Hydra Flow - Pipeline Automatizada para ETL de Dados

Bianca Motta Martins[1], **Carlos Alberto da Silva**[2]

[1]Bacharelado em Sistemas de Informação
[2]Faculdade de Computação (FACOM)
Universidade Federal de Mato Grosso do Sul (UFMS)
Campo Grande - MS

b.motta@ufms.br, carlos.silva@ufms.br

Abstract. *This paper presents Hydra Flow, a data pipeline based on the ETL (Extract, Transform and Load) architecture for collecting, transforming, and storing data obtained from web APIs in JSON format. The solution uses Apache Airflow for orchestration, MinIO as S3-compatible storage, and Apache Parquet for persisting processed data. The architecture is organized into raw, staged, and curated layers, enabling the separation of raw, processed, and refined data for analytical purposes. During the ingestion stage, jq expressions are used for data extraction and initial filtering, while subsequent stages perform data standardization and transformation. The obtained results demonstrate the feasibility of using open-source tools to build automated, modular, and maintainable data pipelines.*

Resumo. *Este trabalho apresenta o Hydra Flow, uma pipeline de dados baseada na arquitetura ETL para coleta, transformação e armazenamento de dados provenientes de APIs web em formato JSON. A solução utiliza Apache Airflow para orquestração, MinIO como armazenamento compatível com S3 e Apache Parquet para persistência dos dados processados. A arquitetura é organizada em camadas raw, staged e curated, permitindo a separação entre dados brutos, tratados e refinados para consumo analítico. Durante a ingestão, expressões jq são utilizadas para extração e seleção inicial dos dados, enquanto etapas posteriores realizam sua padronização e transformação. Os resultados obtidos demonstram a viabilidade da utilização de ferramentas de código aberto para construção de pipelines de dados automatizadas, modulares e de fácil manutenção.*

Palavras-chave: ETL. Pipeline de dados. Airflow. MinIO. Engenharia de dados.

1. Introdução

A crescente digitalização de processos e serviços tem impulsionado a geração contínua de dados em diferentes formatos e volumes. Aplicações web, APIs, dispositivos conectados, sistemas corporativos e plataformas digitais produzem diariamente grandes quantidades de informações que precisam ser coletadas, armazenadas e processadas para que possam gerar valor. Nesse contexto, a capacidade de construir soluções capazes de automatizar o fluxo de dados tornou-se um requisito importante para organizações e profissionais que dependem de informações atualizadas para análise e tomada de decisão.

No centro desse cenário está o processo ETL (*Extract, Transform and Load*), responsável por extrair dados de diferentes fontes, aplicar transformações necessárias para garantir qualidade e padronização e, por fim, carregá-los em sistemas de armazenamento destinados ao consumo analítico ou operacional. O ETL é largamente adotado em ambientes de engenharia de dados por permitir a integração de informações provenientes de fontes heterogêneas e por contribuir para a construção de bases de dados consistentes e reutilizáveis.

Com a evolução das tecnologias de armazenamento e processamento de dados, surgiram arquiteturas modernas capazes de lidar com grandes volumes de informação de forma escalável. Entre elas destacam-se os *Data Lakes*, que permitem armazenar dados em diferentes estágios de processamento, preservando tanto sua forma original quanto versões tratadas e refinadas para consumo analítico. Paralelamente, ferramentas de orquestração e automação possibilitaram a construção de *pipelines* capazes de executar tarefas de coleta, transformação e armazenamento de forma contínua e confiável.

Apesar da ampla disponibilidade de soluções para processamento de dados, muitas plataformas exigem infraestrutura complexa, possuem custos elevados de implantação ou demandam conhecimentos avançados para configuração e manutenção. Esse cenário cria dificuldades para estudantes, pesquisadores e pequenos projetos que desejam implementar *pipelines* de dados seguindo boas práticas da área sem depender de serviços proprietários ou ambientes de grande porte.

Diante desse contexto, surge o seguinte problema de pesquisa: como construir uma *pipeline* de dados automatizada, baseada em ferramentas de código aberto, capaz de realizar a ingestão, transformação, armazenamento e orquestração de dados de forma organizada, escalável e de fácil manutenção?

Para responder a essa questão, foi desenvolvido o projeto Hydra Flow, uma *pipeline* de dados baseada em Python, Apache Airflow e MinIO, estruturada segundo uma arquitetura em camadas `raw`, `staged` e `curated`. Este trabalho tem como objetivo desenvolver e validar uma solução capaz de coletar, transformar, armazenar e orquestrar dados provenientes de APIs web, utilizando ferramentas de código aberto e seguindo práticas modernas de engenharia de dados. A solução proposta busca demonstrar a viabilidade da aplicação dessas tecnologias na construção de um ambiente reproduzível, modular e de baixo custo para processamento de dados.

2. Fundamentação Teórica

O processo de ETL (*Extract, Transform and Load*) é um dos pilares da integração de dados, sendo responsável pela extração de informações de diferentes fontes, sua transformação conforme regras definidas e posterior carregamento em ambientes de armazenamento e análise. Embora frequentemente associado à construção de *Data Warehouses*, o ETL também desempenha papel estratégico em diferentes cenários de integração e preparação de dados, contribuindo para a padronização, qualidade e reutilização dos dados [Abreu 2008, Kimball and Caserta 2011].

Com a evolução da engenharia de dados, os processos de ETL passaram a integrar arquiteturas mais amplas representadas por *pipelines* de dados. Essas estruturas são compostas por etapas encadeadas responsáveis pela movimentação e processamento contínuo

de informações entre sistemas, permitindo escalabilidade, confiabilidade e redução da intervenção manual [Kleppmann 2017].

O crescimento da geração de dados impulsionado pela digitalização de processos e pela expansão da internet intensificou a adoção de arquiteturas capazes de lidar com grandes volumes de informação. Os ambientes de Big Data são tradicionalmente caracterizados pelos desafios relacionados ao volume, velocidade e variedade dos dados, posteriormente ampliados para incluir veracidade e valor, compondo os chamados 5 Vs do *Big Data* [Gandomi and Haider 2015]. Esse cenário reforça a necessidade de soluções capazes de processar e armazenar dados provenientes de fontes heterogêneas de forma eficiente e escalável.

Nesse contexto, os *Data Lakes* surgiram como uma arquitetura destinada ao armazenamento de grandes volumes de dados em diferentes formatos, preservando-os em seu estado original e permitindo sua utilização em múltiplos cenários de processamento e análise. Diferentemente dos *Data Warehouses*, que tradicionalmente requerem modelagem prévia dos dados, os *Data Lakes* oferecem maior flexibilidade para ingestão de dados estruturados, semiestruturados e não estruturados. Diferentes arquiteturas de *Data Lakes* organizam os dados em zonas de acordo com seu grau de refinamento, separando as etapas de ingestão, armazenamento, processamento e disponibilização dos dados para consumo analítico [Sawadogo and Darmont 2021].

Seguindo esse princípio de organização por níveis de refinamento, este trabalho adota uma arquitetura simplificada composta pelas camadas *raw*, *staged* e *curated*. A camada *raw* preserva os dados em seu formato original, permitindo auditoria e reprocessamento quando necessário. A camada *staged* concentra as etapas intermediárias de limpeza, padronização e transformação dos dados, enquanto a camada *curated* armazena os dados refinados e preparados para consumo analítico. Essa organização facilita a separação das diferentes etapas do processamento de dados, favorecendo a rastreabilidade das transformações realizadas e o reprocessamento das informações ao longo da *pipeline* [Sawadogo and Darmont 2021].

Para implementação desse tipo de arquitetura, soluções compatíveis com o protocolo S3 tornaram-se amplamente utilizadas devido à interoperabilidade com ferramentas analíticas modernas. O MinIO destaca-se como uma alternativa de código aberto para armazenamento de objetos, oferecendo compatibilidade com a API Amazon S3 e permitindo sua utilização como base para *Data Lakes* locais ou distribuídos [MinIO 2026].

Um ponto importante em *pipelines* de dados é a orquestração dos processos. O Apache Airflow é uma plataforma de código aberto destinada à automação, agendamento e monitoramento de fluxos de trabalho. Sua arquitetura baseada em DAGs (*Directed Acyclic Graphs*) permite representar dependências entre tarefas de forma explícita, facilitando o controle, a observabilidade e o reprocessamento de execuções [Apache Software Foundation 2026].

Ferramentas especializadas também desempenham papel importante na transformação de dados semiestruturados. A linguagem jq foi desenvolvida para consulta, filtragem e transformação de documentos JSON de forma declarativa e eficiente. Como JSON é amplamente utilizado para intercâmbio de dados em aplicações web e APIs [Bray 2017], o uso de jq oferece uma alternativa leve e flexível para o pré-processamento de informa-

ções durante a etapa de ingestão [jqlang 2025].

Outro elemento relevante em *pipelines* analíticas é o formato de armazenamento adotado. O Apache Parquet utiliza organização colunar dos dados, possibilitando leituras seletivas, compressão eficiente e melhor desempenho em consultas analíticas quando comparado a abordagens tradicionais orientadas a linhas [Melnik et al. 2010, Stonebraker et al. 2005]. Além disso, o formato é compatível com ferramentas modernas de processamento distribuído, tornando-se uma escolha frequente em ambientes de *Data Lake*.

Dessa forma, os conceitos apresentados nesta fundamentação indicam que soluções de engenharia de dados combinam processos de ETL, *pipelines* automatizadas, mecanismos de orquestração e arquiteturas de armazenamento capazes de suportar o processamento escalável de grandes volumes de dados. Nesse contexto, a utilização de *Data Lakes* organizados por níveis de refinamento fornece uma base arquitetural adequada para o desenvolvimento da solução proposta neste trabalho.

3. Metodologia e Desenvolvimento

3.1. Arquitetura e Organização do código

A arquitetura representada na Figura 1 foi projetada seguindo princípios de separação de responsabilidades, modularização e facilidade de manutenção, permitindo que novas funcionalidades sejam adicionadas com impacto mínimo sobre os componentes já existentes. O projeto organiza o código-fonte segundo o padrão `src/`, muito utilizado em aplicações Python para separar o código de negócio dos demais artefatos do projeto, como documentação, testes e arquivos de infraestrutura.

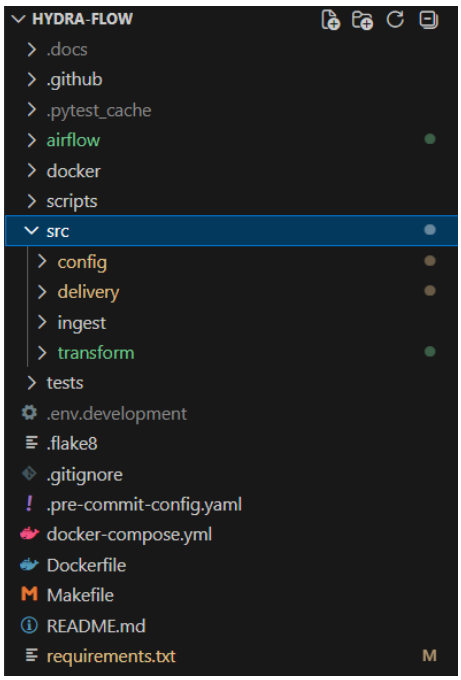

Figura 1. Organização dos arquivos

A estrutura é dividida em pacotes especializados, cada um responsável por uma etapa

específica da *pipeline* de dados. O módulo `config` centraliza as definições das fontes de dados, regras de extração e parâmetros de execução, permitindo que novas integrações sejam adicionadas sem alterações significativas na lógica da aplicação. O módulo `ingest` é responsável pela coleta dos dados nas fontes externas e pela aplicação inicial das regras de extração definidas para cada endpoint. Em seguida, os dados são encaminhados para o módulo `transform`, que concentra as rotinas de limpeza, padronização e enriquecimento dos registros, mantendo a lógica de negócio desacoplada das demais etapas do fluxo. Por fim, o módulo `delivery` executa a persistência dos dados processados nas diferentes camadas do *Data Lake* implementado sobre MinIO.

Além dos módulos responsáveis pela *pipeline*, a arquitetura contempla diretórios auxiliares que contribuem para a organização do projeto. A pasta `airflow` contém os DAGs e arquivos relacionados à orquestração dos processos, permitindo a automação das execuções. O diretório `tests`, localizado na raiz do projeto, concentra testes de escopo global e validações que envolvem a integração entre diferentes componentes da aplicação. Complementarmente, cada módulo funcional possui seu próprio diretório `tests`, contendo testes unitários específicos para suas respectivas responsabilidades. Essa estratégia aproxima os testes do código que está sendo validado, facilitando a manutenção, a identificação de falhas e a evolução independente de cada camada do sistema.

Os diretórios `docker` e `scripts` armazenam recursos de infraestrutura e automação utilizados durante o desenvolvimento e a execução do ambiente local. Já a pasta .github concentra as configurações da *pipeline* de Integração Contínua (CI, do inglês *Continuous Integration*), responsável pela execução automatizada de verificações de qualidade e testes a cada alteração submetida ao repositório. Arquivos como `docker-compose.yml`, `Dockerfile`, `Makefile`, `requirements.txt` e `.pre-commit-config.yaml` complementam a infraestrutura do projeto, fornecendo mecanismos padronizados para construção do ambiente, gerenciamento de dependências, execução de testes e validações de qualidade de código.

Essa organização reduz o acoplamento entre as camadas, melhora a legibilidade do código e favorece a aplicação de testes em diferentes níveis de granularidade. Como consequência, a inclusão de novas fontes de dados, novas transformações ou novos mecanismos de entrega pode ser realizada de forma incremental, preservando a estabilidade dos componentes existentes. Dessa forma, a arquitetura adotada contribui para a escalabilidade, manutenibilidade e evolução contínua do projeto, características fundamentais para *pipelines* de dados que tendem a crescer em complexidade ao longo do tempo.

3.2. Expressões de Extração e Conversão

As regras de extração são definidas por expressões `jq` declaradas em `src/config`. O uso de jq foi escolhido pela expressividade e pela capacidade de transformar JSON de forma declarativa, reduzindo a necessidade de código procedural no estágio de ingestão. Conforme ilustrado na Figura 2, a expressão utilizada permite selecionar apenas chaves específicas do JSON retornado pela API, possibilitando a padronização inicial dos dados recebidos e a remoção antecipada de informações consideradas irrelevantes para os objetivos analíticos do projeto.

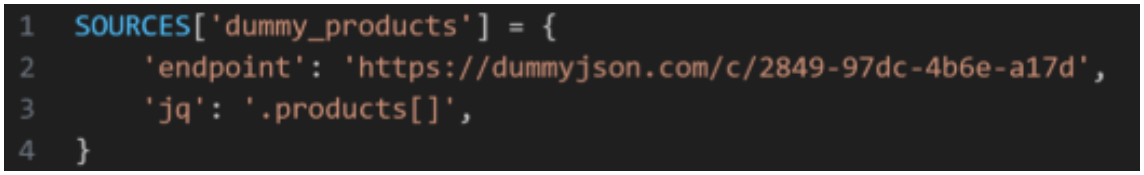

Figura 2. Declaração jq

Embora essa filtragem inicial realize uma pré-limpeza dos dados, ela não caracteriza o processo completo de transformação. Sua principal finalidade é reduzir o volume de dados desnecessários armazenados nas camadas iniciais da *pipeline*, minimizando o consumo de armazenamento e evitando a propagação de informações sem valor para as etapas subsequentes.

Dessa forma, o fluxo implementado mantém as características gerais de um processo **ETL**, uma vez que as transformações efetivas ocorrem posteriormente na camada de processamento. Nessa etapa são aplicadas regras de padronização, normalização e tratamento dos dados, enquanto a utilização do `jq` na ingestão atua apenas como um mecanismo de pré-processamento e seleção dos dados de interesse.

3.3. Infraestrutura de Contêineres e Rede Dedicada

Para replicar um ambiente próximo ao de produção em máquina local, adotou-se Docker Compose para orquestrar MinIO e Apache Airflow. Os serviços são conectados a uma rede Docker dedicada denominada `hydra-flow-network`, o que isola o tráfego do projeto e evita colisões de portas e nomes em ambientes de desenvolvimento. Um container auxiliar (`minio-init`) executa a criação automática do bucket `data-lake` e dos prefixos `raw/`, `staged/` e `curated/` na inicialização. Além disso, o `docker-compose.yml` define verificações de integridade (healthchecks) para MinIO e Apache Airflow e políticas de reinício (`restart: unless-stopped`) para melhorar a resiliência em ambientes de desenvolvimento.

As credenciais sensíveis, como usuário e senha do MinIO e o usuário administrador do Apache Airflow, foram movidas para um arquivo de ambiente local `.env.development`. Esse arquivo é carregado pelos serviços para permitir que MinIO e Apache Airflow inicializem automaticamente com as credenciais corretas sem necessidade de inserção manual.

Como decisão técnica voltada à simplificação do ambiente local, o Apache Airflow é executado por meio do comando `airflow standalone`, que inicializa scheduler, webserver e banco de dados de forma coordenada e expõe a interface web na porta `:8080`. O usuário administrador é criado automaticamente a partir das variáveis definidas em `.env.development`.

3.4. Implementação da DAG

Com o objetivo de automatizar a execução da *pipeline* de dados, foi desenvolvida uma DAG (*Directed Acyclic Graph*) utilizando o Apache Airflow. A DAG foi implementada no diretório `airflow/dags` e projetada para orquestrar o fluxo de processamento de forma sequencial, seguindo a arquitetura em camadas adotada pelo projeto.

A estrutura da DAG é composta por três tarefas principais: `ingest`, `staged` e `curated`. Cada tarefa possui uma responsabilidade específica e atua como mecanismo de integração entre o orquestrador e os módulos especializados do código-fonte.

A tarefa `ingest` é responsável por executar o processo de coleta dos dados definidos na camada de configuração, acionando os componentes do módulo de ingestão. Em seguida, a tarefa `staged` executa as rotinas de transformação e padronização implementadas no módulo de processamento, preparando os dados para consumo analítico. Por fim, a tarefa `curated` aplica as regras de refinamento dos dados e realiza a persistência da versão final destinada às consultas e análises.

As dependências entre as tarefas foram definidas de forma linear (`ingest` → `staged` → `curated`), refletindo o fluxo natural da *pipeline*. Assim cada etapa é executada somente após a conclusão bem-sucedida da anterior. Essa abordagem simplifica o monitoramento das execuções e facilita a identificação de falhas durante o processamento.

Para automatizar a atualização periódica dos dados, a DAG foi configurada com a expressão *cron* `0 * * * 1-5`, que determina sua execução no início de cada hora durante os dias úteis da semana. As expressões cron constituem um padrão utilizado em sistemas operacionais e ferramentas de orquestração para definição de agendamentos periódicos, permitindo especificar horários de execução por meio de uma sintaxe compacta baseada em minutos, horas, dias e meses. A utilização de expressões cron foi escolhida por ser uma solução frequentemente adotada em sistemas de agendamento, permitindo flexibilidade para ajustes futuros na frequência de execução sem alterações na lógica da aplicação. Além do agendamento automático, a DAG também pode ser executada manualmente por meio da interface do Apache Airflow, possibilitando testes, reprocessamentos e validações sob demanda.

A Figura 3 apresenta a DAG implementada e o relacionamento entre as tarefas responsáveis pela execução da *pipeline*.

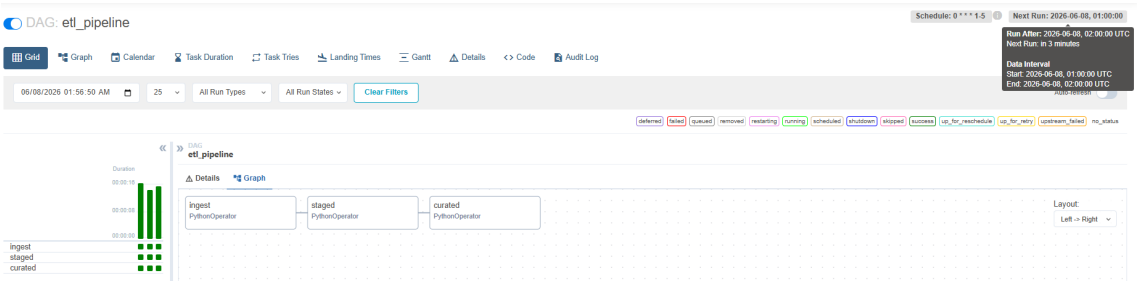

Figura 3. DAG etl-pipeline

3.5. Fluxo Geral da Pipeline

Embora as seções anteriores descrevam individualmente os principais componentes da solução proposta, a Figura 4 complementa a descrição apresentada anteriormente, oferecendo uma visão integrada do funcionamento da solução implementada. O fluxo representa a comunicação entre as APIs utilizadas como fonte de dados, os módulos responsáveis pelas etapas de processamento, a orquestração realizada pelo Apache Airflow e o armazenamento no Data Lake implementado sobre MinIO.

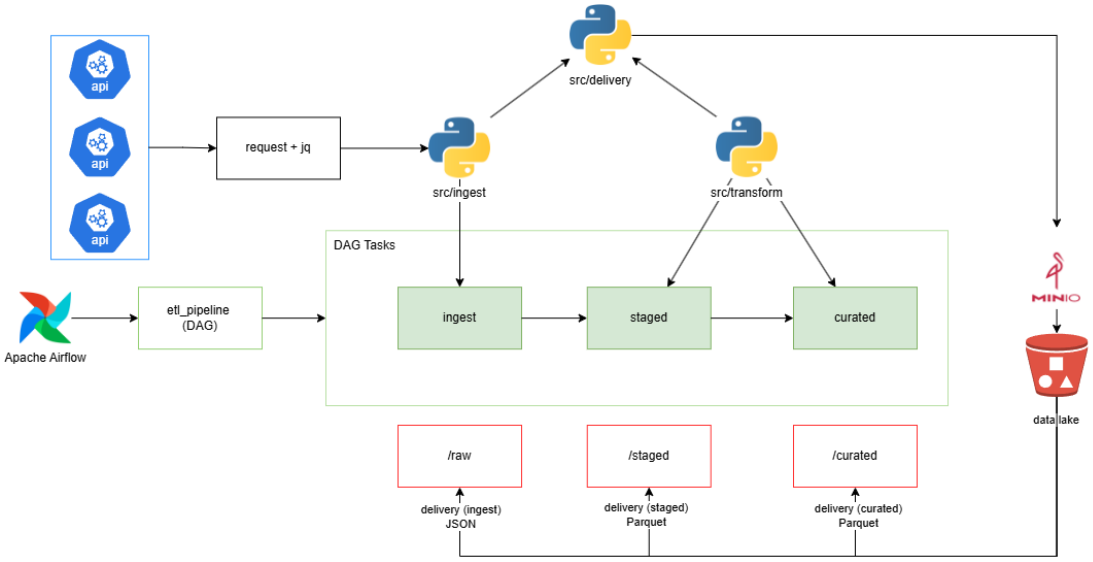

Figura 4. Diagrama do fluxo de execução do projeto

3.6. Escolha de Bibliotecas e Versões

Para reduzir problemas de compatibilidade, as dependências foram fixadas em `requirements.txt`. As bibliotecas utilizadas são:

- `requests==2.31.0` — cliente web maduro e amplamente utilizado;
- `jq==1.3` — binding Python para execução de expressões jq;
- `boto3==1.28.36` — cliente AWS S3 compatível com MinIO;
- `python-dotenv==1.0.0` — leitura de variáveis de ambiente locais;
- `pytest==7.4.0` — framework de testes;
- `flake8==6.1.0` — ferramenta de análise estática;
- `pre-commit==3.4.0` — automação de verificações antes de commits;
- `pandas==2.2.3` — manipulação de dados;
- `pyarrow==12.0.1` — escrever/ler arquivos parquet.

O pinning de versões facilita a reprodução do ambiente em diferentes máquinas e evita regressões provocadas por incompatibilidade em novas versões das bibliotecas.

3.7. Estratégia de Testes e Qualidade

O projeto inclui testes unitários organizados próximos aos pacotes funcionais, permitindo validações isoladas das etapas. A meta futura é implementar testes de integração capazes de validar o fluxo completo entre Apache Airflow e MinIO. A *pipeline* de qualidade executa *flake8* e *pytest* localmente por meio do make, enquanto o pre-commit impede commits com mensagens fora do padrão `<type>: <message>`.

3.8. Integração Contínua

Foi implementada uma *pipeline* básica de Integração Contínua (CI) em `.github/workflows/ci.yml`, acionada em eventos de *push* e *pull request*. Esse *workflow* realiza o checkout do código, configura o `Python 3.11`, instala as dependências com `pip install -r requirements.txt`, executa verificações estáticas com *flake8* e roda testes unitários utilizando `pytest -q`.

A *pipeline* de Integração Contínua garante que *commits* e *pull requests* mantenham padrões mínimos de qualidade antes de serem incorporados ao repositório principal. A cada alteração submetida, são executadas automaticamente verificações de qualidade de código e testes automatizados, reduzindo a probabilidade de regressões e falhas introduzidas durante o desenvolvimento. Os resultados dessas validações ficam disponíveis diretamente na interface dos *pull requests* do GitHub, plataforma utilizada para hospedagem e versionamento do código-fonte do projeto, permitindo que os desenvolvedores identifiquem e corrijam problemas antes da integração das alterações. As Figuras 5 e 6 apresentam, respectivamente, a execução da *pipeline* de CI em andamento e sua conclusão bem-sucedida em um *pull request* submetido ao repositório.

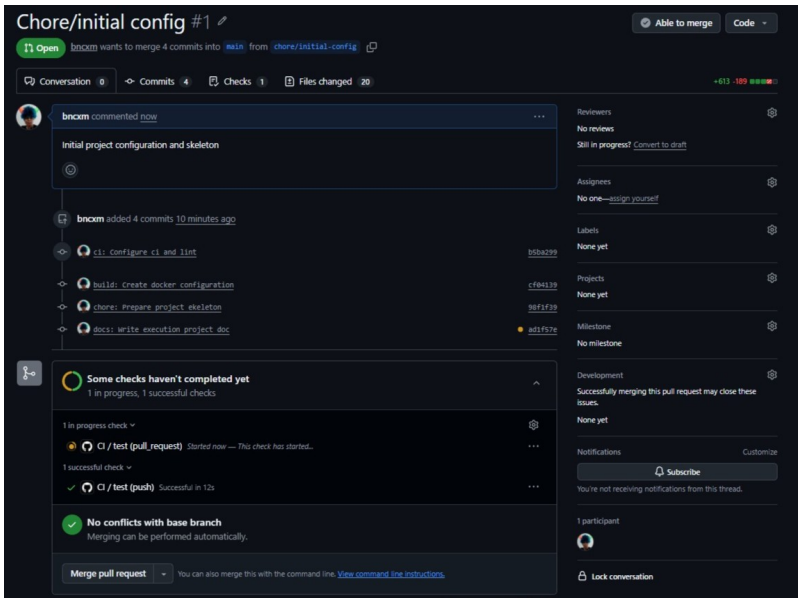

Figura 5. CI em progresso

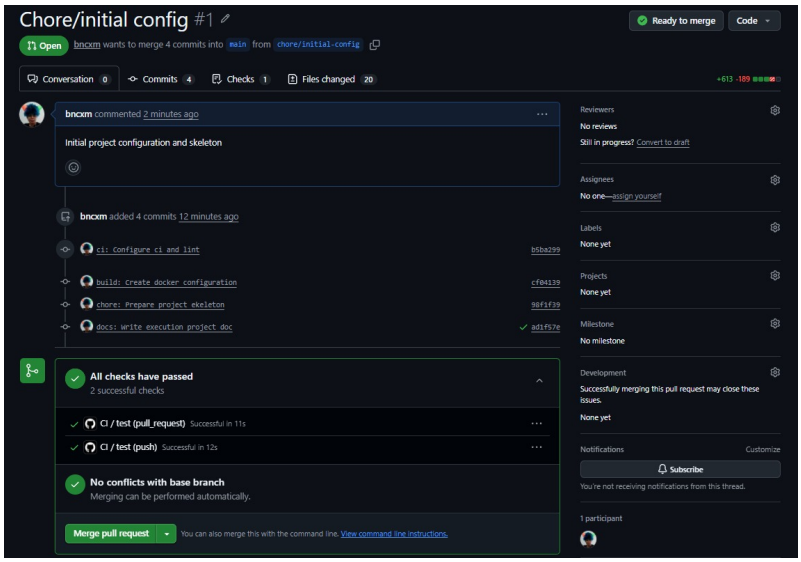

Figura 6. CI finalizada

4. Resultado e Discussão

4.1. Execução da Pipeline

Após a implementação da DAG, a *pipeline* passou a executar automaticamente de acordo com o agendamento configurado. A execução ocorre de forma sequencial, percorrendo as tarefas `ingest`, `staged` e `curated`, responsáveis pelo processamento dos dados nas diferentes camadas do *Data Lake*. A Figura 7 apresenta a DAG carregada no Apache Airflow e configurada para execução periódica, enquanto a Figura 8 demonstra uma execução concluída com sucesso.

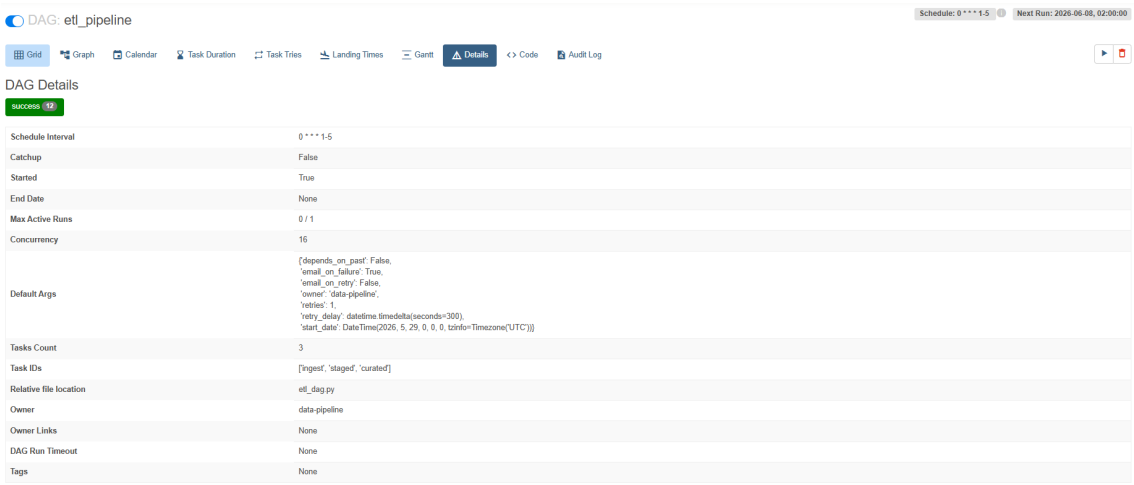

Figura 7. Configuração da DAG

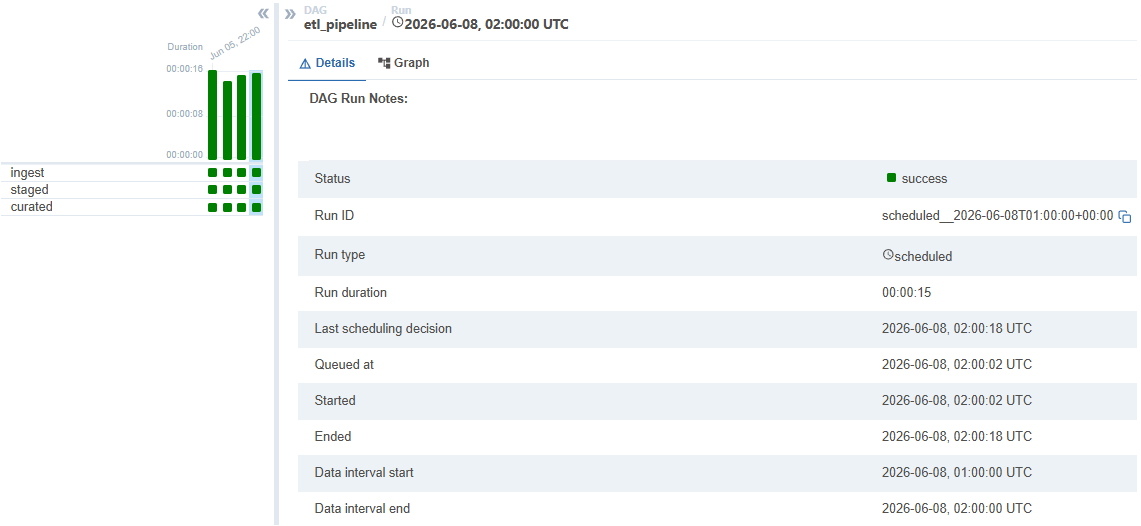

Figura 8. Execução bem-sucedida da DAG

4.2. Resultados nas camadas do Data Lake

A execução da *pipeline* resultou na persistência dos dados nas três camadas do *Data Lake* implementado sobre MinIO. Cada camada possui uma finalidade específica dentro da arquitetura proposta.

Na camada `raw`, os dados são armazenados em formato JSON, preservando integralmente a estrutura recebida da fonte de dados. Essa estratégia permite manter um histórico fiel dos dados originalmente coletados, funcionando como mecanismo de auditoria e rastreabilidade.

A Figura 9 apresenta um exemplo dos dados armazenados na camada `raw`.

Figura 9. Dados JSON camada raw

Na camada `staged`, os dados passam pelo processo de limpeza e padronização descrito na metodologia. Nessa etapa nenhuma informação é removida, sendo realizadas apenas transformações voltadas à melhoria da qualidade e consistência dos dados. Após o processamento, os registros são armazenados em formato `Parquet`.

Durante os testes, o arquivo JSON armazenado na camada `raw` apresentou aproximadamente 1010 B, enquanto o arquivo correspondente em formato `Parquet` ocupou cerca de 20,1 KiB. Esse comportamento é esperado em conjuntos de dados reduzidos, uma vez que o formato Parquet adiciona metadados estruturais necessários para o armazenamento colunar, como definição de esquema, estatísticas e informações de codificação. Embora isso resulte em arquivos maiores em cenários de pequeno volume, o formato tende a apresentar ganhos significativos de compressão e desempenho de leitura à medida que a quantidade de registros aumenta, justificando sua adoção na arquitetura proposta.

A Figura 10 apresenta os dados resultantes da camada `staged`.

Figura 10. Dados parquet na camada staged

Por fim, na camada `curated`, os registros são reduzidos aos atributos considerados relevantes para o contexto analítico definido pelo projeto. Essa etapa gera um conjunto de dados mais enxuto e preparado para futuras consultas, relatórios ou aplicações analíticas.

A Figura 11 apresenta os dados armazenados na camada `curated`.

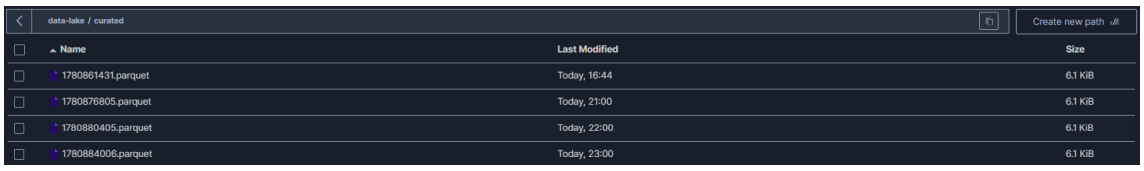

Figura 11. Dados parquet na camada curated

Como estratégia de rastreabilidade, todos os arquivos gerados pela *pipeline* utilizam como nome o *timestamp* correspondente ao momento da ingestão, seguido da extensão do formato armazenado. Dessa forma, cada artefato pode ser associado de maneira inequívoca à execução responsável por sua geração. Durante os testes realizados foram produzidos arquivos como `1780876805.json` na camada `raw` e `1780876805.parquet` nas camadas `staged` e `curated`.

4.3. Potencial de Aplicação

Um possível cenário de aplicação da solução proposta envolve o monitoramento automatizado de indicadores financeiros disponibilizados periodicamente por APIs web. Nesse contexto, a *pipeline* pode ser configurada para executar em horários específicos, realizar a coleta automática dos dados e armazená-los no *Data Lake*, reduzindo atividades manuais de extração e atualização de informações. A partir dos dados armazenados, ferramentas analíticas podem ser utilizadas para construção de relatórios e painéis de acompanhamento, permitindo que o usuário dedique mais tempo à análise das informações e à tomada de decisão. Como a arquitetura foi desenvolvida utilizando ferramentas de código aberto executáveis localmente, todo o processo pode operar sem dependência de serviços proprietários ou custos adicionais de infraestrutura.

5. Conclusão

Este trabalho apresentou o desenvolvimento do Hydra Flow, uma *pipeline* de dados baseada na arquitetura ETL para ingestão, transformação e armazenamento automatizado de dados provenientes de APIs web. A solução foi implementada utilizando ferramentas de código aberto, incluindo Python, Apache Airflow, MinIO e Apache Parquet, organizadas em uma arquitetura composta pelas camadas `raw`, `staged` e `curated`.

Os resultados obtidos demonstraram a viabilidade da implementação de fluxos automatizados de processamento de dados em ambientes de baixo custo, utilizando componentes adotados na engenharia de dados. A arquitetura proposta permitiu a separação entre dados brutos, tratados e refinados, favorecendo rastreabilidade, manutenção e reutilização das informações ao longo do ciclo de processamento.

Além disso, a utilização do Apache Airflow para orquestração e do MinIO como *Data Lake* compatível com S3 mostrou-se adequada para automatizar a execução das etapas da *pipeline* e garantir a persistência dos dados em diferentes níveis de refinamento. A adoção do formato Apache Parquet também se mostrou alinhada às práticas atuais da área, especialmente em cenários que demandam processamento e análise de grandes volumes de dados.

Como trabalhos futuros, destacam-se a ampliação do número de fontes monitoradas, a implementação de mecanismos mais robustos de validação da qualidade dos dados, otimizações na frequência de execução, a adoção de recursos de observabilidade e a implantação da solução em ambientes distribuídos. Também pretende-se integrar a arquitetura a ferramentas de visualização de dados, permitindo a atualização automática de *dashboards* analíticos a partir das informações armazenadas no *Data Lake*, reduzindo ainda mais a intervenção manual no processo de tomada de decisão.

Referências

Abreu, F. S. G. d. G. e. (2008). Desmistificando o Conceito de ETL. *FSMA, Revista de Sistemas de Informação*. Disponível em: https://www.fsma.edu.br/si/Artigos/V2_Artigo1.pdf. Acesso em: 12 jul. 2025.

Apache Software Foundation (2026). Apache Airflow Documentation. Disponível em: https://airflow.apache.org/docs/. Acesso em: 9 jun. 2026.

Bray, T. (2017). The JavaScript Object Notation (JSON) Data Interchange Format. RFC 8259. IETF. Disponível em: https://www.rfc-editor.org/info/rfc8259/. Acesso em: 02 set. 2025.

Gandomi, A. and Haider, M. (2015). Beyond the Hype: Big Data Concepts, Methods, and Analytics. *International Journal of Information Management*. Disponível em: https://www.sciencedirect.com/science/article/pii/S0268401214001066. Acesso em: 10 ago. 2025.

jqlang (2025). jq Manual. jqlang.org. Versão 1.8.1. Disponível em: https://jqlang.org. Acesso em: 07 jun. 2026.

Kimball, R. and Caserta, J. (2011). *The Data Warehouse ETL Toolkit: Practical Techniques for Extracting, Cleaning, Conforming, and Delivering Data*. Wiley Publishing, Indianapolis. E-book.

Kleppmann, M. (2017). *Designing Data-Intensive Applications*. O'Reilly Media, Sebastopol.

Melnik, S. et al. (2010). Dremel: Interactive Analysis of Web-Scale Datasets. In *Proceedings of the VLDB Endowment / VLDB 2010*, pages 330–339, Singapore. Disponível em: https://research.google/pubs/dremel-interactive-analysis-of-web-scale-datasets-2/. Acesso em: 30 out. 2025.

MinIO, Inc. (2026). MinIO AIStor Documentation. Disponível em: https://docs.min.io/aistor/. Acesso em: 9 jun. 2026.

Sawadogo, P. N. and Darmont, J. (2021). On data lake architectures and metadata management. *Journal of Intelligent Information Systems*. Disponível em: https://arxiv.org/abs/2107.11152. Acesso em: 23 jun. 2026.

Stonebraker, M. et al. (2005). C-Store: A Column-Oriented DBMS. In *Proceedings of the 31st International Conference on Very Large Data Bases (VLDB)*. Disponível em: https://web.stanford.edu/class/cs345d-01/rl/cstore.pdf. Acesso em: 20 jan. 2026.