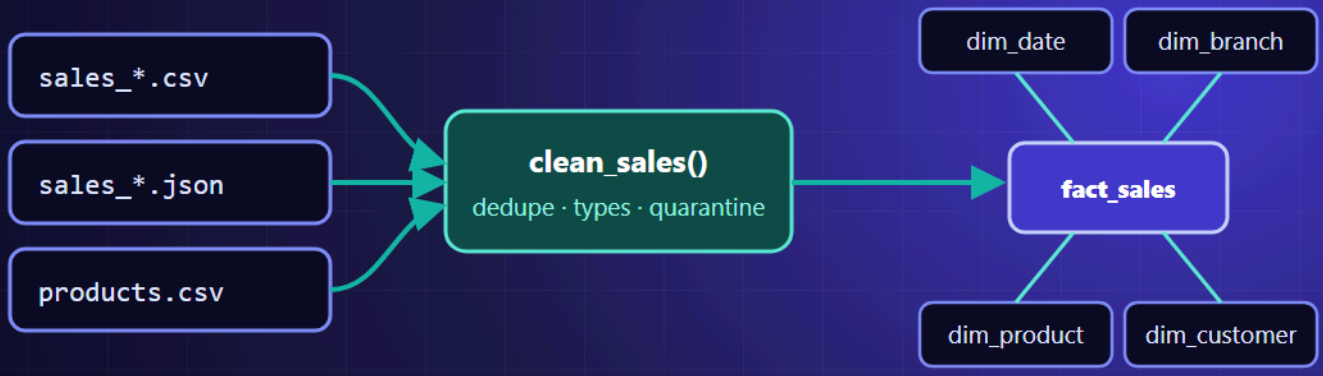



```sql
SELECT branch_id,
  sum(net_amount) AS net
FROM 'sales/*.parquet'
GROUP BY ALL
ORDER BY net DESC;
```



QUALITY CHECKS

run 2026-03-05

unique keys	PASS
not null	PASS
quantity > 0	PASS
reconciles	PASS
quarantined	1 row

idempotent rerun ✓

BUILD RELIABLE DATA PIPELINES

Data Engineering with Python

From messy files to a tested, orchestrated warehouse and reporting mart

pandas 3 · DuckDB · SQL · Parquet · Star Schemas · SCD2
Data Quality · Idempotent Loads · PostgreSQL · dbt · Airflow

INGEST · MODEL · TEST · ORCHESTRATE · OPERATE

THENEXBOOKS

Data Engineering with Python

Build Reliable Data Pipelines with pandas, DuckDB, SQL, dbt, and Airflow
From raw files to a tested, orchestrated, monitored pipeline — built step by step

First Edition

THENEXBOOKS

Data Engineering with Python
Build Reliable Data Pipelines with pandas, DuckDB, SQL, dbt, and Airflow

First Edition, 2026

Copyright © 2026 thenexbooks (thenexbooks.gumroad.com). All rights reserved.

No part of this publication may be reproduced, distributed, or transmitted in any form or by any means without the prior written permission of the publisher, except for brief quotations in reviews and certain other non-commercial uses permitted by copyright law. Code examples, SQL, and configuration files may be used freely in your own projects.

The information in this book is provided "as is" without warranty of any kind. While every precaution has been taken in its preparation, the publisher assumes no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein. Python libraries, databases, orchestration tools, and cloud services change frequently; APIs, defaults, command names, and feature availability can differ between versions. Always consult the official documentation for the versions you use, test pipelines on copies of your data, and back up anything a pipeline can change or delete.

Python is a trademark of the Python Software Foundation. pandas is a trademark of NumFOCUS, Inc. DuckDB is a trademark of the DuckDB Foundation. PostgreSQL is a trademark of the PostgreSQL Community Association of Canada. Apache, Apache Airflow, Airflow, Apache Kafka, Apache Iceberg, Apache Hudi, Apache Spark, Apache Flink, Apache Parquet, and Apache Arrow are trademarks of The Apache Software Foundation. dbt is a trademark of dbt Labs, Inc. Prefect, Dagster, Snowflake, Google BigQuery, Amazon Redshift, Amazon S3, Databricks, Delta Lake, Microsoft Fabric, Azure, GitHub, GitHub Actions, Windows, macOS, and other product names are trademarks of their respective owners. Their use in this book is for identification only and does not imply endorsement. This book is independent and is not affiliated with, sponsored by, or endorsed by any of these organizations.

Brambleway Grocers, the grocery chain used as the running example in this book, is fictional, as are its branches, towns, products, customers, people, and all sample data. Email addresses use the reserved example.com domain.

Preface

Most data problems are not analysis problems. The analysis is usually the easy part: a group-by, a join, a chart. The hard part is everything that has to be true before the analysis can be trusted. The files must arrive. The columns must mean what we think they mean. A row must not be counted twice because a job was rerun, and a bad row must not quietly vanish. Yesterday's numbers must still be there tomorrow, and when something breaks at 03:00, somebody must find out before the people who depend on the numbers do.

That work has a name: data engineering. It is the discipline of moving, cleaning, modeling, and serving data *reliably*. If you already write Python and SQL, you have most of the raw skills. What you need next are the engineering habits that turn a working notebook into a pipeline people can depend on: explicit schemas, idempotent loads, data quality checks, tests, dimensional models, orchestration, monitoring, and secure handling of personal data.

This book teaches those habits by building one real pipeline from start to finish. We use tools that run on a laptop and scale further than most people expect — Python, pandas 3, DuckDB, SQL, and PostgreSQL — and we connect them to the tools used across the industry for orchestration and transformation: Apache Airflow and dbt. Every pandas, DuckDB, and SQL example in the book was run against the sample data, and the output you see is the real output.

Who This Book Is For

- **Analysts and BI developers** who are comfortable with SQL and some Python and want to own the pipelines that feed their reports.
- **Python developers** who are asked to "just load this data" and want to do it properly, without duplicates, silent failures, or 2 a.m. surprises.
- **Data scientists** who spend too much time cleaning and re-cleaning data and want reproducible, tested ingestion.
- **New data engineers** who want a practical, end-to-end foundation before specializing in a particular cloud platform.

You should know basic Python (functions, lists, dictionaries, loops) and basic SQL (SELECT, WHERE, JOIN, GROUP BY). Everything else is explained as we go.

What You Will Learn

- What data engineers do, how batch and streaming differ, and how to choose between CSV, JSON, and Parquet.
- How to ingest and clean messy files with pandas 3 using explicit schemas — and what Copy-on-Write and the new string dtype change in your code.

- How to use DuckDB to run fast analytics SQL directly on files, including window functions, QUALIFY, and nested JSON.
- How to design a star schema with facts and dimensions, and how to track history with slowly changing dimensions (type 2).
- How to build data quality checks, quarantine bad rows, reconcile counts, and test pipeline code.
- How to make loads idempotent and incremental, handle late data and backfills, and load PostgreSQL efficiently.
- How orchestration works, how to write an Airflow DAG, and how dbt organizes SQL transformations, tests, and snapshots.
- How to log, monitor, and alert; how to find performance problems; how to protect personal data; and how to run CI for data pipelines.

How This Book Is Organized

Part I — Foundations (Chapters 1–5) introduces the role, sets up the workspace, compares batch and streaming, explains file formats, and establishes Python habits for pipeline code. **Part II — Ingest and Transform** (Chapters 6–10) covers ingesting and cleaning data with pandas 3, combining and reshaping it, the pandas 3 behaviors you must know, and DuckDB fundamentals. **Part III — Modeling and Warehousing** (Chapters 11–15) covers advanced analytics SQL, dimensional modeling, slowly changing dimensions, building the warehouse, and an overview of cloud warehouses and lakehouses. **Part IV — Reliable Pipelines** (Chapters 16–20) covers data quality, testing, idempotent and incremental loads, and loading PostgreSQL. **Part V — Orchestration and Transformation** (Chapters 21–24) covers orchestration concepts, Apache Airflow, dbt, and logging and monitoring. **Part VI — Production** (Chapters 25–28) covers performance, security and privacy, continuous integration, and a complete capstone. The appendices provide a cheat sheet, templates and reference material, a production checklist, and a glossary.

Each chapter ends with "Try It Yourself" exercises and a **Key Takeaways** box for quick review.

The Running Example: Brambleway Grocers

Throughout the book we build the data pipeline for **Brambleway Grocers**, a fictional grocery chain with four branches. Three branches run an older point-of-sale system that drops one CSV file per day; the newest branch, Harbor Point, runs a cloud system that exports nested JSON. Today the finance team assembles a weekly sales spreadsheet by hand. Nadia, who leads operations and finance, wants trustworthy daily sales per branch by 07:00 every morning. Together with Theo, a senior data engineer, Iris, an analyst, and Sam, a new data engineer learning alongside you, we turn two weeks of raw files — including a few deliberately dirty rows — into a tested, idempotent, orchestrated pipeline with a DuckDB star-schema warehouse and a PostgreSQL reporting mart.

The sample data is small enough to check by hand: 792 raw sales lines over 14 days, 13 products, 20 loyalty customers. Every number in the book — 786 clean lines, $6,003.66 in net sales, three quarantined rows — comes from running the code on that data, so you can reproduce each result yourself. The complete generator script for the sample data is printed in Appendix B, so you can recreate every file with one command.

Versions and Conventions

Examples were verified with **Python 3.12**, **pandas 3.0**, **NumPy 2**, **DuckDB 1.5**, and **PostgreSQL 18**. Most code also works with nearby versions; where pandas 3 behaves differently from pandas 2, the text says so. Airflow examples target **Airflow 3.x** with notes for 2.x, and dbt examples target recent dbt Core 1.x versions; tools in this fast-moving space change often, so check the documentation for your version. Commands are shown for macOS and Linux shells, with Windows differences noted where they matter.

> **TIP**
> A recommended practice or a technique that makes your work easier.

> **NOTE**
> Background information, version differences, or a useful aside.

> **WARNING**
> A common mistake or something that can cause data loss, wrong numbers, or security problems.

Code, SQL, file names, and column names appear in `monospace`. A caption above each listing names the file it belongs to, such as `brambleway/clean.py` or `sql/warehouse.sql`, or labels it as `Output`. Long lines are wrapped to fit the page. Output that was trimmed for space is marked with `...`.

Contents

Preface

PART I — FOUNDATIONS

1 What Data Engineers Do: Pipelines, Roles, and the Brambleway Project

2 Setting Up Your Data Engineering Workspace: Python, pandas, DuckDB, and the Sample Data

3 Batch and Streaming: Processing Models, Latency, and Architecture Choices

4 File Formats That Matter: CSV, JSON, and Parquet

5 Python for Pipelines: Project Layout, Functions, Types, and Configuration

PART II — INGEST AND TRANSFORM

6 Ingesting Files with pandas 3: Explicit Schemas for CSV and JSON

7 Cleaning Data with pandas: Types, Missing Values, Duplicates, and Quarantine

8 Combining and Reshaping: Joins, Group By, Pivots, and Window Operations

9 pandas 3 Behaviors You Must Know: Copy-on-Write, String Dtypes, and Datetimes

10 DuckDB Fundamentals: Analytics SQL Directly on Files

PART III — MODELING AND WAREHOUSING

11 Advanced Analytics SQL in DuckDB: CTEs, Window Functions, QUALIFY, and Nested Data

12 Dimensional Modeling: Star Schemas, Grain, Facts, and Dimensions

13 Slowly Changing Dimensions: Type 1, Type 2, and Point-in-Time Joins

14 Building the Brambleway Warehouse in DuckDB

15 Warehouses and Lakehouses: Cloud Platforms and Open Table Formats

PART IV — RELIABLE PIPELINES

16 Data Quality Checks: Validation Rules, Quarantine, and Reconciliation

17 Testing Data Pipelines: Unit Tests, Fixtures, and Data Contracts

18 Idempotent Loads: Safe Reruns, Delete-Insert, Upserts, and MERGE

19 Incremental Loads: Watermarks, Late Data, and Backfills

20 Loading PostgreSQL: COPY, Upserts, and a Reporting Mart

PART V — ORCHESTRATION AND TRANSFORMATION

21 Orchestration Concepts: From Scripts to a CLI, Scheduling, Retries, and Dependencies

22 Apache Airflow: DAGs, Tasks, and the Brambleway Daily Pipeline

23 dbt: SQL Models, Sources, Tests, Snapshots, and Documentation

24 Logging, Monitoring, and Alerting for Data Pipelines

PART VI — PRODUCTION

25 Performance: Profiling, Memory, Chunking, and Parquet Layouts

26 Security, Privacy, and PII in Data Pipelines

27 Continuous Integration for Data Pipelines

28 Capstone: The Complete Brambleway Pipeline, End to End

APPENDICES

A Data Engineering Cheat Sheet

B Templates and Reference

C Production Checklist

D Glossary

PART I

Foundations

What data engineers actually do, a clean and reproducible workspace, the trade-offs between batch and streaming, the file formats every pipeline meets, and the Python habits that turn scripts into dependable pipeline code.

1 · What Data Engineers Do

2 · Your Data Engineering Workspace

3 · Batch and Streaming

4 · CSV, JSON, and Parquet

5 · Python for Pipelines

CHAPTER 1 · PART I — FOUNDATIONS

What Data Engineers Do: Pipelines, Roles, and the Brambleway Project

Data engineering is the work of moving, cleaning, modeling, and serving data so reliably that other people can stop worrying about it. In this chapter we define the job, map the data lifecycle, explain what "reliable" really means, and meet Brambleway Grocers, the four-branch grocery chain whose sales pipeline we build together for the rest of the book.

The Monday Morning Spreadsheet

Every Monday, someone in the finance office at Brambleway Grocers copies a folder of sales files into a spreadsheet, fixes whatever looks odd, and builds a pivot table of sales per branch. It takes most of the morning, and nobody is sure whether a receipt that appears twice was really sold twice. When the newest branch started sending JSON instead of CSV, the spreadsheet stopped matching the till totals for a week.

Nadia, who leads operations and finance, wants something simple to describe and hard to deliver: **daily sales per branch, ready by 07:00 every morning, and correct**. Not "roughly right", not "right unless a file was late", but numbers she can put in front of the board without a disclaimer. That request is a data engineering problem, and it is the problem this book solves.

If you have written a pandas script that turned a messy export into a tidy report, you have already done a small piece of data engineering. What changes when the script becomes a pipeline is not the syntax but the questions you must answer before anyone trusts the output: what happens when it runs twice, when a file arrives late or is corrected, and who finds out when something breaks.

What Data Engineering Is

A useful working definition: **data engineering is building and operating the systems that move data from where it is produced to where it is used, in a shape people can trust.** The definition has four verbs hidden in it, and each of them is a large part of the job.

Moving data

Data is produced in operational systems: point-of-sale terminals, web applications, sensors, partner exports, spreadsheets. It is rarely used there. Analysts want it in a database they can query; finance wants it in a report; a machine learning model wants it as a training table. Moving data means

extracting it from a source, transporting it, and landing it somewhere durable, without losing rows and without inventing new ones.

Cleaning data

Raw data is honest about what the source system did, which is not the same as being correct. Product codes arrive with stray spaces, timestamps arrive in two different formats, a quantity of zero sneaks in, and the same line is exported twice. Cleaning means turning raw text into typed, validated records and, just as importantly, deciding what to do with rows that cannot be fixed. Throwing them away silently is the most common mistake in the field; we will quarantine them instead, so somebody can look.

Modeling data

Clean rows are still not easy to analyze. Modeling means organizing data into structures that make common questions simple and fast: fact tables of events, dimension tables of the things those events are about, and history-aware records that remember that a customer changed loyalty tier on a particular day. Good models encode business meaning, so that "net sales" is defined once rather than re-derived in every dashboard.

Serving data

Finally, data has to reach the people and programs that use it: a reporting database for dashboards, a file for a partner, an API, a feature table. Serving includes access control, freshness guarantees, and documentation. A perfect table that nobody can find or query has not been served.

Around those four verbs sits **operations**: scheduling, retries, monitoring, testing, and security, where most of the time goes once the first version works.

Data engineering and its neighbors

Job titles in data overlap heavily and vary between companies, so treat the following as a map of responsibilities rather than a set of boxes. Many people, especially in small teams, do several of these at once.

Role	Main question	Typical output
Data analyst	What happened, and why?	Reports, dashboards, ad hoc analysis
Data scientist	What will happen, and what should we do?	Models, experiments, forecasts
Analytics engineer	How do we define and model business metrics in SQL?	Tested, documented transformation models
Data engineer	How does data arrive correctly, on time, every time?	Pipelines, warehouses, quality checks, operations
Platform engineer	What infrastructure do data teams run on?	Clusters, deployment, shared services

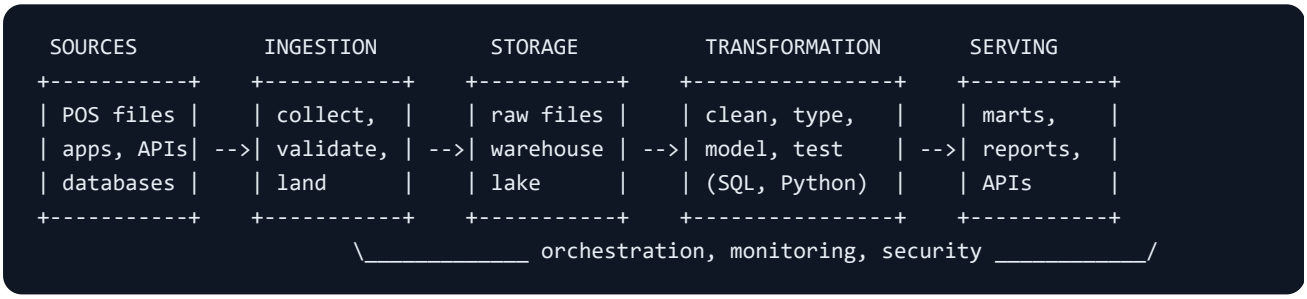
The boundary between data engineering and analytics engineering is especially fuzzy. In this book we cover both sides: the Python ingestion and loading code that a data engineer typically owns, and the SQL modeling and testing that an analytics engineer typically owns. Knowing both makes you better at either.

> **NOTE**
> You do not need a particular job title to use this book. If you are an analyst who is tired of rebuilding the same spreadsheet, a backend developer who has been asked to "just get the data into the warehouse", or a scientist whose models keep breaking because the input changed, the practices here apply to you.

The Data Lifecycle

Most data systems, from a two-person startup to a global retailer, follow the same broad lifecycle. The names change between companies and vendors, but the stages do not.

The data lifecycle



- **Sources** are systems you usually do not control. Their formats change without warning, and their idea of "correct" may differ from yours.
- **Ingestion** collects data and lands it somewhere safe, ideally exactly as received, so you can always replay history.
- **Storage** keeps raw and processed data durable and queryable: files on disk or object storage, a database, a warehouse.
- **Transformation** turns raw records into clean, modeled tables. This is where business rules live.
- **Serving** exposes the results to people and programs with the right access, freshness, and documentation.

Underneath every stage runs the operational layer: scheduling, run records, alarms, and security. Beginners see the arrows between boxes; experienced engineers watch that bottom line.

ETL and ELT

Two acronyms describe where transformation happens. In **ETL** (extract, transform, load), data is cleaned and reshaped in a separate processing step, often in Python or a dedicated tool, and only the finished result is loaded into the target database. In **ELT** (extract, load, transform), raw data is loaded

first, and transformation happens inside the database with SQL.

ETL was the default when database storage was scarce. ELT became popular as analytical databases grew cheaper and tools such as dbt made SQL transformations easy to version and test. Most real pipelines mix both.

Aspect	ETL	ELT
Where logic runs	Outside the database (Python, tools)	Inside the database (SQL)
Raw data kept?	Often not in the database	Yes, in raw or staging tables
Strengths	Complex parsing, files, APIs	Set-based logic, lineage, reuse
Typical tools	pandas, custom code	SQL engines, dbt

The Brambleway pipeline is a deliberate hybrid. Python and pandas handle the parts that are awkward in SQL: reading two file formats, flattening nested JSON, applying fix-up rules, and routing bad rows to quarantine. Once the data is typed and clean, SQL in DuckDB does the modeling, because joins, aggregations, and history tracking are what SQL is good at. Later, in Chapter 23, we rebuild the SQL layer with dbt to show the ELT style end to end.

What "Reliable" Means

"Make it reliable" is the most common requirement in data work and the least specific. We will use five concrete properties throughout the book. When a design decision comes up, we will ask which of these it protects.

Property	Question it answers	Brambleway example
Correct	Are the numbers right?	Net sales matches the receipts, to the cent
Complete	Is anything missing or extra?	Every raw line is loaded, quarantined, or explained
Timely	Is it ready when needed?	Daily branch sales ready by 07:00
Idempotent	Is a rerun safe?	Running a day twice does not double it
Observable	Can we tell what happened?	Each run records counts, status, and errors

Correctness hides a trap: a pipeline can be correct on the data you tested and wrong on next week's data, which is why we write explicit schemas and checks instead of trusting what a library infers. **Completeness** means every input row is accounted for: a pipeline that reads 792 lines and loads 786 must say exactly where the other six went ("three duplicates dropped, three rows quarantined, with reasons"). **Timeliness** is measured against a promise; perfect numbers at 09:30 fail a 07:00 deadline.

Idempotency is the property most people learn about the hard way. An operation is idempotent if running it twice has the same effect as running it once. Pipelines are rerun constantly, and if a rerun appends the same day again, every total for that day silently doubles. We reproduce exactly that failure in Chapter 18 and then fix it. **Observability** means you can answer "what happened last night?" without reading the code, because each run leaves a record of what it read, dropped, quarantined, and loaded, and why it failed if it did.

> **TIP**
>
> When you review any pipeline, yours or someone else's, ask five questions in order: What if this runs twice? What if a file is late? What if a row is malformed? How would I know it failed? How would I know the numbers are right? The answers reveal most design weaknesses in minutes.

Meet Brambleway Grocers

Brambleway Grocers is a fictional regional grocery chain. It is small enough to understand completely and messy enough to be realistic. Everything in the book, from the first line count to the final capstone, uses the same data and the same numbers, so you can check your own results against ours at every step.

The branches

ID	Branch	City	Opened	POS system
B01	Old Town	Millbrook	2014-05-02	legacy_csv
B02	Riverside	Millbrook	2017-09-15	legacy_csv
B03	Hillcrest	Eastfield	2020-03-20	legacy_csv
B04	Harbor Point	Port Avery	2025-11-07	cloud_json

The three older branches run a legacy point-of-sale (POS) system that drops **one CSV file per day** with one row per receipt line. Harbor Point, opened in November 2025, runs a cloud POS that exports **one JSON file per day**, where each receipt contains a nested list of lines. Two formats for one business event is what happens whenever a company switches vendors or pilots a new system.

The sample data

Our data covers two weeks, from Sunday 1 March to Saturday 14 March 2026: 14 CSV files with 600 lines from branches B01 to B03, and 14 JSON files with 80 receipts and 192 lines from B04. There are also reference files for the four branches, 12 products (with a price-change snapshot effective 8 March that adds a 13th product), and 20 loyalty customers (with a snapshot on 8 March in which two customers change). One corrected file is redelivered late, exactly the way real sources behave.

The data is generated by a small, pure-standard-library script with a fixed random seed, printed in full in Appendix B.15 as `tools/make_data.py`. Because the seed is fixed, the files on your machine will be byte-for-byte identical to ours, and every number in the book will match. Chapter 2 walks you through generating them.

The files also contain a handful of deliberate problems, planted where real POS exports tend to have them: an exact duplicate row, a product code with stray spaces and lowercase letters, an unknown product code, a quantity of zero, a price written as `$2.99`, a receipt whose timestamp uses a different date format, a customer ID that does not exist, a missing price, a JSON quantity sent as a string, and a whole JSON receipt sent twice. We will meet each of them in the chapters where they matter, and by Chapter 7 our cleaning code will handle all of them with a precise accounting of what happened to every row.

The people

Four people appear throughout the book. They are fictional, and they stand in for perspectives you will meet on any data team.

- **Nadia**, operations and finance lead, is the customer. She wants daily sales per branch by 07:00 and does not care which tools produce it, only that the numbers are right and on time.
- **Theo**, senior data engineer, reviews designs and code. When Theo asks "what happens if this runs twice?", he is asking the question that will save you a weekend.
- **Iris**, analyst, builds dashboards on the reporting mart. She is the main consumer of what we serve and the first to notice when a definition is unclear.
- **Sam**, new data engineer, learns alongside you. Sam makes the reasonable mistakes that everyone makes once, so you can see them fail safely on the page.

The requirements

After a conversation with Nadia, Theo writes the requirements on one page. They are short, and every later chapter traces back to one of them.

1. Each day's sales files arrive by 02:00 the next morning. The pipeline runs at 06:00, and the table `mart.daily_branch_sales` must be ready by 07:00.
2. Both POS formats are combined into one consistent set of sales lines.
3. Rows that cannot be trusted are never silently dropped. They are stored with a reason for review.
4. Money is exact. Net amount is quantity times unit price minus discount, stored as a decimal, never a float.
5. Rerunning any day, or reloading a corrected file, gives the same result as running it once.
6. Customer and product history is preserved, so a sale is reported with the price list and loyalty tier that applied on the day it happened.
7. Every run records what it did, and a failure produces an alert rather than a silently stale report.

Notice that the list names no tools. Requirements describe outcomes; tools are how we meet them.

The Architecture We Will Build

Here is the whole Brambleway pipeline at the level of detail Theo would sketch on a whiteboard. Each box is a part of the book, and each arrow is a place where something can go wrong and where we will add a safeguard.

Brambleway pipeline architecture



Raw files stay exactly as received, so we can always reprocess them. Python reads both formats as text, applies the cleaning rules, and loads typed rows into a DuckDB file that acts as our warehouse. Inside the warehouse we build a star schema: one fact table at the grain of a single receipt line, surrounded by dimensions for dates, branches, products, and customers. A publishing step copies the reporting tables into PostgreSQL, where Iris's dashboard tool connects with a read-only account. Airflow runs the whole thing every morning, and dbt shows how the SQL layer can be managed as a tested, documented project.

By the end of the book, a single command will process the full two weeks and produce numbers you can verify against ours: 786 clean sales lines across 323 receipts with net sales of $6,003.66, three duplicates dropped, and three rows quarantined with reasons.

The Toolset, and Why We Chose It

Every tool in this book was chosen because it teaches a transferable idea and runs on a laptop without a cloud account. The concepts carry over directly to larger platforms.

- **Python** is the glue. It reads files, calls databases, applies rules that are awkward in SQL, and packages the pipeline as testable functions. We target Python 3.12.
- **pandas** is the most widely used DataFrame library in Python, and the one you will meet in existing code. We use pandas 3, which changed several long-standing behaviors (Copy-on-Write and a new default string type, among others). Chapter 9 covers the changes that matter for pipeline code.
- **DuckDB** is an in-process analytical database: no server to install, a single file on disk, and fast SQL over tables, CSV, JSON, and Parquet files. It is our warehouse, and it lets us practice warehouse modeling on a laptop. We use DuckDB 1.5; most features shown work in recent versions.
- **SQL** is the language of modeling and serving. Nearly every analytical system speaks a dialect of it, so time spent on SQL pays off everywhere.
- **PostgreSQL** is our serving database: a mature, open-source relational database that dashboard tools connect to easily. We use PostgreSQL 18; most of the SQL also works on version 15 and later.
- **dbt** (data build tool) manages SQL transformations as a project with dependencies, tests, and documentation. We use dbt Core, the open-source command-line tool.
- **Apache Airflow** is a widely used workflow orchestrator. We target Airflow 3.x and point out differences for teams still running 2.x.

> **NOTE**
>
> Airflow and dbt need more setup than the rest of the toolset, and their exact behavior depends on the version you install. The chapters that use them (22 and 23) keep the code small and conventional and point out where to check your version's documentation. Everything else in the book, including every pandas, DuckDB, and SQL example, was run and its output pasted from a real execution.

What this book does not cover deeply

We explain the concepts behind Apache Spark and Apache Kafka in Chapter 3, but we do not operate clusters. We describe cloud warehouses and open table formats in Chapter 15 without tying the book to any vendor, and we do not cover machine learning pipelines or infrastructure-as-code in depth. The core skills here (explicit schemas, idempotent loads, modeling, testing, observability) are exactly what those larger systems require.

A First Taste: Looking at One Raw File

Before we install anything beyond Python itself, let us look at a single day of raw data with nothing but the standard library. The point is to see the raw material and notice how quickly an innocent script goes wrong.

If you want to run the code now, follow the "Generating the Sample Data" steps in Chapter 2 first and then come back; the script expects the files under `data/raw/` in your project folder. Otherwise, read along: the output below is exactly what you will see later.

first_look.py

```
import csv
import json
from collections import Counter
from pathlib import Path

path = Path("data/raw/sales/sales_2026-03-01.csv")

with path.open(newline="", encoding="utf-8") as f:
    rows = list(csv.DictReader(f))

print(f"{path.name}: {len(rows)} lines")
print("Columns:", ",".join(rows[0].keys()))
first = rows[0]
print("First line:", first["receipt_id"], first["sku"], repr(first["quantity"]))

receipts = {row["receipt_id"] for row in rows}
print(f"Distinct receipts: {len(receipts)}")

per_branch = Counter(row["branch_id"] for row in rows)
for branch, count in sorted(per_branch.items()):
    print(f"  {branch}: {count} lines")

csv_net = sum(
    int(r["quantity"]) * float(r["unit_price"]) - float(r["discount"]) for r in rows
)
print(f"Net sales in the CSV file: {csv_net:.2f}")

# B04 Harbor Point sends JSON receipts with nested lines
json_path = path.with_suffix(".json")
receipts_b04 = json.loads(json_path.read_text(encoding="utf-8"))
lines_b04 = [line for receipt in receipts_b04 for line in receipt["lines"]]
json_net = sum(l["qty"] * l["price"] - l["discount"] for l in lines_b04)
print(f"{json_path.name}: {len(receipts_b04)} receipts, {len(lines_b04)} lines")
print(f"Net sales in the JSON file: {json_net:.2f}")
print(f"Day total: {len(rows) + len(lines_b04)} lines, net {csv_net + json_net:.2f}")
```

Terminal

```
$ python first_look.py
```

Output

```
sales_2026-03-01.csv: 47 lines
Columns: receipt_id,line_no,sold_at,branch_id,customer_id,sku,quantity,unit_price,discount
First line: B01-20260301-0001 BW-1004 '2'
Distinct receipts: 17
  B01: 15 lines
  B02: 13 lines
  B03: 19 lines
Net sales in the CSV file: 455.03
sales_2026-03-01.json: 4 receipts, 7 lines
Net sales in the JSON file: 94.59
Day total: 54 lines, net 549.62
```

Several lessons are already hiding in this small output.

- **Lines are not receipts.** The CSV file has 47 lines but only 17 receipts, because each row is one line of a receipt. Whenever someone asks "how many sales?", the first engineering question is "sales lines or receipts?". In Chapter 12 we will call this the *grain* of the data.
- **Everything in a CSV file is text.** The quantity prints as `'2'`, a string. CSV has no types; any type you see in a DataFrame was guessed by the reader. Chapter 4 explains why that guess is dangerous, and Chapter 6 shows how to take control of it.
- **Two formats, one business.** B04's sales live in a different file with different field names (`qty` and `price` instead of `quantity` and `unit_price`) and a nested structure. A day is only complete when both are combined: 54 lines and $549.62 for 1 March.
- **Money as floats is a warning sign.** We used `float` to multiply prices, and formatted the result to two decimals so it looks right. Chapter 7 shows why floats and money do not mix, and the warehouse will store money as exact decimals.

The same script, a different day

The CSV part of that script worked on 1 March. Sam, encouraged, points a slightly simplified version of it at other days to build a quick weekly total. Here is the CSV-only version, taking the file name as an argument.

naive_total.py

```
import csv
import sys
from pathlib import Path

path = Path(sys.argv[1])
with path.open(newline="", encoding="utf-8") as f:
    rows = list(csv.DictReader(f))

net = sum(
    int(r["quantity"]) * float(r["unit_price"]) - float(r["discount"]) for r in rows
)
print(f"{path.name}: {len(rows)} lines, net {net:.2f}")
```

Terminal

```
$ python naive_total.py data/raw/sales/sales_2026-03-03.csv
$ python naive_total.py data/raw/sales/sales_2026-03-09.csv
$ python naive_total.py data/raw/sales/sales_2026-03-12.csv
```

Output (trimmed)

```
sales_2026-03-03.csv: 59 lines, net 384.75
...
ValueError: could not convert string to float: '$2.99'
...
ValueError: could not convert string to float: ''
```

Look closely at these three results, because they summarize the whole book in miniature.

The two crashes are the *good* failures. On 9 March a price arrived as `$2.99`, and on 12 March a price was missing entirely. The script stopped loudly, and somebody will notice. Annoying, but safe.

The first result is the dangerous one. It ran without complaint and printed a plausible number, yet it is wrong. The 3 March file contains one line that was exported twice, receipt `B02-20260303-0002` line 1, a six-pack of sparkling water at $3.99. The naive total counts it twice, so the CSV net is overstated by exactly $3.99. Nothing crashed, nothing warned, and if this number had gone into Nadia's report it would have been wrong with no trace of why. Silent errors like this one are why data engineering exists as a discipline.

> **WARNING**
> A pipeline that crashes is inconvenient. A pipeline that quietly produces wrong numbers is harmful, because people make decisions with them. Throughout this book we prefer designs that fail loudly, and we add checks specifically to turn silent errors into visible ones.

A proper fix is not to wrap the conversion in `try` and skip bad rows, which would turn the loud failures into silent ones. It is to decide, row by row and rule by rule, what each problem means: a duplicate is dropped on a defined key, a dollar sign is stripped, a missing price sends the row to quarantine with a reason. That is the cleaning logic we build in Chapters 6 and 7.

How This Book Is Organized

The book follows the pipeline from raw files to production, in six parts. Each chapter adds a working piece to the Brambleway project, so later chapters build on code you have already written.

- **Part I, Foundations** (Chapters 1 to 5): the job, your workspace, processing models, file formats, and the Python habits that make pipeline code maintainable.
- **Part II, Ingest and Transform** (Chapters 6 to 10): reading files with explicit schemas, cleaning with quarantine, reshaping data, pandas 3 behaviors you must know, and DuckDB fundamentals.