DATA ENGINEERING

Building Reliable Data Pipelines with dbt and Airflow

Orchestration, testing and incremental models for production

Houssam Kodad

PDF · DATAFORGE BOOKS

© 2026 DataForge Books. All rights reserved.

“Building Reliable Data Pipelines with dbt and Airflow” and this sample are published by DataForge Books, operated by Houssam Kodad, France. The author asserts the moral right to be identified as the author of this work.

This document is a free promotional sample containing the opening chapter of the full title. It is provided for evaluation only. No part of this publication may be reproduced, distributed, or transmitted in any form or by any means, or stored in a database or retrieval system, without the prior written permission of the publisher, except as permitted by applicable copyright law.

The information in this book is provided on an “as is” basis for general educational purposes. While every effort has been made to ensure accuracy, the publisher and author assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

Questions about this sample or the full edition: **support@dataforgebooks.com**

CONTENTS

Table of Contents

CHAPTER 01

The Warehouse-Centric Pipeline

Every data team eventually arrives at the same realisation: the hard part of a pipeline is not moving the data, it is trusting it. A query that ran yesterday returns different numbers today; a model breaks because an upstream column was renamed; a dashboard quietly shows last week's figures because a job failed at 3 a.m. and nobody noticed. The mechanics of copying rows from one place to another are largely solved. What is not solved — what teams pay for in late nights and lost credibility — is making the result correct, observable and safe to change.

dbt and Airflow have become the default answer to that problem, and for good reason. Between them they cover the two hardest parts of a modern data platform: transforming raw data into trustworthy models, and orchestrating that work reliably across systems and schedules. Used well, they turn a fragile collection of scripts into something that behaves like software — version-controlled, tested, reviewed and recoverable.

But "used well" is doing a lot of work in that sentence. It is entirely possible to adopt both tools and still end up with a platform nobody trusts: models with no tests, a directed acyclic graph that fails silently at dawn, transformations whose logic lives only in the head of whoever wrote them. The difference between the two outcomes is not the tooling. It is a set of patterns and disciplines this book exists to teach.

This first chapter sets the foundation for everything that follows. We will look at why the industry moved from ETL to ELT and what that shift actually changed; the distinct jobs that dbt and Airflow each do well, and how they fit together without overlapping; the layered architecture that keeps a warehouse navigable as it grows; and the reference platform we will build toward, piece by piece, over the coming chapters. By the end you will have the mental model the rest of the book assumes.

Why ELT Replaced ETL

For two decades, the standard shape of a data pipeline was ETL: extract from the source, transform the data in a dedicated processing tier, then load the finished result into the warehouse. The warehouse received only clean, modelled tables. Transformation happened somewhere else — a Java or Informatica job, a Python script, a proprietary tool — and the raw data was usually discarded once it had been processed.

That design was a rational response to scarcity. Warehouse storage was expensive and compute was a fixed, precious resource you protected from heavy transformation work. Doing the transformation

outside the warehouse kept the warehouse lean. The cost was steep but accepted: transformation logic lived in opaque jobs, the raw data was gone so you could not reprocess history, and debugging meant reading code in a tool your analysts could not touch.

The cloud data warehouse inverted those economics. Storage became cheap enough that keeping every raw record indefinitely is a rounding error on the bill. Compute became elastic, billed by the second, and — crucially — the warehouse itself became the single fastest transformation engine most teams have access to. Once that was true, transforming data anywhere else stopped making sense.

ELT is the pattern that embraces the new economics. You extract from sources and load the raw data into the warehouse first, untouched, then transform it in place using SQL. The benefits compound. Raw data is preserved, so when a definition changes you can rebuild history rather than apologise for it. Transformations become version-controlled SQL that anyone fluent in the business can read and review. And the warehouse's query planner — years of engineering you get for free — does the heavy lifting.

```
-- ELT in practice: load raw, then transform in the warehouse with SQL.
-- models/staging/stg_orders.sql
select
    id                      as order_id,
    customer_id,
    cast(amount as numeric)  as amount_eur,
    lower(status)            as status,
    created_at::timestamp    as ordered_at
from {{ source('shop', 'orders') }}
where status != 'test'
```

Where dbt Fits: Transformation as Software

dbt (data build tool) is what turns "transform with SQL in the warehouse" from a good idea into a disciplined engineering practice. At its core it does something deceptively simple: it lets you write each transformation as a SELECT statement in a file, and it takes care of turning that SELECT into a table or view in the right order. You never write CREATE TABLE or worry about dependencies by hand.

The power comes from what that unlocks. Because every model is a file, your entire transformation layer lives in version control, with pull requests, code review and history. Because dbt understands the references between models, it builds a dependency graph automatically and runs them in the correct sequence. And because dbt has first-class testing, you can assert things about your data and have the build fail loudly when they stop being true.

Models reference each other through the ref function rather than hard-coded table names. This is the keystone feature: it is how dbt knows the dependency graph, how it builds objects in order, and how the same project can run against a developer's sandbox and against production by changing one setting. Write ref, and lineage, ordering and environment-safety come for free.

```
-- models/marts/fct_orders.sql
with orders as (
    select * from {{ ref('stg_orders') }}
),
customers as (
    select * from {{ ref('stg_customers') }}
)
select
    o.order_id,
    o.ordered_at,
    c.country,
    o.amount_eur
from orders o
left join customers c using (customer_id)
```

Where Airflow Fits: Orchestration as Code

dbt is excellent at one thing: building your warehouse models correctly and in order. What it deliberately does not do is decide when to run, what to do when an upstream extraction is late, how to retry a transient failure, or how to coordinate work that spans systems beyond the warehouse. That is orchestration, and it is Airflow's job.

Airflow lets you express a workflow as a directed acyclic graph of tasks, defined in Python. Each task is a unit of work — extract this source, run that dbt build, refresh a dashboard cache — and the edges between tasks describe their dependencies. Airflow then schedules the graph, runs tasks in dependency order, retries the ones that fail according to rules you set, and gives you a UI and a history of every run.

Defining pipelines in code, rather than clicking them together in a GUI, is the same bet dbt makes about transformations: that workflows deserve version control, review and testing as much as application code does. A DAG is reviewed in a pull request like anything else, and the same definition runs identically in every environment.

```
from airflow import DAG
from airflow.operators.bash import BashOperator
from pendulum import datetime

with DAG(
    dag_id="analytics",
    schedule="0 6 * * *",          # every day at 06:00
    start_date=datetime(2026, 1, 1),
    catchup=False,
) as dag:
    extract_load = BashOperator(
        task_id="extract_load",
        bash_command="airbyte sync --connection shop_to_warehouse",
    )
    dbt_build = BashOperator(
        task_id="dbt_build",
        bash_command="dbt build --target prod",
    )
    extract_load >> dbt_build
```

How the Two Work Together

It is tempting to read the last two sections as a competition and ask which tool wins. They are not competitors; they answer different questions. dbt knows how your tables depend on each other. Airflow knows how your whole platform's jobs depend on each other. dbt builds the warehouse; Airflow makes sure dbt runs after the data has landed, retries it if the warehouse hiccups, and raises an alarm if freshness slips.

In the simplest integration, an Airflow task shells out to the dbt command-line interface, as in the DAG above. This works and many teams run exactly this in production for years. Its limitation is granularity: to Airflow the entire dbt build is a single opaque box, so when one model out of three hundred fails, the whole task fails and you lose dbt's fine-grained picture of what succeeded.

More sophisticated integrations render the dbt graph into native Airflow tasks, so each model becomes its own task with its own retries, logs and lineage. We will cover both approaches in depth later. For now, hold the division of labour clearly: dbt owns the shape of the warehouse, Airflow owns the timing and reliability of the work that builds it.

The Layered Warehouse: Staging, Intermediate, Marts

A pile of models with no structure becomes unmaintainable just as quickly as a pile of undocumented scripts. The convention that keeps a dbt project navigable is to organise models into layers, each with a clear responsibility, and to let data flow strictly in one direction through them.

The staging layer sits directly on top of raw sources. Staging models do light, mechanical work: renaming columns to a consistent convention, casting types, and lightly cleaning values. There is one staging model per source table, and they contain no business logic — their only job is to give the rest of the project a clean, predictable foundation to build on.

The intermediate layer is where business logic lives: joining staged models together, computing derived fields, and reshaping data into the concepts the business actually uses. The marts layer is the public interface — the fact and dimension tables analysts query directly, named in the language of the business. Keeping these layers distinct means a change to a source ripples through staging without rewriting a mart, and an analyst never has to read raw source quirks to understand a number.

```
models/
├── staging/        -- 1:1 with sources; rename, cast, clean
│   ├── stg_orders.sql
│   └── stg_customers.sql
├── intermediate/   -- business logic; joins and derivations
│   └── int_orders_enriched.sql
└── marts/          -- the tables analysts query
    ├── fct_orders.sql
    └── dim_customers.sql
```

Tests, Freshness and Contracts

The single habit that separates a warehouse people trust from one they quietly route around is testing. dbt makes data tests cheap enough that there is no excuse not to write them. You declare expectations about a model — this key is unique, this column is never null, this status is always one of a known set — and dbt checks them every time it builds.

Tests are declared in YAML alongside the models they describe, which keeps the expectation next to the thing it constrains. Out of the box you get tests for uniqueness, non-null values, accepted value sets and referential integrity; packages add dozens more, and you can write your own as SQL. A test that fails stops the build, so bad data is caught before it reaches a dashboard rather than after a stakeholder reports it.

Two related ideas extend testing further. Source freshness checks assert that raw data has actually arrived recently, catching the silent failure where an upstream feed stops without erroring. Model contracts let you pin a model's column names and types so a breaking change is rejected at build time rather than discovered downstream. Together, tests, freshness and contracts are how a platform earns the right to be trusted.

```
# models/staging/_staging.yml
models:
  - name: stg_orders
    columns:
      - name: order_id
        tests: [unique, not_null]
      - name: status
        tests:
          - accepted_values:
              values: ['placed', 'shipped', 'returned']

sources:
  - name: shop
    freshness:
      warn_after:  {count: 12, period: hour}
      error_after: {count: 24, period: hour}
    tables:
      - name: orders
```

A Reference Architecture for the Book

Throughout this book we build toward one coherent platform, and it is worth seeing the whole shape now even though we will assemble it gradually. Sources — application databases, third-party APIs, event streams — are loaded raw into the warehouse by an ingestion tool. Nothing is transformed on the way in; the raw schema is an immutable record of what arrived.

From there, dbt promotes the raw data through the staging, intermediate and marts layers we just described, with tests and freshness checks gating each step. Airflow sits above all of it: it triggers ingestion, runs the dbt build once the data has landed, handles retries and backfills when things go wrong, and emits the run metadata we will later use for observability and alerting.

You do not need the exact tools we use to follow along — the patterns transfer to whatever warehouse and ingestion stack you run. What matters is the shape: raw data preserved, transformations layered and tested, orchestration that is honest about failure. Every chapter that follows fills in one part of this picture in production-grade detail.

What You Will Need to Follow Along

The examples in this book assume access to a cloud data warehouse — the patterns are identical whichever one you use — and a working dbt installation connected to it. Where Airflow appears, a local instance is enough to run every example; you do not need a production cluster to learn the concepts.

You do not need to be an expert in any of these tools to start. A working knowledge of SQL and comfort reading a little Python is enough. We introduce each dbt and Airflow feature where it earns its place, in the context of a problem it solves, rather than as a feature tour. If you can write a SELECT statement and reason about dependencies, you are ready.

Where We Go From Here

With the foundations in place — why ELT won, what dbt and Airflow each own, how the warehouse is layered, and why testing is non-negotiable — we are ready to start building. The next chapter begins where every pipeline begins: getting data in reliably, with ingestion patterns that are idempotent, replayable and safe to run again after a failure.

Keep the mental model from this chapter close as you read on. Almost every technique in the book is, at bottom, in service of one of three goals: making the warehouse correct, making it observable, and making it safe to change. When a later recommendation seems fussy, ask which of those three it serves — and it will almost always be obvious.

This is a free sample

You've reached the end of the sample chapter.

Get the complete book — every chapter, fully worked — at **dataforgebooks.com**.

FULL EDITION · 284 PAGES · PDF

Read the full title at **dataforgebooks.com**

Questions? support@dataforgebooks.com