável por capturar os dados UDP brutos do jogo. Este módulo realizará a etapa de extração, lendo os pacotes de dados, e a primeira fase de **transformação**, convertendo os valores binários em *Little Endian* para JSON (formato de dados chave-valor) e estruturando-os para processamento. Essa tradução em JSON ocorre para fins de

estudo, a fim de facilitar a implementação, embora possa trazer uma latência um pouco maior à medida que mais pacotes são disparados. Como cada pacote tem uma maneira distinta de ser decodificado por conta do contrato de dados, este módulo precisa direcionar os pacotes certos aos tradutores adequados. Para simplificação do experimento, este trabalho tem seu escopo limitado apenas aos dados de telemetria do carro, ou seja, onde o ID do pacote é igual a 6, sendo possível expandir futuramente.

- **Mensageria:** Após a extração e transformação iniciais, os dados serão publicados em tópicos da mensageria, servindo como uma plataforma de *streaming* de dados. Esta garantirá a transmissão eficiente, tolerante a falhas e escalável dos dados em tempo real.

- **Módulo de camada rápida (*speed layer*):** Este atuará como um dos consumidores da mensageria, lendo os dados vindos da etapa anterior. Este consumidor será responsável por carregar os dados em um banco de dados de séries temporais, permitindo o armazenamento e a recuperação eficientes de grandes volumes de dados de telemetria. Nesta mesma etapa, para a visualização em tempo real, um *dashboard* interativo será desenvolvido e consumirá os dados diretamente do banco temporal, exibindo as métricas processadas em tempo real e fornecendo uma interface visual para acompanhamento e análise.

- **Módulo de camada em lote (*batch layer*):** Um segundo consumidor conecta-se à mensageria para armazenar os dados do carro em um *data lake* local, que será útil, ao final da corrida, para criar o armazém de dados. Após o término do Grande Prêmio, a segunda parte do módulo atuará fora do contexto de tempo real, processando os documentos brutos, realizando a limpeza completa e armazenando-os em um banco de dados relacional, a fim de alimentar modelos de *machine learning* e IA futuramente.

3.2 Ferramentas e tecnologias utilizadas

Para a implementação da *pipeline*, serão utilizadas as seguintes tecnologias de código aberto:

- **Linguagens de programação:** *Python* e *Golang*
- **Orquestração de contêineres:** *Docker*
- **Plataforma de *streaming* de dados:** *Apache Kafka*.
- **Banco de dados de séries temporais:** *InfluxDB*.

- **Ferramenta para *data lake*:** *MinIO.*
- **Banco de dados para *data warehouse*:** *PostgreSQL.*
- **Ferramenta para *dashboards* interativos:** *Grafana.*
- **Fonte de dados de telemetria:** Simulador *F1 23.*

3.3 Estrutura dos dados da telemetria

O simulador *F1 23* disponibiliza uma funcionalidade específica para a transmissão de dados de telemetria via rede. Esse recurso permite o envio de diferentes categorias de informações, incluindo dados do veículo, do piloto, da sessão e das condições meteorológicas.

Originalmente, tal funcionalidade foi desenvolvida para possibilitar a integração de periféricos para simuladores de corrida, permitindo que fabricantes conectassem seus dispositivos ao simulador a fim de receber informações em tempo real, como parâmetros do veículo, e reproduzissem-nas em equipamentos físicos, tais como volantes que simulam os utilizados em carros de Fórmula 1. Na Figura 4, é ilustrado como esses dados são exibidos em um volante de corrida virtual para leitura rápida do piloto.

Figura 4 – Figura ilustrativa de um volante de *Sim Racing*. Adaptado de RMS Sim Racing.

Além dessa aplicação original, o recurso de telemetria também pode ser utilizado para realizar a coleta estruturada de dados do veículo. Qualquer aplicação ou dispositivo conectado à mesma rede do simulador, desde que capaz de receber e interpretar pacotes UDP transmitidos, pode processar essas informações conforme a finalidade desejada.

A documentação oficial da especificação UDP[1] do *F1 23* apresenta o detalhamento completo da estrutura dos pacotes transmitidos, incluindo o conteúdo do cabeçalho — responsável pela identificação do tipo de pacote — e a descrição de cada *payload*. Para os fins deste trabalho, serão utilizados exclusivamente os dados de telemetria veicular, especificamente: velocidade, ângulo de esterçamento do volante, níveis de aceleração e frenagem, marcha atual, entre outros parâmetros relevantes.

3.4 Módulo de pré-processamento

Este é o primeiro e mais crítico módulo componente da arquitetura proposta. Concebido como o ponto de entrada da *pipeline*, este módulo atua como uma vanguarda entre o simulador e a mensageria. A escolha de um componente dedicado para esta função fundamenta-se na necessidade de decodificar os pacotes UDP em *Little Endian* para um formato universal antes de serem publicados na mensageria, garantindo a interoperabilidade dos componentes a seguir. Na Figura 5, está mostrado como todo esse processo deve acontecer de maneira abstrata.

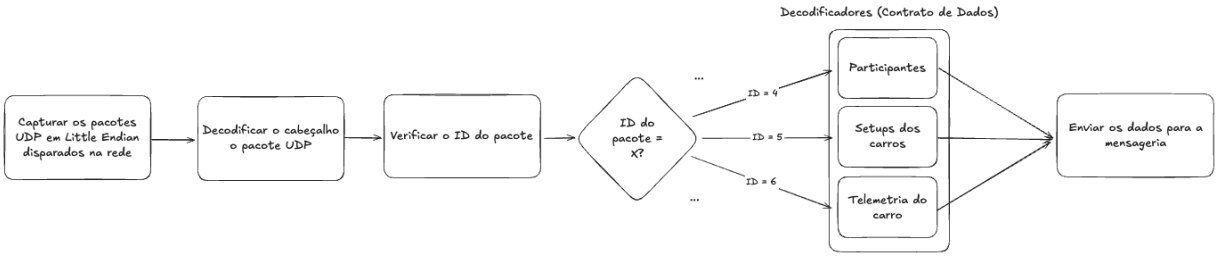

Figura 5 – Figura ilustrativa do diagrama do módulo de pré-processamento

3.4.1 Captura de dados

O funcionamento da captura de dados se baseará na abertura de um *listener* de rede configurado para capturar pacotes baseados no protocolo UDP. Diferente dos protocolos orientados a conexão, a escolha do UDP pelo simulador *F1 23* visa ao desempenho máximo, abdicando de verificações de entrega em favor da velocidade de transmissão.

Estrategicamente, o módulo reservará uma porta de rede específica no sistema hospedeiro para atuar como receptor. Por padrão, adota-se a porta 20777, em conformidade com as especificações técnicas do simulador, embora a arquitetura permita a parametrização de IPs e portas alvo para garantir flexibilidade em redes locais distintas. O papel do conector, portanto, limita-se a validar a integridade básica do pacote recebido

[1] Link para o *download* da documentação: <https://forums.ea.com/discussions/f1-23-en/f1-23-udp-specification/8390745>

e encaminhá-lo imediatamente para a camada de tradução e produção, garantindo que o fluxo de entrada nunca seja bloqueado por processamentos pesados.

Dada a natureza crítica deste componente como "raiz"da *pipeline*, sua estabilidade operacional é mandatória. A lógica de interrupção implementada prevê que, caso o *listener* de rede não possa ser estabelecido — cenário comum quando a porta alvo já está vinculada a outro processo do sistema operacional —, a aplicação deve ser interrompida de forma segura e imediata.

Essa decisão de *design* visa prevenir a execução de uma *pipeline* em estado onde os serviços de processamento e visualização estariam ativos, porém sem fonte de dados. Assim, o conector UDP não apenas realiza a ponte de dados, mas também atua como um validador de integridade do ambiente de execução, assegurando que o processamento só ocorra quando o canal de comunicação com o simulador estiver plenamente estabelecido.

3.4.2 Tradução e produção dos eventos

Após a captura do pacote de rede pelo conector UDP, a *pipeline* ingressa em sua fase semântica. A etapa de tradução e produção de eventos é responsável por converter os pacotes em *Little Endian* em estruturas *JSON*. A escolha da linguagem *Golang* para este módulo foi estratégica, fundamentada em sua eficiência na manipulação de memória e, primordialmente, em seu robusto sistema de tipos por meio de *structs*; com ela, é possível transformar os pacotes binários em um formato intermediário tipado na memória, com os valores mapeados. Por um ângulo técnico, a Figura 6 ilustra como todo esse processo ocorre em tempo de execução.

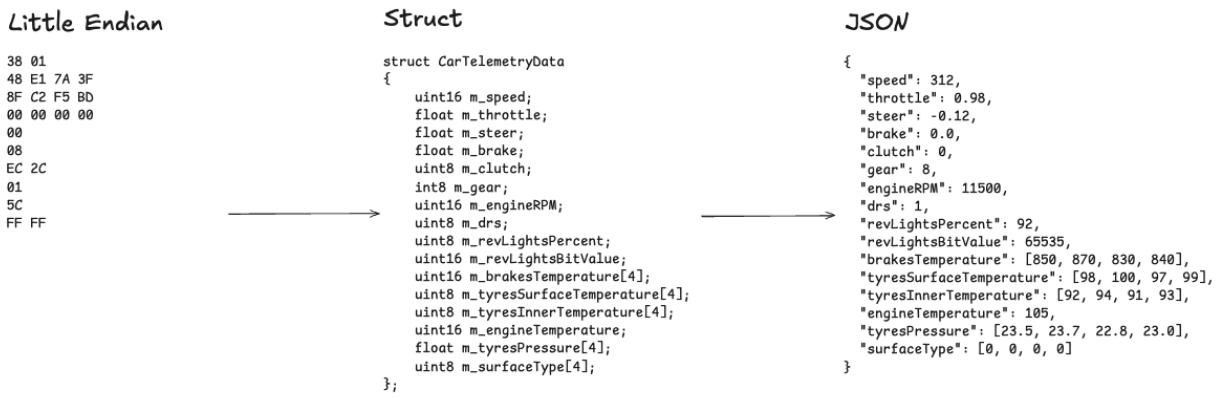

Figura 6 – Figura ilustrativa do diagrama da tradução dos pacotes

A modelagem dos dados neste projeto não deve ser arbitrária; ela deve refletir fielmente o contrato de dados definido na documentação oficial de telemetria do *F1 23*. O

uso de *structs* em *Golang* permite que o mapeamento entre os *offsets* binários do pacote e os campos da aplicação ocorra com custo computacional mínimo.

Conforme ilustrado na Figura 7, o protocolo exige que cada transmissão seja precedida por um cabeçalho padrão que fornece metadados essenciais sobre a origem e a natureza da informação.

```
struct PacketHeader
{
    uint16    m_packetFormat;            // 2023
    uint8     m_gameYear;                // Game year - last two digits e.g. 23
    uint8     m_gameMajorVersion;        // Game major version - "X.00"
    uint8     m_gameMinorVersion;        // Game minor version - "1.XX"
    uint8     m_packetVersion;           // Version of this packet type, all start from 1
    uint8     m_packetId;                // Identifier for the packet type, see below
    uint64    m_sessionUID;              // Unique identifier for the session
    float     m_sessionTime;             // Session timestamp
    uint32    m_frameIdentifier;         // Identifier for the frame the data was retrieved on
    uint32    m_overallFrameIdentifier;  // Overall identifier for the frame the data was retrieved
                                         // on, doesn't go back after flashbacks
    uint8     m_playerCarIndex;          // Index of player's car in the array
    uint8     m_secondaryPlayerCarIndex; // Index of secondary player's car in the array (splitscreen)
                                         // 255 if no second player
};
```

Figura 7 – Estrutura de dados do cabeçalho (*PacketHeader*) conforme especificação do simulador.

A implementação utiliza a tipagem forte para garantir a integridade dos dados desde a origem. Tipos primitivos, como `uint16` para velocidade e `float32` para pressões de pneus, são mapeados diretamente, evitando erros de precisão que poderiam comprometer as análises estratégicas subsequentes.

Devido ao escopo definido anteriormente, uma filtragem eficiente é imperativa para que a solução esteja de acordo com o principal requisito: coletar apenas os dados de telemetria do carro. O módulo opera em uma abordagem de decodificação multinível:

1. **Decodificação de cabeçalho:** O sistema realiza o *parsing* inicial apenas dos metadados (*Header*).
2. **Filtragem por identificador:** O campo *m_packetId* (ou `PacketID` na implementação) é verificado. Para os escopos desta pesquisa, apenas pacotes com identificador igual a 6 — correspondentes à telemetria detalhada do carro — são processados.
3. **Decodificação especialista:** Uma vez validado o identificador, o restante do *buffer* é passado para uma *struct* especialista (`PacketCarTelemetryData`), que extrai os valores brutos para campos legíveis por humanos e sistemas de análise (Figura 8).

```
struct CarTelemetryData
{
    uint16    m_speed;                    // Speed of car in kilometres per hour
    float     m_throttle;                 // Amount of throttle applied (0.0 to 1.0)
    float     m_steer;                    // Steering (-1.0 (full lock left) to 1.0 (full lock right))
    float     m_brake;                    // Amount of brake applied (0.0 to 1.0)
    uint8     m_clutch;                   // Amount of clutch applied (0 to 100)
    int8      m_gear;                     // Gear selected (1-8, N=0, R=-1)
    uint16    m_engineRPM;                // Engine RPM
    uint8     m_drs;                      // 0 = off, 1 = on
    uint8     m_revLightsPercent;         // Rev lights indicator (percentage)
    uint16    m_revLightsBitValue;        // Rev lights (bit 0 = leftmost LED, bit 14 = rightmost LED)
    uint16    m_brakesTemperature[4];     // Brakes temperature (celsius)
    uint8     m_tyresSurfaceTemperature[4]; // Tyres surface temperature (celsius)
    uint8     m_tyresInnerTemperature[4]; // Tyres inner temperature (celsius)
    uint16    m_engineTemperature;        // Engine temperature (celsius)
    float     m_tyresPressure[4];         // Tyres pressure (PSI)
    uint8     m_surfaceType[4];           // Driving surface, see appendices
};

struct PacketCarTelemetryData
{
    PacketHeader         m_header;             // Header

    CarTelemetryData     m_carTelemetryData[22];

    uint8                m_mfdPanelIndex;      // Index of MFD panel open - 255 = MFD closed
                                               // Single player, race – 0 = Car setup, 1 = Pits
                                               // 2 = Damage, 3 =  Engine, 4 = Temperatures
                                               // May vary depending on game mode
    uint8                m_mfdPanelIndexSecondaryPlayer;   // See above
    int8                 m_suggestedGear;      // Suggested gear for the player (1-8)
                                               // 0 if no gear suggested
};
```

Figura 8 – Mapeamento da *struct* de telemetria do carro para tipos primitivos.

3.4.3 Produção do evento de telemetria do carro

A etapa final deste módulo consiste na "produção"do evento (novo registro da telemetria coletado e tratado). Após a decodificação binária bem-sucedida, os dados são encapsulados em um objeto de *payload* simplificado, formatado em JSON. Esta transição de binário para JSON é crucial na arquitetura orientada a eventos, pois transforma um dado proprietário de rede em um formato de intercâmbio universal, facilitando o consumo por diferentes ferramentas, como o barramento *Apache Kafka* e bancos de dados de séries temporais.

O módulo, então, atua como um produtor, despachando o evento para o tópico correspondente na mensageria. Este desacoplamento garante que o tradutor não precise aguardar a gravação no banco de dados para processar o próximo pacote UDP, sustentando a premissa de processamento em tempo real.

3.5 Mensageria

No centro da arquitetura orientada a eventos desenvolvida, situa-se o subsistema de mensageria, implementado através do *Apache Kafka*. Este componente atua como o

sistema nervoso central da *pipeline*, sendo responsável pela orquestração, persistência temporária e distribuição dos eventos de telemetria processados. A inclusão de um serviço de mensageria de alta performance justifica-se pela necessidade de gerenciar o fluxo massivo de dados provenientes do simulador, sem comprometer a integridade da captura original.

3.5.1 Desacoplamento temporal e funcional

Uma das principais premissas desta arquitetura é o desacoplamento absoluto entre o módulo produtor (pré-processador em *Golang*) e os módulos consumidores (camadas rápida e em lote, ambos em *Python*). O uso do *Kafka* permite o que a literatura define como desacoplamento temporal: o produtor pode publicar dados a uma velocidade superior à capacidade de processamento imediata dos consumidores, uma vez que o *broker* atua como um *buffer* resiliente.

Funcionalmente, este desacoplamento significa que o módulo de pré-processamento não possui conhecimento sobre o destino final dos dados. Sua única responsabilidade é entregar os dados limpos, enquanto a mensageria garante que o evento traduzido chegará ao tópico correto, o que simplifica a lógica do componente e reduz o esforço computacional de escrita, permitindo que o sistema suporte as altas taxas de atualização exigidas pela telemetria da Fórmula 1.

3.5.2 Estratégia de tópicos e distribuição de carga

Dentro da arquitetura, os dados são organizados em tópicos, que funcionam como categorias lógicas de eventos. O tópico principal desta solução armazena os pacotes de telemetria do carro (ID 6) em seu estado traduzido para JSON.

Diferente dos sistemas de fila tradicionais, o *log* de eventos do *Kafka* permite que os mesmos dados sejam lidos simultaneamente por múltiplos consumidores independentes, sem que a leitura de um interfira na posição de leitura do outro. Esta característica é o pilar que sustenta a replicação dos dados em seu estado original para frentes de trabalho com objetivos distintos, garantindo que a "verdade única"do evento seja preservada em toda a rede.

3.5.3 Dualidade de processamento: camada rápida vs. camada em lote

A arquitetura proposta tira proveito do modelo produtor-consumidor para implementar dois caminhos de processamento distintos, originados da mesma fonte de dados:

- **Camada rápida (*speed layer*):** Consumidor focado em baixa latência, responsável por alimentar o banco de dados de séries temporais *InfluxDB*. Este caminho

provê dados quase instantâneos para o *paddock* virtual, permitindo a visualização em tempo real da performance do veículo através de *dashboards* dinâmicos.

- **Camada em lote (*batch layer*):** Consumidor focado em durabilidade e análise pós-corrida, responsável por persistir os dados brutos no *data lake* (*MinIO*). Estes dados são fundamentais para o armazenamento de longo prazo, permitindo auditorias de corrida, treinamento de modelos de IA para predição de desgaste de componentes e revisões estratégicas de pilotagem.

Esta dualidade assegura que a solução atenda tanto às necessidades operacionais imediatas do engenheiro de corrida quanto às demandas analíticas de longo prazo da equipe técnica, conferindo robustez e escalabilidade à aplicação.

3.6 Camada rápida (*speed layer*)

A camada rápida, ou *speed layer*, constitui o segmento da *pipeline* de dados focado em baixa latência e processamento quase instantâneo (*near real-time*). Dentro da arquitetura orientada a eventos proposta, este fluxo é responsável por transformar os eventos brutos publicados no barramento de mensageria em inteligência acionável para o corpo de engenharia no *paddock*. A denominação "rápida"refere-se à alta frequência de atualização e à necessidade de que os dados cheguem ao seu destino final enquanto ainda possuem valor estratégico operacional.

Diferente dos sistemas de processamento em lote (*batch processing*), onde a prioridade é a volumetria, a camada rápida é regida por restrições temporais severas. Nesta solução, estabeleceu-se um limite de latência total de no máximo 5 segundos, devido a limitações de *hardware* e banda larga local — compreendendo desde a emissão do pacote UDP pelo simulador até a renderização visual no terminal de análise. Este requisito é fundamental para permitir que decisões táticas, como o ajuste do *setup* eletrônico do carro ou a definição da janela de *pit stop*, sejam embasadas em dados que reflitam o estado atual do veículo na pista e não sua condição em voltas anteriores.

Para sustentar este desempenho, este módulo utiliza componentes especializados em processamento de fluxo e armazenamento de séries temporais. A estratégia fundamenta-se no processamento sem estado (*stateless*) no nível do consumidor, garantindo que cada evento de telemetria seja tratado de forma individual e paralela. Esta abordagem evita o acúmulo de atrasos decorrentes de janelas de processamento complexas, assegurando que a vazão (*throughput*) do sistema acompanhe a taxa de disparo de sensores do simulador.

Abaixo, detalham-se os componentes internos que integrarão esta via de processamento, abrangendo o consumo da fila, a persistência otimizada e a camada de *interface* homem-máquina.

3.6.1 Consumidor para *dashboard*

O papel do consumidor na camada rápida é atuar como um subscritor especializado dentro da topologia de mensageria. Sua função primordial é manter uma conexão persistente com o barramento de eventos, agindo como um mediador entre o fluxo de mensagens brutas e o armazenamento otimizado para visualização.

A lógica de projeto para este módulo fundamenta-se na premissa da simplicidade de transformação. Uma vez que os dados de telemetria já circulam no sistema em formato JSON, o consumidor não precisa despender recursos computacionais em decodificações complexas. O foco operacional desloca-se, portanto, para a organização semântica: o componente identifica os campos de interesse (como velocidade, rotação do motor e temperatura) e os organiza segundo o modelo de dados exigido pelo banco de dados de séries temporais.

Diferente de uma abordagem de persistência convencional, o raciocínio aplicado aqui trata o dado como um "evento discreto no tempo". O consumidor assegura que cada pacote recebido seja carimbado com sua respectiva referência temporal, permitindo que o sistema de visualização reconstrua a telemetria do carro de forma linear e contínua.

Esta estratégia de "escuta e carga"é o que permite que a aplicação cumpra o requisito de baixa latência. Ao isolar a lógica de inserção no banco de dados em um módulo independente, a arquitetura garante que, mesmo sob condições de tráfego intenso no simulador, o fluxo para o *dashboard* permaneça fluido, fornecendo aos engenheiros uma visão fiel e imediata do comportamento do veículo na pista.

3.6.2 Banco de dados de séries temporais

Os eventos tratados pelo consumidor da camada rápida são persistidos em um banco de dados especificamente projetado para séries temporais (*Time-Series Database*). Esta escolha arquitetural é fundamental para garantir que cada métrica capturada — seja ela a rotação do motor ou a pressão dos pneus — possua um registro cronológico preciso do exato momento em que o dado foi gerado e processado pela *pipeline*. Em um domínio onde a telemetria é transmitida em frequências que podem variar de 60 a 120 Hz, a dimensão temporal torna-se o índice primário e mais crítico para a integridade da análise.

A principal justificativa para a utilização deste modelo de armazenamento, em detrimento de bancos de dados relacionais convencionais, reside na sua eficiência operacional de escrita e leitura. Durante uma sessão de corrida, o volume de pacotes recebidos é massivo e ininterrupto; portanto, o banco de dados deve possuir uma alta taxa de ingestão (*throughput*) para evitar que o consumidor de mensagens sofra retenções ou perdas de dados devido a bloqueios de I/O. A estrutura de um banco de séries temporais é oti-

mizada para inserções sequenciais, permitindo que o sistema suporte a carga máxima do simulador sem degradar a performance global da aplicação.

Além da robustez na escrita, a rapidez na recuperação dos dados é vital para a utilidade do sistema no *paddock*. O motor de consulta deste banco é otimizado para extrair volumes significativos de dados históricos em frações de segundo, o que possibilita a geração de visualizações analíticas complexas e essenciais para o engenheiro de pista, tais como:

- **Telemetria comparativa:** Sobreposição de gráficos de aceleração *versus* frenagem para identificar a eficiência do piloto em zonas de frenagem.
- **Evolução de velocidade:** Monitoramento da velocidade média e máxima em diferentes setores da pista para ajustes de *setup* aerodinâmico.
- **Análise de transmissão:** Mapeamento das trocas de marchas ao longo do tempo, auxiliando na identificação de padrões de condução e desgaste mecânico.

Dessa forma, a rapidez de resposta do banco de dados assegura que o acesso à informação ocorra de forma quase instantânea. Esta característica é o que permite que a decisão estratégica seja tomada com base no estado atual do veículo, garantindo que o fluxo de informação entre o carro e o engenheiro ocorra sem gargalos tecnológicos, preservando a fidelidade necessária para o ambiente de competição.

3.6.3 Visualização dos dados

A etapa final desta camada consiste na materialização dos dados através de uma camada de visualização dinâmica. Este componente representa a *interface* entre o sistema de processamento e o corpo técnico de engenharia, tendo como objetivo primordial a exibição fiel e em tempo real da telemetria capturada. Mais do que uma simples demonstração funcional da *pipeline*, esta seção da arquitetura é concebida como uma ferramenta de suporte à decisão, transformando métricas brutas em indicadores visuais acionáveis.

Para que o sistema cumpra seu propósito operacional, o *dashboard* deve apresentar um elevado grau de fidelidade temporal. Isso implica que as oscilações de sensores no simulador devem ser refletidas nos gráficos de forma quase instantânea, permitindo que os engenheiros acompanhem a evolução do veículo durante cada volta. A percepção de continuidade no fluxo de dados é essencial para a identificação de anomalias térmicas, picos de rotação ou falhas de pressão, situações que exigem uma resposta assertiva e imediata para garantir a integridade do carro e a performance na competição.

Sob a ótica de engenharia de *software*, a ferramenta selecionada para esta função deve atender a requisitos estritos de eficiência e interoperabilidade. É imperativo que a

solução possua capacidade de integração nativa com o banco de dados de séries temporais, estabelecendo uma conexão persistente que permita a consulta contínua (*polling* ou *streaming*) sem a necessidade de intermediários que possam elevar a latência do sistema.

Ademais, prioriza-se uma *interface* que seja leve e de rápida configuração, minimizando o consumo de recursos do sistema hospedeiro e facilitando a manutenção da lógica de visualização. Ao disponibilizar gráficos de fácil interpretação — como comparativos de aceleração por frenagem e mapas de marchas —, a camada de visualização encerra o ciclo da camada rápida, garantindo que o valor contido no dado original seja entregue ao usuário final com a rapidez e a precisão exigidas pelo ambiente de *Sim Racing*.

3.7 Camada em lote (*batch layer*)

Em contraste com a urgência temporal da camada rápida, a camada em lote (*batch layer*) é projetada para garantir a durabilidade, a integridade e a profundidade analítica dos dados coletados. Esta vertente da arquitetura atua como o repositório histórico da solução, servindo não apenas como um sistema de *backup* resiliente, mas como a fundação para processos de auditoria, análise de performance comparativa e treinamento de modelos de IA. Enquanto a camada rápida foca no "agora", o camada em lote dedica-se a "contar a história"completa do comportamento do piloto e do veículo ao longo de toda a sessão de corrida.

3.7.1 *Buffering* e *data lake*

A primeira fase desta trilha consiste na captura e armazenamento dos eventos em seu estado original e decodificado (bruto). Para evitar o custo computacional de operações de escrita fragmentadas em sistemas de arquivos, a arquitetura utiliza uma estratégia de *buffering* e *micro-batching*. Os dados provenientes do barramento de mensageria são acumulados em memória por um consumidor especializado. Este acúmulo persiste até que um dos dois gatilhos configurados seja acionado: a ocupação total do limite do *buffer* ou a expiração de um intervalo de tempo pré-determinado.

Uma vez atingido o gatilho, o módulo despacha o conjunto de eventos em formato JSON para o *data lake*, implementado através do sistema de armazenamento de objetos *MinIO*. Esta abordagem de "escrever uma vez e ler várias"(*write-once-read-many*) garante que os dados permaneçam imutáveis e disponíveis para qualquer necessidade futura de reprocessamento. O uso do formato JSON nesta etapa é estratégico, pois preserva a flexibilidade do esquema original do simulador, permitindo que novas variáveis de telemetria sejam adicionadas sem a necessidade de reestruturação imediata do armazenamento.

3.7.2 Processamento pós-corrida e integração relacional

A segunda etapa desta camada ocorre de forma assíncrona, preferencialmente após o encerramento da sessão de corrida, quando os recursos computacionais podem ser redirecionados da ingestão para a transformação. Este processo, caracterizado como uma *pipeline* de ETL, tem como objetivo converter os arquivos JSON esparsos em um modelo de dados relacional estruturado.

Durante essa transformação, o sistema realiza a manipulação completa dos dados. Informações que no JSON original estavam dispersas são agora consolidadas em tabelas relacionais, permitindo o cruzamento com outras fontes de dados, como tabelas de circuitos, histórico de pilotos e condições meteorológicas de sessões passadas. A transição para um modelo relacional é mandatória para permitir consultas SQL complexas, que seriam inviáveis ou extremamente custosas de serem executadas diretamente sobre arquivos de texto bruto no *data lake*, sem uma estrutura adequada.

4 Implementação da solução

4.1 Arquitetura detalhada da implementação

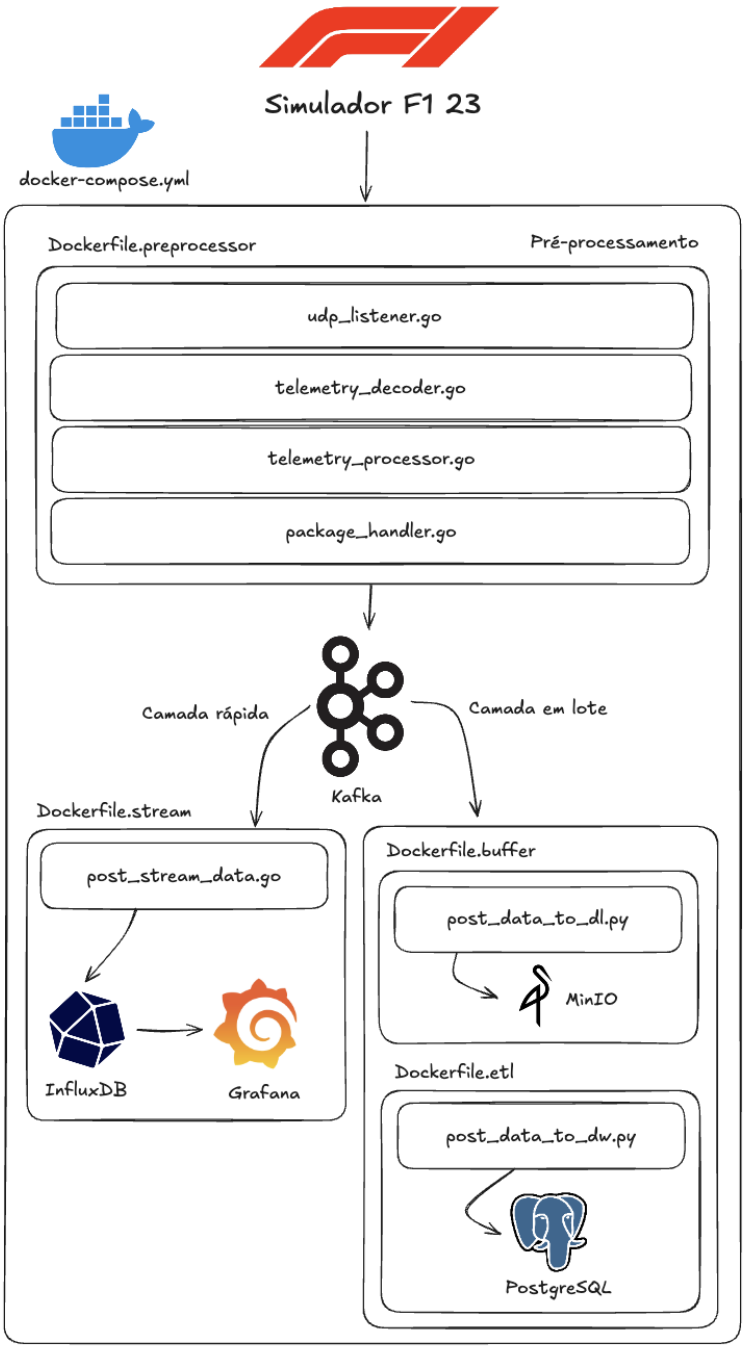

Figura 9 – Figura da arquitetura detalhada da solução

A Figura 9 ilustra graficamente como a solução foi implementada na prática. Todos os códigos completos e a configuração dos serviços utilizados na aplicação podem ser encontrados no repositório do *GitHub*[1].

O `docker-compose.yml` é responsável por intermediar a conexão com o simulador, subir os contêineres e gerenciar toda a aplicação, garantindo que ela possa ser executada da mesma forma, independentemente do *hardware*.

Todos os módulos são encapsulados por um `Dockerfile`; ele garantirá a ativação de cada módulo, especificando detalhadamente como cada um deve iniciar e operar. Isso permite a divisão da aplicação em pequenos componentes, facilitando a manutenção e a operação.

O primeiro módulo que é chamado é o conector UDP; ele ficará ativo a todo momento, capturando os dados emitidos pelo simulador. Dentro dele, encontramos o `udp_listener.go`, responsável por expor uma porta do hospedeiro à rede e coletar os pacotes UDP da telemetria.

Em sequência, o módulo de tradução; dentro dele, temos os arquivos:

- `telemetry_decoder.go`: responsável por decodificar a cadeia de *bits* dos pacotes.
- `telemetry_processor.go`: transforma os dados do passo anterior em JSON para postagem.
- `package_handler.go`: publica o JSON na mensageria.

Assim que os dados chegam à mensageria, o *Kafka* os distribui no tópico `f1-telemetry`, que é onde tanto o produtor quanto os consumidores ficam conectados para se comunicarem. O dado é distribuído entre as duas camadas separadas, sendo a rápida aquela que levará os dados ao *paddock* para visualização em tempo real e a em lote, que salvará os dados em um armazenamento seguro para uso posterior.

A camada rápida conta com o `post_stream_data.py` para persistir os JSON vindos da mensageria no *bucket* adequado do *InfluxDB*, readequando os dados para o formato esperado. Como o *Grafana* se conecta nativamente e diretamente ao banco temporal, consumindo os dados por meio de consultas, nenhum código é necessário para essa tarefa.

Já a camada em lote é dividida em duas etapas: durante a corrida e o pós. Durante a corrida, o `post_data_to_dl.py` permanece ativo, armazenando os dados em memória até que ela fique cheia ou o tempo de permanência se esgote. Ao passar esse tempo, persiste os dados no *data lake* (representado na forma do `MinIO`).

[1] <https://github.com/joselflima/F1-Telemetry>

Depois da corrida, um *job* secundário pode ser executado para transformar os dados do *lake* em formatos relacionais para o *data warehouse*, representado na forma do arquivo `post_data_to_dw.py`.

4.2 Configuração do simulador

A fim de permitir o modo de telemetria do simulador *F1 23*, é necessário ajustar algumas configurações antes de iniciar a implementação.

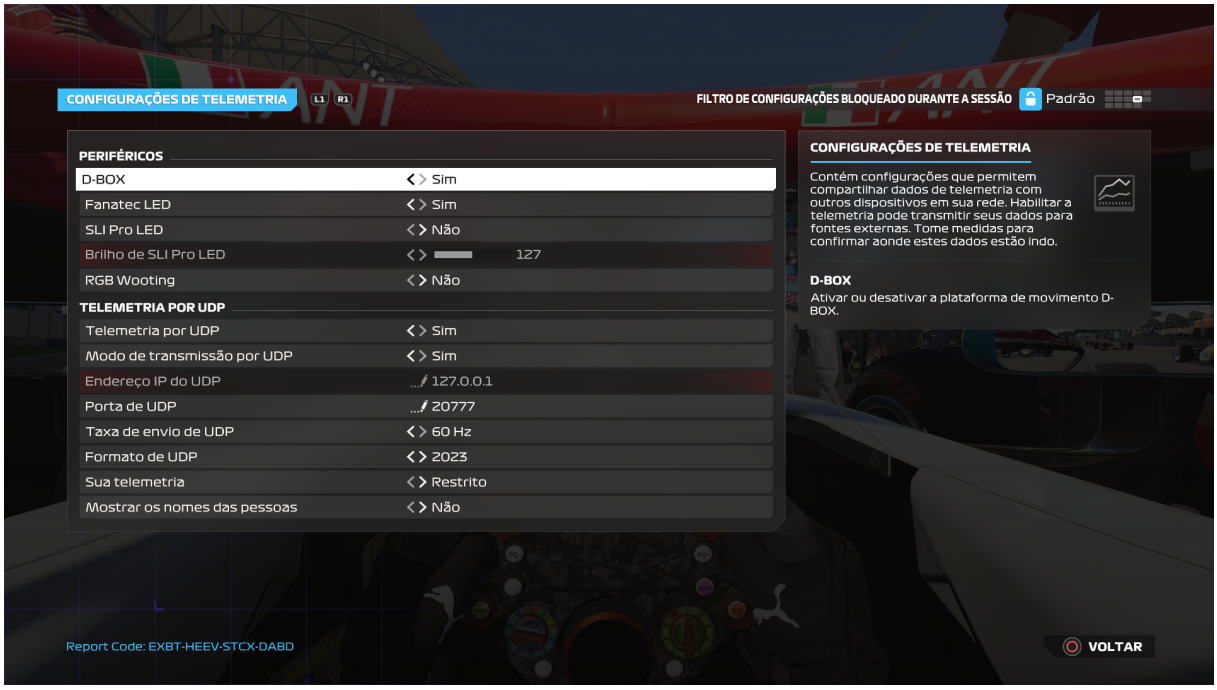

Figura 10 – Configuração de telemetria do simulador

Conforme apresentado na Figura 10, a ativação da telemetria é realizada por meio do menu de configurações do simulador, na seção dedicada à transmissão de dados. Nessa etapa, devem ser definidos os parâmetros de envio via protocolo UDP, conforme listado a seguir:

- **Telemetria por UDP**: Sim
- **Modo de transmissão por UDP**: Sim
- **Porta de UDP**: 20777 (outra porta pode ser escolhida aqui)
- **Taxa de envio de UDP**: 60Hz
- **Formato de UDP**: 2023

- **Sua telemetria**: Restrito
- **Mostrar os nomes das pessoas**: Não (opcional)

4.3 Estrutura do repositório no *Git*

A organização do repositório foi estruturada de forma modular, com o objetivo de separar claramente os diferentes papéis da solução, como recepção dos dados de telemetria, decodificação dos pacotes, processamento em fluxo, processamento em lote e utilitários compartilhados, além de, claro, refletir a arquitetura proposta na Figura 9. Essa divisão favorece a manutenção do sistema, a evolução independente de seus componentes e a maior legibilidade da base de código.

A estrutura atual do repositório contempla diretórios voltados tanto ao desenvolvimento da aplicação quanto ao suporte operacional e documental do projeto. Na Figura 11, detalham-se os principais diretórios e arquivos do projeto:

```
F1-TELEMETRY/
|__ docs/                  # documentação do projeto
|__ src/                   # código-fonte completo
|   |__ batch/             # camada em lote
|   |   |__ buffer/        # módulo de armazenamento no data lake
|   |   |__ etl/           # módulo de armazenamento no data warehouse
|   |__ preprocessor/      # pré-processamento
|   |__ stream/            # camada rápida
|   |__ utils/             # funções utilitárias
|__ tests/                 # testes de qualidade do projeto
|__ docker-compose.yml     # orquestração da solução
|__ requirements.txt       # bibliotecas para o Python
|__ go.mod                 # bibliotecas para o Golang
|__ README.md              # documento de entrada do projeto
```

Figura 11 – Estrutura do repositório de código do trabalho

Essa organização evidencia uma separação clara entre os componentes responsáveis pela ingestão dos dados, sua interpretação, seu processamento em tempo real e seu tratamento em lote, além de contemplar elementos de suporte, documentação e testes.

Como resultado, a estrutura adotada contribui para a escalabilidade, a legibilidade e a governança do projeto.

4.4 Configuração do contêiner

Dentro do arquivo `docker-compose.yml`, são definidas três seções principais: `services`, `networks` e `volumes`.

A seção `services` descreve os serviços necessários para a execução da aplicação, incluindo bancos de dados, sistemas de mensageria e demais componentes que compõem a arquitetura.

A seção `networks` é responsável pela configuração da rede interna do ambiente conteinerizado. Por meio dela, define-se uma rede virtual do tipo *bridge*, que permite a comunicação isolada e controlada entre os serviços, garantindo que os contêineres consigam se identificar e interagir entre si de forma segura e independente da rede externa.

Por fim, a seção `volumes` trata da persistência de dados. Nela, são definidos volumes que atuam como mecanismos de armazenamento desacoplados do ciclo de vida dos contêineres, assegurando que informações críticas — como dados de bancos e configurações — sejam mantidas mesmo após reinicializações ou recriações dos serviços.

4.5 Implementação do módulo de pré-processador

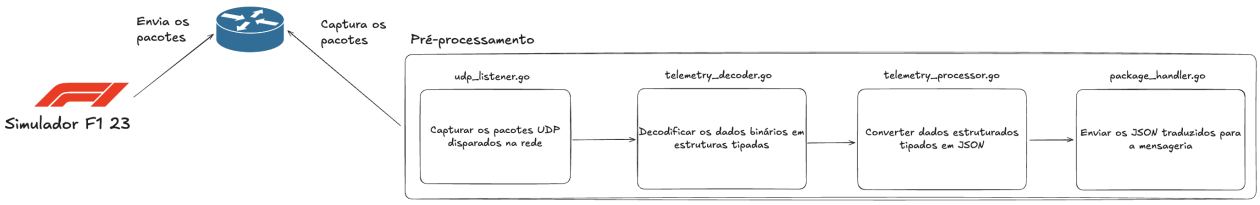

Figura 12 – Figura da arquitetura do módulo de pré-processamento

Evidenciado pelo diagrama apresentado na Figura 12, a jornada dos dados inicia-se no `udp_listener.go`, que contém a interface de recepção de baixo nível. Este componente configura um *socket* de escuta no protocolo UDP, operando por padrão na porta 20777. Sua responsabilidade técnica é a captura de datagramas binários brutos, funcionando como um *gateway* de entrada que desacopla a recepção física dos pacotes do seu processamento lógico e semântico.

O fluxo de tradução e produção dos dados de telemetria é estruturado em três etapas sequenciais: decodificação, processamento e publicação. Essa separação promove modularidade, clareza de responsabilidades e maior facilidade de manutenção do sistema.

A etapa inicial ocorre no `telemetry_decoder.go`, responsável por disponibilizar as funções de decodificação dos dados binários brutos recebidos. Este componente realiza a desserialização dos *bytes*, respeitando a ordenação *Little Endian*, convertendo-os em estruturas tipadas da aplicação, especificamente na *struct* `PacketCarTelemetryData`. Esse processo permite a interpretação correta de variáveis como temperatura, pressão e parâmetros do motor.

Na sequência, o `telemetry_processor.go` contém o código que executa o processamento dos dados estruturados. Nesta etapa, o sistema identifica e isola as informações relevantes com base no índice `PlayerCarIndex`, garantindo que apenas os dados do piloto monitorado sejam considerados. O resultado desse processamento é consolidado na estrutura `TelemetryPayload`, que é enriquecida com anotações JSON, tornando-a adequada para transporte e consumo por sistemas externos.

Por fim, o `packet_handler.go` hospeda as funções responsáveis pela orquestração final e envio dos dados processados. A partir da análise do cabeçalho unificado (`PacketHeader`), especialmente do campo `PacketID`, o sistema filtra os pacotes de interesse — neste trabalho, especificamente os de ID 6, correspondentes à telemetria do veículo. Após essa validação, o *handler* encaminha os dados já processados para a camada de mensageria.

Os componentes citados acima são todos orquestrados por um único arquivo denominado `main.go`, garantindo que a sequência lógica proposta seja cumprida e o código possa operar de maneira desacoplada, promovendo facilidade na manutenção e na criação de testes unitários para a qualidade da solução.

Dessa forma, a *pipeline* implementada transforma dados binários brutos em eventos estruturados e consumíveis, assegurando robustez, eficiência e desacoplamento entre as etapas do processamento.

4.6 Configuração do *Kafka*

No contexto deste trabalho, o *Apache Kafka* opera na porta padrão `9092` e utiliza um único tópico denominado `f1-telemetry`, responsável por centralizar o fluxo de eventos de telemetria. Tanto produtores quanto consumidores interagem com este tópico para a publicação e leitura das mensagens.

A criação do tópico pode ser realizada via CLI do Kafka, conforme o exemplo na Figura 13:

Figura 13 – Exemplo de comando para criação de tópicos no *Kafka*

No que se refere ao consumo, o sistema utiliza grupos distintos para atender finalidades diferentes no *pipeline* de processamento. O grupo `influxdb-writer-group` é responsável pelo consumo dos eventos destinados à *speed layer*, enquanto o grupo `minio_saver_group_v3` realiza o consumo voltado à *batch layer*. Dessa forma, ambos os serviços podem ler as mesmas mensagens publicadas no tópico `f1-telemetry`, porém de maneira independente, mantendo seus próprios *offsets* e seu próprio estado de processamento.

Os grupos de consumidores (*consumer groups*) constituem um mecanismo essencial do *Kafka* para organizar o consumo de mensagens. Quando consumidores pertencem ao mesmo grupo, as partições de um tópico são distribuídas entre eles, de modo que cada mensagem seja processada uma única vez dentro daquele grupo. Entretanto, consumidores pertencentes a grupos diferentes podem consumir independentemente os mesmos dados de um mesmo tópico. Essa característica é particularmente importante neste trabalho, pois permite que a mesma telemetria alimente simultaneamente a camada de processamento em tempo real e a camada de persistência histórica, sem duplicação de publicação por parte do produtor.

A visualização dos grupos de consumidores pode ser realizada por meio do comando na Figura 14:

Figura 14 – Exemplo de comando para listagem de grupos no *Kafka*

De forma complementar, a inspeção do tópico configurado pode ser realizada com o comando na Figura 15:

```
kafka-topics.sh --describe \
  --topic f1-telemetry \
  --bootstrap-server localhost:9092
```

Figura 15 – Exemplo de comando para descrição de tópicos no *Kafka*

Assim, a utilização de grupos de consumidores permite o desacoplamento entre os serviços produtores e consumidores, além de viabilizar que diferentes camadas da arquitetura processem o mesmo fluxo de eventos de maneira independente, escalável e resiliente.

4.7 Implementação da camada rápida

Como definido anteriormente, a camada rápida será responsável por mover os dados pré-processados ao *dashboard* que poderá ser acompanhado dentro do *paddock*. Na Figura 16, é ilustrado como esta camada funcionará: transformar os dados JSON em pontos do *InfluxDB*, persistí-los no banco e exibir em um painel dentro do *Grafana*.

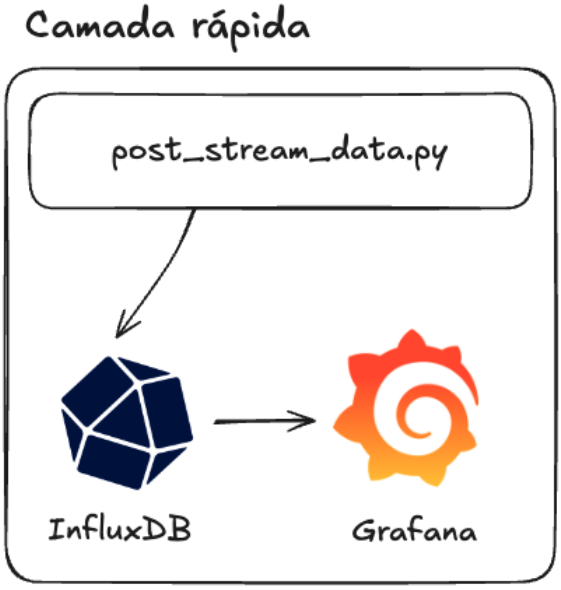

Figura 16 – Figura da arquitetura da camada rápida

4.7.1 Implementação do consumo e persistência da camada rápida

A etapa de consumo da mensageria e persistência na camada rápida será consolidada no arquivo `post_stream_data.py`, responsável por consumir os eventos do tópico `f1-telemetry` por meio do grupo `influxdb-writer-group` e registrar os dados no *bucket* correspondente do *InfluxDB*. Para a comunicação com o banco temporal, recomenda-se a utilização da biblioteca oficial `influxdb-client` para Python, mantida pela InfluxData, por oferecer suporte nativo ao modelo de autenticação baseado em *token*, organização e *bucket*, além de recursos adequados para a escrita de pontos temporais de forma estruturada.

4.7.2 Configuração do *InfluxDB*

Ao entrar no menu principal do banco de dados temporal, pode-se consultar todos os *buckets* presentes naquele banco. No caso deste trabalho, terá apenas um *bucket* chamado `f1-bucket`, responsável por armazenar todos os dados a serem exibidos na camada de visualização para o *paddock*.

A Figura 17 exibe como o *bucket* é carregado no menu principal do *InfluxDB*, com opções para visualizar o conteúdo do *bucket*, adicionar dados e configurações.

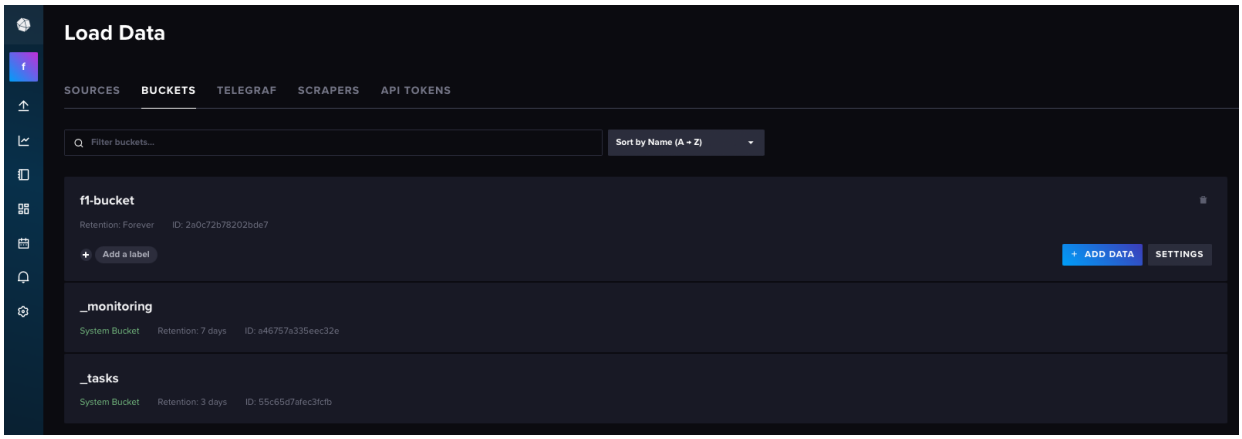

Figura 17 – Figura do menu de *buckets* do *InfluxDB*

Dentro do `f1-bucket`, serão salvos os pontos de dados; aqui, haverá uma métrica chamada `car_telemetry`, a qual contém todos os campos que vêm do simulador.

O banco retorna suas consultas em um formato específico, como mostrado na Figura 18. No resultado, é possível visualizar o nome da métrica, do campo, do valor e das datas e horas de início e fim da inserção.

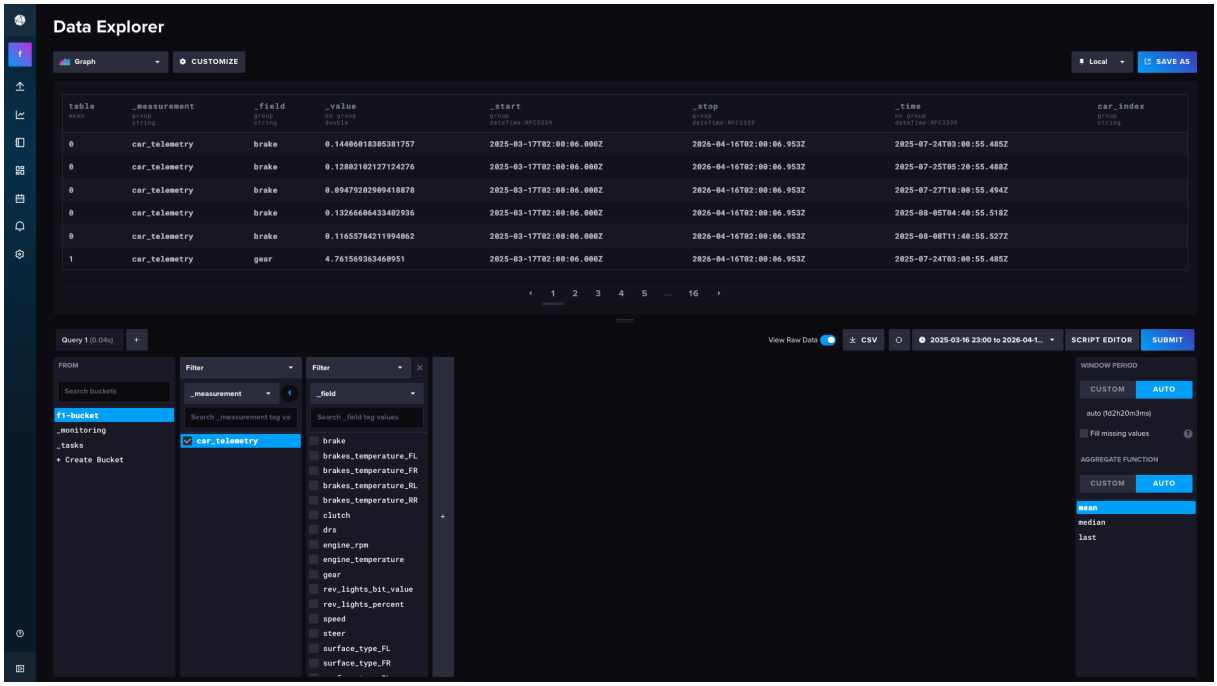

Figura 18 – Figura dos dados armazenados na tabela temporal de telemetria

O *InfluxDB* tem dois tipos de linguagens de consulta disponíveis: *InfluxQL* (similar ao SQL padrão) e *Flux*. No caso deste trabalho, para consultas tanto no banco quanto na camada analítica, usa-se o *Flux* para isso; uma linguagem de consulta em formato de *script* desenhada para transformações de dados complexas e amplamente utilizada nas versões 2.x do banco (a família de versões utilizada neste trabalho). Uma das suas principais características é o uso do operador *pipe* (|>) para operações encadeadas.

Na Figura 19, é ilustrado como a consulta para testes dos dados da telemetria dentro do banco temporal foi construída com o *Flux*. Ela seleciona os dados do *bucket* em um determinado período de tempo filtrados pela métrica `car_telemetry` e resume os valores dos campos em média.

Figura 19 – Figura dos dados armazenados na tabela temporal de telemetria

4.7.3 Configuração do *Grafana*

Por fim, na camada analítica, o *Grafana* entra para a construção do *dashboard* por sua agilidade de construção e facilidade de configuração e integração com diversos tipos de bancos de dados.

Na Figura 20, é mostrado como o *Grafana* exibe todos os painéis criados até o momento; como neste trabalho haverá apenas o de telemetria dos carros, logo no início, é possível encontrar o painel de telemetria.

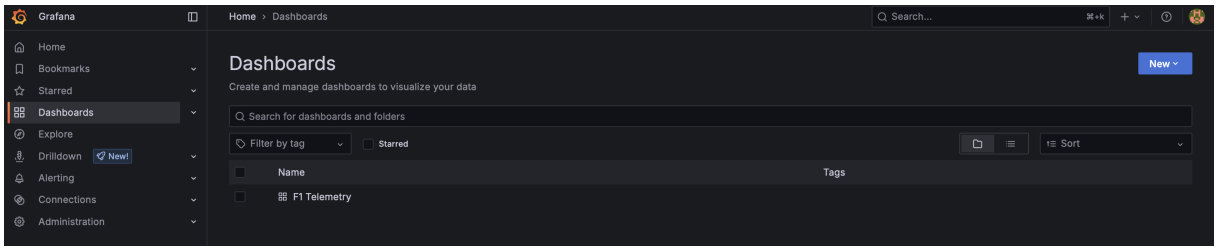

Figura 20 – Figura ilustrativa do menu de visualizações do *Grafana*

Ao entrar no painel, conforme mostra a Figura 21, os gráficos de pressão dos pneus, DRS, velocidade atual, aceleração, frenagem e marcha atual são exibidos na tela em seus respectivos espaços. A vantagem do *Grafana* nesta etapa é a facilidade de posicionar os gráficos no painel e a maneira como serão apresentados.

Na parte superior direita, é definido o tempo de atualização geral dos gráficos, configurado para consultar os dados do *InfluxDB* a cada 1 segundo; caso seja necessário, mais para a direita, é possível recarregar as visões manualmente.

Figura 21 – Figura ilustrativa do *dashboard* da telemetria

A configuração de cada gráfico é bem parecida. Conforme ilustrado pela Figura 22, é exibida uma prévia, no meio, de como ficará o gráfico final depois que todas as configurações forem aplicadas. Abaixo, é possível selecionar a fonte dos dados e a consulta a ser executada para alimentar a visão.

No lado direito, é possível selecionar o tipo de gráfico (barras, colunas, velocímetro, linhas, dispersão etc.), título, descrição, tipo de cálculo, orientação do gráfico, entre várias outras opções que podem ser aplicadas conforme necessário.

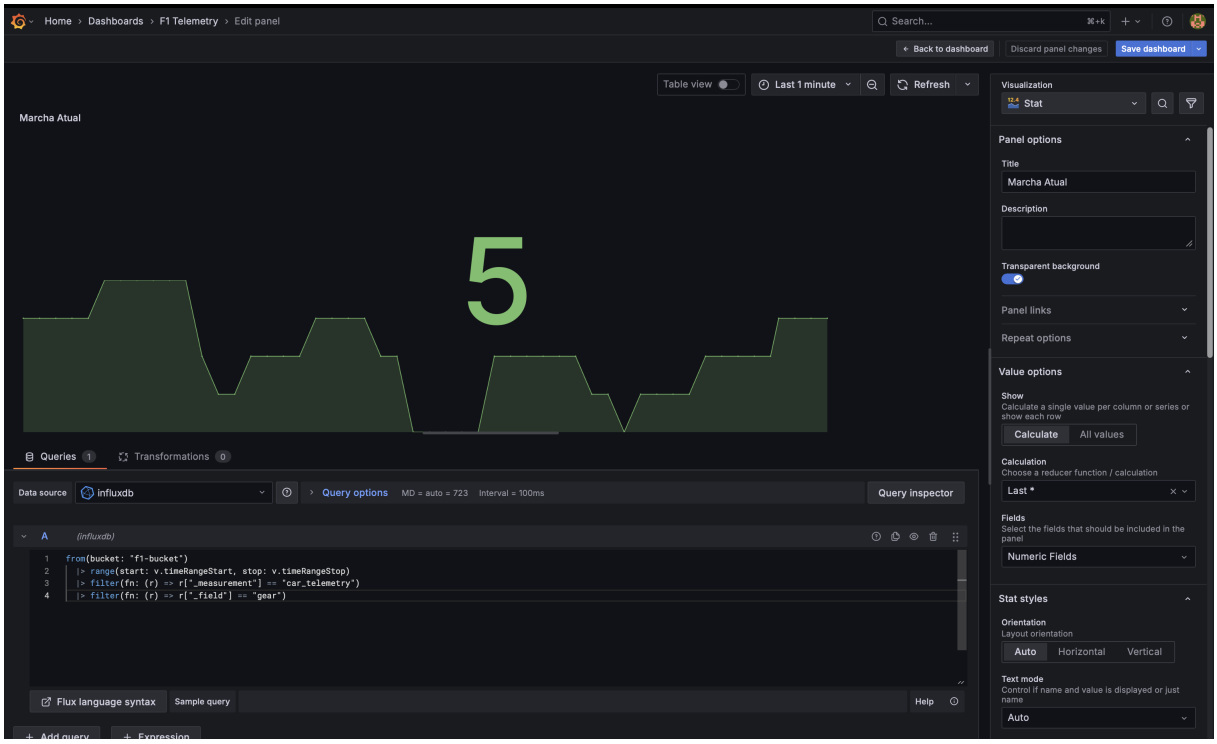

Figura 22 – Figura ilustrativa do menu de configuração dos painéis

4.8 Implementação da camada em lote

4.8.1 Implementação do consumidor da camada em lote

Para atender aos requisitos de armazenamento de longo prazo e permitir análises retrospectivas complexas, o sistema integra um consumidor dedicado à camada de processamento em lote (*batch layer*). Implementado em *Python*, este componente é responsável por extrair mensagens do barramento de eventos *Apache Kafka* e persistir em um repositório de objetos compatível com a API S3, utilizando para esse fim o MinIO.

A arquitetura deste consumidor prioriza a resiliência e a eficiência de escrita. Através das funções `connect_kafka` e `connect_minio`, o sistema implementa uma lógica de *retry* (tentativas de reconexão) que assegura a continuidade da operação, mesmo em cenários de instabilidade na infraestrutura de rede ou demora na inicialização dos serviços. O consumidor é configurado dentro de um grupo de consumo (*Consumer Group*), o que garante que as mensagens sejam processadas de forma coordenada e que o estado do consumo (*offsets*) seja preservado.

O diferencial técnico desta implementação reside na estratégia de *buffering* e compactação de dados. Diferente da *speed layer*, que processa eventos individualmente para minimizar a latência, a *batch layer* agrupa mensagens em memória para otimizar as operações de entrada e saída (I/O) no armazenamento de objetos. A persistência dos dados

no *MinIO* é disparada por dois critérios configuráveis:

- **Intervalo temporal (`FLUSH_INTERVAL`):** Garante que os dados sejam persistidos periodicamente, evitando que informações fiquem retidas em memória por tempo excessivo.
- **Volume de mensagens (`FLUSH_MAX_MESSAGES`):** Aciona a escrita assim que o *buffer* atinge um limite crítico de mensagens, otimizando o tamanho dos arquivos gerados.

O processo de descarte do *buffer* (*flush*) é gerenciado pela função `flush_buffer`, que converte a lista de objetos JSON em um fluxo de bytes (`BytesIO`) e realiza o *upload* para o *bucket* de telemetria. Cada arquivo é nomeado dinamicamente com base no *timestamp* de criação (ex: `telemetry_YYYYMMDD_HHMMSS.json`), facilitando a organização cronológica e a posterior ingestão por ferramentas de *Big Data* ou sistemas de *Data Discovery*. Assim, a *batch layer* consolida o histórico bruto da telemetria, servindo como a fonte da verdade para auditorias e treinamentos de modelos de análise.

4.8.2 Configuração do *data lake* com *MinIO*

Ao entrar no *MinIO*, a primeira tela, assim como está na Figura 23, é justamente os *buckets* criados, nos quais estamos trabalhando - *telemetry* e *telemetry-processed*.

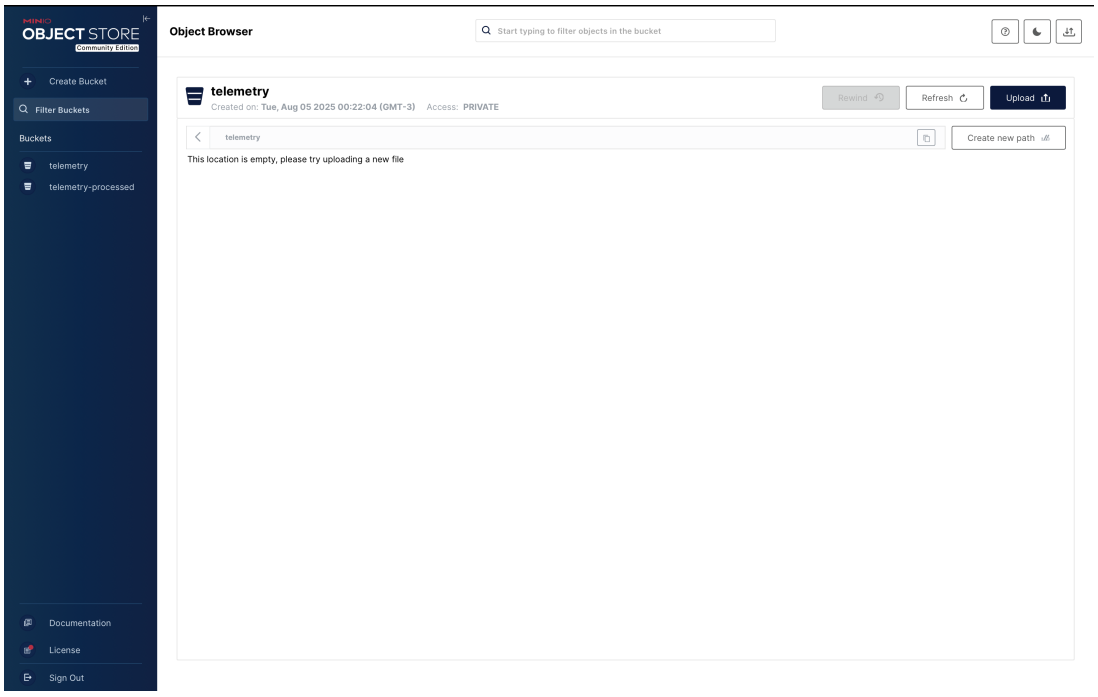

Figura 23 – Figura da tela inicial do MinIO

O *bucket telemetry* é para onde vão os dados crus recebidos da mensageria; serve como um depósito de escrita rápida e armazena em documentos os próximos JSONs a serem processados pelo módulo de ETL pós-corrida.

Já o *bucket telemetry-processed* é onde ficam armazenados os JSONs já processados e cujos dados foram devidamente guardados no *PostgreSQL*. Esse segundo repositório existe como uma reserva segura para os dados em caso de necessidade de reprocessamento de algum dado específico.

Na Figura 24, é possível visualizar como ficam os arquivos salvos nesse local assim que são processados - também é possível encontrar o tempo em que foram modificados pela última vez e seu tamanho no disco de armazenamento.

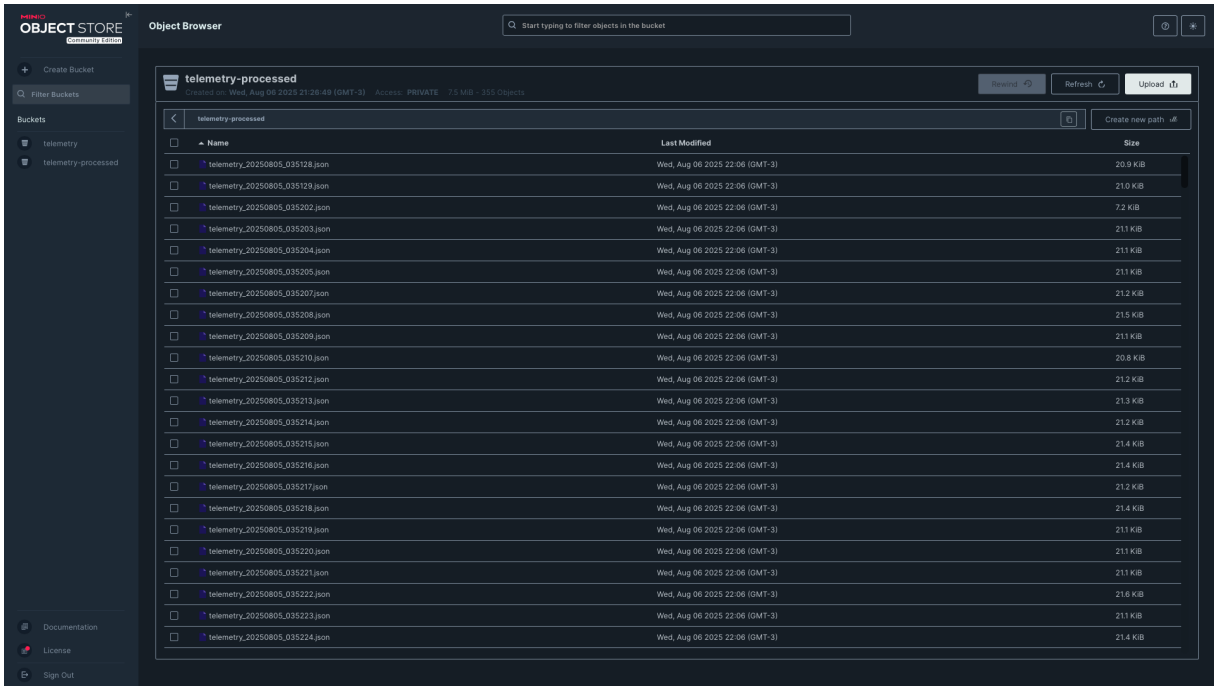

Figura 24 – Figura da tela do *bucket telemetry-processed*

4.8.3 Implementação do módulo de ETL pós-corrida

Para consolidar a transição dos dados da camada de armazenamento bruto (*Data Lake*) para um ambiente de análise relacional após o término da corrida, o sistema utiliza o módulo de ETL pós-corrida presente no arquivo `save_data_to_dw.py`. Este componente executa o processo de extração, transformação e carregamento, sendo responsável por converter os arquivos JSON acumulados no *MinIO* em registros estruturados dentro de um banco de dados *PostgreSQL*, que atua como o *data warehouse* da solução.

O fluxo de processamento operacionalizado por este módulo é dividido em três etapas fundamentais:

1. **Extração e leitura:** O *script* realiza a varredura do *bucket* de origem (`telemetry`), identificando todos os objetos persistidos durante a sessão de telemetria. Cada arquivo é baixado para a memória e desserializado de seu formato original em JSON para estruturas de dicionários *Python*, permitindo o acesso programático aos campos capturados.

2. **Carga estruturada:** Utilizando a biblioteca `psycopg2`, o componente estabelece uma conexão com o *PostgreSQL* e executa a inserção dos dados por meio de instruções SQL parametrizadas. Esta etapa é crucial para a integridade referencial, pois mapeia as métricas de telemetria (como *speed*, *engine_temperature* e *tyres_pressure*) para colunas tipadas, permitindo a execução de consultas analíticas complexas e agregações que seriam custosas em sistemas de arquivos brutos.

3. **Gestão do ciclo de vida dos dados:** Após a confirmação da transação no banco de dados (*commit*), o *script* gerencia a movimentação dos arquivos processados. Utilizando operações de cópia e remoção atômica, os objetos são transferidos para o *bucket* `telemetry-processed`. Este procedimento garante a idempotência do processo, evitando a duplicidade de registros em execuções futuras e mantendo o *data lake* organizado para novas sessões de captura.

Dessa forma, o módulo pós-sessão transforma o repositório de eventos brutos em uma base de conhecimento estruturada, possibilitando que ferramentas de *business intelligence* (BI) e algoritmos de análise estatística consultem o histórico completo das sessões com alta performance e confiabilidade.

Essa abordagem também permite que as equipes executem essa etapa da *pipeline* de forma assíncrona e a qualquer momento.

4.8.4 Configuração do *data warehouse PostgreSQL*

Para configurar o banco de dados, começamos criando a tabela que receberá os dados vindos do *data lake*. Para fins didáticos, este trabalho possui apenas uma tabela principal chamada *telemetry*, gerada a partir do código da Figura 25:

```sql
CREATE TABLE telemetry (
    id SERIAL PRIMARY KEY,
    car_index INT,
    speed INT,
    throttle FLOAT,
    steer FLOAT,
    brake FLOAT,
    clutch FLOAT,
    gear INT,
    engine_rpm INT,
    drs INT,
    rev_lights_percent INT,
    rev_lights_bit_value INT,
    engine_temperature INT,
    brakes_temperature INT[],
    tyres_surface_temperature INT[],
    tyres_inner_temperature INT[],
    tyres_pressure FLOAT[],
    surface_type INT[]
);
```

Figura 25 – Figura do código SQL para criação da tabela de telemetria

Após a inserção dos primeiros registros usando o módulo de ETL pós-corrida, a tabela dentro do banco fica da forma como ilustra a Figura 26: dados organizados e prontos para análise posterior.

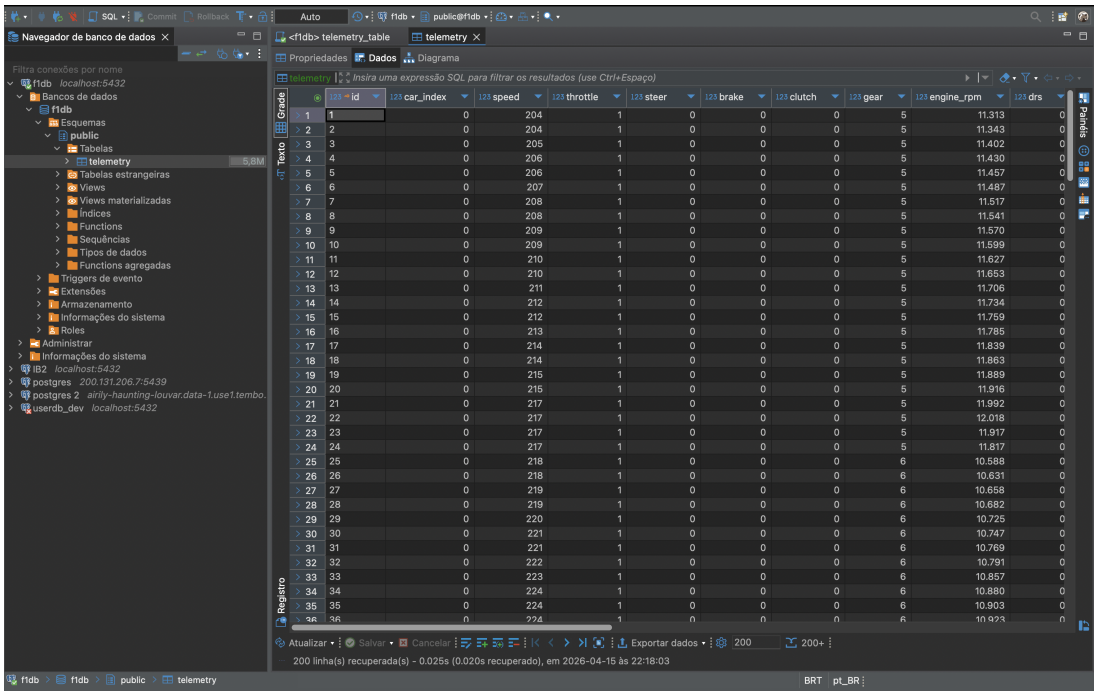

Figura 26 – Figura da tabela de telemetria no banco de dados relacional

4.9 Desafios encontrados e soluções

4.9.1 Erro na coleta dos dados de frenagem do carro

Durante a implementação do módulo de pré-processamento, foi identificada uma inconsistência na captura dos dados de frenagem do veículo. No contexto do simulador *F1 23*, a aceleração e a frenagem podem ser controladas de forma analógica por meio dos gatilhos do controle, de modo que a intensidade da pressão aplicada pelo jogador é convertida em valores proporcionais de entrada. Em condições adequadas de configuração, esse comportamento permite que a telemetria registre níveis graduais de aceleração e frenagem, e não apenas estados discretos.

Entretanto, durante os testes iniciais da *pipeline*, observou-se que o valor capturado para a frenagem estava sendo recebido de forma binária, assumindo apenas os valores 0 ou 1. Esse comportamento inviabilizava a representação fiel da progressão do acionamento do acelerador, comprometendo a qualidade do dado coletado e limitando análises posteriores que dependem de maior precisão na leitura da entrada do piloto. A Figura 27 ilustra, pelo gráfico de frenagem, como este dado estava chegando ao *dashboard*, o que ajudou a diagnosticar o problema.

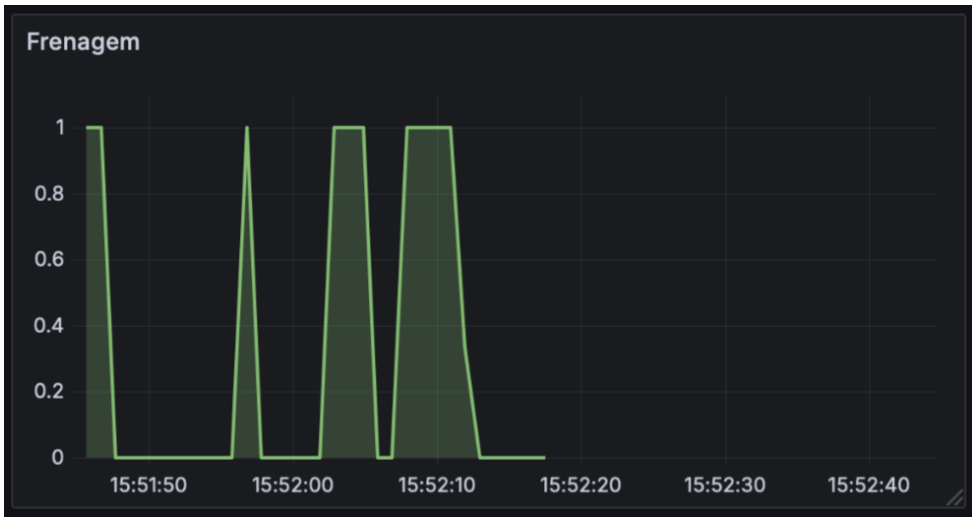

Figura 27 – Demonstração do gráfico de frenagem com erro nos dados

Após a investigação do problema, verificou-se que a limitação não estava na etapa de consumo ou tradução dos pacotes de telemetria, mas sim na forma como o simulador estava configurado para interpretar a entrada do controle. Dessa forma, foi realizado um ajuste na configuração de calibração do controle dentro do próprio *F1 23* para padronizar a linearidade e a saturação do eixo de direção, acelerador e freio.

Com a correção aplicada, os dados de frenagem passaram a refletir de forma mais precisa a variação da pressão exercida no gatilho do controle, possibilitando a coleta de

valores intermediários e, consequentemente, uma representação mais fiel da dinâmica de condução do veículo. A Figura 28 apresenta a configuração utilizada no simulador para viabilizar essa captura com maior precisão.

Figura 28 – Ajustes de calibração do controle do simulador

4.9.2 Substituição do *Python* pelo *Golang* no módulo de pré-processamento

Inicialmente, previa-se a utilização de *Python* na etapa de pré-processamento dos dados de telemetria. Entretanto, no decorrer da implementação, constatou-se que essa abordagem não era a mais apropriada para a interpretação dos pacotes binários enviados pelo simulador via UDP.

A telemetria transmitida pelo *F1 23* é disponibilizada em formato binário, exigindo a leitura e conversão dos pacotes com base na documentação técnica fornecida pela *Electronic Arts*, que define a estrutura, a ordem e os tipos de cada campo presente nas mensagens. Nesse contexto, a etapa de pré-processamento exigia um mapeamento preciso entre o conteúdo bruto recebido da rede e sua representação estruturada na aplicação.

Diante das dificuldades encontradas na implementação dessa tradução em *Python*, optou-se pela adoção do *Golang* nessa camada da solução. A linguagem mostrou-se mais adequada por oferecer recursos mais diretos para a manipulação de estruturas binárias e por favorecer a implementação de rotinas de rede com bom desempenho.

A substituição permitiu a correta interpretação dos pacotes de telemetria e viabilizou a geração de eventos estruturados, tornando possível o encaminhamento consistente dos dados para as etapas subsequentes da pipeline.

4.9.3 Atualização forçada do *dashboard* para a cada um segundo

Por padrão, o *Grafana* atualiza todos os gráficos a cada 5 segundos como uma medida protetiva ao *hardware* e à fonte de dados conectada, evitando múltiplas requisições simultâneas e prevenindo gargalos. Porém, para reproduzir com maior fidelidade o cenário real de corrida, foi necessário realizar um ajuste nos parâmetros internos do *Grafana* para que, por padrão, o tempo mínimo para o recarregamento dos dados fosse de um segundo.

No `docker-compose.yml`, a variável `GF_DASHBOARDS_MIN_REFRESH_INTERVAL`, com valor de um segundo, foi adicionada ao ambiente do *Grafana*, conforme ilustrado pela Figura 29. Isso possibilitou que o *Grafana* atualizasse todos os painéis a cada um segundo automaticamente por padrão, resolvendo o problema e simulando, com maior precisão, a chegada de dados em tempo real na camada rápida.

```
environment:
  - GF_SECURITY_ADMIN_USER=admin
  - GF_SECURITY_ADMIN_PASSWORD=grafana
  - GF_DASHBOARDS_MIN_REFRESH_INTERVAL=1s # Ajuste de atualização do Grafana
```

Figura 29 – Ajustes de intervalo mínimo de atualização do *Grafana*

4.10 Considerações finais do capítulo

A construção deste trabalho demandou um período significativo de pesquisa e planejamento, desde a definição da fonte de dados até a seleção das tecnologias mais adequadas para cada etapa da solução.

Ao analisar o sistema desenvolvido de forma integrada, observando a comunicação entre os componentes e sua atuação coordenada em função do objetivo principal, evidencia-se a complexidade envolvida na implementação de uma arquitetura orientada a eventos com processamento em tempo real. A execução de uma corrida simultaneamente à visualização dos dados no *dashboard* permitiu validar, na prática, o funcionamento da solução proposta.

Além do desafio de desenvolver uma aplicação com múltiplas camadas e tecnologias distintas, destacaram-se aspectos relevantes relacionados à tomada de decisões técnicas, à adaptação de ferramentas e à resolução de problemas não previstos inicialmente. Nesse contexto, o trabalho contribuiu significativamente para o aprofundamento do conhecimento em engenharia de dados, especialmente no que se refere à construção de fluxos distribuídos e ao processamento de dados em tempo real.

Outro ponto a destacar é o uso do UDP em vez do TCP (protocolo de controle de transmissão) no desenvolvimento da aplicação. O UDP é amplamente utilizado em cenários que exigem baixa latência de transmissão de dados, o que é extremamente vantajoso e necessário na Fórmula 1, e é algo que o TCP não consegue garantir. Porém, o uso do UDP aqui pode acarretar perdas de pacotes de dados, visto que não há uma garantia de entrega dos pacotes nem uma conexão estabelecida entre o simulador e a aplicação. Em resumo, ao usar o UDP, aceita-se a perda de alguns pacotes de dados, trazendo ruídos na visualização, para ter acesso a eles no menor espaço de tempo possível.

5 Conclusão e trabalhos futuros

5.1 Revisão dos objetivos

O objetivo central deste trabalho consistiu em construir uma *pipeline* de dados capaz de processar a telemetria dos carros de Fórmula 1 de forma a atender, simultaneamente, duas necessidades distintas e complementares: a disponibilização dos dados em tempo real para acompanhamento operacional no *paddock* e o armazenamento dessas informações em uma camada de dados para uso analítico posterior.

À luz dos resultados apresentados nos capítulos anteriores, entende-se que esse objetivo foi alcançado de maneira satisfatória. A solução implementada mostrou-se capaz de captar os dados de telemetria gerados pelo simulador, encaminhá-los por meio de uma arquitetura orientada a eventos e distribuí-los para múltiplas rotas de consumo sem comprometer a proposta central do trabalho. Com isso, demonstrou-se, na prática, a viabilidade de uma abordagem em que o mesmo fluxo de dados atende tanto a demandas de baixa latência quanto a necessidades de persistência para análises futuras.

No que se refere à primeira finalidade da *pipeline*, relacionada ao consumo em tempo real, os resultados evidenciam que os dados puderam ser recebidos, tratados e disponibilizados para visualização quase imediata, permitindo o acompanhamento dinâmico de métricas relevantes do carro durante a execução da corrida. Essa etapa valida a capacidade da arquitetura de sustentar o uso operacional da telemetria em cenários nos quais a informação precisa ser consumida com rapidez e de forma contínua.

Em paralelo, a segunda finalidade também foi contemplada com êxito, uma vez que os mesmos eventos processados puderam ser persistidos em uma camada de armazenamento voltada para o uso posterior. Essa trilha amplia o valor da solução proposta, pois transforma o dado telemétrico em ativo reaproveitável para estudos históricos, análises exploratórias, aplicações de aprendizado de máquina e outras abordagens analíticas que extrapolam o contexto imediato da corrida.

Dessa forma, conclui-se que a principal contribuição deste trabalho esteve na construção de uma *pipeline* capaz de conciliar, em uma mesma arquitetura, o processamento em tempo real e a persistência para uso futuro. Tal resultado reforça a adequação da abordagem adotada e evidencia o potencial da solução como base para evoluções posteriores no campo da engenharia de dados aplicada à Fórmula 1.

5.2 Aplicações futuras

Embora a solução desenvolvida tenha atendido ao objetivo proposto de disponibilizar os dados de telemetria em tempo real e armazená-los para uso posterior, o trabalho também evidenciou possibilidades de evolução que podem ampliar sua robustez, organização e potencial analítico. Nesse sentido, apresentam-se a seguir algumas frentes que podem orientar a continuidade da pesquisa e o amadurecimento da arquitetura implementada.

5.2.1 Melhoria da segurança da *pipeline*

Como continuidade deste trabalho, uma evolução relevante consiste na incorporação de mecanismos de segurança à *pipeline* de dados. Embora a solução proposta tenha sido desenvolvida com foco na validação funcional do fluxo de telemetria, sua aplicação em contextos mais próximos de ambientes reais demandaria cuidados adicionais relacionados à autenticação entre serviços, proteção de credenciais, controle de acesso e eventual criptografia da comunicação entre os componentes, garantindo que os dados do carro do piloto de uma equipe não podem ser acessados por terceiros, prejudicando a competitividade do esporte. O avanço nessa direção contribuiria para elevar a confiabilidade da arquitetura e torná-la mais aderente a cenários de produção.

5.2.2 Aplicação da arquitetura medalhão no contexto da Fórmula 1

Outra possibilidade de evolução está na reorganização da camada em lote, segundo os princípios da arquitetura medalhão. Essa abordagem permitiria estruturar os dados em diferentes níveis de tratamento, separando registros brutos, dados refinados e informações prontas para consumo analítico. No contexto da telemetria da Fórmula 1, tal organização poderia facilitar a rastreabilidade das transformações, melhorar a qualidade dos dados e simplificar o reaproveitamento das informações em diferentes finalidades, como análise histórica, construção de indicadores e treinamento de modelos preditivos.

5.2.3 Implementação da *star schema* no *data warehouse*

Também se destaca, como trabalho futuro, a evolução da modelagem do *data warehouse* para uma abordagem dimensional baseada em *star schema*. Embora o armazenamento relacional implementado já permita a persistência estruturada dos dados processados, a adoção de uma modelagem em fatos e dimensões tenderia a tornar as consultas analíticas mais claras, eficientes e aderentes ao consumo por ferramentas de *business intelligence*. Essa mudança ampliaria o potencial da solução para análises multidimensionais de desempenho, comparação entre voltas, estudo de comportamento do piloto e consolidação de métricas históricas.

5.2.4 Avaliação de formatos de serialização na *pipeline* de eventos

Durante o desenvolvimento deste trabalho, optou-se pela serialização dos dados de telemetria em formato JSON na etapa de pré-processamento, antes da publicação no Apache Kafka. Essa decisão foi motivada pela necessidade de interoperabilidade entre os módulos escritos em Go e Python, pela facilidade de depuração durante o desenvolvimento e pela adequação ao *SLA* de latência de cinco segundos estabelecido para a *pipeline*. Entretanto, o formato JSON introduz uma expansão considerável no tamanho do *payload* em relação à representação binária original emitida pelo simulador, o que representa um compromisso entre legibilidade e eficiência de transporte.

Como trabalho futuro, propõe-se a condução de um estudo comparativo entre o JSON e formatos de serialização binária com suporte a esquema, como o *Protocol Buffers* (Protobuf) e o *Apache Avro*. A investigação deve mensurar, sob condições controladas, a latência de ponta a ponta entre a recepção do pacote UDP e o consumo nos módulos da *pipeline*, o tamanho médio dos *payloads* publicados no tópico Kafka e o *throughput* máximo sustentável para cada formato. Para estressar os limites da solução, recomenda-se simular frequências de ingestão progressivamente maiores, como 60 Hz, 120 Hz e 240 Hz, identificando o ponto a partir do qual o JSON passa a violar o *SLA* enquanto os formatos binários ainda o sustentam. Esse experimento forneceria evidências empíricas para orientar decisões arquiteturais em cenários de maior exigência de desempenho, além de contribuir para a consolidação de um protocolo de avaliação replicável em trabalhos subsequentes.

5.2.5 Expansão da *pipeline* para múltiplos tipos de pacotes de telemetria

A solução implementada neste trabalho concentrou-se na coleta e no processamento dos pacotes de telemetria do carro, identificados pelo *PacketID* 6 na especificação UDP do simulador *F1 23*. Essa delimitação foi intencional e justificada pelo caráter experimental do trabalho, cujo objetivo central era demonstrar a viabilidade da arquitetura proposta para o processamento de dados em tempo real. Com a validação funcional da *pipeline* estabelecida, abre-se espaço para uma evolução natural da solução em direção a uma cobertura mais abrangente do ecossistema de dados disponibilizado pelo simulador.

Como trabalho futuro, propõe-se a expansão do módulo de pré-processamento para contemplar os demais tipos de pacotes transmitidos via UDP pelo *F1 23*, entre os quais se destacam os dados de sessão, contendo informações sobre o circuito, condições climáticas e fase da corrida; os dados do piloto, com métricas de comportamento; os dados de posição dos carros na pista; e os dados de danos e estado dos componentes do veículo. Cada novo tipo de pacote demandaria a definição de contratos de dados correspondentes no módulo de tradução em *Golang*, a criação de tópicos dedicados no *Apache Kafka* e a adequação dos esquemas de armazenamento, tanto no banco de séries temporais quanto no *data*

warehouse relacional. Essa expansão não apenas aumentaria a riqueza analítica da solução, como também permitiria a construção de análises cruzadas entre diferentes dimensões da corrida, como a correlação entre as condições da pista e o desempenho do piloto, ou entre a estratégia de *pit stop* e a degradação de componentes. A arquitetura orientada a eventos adotada neste trabalho foi projetada para suportar esse tipo de crescimento de forma modular, tornando essa evolução tecnicamente viável sem a necessidade de reestruturação dos componentes já implementados.

Referências

ABDI, G. E.; EDWARD, E.; CANCINO, H. N.; ALAGBE, D.; ADENIYI, D. Event-driven etl architectures in cloud-native systems. 12 2025. Disponível em: <https://www.researchgate.net/publication/399084858_Event-Driven_ETL_Architectures_in_Cloud-Native_Systems>. Citado na página 23.

AL-DHIEF, F.; SABRI, N.; LATIFF, N. M. A.; MALIK, N. N. N. A.; GHANI, M. K. A.; MOHAMMED, M.; AL-HADDAD, R.; DAWOOD, Y.; GHANI, M.; OBAID, O. I. Performance comparison between tcp and udp protocols in different simulation scenarios. **International Journal of Engineering Technology**, v. 7, p. 172–176, 01 2018. Citado na página 21.

BANERJEE, A.; JINDAL, A.; SHANKAR, A.; SACHDEVA, V.; HEGDE, K. Motorsport data acquisition system and live telemetry using fpga based can controller. **Journal of Physics: Conference Series**, v. 2161, p. 012041, 01 2022. Citado na página 28.

CHAI, C.; WANG, J.; LUO, Y.; NIU, Z.; LI, G. Data Management for Machine Learning: A Survey . **IEEE Transactions on Knowledge & Data Engineering**, IEEE Computer Society, Los Alamitos, CA, USA, v. 35, n. 05, p. 4646–4667, maio 2023. ISSN 1558-2191. Disponível em: <https://doi.ieeecomputersociety.org/10.1109/TKDE.2022.3148237>. Citado na página 18.

FAYYAD, U.; PIATETSKY-SHAPIRO, G.; SMYTH, P. From data mining to knowledge discovery in databases. **AI Magazine**, v. 17, n. 3, p. 37, Mar. 1996. Disponível em: <https://ojs.aaai.org/aimagazine/index.php/aimagazine/article/view/1230>. Citado na página 19.

HADJICHRISTOFI, C.; DIOCHNOS, S.; ANDRESAKIS, K.; VESCOUKIS, V. Using time-series databases for energy data infrastructures. **Energies**, v. 17, p. 5478, 11 2024. Citado na página 21.

HAI, R.; GEISLER, S.; QUIX, C. Data lake: A survey of functions and architectures. **arXiv preprint arXiv:2106.09592**, 2021. Disponível em: <https://arxiv.org/abs/2106.09592>. Citado na página 23.

HASHEM, I. A. T.; YAQOOB, I.; ANUAR, N. B. M.; MOKHTAR, S.; GANI, A.; KHAN, S. U. The rise of "big data" on cloud computing: Review and open research issues. **Information Systems**, Elsevier, v. 47, p. 98–115, 2015. Citado na página 19.

HOU, F.; LI, Z.; ZHANG, S. Analysis and improvement paths of curriculum goal achievement in data visualization based on the obe concept. **International Journal of Education and Humanities**, v. 22, p. 40–43, 02 2026. Citado na página 22.

KHATTACH, O.; MOUSSAOUI, O.; HASSINE, M. End-to-end architecture for real-time iot analytics and predictive maintenance using stream processing and ml pipelines. **Sensors**, v. 25, p. 2945, 05 2025. Citado na página 29.

KIRAN, M.; MURPHY, P.; MONGA, I.; DUGAN, J.; BAVEJA, S. Lambda architecture for cost-effective batch and speed big data processing. In: . [S.l.: s.n.], 2015. p. 2785–2792. Citado na página 29.

LAMER, A.; SAINT-DIZIER, C.; PARIS, N.; CHAZARD, E. Data lake, data warehouse, datamart, and feature store: Their contributions to the complete data reuse pipeline. **JMIR Med Inform**, v. 12, p. e54590, Jul 2024. ISSN 2291-9694. Disponível em: <https://medinform.jmir.org/2024/1/e54590>. Citado na página 23.

LAZZARI, L.; FARIAS, K. **Uncovering the Hidden Potential of Event-Driven Architecture: A Research Agenda**. 2023. Disponível em: <https://arxiv.org/abs/2308.05270>. Citado 2 vezes nas páginas 23 e 24.

LIMA, G. de; JUNIOR, L.; CARVALHO, C. de; JUNIOR, P.; JUNIOR, L. Barbosa de A.; SILVA, F.; BRANDãO, D. Development and implementation of a communication network in a formula sae car. In: . [S.l.: s.n.], 2023. Citado na página 28.

LUZICH, M. **Data-Driven Dominance: How Real-Time Analytics Are Reshaping F1 Race Strategy**. 2025. <https://michaelluzich.com/data-driven-dominance-how-real-time-analytics-are-reshaping-f1-race-strategy/>. Citado na página 14.

MENDES, P. F. **Formula student racing championship: design and implementation of the management and graphical interface of the telemetry system**. Dissertação (Mestrado), nov. 2011. Citado na página 20.

MERCEDES. Feature: Data and electronics in f1, explained! 2022. Citado 4 vezes nas páginas 11, 12, 15 e 19.

MSAKAMALI. **F1 Data Analysis and Tactical Insights: Exploring Formula 1 Race Performance Strategies**. 2024. <https://www.theseus.fi/bitstream/handle/10024/856650/Msakamali_Baraka.pdf?sequence=6>. Citado 2 vezes nas páginas 11 e 14.

NERI, G.; MARSHALL, S.; CHAN, H. K.-H.; YAGHI, A.; TABOR, D.; SINHA, R.; MAZUMDAR, S. Data visualization in ai-assisted decision-making: a systematic review. **Frontiers in Communication**, 2025. Disponível em: <https://www.frontiersin.org/journals/communication/articles/10.3389/fcomm.2025.1605655>. Citado na página 22.

REIS, J.; HOUSLEY, M. **Fundamentals of Data Engineering: Plan and Build Robust Data Systems**. [S.l.]: O'Reilly Media, 2022. Citado na página 18.

RICHMAN, J. **What Is Real-Time Processing (In-depth Guide For Beginners)**. 2025. <https://estuary.dev/blog/what-is-real-time-processing/#:~:text=Real%2Dtime%20data%20processing%20refers,available%20for%20use%20almost%20instantly.> Citado na página 12.

SAJIDA, S.; RAMAKRISHNA, S. A study of extract-transform-load (ETL) processes. **International Journal of Engineering Research & Technology (IJERT)**, v. 3, n. 18, 2015. Special Issue: NCACI - 2015. Citado 2 vezes nas páginas 19 e 20.

SANTOS, D. A. M.; SILVA, F. W. R.; MACHADO, J. Telemetria automotiva e suas aplicações da preditiva. In: **Anais do 3° Simpósio de TCC, das Faculdades FINOM e Tecsoma**. [S.l.: s.n.], 2020. p. 1052–1074. Citado na página 20.

SCISURE. **Data Integrity in Labs: Why It is Essential and How to Achieve It**. 2025. Https://www.scisure.com/blog/data-integrity-in-labs-why-its-essential-and-how-to-achieve-it. Citado na página 12.

SEENIVASAN, D. Etl (extract, transform, load) best practices. **International Journal of Computer Trends and Technology**, v. 71, p. 40–44, 01 2023. Citado na página 19.

SHAH, B.; JAT, P.; SASHIDHAR, K. **Performance Study of Time Series Databases**. 2022. Citado na página 21.

SUMAN, S. **The Importance of data quality in decision-making**. 2025. Https://www.expresscomputer.in/guest-blogs/the-importance-of-data-quality-in-decision-making/125738/. Citado na página 12.

TAN, F. **Development of Electric Formula SAE Real-time Telemetry Software**. 2011. Disponível em: <https://api.semanticscholar.org/CorpusID:111186258>. Citado na página 28.

TANENBAUM, A. S.; WETHERALL, D. J. **Computer Networks**. 5th. ed. Boston: Pearson Education, 2011. ISBN 978-0132126953. Disponível em: <https://www.inf.ufsc.br/~bosco.sobral/ensino/ine5645/Computer-Networks---A-Tanenbaum---5th-edition.pdf>. Citado na página 21.

TUKEY, J. W. The future of data analysis. **The Annals of Mathematical Statistics**, Institute of Mathematical Statistics, v. 33, n. 1, p. 2, 1962. Disponível em: <http://www.mat.ufrgs.br/~viali/estatistica/mat2274/material/textos/2237638.pdf>. Citado na página 18.

WIJESINGHE, T. M. H. S. B.; ABEYSINGHE, A. M. H. R.; SAMARANAYAKE, V. S. G. P. Big data for predictive maintenance in industry 4.0: Enhancing operational efficiency and equipment reliability. **International Journal of Computer Applications Technology (IJCAT)**, v. 13, n. 10, p. 1003–1010, 2024. Citado na página 15.

XU, J.; ZHANG, J.; LI, J.; LI, G. Real-time big data stream processing: A survey. **Future Generation Computer Systems**, Elsevier, v. 82, p. 681–692, 2018. Citado na página 19.