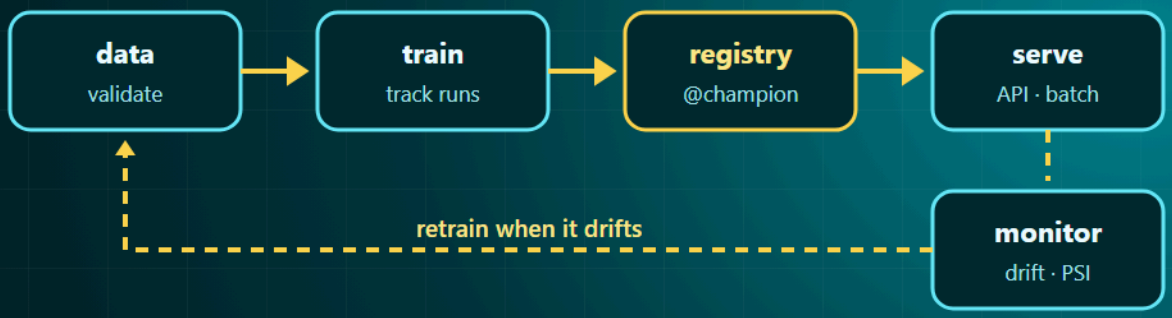
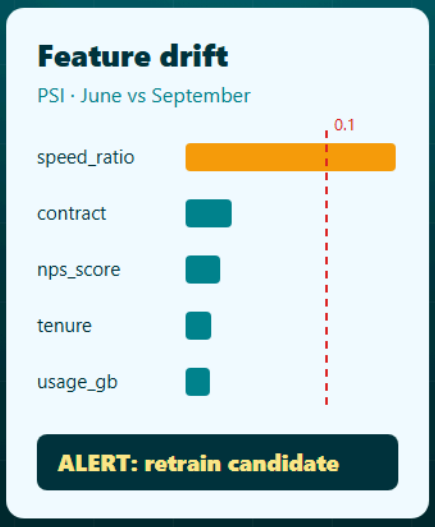



```python
with mlflow.start_run():
  pipe.fit(X_train, y_train)
  mlflow.log_metric("roc_auc", auc)
  if gate_passed(metrics):
    register(pipe, "churn")
```

```
$ python -m pytest
all tests passed · gate: roc_auc OK
```



FROM NOTEBOOK TO PRODUCTION

MLOps in Practice

Ship, test, serve, and monitor machine learning models with confidence

Reproducible projects · Data validation · MLflow tracking
Model registry · pytest · GitHub Actions · FastAPI · Drift

PIPELINES · REGISTRY · CI/CD · SERVING · MONITORING

THENEXBOOKS

MLOps in Practice

Taking Machine Learning Models from Notebook to Production with MLflow, FastAPI, and CI/CD
Reproducible pipelines, a model registry, tested services, and monitoring — built step by step on one real project

First Edition

THENEXBOOKS

MLOps in Practice
Taking Machine Learning Models from Notebook to Production with MLflow, FastAPI, and CI/CD

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. The code examples 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 packages, MLflow, FastAPI, GitHub Actions, and container tools change frequently; command options, defaults, and output can differ between versions. The Python code in this book was run with the versions listed in the Preface on a CPU-only Windows machine; container builds and GitHub Actions workflows are shown as written and were not executed there, as the text points out. Always check the current documentation for the versions you use. The model in this book predicts customer churn from synthetic data; it is a teaching example, not advice for any real business decision.

Python is a trademark of the Python Software Foundation. MLflow is a trademark of LF Projects, LLC. Docker is a trademark of Docker, Inc. GitHub and GitHub Actions are trademarks of GitHub, Inc. Windows is a trademark of the Microsoft group of companies. Linux is a trademark of Linus Torvalds. 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.

Fairhollow Broadband, the internet service provider used as the running example in this book, is fictional, as are its employees, customers, competitors, and all sample data. The customer data is synthetic and is generated by a seeded script printed in the book; it describes no real person.

Preface

Training a model that scores well in a notebook is the easy part. The hard part starts the next morning, when someone asks whether the model can run every month, whether the numbers can be reproduced, which data it was trained on, how the call center will get its predictions, what happens when a bad file arrives, and how anyone will know when the model quietly stops working. Those questions are what MLOps answers.

MLOps is not a single tool. It is a set of engineering habits — reproducibility, validation, tracking, testing, automation, safe deployment, and monitoring — applied to systems whose behavior depends on data as much as on code. This book teaches those habits by doing them. We take one classic machine learning model, a customer churn model, and carry it all the way from a single script to a tested, versioned, served, monitored, and retrainable system.

Everything is hands-on and small enough to run on a laptop. We use widely adopted open-source tools — scikit-learn, MLflow, FastAPI, pytest, and GitHub Actions — and we explain the ideas underneath them, so the lessons carry over to whatever platform your team uses.

Who This Book Is For

- **Data scientists** who can train a model and want to see it used reliably in production.
- **Software and backend engineers** who are asked to deploy, test, or operate machine learning models.
- **Data and ML engineers** who want a clear, end-to-end reference for pipelines, registries, serving, and monitoring.
- **Technical leads** who need to decide what "production-ready" should mean for their team's models.

You should know basic Python (functions, modules, virtual environments, running scripts) and have trained at least one simple model. We review the machine learning we need, but this is not a first course in machine learning, and it does not cover large language model applications.

What You Will Learn

- How to turn a notebook into a packaged Python project with a src layout, a typed configuration file, pinned dependencies, and reproducible runs.
- How to generate and validate data with an explicit data contract, and how to keep features identical in training and serving.
- How to track experiments, compare runs, and manage model versions and aliases with MLflow tracking and the MLflow Model Registry.
- How to evaluate a model the way the business uses it: thresholds, capacity limits, calibration, slices, and costs.

- How to test data, features, pipelines, models, and APIs with pytest, and how to automate checks and training with GitHub Actions.
- How to serve predictions online with FastAPI, harden the service, package it in a container, and run idempotent batch scoring jobs.
- How to release models safely, monitor them in production, measure drift with PSI and statistical tests, retrain, and promote a challenger only when it is better.
- How to document and govern models with model cards, audit trails, and clear ownership.

How This Book Is Organized

Part I — Foundations (Chapters 1–5) explains what MLOps is, frames the churn problem, and builds a reproducible project. **Part II — Data and Features** (Chapters 6–10) covers the data generator, data validation, feature pipelines, leakage and training–serving skew, and versioning. **Part III — Experiments and the Model Registry** (Chapters 11–15) covers MLflow tracking, the training pipeline, business-aware evaluation, the registry, and model packaging. **Part IV — Testing and Automation** (Chapters 16–19) covers testing code and models and automating with CI/CD. **Part V — Deployment and Serving** (Chapters 20–24) covers the FastAPI service, hardening, containers, batch scoring, and release strategies. **Part VI — Monitoring and Operations** (Chapters 25–28) covers monitoring, drift, retraining, governance, and a complete case study. The appendices provide a cheat sheet, templates and reference material, a production checklist, and a glossary.

Each chapter includes runnable code, Tips, Notes, and Warnings, a "Try It Yourself" section, and ends with a **Key Takeaways** box.

The Running Example: Fairhollow Broadband

Fairhollow Broadband is a fictional regional internet service provider with customers in metro, suburban, and rural areas. Its retention team can call only a limited number of customers each month, so it wants a model that ranks customers by their risk of cancelling. We build the project `fairhollow-churn`: a seeded generator for customer data, a data contract, a scikit-learn pipeline, MLflow tracking and a model registry, a test suite, a FastAPI prediction service, a batch scoring job, a monitoring report, a retraining job, and the CI workflows that tie them together. In the later chapters a network slowdown and a competitor's fiber launch change Fairhollow's data — and we watch our monitoring catch it. Because the data is generated from a fixed seed, every number in this book can be reproduced on your machine.

Versions and Conventions

The code was run with **Python 3.12**, **scikit-learn 1.9**, **pandas 3.0**, **MLflow 3.17**, **FastAPI 0.143**, and **pytest 9** on Windows, using the CPU only. Most of it works on other recent versions too; where a behavior is new or version-specific, a Note says so. Commands are shown for a terminal opened in the

project folder; where Windows PowerShell and macOS/Linux shells differ, both forms are given. Container images and GitHub Actions workflows are written to run on standard Linux runners; they were not executed while writing this book, and their sample output is labeled as illustrative. Numbers are shown in US format, such as `0.7740` and `$59.00`.

> **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 wrong results, outages, or security problems.

Code, file names, and commands appear in `monospace`; long lines are wrapped to fit the page. A caption above a code block names the file it belongs to, such as `src/fairhollow_churn/train.py`, or says **Output** when it shows what a command printed. MLflow prints extra INFO log lines while it works; we trim them from the output shown in the book.

Contents

Preface

PART I — FOUNDATIONS

1 What MLOps Is: From Notebook to Production

2 Framing the Churn Problem: Business Goals, Metrics, and a Baseline

3 Project Structure: The src Layout, Packaging, and Configuration

4 Reproducible Environments: Python Versions, Dependencies, and Lock Files

5 Reproducible Runs: Seeds, Data Fingerprints, and Deterministic Pipelines

PART II — DATA AND FEATURES

6 The Fairhollow Dataset: A Seeded Synthetic Data Generator

7 Data Validation: Schemas, Contracts, and Quality Checks

8 Feature Pipelines with scikit-learn: ColumnTransformer and Custom Transformers

9 Training–Serving Skew and Leakage: Keeping Features Consistent

10 Data and Model Versioning: Hashes, Manifests, and Lineage

PART III — EXPERIMENTS AND THE MODEL REGISTRY

11 Experiment Tracking with MLflow: Runs, Parameters, Metrics, and Artifacts

12 The Training Pipeline: From Config File to Registered Model

13 Evaluation Beyond Accuracy: Thresholds, Calibration, Slices, and Business Cost

14 The MLflow Model Registry: Versions, Aliases, and Promotion

15 Model Packaging: MLflow Models, Signatures, and Safe Serialization

PART IV — TESTING AND AUTOMATION

16 Testing ML Code: Unit Tests for Data, Features, and Pipelines with pytest

17 Testing Models: Quality Gates, Behavioral Tests, and Regression Checks

18 CI for ML with GitHub Actions

19 Continuous Training and Delivery: Pipelines, Triggers, and Release Gates

PART V — DEPLOYMENT AND SERVING

20 Serving Predictions with FastAPI

21 Hardening the Prediction Service: Validation, Errors, Logging, and Performance

22 Containers for ML: Dockerfiles, Images, and Runtime Configuration

23 Batch Scoring: Scheduled Jobs, Partitions, and Idempotent Outputs

24 Deployment Strategies: Shadow Mode, Canaries, A/B Tests, and Rollback

PART VI — MONITORING AND OPERATIONS

25 Monitoring ML Systems: Service Health, Prediction Logs, and Delayed Labels

26 Detecting Drift: PSI, KS Tests, and Prediction Drift

27 Retraining: Triggers, Champion vs Challenger, and Safe Promotion

28 Governance and Case Study: Model Cards, Audit Trails, and the Fairhollow Platform

APPENDICES

A MLOps Cheat Sheet

B Templates and Reference

C Production Checklist

D Glossary

PART I

Foundations

What MLOps really means for a classic machine learning model, how to frame the Fairhollow churn problem in business terms, and how to turn a notebook into a reproducible Python project with a clean layout, a configuration file, pinned environments, and runs you can repeat exactly.

1 · What MLOps Is

2 · Framing the Churn Problem

3 · Project Structure and Configuration

4 · Reproducible Environments

5 · Reproducible Runs

CHAPTER 1 · PART I — FOUNDATIONS

What MLOps Is: From Notebook to Production

A model that scores well in a notebook is the beginning of the work, not the end. In this chapter we look at what changes when a model has to run every month for real people, define MLOps as a set of engineering practices rather than a product you buy, meet the team at Fairhollow Broadband, and set up the Python environment you will use for the rest of the book.

A Model That Worked on Tuesday

Let us start with a story you may recognize. Kwame, an ML engineer at Fairhollow Broadband, a regional internet service provider, spent a week in a notebook. He loaded a file of 6,000 customers exported at the end of June, one-hot encoded a few columns, filled the missing survey scores with the median, trained a logistic regression, and printed a test ROC AUC of 0.7759. The number looked good. He pasted a screenshot into the team chat and moved on to something else.

Over the next month, the questions arrived one by one.

- **Imogen**, Head of Customer Retention, asked for a list of customers to phone next month. Her team can call at most 20% of customers each month. Which customers, exactly, and how many of the people on the list will really cancel?
- **Marisol**, the data engineer who produces the monthly snapshot files, mentioned in passing that the September export would have a new plan code and that she had renamed a column "for consistency".
- **Linnea**, the platform engineer, asked how the customer portal could get a churn score for a single customer while they were on the phone with support. Was there an API? What Python version did it need?
- **Rosalind**, the risk and compliance lead, asked which data the model was trained on, who approved it, and how she could reproduce the score a customer received on a given day.

Kwame tried to rerun the notebook to answer them. Some cells had been run out of order, so the final state of the notebook did not match the code from top to bottom. A library had been upgraded in the meantime. The June file on his laptop had been overwritten with a newer export. He got a slightly different AUC, and he could not explain why.

Nothing in this story is a modeling problem. The logistic regression was fine. Every problem is an *engineering* problem: how the model gets its data, how the training is repeated, how the result is stored and shipped, how anyone knows it is still working, and how people can trust it. That gap,

between "a model that worked on Tuesday" and "a model the business can depend on every month", is what this book is about.

What Changes Between a Notebook and a Production System

Traditional software changes in one main way: people change the code. If the code and its dependencies stay the same, the program behaves the same. A machine learning system has three moving parts instead of one, and each can change on its own schedule.

What changes	Example at Fairhollow	Why it hurts
Data	Metro speeds drop because of congestion; a competitor launches fiber in suburban areas	The model is unchanged, but its inputs and the world it describes are not
Code	Someone edits the feature logic, upgrades pandas, or fixes a bug in a SQL export	The same model file may receive differently shaped inputs
Model	A retrained version replaces the old one	Scores move for every customer, including ones nobody looked at

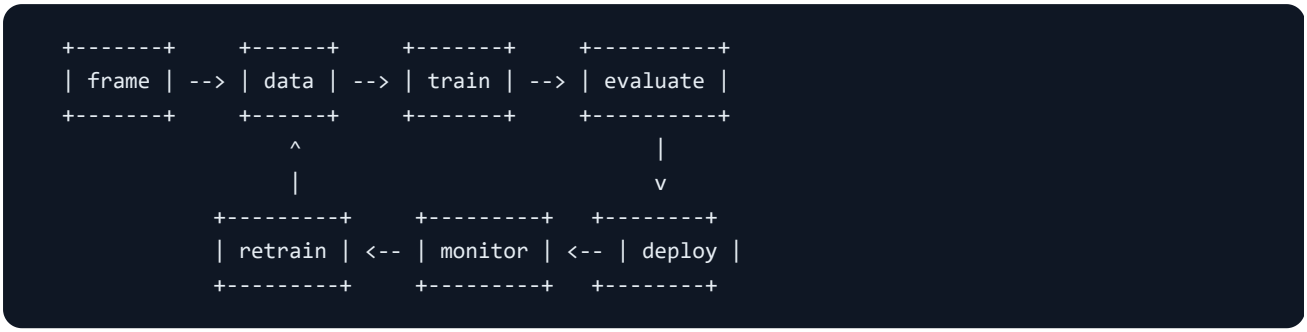
In a notebook, all three are frozen at the moment you press "Run All". In production, they drift apart. The data that arrives in September is not the data the model learned from in June. The code that prepares features for the live API may not be the code that prepared them in training. The model that answers today may be a different version from the one that answered last month, and someone will eventually ask which one was used.

There is a second, quieter difference: **machine learning systems often fail silently**. A web server that crashes returns an error, and someone gets paged. A churn model that receives speeds in percent instead of as a ratio does not crash. It happily returns a probability. In Chapter 9 we will see a customer whose correct score is 0.9211 drop to 0.0 because one field was sent in the wrong unit, with no error anywhere. Much of MLOps is about turning silent failures into loud ones: validation that rejects bad files, tests that check model behavior, monitoring that raises an alert when the inputs or the outcomes shift.

> **NOTE**
>
> When we say "production" in this book, we do not mean anything exotic. For Fairhollow it means two things: a monthly batch job that produces Imogen's call list, and a small HTTP service that returns a score for one customer. Both run without Kwame sitting in front of them, both must give the same answer for the same input, and both must be explainable afterward. That is enough to need every practice in this book.

The Machine Learning Lifecycle

A useful way to organize the work is as a loop rather than a line. A model is never "done"; it is trained, used, watched, and eventually replaced.



Each stage has its own questions and its own chapters:

- **Frame.** What decision will the model support, what exactly is the label, and how will we measure success? For Fairhollow the decision is "whom does the retention team call this month", and the label is "cancelled within 60 days after the snapshot". Chapter 2 is entirely about framing, because a model that answers the wrong question cannot be rescued later.
- **Data.** Where do the inputs come from, what do valid inputs look like, and how do we know a file is the file we think it is? Part II covers generation, validation, feature pipelines, skew and leakage, and versioning.
- **Train.** A training run should be a command with a configuration file, not a sequence of cells. Chapters 3 to 5 make the project and runs reproducible; Chapters 11 and 12 add experiment tracking and a training pipeline.
- **Evaluate.** Is the model good enough *for the decision*? That means thresholds, calibration, performance on slices such as region and contract, and business cost. Chapter 13 covers it, and Chapter 17 turns the most important checks into automated tests.
- **Deploy.** Package the model with its metadata, register it, and serve it through an API or a batch job. Part III ends with the registry and packaging; Part V covers serving, containers, batch scoring, and release strategies.
- **Monitor.** Watch the service, the inputs, the predictions, and, when labels finally arrive, the real outcomes. Chapters 25 and 26.
- **Retrain.** Decide when to train again, compare the new model to the current one on fresh data, and promote it only if it is better. Chapter 27.

The arrow from "retrain" back to "data" matters. Retraining is not the same as starting from scratch: it reuses the same pipeline, the same configuration format, the same tests, and the same gates. That reuse is only possible if the earlier stages were built as engineering artifacts rather than one-off notebook cells. Every shortcut taken in the first pass of the loop has to be paid for on every later pass.

What MLOps Is, and What It Is Not

MLOps is the set of engineering practices that make machine learning systems reproducible, testable, deployable, observable, and accountable over time. The name borrows from DevOps, and many ideas carry over directly: version control, automated tests, continuous integration, repeatable deployments, monitoring, and small, reversible changes. What ML adds is the data and the model as first-class things that need the same care as code.

It helps to be clear about what MLOps is not.

- **It is not a tool.** You cannot install MLOps. A tracking server, a model registry, or a cloud ML platform can support the practices, but a team that uses a registry without ever recording which data trained which model has the tool and not the practice. In this book we use specific open-source tools, but we always explain the practice first so you can map it to whatever your organization uses.
- **It is not only deployment.** Getting a model behind an API is one chapter out of twenty-eight. Most of the failures we will study happen before deployment (a leaked feature, an unvalidated file) or long after it (drift, a competitor's launch).
- **It is not a separate team's job.** Practices only work when the people who build models, move data, run platforms, and use predictions agree on them. In a small company one person may wear all of these hats; the practices stay the same.
- **It is not big infrastructure by default.** Our reference project runs on a laptop with a SQLite file as its tracking store. You can add a shared server, containers, and orchestrators when the team or the load needs them, and we show where those fit.

Why ML needs more than DevOps

A regular service has a specification: given this request, return that response. A churn model has no such specification for individual customers. It is a statistical summary of past behavior, and its quality is only visible in aggregate and only after a delay. At Fairhollow, a customer's label is known 60 days after the snapshot, so a model can be wrong for two months before anyone can measure it. That is why the ML-specific practices exist: validating data before trusting it, tracking experiments so results can be compared honestly, checking behavior rather than exact outputs in tests, and monitoring distributions as an early warning while the real labels are still on their way.

Maturity Levels, in Plain Words

Teams usually grow into MLOps in steps. Several published frameworks describe these steps with numbered levels; we will not tie ourselves to any particular vendor's definition. In plain words, there are three broad stages:

Stage	What it looks like	Typical pain

Manual	Notebooks, a model file copied to a server by hand, retraining when someone remembers	Nobody can reproduce last quarter's model; deployments are rare and scary
Automated training	Training is a command driven by a config file; runs are tracked; tests and a quality gate run on every change	Models are trained reliably, but shipping one still needs manual steps
Automated delivery	CI builds and tests the service; new models are registered, compared, promoted, and deployed through gates, with monitoring and rollback	More moving parts to maintain; needs discipline about approvals

Not every model needs the last stage. A report that is refreshed once a quarter for a handful of analysts may be fine with automated training and a manual release. A model that sets prices for every customer every hour probably is not. The point is to choose the level deliberately and to know which risks you accept.

Fairhollow starts the book at the manual stage: one notebook and one AUC. By the end, it has automated training with a quality gate, a registry with aliases, CI that runs tests and a training smoke test, a monthly retraining workflow with human approval, a container recipe, batch and online serving, and a monitoring report that compares September to June. That is roughly the "automated delivery" stage, built with tools you can run on your own machine.

> **TIP**
>
> When you join a team or start a project, ask three questions to find its current stage: "Can you retrain last month's model from scratch and get the same metrics?", "What has to happen before a new model version reaches users?", and "How would you know if the model stopped working?" Vague answers to any of them point to where the first improvement should go.

The People Around the Model

Throughout the book you will work alongside Kwame, who owns the model and the `fairhollow-churn` repository. The other people at Fairhollow show up when their questions become relevant. They are fictional, but each represents a role you will meet in real projects.

Person	Role	Owns	Typical question
Imogen	Head of Customer Retention	The call list and the 20% monthly capacity	"How many of the people we call will actually leave?"
Kwame	ML engineer	The model and the repository	"Is this version really better than the last one?"
Marisol	Data engineer	Warehouse exports and monthly snapshot files	"What happens if a column changes?"

Linnea	Platform engineer	CI runners, containers, the API deployment	"What does it need to run, and how do I roll it back?"
Rosalind	Risk and compliance lead	Approvals, audit trail, model documentation	"Who approved this, and on which data?"

Notice that only one of the five writes model code. The others are customers of the model, suppliers of its data, or people who must sign off on it. A good share of MLOps is making the model legible to them: a config file Marisol can read, a container Linnea can run, a model card Rosalind can approve, and a call list Imogen can act on.

There is one more character, offstage: **Brightline Fiber**, a fictional competitor that launches fiber service in Fairhollow's suburban areas in the summer of 2026. Brightline is the reason the September data looks different from June in a way none of our features can see. We will meet its effect in Chapter 26.

The Tool Map of This Book

We picked a small set of widely used, open-source tools. Open source matters here for three reasons: you can run every example on your own machine without an account or a bill, you can read the source when the documentation is unclear, and the practices transfer to commercial platforms, which often wrap or imitate these same tools.

Concern	Tool in this book	Main chapters
Data handling	pandas, NumPy	6, 7
Models and features	scikit-learn pipelines	8, 9, 12
Configuration	YAML + Pydantic	3, 12
Experiment tracking	MLflow Tracking	11, 12
Model registry and format	MLflow Model Registry, MLflow Models	14, 15
Tests and linting	pytest, ruff	16, 17
CI and scheduled training	GitHub Actions	18, 19
Online serving	FastAPI, Pydantic, uvicorn	20, 21
Containers	Docker (recipe only)	22
Batch scoring and monitoring	pandas, SciPy, our own modules	23, 25, 26

The versions we tested with are Python 3.12, scikit-learn 1.9, pandas 3.0, NumPy 2.5, MLflow 3.x (tested with 3.17), recent versions of FastAPI and Pydantic 2, and pytest 9. Chapter 4 explains how the project records these exact versions and why the reader of a book, like a colleague joining a team, needs them.

> **WARNING**
>
> Two tools in the map could not be executed on the machine used to write this book: Docker and GitHub Actions runners. The Dockerfile and the workflow files are complete and every command inside them was run locally, but the container builds and the CI logs you will see are labeled **Output (illustrative)**. Treat those outputs as "what you should see something like", not as recorded results. Everything else in the book was run, and the outputs are pasted from those runs.

We also mention tools we do not use in depth, such as orchestrators (Airflow, Prefect, Dagster), feature stores (Feast), data versioning tools (DVC, lakeFS), validation libraries (pandera, Great Expectations), and monitoring products (Evidently, NannyML, Prometheus with Grafana). For each, we explain the problem it solves and where it would fit in our architecture, so you can recognize when you need one. We deliberately do not turn the book into a vendor tour.

Where We Are Going: The Final Architecture

Here is the system we will have built by Chapter 28. Do not worry about the details yet; every box gets its own chapter. The point is to see how the pieces connect, and that each arrow is a place where something can go wrong.



A few design choices are already visible in the diagram:

- **One validation step guards every entry point.** The same data contract runs before training and before batch scoring, and the API enforces an equivalent contract on each request. A broken file stops the pipeline with a report instead of producing quiet nonsense.
- **The model artifact is the whole pipeline.** Feature engineering, imputation, encoding, and the classifier travel together, so training and serving cannot disagree about how features are built.
- **Serving never talks to the tracking server.** The champion model is exported to a plain folder with its metadata. The API and the batch job load that folder. If the tracking server is down, predictions continue.
- **Promotion is a decision with evidence.** A new model becomes champion only after it passes the quality gate and beats the current champion on the same fresh data. The registry tags record which version it replaced and why.

How to Read This Book

The book is organized in six parts that follow the lifecycle: foundations, data and features, experiments and the registry, testing and automation, deployment and serving, and monitoring and operations. Each chapter adds files to one project, `fairhollow-churn`, so the chapters build on one another. If you already know a topic, you can skim its chapter, but keep the files: later chapters import them.

Conventions

- **Commands** are shown as `python -m ...`, which works the same way on Windows, macOS, and Linux once your virtual environment is active. When shells differ, we show both PowerShell and bash.
- **File captions** above code blocks name the file, such as `src/fairhollow_churn/config.py`. The final version of every file matches the complete project; when a chapter shows a simpler early version, it says so.
- **Outputs** are pasted from real runs. Run IDs, timestamps, and request IDs will differ on your machine, and so may timings. Metrics computed from our seeded data should match to the last decimal place if you use the same library versions. We trim MLflow's informational log lines from outputs and say so in each chapter where that happens.
- **Business numbers**, such as the cost of a call or how many contacted customers stay, are fictional assumptions made up for Fairhollow. We label them every time, because it is easy to forget that a nice-looking dollar figure rests on guesses.
- **The data is synthetic.** It is generated with fixed seeds, so your files will be byte-for-byte identical to ours. It does not describe any real broadband market; any churn rate you see is a property of Fairhollow only.

Who this book assumes you are

We assume you can write Python functions and modules, create a virtual environment, run scripts from a terminal, and that you have trained at least one simple model, for example with scikit-learn. We do not teach machine learning algorithms: when we need logistic regression or gradient boosting, we use them and explain only what matters for engineering decisions. If terms like "train/test split" or "ROC AUC" are new to you, a short introduction to scikit-learn first will make this book much easier.

Setting Up Python 3.12 and a Virtual Environment

Before writing any project code, we need a clean, isolated Python environment. This is the first reproducibility practice in the book, and it costs less than a minute.

Check your Python version

The project is tested with Python 3.12. Other recent versions will probably work, and the project declares that it accepts 3.11 or newer, but if you want your numbers to match ours exactly, use 3.12. On Windows, the Python launcher `py` lets you pick a version explicitly; on macOS and Linux, the interpreter is usually called `python3` or `python3.12`.

PowerShell

```
py -3.12 --version
```

Terminal

```
python3.12 --version
```

Output

```
Python 3.12.0
```

Any 3.12.x patch release is fine. If the command is not found, install Python 3.12 from python.org or with your system's package manager, then try again.

Create the project folder and the virtual environment

A **virtual environment** is a folder that contains its own Python interpreter link and its own `site-packages` directory. Packages you install while it is active go into that folder and nowhere else. This keeps the project's dependencies separate from other projects and from the system Python, and it means you can delete and recreate the environment at any time without fear.

PowerShell

```
mkdir fairhollow-churn
cd fairhollow-churn
py -3.12 -m venv .venv
```

```
.venv\Scripts\Activate.ps1
```

Terminal

```
mkdir fairhollow-churn
cd fairhollow-churn
python3.12 -m venv .venv
source .venv/bin/activate
```

In the Windows Command Prompt (cmd.exe) the activation command is `.venv\Scripts\activate.bat`. After activation, your prompt usually shows `(.venv)` at the start, and the plain command `python` refers to the environment's interpreter. From now on we write `python` everywhere and assume the environment is active.

Check that you are really inside the environment. The interpreter path should point into your project's `.venv` folder, and a fresh environment contains almost nothing:

Terminal

```
python -c "import sys; print(sys.prefix)"
python -m pip list
```

Output

```
C:\Users\you\projects\fairhollow-churn\.venv
Package Version
------- -------
pip     23.2.1
```

Your path and your pip version will differ; what matters is that the path ends in `.venv` and that the package list is short. If you see dozens of packages, the environment is not active, or you created it with an option that includes system packages.

> **NOTE**
>
> On Windows, PowerShell may refuse to run `Activate.ps1` because script execution is disabled on your machine. Execution policy is a security setting that your organization may manage, so check with whoever administers your machine before changing it. You do not need activation to use the environment: calling the interpreter by its path, `.venv\Scripts\python.exe -m pip list`, works the same way, and you can use the cmd.exe activation script instead.

Install the tested versions

Create a file called `requirements-lock.txt` in the project folder with the exact versions we tested. Chapter 4 explains why this file exists alongside the looser version ranges in `pyproject.toml`, and what it does and does not guarantee. For now, think of it as "the versions the book was checked against".

requirements-lock.txt

```
# Exact versions the reference project was tested with (Python 3.12, Windows, CPU only).
# Install with:  python -m pip install -r requirements-lock.txt
numpy==2.5.3
pandas==3.0.6
scikit-learn==1.9.1
scipy==1.18.1
joblib==1.6.0
pyyaml==6.0.3
pydantic==2.13.5
mlflow==3.17.0
skops==0.16.0
fastapi==0.143.0
starlette==1.7.0
uvicorn==0.54.0
pytest==9.1.1
httpx==0.28.1
```

Terminal

```
python -m pip install -r requirements-lock.txt
python -m mlflow --version
```

Output

```
python -m mlflow, version 3.17.0
```

The install downloads MLflow and its dependencies, so it takes a while the first time. We use `python -m pip` rather than a bare `pip` command on purpose: it guarantees that the pip you run belongs to the Python you mean, which is the most common source of "I installed it but Python cannot find it" problems.

> **TIP**
>
> Add `.venv/` to your `.gitignore` right away (Chapter 3 gives the full file). A virtual environment contains absolute paths and platform-specific binaries; it belongs on your machine, never in the repository. What belongs in the repository is the *recipe* to recreate it: the Python version, `pyproject.toml`, and the lock file.

Initialize version control

Finally, make the folder a Git repository. Everything in this book assumes that code and configuration are under version control, and from Chapter 5 onward each training run records the Git commit it was trained from.