# Relatório <11> — < — Prática: Pipelines de Dados — Airflow (I)>

<Jonas Correia>

## 1. Why data orchestration?

A orquestração de dados consiste em coordenar e automatizar processos relacionados à movimentação, transformação e integração de dados entre diferentes sistemas, garantindo que os fluxos de trabalho sejam eficientes, organizados e confiáveis. Essa prática é essencial em um cenário em que os dados se tornam cada vez mais centrais para decisões organizacionais e operacionais. Um orquestrador de dados, como o Apache Airflow, torna esse processo mais estruturado, especialmente ao lidar com tarefas interdependentes.

Em um fluxo de trabalho de dados típico, como a execução de scripts para extrair, limpar, transformar e carregar dados, a definição de dependências entre essas tarefas é crucial. Um orquestrador garante que cada etapa do processo ocorra na ordem correta, como executar a extração antes da limpeza e assim por diante. Caso uma etapa falhe, como a de limpeza, as tarefas subsequentes não serão executadas. Essa abordagem automatizada elimina a necessidade de monitoramento manual constante e reduz o risco de erros ao longo do processo.

Sem um orquestrador, o gerenciamento de falhas é mais trabalhoso, exigindo a identificação manual da tarefa problemática, a análise de registros e a execução manual da correção. Isso pode ser especialmente desafiador em cenários complexos ou em horários inconvenientes, como finais de semana. O orquestrador, por outro lado, notifica automaticamente sobre falhas e pode reiniciar a tarefa de forma automatizada, economizando tempo e esforço.

À medida que os fluxos de trabalho de dados crescem em escala, com milhares de tarefas e múltiplos pipelines, torna-se inviável gerenciá-los manualmente em uma única máquina. Limitações de recursos computacionais e a complexidade do processo tornam essa abordagem insustentável. O Apache Airflow permite distribuir tarefas em várias máquinas, tornando possível gerenciar fluxos de trabalho de dados em larga escala com eficiência.

Além de proporcionar escalabilidade, o orquestrador simplifica o monitoramento e o gerenciamento dos fluxos de trabalho, oferecendo uma visão clara de todas as etapas e permitindo a criação de fluxos mais sofisticados. A capacidade de gerenciar fluxos de trabalho de forma organizada e confiável torna ferramentas como o Airflow indispensáveis em ambientes de dados modernos, onde a complexidade e o volume de informações são cada vez maiores.

## 2. Why Airflow?

No vídeo, são apresentados cinco motivos pelos quais o Apache Airflow é uma ferramenta essencial para orquestração de dados, além de como ele se destaca em comparação a outros orquestradores. A explicação começa destacando o papel de um orquestrador de dados, que é fundamental para coordenar fluxos de trabalho complexos. Como exemplo, o vídeo ilustra a execução de tarefas de extração, limpeza, transformação e carregamento de dados, evidenciando como o Airflow facilita a criação de dependências e a automação desse processo. Sem um orquestrador, a execução ordenada e monitoramento dessas tarefas seria extremamente trabalhosa e sujeita a falhas.

A escalabilidade é apontada como um dos grandes diferenciais do Airflow. Ele pode gerenciar desde fluxos simples até milhares de tarefas simultaneamente, sendo utilizado por grandes empresas como Airbnb e Walmart. A possibilidade de integração com tecnologias como Celery, Kubernetes e Dask permite que o Airflow distribua tarefas entre múltiplas máquinas, garantindo desempenho em grande escala. Além disso, o componente de agendamento do Airflow, com suporte a múltiplos agendadores, assegura a continuidade e confiabilidade da execução, mesmo em caso de falhas.

Outro motivo relevante são as vastas integrações disponíveis. O Airflow se conecta com uma ampla gama de ferramentas e serviços, como Airbyte, DBT, Snowflake e AWS, tornando a orquestração de fluxos de trabalho flexível e eficiente. Caso uma integração específica não esteja disponível, é possível criar plugins personalizados, evidenciando a modularidade e capacidade de personalização da ferramenta.

A acessibilidade do Airflow, fundamentada na linguagem Python, é destacada como outro ponto de força. Além de permitir a criação dinâmica de pipelines, o Airflow é suficientemente flexível para gerenciar tarefas com entradas desconhecidas até o momento da execução, tornando-o adequado para fluxos de trabalho complexos e adaptáveis. Adicionalmente, os recursos de monitoramento, como notificações de erros e linhagem de dados, permitem rastrear dependências e compreender os impactos de falhas, essenciais para um ambiente de produção confiável.

Por fim, o vídeo enfatiza a importância da comunidade ativa do Airflow. Com milhares de usuários e colaboradores, o projeto recebe atualizações constantes e possui suporte robusto, garantindo sua relevância a longo prazo. A combinação de escalabilidade, flexibilidade, monitoramento avançado, suporte a integrações e uma comunidade vibrante consolida o Apache Airflow como uma solução de destaque para orquestração de dados.

## 3. The Core Components

O Airflow é uma plataforma de orquestração de workflows, amplamente utilizada para automatizar e agendar processos de dados, como pipelines. Sua arquitetura é composta por diversos componentes essenciais que permitem a execução e monitoramento eficientes desses processos. A compreensão dos componentes do Airflow é fundamental para aproveitar todo o seu potencial em automação de tarefas.

Ao iniciar o Airflow, o primeiro componente com o qual o usuário interage é a interface de usuário, que é um painel visual para monitoramento e gerenciamento dos workflows. Esta interface é alimentada por um servidor web, responsável por fornecer o acesso à interface e garantir que os dados visíveis sejam atualizados e acessíveis em tempo real. O servidor web é, portanto, um dos componentes mais fundamentais, ao permitir que os usuários interajam diretamente com a plataforma.

Os dados apresentados na interface de usuário são os chamados "metadados". Esses metadados incluem informações detalhadas sobre os pipelines de dados, o status das tarefas, suas dependências e seus tempos de execução. Para garantir a persistência e integridade dessas informações, os metadados são armazenados em um banco de dados relacional, como MySQL ou Postgres. Esse banco de dados é crucial para o funcionamento do Airflow, pois mantém o registro de todas as execuções e configurações, permitindo o gerenciamento e a consulta dessas informações posteriormente.

Outro componente essencial do Airflow é o agendador, que pode ser descrito como o "coração" do sistema. Ele é responsável por gerenciar e monitorar a execução das tarefas. O agendador determina quando as tarefas devem ser executadas, levando em consideração as dependências entre elas. Por exemplo, se uma tarefa depende de outra para ser executada, o agendador garante que a primeira tarefa seja concluída antes da execução da segunda. Além disso, o agendador é responsável por verificar se as condições de execução das tarefas são atendidas, o que inclui a verificação de agendamentos, tempos e dependências.

O agendador trabalha de forma colaborativa com o executor. Embora o executor não seja visível como um componente independente, ele desempenha um papel vital na plataforma. O executor é responsável por decidir como e em qual sistema as tarefas serão executadas. Ele organiza as tarefas em uma fila e as encaminha para o sistema de execução adequado. Dependendo das necessidades da infraestrutura, o Airflow oferece diferentes tipos de executores. Por exemplo, no caso de um ambiente Kubernetes, pode-se utilizar o executor Kubernetes, que distribui a execução das tarefas nos pods do Kubernetes. Para ambientes com

múltiplos workers, pode-se optar pelo executor Celery, que distribui as tarefas através de múltiplos workers de Celery. Para execução em um único computador, o executor local pode ser suficiente, mas cada tipo de executor oferece vantagens específicas dependendo do cenário.

## 4. The Core Concepts

No Airflow, os conceitos fundamentais que estruturam sua operação são as tarefas, operadores, dependências e os DAGs (Grafos Acíclicos Direcionados). Esses componentes formam a base para a criação e orquestração de pipelines de dados, permitindo que processos sejam automatizados e controlados de maneira eficaz.

Uma tarefa no Airflow é a unidade básica de execução. Cada tarefa representa uma ação individual que será realizada dentro de um fluxo de trabalho. Por exemplo, uma tarefa pode envolver a execução de uma consulta SQL, o disparo de um script Bash, ou a execução de um código Python. Essas ações são realizadas por meio de operadores, que são templates predefinidos que encapsulam a lógica necessária para realizar determinada tarefa. O Airflow oferece uma vasta gama de operadores prontos para uso, evitando a necessidade de reinventar soluções para tarefas comuns. Alguns exemplos incluem o operador PostgresOperator, para executar comandos SQL em um banco de dados Postgres, o BashOperator, para rodar scripts Bash, e o PythonOperator, para executar funções Python.

Normalmente, os pipelines no Airflow são compostos por várias tarefas, que muitas vezes precisam ser executadas em uma ordem específica. Para garantir que isso aconteça, as tarefas podem ser interligadas por dependências, o que significa que uma tarefa deve ser concluída com sucesso antes que a próxima comece. Essa dependência entre as tarefas é uma característica crucial para a criação de workflows complexos e controlados. Ao definir dependências entre tarefas, o Airflow assegura que as etapas do fluxo de trabalho sejam realizadas na ordem certa, respeitando a lógica de execução dos dados.

Quando as tarefas são organizadas e suas dependências são estabelecidas, formam o que é chamado de DAG (Directed Acyclic Graph), ou gráfico acíclico direcionado. Um DAG é, essencialmente, um pipeline de dados. Em termos gráficos, ele é composto por nós, que representam as tarefas, e as bordas, que representam as dependências entre essas tarefas. O gráfico é chamado de "acíclico" porque não pode haver ciclos, ou seja, uma tarefa não pode depender de si mesma indiretamente. Isso garante que o fluxo de execução siga uma sequência lógica, sem voltar para etapas anteriores.

O DAG pode ser visualizado na interface do usuário do Airflow, onde a ordem de execução das tarefas e suas dependências são representadas

graficamente. O exemplo de DAG mostrado ilustra um pipeline de dados comum, onde as tarefas de extração, transformação e carga (ETL) são realizadas em sequência, e as setas indicam a ordem de execução. Esse pipeline é um exemplo claro de como tarefas interconectadas, com suas dependências bem definidas, formam um fluxo de trabalho completo e automatizado.

Portanto, ao trabalhar com o Airflow, o entendimento desses conceitos — tarefas, operadores, dependências e DAGs — é fundamental para criar pipelines de dados eficientes e funcionais. As tarefas são as ações que serão executadas, os operadores são os modelos predefinidos para facilitar a execução dessas ações, as dependências controlam a ordem de execução e o DAG conecta tudo isso, garantindo que o fluxo de dados aconteça de forma acíclica e lógica.

## 5. How does Airflow work?

Quando você cria e executa um DAG no Airflow, diversos componentes trabalham juntos para orquestrar a execução do pipeline de dados. O processo começa com a criação de um novo arquivo Python que define o seu DAG, o qual deve ser colocado no diretório específico de DAGs. Uma vez que o arquivo é adicionado, o agendador do Airflow entra em ação para monitorar o diretório de DAGs, verificando novos arquivos a cada cinco minutos, por padrão.

Ao identificar um novo arquivo, o agendador realiza a serialização do DAG, que é o processo de armazenar o código do DAG no banco de dados de metadados (Metadatabase). A serialização facilita o acesso ao DAG, já que a leitura de dados de um banco de dados é mais eficiente do que ler diretamente de um diretório local. Esse processo de serialização permite que o servidor web e outros componentes do Airflow acessem o DAG de forma mais rápida e organizada.

Após a serialização, o agendador verifica as condições para a execução do DAG, com base nas configurações definidas no arquivo. Se tudo estiver configurado corretamente, o agendador cria uma instância de execução do DAG, que é simplesmente uma execução específica do DAG em um determinado momento. Essa execução pode ter diferentes estados, como "em execução" ou "completa". O agendador, então, passa a monitorar as tarefas definidas no DAG para verificar se elas estão prontas para ser executadas.

Quando uma tarefa está pronta, o agendador cria uma instância de tarefa para cada tarefa do DAG. Assim como a execução do DAG, as instâncias de tarefa podem ter diferentes estados, como "programado", "em execução" ou "concluído". O agendador então envia a instância da tarefa para o executor. O executor, por sua vez, não executa a tarefa diretamente, mas define em qual sistema a tarefa será executada. O executor coloca a instância de tarefa em uma

fila, e essa fila é usada para organizar a execução das tarefas, embora o usuário não veja diretamente esse processo.

Uma vez que a tarefa está enfileirada, o status da tarefa é atualizado no Metadatabase para refletir esse estado. Um trabalhador (worker), que pode ser um processo interno ou externo, dependendo da configuração do Airflow, pega a instância da tarefa da fila e a executa. Após a execução, o trabalhador atualiza o estado da instância da tarefa no Metadatabase com o status final, como "sucesso" ou "falha".

O agendador, então, verifica se há mais tarefas a serem executadas. Se todas as tarefas foram completadas, o agendador marca a execução do DAG como bem-sucedida ou falhada, com base nos estados finais das tarefas. Todo esse processo acontece em segundo plano, enquanto o usuário pode acompanhar o progresso e o estado das tarefas por meio da interface web do Airflow, que permite monitorar em tempo real o fluxo das execuções do pipeline de dados.

Essa interação entre os componentes do Airflow — agendador, executor, workers e metadatabase — é fundamental para a execução eficiente de pipelines de dados. Embora o processo seja mais complexo em cenários avançados, essa visão geral fornece uma compreensão básica de como o Airflow orquestra a execução de tarefas em um DAG e mantém o controle sobre o estado e progresso das tarefas em tempo real.

## 6. Airflow limitations

O Apache Airflow, embora seja uma ferramenta poderosa para orquestração de workflows e pipelines de dados, possui algumas limitações que é importante entender ao planejar seu uso em diferentes cenários.

A primeira limitação é relacionada ao streaming. O Airflow não é adequado para processamento de dados em tempo real ou de forma contínua. Ele é projetado para orquestrar tarefas em lote, ou seja, ele processa dados em intervalos de tempo definidos, como a cada minuto, mas isso implica que sempre haverá alguma latência entre a execução das tarefas. Embora seja possível integrar o Airflow com sistemas de streaming, como o Kafka, a execução dos pipelines ainda não será em tempo real. O Airflow não foi desenvolvido para lidar com dados em tempo real de forma imediata, sendo mais eficaz para tarefas que podem ser executadas periodicamente.

A segunda limitação se refere ao processamento de dados. Embora o Airflow possa ser usado para orquestrar tarefas de processamento de dados, ele

não é projetado para ser uma plataforma de processamento intensivo de dados, como o Apache Spark. Se você tentar processar grandes volumes de dados diretamente no Airflow, pode enfrentar problemas de estouro de memória (memory overflow) devido à limitação de recursos do próprio Airflow. Isso ocorre porque o Airflow é principalmente um orquestrador e não uma ferramenta de processamento de dados. Em vez de processar grandes quantidades de dados diretamente no Airflow, é recomendável usá-lo para acionar tarefas ou scripts que deleguem o processamento para ferramentas ou serviços externos especializados, como o Spark, que são melhor equipados para lidar com grandes volumes de dados de forma eficiente.

## 7. [Practice] Installing Airflow

A instalação e configuração do Apache Airflow podem ser realizadas por diferentes métodos, cada um com suas peculiaridades, vantagens e desvantagens. Para entender a abordagem mais eficiente e prática, é necessário analisar as etapas e fundamentos técnicos de cada processo apresentado.

A instalação via pip, o gerenciador de pacotes do Python, é considerada a forma mais manual e básica de configuração. O pip é uma ferramenta amplamente usada para gerenciar bibliotecas e pacotes Python, permitindo a instalação direta de ferramentas como o Airflow. Porém, a instalação manual via pip exige o cumprimento de diversos pré-requisitos e comandos sequenciais. Por exemplo, para usuários de sistemas operacionais Windows, é necessário garantir a presença do Python (versão 3.7 ou superior), o pip e o subsistema Windows para Linux (WSL) habilitado com o Ubuntu. Além disso, é preciso configurar um ambiente virtual Python (virtualenv), uma prática recomendada para evitar conflitos entre bibliotecas já instaladas no sistema e as dependências específicas do Airflow.

Dentro do ambiente virtual, o Airflow requer um diretório de configuração definido, conhecido como AIRFLOW_HOME, onde são armazenados metadados, arquivos de log e o banco de dados SQLite padrão. Outro passo crítico é o uso de um arquivo de restrição para controlar as versões das dependências do Python. Essa abordagem ajuda a prevenir conflitos de versão que poderiam comprometer a funcionalidade do Airflow. Apesar de todos esses passos culminarem na instalação e inicialização do Airflow, essa abordagem tem limitações. A instalação básica por pip não suporta execução paralela eficiente de tarefas, o que exigiria a migração para um banco de dados robusto, como o PostgreSQL, além de alterações adicionais no arquivo de configuração do Airflow.

Em contraste, a instalação utilizando o Docker oferece uma solução mais moderna e eficiente. O Docker é uma ferramenta que utiliza contêineres para isolar aplicações do ambiente subjacente, garantindo que o software funcione de

maneira consistente independentemente do sistema operacional. A configuração via Docker envolve a criação ou utilização de uma imagem Docker pré-configurada para o Airflow. O processo é simples: o arquivo Docker, contendo instruções para configurar o ambiente, é executado para gerar uma imagem Docker, que, por sua vez, cria um contêiner funcional do Airflow. A vantagem dessa abordagem está na sua simplicidade e portabilidade: o ambiente criado pode ser facilmente replicado ou compartilhado, minimizando problemas de compatibilidade.

No entanto, o método recomendado para criar um ambiente local de Airflow de acordo com as práticas recomendadas é através da CLI do Astro. A CLI (Command Line Interface) do Astro é uma ferramenta de código aberto projetada para simplificar a configuração e o gerenciamento de ambientes de Airflow. Após instalar a CLI, o comando astro dev init gera automaticamente um conjunto de arquivos e diretórios essenciais para um ambiente funcional do Airflow. Entre os diretórios gerados estão:

- dags: para armazenar os arquivos DAG, que contêm a definição dos pipelines de dados.

- include: para guardar scripts auxiliares, como consultas SQL ou funções Python.

- plugins: para customizações e extensões do Airflow.

- tests: para realizar testes e validar os pipelines de dados.

Além disso, a CLI do Astro inclui arquivos úteis, como o Dockerfile, que utiliza a imagem de runtime do Astro, e o requirements.txt, que facilita a instalação de pacote Python necessários para os pipelines. Após a inicialização do ambiente, o comando astro dev start configura automaticamente todos os contêineres necessários para a execução do Airflow, como o banco de dados de metadados, o agendador e o servidor web. Isso simplifica o processo e garante que o ambiente esteja alinhado com as melhores práticas.

A interface de usuário do Airflow, acessada via navegador na porta 8080, permite gerenciar e monitorar os pipelines de dados com facilidade. Em empresas, o método com Docker e CLI do Astro é preferido pela eficiência e conformidade com padrões industriais, evitando os desafios e limitações associados à instalação manual. Por isso, essas abordagens modernas são amplamente recomendadas para implementar o Airflow em ambientes de produção ou desenvolvimento.

## 8. [Practice] Quick tour of Airflow UI

A interface de usuário do Airflow é uma ferramenta robusta para gerenciar fluxos de trabalho de dados, com diversas visualizações projetadas para fornecer informações claras sobre a execução de tarefas. O ponto de partida dessa interface é a visualização de DAGs (Directed Acyclic Graphs), que exibe os fluxos de trabalho ativos e inativos configurados no sistema. Nesta visualização, o usuário pode rapidamente verificar o status de execução dos DAGs, incluindo tarefas concluídas com sucesso, falhas ou aquelas em andamento.

Os DAGs podem ser filtrados com base em sua atividade (ativos ou pausados) ou no estado de suas tarefas (em execução ou falhando). Essa funcionalidade é especialmente útil para organizações que gerenciam muitos pipelines de dados e precisam se concentrar na solução de problemas específicos. Cada DAG é apresentado com várias informações organizadas em colunas.

Na primeira coluna, há um controle para pausar ou retomar a execução do DAG. Isso é crucial para interromper pipelines que não devem ser agendados temporariamente. A próxima coluna exibe o nome ou ID do DAG, que é um identificador exclusivo usado para gerenciá-lo. No Airflow 2.9 e versões posteriores, o nome exibido também pode ser personalizado, facilitando a identificação. Logo abaixo do nome, podem ser atribuídas tags para categorizar os DAGs com base em temas ou funcionalidades, o que facilita o gerenciamento em equipes distintas.

Outros campos incluem o proprietário, que geralmente representa o responsável pelo DAG, embora essa informação seja mais relevante em auditorias do que na interface principal. A programação do DAG indica a frequência de execução, como diariamente ou em intervalos específicos, enquanto as colunas de status mostram informações sobre execuções passadas e futuras. O usuário pode visualizar detalhes como a última execução concluída e a próxima execução programada.

Além da visualização principal, outras ferramentas complementam a análise e o controle dos DAGs. Por exemplo, a visualização em grade apresenta uma linha do tempo de execuções do DAG e permite acompanhar o histórico de tarefas específicas, com quadrados coloridos indicando o status de cada instância de tarefa. Essa organização é útil para identificar rapidamente problemas e tarefas bloqueadas. Informações adicionais, como durações máxima e mínima das execuções, também estão disponíveis, ajudando na configuração de limites de tempo.

A visualização em gráfico, por sua vez, é essencial para compreender as dependências entre tarefas dentro de um DAG. Aqui, as tarefas são representadas por nós conectados, com cores indicando seus estados, como concluído ou em

execução. Essa abordagem visual torna fácil identificar a sequência de operações e compreender a lógica do pipeline.

Já o gráfico de Gantt é uma ferramenta analítica poderosa, usada para avaliar a duração e sobreposição das tarefas. Retângulos representam a execução das tarefas, e sua extensão reflete o tempo gasto. Partes em cinza indicam o tempo em fila, enquanto cores como verde ou vermelho mostram se a execução foi bem-sucedida ou falhou. Esse tipo de gráfico é particularmente útil para encontrar gargalos no pipeline e priorizar otimizações.

Na visualização de código, o foco é no script subjacente ao DAG, permitindo verificar e depurar o código diretamente na interface. Também é possível confirmar quando o DAG foi processado pela última vez pelo agendador do Airflow, uma função valiosa para garantir que as alterações feitas no código foram reconhecidas pelo sistema.

Por fim, a visualização de duração da tarefa fornece insights detalhados sobre o desempenho de tarefas ao longo de múltiplas execuções. Comparar a duração entre diferentes execuções pode destacar variações anormais ou identificar padrões que precisam ser otimizados.

Além dessas visualizações principais, a interface do Airflow oferece recursos adicionais, como gerenciamento de variáveis, conexões e conjuntos de dados, que enriquecem a experiência de administração do pipeline. Esses recursos serão explorados conforme a necessidade dos casos de uso específicos, ampliando ainda mais o controle e a análise disponíveis para o usuário.

## 9. [Practice] Quick tour of Airflow CLI

A interface de linha de comando (CLI) do Apache Airflow é uma ferramenta poderosa e flexível para gerenciar pipelines de dados e realizar operações que nem sempre estão disponíveis ou são práticas na interface de usuário. Ela é especialmente útil em cenários onde o acesso à interface gráfica não está disponível, em pipelines automatizados de integração e entrega contínua (CI/CD) e para executar tarefas de forma rápida e eficiente.

Um exemplo típico de uso da CLI em pipelines CI/CD é para testar e implantar fluxos de trabalho de dados em ambientes de produção. O uso de comandos do Airflow diretamente em scripts de automação permite maior controle e integração com outras ferramentas.

O acesso à CLI depende de como o Airflow está configurado. Se o Airflow estiver instalado diretamente na máquina local, basta executar os comandos no terminal. Em ambientes com Docker, é necessário acessar o

contêiner correspondente (por exemplo, usando docker exec). Em setups baseados no Astro, um simples comando astro dev bash permite o acesso.

Entre os comandos mais úteis estão aqueles relacionados ao banco de dados de metadados. O banco de dados do Airflow armazena informações cruciais sobre os DAGs e suas execuções, e comandos como airflow db check verificam se ele está acessível. Manter o banco organizado é fundamental, e o comando airflow db clean move dados antigos para tabelas de arquivo, enquanto airflow db export archived exporta esses dados para arquivos CSV. Para apagar definitivamente os arquivos arquivados, pode-se usar airflow db drop archive. Em instalações manuais, o banco deve ser inicializado com airflow db init, mas esse processo é automático em ambientes gerenciados como Astro ou Docker.

Para gerenciamento de DAGs, a CLI também oferece funcionalidades robustas. Comandos como airflow dags backfill permitem executar ou reexecutar DAGs para intervalos de datas específicos, agilizando a execução de tarefas retroativas sem depender da interface gráfica. Já o comando airflow dags list verifica se um DAG específico foi reconhecido pelo sistema e está disponível na interface de usuário. O comando airflow dags trigger também pode ser usado para iniciar manualmente um DAG, enquanto airflow dags pause e airflow dags unpause gerenciam seu status.

A CLI também é essencial para testar e depurar tarefas individuais. O comando airflow tasks test executa uma tarefa isoladamente, sem verificar dependências ou atualizar metadados, garantindo que ela funcione corretamente antes de ser integrada ao DAG. Essa prática é recomendada sempre que uma nova tarefa é adicionada, para evitar problemas em execuções completas.

Por fim, a CLI oferece comandos para diversas outras operações, desde o gerenciamento de conexões e variáveis até a manipulação de logs e conjuntos de dados. Uma das melhores maneiras de explorar suas possibilidades é utilizar o comando de ajuda (airflow --help) ou consultar a documentação oficial para obter uma lista completa de comandos e suas descrições.

## 10. DAG scheduling: the basics

Na programação de DAGs no Apache Airflow, a estrutura básica envolve dois parâmetros fundamentais: a data de início e o cronograma. A data de início define quando o pipeline de dados começará a ser executado, enquanto o cronograma determina a frequência de execução desse pipeline.

A data de início é o primeiro ponto de partida. Se você configurar uma data de início, por exemplo, para 1º de janeiro de 2024, isso significa que o DAG só começará a ser executado a partir dessa data. Após definir a data de início, o próximo passo é configurar o cronograma. O cronograma é um parâmetro

essencial para a periodicidade do pipeline e pode ser definido através de uma expressão cron. O formato cron é uma cadeia de caracteres com campos que especificam o minuto, hora, dia do mês, mês, dia da semana e ano (em alguns casos, um campo adicional de ano pode ser utilizado).

No entanto, o Airflow simplifica esse processo oferecendo predefinições de cronograma que ajudam os usuários a configurar de maneira mais amigável as execuções. Por exemplo, se você deseja que o DAG seja executado diariamente à meia-noite, pode usar a predefinição "@daily", sem precisar se preocupar em escrever a expressão cron manualmente.

Uma vez configurados esses parâmetros, o DAG entra em execução com base no cronograma definido. Por exemplo, se a data de início for 1º de janeiro de 2024 e o cronograma for diário, o Airflow aguardará até a meia-noite de 2 de janeiro de 2024 para executar a primeira instância do DAG, que irá processar os dados referentes ao intervalo entre 1º e 2 de janeiro. O processo se repetirá a cada dia, gerando uma nova instância do DAG que abrange o intervalo do dia anterior.

Cada execução de um DAG é chamada de "dag run", e cada "dag run" está associado a um intervalo de dados específico. O Airflow cuida da execução de cada instância do DAG de acordo com o cronograma, mas é importante entender o comportamento quando uma execução ainda não foi concluída enquanto uma nova está programada para começar. Por padrão, o Airflow permite que as execuções sejam sobrepostas, ou seja, uma nova execução pode começar mesmo que a anterior não tenha sido finalizada. Esse comportamento pode ser problemático em alguns cenários, por isso existe um parâmetro chamado max_active_runs que limita o número de execuções simultâneas para um determinado DAG. Se o parâmetro max_active_runs for configurado como 1, o Airflow garantirá que apenas uma execução do DAG aconteça por vez, forçando a segunda execução a aguardar a conclusão da primeira.

Além disso, existe o parâmetro max_active_runs_per_dag, que permite controlar a quantidade de execuções simultâneas de DAGs dentro de uma instância do Airflow, podendo ser útil quando se quer garantir que todos os DAGs tenham uma execução por vez.

## 11. Backfill and Catchup

Os conceitos de backfilling e catch up no Apache Airflow estão diretamente relacionados ao comportamento de recuperação de execuções de DAGs em cenários de agendamento.

**Catch Up**

O catch up refere-se ao comportamento padrão do Airflow, onde, caso o DAG seja pausado ou não tenha sido executado por algum período, ele tentará "preencher" as execuções perdidas. Isso significa que, se o seu DAG tiver uma data de início em 2 de janeiro de 2024 e a programação for diária, mas o DAG for pausado até 5 de janeiro, ao despausar, o Airflow automaticamente tentará executar as três instâncias faltantes para os dias 2, 3 e 4 de janeiro. Isso acontece para "recuperar" os dias em que o DAG não foi executado.

Este comportamento pode ser útil em algumas situações, mas pode levar a sobrecarga do sistema, especialmente se o intervalo de tempo sem execução for grande ou se o DAG for pesado. Para evitar isso, o Airflow permite que você desative o catch up. Ao definir o parâmetro catchup=False no DAG, você impede que o Airflow execute automaticamente as instâncias faltantes. Assim, ao despausar o DAG no dia 5 de janeiro, apenas a execução mais recente (para esse dia) será executada, ignorando as execuções anteriores.

**Backfilling**

O backfilling é um processo manual que permite preencher as execuções de DAG que estão ausentes. Se, por exemplo, você originalmente configurou o DAG para iniciar em 2 de janeiro de 2024, mas, ao perceber um erro, deseja que ele comece em 1º de janeiro de 2024, você pode usar o mecanismo de backfilling para executar manualmente a instância do DAG que falta para o dia 1 de janeiro. O backfilling pode ser feito através da interface de linha de comando ou da interface do usuário.

No backfilling, o Airflow não executa automaticamente as instâncias faltantes, como faz com o catch up. Ele requer uma ação explícita do usuário para iniciar as execuções para as datas anteriores. Isso é útil quando você percebe que algo foi configurado de forma errada e quer ajustar o histórico de execuções do DAG para preencher lacunas ou corrigir erros.

**Recomendações**

- Desative o catch up: Como prática recomendada, muitos usuários preferem desativar o catch up, pois ele pode gerar uma quantidade excessiva de execuções, o que pode levar a sobrecarga nos recursos e confusão na visualização e no gerenciamento das execuções.

- Use backfilling manualmente: O backfilling deve ser usado com cuidado e apenas quando necessário. Ele permite corrigir datas de execução ausentes, mas não deve ser uma prática automática ou frequente.

Esses dois conceitos são essenciais para garantir que os pipelines de dados sejam executados corretamente, mesmo que ocorram atrasos ou ajustes nas configurações de agendamento. A compreensão de como e quando usar catch up e backfilling é fundamental para manter o controle e a eficiência no gerenciamento de DAGs no Airflow.

## 12. [Practice] Everything you need to backfill DAGs with Airflow

Backfilling, no contexto de pipelines de dados e orquestração, refere-se ao processo de executar novamente um ou mais DAGs (Directed Acyclic Graphs) ou tarefas individuais desses DAGs em intervalos de tempo anteriores. Essa prática é essencial quando há mudanças no fluxo, falhas em execuções passadas ou necessidades específicas de reprocessamento de dados históricos. A seguir, detalha-se o conceito, casos de uso, mecanismos e boas práticas associadas ao backfilling.

Quando um DAG ou tarefa sofre alterações, falha ou precisa ser executado retroativamente, o backfilling permite recuperar os intervalos que ainda não foram processados. Por exemplo, ao adicionar uma nova tarefa a um DAG que já estava em execução, pode ser necessário que essa tarefa seja aplicada retroativamente aos dias anteriores. Outro exemplo comum é reexecutar DAGs que falharam em certos intervalos, o que é usual em ambientes de produção.

O backfilling no Apache Airflow, uma plataforma de orquestração de workflows, é frequentemente gerido por meio de sua CLI (Command Line Interface), em vez da interface gráfica do usuário. O comando básico utilizado para este propósito é o airflow dags backfill, que permite configurar intervalos de tempo específicos para execução retroativa de DAGs ou tarefas. Há várias opções que controlam como o backfill é realizado, como a possibilidade de reexecutar tarefas falhas, redefinir execuções de DAGs, e até mesmo limitar tentativas automáticas.

Casos de Uso do Backfilling:

1. Criação de novos DAGs: Quando um novo DAG é implementado e precisa processar dados de períodos passados.

2. Execuções com falhas: DAGs ou tarefas que falharam precisam ser reexecutados para garantir que os dados sejam processados corretamente.

3. Alterações no DAG: Modificações no código de um DAG, como a adição de novas tarefas ou alterações em dependências, exigem backfilling para aplicar essas mudanças a execuções anteriores.

4. Execução seletiva: Quando apenas um subconjunto de tarefas ou intervalos específicos deve ser reprocessado.

Ao configurar o backfilling, é crucial considerar o estado atual do DAG e das tarefas. Por padrão, DAGs pausados podem ser preenchidos sem necessidade de ativação, e o comando de backfill pode ser configurado para respeitar ou desconsiderar limites de tentativas definidos para tarefas. A reexecução de tarefas e DAGs requer o uso de opções específicas como --rerun-failed-tasks (para reexecutar tarefas falhas) e --reset-dagruns (para redefinir DAGs).

O desempenho do backfilling pode ser otimizado de diversas maneiras:

- Ordem de execução: Por padrão, o Airflow processa os DAGs do mais antigo para o mais recente, mas isso pode ser alterado com a opção --run-backwards.

- Clonagem de DAGs: Uma prática recomendada é criar um clone do DAG original exclusivamente para backfilling. Isso evita interferências no DAG principal, que pode continuar operando normalmente em seus intervalos futuros.

- Uso de pools: DAGs que executam backfilling devem ser associados a pools específicos para evitar a saturação de recursos compartilhados. No Airflow, pools permitem limitar o número de tarefas em execução simultânea.

- Configuração de recursos: Em ambientes com alta demanda computacional, como ao usar executores Celery ou Kubernetes, é possível isolar os recursos destinados ao backfilling, evitando impacto nos DAGs regulares.

Por fim, é possível utilizar configurações avançadas, como filtros baseados em padrões de ID de DAGs para backfill de múltiplos DAGs simultaneamente, ou definir configurações específicas para execuções durante o processo. Esses recursos são especialmente úteis para equipes que gerenciam grandes volumes de workflows.

O backfilling é uma ferramenta poderosa e indispensável para garantir a consistência de dados processados em sistemas de orquestração como o Airflow. Contudo, sua implementação deve ser planejada cuidadosamente para evitar impactos indesejados na execução de DAGs em produção.

## 13. [Practice] Ensure you can backfill your data pipelines

O conceito de backfill no contexto de pipelines de dados, especialmente no Airflow, refere-se ao processo de executar um DAG para períodos de tempo passados. Ele é essencial em situações como a adição de novas tarefas a um pipeline existente ou quando há necessidade de corrigir ou preencher lacunas nos dados. No entanto, um aspecto fundamental para o sucesso do backfill é garantir que as tarefas sejam idempotentes.

Idempotência, nesse contexto, implica que a execução repetida de uma mesma tarefa sempre produzirá o mesmo resultado, sem gerar efeitos colaterais indesejados, como a duplicação de dados ou inconsistências. Essa característica é especialmente importante em tarefas que realizam operações de gravação em bancos de dados ou interagem com sistemas externos.

Um exemplo de problema de idempotência ocorre quando uma tarefa usa a função NOW() em sua lógica de processamento. NOW() retorna o timestamp do momento em que a consulta SQL é executada, mas isso se torna problemático em situações de backfill. Por exemplo, se um DAG precisa recuperar dados históricos, como o número de vendas diárias, o uso de NOW() resultaria sempre na data atual, ignorando o contexto temporal do backfill. Essa abordagem gera duplicação de dados ou confusão nos pipelines subsequentes, visto que a tarefa continua buscando informações do presente, mesmo quando deveria processar dados históricos.

A solução para esse problema é utilizar variáveis que respeitem o contexto temporal de cada execução do DAG. No Airflow, a variável {{ data_interval_end }} pode ser empregada para identificar o término do intervalo de dados da execução em questão. Subtraindo um dia dessa variável, é possível obter a data correta para recuperar os dados históricos, garantindo que cada execução do DAG seja específica e precisa.

Além disso, ao realizar operações de escrita no banco de dados, é recomendável utilizar comandos como merge, upsert (inserção com atualização) ou substituição, em vez de simplesmente acrescentar (insert). Isso evita a duplicação de dados, especialmente em cenários de reexecução do pipeline. Essa prática reforça a idempotência, assegurando que os dados resultantes sejam consistentes, mesmo após múltiplas execuções.

A idempotência, portanto, não é apenas uma questão técnica, mas uma prática essencial para garantir a integridade e confiabilidade dos pipelines de dados, permitindo que eles sejam reexecutados, corrigidos e ajustados sem comprometer a qualidade dos resultados.

## 14. Dealing with timezones in Airflow

Fusos horários desempenham um papel crucial no desenvolvimento e operação de sistemas computacionais distribuídos, incluindo plataformas de orquestração de dados como o Apache Airflow. Esses fusos representam divisões geográficas onde se adota um horário padrão uniforme, comumente definido em relação ao Tempo Universal Coordenado (UTC). O UTC, amplamente aceito em sistemas computacionais, elimina ambiguidades temporais causadas por deslocamentos geográficos ou ajustes sazonais, como o horário de verão. Por

exemplo, cidades como Paris e Nova York possuem deslocamentos de UTC+2 e UTC-4, respectivamente, dependendo do período do ano.

No contexto de desenvolvimento com Python, é importante diferenciar entre objetos datetime ingênuos e conscientes. Um objeto ingênuo não contém informações sobre o fuso horário, o que pode levar a interpretações errôneas e inconsistências, especialmente em sistemas distribuídos ou quando há mudanças no horário local. Por outro lado, objetos conscientes possuem informações explícitas sobre o fuso horário, permitindo cálculos precisos e consistentes. O Airflow, ao empregar a biblioteca pendulum, que suporta nativamente manipulações de data e hora com fusos horários, ajuda a mitigar os problemas comuns associados a fusos horários. No entanto, é importante destacar que, por padrão, o Airflow armazena e processa todas as informações temporais em UTC, independentemente do fuso horário local definido.

No agendamento de tarefas com o Airflow, é possível usar expressões cron ou objetos timedelta, que diferem significativamente em como tratam mudanças temporais, como o horário de verão. Uma expressão cron define horários fixos para a execução, ajustando-se automaticamente para manter a consistência no horário local. Por exemplo, um DAG programado para rodar às 2h continuará a ser executado às 2h no horário local, mesmo com ajustes de horário. Em contrapartida, ao usar um timedelta, o Airflow mantém intervalos absolutos entre as execuções, independentemente do horário local ajustado. Isso significa que, durante a transição para o horário de verão, uma execução prevista para 2h antes do ajuste será realizada às 3h após o ajuste, preservando o intervalo absoluto.

Essa diferença torna-se evidente durante mudanças sazonais. Enquanto uma expressão cron pode omitir uma execução em função do ajuste de horário, um agendamento baseado em timedelta não sofrerá tal impacto, garantindo que os intervalos temporais sejam mantidos. A escolha entre essas abordagens deve ser orientada pelas necessidades específicas do pipeline. Processos sensíveis ao horário local, como relatórios diários, podem se beneficiar de expressões cron, enquanto pipelines que exigem intervalos regulares são melhor atendidos com timedelta.

A configuração de fusos horários no Airflow também permite ajustes para atender a requisitos específicos de exibição. Embora os dados sejam sempre processados em UTC, é possível exibir informações em fusos horários locais, configurando o parâmetro default_timezone. Essa flexibilidade evita que alterações no fuso horário afetem o processamento interno, mas proporciona conveniência ao apresentar resultados para usuários finais em contextos locais.

Conceitualmente, trabalhar com fusos horários exige uma abordagem rigorosa e deliberada, especialmente em sistemas distribuídos. O uso de objetos

conscientes e a padronização no UTC são práticas fundamentais para mitigar problemas de consistência e confiabilidade em plataformas como o Airflow. Além disso, entender as nuances entre agendamentos fixos e intervalos absolutos é essencial para escolher estratégias que atendam melhor às necessidades operacionais. O estudo de fusos horários e suas implicações computacionais é, portanto, uma área fundamental para a engenharia de dados moderna, dada a natureza global e interconectada dos sistemas contemporâneos.

## 15. How to make your tasks dependent

A gestão de dependências entre tarefas em pipelines orquestrados com Apache Airflow é um aspecto fundamental para garantir a execução correta e ordenada das operações. As dependências padrão, que são definidas usando funções como set_upstream e set_downstream ou operadores bitshift do Python, geralmente se aplicam ao DAGRun atual, ou seja, ao conjunto de execuções do DAG em um intervalo de tempo específico. No entanto, existem casos em que é necessário estabelecer dependências entre instâncias de tarefas em diferentes execuções de DAGRuns consecutivos. Para atender a essas necessidades, o Airflow oferece os parâmetros depends_on_past e wait_for_downstream.

O parâmetro depends_on_past permite que uma tarefa em um DAGRun dependa explicitamente de sua própria instância no DAGRun anterior. Isso significa que a tarefa no DAGRun atual só será executada se tiver sido concluída com sucesso no DAGRun anterior. Por exemplo, em um pipeline onde a tarefa "B" falhou no segundo DAGRun, o uso de depends_on_past impede que "B" seja executada no terceiro DAGRun até que a falha seja resolvida. Isso garante consistência e evita a propagação de erros ao longo das execuções. É importante observar que, na primeira execução de um DAGRun, depends_on_past não tem efeito, pois não há um histórico anterior.

Outro parâmetro complementar é o wait_for_downstream. Enquanto depends_on_past foca na continuidade da própria tarefa ao longo dos DAGRuns, wait_for_downstream estende a lógica para considerar o estado das tarefas imediatamente dependentes. Esse parâmetro impede que uma tarefa em um DAGRun atual seja executada até que todas as tarefas subsequentes do DAGRun anterior sejam concluídas com sucesso. Em pipelines concorrentes, isso evita situações em que múltiplos DAGRuns estejam acessando ou processando o mesmo recurso simultaneamente, o que poderia causar inconsistências nos dados ou em recursos compartilhados.

Esses dois parâmetros são configurados no nível da tarefa, mas podem ser aplicados globalmente ao DAG por meio do dicionário default_args. Por padrão, ambos estão desativados (False), exigindo configuração explícita quando necessário.

A utilização desses mecanismos de controle de dependência é especialmente útil em cenários de pipelines sequenciais ou em situações onde o processamento incremental de dados depende fortemente da execução bem-sucedida de passos anteriores. Eles são ferramentas essenciais para evitar falhas em cascata e garantir que as execuções do DAG respeitem a lógica de dependências definida, não apenas dentro de um único ciclo de execução, mas também ao longo de múltiplos ciclos.

## 16. How to structure your DAG folder

A organização da pasta de DAGs no Apache Airflow é essencial para a manutenção e a eficiência, especialmente à medida que o número de DAGs e suas dependências aumenta. A pasta de DAGs, configurada pelo parâmetro dags_folder no arquivo de configuração do Airflow (airflow.cfg), é o local onde todos os arquivos Python correspondentes aos DAGs são armazenados. Por padrão, essa pasta é criada dentro do diretório especificado pela variável de ambiente AIRFLOW_HOME, mas o caminho deve ser absoluto.

Uma abordagem para manter essa estrutura organizada é o uso de arquivos compactados no formato ZIP. Essa técnica permite agrupar um DAG e suas dependências em um único arquivo, facilitando a portabilidade e o gerenciamento. O Airflow é capaz de carregar DAGs diretamente de arquivos ZIP, desde que os DAGs estejam localizados na raiz do arquivo compactado. Essa funcionalidade é especialmente útil em cenários onde módulos externos ou pacotes personalizados precisam ser incorporados ao DAG. Além disso, a combinação com ferramentas como *virtualenv* e o uso do pip para instalação de dependências específicas no ambiente virtual possibilita maior flexibilidade. Após a criação do ambiente, basta compactá-lo e movê-lo para a pasta de DAGs.

Outra abordagem para gerenciar DAGs de maneira eficiente é o uso da classe DagBag. O DagBag permite carregar DAGs de diferentes pastas, independentemente do diretório principal configurado no airflow.cfg. Essa funcionalidade é útil quando se deseja dividir os DAGs em estruturas mais organizadas ou mesmo mantê-los em repositórios separados. No entanto, essa técnica apresenta limitações: erros em DAGs carregados de outras pastas não são exibidos na interface web do Airflow, e tais DAGs não aparecem na saída do comando airflow list_dags. Apesar disso, os benefícios de segmentar os DAGs em diferentes locais podem superar essas restrições em muitos casos.

Além disso, o uso de um arquivo .airflowignore, semelhante ao .gitignore utilizado no Git, é uma prática recomendada para evitar que arquivos ou pastas desnecessárias sejam processados pelo Airflow. Cada linha do arquivo pode conter padrões definidos como expressões regulares para excluir arquivos ou diretórios que não contenham DAGs. Isso reduz o consumo desnecessário de

recursos, já que o Airflow realiza verificações periódicas na pasta de DAGs para identificar novas entradas.

Essas estratégias não apenas ajudam na organização e clareza da estrutura dos arquivos, mas também otimizam o desempenho do Airflow em ambientes com grande número de DAGs. Apesar de não haver uma prática universal para a organização da pasta de DAGs, é essencial manter a clareza na hierarquia de arquivos, adaptando as técnicas às necessidades específicas do projeto e da equipe.

## 17. How to deal with failures in your DAGs

Em um ambiente de engenharia, falhas são inevitáveis, e lidar com elas é uma parte essencial do trabalho, especialmente em sistemas de orquestração como o Apache Airflow. O Airflow oferece diversas funcionalidades para monitorar e reagir a problemas tanto no nível dos DAGs quanto no nível das tarefas. Essas ferramentas são cruciais para garantir que os pipelines de dados continuem funcionando conforme esperado, mesmo diante de imprevistos.

No nível dos DAGs, existem parâmetros que permitem configurar a detecção e o tratamento de falhas de execução. O parâmetro dagrun_timeout estabelece um tempo limite máximo para que um DagRun esteja ativo. Caso o tempo limite seja atingido, o DagRun é interrompido, permitindo a criação de novos. Isso é particularmente útil quando combinado com o parâmetro max_active_runs, que controla o número máximo de execuções simultâneas de um DAG. Essa configuração ajuda a gerenciar a capacidade de processamento do sistema e a evitar congestionamentos, especialmente durante execuções em massa, como reprocessamento de dados históricos.

Outros parâmetros importantes incluem on_failure_callback e on_success_callback, que permitem a execução de funções específicas ao final de um DagRun, seja em caso de sucesso ou falha. Esses callbacks fornecem um mecanismo flexível para reagir dinamicamente ao estado dos DAGs, utilizando informações contextuais do DagRun e de suas tarefas associadas.

No nível das tarefas, o Airflow fornece uma série de opções para alertas e controle de falhas. A configuração de alertas por e-mail, habilitada pelos parâmetros email_on_failure e email_on_retry, permite notificações automáticas em caso de falhas ou novas tentativas. Para que esses e-mails sejam enviados, é necessário configurar um servidor SMTP no arquivo de configuração do Airflow.

Além disso, o Airflow possibilita gerenciar novas tentativas de execução por meio de parâmetros como retries, que define o número máximo de tentativas, e retry_delay, que especifica o intervalo entre essas tentativas. O uso do retry_exponential_backoff introduz atrasos progressivamente maiores entre

tentativas subsequentes, enquanto max_retry_delay estabelece um limite para esses atrasos. Juntos, esses parâmetros oferecem um controle refinado sobre o comportamento de repetição, otimizando os recursos e minimizando o impacto de falhas transitórias.

O tempo máximo permitido para a execução de uma tarefa pode ser configurado com o parâmetro execution_timeout. Caso uma tarefa ultrapasse esse limite, ela será marcada como falha, evitando que processos travados prejudiquem o fluxo do pipeline. Assim como nos DAGs, é possível especificar callbacks no nível das tarefas, como on_failure_callback e on_success_callback, permitindo reações específicas e automatizadas ao estado de execução.

Essas ferramentas demonstram o compromisso do Airflow com a resiliência operacional, fornecendo aos engenheiros meios de monitorar, detectar e reagir a falhas de maneira eficaz. Configurações adequadas desses parâmetros, baseadas em análises das execuções anteriores e em métricas de desempenho, podem reduzir significativamente o impacto de problemas, garantindo a confiabilidade e a consistência dos pipelines de dados.

## 18. How to test your DAGs

A prática de testes unitários é fundamental em engenharia de software para assegurar que mudanças no código não introduzam erros inesperados, especialmente em ambientes colaborativos. No contexto do Apache Airflow, os testes são indispensáveis para validar os DAGs antes de sua implantação em produção. Para isso, utilizamos o Pytest, uma ferramenta de teste amplamente adotada pela comunidade Python por sua simplicidade e flexibilidade.

No Airflow, os testes são organizados em categorias que abrangem diferentes aspectos do pipeline. A primeira delas, a validação de DAGs, tem como objetivo garantir que os DAGs sejam tecnicamente válidos. Isso inclui verificar se não há ciclos, o que inviabilizaria sua execução, e certificar-se de que argumentos padrão, como e-mails de alerta, estão configurados corretamente. Esses testes são comuns a todos os DAGs e evitam erros de configuração que possam interromper a execução.

Outra categoria importante é a de definição de DAGs e pipelines, que foca na estrutura do DAG. Esses testes garantem que mudanças realizadas no código refletem as alterações intencionais e verificam aspectos como o número de tarefas, suas dependências e o tipo de cada uma delas. Essa análise é específica para cada DAG e complementa a validação técnica.

Os testes unitários concentram-se nos operadores e sensores personalizados criados pelo desenvolvedor. Como o Airflow atua como um orquestrador e não realiza o processamento de dados diretamente, a lógica de

processamento deve ser implementada em ferramentas externas, como Spark ou bancos de dados, que têm seus próprios testes unitários. No Airflow, esses testes avaliam os componentes individuais criados internamente para assegurar que funcionam como esperado.

Os testes de integração verificam a interação entre as tarefas de um DAG, garantindo que elas cooperem adequadamente. Por exemplo, asseguram que uma tarefa pode buscar os dados necessários para sua execução ou que consegue se comunicar com sistemas externos, como bancos de dados. Para esses testes, é comum configurar um ambiente de "teste" que simula parcialmente o ambiente de produção.

Por fim, os testes de pipeline de ponta a ponta avaliam o funcionamento completo de um DAG, da primeira à última tarefa, para verificar se ele produz os resultados esperados e mantém um desempenho aceitável. Esses testes utilizam uma cópia maior dos dados de produção em um ambiente de "aceitação", simulando condições reais antes da implantação final.

A estruturação desses testes requer ambientes específicos. No ambiente de desenvolvimento, utilizam-se pequenos conjuntos de dados fictícios para validar a funcionalidade básica dos DAGs. No ambiente de "teste", os dados são mais próximos do real, permitindo a execução de testes de integração. O ambiente de "aceitação" utiliza uma cópia completa dos dados de produção para validar o pipeline como um todo e assegurar que ele atende aos requisitos antes de ser liberado. Finalmente, o ambiente de produção é onde os DAGs são disponibilizados para os usuários finais.

A integração com CI/CD automatiza o ciclo de testes e implantação. Alterações no código começam em ramificações de "recurso" e, após revisões e validações, são mescladas à ramificação de desenvolvimento. Os testes iniciais garantem a estabilidade do código antes que ele seja integrado à ramificação de "teste", onde os testes de integração são realizados. Passando nessa etapa, o código segue para a ramificação de "aceitação", onde é validado pelo proprietário do produto e pela equipe. Somente após uma última revisão é que o código é implantado na produção.

Esse processo, embora não elimine completamente a possibilidade de falhas, reduz significativamente os riscos, garantindo que os DAGs sejam confiáveis e cumpram suas funções no ambiente de produção. A combinação de testes abrangentes, ambientes bem estruturados e automação com CI/CD oferece uma base sólida para o desenvolvimento e a manutenção de pipelines de dados robustos.