Construção de Pipelines de Dados sobre Obras Públicas em Pernambuco: Abordagem Prática com o *Apache Airflow*

Henrique César José da Silva [1], Gabriel Alves de Albuquerque Júnior [2]

[1]Departamento de Estatística e Informática – Universidade Federal Rural de Pernambuco
Rua Dom Manuel de Medeiros, s/n, - CEP: 52171-900 – Recife – PE – Brasil

henrique.cesar@ufrpe.br, gabriel.alves@ufrpe.br

Resumo. *Este estudo apresenta uma abordagem prática para a construção de pipelines de dados voltados à coleta, transformação e armazenamento de informações relacionadas a obras públicas no estado de Pernambuco. O objetivo central é desenvolver fluxos de trabalho eficientes e automatizados para a extração de dados a partir de portais de transparência públicos, e a subsequente consolidação dessas informações. Com base em tecnologias de Engenharia de Dados, o framework Apache Airflow foi selecionado para a orquestração dos processos, permitindo o agendamento e monitoramento dos fluxos de trabalho.*

Abstract. *This study presents a practical approach to building data pipelines focused on collecting, transforming, and storing information related to public works in the state of Pernambuco. The central objective is to develop efficient and automated workflows for extracting data from public transparency portals and subsequently consolidating this information. Based on Data Engineering technologies, the Apache Airflow framework was chosen to orchestrate the processes, enabling the scheduling and monitoring of these workflows.*

1. Introdução

Em 2020, a Controladoria-Geral da União (CGU) apresentou um diagnóstico relevante sobre o estado de serviços de obras públicas no Brasil, o estudo que analisou 32.480 obras em todo o país, identificou que 10.916 (33,6%) tinham problemas de execução. Também foi revelado que 6.698 (64,30%) das obras paralisadas estavam localizadas em municípios com menos de 50 mil habitantes, uma quantidade oito vezes maior que municípios com mais de 500 mil habitantes, inferindo que municípios com menor capacidade técnico-administrativa são mais propensos a serem afetados por adversidades nos processos que se referem a execução: licitações, contratações e o acompanhamento das obras. A CGU também identificou que os maiores motivos de paralisação desses empreendimentos foram problemas técnicos na execução, desistência ou abandono de obra pelas empresas e contratos rescindidos [CGU 2020].

A condição do estado das obras no estado de Pernambuco apresenta algumas semelhanças em relação aos dados nacionais. O Tribunal de Contas do Estado analisou 4.935 contratos de obras e identificou que 1.754 (35,5%) delas estavam declaradamente paralisadas ou inacabadas, ou apresentavam fortes indícios de paralisação. Um dado ainda mais alarmante é encontrado na análise dos valores dos contratos que juntos totalizam R$ 12,5 bilhões, onde cerca de R$ 8,6 bilhões estavam alocados para obras paralisadas, dos quais R$ 2,5 bilhões já tinham sido gastos sem qualquer retorno para a população. O

relatório também revela que as categorias mais afetadas são em ordem: mobilidade/transporte, educação, saneamento, edificações administrativas e saúde [TCE/PE 2021].

Esses números trazem à tona a necessidade de fortalecer mecanismos de fiscalização e transparência para o desenvolvimento de uma administração pública mais eficiente. O acesso à informação desenvolve a confiança nas instituições governamentais e também capacita a sociedade a compreender as decisões de políticas públicas e monitorar sua implementação [Ferry and Eckersley 2014]. Em um contexto mais amplo, a disponibilização de informações acessíveis sobre os serviços públicos não apenas permite o acompanhamento dos projetos, mas também habilita os cidadãos a participar ativamente na tomada de decisões que afetam suas comunidades e suas vidas.

Este estudo propõe uma abordagem prática para a construção de pipelines de dados sobre obras públicas do estado de Pernambuco, com o uso do *Apache Airflow*. A proposta consiste na extração de informações de portais de transparência públicos, no tratamento e padronização dos dados coletados por meio da integração das fontes. Dessa forma, busca-se criar uma base de dados única e abrangente, que possa ajudar no desenvolvimento de uma gestão mais transparente, eficiente e estratégica desses serviços.

O processo de construção dos pipelines de dados envolve desafios como a heterogeneidade das fontes de dados, a variabilidade nos formatos e estruturas, além da necessidade de atualizações periódicas [Kakish and Kraft 2012]. Para enfrentá-los e refletir a realidade dinâmica das obras, foram utilizadas ferramentas de Engenharia de Dados, aplicando a flexibilidade e escalabilidade proporcionadas pelo *Apache Airflow* para orquestrar os fluxos de trabalho envolvidos no tratamento dos dados.

No decorrer deste artigo, serão apresentados detalhes do processo de construção dos pipelines de dados através de duas fontes públicas de dados, a API de Dados Abertos do Tribunal de Contas do Estado de Pernambuco e o Painel de Obras do Ministério de Gestão e Inovação em Serviços Públicos do Governo Federal. Contemplando o planejamento e concepção do fluxo de trabalho até a implementação e avaliação dos resultados alcançados. Esperamos que este estudo estimule discussões e inspire novas pesquisas na área de Engenharia de Dados aplicada à gestão de serviços públicos, contribuindo para o aprimoramento contínuo da administração pública e para a concretização de projetos que ajudem no bem-estar da sociedade.

1.1. Motivação

Historicamente, a gestão pública em diversos estados e municípios brasileiros enfrenta desafios relacionados à qualidade dos bens e serviços públicos disponibilizados para a sociedade [Jr 2011]. Embora este trabalho utilize o estado de Pernambuco como um estudo de caso ilustrativo, a abordagem adotada é genérica e abstrata, buscando fornecer *insights* que possam ser aplicados a qualquer estado brasileiro.

No Brasil, o acesso à informação é assegurado pela Lei de Acesso à Informação (Lei nº 12.527/2011), que reconhece o direito do cidadão de obter informações produzidas ou custodiadas pelos órgãos públicos [Brasil 2011]. Além de fortalecer a transparência e a disponibilidade de informações sobre as ações do governo e das autoridades públicas, o acesso à informação é importante para garantir a liberdade de imprensa e a livre expressão [Androniceanu 2021], pois permite que jornalistas e outros profissionais possam investigar e reportar sobre as ações do governo e outras questões de interesse público. Isso

também ajuda a garantir que as políticas públicas e as decisões estejam alinhadas com os interesses e as necessidades da sociedade.

Nesse sentido, o fornecimento de dados abertos têm o potencial de contribuir significativamente para a concretização desse direito, ao fornecer um conjunto de dados relevantes para análise e compilação de informações sobre os projetos governamentais. A escolha do uso de pipelines de dados é motivada pelas vantagens intrínsecas que essa abordagem oferece, pois além de possibilitar o acesso aos dados, também traz benefícios adicionais em termos de manutenibilidade e eficiência como a capacidade de paralelizar os processos de coleta, higienização e atualização de dados, para manter as informações sempre atualizadas e prontamente disponíveis.

2. Trabalhos Relacionados

Com o intuito de verificar a qualidade dos dados disponibilizados, [Nazário et al. 2012] apresentam um estudo detalhado sobre o uso do Portal da Transparência do governo federal brasileiro como fonte de dados públicos e sua utilidade para a realização de análises.

Os autores conduziram uma revisão da literatura sobre transparência e disponibilidade de dados públicos, destacando a importância dessas informações para a tomada de decisões e a importância do portal da transparência como meio de acesso aos dados. Em seguida, realizaram análises estatísticas descritivas dos dados disponíveis na plataforma, baseada em dezesseis critérios. Os resultados mostram que o portal da transparência oferece uma grande quantidade de informações sobre gastos públicos, entretanto os autores identificaram que muitos dos dados disponíveis não são de fácil interpretação e podem apresentar dificuldades para serem utilizados em análises empíricas por pessoas leigas.

[Smanio and Nunes 2016] aborda o uso da transparência e do monitoramento das políticas públicas como ferramentas para o fortalecimento do controle social. O estudo destaca a importância do acesso à informação e da participação da sociedade na gestão pública. Ressaltando que o desenvolvimento de uma cultura de monitoramento e avaliação das políticas públicas, pode ajudar a garantir a efetividade das ações governamentais e o uso adequado dos recursos públicos. Os autores destacam que o controle social é fundamental para o fortalecimento da democracia e para o combate à corrupção, e que a transparência e o acesso à informação são pré-requisitos para o exercício desse controle.

[Silva et al. 2014] destaca a importância da transparência governamental como um dos pilares da democracia, alcançada por meio do acesso dos cidadãos às informações governamentais. O artigo trata sobre a conexão entre transparência e informação deu origem aos conceitos de Governo Aberto (Open Government) e Dados Abertos (Open Data). O primeiro destaca a intenção ampla de um governo em ser transparente, enquanto o segundo indica o caminho para concretizar essa transparência.

O artigo propõe uma análise abrangente e contextualizada sobre a importância dos dados abertos e do governo aberto no contexto brasileiro e internacional. Apesar do quadro regulatório existente e do trabalho já realizado pelos atores envolvidos nessa política, [Silva et al. 2014] conclui que ainda há muito a ser feito para estabelecer uma cultura de dados abertos no Brasil. Para melhorar os resultados, é necessário criar um conjunto mais amplo de incentivos para as instituições, incluindo campanhas de conscientização sobre a importância do tema, a fim de estimular ainda mais a participação coordenada e ativa

das instituições públicas em iniciativas de dados abertos que abranjam os três poderes do governo: Executivo, Legislativo e Judiciário.

[Engin and Treleaven 2018] fornecem uma visão abrangente dos potenciais benefícios e desafios da governança digital, discutindo o impacto do uso de ferramentas tecnológicas como inteligência artificial (IA), big data, blockchain e análises comportamentais/preditivas nos serviços governamentais, argumentando que estas tecnologias podem melhorar a eficiência e a transparência das operações do setor público, conduzindo a melhores resultados para os cidadãos. Os autores também destacam os desafios da adaptação das tecnologias digitais às infraestruturas existentes, bem como a necessidade de educar e requalificar os funcionários públicos para adotarem as novas tecnologias.

Outros desafios sociais abordados incluem preocupações em torno do uso de dados privados de cidadãos, a justiça das práticas de tomada de decisão algorítmica, a transparência das operações públicas, a responsabilização por quaisquer danos causados por processos assistidos por computador e o potencial para perdas de empregos. Os autores enfatizam a importância de abordar estas questões para garantir que o governo digital seja implementado de forma responsável e ética.

[Figueiras et al. 2016] apresentam um pipeline de processamento de dados baseado em Big Data para coletar e harmonizar dados de fontes heterogêneas em um cenário de cobrança de pedágio em rodovias. Além disso, é apresentada uma interface web para validação do fluxo e facilitação do processo de coleta e harmonização de dados pelo usuário. O trabalho também discute possíveis melhorias futuras, como a implementação de um fluxo de dados em tempo real e a aplicação da solução em um servidor completo. O processo ETL (Extract-Transform-Load) realizado neste estudo consistiu em extrair dados brutos de tráfego de fontes heterogêneas, transformá-los em um formato unificado e carregá-los em um repositório de dados. Para isso, foram utilizadas ferramentas como o Apache Spark, para processar e transformar os dados, e o MongoDB, para armazenar os dados transformados. O processo de transformação incluiu a limpeza e normalização dos dados, bem como a aplicação de regras de negócios para garantir a qualidade dos mesmos. O objetivo do processo foi criar um conjunto de dados unificado e limpo que pudesse ser usado para análise e modelagem de negócios.

Os trabalhos acadêmicos mencionados compartilham um foco central na importância da transparência, acessibilidade e qualidade dos dados públicos, no contexto de aplicação de políticas públicas ou de iniciativas privadas. Eles destacam como o acesso à informação e a disponibilidade de dados desempenham um papel fundamental na promoção do controle social, na tomada de decisões informadas e na prestação de contas dos poderes públicos. Este trabalho pode agregar à literatura acadêmica apresentando métodos de alcançar maiores níveis de transparência, qualidade das informações e fortalecimento do controle social por meio de ferramentas tecnológicas.

3. Fundamentação Teórica

Nesta seção, com o intuito de estabelecer uma base de conhecimento sólida para compreensão e desenvolvimento deste trabalho, serão apresentados os conceitos fundamentais e uma visão geral das ferramentas, metodologias e técnicas utilizadas. Os tópicos abordados nesta seção incluem a Engenharia de Dados, um campo fundamental que fornece as bases para a organização e processamento eficiente dos dados; o *Apache Airflow*, uma

poderosa plataforma de orquestração que garante a execução automatizada e confiável de fluxos de trabalho; o conceito de Datalake com a arquitetura de medalhas, permitindo o armazenamento flexível e escalável dos dados; além de outras ferramentas de suporte que desempenham papéis cruciais na execução do projeto.

3.1. Pipelines de Dados

Pipelines de dados são conjuntos de processos que movem e transformam dados de várias fontes para um destino onde novos valores podem ser criados, com o objetivo de processar dados brutos e transformá-los em informações úteis para análises e tomadas de decisão. Os pipelines são compostos por etapas que incluem a ingestão de dados, limpeza, transformação, modelagem, orquestração de fluxo de trabalho e entrega. Esses processos ocorrem em sequência, onde cada etapa prepara os dados para a seguinte [Muller 2020].

A primeira etapa de um pipeline é a ingestão de dados, que pode envolver coletas de dados a partir de várias fontes, como bancos de dados, arquivos, APIs, páginas web e dispositivos IoT. A ingestão pode ser feita de forma manual ou automatizada, dependendo da complexidade. A limpeza é um processo que visa garantir a integridade dos dados, onde podem ser aplicadas ações de remoção de dados duplicados, incompletos ou inconsistentes, para aumentar a precisão e confiabilidade.

Na etapa de transformação os dados são convertidos em um formato utilizável para o desenvolvimento de análises, podendo incluir a conversão para um formato padronizado, agregação de dados ou aplicação de cálculos para criar novos valores. A modelagem envolve a identificação das entidades, atributos e relacionamentos entre os elementos, e a geração de esquemas que definem como os dados serão armazenados e organizados, esses modelos podem ser úteis para o aprendizado de máquina ou visualizações de dados.

Na orquestração os componentes do pipeline são coordenados e executados em sequência, possivelmente com a criação de fluxos de trabalho automatizados. Esta etapa permite que as organizações automatizem o processamento de dados, economizando recursos. A entrega de dados é a etapa final, e envolve a disponibilização dos dados processados para os usuários finais, podendo incluir documentos como relatórios e== dashboards.

Os pipelines de dados são ferramentas úteis para análise de dados e aprendizado de máquina, permitindo que as organizações processem grandes quantidades de dados de várias fontes e os transformem em informações para realização de decisões estratégicas. Existem várias ferramentas e tecnologias disponíveis para criar pipelines de dados, incluindo ferramentas de código aberto como Apache Kafka, Apache Spark e Apache Airflow, e ferramentas comerciais como AWS Glue, Google Cloud Dataflow e Microsoft Azure Data Factory. A escolha da ferramenta depende das necessidades específicas da organização, como volume de dados, complexidade do pipeline e orçamento disponível.

3.2. Apache Airflow

O *Apache Airflow* é uma plataforma de gerenciamento de pipelines de código-aberto criada pela Airbnb em 2014. O objetivo principal desta ferramenta é permitir que os usuários programem, monitorem e gerenciem fluxos de trabalho complexos de forma mais eficiente [Apache 2021].

O *Apache Airflow* usa uma abordagem baseada em *DAGs* (*Directed Acyclic Graphs*), grafos direcionados e acíclicos que representam fluxos de dados em um pipeline, sendo uma representação visual de um conjunto de tarefas que precisam ser executadas em uma ordem específica, onde cada tarefa pode depender do resultado de outras tarefas [Feng 2020]. No Apache Airflow, os DAGs são definidos em arquivos Python que descrevem a sua estrutura e suas dependências. O Airflow usa essas informações para executar as tarefas de acordo com as dependências definidas. A Figura 1 ilustra os componentes que compõem a arquitetura do Airflow e como eles se interligam.

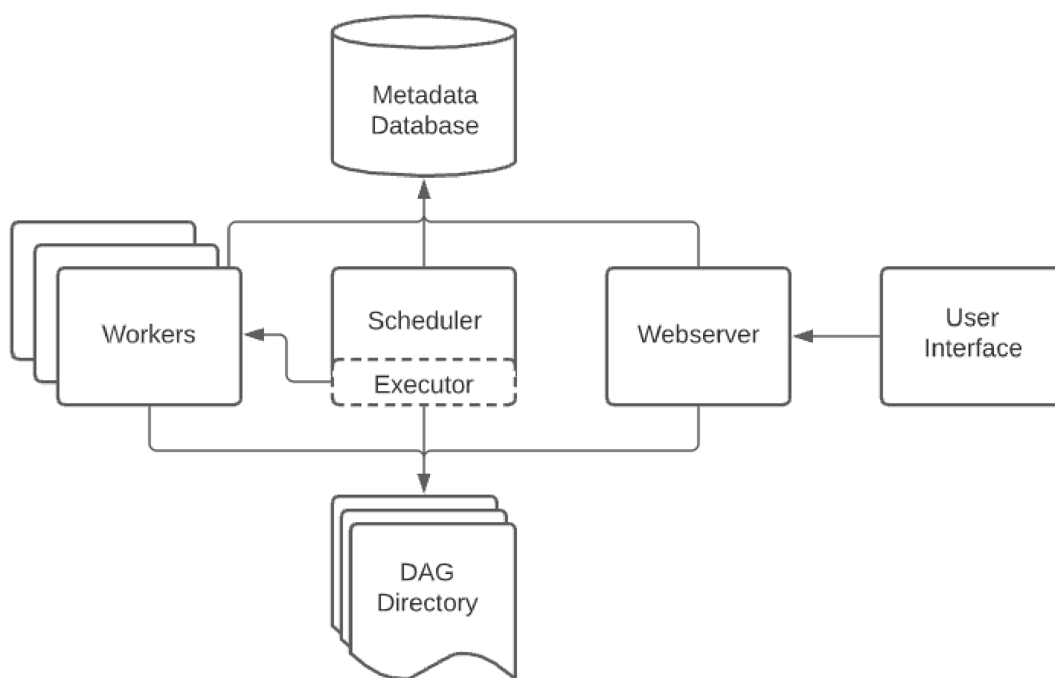

Figura 1. Ilustração da Arquitetura do *Apache Airflow*.
Fonte: Airflow documentation.

- *DAG Directory*: é o diretório onde são armazenados os arquivos *Python* que contém as definições dos DAGs.
- *Scheduler*: é responsável por ler o diretório de DAGs e agendar a execução das tarefas. Ele se comunica com o banco de metadados para determinar o estado dos pipelines, e com o executor que irá inicializar as tarefas programadas. O *scheduler* é escalável horizontalmente e pode ser executado em *cluster* para alta disponibilidade e balanceamento de carga.
- *Workers*: são as máquinas responsáveis por executar as tarefas. Eles podem ser configurados para serem executados em um ou mais nós e são responsáveis por buscar as tarefas da fila de tarefas (*task queue*), executá-las e retornar os resultados. Cada *worker* pode executar várias tarefas simultaneamente, dependendo do número de *slots* disponíveis e do nível de concorrência configurado.
- *Executor*: é responsável por executar os *workers* e dividir as tarefas entre eles, utilizando uma fila para receber as tarefas agendadas pelo servidor e as executa em paralelo. O *Airflow* suporta diferentes tipos de executor, como *LocalExecutor*

que executa tarefas no mesmo host do servidor e o *CeleryExecutor* que executa tarefas em um *cluster* de *workers Celery*.
- *Banco de Metadados*: é o componente responsável por armazenar informações relacionadas aos fluxos de trabalho, como tarefas, conexões de banco de dados, variáveis e outras configurações necessárias. Os metadados são armazenados em um banco de dados relacional.
- *Web Server*: é o componente responsável por prover uma interface gráfica ao usuário, onde é possível visualizar e gerenciar tarefas, *DAGs*, *logs*, conexões e variáveis, além de monitorar o estado do fluxo de trabalho.

Além desses componentes, o Airflow também possui recursos adicionais, como os *plugins* que são extensões para adicionar funcionalidades extras e *hooks* para se comunicar com diferentes tipos de sistemas externos. A arquitetura é altamente configurável e escalável, permitindo que os usuários personalizem e adaptem a plataforma para atender às suas necessidades específicas.

3.3. Data Lake e a Arquitetura de Medalhas

Data lake é uma estrutura de armazenamento flexível e escalável que permite a ingestão de grandes volumes de dados de diferentes formatos e tipos, não requerendo um esquema pré-definido diferentemente de sistemas de armazenamento tradicionais, como bancos de dados relacionais [Nargesian et al. 2019]. Um *data lake* permite a ingestão de dados em sua forma bruta, preservando a sua integridade original, oferecendo uma abordagem mais adaptável para a organização e análise de dados e permitindo a exploração de informações de várias fontes e estruturas, o que pode resultar em *insights* mais direcionados e significativos, conforme cita [Fang 2015], essa capacidade de lidar com dados não estruturados, semi-estruturados e estruturados, torna-o uma solução poderosa para o processamento e análise em larga escala.

Um conceito de organização utilizado em modelos de *data lake* para armazenar os dados em diferentes níveis de camadas, é a arquitetura de medalhas. Ela é útil para classificar os dados de acordo com sua qualidade e confiabilidade, permitindo a escolha da melhor base de dados para atender as necessidades do analista [L'Esteve 2023]. Esta arquitetura é composta por três níveis:

- Na camada bronze, são armazenados os dados brutos sem qualquer processamento ou transformação, estes dados são considerados de baixa qualidade ou valor agregado, já que não foram submetidos a nenhum tipo de tratamento ou validação.
- Na camada prata, há uma organização e limpeza dos dados tornando-os mais legíveis e estruturados, mas ainda podem possuir margem de erro ou incerteza. Neste nível, os dados são mais confiáveis do que no nível bronze, mas ainda precisam ser utilizados com cautela.
- Na camada ouro, são armazenados os dados considerados de alta qualidade e confiabilidade, que já passaram por uma série de transformações, resultando em informações mais valiosas e prontas para serem utilizadas em análises de negócios e tomada de decisões estratégicas.

A Figura 2 é uma representação gráfica desta arquitetura.

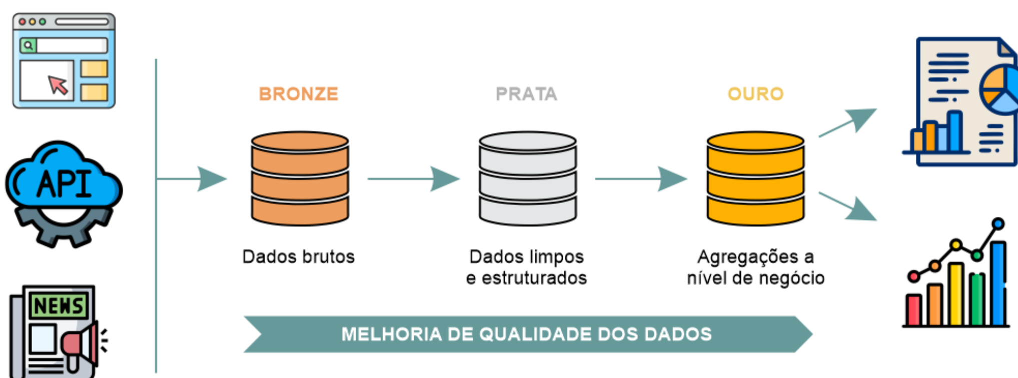

Figura 2. Arquitetura de medalhas.
Fonte: flaticon.com (adaptado).

3.4. Outras ferramentas

Nesta subseção, serão explorados os conceitos fundamentais das ferramentas que ampliaram as capacidades do *Apache Airflow*, contribuindo significativamente para a execução bem-sucedida dos *DAGs* projetados. As quatro ferramentas selecionadas - Selenium, Pandas, Localstack (com foco no S3) e Docker - desempenham papéis distintos e complementares, agregando automação, flexibilidade, e eficiência ao ambiente de execução.

Para auxiliar na extração de dados, foi utilizado o Selenium, uma ferramenta para a automação de interações com websites e a extração de dados de páginas da web [Selenium 2023]. No contexto do *Apache Airflow*, o Selenium é frequentemente usado para coletar dados de fontes web, como páginas de notícias, redes sociais ou plataformas de dados que não dispõem de recursos de extração sem a interação manual do usuário. Os conceitos fundamentais do Selenium incluem:

- Web Drivers: Drivers específicos projetados para realizar a comunicação com os navegadores (por exemplo, ChromeDriver e FirefoxDriver) automatizando interações em sites.
- Localização de Elementos: A capacidade de identificar e interagir com elementos HTML em uma página da web, como botões, campos de texto e links.
- Interações e Extração: Automatização de ações, como clicar em botões, preencher formulários e extrair informações de páginas da web.

Na etapa de transformação de dados, foi utilizada a biblioteca Pandas, criada por Wes McKinney em 2008, que é uma ferramenta poderosa e amplamente utilizada no campo da engenharia e análise de dados. Ela é construída sob a linguagem de programação *Python* e oferece estruturas de dados de alto desempenho e ferramentas de manipulação de dados flexíveis e eficazes [Pandas 2023]. Seus conceitos fundamentais incluem:

- DataFrame: Uma estrutura de dados bidimensional que organiza dados em colunas e linhas, semelhante a uma tabela de banco de dados ou planilha.

- Seleção e Indexação: A capacidade de selecionar e acessar dados específicos dentro de um DataFrame com base em critérios definidos.
- Transformação de Dados: A aplicação de funções, operações e métodos para alterar, limpar ou agregar dados.
- Manipulação de Valores Ausentes: Lidar com valores ausentes por meio de preenchimento, remoção ou imputação.

Para a persistência de informações sobre os dados extraídos foi utilizado o MySQL, um sistema popular de gerenciamento de banco de dados (SGBD) relacional e de código aberto. O MySQL é conhecido por sua confiabilidade, escalabilidade e eficiência e alguns dos seus conceitos fundamentais são:

- Tabelas e Esquemas: As tabelas são estruturas que armazenam dados de maneira organizada, e os esquemas definem a estrutura das tabelas.
- Consultas SQL: A linguagem SQL (*Structured Query Language*) é usada para realizar consultas, inserções, atualizações e exclusões de dados em um banco de dados MySQL.
- Chaves Primárias e Estrangeiras: São usadas para definir relacionamentos entre tabelas e garantir a integridade dos dados.
- Restrições de integridade: São regras aplicadas a um banco de dados relacional para garantir a qualidade, a consistência e a precisão dos dados armazenados.

Para realizar a simulação fidedigna de um armazenamento em nuvem, em um ambiente de desenvolvimento local foi utilizada a LocalStack. Uma ferramenta que permite simular serviços da nuvem AWS (*Amazon Web Services*) em ambientes locais, incluindo o Amazon S3, que é um serviço de armazenamento em nuvem amplamente utilizado. Os conceitos fundamentais da LocalStack (S3) incluem:

- *Buckets* e Objetos: O Amazon S3 organiza dados em "*buckets*" (contêineres) e "objetos" (arquivos).
- Simulação de Armazenamento: A capacidade de criar, modificar e acessar objetos em *buckets* locais, como se estivessem na nuvem.
- Testes em Ambiente Controlado: Permite a realização de testes de integração e desenvolvimento que envolvem o Amazon S3 sem a necessidade de acesso à AWS real, de forma ágil e sem custos.

Para execução do *Apache Airflow* e das demais ferramentas foi utilizado o Docker, uma plataforma de virtualização de contêiner utilizada para criar, implantar e executar aplicativos em ambientes isolados e consistentes. Para o contexo do projeto, o Docker foi empregado para criar contêineres que hospedam os componentes do Airflow e os serviços LocalStack, MySQL e Selenium. Os conceitos fundamentais do Docker incluem:

- Contêineres: Ambientes isolados que incluem todas as dependências e configurações necessárias para executar um aplicativo ou serviço.
- Imagens: Modelos para contêineres e incluem todos os recursos necessários, como bibliotecas, código e configurações.
- Orquestração: O Docker pode ser usado com ferramentas de orquestração, como Docker Compose e Kubernetes, para gerenciar e escalar contêineres.

4. Materiais e Métodos

Esta seção apresenta o desenvolvimentos das etapas e técnicas utilizadas para construção dos pipelines de dados que realizam os processos de coleta, transformação e armazenamento dos dados referentes a obras públicas do estado de Pernambuco, as etapas envolvem desde a criação da infraestrutura dos recursos até a montagem dos pipelines de dados.

4.1. Definição de Requisitos

Após a definição do contexto desse projeto, a etapa subsequente consistiu em identificar fontes de informação que atendessem às necessidades específicas para a análise de obras públicas de Pernambuco. Por meio de pesquisa exploratória foram encontradas duas fontes que contêm dados acerca de obras realizadas no estado: a API de Dados Abertos do Tribunal de Contas de Pernambuco[1] e o Painel de Obras Públicas do Ministério de Gestão e Inovação em Serviços Públicos do Governo Federal[2].

Tribunal de Contas
ESTADO DE PERNAMBUCO
API de Dados Abertos

Receitas

Método	Descrição
ReceitasEstaduais	Relação das Receitas Estaduais
ReceitasMunicipais	Relação das Receitas Municipais
ReceitasPrevistas	Relação das Receitas Previstas (apenas Municipais)

Despesas

Método	Descrição
DespesasEstaduais	Relação das Despesas Estaduais
DespesasMunicipais	Relação das Despesas Municipais - Chave composta (ID_EMPENHO, ANOREFERENCIA, NUMERO_EMPENHO e ID_UNIDADE_GESTORA)
EmpenhoLiquidacao	Relação das Liquidações dos Empenhos Municipais e Estaduais
EmpenhoPagamento	Relação dos Pagamentos dos Empenhos (apenas Municipais)
EmpenhoResumo	Relação dos Valores Originais, Reforços e Estornos dos Empenhos Municipais e Estaduais
TipoCredorMunicipal	Lista dos tipos de credores Municipal
TipoCredorEstadual	Lista dos tipos de credores Estadual
ItemEmpenhoEstadual	Relaçao de cada parte que compõem os empenhos Estaduais
ComparativoPrecoEstado	Itens dos empenhos estaduais com valores para comparação de preços
TransferenciasConcedidasMunicipais	Relação das transferências concedidas entre unidades juirisdicionadas do mesmo municipio

Fornecedores

Figura 3. Ilustração da documentação da API de Dados Abertos do TCE-PE.
Fonte: Tribunal de Contas do Estado de Pernambuco.

A API de Dados Abertos disponibilizada pelo Tribunal de Contas do Estado de Pernambuco (TCE-PE) é uma plataforma virtual que viabiliza a provisão de informações como um serviço, mediante o uso de URLs que possibilitam a realização de consultas mediante o envio de parâmetros. Estes procedimentos possibilitam à obtenção dos dados nos formatos XML, JSON e HTML. Os conjuntos de dados disponibilizados são de livre utilização tanto pela população em geral como pelos órgãos governamentais, no intuito de alavancar a concepção de aplicações, a condução de investigações automatizadas e outras formas tecnológicas que concorram para o fomento da vigilância cívica.

A Figura 4 apresenta a tela inicial da segunda fonte utilizada, o Painel de Obras, uma ferramenta que provê informações concernentes ao progresso físico e financeiro das obras executadas na Plataforma +Brasil, bem como aquelas decorrentes do Programa instituído pelo Decreto nº 6.025, datado de 22 de janeiro de 2007.

[1]Disponível em https://sistemas.tce.pe.gov.br/DadosAbertos
[2]Disponível em https://obras.paineis.gov.br/extensions/painel-obras/painel-obras.html

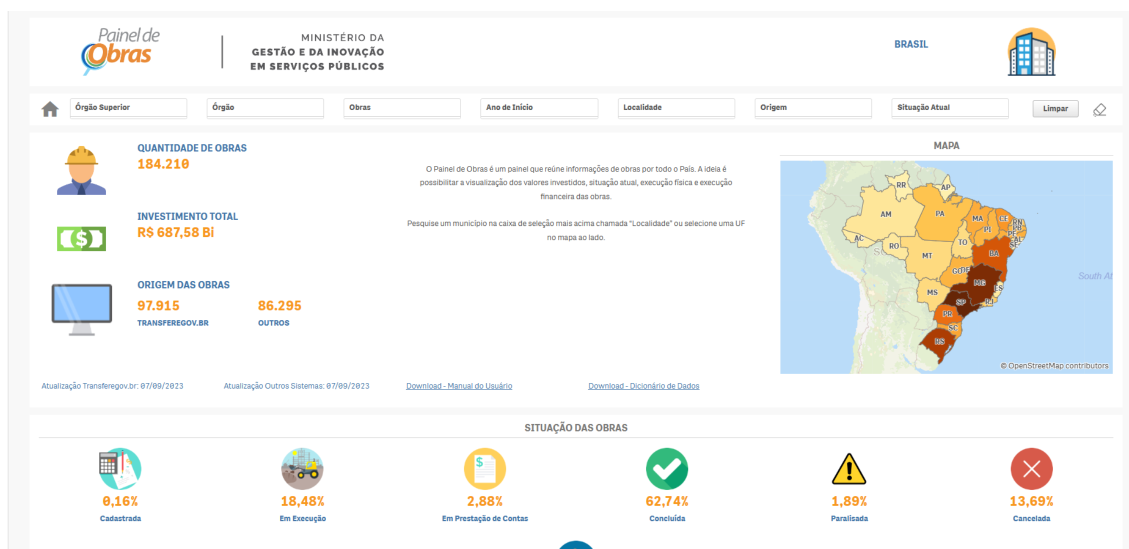

Figura 4. Página inicial do Painel de Obras.
Fonte: Painel de Obras.

O uso desta solução pelos órgãos públicos é definido pelo teor do Decreto nº 10.012/2019, o qual preceitua que as entidades e instituições detentoras de empreendimentos advindos do mencionado programa encaminhem os dados ao Ministério da Economia, com o intuito de conferir publicidade à execução desses recursos, centralizar a disponibilização destes dados e fomentar a transparência [DOU 2019].

4.2. Configuração do *Apache Airflow* e Serviços Adicionais

A instalação do *Apache Airflow* pode ser realizada de diversas maneiras. A abordagem escolhida para o projeto foi utilizando a stack Docker, para facilitar a portabilidade e o isolamento de recursos. O Airflow possui imagens Docker oficiais publicadas no portal DockerHub que podem ser estendidas e customizadas pelo usuário.

Como pré-requisito, é necessário que o usuário possua o Docker e o Docker Compose instalados em sua estação de trabalho para fazer o uso do arquivo `docker-compose.yml` que é disponibilizado na documentação oficial do Airflow. Esse arquivo define os contêineres criados e executados a partir das imagens dos serviços.

Analisando o código do `docker-compose.yml` oficial da versão 2.5.1 do *Apache Airflow*, nota-se que inicialmente é configurado uma seção intitulada `x-airflow-common`, que possibilita a reutilização de código em diversas partes posteriores do arquivo. Dentro desta seção, encontra-se a base de configuração comum para todos os serviços do *Apache Airflow*, listados abaixo.

- `&airflow-common`: Âncora que permite o reuso de código via notação `*airflow-common`;
- `image`: Definição da imagem Docker utilizada para a execução do *Apache Airflow*. A variável de ambiente `AIRFLOW_IMAGE_NAME` é usada para a flexibilização, possibilitando que os usuários alterem a imagem. Caso não seja fornecida, a imagem padrão será `apache/airflow:2.5.1`;

- `environment`: Um conjunto de variáveis de ambiente estabelecidas para a configuração do *Apache Airflow*. Estão inclusas configurações como o executor, a conexão do banco de dados, chaves de criptografia, entre outras.
- `volumes`: Configuração do mapeamento de volumes, permitindo que diretórios do host sejam vinculados a diretórios dentro dos contêineres do *Apache Airflow*. Isso viabiliza a persistência de dados críticos, como *DAGs*, logs, operadores e scripts. A variável `AIRFLOW_PROJ_DIR` é empregada para indicar o caminho base do projeto, que por padrão será a localização atual do arquivo. Os caminhos específicos dentro do container são definidos como `/opt/airflow/dags`, `/opt/airflow/logs`, `/opt/airflow/operators` e `/opt/airflow/scripts`.
- `depends_on`: Delimita as dependências entre serviços. No contexto do *Apache Airflow*, é especificado que o serviço webserver do Airflow depende dos serviços redis e postgres. A condição service_healthy é aplicada para assegurar que os serviços dependentes estejam saudáveis antes de prosseguir com a inicialização.

```
version: '3'
x-airflow-common:
  &airflow-common
  image: ${AIRFLOW_IMAGE_NAME:-apache/airflow:2.5.1}
  environment:
    &airflow-common-env
    AIRFLOW__CORE__EXECUTOR: CeleryExecutor
    # Outras variaveis omitidas...
  volumes:
    - ${AIRFLOW_PROJ_DIR:-.}/dags:/opt/airflow/dags
    - ${AIRFLOW_PROJ_DIR:-.}/logs:/opt/airflow/logs
    - ${AIRFLOW_PROJ_DIR:-.}/plugins:/opt/airflow/plugins
  user: "${AIRFLOW_UID:-50000}:0"
  depends_on:
    &airflow-common-depends-on
    redis:
      condition: service_healthy
    postgres:
      condition: service_healthy
...
```

Quadro 1. Seção do docker-compose.yml oficial do *Apache Airflow*

A definição dos contêineres é realizada na próxima seção do arquivo, que por padrão contém os seguintes serviços descritos abaixo.

- `postgres`: Oferece um banco de dados baseado no sistema PostgreSQL, para armazenar e gerenciar os dados pertinentes ao *Apache Airflow*.
- `redis`: trata-se de um serviço baseado no sistema de armazenamento em memória chamado Redis. Sua funcionalidade primordial reside na facilitação da comunicação eficiente entre os diversos serviços do *Apache Airflow*, para adotar uma abordagem de troca de mensagens em tempo real, propiciando uma interação dinâmica e reativa entre os elementos do ecossistema.
- `airflow-webserver`: Representa uma entidade de interface web, a qual viabiliza a interação dos usuários com o ambiente do *Apache Airflow*.
- `airflow-scheduler`: Serviço dedicado à tarefa de programação e orquestração dos pipelines de fluxo de trabalho.

- `airflow-worker`: Opera como uma "mão de obra" distribuída, responsável por executar efetivamente as tarefas definidas nas *DAGs*.
- `airflow-triggerer`: Atua como um elemento desencadeador, provocando o início das tarefas e fluxos de trabalho predefinidos de acordo com eventos ou condições específicas, permitindo a automatização e a reatividade nos processos.
- `airflow-init`: Responsável por iniciar o ambiente do *Apache Airflow* e, adicionalmente, criar o banco de dados e realizar as configurações iniciais necessárias para a execução da ferramenta.
- `airflow-cli`: Representa uma entidade que disponibiliza uma interface de linha de comando para interação com o sistema *Apache Airflow*. A funcionalidade fornecida pelo serviço de linha de comando permite a administração ágil e flexível do ambiente do Airflow, concedendo aos usuários a capacidade de controlar e monitorar as operações por meio de comandos específicos.
- `flower`: Oferece uma interface web dedicada ao monitoramento das tarefas e operações do sistema *Apache Airflow*. Por meio dessa interface visual, conhecida como Flower, os usuários podem supervisionar o estado e o progresso dos trabalhos, observar os detalhes das tarefas em execução e revisar informações sobre as *DAGs* e seus componentes.

Estendendo o conteúdo original do arquivo foram definidos mais serviços responsáveis pela execução das ferramentas auxiliares usadas no suporte dos processos de extração, persistência, e armazenamento de arquivos.

- `chrome-selenium`: Um serviço criado a partir da imagem selenium/standalone-chrome, que oferece um ambiente isolado e pré-configurado com o selenium server, o webdriver do Chrome e o próprio navegador.
- `mysql`: um serviço que inicializa uma instância para execução do banco de dados MySQL, para registro do histórico de arquivos extraídos.
- `localstack`: Um serviço que inicializa a LocalStack, para simular serviços o uso do AWS S3 em um ambiente local.

Cada serviço desempenha um papel essencial na configuração, administração e execução bem-sucedida do sistema, colaborando de maneira sinérgica para criar um ambiente robusto de orquestração de fluxos de trabalho.

Para iniciar os contêineres é utilizado o comando `docker-compose up -d`. A durante a primeira execução dos serviços, as pastas listadas abaixo são adicionadas no diretório raiz do projeto.

- `/dags`: Diretório onde o usuário adiciona os scripts de definição dos *DAGs*;
- `/plugins`: Diretório onde o usuário pode adicionar extensões de recursos;
- `/logs`: Diretório onde são armazenados os registros de atividade das ações realizadas pelo *Apache Airflow*.

Após o término da inicialização dos serviços, é possível acessar a interface gráfica do *Apache Airflow* através da porta que foi definida no serviço `airflow-webserver`, por padrão 8080. A Figura 5 ilustra a tela inicial da ferramenta.

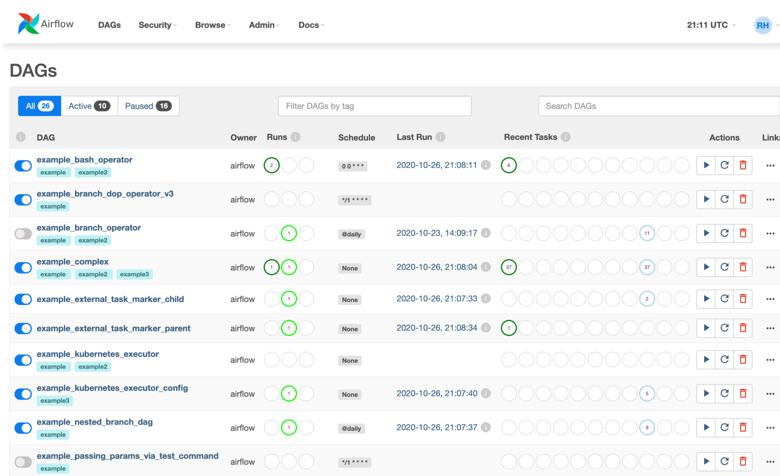

Figura 5. Página de *DAGs* da interface gráfica do *Apache Airflow*.
Fonte: O autor.

4.2.1. Criação do Datalake

O datalake deve ser um repositório centralizado que permita o armazenamento e gerenciamento de grandes volumnes de dados. O Amazon S3 é um serviço popular para a criação de datalakes devido a sua capacidade de escala e sua durabilidade.

Para realizar a emulação de um *bucket* S3 no ambiente de desenvolvimento local, é necessário que o serviço da LocalStack esteja em execução e que o usuário possua a ferramenta de linha de comando da AWS (AWS CLI). A criação do *bucket* que servirá como o repositório para o datalake, é realizada pelo comando apresentado no Quadro 2.

```
aws --endpoint-url=http://localhost:4566 s3api create-bucket \
--bucket datalake-obras-pernambuco \
--region us-east-1 \
--acl public-read
```

Quadro 2. Comando para criaçao do *bucket* s3 na LocalStack

- O parâmetro `endpoint-url` define o endpoint que o AWS CLI deverá utilizar para fazer a execução da ação, neste caso referenciando o endereço da instância da LocalStack em vez da AWS real;
- O `s3api create-bucket` instrui o AWS CLI a criar um novo *bucket* no Amazon S3;
- O parâmetro `bucket` especifica o nome do *bucket* que será criado, que neste caso é o "atalake-obras-pernambuco";
- O parâmetro `region` define a região geográfica fictícia do *bucket* como `us-east-1`.
- O parâmetro `acl` define a política de controle de acesso (ACL) do *bucket* como `public-read`, tornando-o público para leitura.

A verificação de criação do *bucket* pode ser feita através do próprio AWS CLI através do comando `aws -endpoint-url=http://localhost:4566 s3 ls` ou ainda utilizando a interface gráfica do LocalStack disponível em

app.localstack.cloud como ilustra a Figura 6. Os diretórios serão definidos a partir do momento que os arquivos forem enviados ao *bucket*.

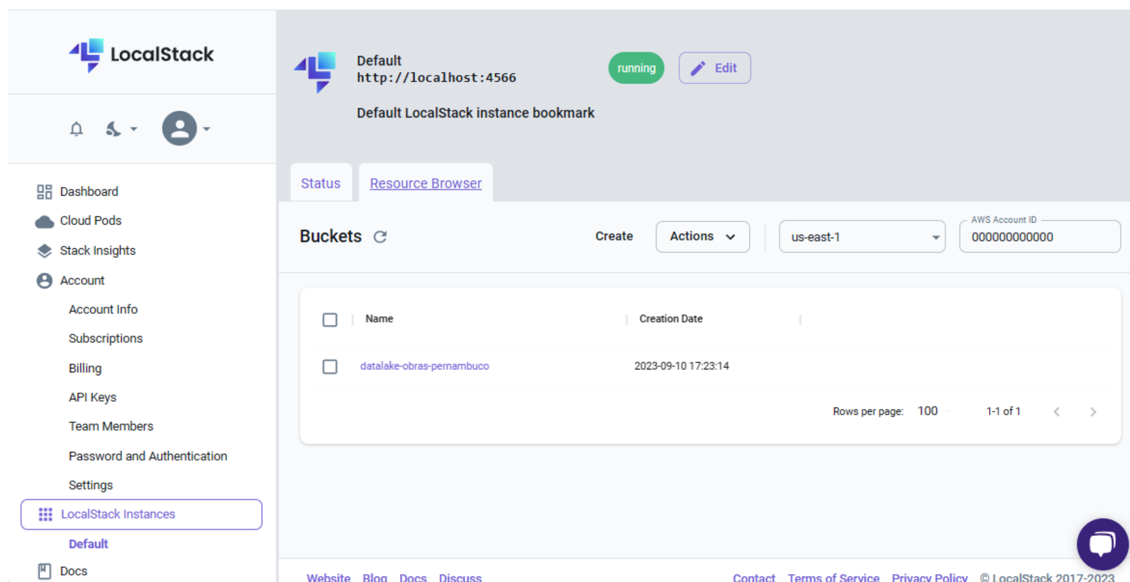

Figura 6. Ilustração do dashboard da LocalStack, com o *bucket* criado.
Fonte: O autor.

4.3. Definição dos *DAGs*

Os *DAGs* representam visualmente a ordenação sequencial das tarefas e seus inter-relacionamentos, sendo responsáveis pela definição da orquestração sistemática e do agendamento das atividades inerentes aos processos ETL. Essa seção apresenta a base da criação dos *DAGs* que serão criados para esse estudo.

Os *DAGs* devem ser escritos em arquivos *Python*. O diretório comum para armazenamento dos *DAGs* é geralmente dentro do diretório de configuração do Airflow, mas o caminho completo pode variar dependendo da instalação e configurações específicas. Geralmente este diretório é semelhante a /path/to/airflow/dags/. O usuário também pode configurar o diretório de *DAGs* editando o arquivo "airflow.cfg" e alterando a configuração dags_folder para o caminho desejado.

Após a criação do arquivo *Python* dentro do diretório de *DAGs*, foram definidos os caminhos utilizados para cada camada da arquitetura do *datalake* durante a execução dos *DAGs* com as constantes apresentadas no Quadro 3.

```
AIRFLOW_HOME = os.getenv('AIRFLOW_HOME')
DATA_ATUAL = "{{ ds }}"

CAMINHO_DATALAKE = f'{AIRFLOW_HOME}/datalake/tcepe'
CAMADA_BRONZE = f'{CAMINHO_DATALAKE}/bronze/date={DATA_ATUAL}'
CAMADA_PRATA = f'{CAMINHO_DATALAKE}/silver/date={DATA_ATUAL}'
CAMADA_OURO = f'{CAMINHO_DATALAKE}/gold/date={DATA_ATUAL}'
```

Quadro 3. Definição de constantes comuns aos pipelines gerados

- AIRFLOW_HOME: É a constante utilizada para recuperar o diretório raiz do Airflow através da variável de ambiente de mesmo nome;

- `DATA_ATUAL`: É uma constante que recebe o valor `{{ ds }}`, que no contexto do *Apache Airflow*, refere-se a uma variável Jinja que representa a data de execução do *DAG* atual. Jinja é um mecanismo de template de código aberto para *Python*, utilizado para permitir a definição dinâmica de valores;
- `CAMINHO_DATALAKE`, É a constante definida para salvar o diretório raiz temporário dos arquivos manipulados e posteriormente enviados para o datalake;
- `CAMADA_BRONZE`: É a constante definida para salvar o diretório onde ficarão os arquivos brutos da extração proveniente das fontes;
- `CAMADA_PRATA`: É a constante definida para salvar o diretório onde ficarão os arquivos convertidos após a fase de extração;
- `CAMADA_OURO`: É a constante definida para salvar o diretório onde ficarão os arquivos processados e prontos para uso em análises.

Em seguida foi criado um dicionário de parâmetros (apresentado no Quadro 4) que definem configurações padrões para os DAGs. Essas configurações são utilizadas na definição de cada *DAG*, no bloco de contexto `with DAG(...)`.

```
default_args = {
    'owner': 'Henrique Cesar',
    'depends_on_past': False,
    'retries': 0,
    'start_date': datetime(2023, 8, 20),
    'schedule_interval': '0 0 1 * *',
}

with DAG('tcepe_obras', default_args=default_args) as dag:
    ...
```

Quadro 4. Dicionário de parâmetros e aplicação ao *DAG*.

- `owner`: Define o nome do proprietário do *DAG*, sendo útil para rastrear quem é responsável por aquele fluxo específico;
- `depends_on_past`: Define se as tarefas do *DAG* dependem do sucesso da execução da tarefa anterior. Se definido como `True`, a próxima tarefa só será executada se a anterior for concluída com sucesso;
- `retries`: Define o número de tentativas que o *Airflow* fará para executar novamente uma tarefa se ela falhar na execução;
- `start_date`: Define a data e hora a partir da qual o *DAG* começará a ser executado. No exemplo, está definido como `datetime(2023, 8, 20)`, indicando que o *DAG* começará a ser agendado a partir de 20 de agosto de 2023;
- `schedule_interval`: Define uma expressão em *cron job* com a frequência com que o DAG será agendado. No exemplo, está definido como "`0 0 1 * *`", o que significa que o *DAG* será executado automaticamente todo dia 1 de cada mês.

Dentro do bloco de definição de um *DAG*, existe a flexibilidade de definir tanto operadores individuais quanto blocos de *TaskGroup*, como demonstrado no Quadro 5.

Os operadores no *Apache Airflow* são classes que definem ações ou tarefas individuais que compõem um *DAG*. Eles representam unidades discretas de trabalho e são usados para definir o que deve ser feito em cada etapa de um fluxo de trabalho. Os operadores utilizados neste projeto foram:

- O *EmptyOperator* que é um operador simples e sem ação efetiva além de marcar um ponto de verificação em um *DAG*.
- O *BashOperator* que permite executar comandos Bash dentro do *DAG*. Como entrada o usuário especifica o comando que será executado quando essa tarefa for acionada. No contexto desse projeto, este operador foi utilizado para criação de diretórios temporários durante a execução dos *DAGs*.
- O *PythonOperator* que permite executar uma função *Python* personalizada como parte do *DAG*. Como entrada o usuário define a função que será executada quando a tarefa é acionada.
- O *BranchPythonOperator* que é usado para adicionar ramificação condicional em um *DAG*. Ele executa uma função *Python* que retorna o ID da tarefa a ser executada em seguida, com base em alguma lógica condicional. Dependendo da saída da função *Python*, o fluxo do *DAG* pode ser direcionado para diferentes tarefas.
- Um operador personalizado chamado *TcePeOperator*, criado com base no *HttpOperator* e utilizado para fazer a conexão do Airflow com o serviço de API de Dados Abertos do TCE/PE.

Os blocos de *TaskGroup* permitem agrupar várias tarefas relacionadas em uma unidade lógica. Isso facilita a visualização e organização do fluxo de trabalho. Dentro deste bloco é possível definir operadores individuais e outros blocos de grupo, criando uma hierarquia de tarefas.

```
with DAG('tce_pe_obras', default_args=default_args) as dag:
    cria_diretorios = BashOperator(
        task_id='cria_diretorios',
        bash_command=f"mkdir -p {CAMADA_BRONZE} {CAMADA_PRATA} {CAMADA_OURO}",
    )

    with TaskGroup('extracao', tooltip="Tarefas de extracao") as extracao:
        extrai_obras = TcePeOperator(
            task_id="extrai_obras",
            endpoint_id="obras",
            extract_folder=CAMADA_BRONZE,
        )
        extrai_dados_contratuais = TcePeOperator(
            task_id="extrai_dados_contratuais",
            endpoint_id="obras_dados_contratuais",
            extract_folder=CAMADA_BRONZE,
        )
        extrai_dados_auditoria = TcePeOperator(
            task_id="extrai_dados_auditoria",
            endpoint_id="obras_dados_auditoria",
            extract_folder=CAMADA_BRONZE,
        )

    cria_diretorios >> extracao
```

Quadro 5. DAG composto por operador (cria_diretorios) e TaskGroup (extracao).

A ordem de execução das tarefas em *DAGs* é definida explicitamente por meio da definição das dependências, através do operador *double shift* (») do *Python*, onde a tarefa inserida à direita do operador será dependente da execução da esquerda.

4.4. Extração dos Dados

Esta seção apresenta os componentes criados para a etapa de extração dos dados. Para realização das atividades de extração na API de Dados Abertos foi realizado o desen-

volvimento de um hook e um operador personalizados que habilitaram a obtenção das informações de forma fluida e estruturada. Para o Painel de Obras que carece de um método de extração por meio de URL, foi criado um script na linguagem *Python* com o propósito de baixar o relatório de obras disponibilizado através de uma planilha que pode ser exportada após interações com o portal e o preenchimento automatizado de filtros.

4.4.1. Extração da API de Dados Abertos do TCE/PE

Para tornar possível a comunicação com a API de Dados Abertos de Pernambuco, o primeiro passo foi a criação de uma conexão, que no AirFlow são configurações para permitir o acesso seguro a recursos externos, como bancos de dados, APIs e outros serviços web. As conexões podem incluir informações de autenticação, como credenciais de usuário, chaves de API e tokens de autenticação.

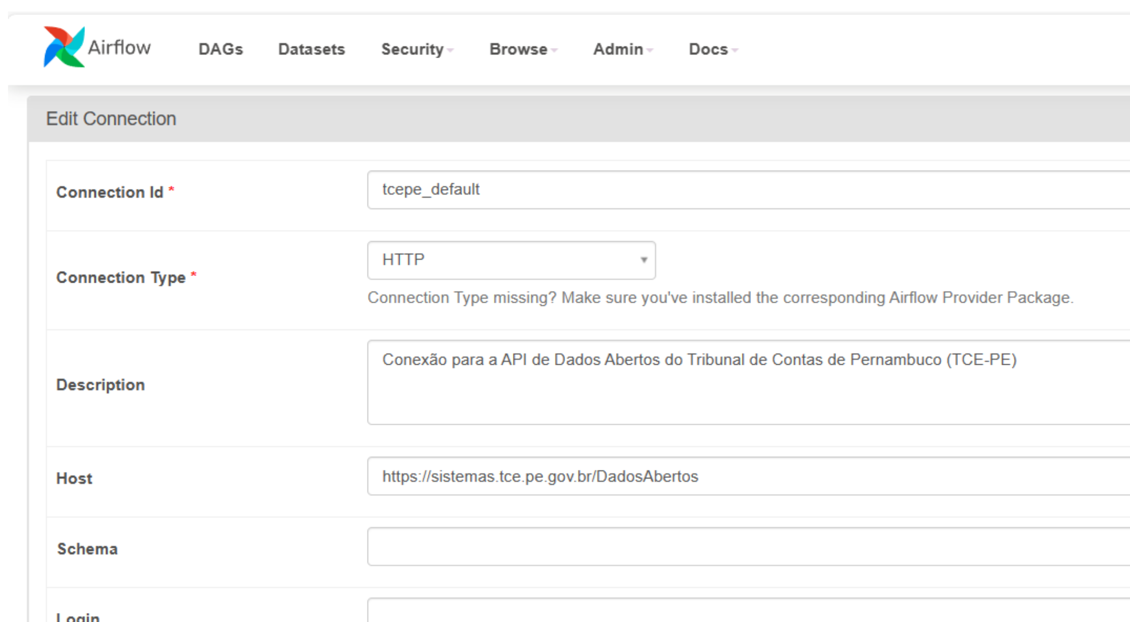

Figura 7. Definição da conexão criada no painel do Airflow.
Fonte: O autor.

Ao definir uma conexão é possível utilizar essas informações sem a necessidade de armazená-las diretamente no código. Atualmente a API do TCE-PE não necessita de meios de autenticação, porém é considerada uma boa prática usar o recurso de conexão do Airflow, pois ela garante que as integrações sejam gerenciadas de forma centralizada evitando exposição acidental de informações confidenciais nos códigos do fluxo de trabalho. Se futuramente for necessário adicionar credenciais de acesso para esses recursos ou configuração para ambientes de desenvolvimento diferentes, será possível atualizar as conexões sem necessidade de mudança do código. A Figura 7 ilustra a conexão do tipo HTTP criada através da página de conexões, no *Airflow UI*.

A interação com a conexão ocorre através dos *hooks*, que são componentes fundamentais para encapsulamento da lógica de conexão e interação com sistemas externos, como bancos de dados, serviços em nuvem, sistemas de mensagens e outras fontes de

dados. Eles são elementos chave da arquitetura modular do *Airflow* e desempenham um papel crucial na facilitação da comunicação e integração entre os fluxos de trabalho orquestrados pelo *Airflow* e os sistemas externos com os quais eles interagem [Feng 2020].

Os *hooks* servem como uma camada de abstração que separa a complexidade das interações externas da lógica do próprio fluxo de trabalho. Ao fornecer métodos e interfaces padronizados para acessar e manipular dados e serviços externos, os *hooks* eliminam a necessidade de escrever código repetitivo e específico para cada sistema externo. Resultando em uma estrutura mais modular e reutilizável para o desenvolvimento de *DAGs*.

```python
import json
import requests
from airflow.providers.http.hooks.http import HttpHook

class TcePeHook(HttpHook):

    def __init__(self, endpoint_id, conn_id=None):
        self.endpoint_id = endpoint_id
        self.conn_id = conn_id or "tcepe_default"
        super().__init__(http_conn_id=self.conn_id)

    def get_url(self, endpoint_id):
        suffix = None
        if endpoint_id == 'obras':
            suffix = "Obras!json"
        elif endpoint_id == 'obras_dados_contratuais':
            suffix = "ObrasDadosContratacao!json"
        elif endpoint_id == 'obras_dados_auditoria':
            suffix = "DadosObrasAuditoria!json"
        return f"{self.base_url}/{suffix}"

    def connect(self, url, session):
        request = requests.Request("GET", url)
        prep = session.prepare_request(request)
        return self.run_and_check(session, prep, {})

    def download(self, url, session):
        response = self.connect(url, session)
        return json.loads(response.content.decode("ISO-8859-1"))

    def run(self, **kwargs):
        session = self.get_conn()
        url_raw = self.get_url(self.endpoint_id)

        return self.download(url_raw, session)
```

Quadro 6. Código da classe TcePeHook.

A classe `TcePeHook`, apresentada no Quadro 6, é uma extensão de um *hook* do tipo HTTP e é projetada para facilitar a interação com a API de Dados Abertos do TCE/PE. Ela inclui métodos que realizam ações diferentes para se conectar ao serviço web, solicitar dados e manipular a resposta. O método construtor da classe `TcePeHook` aceita como parâmetros o `endpoint_id` que é utilizado para identificar o *endpoint* da API com o qual o usuário deseja interagir, e opcionalmente, o `conn_id` é um identificador opcional para indicar a conexão HTTP que deve ser utilizada pelo Apache Airflow.

O método `get_url` recebe o parâmetro `endpoint_id` e mapeia para uma URL completa do endpoint correspondente. A URL é composta pelo caminho base (`self.base_url`) e o sufixo indicado. É o método `connect` que realiza a conexão

HTTP com a URL fornecida usando a biblioteca requests. Ele cria uma solicitação do tipo GET usando o endereço e a sessão fornecida como argumentos. Em seguida, prepara a solicitação e a executa usando o método `run_and_check` herdado de `HttpHook`, retornando a resposta da solicitação. O método download utiliza-o para obter a resposta da solicitação fornecida. Em seguida, decodifica o conteúdo da resposta usando a codificação utilizada no conteúdo retornado pela API de Dados Abertos, ISO-8859-1, e converte os dados em formato JSON retornando-os como um objeto.

O método `run` é responsável por executar a lógica principal da interação com o serviço web. Ele obtém a conexão HTTP usando o método `get_conn`, a URL completa do endpoint usando o método `get_url` e, em seguida, invoca o método `download` para obter os dados do serviço web, retornando-os no final.

Para utilizar o *hook* personalizado dentro dos *DAGs*, é necessário a criação de um operador que encapsule as ações abstraindo operações realizadas.

```
import json
from os.path import join
from airflow.models import BaseOperator
from hooks.tcepe_hook import TcePeHook


class TcePeOperator(BaseOperator):

    template_fields = ['extract_folder']

    def __init__(self, endpoint_id, extract_folder, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.extract_folder = extract_folder
        self.endpoint_id = endpoint_id

    def execute(self, context):
        path_to_save = join(self.extract_folder, self.endpoint_id + '.json')
        with open(path_to_save, "w", encoding="utf-8") as output_file:
            json_data = TcePeHook(self.endpoint_id).run()
            output_file.write(json.dumps(json_data, ensure_ascii=False))
```

Quadro 7. Código da classe TcePeOperator.

O operador `TcePeOperator`, apresentado no Quadro 7, encapsula a lógica de interação com o serviço web e extração de dados para um arquivo JSON. Ele estende a funcionalidade da classe `BaseOperator` e utiliza um *hook* personalizado para facilitar a comunicação com o serviço web.

O método construtor da classe aceita três parâmetros obrigatórios: um identificador específico para o *endpoint* do serviço web (`endpoint_id`), o diretório onde os dados extraídos serão salvos (`extract_folder`); e argumentos que permitem a passagem de argumentos adicionais (`*args` e `**kwargs`). O construtor invoca a classe pai (`BaseOperator`) e atribui os valores dos parâmetros aos atributos do objeto.

O método `execute` é a parte central do operador, onde a lógica de execução do fluxo de trabalho é implementada. Ele é chamado quando o operador é executado em uma *DAG*. Dentro deste método, a variável `path_to_save` é criada para guardar o caminho completo ao arquivo onde os dados serão salvos. O bloco de contexto `with open(...)` abre o arquivo para escrita no caminho especificado, nesta etapa do processo o hook `TcePeHook` é instanciado com o `endpoint_id` fornecido, iniciando a

interação com o serviço de API do TCE-PE.

Após a implementação, o TcePeOperator está apto a ser utilizado em *DAGs*, o Quadro 8 exemplifica como o operador pode ser definido.

```
extrai_obras = TcePeOperator(
    task_id="extrai_obras",
    endpoint_id="obras",
    extract_folder=CAMADA_BRONZE,
)
```

Quadro 8. Código da função principal de extração com o Selenium.

4.4.2. Extração do Painel de Obras

Para extração dos dados de maneira automatizada a partir do portal do Painel de Obras, foi utilizado um `PythonOperator` que possibilita a criação de tarefas capazes de executar uma função *Python* em um DAG. Para o uso neste operador, foi desenvolvido um *script* que importa a biblioteca `selenium`, utilizando-a em conjunto com um servidor remoto que executa um agente de *webdriver* para interação com o navegador web.

O servidor foi inicializado em um contêiner Docker e sua configuração foi inserida no arquivo docker-compose do projeto conforme ilustra o Quadro 9.

```
chrome-selenium:
    image: selenium/standalone-chrome
    hostname: chrome
    shm_size: 2gb
    ports:
      - 4444:4444
    restart: always
    volumes:
      - ./datalake/tmp:/downloads
```

Quadro 9. Definição de um serviço que executa o webdriver do Selenium, no Docker Compose.

O *script Python* conduz o processo de extração utilizando a abordagem de web scrapping, por meio de uma série de ações programadas e predefinidas, visando a interação com elementos da página, o download e movimentação de arquivos para o diretório de destino. Sendo modularizado em três partes principais: configuração do agente de webdriver, busca na página web e download de arquivos, os parâmetros de entrada estão descritos a seguir, na Tabela 1.

Nome da Coluna	Tipo	Descrição
`webdriver`	`object`	Objeto de configurações para o agente do webdriver.
`webdriver.endpoint`	`str`	URL do agente do webdriver.
`webdriver.opcoes`	`str[]`	Lista de valores para configuração de opções do webdriver.

Continua na próxima página

Tabela 1 – Continuação da Página Anterior

Nome da Coluna	Tipo	Descrição
webdriver .opcoes_experimentais	object[]	Lista de valores para configuração de opções experimentais do webdriver.
pesquisa	object	Objeto de instruções para a pesquisa.
pesquisa.url	str	Página web onde será realizada a pesquisa.
pesquisa .espera_em_segundos	int	Tempo de espera em segundos para carregamento da página.
pesquisa .elementos_pagina	object	Lista de elementos da página para interação via webdriver.
pesquisa .elementos_pagina .valor	str	Valor para busca do elemento na página.
pesquisa .elementos_pagina .tipo_busca	str	Estratégia de busca do elemento na página.
pesquisa .elementos_pagina .acoes	object[]	Ações de interação com o elemento na página.
download	object	Objeto de instruções para o download.
download .extensao_esperada	str	Extensão do arquivo esperado no diretório de download.
download .espera_em_segundos	int	Tempo limite de espera (em segundos) para o término do download.
download .diretorio_download	str	Diretório final de download do arquivo.
download.mover_para	str	Lista de elementos da página para interação via webdriver.

Tabela 1. Parâmetros de entrada para o script de extração do Painel de Obras.

A função principal web_extrair_dados (Quadro 10) orquestra o fluxo de extração com base em um objeto de configuração (params) que contém os parâmetros necessários para realizar todo o processo de extração. O script é protegido por um bloco try...except, que captura e trata exceções que possam ocorrer durante o processo. O agente do *webdriver* é encerrado ao final, assegurando a liberação de recursos.

```
def web_extrair_dados(webdriver, pesquisa, download):
    try:
        wd = _inicializar_webdriver(webdriver)
        _executar_pesquisa(wd, pesquisa)

        if _aguardar_download(download):
            _mover_arquivo_baixado(download)
            logging.info('Arquivo salvo com sucesso.')
        else:
            logging.error('Erro: Download do arquivo nao foi concluido.')

                                        # continua na proxima pagina...
```

```
    except Exception as e:
        logging.error(f'Erro:␣{e}')

    finally:
        if 'wd' in locals():
            wd.quit()
```

Quadro 10. Código da função principal de extração com o Selenium.

A inicialização do agente do *webdriver* é realizada na função interna `_inicializar_webdriver` (Quadro 11) de acordo com os parâmetros fornecidos pelo dicionário de configuração (`webdriver`) que pode conter definições para o navegador, incluindo a localização do diretório padrão de downloads.

```
def _inicializar_webdriver(config_navegador):
    endpoint = config_navegador['endpoint']
    opcoes = config_navegador.get('opcoes', [])
    opcoes_experimentais = config_navegador.get('opcoes_experimentais', [])

    options = Options()
    for opcao in opcoes:
        options.add_argument(opcao)

    for opcao in opcoes_experimentais:
        for key, value in opcao.items():
            options.add_experimental_option(key, value)

    return webdriver.Remote(endpoint, options=options)
```

Quadro 11. Código da função auxiliar que iniciallliza o webdriver.

A busca na página é executada na função `_executar_pesquisa`. O agente do *webdriver* direciona o navegador para a URL da página informada e aguarda até que elementos estejam disponíveis. Em seguida, o *script* executa uma sequência de ações pré-configuradas em cada elemento, permitindo a interação com os componentes. As ações podem incluir cliques em elementos, envio de teclas e pausas para espera.

```
def _executar_pesquisa(webdriver, config_pesquisa):
    webdriver.implicitly_wait(config_pesquisa.get('implicity_wait', 0))
    webdriver.get(config_pesquisa.get('url'))

    for params_elemento in config_pesquisa.get('elementos_pagina', []):
        if 'atrasar_segundos' in params_elemento:
            time.sleep(params_elemento['atrasar_segundos'])

        element = webdriver.find_element(by=params_elemento['tipo_busca'],
            value=params_elemento['valor'])
        _interagir_elemento(element, params_elemento.get('acoes', []))

def _interagir_elemento(elemento, acoes):
    for acao in acoes:
        tipo_acao, valor = list(acao.items())[0]
        if tipo_acao == 'click':
            elemento.click()
        elif tipo_acao == 'send_keys':
            elemento.send_keys(valor)
        elif tipo_acao == 'sleep':
            time.sleep(valor)
        # Adicionar mais acoes se necessario
```

Quadro 12. Código das funções de consulta e interação com a página web.

As etapas de download e movimentação do arquivo com os dados extraídos são coordenadas pelas funções apresentadas no Quadro 13, `_aguardar_download` e `_mover_arquivo_baixado`, em conformidade com os parâmetros estabelecidos no dicionário `config_download`. A primeira função aguarda até que um arquivo com a extensão especificada pelo usuário seja identificado no diretório de downloads do navegador, e uma vez que ela confirma a conclusão do download, a segunda função assume a responsabilidade da transferência do arquivo para o diretório de destino também informado pelo usuário.

```
def _aguardar_download(config_download):
    diretorio_download = config_download.get('diretorio_download', '.')
    tempo_atual = 0
    tempo_maximo_espera = config_download.get('espera_em_segundos', 60)

    while tempo_atual <= tempo_maximo_espera:
        if any(file.endswith(config_download.get('extensao_esperada', '.xlsx'))
             for file in os.listdir(diretorio_download)):
            return True
        time.sleep(1)
        tempo_atual += 1

    return False


def _mover_arquivo_baixado(config_download):
    diretorio_download = config_download.get('caminho_destino', '.')
    downloaded_file = next(file for file in os.listdir(diretorio_download)
        if file.endswith(config_download.get('extensao_esperada', '.xlsx')))
    source = os.path.join(diretorio_download, downloaded_file)
    target = os.path.join(config_download.get('mover_para', ''), 'data.xlsx')
    os.replace(source, target)
```

Quadro 13. Código das funções de download e salvamento dos dados.

4.5. Transformação dos Dados

Esta seção descreve as etapas elaboradas para a fase de transformação dos dados, a qual abrange um conjunto de operações cruciais para a preparação e estruturação das informações brutas em formatos passíveis de análise e aplicação. Dentro deste contexto, contempla-se a criação de abordagens customizadas para a conversão dos arquivos brutos no formato JSON para o formato CSV, a unificação dos distintos arquivos resultantes e a subsequente limpeza dos dados, preparando-os para a etapa de carregamento.

4.5.1. Conversão dos Arquivos Brutos

Após a etapa de extração dos dados, que geraram arquivos nos formatos JSON (API de Dados Abertos) e XLSX (Painel de Obras), foi realizada a tarefa de conversão para o formato CSV que traz algumas vantagens de uso como processamento, padronização e armazenamento dos dados, visto que o CSV é um formato tabular, que organiza os registros em linhas e colunas, separados por um caractere especial (normalmente a vírgula), semelhante a uma planilha e ocupa menos espaço em disco do que os arquivos JSON, já que não precisa armazenar metadados associados a cada registro [Mitlöhner et al. 2016]. Além disso, esse processo reduz a dependência de softwares proprietários como o Microsoft Excel, visto que o CSV é um formato mais comumente aceito por diversos sistemas

de streamming e processamento de arquivos em lote, o que é especialmente relevante ao trabalhar com grandes volumes de dados [Belov and Nikulchev 2021].

Para realizar esta atividade, foi desenvolvido um *script Python* que faz a conversão de arquivos, de forma parametrizada. A função principal do *script*, apresentada no Quadro 14, é chamada `converter_para_csv` e recebe como argumento um dicionário chamado (`arquivos`), contendo as informações sobre os arquivos a serem convertidos. A função percorre cada item do dicionário, e para cada, é extraído o caminho atual do arquivo. Com base na extensão do arquivo (JSON ou XLSX) é chamada uma função auxiliar correspondente que tenta carregar um *dataframe* utilizando a biblioteca Pandas `pd`, se o arquivo não possuir uma extensão válida, é lançada uma exceção `ValueError`. Após a definição do *dataframe* é chamada a função auxiliar `_salvar_em_csv`.

```
def converter_para_csv(arquivos):
    for arquivo, params in arquivos.items():
        caminho_arquivo = os.path.join(params['caminho_base'], arquivo)
        try:
            if arquivo.endswith('.json'):
                df = _gerar_dataframe_de_json(caminho_arquivo, params)
            elif arquivo.endswith('.xlsx'):
                df = _gerar_dataframe_de_xlsx(caminho_arquivo, params)
            else:
                raise ValueError('O␣arquivo␣contem␣uma␣extensao␣invalida.')
            _salvar_em_csv(df, params)
        except Exception as e:
            logging.error(f'Erro␣ao␣converter␣{arquivo}␣em␣formato␣CSV.', e)
```

Quadro 14. Código da função principal para conversão de JSON para CSV.

As funções de geração de *dataframe* são personalizadas para tratar casos específicos de dados, como aplicar filtros a arquivos JSON ou selecionar tabelas em arquivos XLSX. Essa abordagem aumenta a modularidade e permite a reusabilidade de código.

A função auxiliar responsável por ler arquivos JSON e criar um *dataframe* do Pandas é a `_gerar_dataframe_do_json` (Quadro 15) que recebe como argumentos o caminho do arquivo (`arquivo_json`) e um dicionário de parâmetros (`params`). Se especificado um parâmetro de filtro contendo a indicação do caminho para um nó específico do JSON (ex.: `$.resposta.conteudo`), a função realiza o filtro dos dados a partir de um nível da estrutura, abrindo o conteúdo do arquivo e recuperando o valor do nó desejado. Esta funcionalidade é útil para casos onde os dados que precisam ser convertidos não estão na raiz do arquivo JSON. Caso o filtro não seja informado, a função apenas gera o *dataframe* com base no caminho onde o arquivo está localizado.

```
def _gerar_dataframe_de_json(arquivo_json, params):


    if params['filtro'] is not None:
        with open(arquivo_json, 'r', encoding=params.get('encoding', 'utf8'))
            as arquivo:
            conteudo = arquivo.read().replace('\n', '').replace('\r', '')
            conteudo_json = json.loads(conteudo)

        camadas = params['filtro'].lstrip('$.').split('.')
        for camada in camadas:


                                                # continua na proxima pagina...
```

```
        try:
            if camada.isdigit():
                camada = int(camada)
            conteudo_json = conteudo_json[camada]
        except Exception as e:
            raise ValueError('Erro na realizacao do filtro JSON.', e)

    return pd.read_json(conteudo_json)

return pd.read_json(arquivo_json)
```

Quadro 15. Função auxiliar que transforma arquivos JSON em DataFrame Pandas.

A função auxiliar responsável por ler arquivos XLSX e criar um *dataframe* do Pandas é a `_gerar_dataframe_do_xlsx` que recebe como argumentos o caminho do arquivo (`arquivo_xlsx`) e um dicionário de parâmetros (`params`). A função utiliza um parâmetro para identificação da tabela na planilha do arquivo e retorna o *dataframe*.

```
def _gerar_dataframe_de_xlsx(arquivo_xlsx, params):
    return pd.read_excel(arquivo_xlsx, sheet_name=params['nome_tabela'])
```

Quadro 16. Função auxiliar que transforma arquivos XLSX em DataFrame Pandas.

A função `_converter_para_csv` recebe o nome do arquivo (`arquivo_json`) e um dicionário de parâmetros (`params`), lê o conteúdo do arquivo JSON, e se necessário, aplica filtros por nível e altera o nome das colunas para o CSV. Em seguida, salva o conteúdo em um arquivo CSV após a conversão dos dados para um *DataFrame* da biblioteca Pandas.

```
def _converter_para_csv(arquivo_json, params):
    arquivo_entrada = os.path.join(params['caminho_base'], arquivo_json)
    arquivo_saida = os.path.join(params['caminho_destino'], params['salvar_como
        '])
    if not os.path.isfile(arquivo_entrada):
        raise FileNotFoundError(f'O arquivo "{arquivo_entrada}" nao foi 
            encontrado.')
    with open(arquivo_entrada, 'r', encoding='utf-8') as file:
        conteudo_json = file.read().replace('\r\n', '').replace('\n', '')
        data = json.loads(conteudo_json)
    if params['filtro_json'] is not None:
        data = _carregar_json_do_ponto(data, params['filtro_json'])
    df = pd.DataFrame(data)
    if params['renomear_colunas'] is not None:
        colunas_renomeadas = dict(params['renomear_colunas'].items())
        df.rename(columns=colunas_renomeadas, inplace=True, errors='ignore')
    df.to_csv(os.path.join(arquivo_saida), index=False)
```

Quadro 17. Código da função interna que realiza a conversão dos arquivos.

4.5.2. Unificação dos Arquivos

A unificação dos arquivos de obras, auditorias e dados contratuais obtidos da API de Dados do TCE-PE, promove a eliminação de dados duplicados, e converge os resultados obtidos para realizar análises mais consistentes.

O *script* abaixo é um programa em *Python* que tem como objetivo mesclar arquivos CSV com base em um conjunto de parâmetros configuráveis. A mesclagem é realizada por meio da combinação de *DataFrames* da biblioteca Pandas, que permite a manipulação e análise eficiente de dados. Essa etapa é constituída por duas funções.

```
def csv_unificar_arquivos(arquivos):
    for file, params in arquivos.items():
        arquivo_entrada = os.path.join(params['caminho_base'], file)
        arquivo_saida = os.path.join(params['caminho_destino'], params['
            salvar_como'])

        try:
            with open(arquivo_entrada) as conteudo_arquivo:
                df_esquerdo = pd.read_csv(conteudo_arquivo)

            df_unificado = _unificar_dataframes(df_esquerdo, params)

            with open(arquivo_saida, 'w', newline='') as conteudo_arquivo:
                df_unificado.to_csv(conteudo_arquivo, index=False)

            logging.info(f'Arquivo␣{file}␣mesclado␣e␣salvo␣em␣{arquivo_saida}')
        except FileNotFoundError:
            logging.error(f'Erro:␣Arquivo␣{file}␣nao␣encontrado.')
        except Exception as e:
            logging.error(f'Erro␣ao␣mesclar␣{file}:␣{e}')


def _unificar_dataframes(df_esquerdo, params):
    caminho_df_direito = os.path.join(params['caminho_base'], params['
        unificar_com'])
    df_direito = pd.read_csv(caminho_df_direito)

    right_on = params['parametros_unificacao']['right_on']
    left_on = params['parametros_unificacao']['left_on']
    how = params['parametros_unificacao']['how']

    df_unificado = pd.merge(df_esquerdo, df_direito, left_on=left_on, right_on=
        right_on, how=how)

    if left_on != right_on and how == 'left':
        df_unificado = df_unificado.drop(right_on, axis=1)

    if 'proximo' in params and params['proximo'] is not None:
        return _unificar_dataframes(df_unificado, params['proximo'])
    else:
        return df_unificado
```

Quadro 18. Código da função principal que orquestra a unificação de arquivos.

A função `csv_unificar_arquivos` é responsável pela coordenação do processo de mesclagem. Ela itera por um dicionário (`arquivos`), onde cada chave representa um arquivo a ser mesclado e os valores associados são os parâmetros de configuração para a mesclagem. O *script* lê os *dataFrames* e utiliza a função interna chamada `_unificar_dataframes` para realizar a operação. O resultado final é gravado em um novo arquivo CSV.

A função interna `_unificar_dataframes` recebe dois parâmetros, um *dataFrame* da biblioteca Pandas (`df_esquerdo`) e um dicionário de parâmetros (`params`). Com base nos parâmetros `left_on`, `right_on` e `how`, a função executa a mesclagem do *dataFrame* com outro obtido de um arquivo CSV (`df_direito`). Além disso, ela

manipula a remoção de colunas redundantes após a mesclagem, dependendo da estratégia utilizada. No fim, a recursão é empregada para permitir a mesclagem sequencial de *dataFrames* com base em configurações sucessivas.

4.5.3. Limpeza dos Dados

A etapa de limpeza dos dados, que sucede à unificação dos arquivos do processo de transformação dos dados, tem uma grande importância no ciclo de tratamento de informações. Compreendendo uma série de atividades, para a identificação, correção e eliminação de anomalias, irregularidades e imprecisões inerentes aos dados consolidados.

O *script* abaixo tem o propósito de aplicar transformações específicas a arquivos CSV com base em um conjunto de parâmetros configuráveis. As transformações são definidas como funções, permitindo que valores existentes em colunas dos arquivos sejam modificados de acordo com a lógica determinada pelas ações fornecidas.

```
def csv_aplicar_transformacoes(arquivos):
    for csv_file, params in arquivos.items():
        caminho_entrada = os.path.join(params['caminho_base'], csv_file)
        caminho_saida = os.path.join(params['caminho_destino'], params['
            salvar_como'])

        try:
            with open(caminho_entrada, 'r', encoding='utf-8') as
                arquivo_entrada, \
                    open(caminho_saida, 'w', encoding='utf-8', newline='') as
                        arquivo_saida:
                reader = csv.reader(arquivo_entrada)
                nomes_colunas = next(reader)
                indices_colunas = {nome: indice for indice, nome in enumerate(
                    nomes_colunas)}

                writer = csv.DictWriter(arquivo_saida, fieldnames=nomes_colunas
                    )
                writer.writeheader()

                for linha in reader:
                    linha_dict = {nome: linha[indice] for nome, indice in
                        indices_colunas.items()}
                    linha_transformada = {nome: params['funcoes_transformacao'
                        ].get(nome, lambda x: x)(valor)
                                          for nome, valor in linha_dict.items()
                                              }
                    writer.writerow(linha_transformada)

                logging.info(f"Transformacoes aplicadas ao arquivo {csv_file}")
        except FileNotFoundError:
            logging.error(f"Erro: Arquivo {csv_file} nao encontrado.")
        except Exception as e:
            logging.error(f"Erro ao processar {csv_file}: {e}")
```

Quadro 19. Código da função que aplica transformações nos dados.

A execução do *script* inicia percorrendo um dicionário chamado (`arquivos`), onde cada chave representa um nome de arquivo CSV e os valores associados são os parâmetros correspondentes. Para cada arquivo presente no dicionário, o *script* constrói os caminhos completos de entrada e saída dos arquivos com base nos parâmetros de caminho base (`caminho_base`), diretório de destino (`caminho_destino`) e nome de

salvamento (salvar_como).

Dentro de um bloco de tratamento de erros, o *script* abre o arquivo de entrada (caminho_entrada) para leitura e o arquivo de saída (caminho_saida) para escrita, utilizando a biblioteca csv para criar objetos *reader* e *writer*. O objeto *reader* é responsável por ler o conteúdo do arquivo de entrada, enquanto o objeto *writer* é utilizado para escrever o conteúdo transformado no arquivo de saída. A primeira linha do arquivo (nomes_colunas) é lida pelo *reader*, um mapeamento de índices para nomes de colunas é criado, e o *writer* inicialmente escreve o cabeçalho no arquivo de saída. Logo após o *script* percorre cada linha do arquivo de entrada, transforma os valores das colunas com base nas funções de transformação definidas nos parâmetros (funcoes_transformacao) e escreve a linha transformada no arquivo de saída. As funções de transformação podem ser definidas e aplicadas a cada coluna específica através de funções lambda. Desta forma, esse *script* oferece uma maneira automatizada e configurável de aplicar transformações a arquivos CSV, permitindo a modificação controlada de valores em colunas específicas presentes nos arquivos, adequando-os a necessidades específicas.

O script registra mensagens de log informando que as transformações foram aplicadas ao arquivo, caso a execução tenha sido bem-sucedida. Em caso de exceções, como o arquivo não ser encontrado (FileNotFoundError) ou qualquer outro erro (Exception) durante o processamento, mensagens de *log* de erro são registradas, permitindo rastreamento e monitoração do processamento e das transformações aplicadas aos arquivos, facilitando a depuração e o diagnóstico de possíveis problemas.

4.6. Carregamento dos dados

Esta seção apresenta os componentes concebidos para a fase de carregamento dos dados, uma etapa crucial que envolve operações essenciais para incorporar os dados preparados na fase de transformação a um ambiente de armazenamento que pode ser compartilhado. Nesse contexto, engloba-se a criação de um ambiente de armazenamento isolado para simulação de um *datalake*, a introdução de um mecanismo de verificação prévia dos arquivos já baixados antes do procedimento de transformação dos dados, e a subsequente transferência dos dados processados para o *datalake*.

4.6.1. Verificação de Mudança nos Dados

Durante os testes de execução dos pipelines constatou-se que não há informações sobre intervalos regulares de atualização dos dados presentes tanto na API de Dados Abertos quanto no Painel de Obras. Definir um período de execução de forma aleatória e sem verificar se as informações já foram extraídas em uma execução anterior, pode provocar processamento desnecessário e sub-provisionamento de recursos. Para evitar isso, foi criado uma tabela de metadados que guarda o *hash* de cada arquivo bruto extraído e uma função que faz essa verificação consultando e atualizando essa tabela de registros.

```
CREATE DATABASE IF NOT EXISTS airflow_obras;

USE airflow_obras;

                                        # continua na proxima pagina...
```

```
CREATE TABLE IF NOT EXISTS files_history (
  hash_file CHAR(64) NOT NULL, path_file VARCHAR(256) NOT NULL,
  created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
  updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP
);
```

Quadro 20. Código do script de criação do banco e tabela de dados.

O *script* SQL ilustra o conjunto de instruções utilizado para a criação e gerenciamento de uma estrutura de banco de dados relacional. Essa estrutura é elaborada para atender às necessidades específicas do fluxo de trabalho relacionado à administração e rastreamento de arquivos extraídos no contexto do sistema *Airflow*. A primeira cláusula `CREATE DATABASE IF NOT EXISTS` inicia a criação de um banco de dados denominado `airflow_obras`. Essa instrução assegura que o banco de dados seja criado apenas se ainda não existir. O comando `USE airflow_obras;` define que as instruções subsequentes serão executadas no escopo desse banco de dados recém-criado, estabelecendo o contexto para as operações a seguir.

A próxima parte do *script* se concentra na criação de uma tabela denominada `files_history`. Esta tabela tem como objetivo registrar informações relacionadas a arquivos, o seu hash, caminho, e marcas temporais de criação e atualização. O campo `hash_file` é definido como um caractere de 64 posições, destinado a armazenar valores de hash criptográfico associados aos arquivos. O campo `path_file` é um campo de texto que acomoda o caminho completo do arquivo em questão. Os campos `created_at` e `updated_at` são marcadores temporais que assumem os valores das datas e horários de criação e atualização dos registros, respectivamente. Para estes dois campos, são utilizadas as palavras-chave `TIMESTAMP` e `DEFAULT CURRENT_TIMESTAMP`, assegurando que sejam preenchidos automaticamente com as informações temporais relevantes. O campo `updated_at` possui uma especificidade adicional, que é a cláusula `ON UPDATE CURRENT_TIMESTAMP`, essa cláusula garante que ele seja atualizado automaticamente com o valor do *timestamp* no momento em que ocorrer uma atualização dos registros da tabela.

No desenvolvimento do código *Python*, a função principal é chamada `check_files_in_database`, que recebe uma lista de arquivos a serem verificados e dois fluxos, `fluxo_com_mudancas` e `fluxo_sem_mudancas`, que indicam os possíveis caminhos a serem seguidos após a verificação. Além disso, ela também aceita argumentos do *Apache Airflow* (`**kwargs`), como o objeto `ti` que é usado para manipular informações de execução.

Dentro dessa função, a verificação de cada arquivo é realizada. Primeiro, é calculado o hash SHA-256 do arquivo e é consultado o banco de dados para verificar se esse hash já está registrado na tabela `files_history`. Se o *hash* estiver presente, significa que o arquivo já existe e não houve mudanças nos dados. Caso contrário, o arquivo é novo e será copiado para um destino especificado. A função utiliza o contexto `closing` para garantir que a conexão com o banco de dados seja fechada corretamente após o uso. Ela também manipula exceções que podem ocorrer durante o processo, como erros de arquivos não encontrados ou permissão negada.

Finalmente, a função retorna o fluxo apropriado com base na existência de novos arquivos. Se houver pelo menos um novo arquivo, o fluxo `fluxo_com_mudancas` é

retornado; caso contrário, o fluxo `fluxo_sem_mudancas` é retornado.

A função auxiliar `_copy_file` é responsável por copiar um arquivo de uma localização para outra, com o registro das operações em *logs*. Enquanto a `_calculate_sha256_hash` é usada para calcular o *hash* SHA-256 de um arquivo, que é uma representação única dos dados do arquivo.

4.6.2. Envio para o Datalake

Finalizando a etapa de carregamento dos dados, foi implementado o script destinado a efetuar a transferência dos arquivos para o *datalake* localizado no serviço de armazenamento da AWS, o S3.

```
def upload_arquivos_para_s3(bucket_name, dir_to_update, endpoint_url,
    access_key=None, secret_key=None):
    if not os.path.exists(dir_to_update):
        logging.error(f"Diretorio '{dir_to_update}' nao encontrado.")
        return

    s3_client = boto3.client(
        's3',
        endpoint_url=endpoint_url,
        aws_access_key_id=access_key,
        aws_secret_access_key=secret_key
    )

    try:
        for root, _, files in os.walk(dir_to_update):
            for file in files:
                local_path = os.path.join(root, file)
                s3_key = os.path.relpath(local_path, dir_to_update)
                try:
                    s3_client.upload_file(local_path, bucket_name, s3_key)
                    logging.info(f"Arquivo '{local_path}' enviado para o bucket
                        '{bucket_name}' como '{s3_key}'.")
                    os.remove(local_path)
                    logging.info(f"Arquivo '{local_path}' excluido localmente
                        apos o envio.")
                except Exception as e:
                    logging.error(f"Erro ao enviar o arquivo '{local_path}'
                        para o bucket '{bucket_name}': {e}")

        _deletar_diretorios_vazios(dir_to_update)
    except Exception as e:
        logging.error(f"Erro ao percorrer o diretorio '{dir_to_update}': {e}")
```

Quadro 21. Função que realiza o upload dos arquivos para o S3.

O mecanismo delineado no *script* do Quadro 21 opera mediante a utilização de uma biblioteca *Python* denominada “`boto3`”, a qual estabelece comunicação com os serviços AWS, mais especificamente o *Amazon S3*. Sob essa abordagem, os arquivos que passaram pelo processo de transformação são submetidos a um ciclo de envio, através do qual cada arquivo é individualmente carregado no *bucket*. O *script* também contempla a remoção dos arquivos locais após a conclusão bem-sucedida da transferência, otimizando a gestão de espaço em disco.

Adicionalmente, uma função interna do *script*, apresentada no Quadro 22, realiza a eliminação de diretórios vazios no âmbito do sistema local. Tal medida tem o intuito

de manter a organização e a limpeza do ambiente, removendo diretórios sem arquivos, os quais não são mais necessários após o processo de transferência ter sido consumado.

```
def _deletar_diretorios_vazios(base_dir):
    for root, dirs, _ in os.walk(base_dir, topdown=False):
        for dir_name in dirs:
            dir_path = os.path.join(root, dir_name)
            if not os.listdir(dir_path):
                os.rmdir(dir_path)
                logging.info(f"Diretorio vazio '{dir_path}' excluido localmente
                    .")
```

Quadro 22. Função que a deleção de diretórios vazios.

Dessa maneira, o *script* atua como um agente crucial entre as etapas de transformação e armazenamento dos dados. Sua execução confere a integridade e a consistência dos dados preparados e, ao mesmo tempo, promove a organização e otimização do ambiente local, culminando em uma etapa de carregamento bem definida e eficaz no contexto do fluxo de trabalho de ETL.

4.7. Montagem dos Pipelines

Com o desenvolvimento e preparação das ferramentas, componentes e funções que constituem os pilares das etapas de extração, transformação e carregamento dos dados, é possível fazer a definição dos operadores dentro dos *DAGs*.

Para o *DAG* da API de Dados Abertos foram estabelecidas as seguintes tarefas:

- `cria_diretorios`, que utiliza o BashOperator para criar as pastas de destino dos arquivos brutos e processados durante a execução do *DAG*;
- `extrai_obras`, `extrai_dados_auditoria` e `extrai_dados_contratacao`, que utilizam o operador personalizado (`TcePeOperator`) para fazer a extração a partir de endpoints fornecidos pela API;
- `verifica_novos_arquivos`, que utiliza `BranchPythonOperator` e chama a função de verificação prévia dos arquivos antes de realizar o processo de transformação. O `BranchPythonOperator` atua como um ponto de decisão para definir o caminho que o fluxo irá seguir, ou seja, realizar ou não o processamento dos arquivos brutos;
- `converte_obra_para_csv`, `converte_dados_auditoria_para_csv` e `converte_dados_contratuais_para_csv`, que utilizam `PythonOperator` e chamam a função de conversão de arquivos JSON para CSV;
- `unifica_arquivos`, que utiliza o `PythonOperator` e chama a função de unificação de arquivos;
- `higieniza_dados`, que utiliza o `PythonOperator` e chama a função de higienização dos dados;
- `envia_ao_datalake`, que utiliza o `PythonOperator` e realiza a chamada a função de envio dos dados ao datalake;
- `finaliza_dag`, é a tarefa final que utiliza o `EmptyOperator` e serve apenas para indicar o final do fluxo.

E para o *DAG* da API do Painel de Obras foram estabelecidas as seguintes tarefas:

- `cria_diretorios`, que utiliza o `BashOperator` para criar as pastas de destino dos arquivos brutos e processados durante a execução do *DAG*;
- `extrai_obras`, que utiliza o `PythonOperator` e realiza chamada a função que extrai dados do Painel de Obras usando o webdriver do Selenium;
- `verifica_novos_arquivos`, que utiliza `BranchPythonOperator` e chama a função de verificação prévia dos arquivos antes de realizar o processo de transformação;
- `converte_obra_para_csv`, que utiliza o `PythonOperator` e chama a função de conversão de arquivos JSON para CSV;
- `higieniza_dados`, que utiliza o `PythonOperator` e chama a função de higienização dos dados;
- `envia_ao_datalake`, que utiliza o `PythonOperator` e realiza a chamada a função de envio dos dados ao datalake;
- `finaliza_dag`, é a tarefa final que utiliza o `EmptyOperator` e serve apenas para indicar o final do fluxo.

É necessário estabelecer a ordem de execução e dependências entre as tarefas dos *DAGs*. A forma de ordenação dessas tarefas impactam diretamente em variáveis como tempo de processamento e uso de memória dos *DAGs* criados, algumas possibilidades de arranjo foram testadas e seus resultados serão discutidos na próxima seção.

5. Resultados

Na etapa de extração do pipeline gerado o processo ETL da API de Dados Abertos do TCE-PE, o fluxo inicia criando os diretórios das camadas do datalake com a data da execução, depois coleta e salva na camada bronze, o conteúdo das respostas da requisição feita aos endpoints da API de Dados Abertos em arquivos JSON, contendo os dados gerais sobre as obras, dados de auditoria e dados contratuais.

No início do processo de transformação, os dados são convertidos para o formato CSV e publicados na camada prata, esses arquivos são utilizados para unificação e higienização dos dados para padronização aplicando as funções de transformação descritas na Tabela 2, que realizam a formatação dos valores de entrada como data, valores e de números de documentos.

Função	**Descrição**	**Campos utilizados**
`sanitize_date`	Função que faz uma sequência de transformações no valor de entrada, baseada em uma lista de expressões regulares.	obra-data_auditoria; obra-iniciada_em
`to_uppercase`	Função que transforma todos os caracteres alfabéticos do valor de entrada para maiúsculos.	obra-municipio
`remove_accents`	Função que retira todos as acentuações do valor informado.	obra-municipio

Continua na próxima página

Tabela 2 – Continuação da Página Anterior

Nome da Coluna	Tipo	Descrição
`to_integer`	Função que tenta transformar o valor em um número inteiro.	obra-prazo_inicial; obra-prazo_aditado; obra-prazo_paralisacao
`sanitize_cep`	Função que higieniza os valores de código postal.	endereco-codigo_postal
`add_sufix`	Função que adiciona um sufixo ao valor de entrada	obra-municipio
`chain_functions`	Função que possibilita o encadeamento de múltiplas funções de transformação.	obra-municipio

Tabela 2. Funções de transformação de valores.

Por fim, o resultado da execução do pipeline, gerou um arquivo CSV contendo 2.462 registros de obras, em 179 municípios pernambucanos e em 92 categorias. Também é possível recuperar outras informações como local de execução da obra, data inicial, prazos determinados para execução e prazo de paralisação, além de informações sobre o órgão público responsável pela licitação da obra e pessoa física ou jurídica contratada para a execução. Esses dados podem revelar *insights* valiosos para uma gestão eficaz destes e de futuros projetos como a identificação de prioridades de investimento, avaliação de desempenho de órgãos e contratados, identificação de impacto de paralisações e montagem de estratégias para mitigá-las.

Com o objetivo de analisar e avaliar as variações de tempo em diferentes abordagens de execução, um estudo comparativo do desempenho de três diferentes modelos de execução foi aplicado no contexto da API de Dados Abertos. Os três modelos de *DAG* foram examinados quanto ao tempo de execução em 30 iterações cada.

A primeira abordagem envolveu a execução sequencial das operações, onde cada operador só inicia após a conclusão do anterior. Isso resulta em um tempo mínimo de execução igual ao tempo da operação mais demorada e um tempo máximo que considera a soma de todas as operações. O tempo médio é ponderado pelo tempo das operações subsequentes, refletindo um aumento gradual no tempo à medida que as operações avançam. A desvantagem desta abordagem é a ineficiência devido à falta de paralelismo, o que resulta em um tempo médio mais próximo do tempo máximo.

Na segunda abordagem, as operações de extração e conversão foram executadas em paralelo, reduzindo significativamente o tempo total de execução. Isso foi possível dividindo a *DAG* em três caminhos independentes, cada um com sua própria cadeia de operadores. O tempo mínimo de execução é determinado pelo caminho mais longo, onde a operação mais demorada define o limite inferior. O tempo máximo continua a ser a soma dos tempos das operações em todos os caminhos.

A terceira abordagem, apresenta uma otimização adicional, introduzindo uma etapa de verificação condicional de dados antes da conversão. Isso reduz o tempo mínimo de execução, uma vez que a conversão só ocorre se houver dados novos, evitando operações desnecessárias. O tempo máximo continua a ser definido pelo caminho mais longo,

mas a presença da verificação de dados pode reduzir significativamente o tempo médio em comparação com os modelos de execução anteriores, já que a conversão condicional evita o processamento quando não é necessário. Isso ilustra a importância de estratégias inteligentes de execução para otimizar o tempo de processamento em pipelines de dados.

As abordagens mencionadas, são ilustradas abaixo nas Figuras 8, 9 e 10, respectivamente:

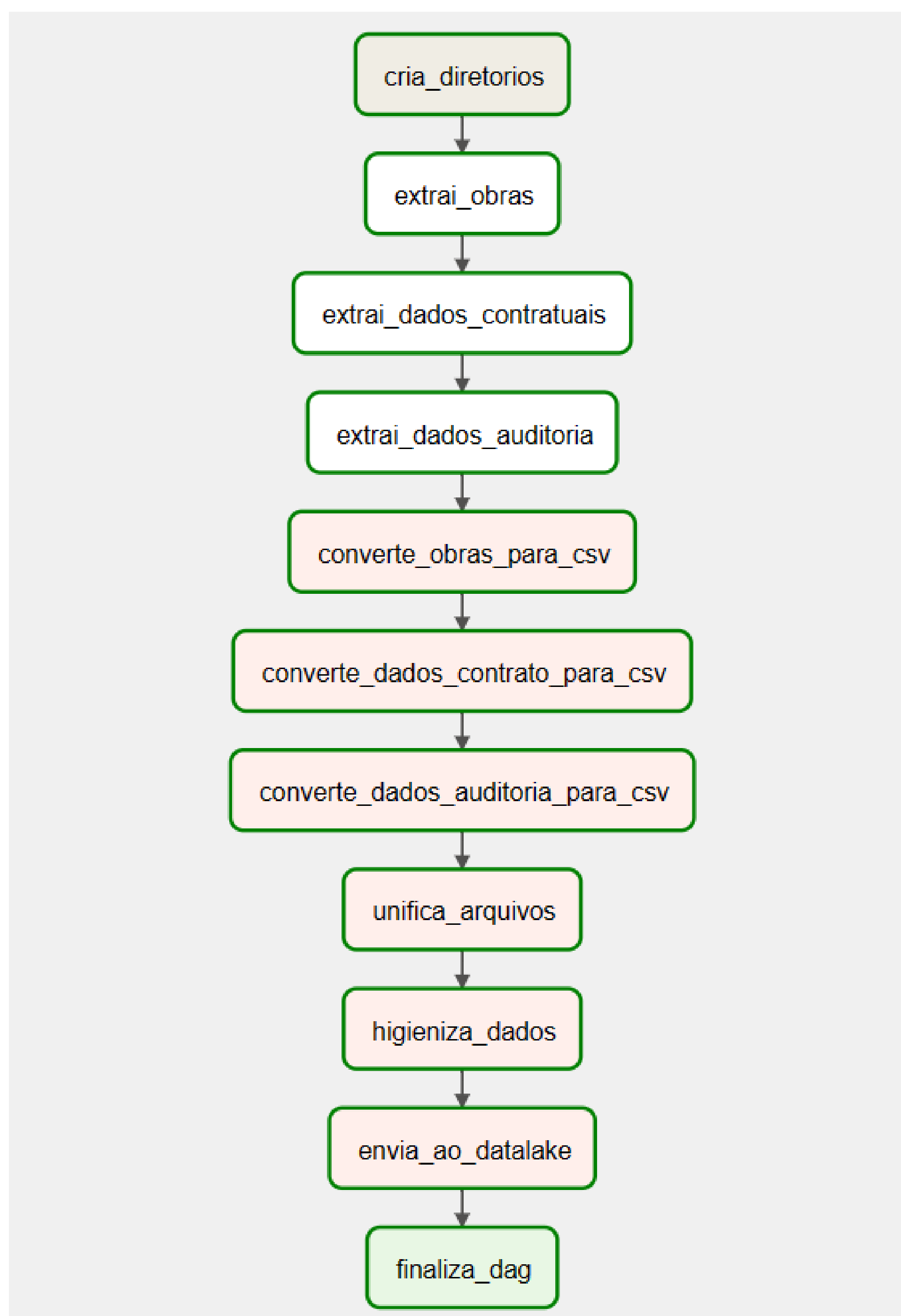

Figura 8. Ilustração do gráfico de execução do *DAG* de modelo linear.
Fonte: O autor.

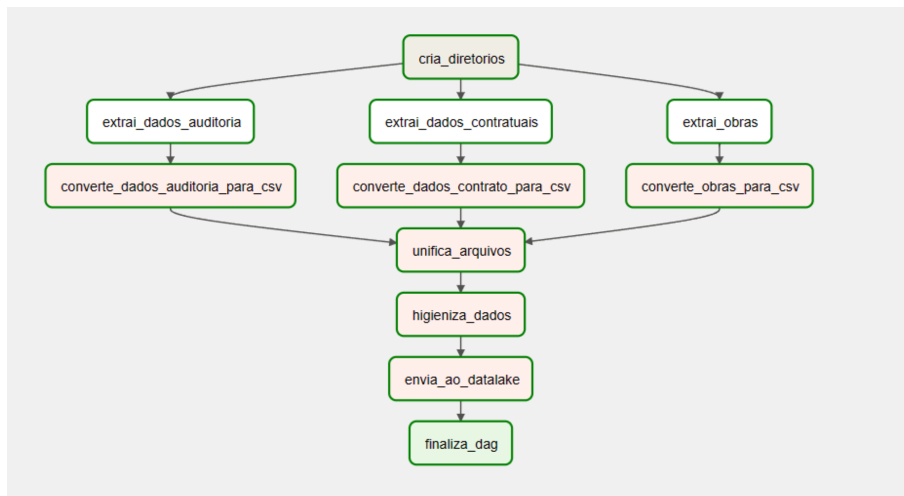

Figura 9. Ilustração do gráfico de execução do *DAG* de modelo paralelo.
Fonte: O autor.

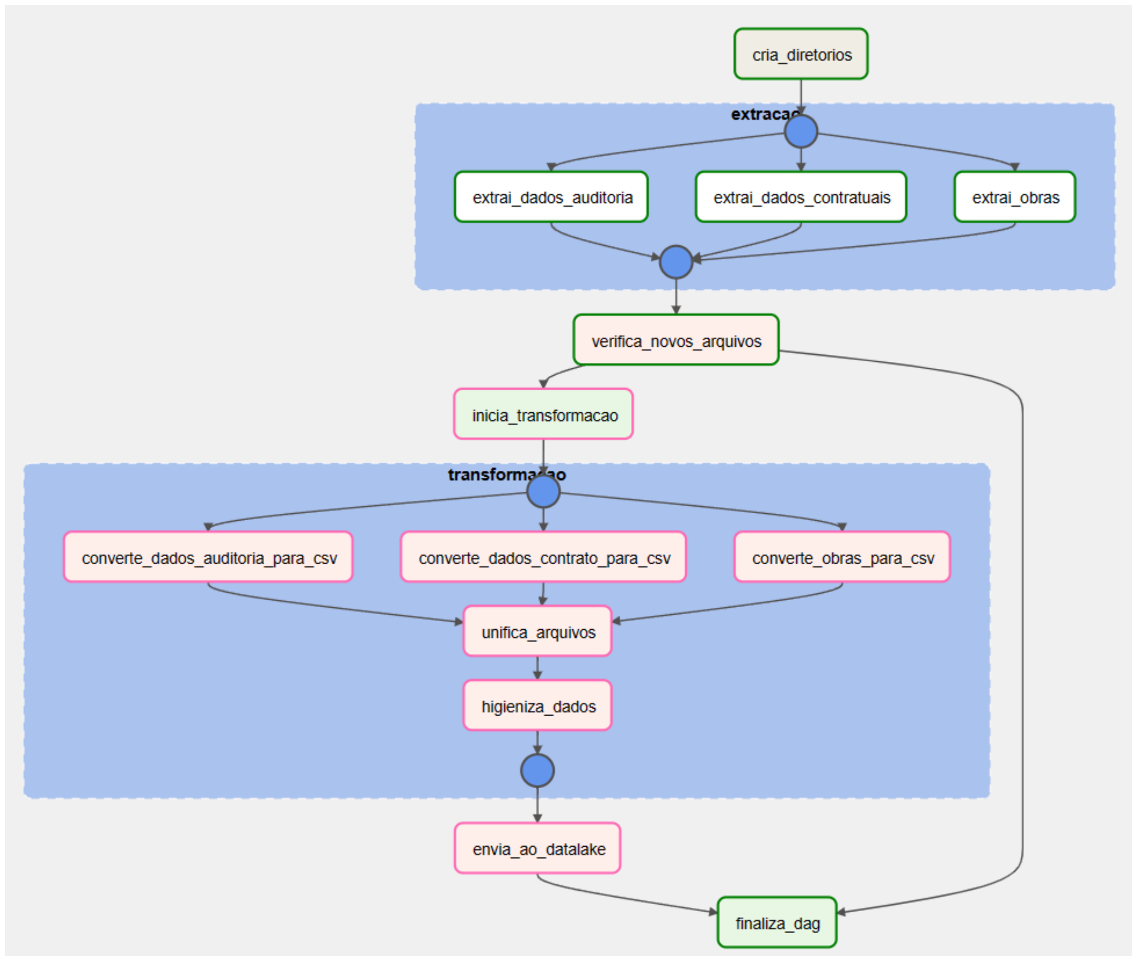

Figura 10. Ilustração do gráfico de execução do *DAG* de modelo paralelo com conversão condicionada.
Fonte: O autor.

Os resultados descritos na Tabela 3 revelam claramente a influência das diferentes abordagens nos tempos de execução. O segundo modelo demonstrou a capacidade de reduzir significativamente o tempo de processamento, aproveitando o paralelismo das tarefas. Além disso, a introdução de uma tarefa de verificação prévia no terceiro modelo, teve um impacto notável na redução do tempo médio de execução, resultando em maior eficiência no processo de transformação. Porém deve-se ressaltar que o processo também pode depender de fatores externos, como a velocidade de download e upload dos dados.

Modelo	Tempo mínimo	Tempo Máximo	Tempo Médio	Desvio Padrão
Linear	21s	44s	24s	6.42s
Paralelo	7s	14s	12s	2.38s
Condicional	6s	19s	9s	4,81s

Tabela 3. Resultados do comparativo de modelos de execução dos *DAGs*

A vantagem de usar operadores paralelos para extração de dados no *Apache Airflow* está relacionada ao aumento da eficiência e da escalabilidade do processo de extração de dados. Os operadores paralelos permitiram que tarefas fossem executadas simultaneamente em diferentes nós de execução, o que reduziu significativamente a média do tempo de execução de um fluxo de trabalho em aproximadamente 50%. Isso é especialmente importante em tarefas de engenharia de dados, que podem envolver grandes quantidades de dados e, portanto, demandar bastante tempo de processamento.

Além disso, o uso de operadores paralelos pode tornar o processo de extração de dados mais escalável, permitindo que o fluxo de trabalho seja executado em múltiplos nós, sendo útil em ambientes de computação distribuída, onde é possível escalar horizontalmente a capacidade de processamento adicionando mais nós de execução ao sistema.

De modo semelhante ao *DAG* da API de Dados Abertos, o pipeline gerado para o processo do Painel de Obras cria um diretório na camada bronze com a data da execução, mas desta vez salva a planilha obtida da interação automatizada da função de extração via *Selenium* do portal web. Essa planilha contém várias informações inerentes às obras como órgão responsável, situação da obra, endereços e informações financeiras. A relação de todos os campos está descrita na Tabela 4.

Nome da Coluna	Tipo	Descrição
ID Obra	Texto	Identificador único da obra.
Código Transação Obra	Texto	Código de transação associado à obra
Origem	Texto	Origem da obra
Nº Instrumento	Texto	Número do instrumento associado à obra
Órgão Superior	Texto	Órgão superior responsável pela obra
Órgão	Texto	Órgão responsável pela obra
Objeto	Texto	Descrição do objeto da obra
Título	Texto	Título da obra
Situação Atual	Texto	Situação atual da obra
Ano Início Obra	Ano	Ano de início da obra

Continua na próxima página

Tabela 4 – Continuação da Página Anterior

Nome da Coluna	Tipo	Descrição
Data Início	Data	Data de início da obra
Ano Fim Obra	Ano	Ano previsto para o término da obra.
Data Fim	Data	Data prevista para o término da obra
UF	Texto	Unidade federativa onde a obra está localizada
Município	Texto	Município onde a obra está localizada
Endereço	Texto	Endereço da obra
Latitude	Texto	Coordenada de latitude da localização da obra
Longitude	Texto	Coordenada de longitude da localização da obra
Link	Texto	Link relacionado à obra
Execução Física	Porcentagem	Porcentagem de execução física da obra
Execução Financeira	Texto	Status da execução financeira da obra
Investimento Total	Moeda (Real)	Valor total do investimento na obra
Modalidade	Texto	Modalidade da obra
Tipo de Instrumento	Texto	Tipo de instrumento associado à obra
Tipo	Texto	Tipo da obra
Subtipo	Texto	Subtipo da obra
CNPJ Executor	Texto	CNPJ do executor da obra
Funcional Programática	Texto	Código funcional programático associado à obra
Emenda	Texto	Informação sobre a existência de emenda para a obra
Plurianual Prioritário	Booleano	Indicação de projeto plurianual prioritário
PróBrasil	Booleano	Indicação de participação no programa "Pró-Brasil"
Restos a Pagar	Texto	Indicação de existência de restos a pagar
Valor Conclusão	Moeda (Real)	Valor previsto para a conclusão da obra
Valor Empenhado	Moeda (Real)	Valor empenhado para a obra
Valor Repasse	Moeda (Real)	Valor total de repasse para a obra.
Valor Desembolsado	Moeda (Real)	Valor desembolsado até o momento para a obra
Motivo Paralisação	Texto	Motivo da paralisação da obra
Causa Paralisação	Texto	Causa da paralisação da obra
Justificativa Tratativa	Texto	Justificativa para a tratativa da obra
Data Previsão Retomada	Data	Data prevista para a retomada da obra
Data Criação	Data	Data de criação dos registros relacionados à obra

Continua na próxima página

Tabela 4 – Continuação da Página Anterior

Nome da Coluna	Tipo	Descrição
Data Atualização	Data	Data da última atualização dos registros relacionados à obra

Tabela 2. Lista de dados coletados do Painel de Obras.

No processo de transformação dos dados, a planilha é carregada em um *dataframe* do Pandas e os dados são convertidos para o formato CSV, sendo publicados na camada prata, o arquivo gerado é então utilizado higienização dos dados e padronização de formatos de informações como formato de data, valores e de números de documentos.

O resultado da execução do pipeline, gerou um arquivo CSV contendo 7.243 registros de obras, em 184 municípios pernambucanos e em 24 categorias.

6. Considerações Finais

A implementação de fluxos de dados automatizados no contexto de gestão de obras públicas do estado de Pernambuco, demonstrou uma abordagem eficaz para melhorar a eficiência operacional contribuindo para a diminuição do trabalho humano necessário para obtenção de informações. Os resultados apresentados revelam a significativa contribuição de dados obtidos por meio da implementação dos pipelines. O amplo número de registros de obras, abrangendo diversos municípios e categorias, fornece uma base sólida para análises detalhadas e uma visão abrangente das atividades de construção em Pernambuco.

A disponibilidade desses dados oferece ótimas oportunidades para a tomada de decisões no âmbito da gestão pública com a identificação de informações, como localização, datas e responsabilidades, que possibilitam uma análise aprofundada que pode orientar estratégias e prioridades de investimento futuras. Além disso, os dados coletados permitem a avaliação de desempenho tanto de órgãos públicos quanto de contratados, trazendo insumos para o acompanhamento e a prestação de contas, análises que pode levar a melhorias na eficiência e transparência das operações públicas.

No entanto, há várias oportunidades de melhoria que podem ser exploradas para aprimorar a qualidade e abrangência da análise de dados relacionadas a essa área. Primeiramente, a possível expansão da abordagem para incluir obras de todo o território nacional seria uma etapa natural. Isso permitiria uma visão abrangente desses serviços públicos, facilitando a análise comparativa entre diferentes regiões e auxiliando na tomada de decisões estratégicas em uma escala mais abrangente, proporcionando uma visão mais completa acerca das políticas públicas relacionadas à infraestrutura do Brasil.

A adição de informações relevantes de outras fontes, como o Instituto Brasileiro de Geografia e Estatística (IBGE), seriam fundamentais para enriquecer a análise, incluindo informações geoespaciais, demográficas e econômicas, para realização de análises mais robustas. Essas análises podem ajudar a identificar áreas prioritárias para investimentos, levando em consideração fatores como densidade populacional, infraestrutura existente e necessidades locais específicas. Outro ponto importante a ser considerado é a inclusão de informações detalhadas sobre a situação jurídica dos executores das obras. Isso permitiria uma análise mais profunda da eficiência e confiabilidade das empresas contratadas

para realizar as obras públicas. A identificação de empresas com histórico de atrasos, baixa qualidade de entrega ou irregularidades poderia direcionar esforços de fiscalização e garantir a integridade dos projetos.

Além disso, informações adicionais, como o histórico de custos e prazos de projetos anteriores, a alocação de recursos (incluindo mão-de-obra e materiais) e os impactos ambientais das obras, também poderiam ser incorporadas.

A aplicação de pipelines automatizadas de dados no contexto de obras públicas em Pernambuco representa mais uma abordagem que pode ser explorada em uma área que ainda carece do uso de tecnologias para otimização de processos. E demonstra que a busca contínua por melhorias, pode enriquecer ainda mais a análise de dados e fortalecer a tomada de decisões no âmbito da gestão pública. Essas melhorias não apenas aprimoram a eficiência operacional, mas também contribuem para um desenvolvimento mais sustentável e equitativo da sociedade brasileira.

Referências

Androniceanu, A. (2021). Transparency in public administration as a challenge for a good democratic governance. *ADMINISTRATIE SI MANAGEMENT PUBLIC*.

Apache (2021). Airflow documentation.

Belov, V. and Nikulchev, E. (2021). Analysis of big data storage tools for data lakes based on apache hadoop platform. *International Journal of Advanced Computer Science and Applications*, 12(8).

Brasil (2011). Lei nº 12.527, de 18 de novembro de 2011. Regula o acesso a informações previsto no inciso XXXIII do art. 5º, no inciso II do § 3º do art. 37 e no § 2º do art. 216 da Constituição Federal. Diário Oficial da União, Brasília, DF, 18 nov. 2011. Disponível em: http://www.planalto.gov.br/ccivil_03/_ato2011-2014/2011/lei/l12527.htm. Acesso em: 06 de fevereiro de 2023.

CGU (2020). Relatório de auditoria nº 843821 - comitê interministerial de governança cig - levantamento de obras paralisadas. Disponível em: https://eaud.cgu.gov.br/relatorios/download/900153. Acesso em 10 de setembro de 2023.

DOU (2019). Decreto presidencial nº 10.012, de 05 de setembro de 2019. Brasília, DF: Diário Oficial da União, 06 de Setembro de 2019. Seção 1, p. 5.

Engin, Z. and Treleaven, P. (2018). Algorithmic government: Automating public services and supporting civil servants in using data science technologies. *The Computer Journal*, 61(9):1280–1294.

Fang, H. (2015). Managing data lakes in big data era: What's a data lake and why has it became popular in data management ecosystem. In *The 5th Annual IEEE International Conference on Cyber Technology in Automation, Control and Intelligent Systems*, pages 8–12, Shenyang, China.

Feng, A. (2020). *Data Pipelines with Apache Airflow*. Manning Publications, Shelter Island, NY.

Ferry, L. and Eckersley, P. (2014). Accountability and transparency: A nuanced response to etzioni. *Public Administration Review*, 75:11 – 12.

Figueiras, P., Costa, R., Guerreiro, G., Antunes, H., Rosa, A., and Jardim-Gonçalves, R. (2016). User interface support for a big etl data processing pipeline: An application scenario on highway toll charging models. *CTS, UNINOVA, Dep*.

Jr, J. C. C. (2011). Planejamento governamental e gestão pública no brasil: Elementos para ressignificar o debate e capacitar o estado. Technical report, IPEA, Brasília, DF.

Kakish, K. and Kraft, T. A. (2012). Etl evolution for real-time data warehousing. In *Conisar Proceedings*, page 12, New Orleans, Louisiana, USA. EDSIG (Education Special Interest Group of the AITP).

L'Esteve, R. C. (2023). *Designing a Secure Data Lake*, pages 183–201. Apress, Berkeley, CA.

Mitlöhner, J., Neumaier, S., Umbrich, J., and Polleres, A. (2016). Characteristics of open data csv files. In *2nd International Conference on Open and Big Data (OBD)*, pages 72–79, Vienna, Austria.

Muller, C. (2020). *Data Pipelines Pocket Reference*. O'Reilly Media, Sebastopol, CA, 1st edition.

Nargesian, F., Zhu, E., Miller, R. J., Pu, K. Q., and Arocena, P. C. (2019). Data lake management: Challenges and opportunities. *Proc. VLDB Endow.*, 12(12):1986–1989.

Nazário, D. C., da Silva, P. F., and Rover, A. J. (2012). Avaliação da qualidade da informação disponibilizada no portal da transparência do governo federal. *Revista Democracia Digital*, (6):180–199.

Pandas (Acesso em 2023). pandas 1.3.4 documentation. Acessado em 09 de setembro de 2023.

Selenium (Acesso em 2023). Selenium webdriver documentation. `https://selenium.dev/documentation/en/`. Acessado em 09 de setembro de 2023.

Silva, C. F. d., Vaz, W., Santos, E. M. F. d., Balaniuk, R., and Chaves, M. C. (2014). Open data: a strategy for increased public management transparency and modernization. *Revista TCU*, Ed. 131:23–29.

Smanio, G. P. and Nunes, A. R. S. (2016). Transparência e controle social de políticas públicas: efetivação da cidadania e contribuição ao desenvolvimento. *Interfaces Científicas - Humanas e Sociais*, 4(3):83–96.

TCE/PE (2021). Diagnóstico de obras paralisadas. Disponível em: https://www.tce.pe.gov.br/internet/docs/tce/Diagnostico_Obras%20Paralisadas.pdf. Acesso em 10 de setembro de 2023.

A. Anexo de funções de transformação

```
import re
from datetime import datetime

from unidecode import unidecode


def to_integer(x):
    if type(x) == int:
        return x
    return 0


def sanitize_cep(x):
    field = re.sub(r'\D', '', x)
    return field if field != '' else ''


def sanitize_documents(x):
    field = re.sub(r'[.-]', '', x)
    return field if field != '' else ''


def add_suffix(x, suffix):
    return x+suffix if x != '' else x


def sanitize_date(x, actual_format, new_format):
    try:
        actual = datetime.strptime(x, actual_format)
        return actual.strftime(new_format)
    except ValueError:
        return x


def replace_words(x, words):
    return words[x] if x in words else x


def remove_non_digits(x):
    return re.sub(r'\D', '', x)


def remove_accents(x):
    return unidecode(x)


def to_uppercase(x):
    return x.upper()


def chain_functions(x, functions):
    for (fun, extra_args) in functions:
        x = fun(x, **extra_args)
    return x
```