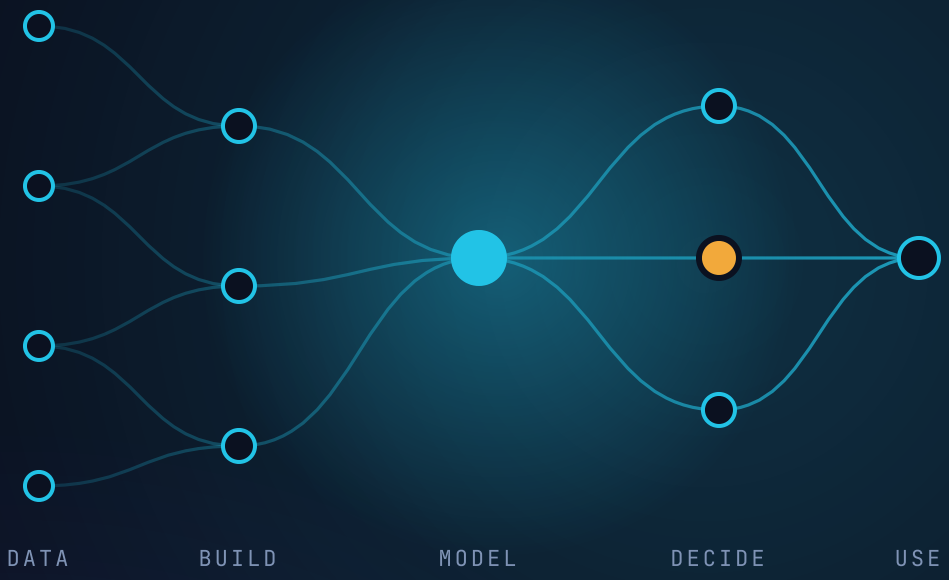


Governed MLOps on Google Cloud

Take models from notebook to production

WITH PIPEWRIGHT

KUBEFLOW · AGENT PLATFORM · BIGQUERY ML · RAY

Rakesh Gupta

Governed MLOps on Google Cloud

Take models from notebook to production with Pipewright

Rakesh Gupta

Governed MLOps on Google Cloud
Take models from notebook to production, with Pipewright

First edition, November 2026. Free edition.

Copyright © 2026 Pipewright. This free edition is licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International licence (CC BY-NC-ND 4.0): you may copy and share it unchanged and free of charge, with credit to the author. https://creativecommons.org/licenses/by-nc-nd/4.0/

The running example uses the *Online Retail II* dataset: Chen, D. (2012), UCI Machine Learning Repository, https://doi.org/10.24432/C5CG6D, licensed under CC BY 4.0.

The author created Pipewright. Pipewright Enterprise is a commercial product of Fluxor Software; the open-source core used in this book is free under the Apache License 2.0.

Google Cloud, BigQuery, Vertex AI, Agent Platform, Google Kubernetes Engine and Bigtable are trademarks of Google LLC. Kubeflow, KServe and Ray are trademarks of their respective owners. This book is not endorsed by Google.

Every effort has been made to make the commands and configuration in this book correct for Pipewright 1.3.0. They are provided as they are, without warranty.

getpipewright.dev

Contents

Preface

How to use this book

PART 1. A GOVERNED PATH

1 Why most models never reach production, and the ones that do drift away

2 One use case, one YAML file

PART 2. DATA YOU CAN TRUST

3 Data contracts and quality gates

4 Features without leakage

5 Reproducible runs

6 Synthetic data for MLOps

PART 3. PIPELINES AND TRAINING

7 Pipelines that release like software

8 Models for SQL teams: BigQuery ML in production

9 Training at scale

10 Evaluation gates

11 Features that earn their place

PART 4. DEPLOYMENT

12 Promotion between environments

13 Serving, canaries and A/B tests

PART 5. OPERATIONS

14 Monitoring drift and performance

15 Retraining with a person in the loop

16 CI/CD and approvals in GitHub

PART 6. GOVERNANCE

17 Lineage: every model's full story

18 Evidence, audit and running a fleet

PART 7. PUTTING A MODEL TO WORK

19 From endpoint to app

PART 8. BUILDING WITH AI

20 Building with a coding agent

APPENDICES

A Your Google Cloud project

B Every setting of a use case

C Running it on GKE

D Google Cloud at a glance

About the author · Data and credits · Where to go next

Preface

Most machine learning models never reach production. Of the ones that do, many are quietly switched off a year later, not because the model was wrong, but because nobody could keep it right. I have watched this happen in teams of every size, from start-ups to some of the largest companies in their industries, and the reasons are almost never about the model.

They are about everything around it. Where did the data come from, and will it look the same next week? Which code produced this version, and can we produce it again? Who decided it was good enough? What is it doing now that the season has changed? These questions have answers, but only if the path a model takes to production is built to record them. In most organisations it is built by hand, differently for each model, by whoever needed it that month.

This book is about building that path once, so that every model takes it: data checked before training, models compared with the one already in use, a person approving each change, drift noticed early, retraining decided rather than scheduled, and every version traceable from its data to its use. On Google Cloud, with real data, end to end.

The path is built with Pipewright, an open-source platform I created for exactly this. It runs on the services Google Cloud already provides, from Agent Platform pipelines and BigQuery ML to Kubeflow and Ray on GKE, and it turns one file per use case into a governed lifecycle. You do not have to use Pipewright to use what is in this book: the ideas hold whichever tools fill them. But a real tool, running real data, makes the ideas concrete, and every command in these pages runs as printed.

The running example is a small online gift retailer in the United Kingdom, whose two years of transactions are published openly. Its Christmas peak, its customers who drift away, its analysts who know SQL better than Python and its single cloud engineer are, I suspect, more like your organisation than any demonstration dataset.

I hope this book saves you the year it usually takes to learn these lessons.

Rakesh Gupta
London, November 2026

How to use this book

Who it is for. Data scientists and ML engineers who want their models in production and kept there; analysts who build models in SQL; platform engineers who run Google Cloud for those teams; and the leads who are accountable for what the models do. No chapter assumes you have run machine learning in production before.

How it is organised. Eight parts follow the life of a model. Part 1 sets out the problem and the shape of the solution. Part 2 is about data you can trust, including synthetic data. Part 3 covers pipelines and training, for code and for SQL. Part 4 takes models to production, Part 5 keeps them there, Part 6 makes the whole path visible and accountable, Part 7 puts a model to work in a web app, and Part 8 shows how a coding agent can help you build all of it, safely.

Each chapter opens with the retailer's problem, explains the idea in general terms, shows how Pipewright handles it, and ends with **Try it**, a hands-on section you can follow on your own Google Cloud project, and **In short**, a summary of the chapter.

What you need. For Chapter 1, only Python 3.11 or later. From Chapter 2 on, a Google Cloud project you can create resources in; Appendix A explains how to set one up and what it costs. The example code and configuration are on GitHub at github.com/fluxorsoft/pipewright-ebook, one complete folder per chapter.

Conventions. Commands are shown as you type them. Configuration is YAML, and a line ending in `# ...` explains the line. Where something is part of Pipewright Enterprise rather than the open-source core, the text says so.

Chapter 1. Why most models never reach production, and the ones that do drift away

The retailer's starting point

Every year, in the weeks before Christmas, a small online gift retailer in the United Kingdom sells more than twice as much as it does in a typical month. Its customers, many of them shops buying stock to resell, place bigger orders, more often, and then go quiet in January. The business knows this pattern by heart. Its machine learning models do not, unless someone teaches them, and keeps teaching them as the pattern moves.

This book follows that retailer. Its data is real: two years of transactions, published openly so that anyone can study them. The team around the data is a story we tell, but every problem the team meets is one you will meet, and every number we quote comes from the data itself. By the end of the book the retailer runs five machine learning use cases on Google Cloud, released, approved, watched, retrained and traced the same way. This chapter is about why that is hard, and why most organisations never get there.

1.1 The retailer and its data

The retailer sells unique gift items for all occasions, online only. Between 1 December 2009 and 9 December 2011 it recorded 1,067,371 invoice lines: 5,305 products bought by 5,942 identified customers in 43 countries. The United Kingdom accounts for 85% of its sales by value. Each line says what was bought, how many, at what price, when, by which customer and from which country. That

is all. No web logs, no marketing data, no customer profiles: the plain transaction history most businesses have.

It is enough to see the problem this book is about. Figure 1.1 shows the retailer's sales week by week.

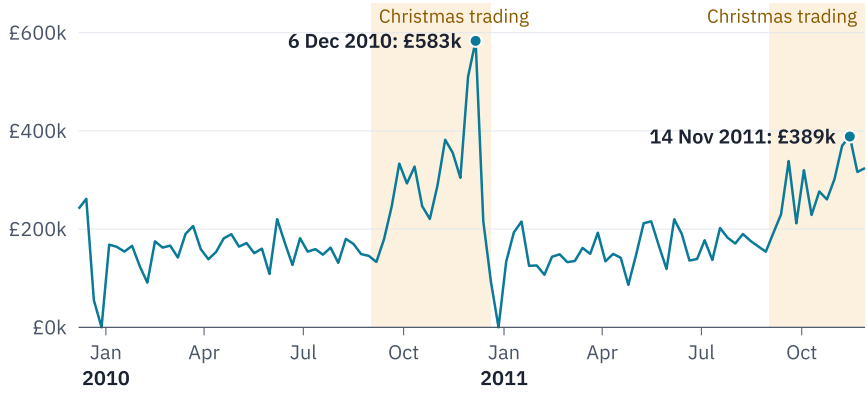


Figure 1.1 The retailer's weekly sales, by value, cancellations left out. Sales more than double from September to early December, then drop to nothing while the business closes for Christmas. Data: Online Retail II, UCI.

Three things stand out, and each becomes a model later in the book.

- **The year has a shape.** November sales are 2.2 times the average month from January to August, in both years. A forecast of next week's demand that ignores the season is wrong exactly when being wrong costs most.
- **The shape moves.** The 2010 peak came in the week of 6 December, at £583,000. In 2011 the biggest week was 14 November, at £389,000: earlier and lower. A model trained on 2010 and left alone would have planned for a December that never came.
- **Customers leave quietly.** Of the 4,266 customers who bought something in the first twelve months, 1,540 (36%) bought nothing at all in the next twelve. Nobody cancels a

subscription at a shop: they simply stop coming back. Knowing who is about to stop, while there is still time to do something, is worth a great deal.

Each of these is a reasonable first project for a data team. A forecast of demand per product. Segments of customers by how recently, how often and how much they buy. A score for each customer's risk of lapsing. None of them needs a breakthrough in machine learning. And yet, in most organisations, projects like these stall.

1.2 What happens after the notebook

Imagine the retailer's first data scientist builds the lapse model. It takes a few weeks. She pulls a year of transactions into a notebook, works out for each customer how long it has been since their last order, how often they order and how much they spend, labels the customers who did not come back, and trains a classifier. It ranks customers well on the months she held back. She shows the results, everyone agrees it is useful, and someone asks the question that ends most projects: *so how do we use it every week?*

A model in a notebook has done the first third of its life. Figure 1.2 shows the rest.

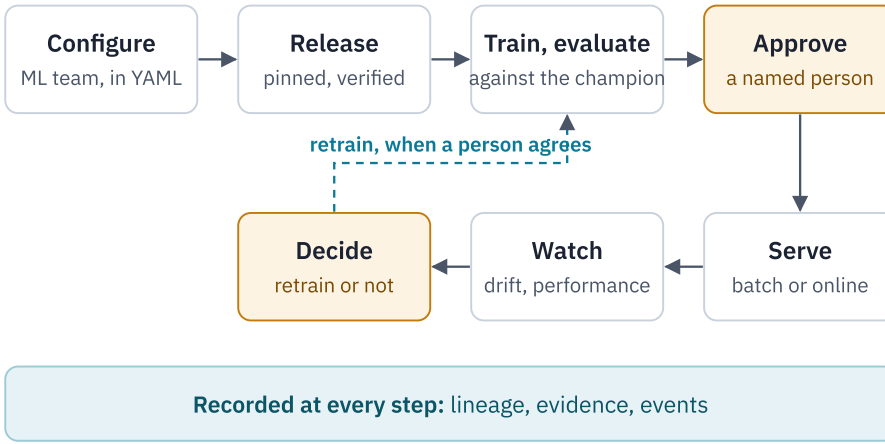


Figure 1.2 What happens to a model after the notebook. Two steps belong to people, not machines: approving a model before it serves, and deciding whether a drifting model should be retrained. Every step leaves a record.

The model has to be **configured** so that it can be run by something other than its author. It has to be **released**: the exact code, libraries and settings that produced it fixed, so that next week's run is the same run. It has to be **trained and evaluated** on a schedule, and compared with the model already in use, because a new model is only worth deploying if it is better than the old one: first on held-out history, and in the end on real customers, in an A/B test (Chapter 13). Someone has to **approve** it. It has to **serve**: score every customer each Monday, or answer a request while a customer is at the checkout. It has to be **watched**, because the data it sees will drift away from the data it learned from. And when it drifts, someone has to **decide** whether to retrain it, which takes us back to training.

Underneath all of it runs a record: which data, which code and which settings made each version, who approved it, and what it did once it was live. Nobody asks for that record until the day they need it, and then it is too late to create it.

1.3 Why models stall before production

Look at Figure 1.3. The notebook covers four boxes. Production needs those four and twelve more.

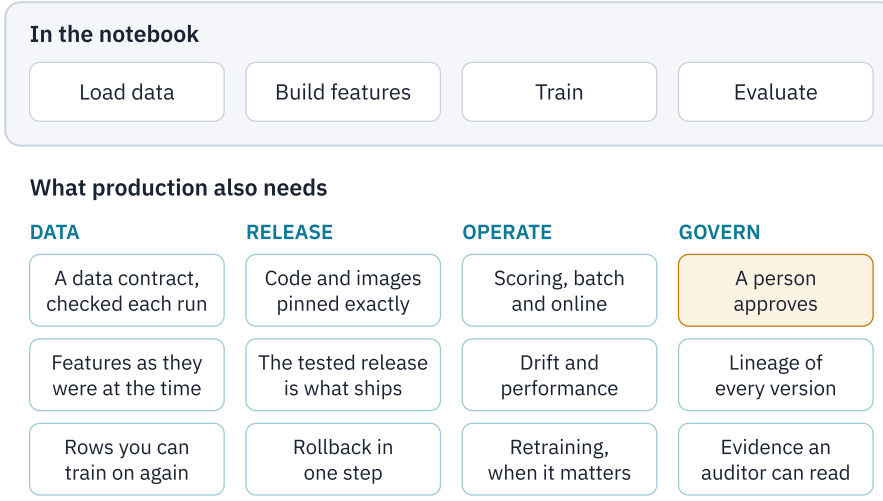


Figure 1.3 The gap between a working notebook and a model in production. Each of the twelve boxes is a small project; together they are the reason most models never leave the notebook.

None of the twelve is difficult on its own. A data contract is a list of rules. Pinning a release is a matter of recording versions. A drift check is a statistic. The difficulty is that there are twelve of them, they depend on each other, and in most organisations each team builds them again, by hand, for each model. Three patterns repeat.

Every team builds its own glue. The lapse model needs a scheduled job, a place to store versions, a way to score customers and a check for drift. The forecasting team, a month later, builds the same things differently. Two years on there are a dozen models, each with its own scripts, its own conventions and one person who understands it.

Decisions happen in email. Whether a new version is good enough to replace the old one is a judgement call. In most teams it happens in a chat thread or a meeting, and the model is deployed by whoever has the permissions. When a customer complains, or an auditor asks, nobody can show who decided what, or on which evidence.

Nothing is recorded as it happens. Which rows was version 4 trained on? Was it the same feature code as version 3? Which model was scoring customers on 14 November? These questions have answers, but only if something wrote them down at the time. Rebuilding them afterwards takes days, and the answer is rarely certain.

If you recognise your own organisation in one of these patterns, you are in good company: they are what happens by default, when each team solves its own problem well and nobody owns the path between them.

The usual response is to hire a platform team to build the missing twelve boxes once, for everyone. That is the right instinct, and it takes a year. This book is about getting the same result without that year.

> **Pause and think.** Think of the last model your organisation built. Which of the twelve boxes did it get, and who built them? If its author left tomorrow, could someone else retrain it next month?

1.4 Why the ones in production drift away

Suppose the retailer gets its lapse model into production anyway. It scores every customer on Monday mornings, and the account team

calls the ones most at risk. For a few months it works.

Then the season turns. In September orders double, regular customers order more often than usual, and the model, which learned that frequent ordering means a loyal customer, marks almost everyone as safe. In January the opposite happens: customers who always go quiet after Christmas look as if they are leaving, and the account team spends a week calling shops that were always going to come back in February. Nobody changed the model. The world it was describing changed under it.

This is **drift**, and it is the normal state of a model in production, not a failure. The retailer's own data shows two kinds of it. The season moves every year, so the inputs a model sees in November look nothing like the inputs it saw in July. And the season itself shifts, as the earlier, lower peak of 2011 shows, so even a model that understands Christmas has to relearn which Christmas it is in.

Drift means a model in production needs three things a model in a notebook never does: something that **notices** when the data has moved, someone who **decides** whether it matters, and a way to **retrain and redeploy** without starting the project again. A team without those quietly turns off its models one by one, as people stop trusting them.

So when you plan a model, plan its second year, not only its first release. The question is not whether it will drift, but who will notice, and what they will be able to do about it.

> **Pause and think.** Which of your models depends on a season, a price, a policy or a customer mix that has changed since it was trained? How would you find out today, and how long would it take?

1.5 Two teams, one path

The twelve boxes split naturally between two kinds of work, and so between two teams. Figure 1.4 shows the split this book uses.

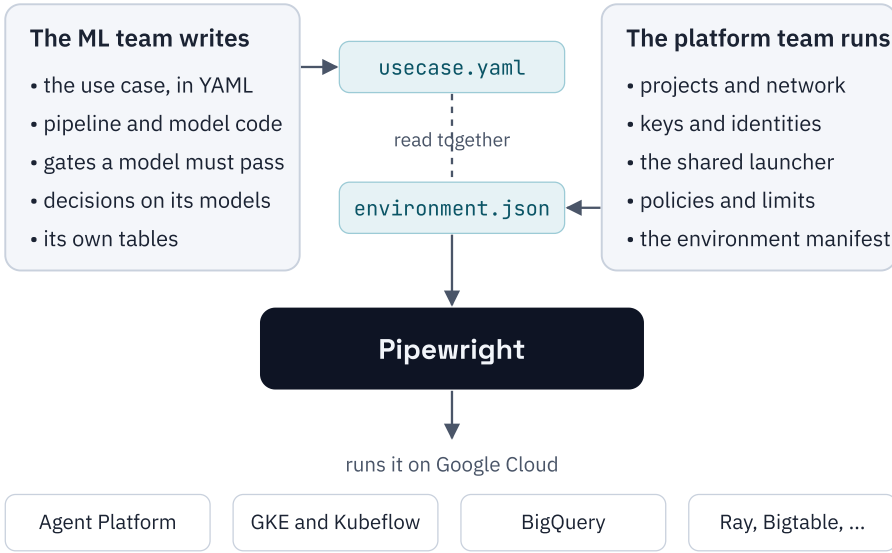


Figure 1.4 The ML team describes its use case in a file; the platform team describes the environment in a manifest. Pipewright reads both and runs the use case on Google Cloud. Neither team needs to know the other's details.

The **ML team** knows the problem: which data, which model, what good enough means, and when a model should be replaced. It writes that down, in one file per use case, along with its own model code.

The **platform team** knows the environment: the Google Cloud projects, the network, the encryption keys, the service identities, the policies and the limits. It publishes that once, as an environment manifest, and every use case runs inside it.

The two meet in the middle, through files rather than tickets. The ML team does not need to know how the network is built, and the

platform team does not need to know what a lapse model is. That separation is what lets a small platform team support many models, and what lets a new model reach production in days instead of months.

At the retailer, the ML team is one data scientist and two analysts who know SQL well. The platform team is one cloud engineer who also looks after the website. The split still holds: the roles exist even when the people overlap.

1.6 What “governed” means in this book

“Governance” often means paperwork that slows teams down. In this book it means four properties of the path a model takes, each of which saves time rather than costing it.

Property	It means	Without it
Repeatable	Every run of a use case uses exactly the code, settings and data rules that were released	“It worked last week” becomes a research project
Reviewed	A named person approves each model before it serves, and each retraining before it happens	Models change because someone had permission, not because someone decided
Recorded	Each version knows its data, its code, its evaluation, its approvals and where it serves	Answering “where did this come from?” takes days
Reversible	Any release or model can be rolled back in one step, to a version that is known to work	A bad model stays live while someone works out how to undo it

Every chapter that follows adds one or more of these to the retailer’s path. None of them needs a separate governance process: they come from running the path the same way every time.

1.7 The retailer's five use cases

By the end of the book the retailer runs these, each a use case of its own:

Use case	The question it answers	Built by	Where in the book
`customer-segments`	Which groups of customers behave alike?	the analysts, in SQL	Chapter 8
`demand-forecast`	How many of each product will sell next week?	the analysts, in SQL	Chapters 8 and 16
`lapse-risk`	Which customers will not buy again within 90 days?	the data scientist	every chapter from 2 on
`checkout-offer`	Which offer should a known customer see at checkout?	the data scientist	Chapters 13 and 19
`synthetic-customers`	Can an outside agency study customer histories without seeing real ones?	data and governance	Chapter 6

Two are built entirely in SQL, by analysts who never write Python. One answers in real time. One produces data rather than predictions. All five go through the same path: released the same way, approved the same way, watched the same way and recorded the same way.

1.8 Where Pipewright fits

The path in this book is built with **Pipewright**, an open-source MLOps platform for Google Cloud. A use case is one folder: a YAML file that describes it, the model's own code, and the schemas of its tables. Pipewright builds, releases, runs, monitors and retrains it, with a person in every decision that matters, and records what happened.

It runs on the services Google Cloud already provides rather than replacing them. Figure 1.5 shows one environment as Pipewright uses it: what the platform team sets up once, what every run uses, where the data, the models and the records live, and the chapter that introduces each service. You do not need to know any of them yet. Each chapter introduces the services it uses, in a short box marked **On Google Cloud**, just before it uses them, and Appendix D lists them all with their documentation.

Read Figure 1.5 from the top. A release puts the team's images and pipeline templates in Artifact Registry and its contract in Cloud Storage. A run request reaches the launcher, a small shared service on Cloud Run, which starts the pipeline on Agent Platform Pipelines as the team's own identity. The pipeline reads its data and features from BigQuery, writes its tables back there, and registers the model it trained in Model Registry; a person decides which version becomes the one that serves. Logs, alerts and an event log record what happened, so the questions of Section 1.3 have answers.

That is the case for a tool like Pipewright. Every box in the figure is a Google Cloud service you could wire up yourself, and every team that does wires them a little differently. Pipewright is the wiring, done once, the same way for every use case: the release, the launcher, the identities, the gates, the records. The same use case file runs on Agent Platform or on Kubeflow in GKE, trains in containers, on Ray or in BigQuery ML, and serves on Agent Platform endpoints or KServe.

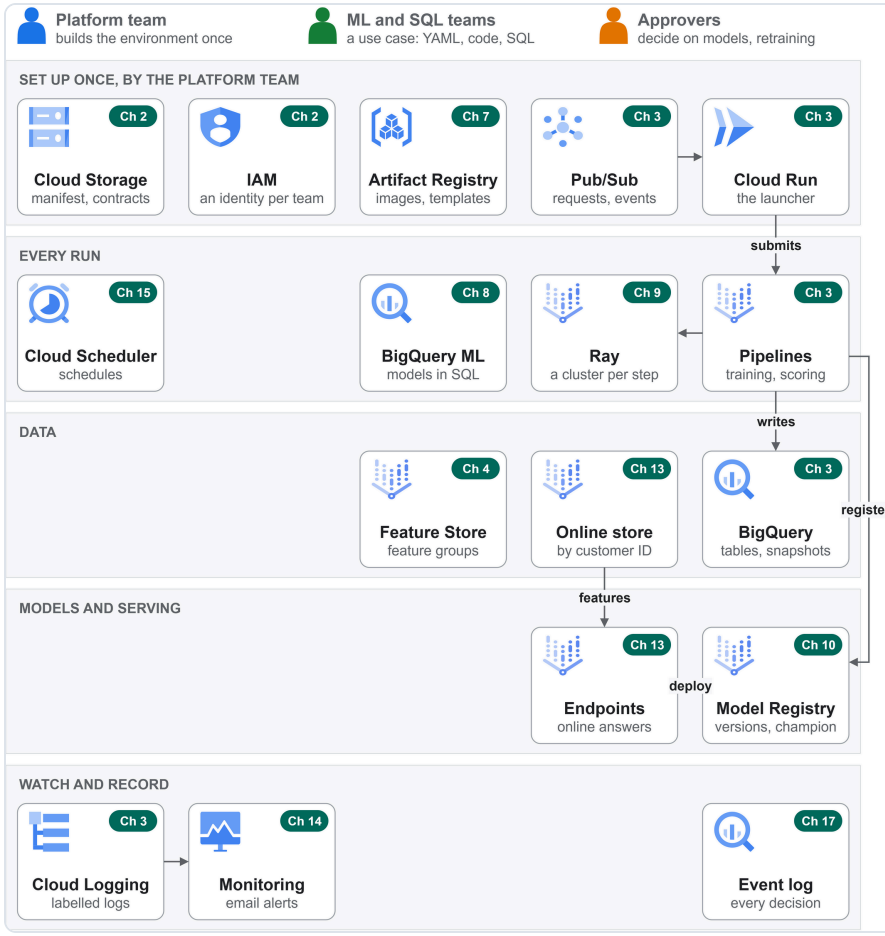


Figure 1.5 One environment on Google Cloud, as Pipewright uses it, with the chapter that introduces each service. The platform team builds it once; ML and SQL teams work only with configuration, code and the pipewright command. Pipelines are Agent Platform Pipelines (formerly Vertex AI Pipelines), Ray runs on Agent Platform, and the online store is the Feature Store's, on Bigtable. In a regulated environment the platform adds Cloud KMS keys and a VPC perimeter; on private GKE, Kubeflow Pipelines, KServe and KubeRay take the places of the pipelines, endpoints and Ray (Appendix C).

The core of Pipewright is open source under the Apache 2.0 licence. An Enterprise edition adds what larger organisations ask for around it: a private-by-default environment blueprint, write-once evidence packs, a fleet view across every environment, and governed

releases of synthetic data. Where a chapter uses an Enterprise feature, it says so. This book uses Pipewright 1.3.0. Its chapters were first built on 1.1.0, and writing them found gaps that 1.2.0 fixed; where a chapter met one, a short note marked **Since Pipewright 1.2.0** says what changed. Pipewright 1.3.0 added AI build assistance, the subject of Chapter 20, and changed nothing the earlier chapters rely on; where it adds something useful to one of them, a note marked **Since Pipewright 1.3.0** says so.

You do not need Pipewright to use the ideas in this book: the twelve boxes are the same whichever tool fills them. But seeing them filled, for real data, end to end, is the fastest way to understand them.

Thinking it through

On where to start. You do not need all twelve boxes before your first model ships. You need to know which ones you are missing, and to add them in the order your risks demand: usually a release you can repeat and roll back first, then a person in the decisions, then monitoring.

On who owns the path. Most stalled models have an owner and a platform, but nobody who owns the path between them. If you cannot name that person, or that team, that is the first gap to close.

On governance. If governance in your organisation means forms filled in after the fact, try the four properties above as a test instead: a path that is repeatable, reviewed, recorded and reversible produces the evidence as it runs.

Try it: your first use case

You need Python 3.11 or later. No Google Cloud project yet: this first step runs on your own machine.

Install Pipewright and check it:

```
python -m venv .venv
source .venv/bin/activate          # on Windows:
.venv\Scripts\activate
pip install "pipewright-mlops[gcp]==1.3.0"
pipewright --version               # pipewright 1.3.0
```

Create a folder for the retailer's MLOps repository and the first use case. `--from minimal` starts with only what is required:

```
mkdir retailer-mlops && cd retailer-mlops && git init
pipewright new-usecase --name lapse-risk --tenant retail \
  --owner ml-team --from minimal
```

Open `usecases/lapse-risk/usecase.yaml`. It is short: the team that owns the use case, the tenant (the share of the environment the platform team gives this team), the training pipeline, the model's name, what starts a run, and one evaluation gate:

```
  evaluation:
    gates:
      - { metric: auc, min: 0.7 }   # below this: not registered
```

That one line is the first piece of governance in this book: a model that does not reach the bar is never registered, whoever trained it. Ask Pipewright what any key means:

```
pipewright explain evaluation.gates
pipewright explain model.promotion
```

In the next chapters this use case grows: real features from the retailer's data, a label window that defines what "lapsed" means, a

pipeline, gates, releases, and in time a production environment on Google Cloud.

In short

- The retailer's data shows the three problems most models face: a season that doubles demand, a season that moves, and customers who leave without saying so.
- A notebook covers four of the sixteen things a model in production needs. The other twelve are why most models never leave the notebook.
- Models in production drift as the world changes. They need something to notice, someone to decide, and a way to retrain without starting again.
- The work splits between an ML team, which describes each use case, and a platform team, which describes the environment. Files, not tickets, join them.
- Governed means repeatable, reviewed, recorded and reversible: properties of the path, not extra process.

Next: Chapter 2. One use case, one YAML file. What goes in the file, what stays out, and why describing a use case in configuration lets two teams work at their own speed.

Data: Chen, D. (2012). Online Retail II [Dataset]. UCI Machine Learning Repository. https://doi.org/10.24432/C5CG6D. Licensed under CC BY 4.0. Figures computed by `content/scripts/retail_facts.py`.

Chapter 2. One use case, one YAML file

Configuration as the contract between two teams

The retailer's lapse model works in its notebook, and the team wants it to run every Monday. Before anyone can schedule it, someone has to answer a list of questions. Which table does it read? Which service account runs it? What score must a new version beat? Which day does it run, and in which time zone? Who is allowed to make a new version the one that scores customers?

In the notebook, the answers are everywhere. The table name is a string in cell 3. The score to beat is a number in cell 14. The schedule does not exist yet; when it does, it will be a line in a cron file on someone's machine. The service account is in a shared document the cloud engineer wrote last spring. Each answer is a decision someone made, and none of them can be seen, reviewed or changed without reading the code.

This chapter is about putting those decisions in one place: a use case folder whose YAML files describe what the model is, what it needs and what it must prove, while the code describes only how it learns.

2.1 Settings scattered are decisions hidden

A setting buried in code has three problems, and the retailer meets all of them in its first month.

Nobody can review it. When the data scientist lowers the score a model must reach, from 0.75 to 0.70, the change is one character in a notebook. Nobody notices until the model that should have been rejected is scoring customers.

It differs between environments by accident. The test copy of the notebook reads the test dataset because someone edited the cell. The production copy reads production because someone remembered to edit it back. One day someone forgets.

Nothing else can read it. The platform needs to know which identity runs the model, which tables it writes and when it runs. If those live in Python, the platform has to run the code to find out, or ask the person who wrote it.

The cure is the same for all three: decisions go in configuration, which people review and machines read; mechanics stay in code, which only the team needs to understand.

2.2 What belongs in configuration, and what stays in code

The line is not always obvious, so this book uses a simple test. A setting belongs in configuration if a reviewer would want to approve a change to it, if it can differ between environments, or if the platform has to read it. Everything else is code.

Decision	Where it lives	Why
What the input data must look like	configuration (`data.contract`)	a change to it changes what the model may learn from
The score a new model must reach	configuration (`evaluation.gates`)	a reviewer should approve a lower bar
When the model runs	configuration (`triggers`)	it differs between test and production
Who makes a model the champion	configuration (`approvals`)	it is a governance decision, not an implementation detail
Which tables it writes, and their shape	configuration (`outputs`, `schemas/`)	the platform creates them
The model's architecture and training loop	code (`components/`)	only the team needs to read it
How features are computed from raw columns	code, for now	Chapter 4 moves the reusable part into feature tables

Configuration is not a place to hide code. A YAML file full of conditions and expressions is code with worse tools. The configuration in this book holds facts and decisions: names, thresholds, schedules, lists. The logic stays in Python or SQL, where it can be tested.

When you are unsure where a setting goes, ask who would want to be told if it changed. If the answer is someone other than the person changing it, it is a decision, and it belongs where that someone can see it.

> **Pause and think.** Pick a model your team runs. Where is the score it must reach written down, and who would notice if it changed tomorrow? If the answer is "in a notebook" and "nobody", that is the first setting to move.

2.3 The anatomy of a use case

A use case is one folder. Figure 2.1 shows the one Chapter 1 created for `lapse-risk`, with what each part holds.

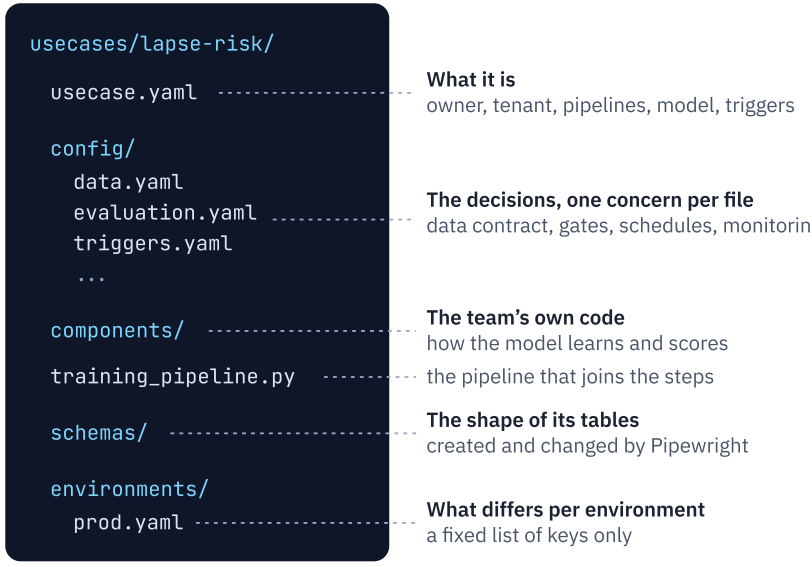


Figure 2.1 One use case, one folder. The YAML files hold the decisions, one concern per file; the code holds how the model learns; schemas describe its tables; the environments folder holds the few settings that differ per environment.

`usecase.yaml` says what the use case is: its name, the team that owns it, the tenant it belongs to, its pipelines, its model, what starts a run. It is short on purpose. Its `include` line pulls in the rest:

```
spec:
  tenant: retail
  include: [config/*.yaml]
```

`config/` holds the decisions, one concern per file: the data contract in `data.yaml`, the gates in `evaluation.yaml`, the schedules in `triggers.yaml`. Each section lives in exactly one file, so there is never a question of which copy wins. A reviewer who cares about the gates opens one file.

`components/` and the pipeline file are the team's own code. Pipewright does not dictate how the model is built: any Python framework works. It asks only that each step reads the parameters the configuration supplies and writes a record of what it evaluated.

`schemas/` describes the tables the use case writes. The team never creates a table by hand: Pipewright creates them, and later applies changes that only add, never break.

`environments/` holds the settings that differ between environments. Section 2.5 explains why it is short.

2.4 The other half: the environment manifest

Look at the use case again. It names a tenant, `retail`, but no project, no dataset, no service account and no bucket. Those belong to the platform team, and they live in the other half of the contract: the **environment manifest**.

The platform team publishes one manifest per environment. It says which Google Cloud project the environment is, where its registries and buckets are, which orchestrator runs pipelines, and, for each tenant, which identity runs its pipelines and which dataset holds its tables:

```
"tenants": {
  "retail": {
    "runner_service_account": "ml-runner-retail@retail-ml-prod...",
    "bigquery_dataset": "retail",
    "landing_prefix": "incoming/retail/"
  }
}
```

The use case refers to these by name, never by value. Where the training pipeline needs the dataset, the configuration says `${resources.bigquery_dataset}`, and each environment fills in its own. The ML team never types a project ID. When the platform team moves the retailer to a new project, it changes the manifest and nothing in the use case moves.

The manifest also carries the platform's **policies**: limits the ML team cannot override from its side. Section 2.5 shows one at work.

> **On Google Cloud: Cloud Storage.** Google Cloud's object store: files of any size, kept in buckets, addressed by paths such as `gs://bucket/incoming/retail/`.
>
> *Why it matters.* ML work needs somewhere to put the things that are not tables: the files that arrive, the compiled pipelines, a run's working files. A bucket is cheap, durable and readable by any service you allow.
>
> *In Pipewright.* The platform team's artifacts bucket holds the environment manifest, each release's contract and every run's working files. A landing bucket takes the files that start runs, one folder per team.
>
> Read more: https://docs.cloud.google.com/storage/docs

On Google Cloud: BigQuery. Google Cloud's data warehouse: tables you query with SQL, billed by the data a query reads, with no servers to run.

Why it matters. Training data, predictions and the records of what happened are all tables. Keeping them in one warehouse means every question about a model is a query, not a hunt through files.

In Pipewright. Each team gets one dataset. Its raw data, snapshots, predictions, metrics and monitoring tables live there, and every pipeline step reads and writes it as the team's identity.

Read more: https://docs.cloud.google.com/bigquery/docs

On Google Cloud: IAM. Identity and Access Management: who (a person or a service account) may do what, on which resource. A service account is an identity for software, not for a person.

Why it matters. A pipeline that runs as an all-powerful account can read every team's data. Giving each team's runs an identity of their own is what keeps one team's mistake inside its own dataset.

In Pipewright. The manifest names a runner service account per team, and every step of that team's runs uses it. The launcher and CI have their own accounts, and nothing uses a downloaded key.

Read more: https://docs.cloud.google.com/iam/docs/overview

2.5 One use case, many environments

The retailer runs two environments: **test**, where models are trained and checked, and **production**, where they score customers. The use case is the same in both, with a few deliberate differences. Figure 2.2 shows how they combine.

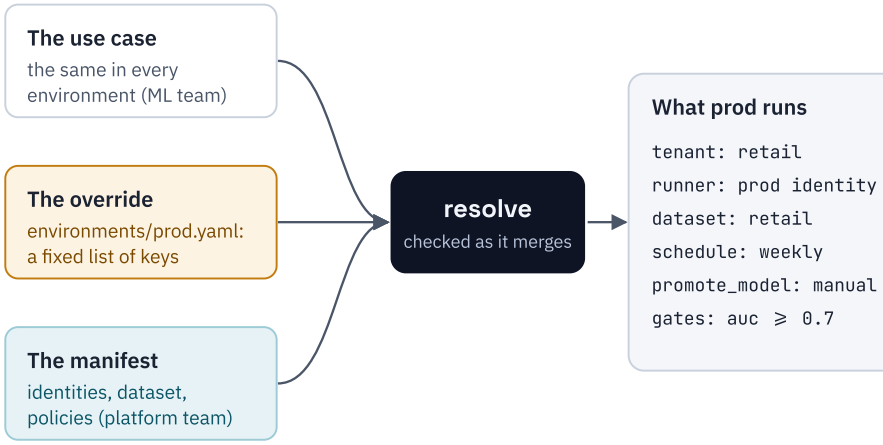


Figure 2.2 Resolving: the use case, the environment's override file and the platform's manifest combine into the configuration that one environment runs. Pipewright checks the result as it merges.

In test, the team wants fast iteration: a model that passes its gates becomes the champion at once. In production, a person decides, and the model retrains every Monday morning. Two short files say so:

```
# usecases/lapse-risk/environments/test.yaml
approvals:
  promote_model: auto
```

```
# usecases/lapse-risk/environments/prod.yaml
triggers:
  schedules:
    - { name: weekly, cron: "0 6 * * 1",
        timezone: Europe/London, pipeline: training }
```

Only a fixed list of keys may differ per environment: parameters, compute for each step, triggers, training's hyperparameters and

tuning budget, serving, approvals, whether retraining is on, alerts and retention. Try to change anything else, the gates for example, and Pipewright refuses:

```
spec.environments.prod: Additional properties are not allowed
('evaluation' was unexpected) (only parameters, steps, triggers,
training.hyperparameters, training.tuning, model.serving, approvals,
retraining.enabled, monitoring.alerts and
outputs.bigquery.retention_days
may vary per environment)
```

This is the most important rule in the chapter. **The gates, the data contract and the features are the same in every environment**, so a model that passed in test passed the same test it would face in production. If production could quietly lower its bar, a result in test would mean nothing.

You will be asked, sooner or later, for a production-only exception: a lower bar “just until the data improves”. The rule exists so that the answer is a visible change to the use case, reviewed like any other, and not a quiet edit to one environment.

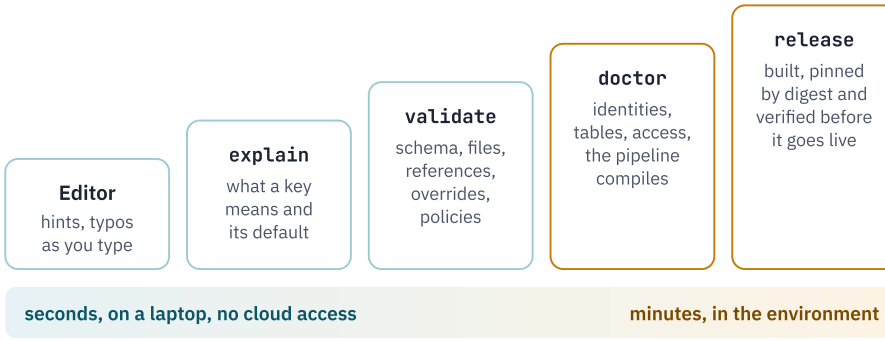
> **Pause and think.** Which of your settings differ between test and production today? Which of those differences did someone decide, and which just happened?

The platform team has the last word inside its environment. If the production manifest says that only a person may make a model the champion, an override that tries otherwise is refused, with the reason:

```
approvals.promote_model: auto is not allowed in prod (platform
policy)
```

2.6 Checked before anything runs

A mistake in a configuration file is cheap to fix on a laptop and expensive to find in a pipeline that ran for an hour. So the use case passes five checks, each catching what the one before cannot (Figure 2.3).



Each check catches what the one before cannot; most mistakes stop at the first three.

Figure 2.3 From the editor to the release. The first three run in seconds with no cloud access; doctor and release check the use case against the real environment.

- **The editor.** Every YAML file Pipewright writes starts with a line that points the editor at Pipewright's schemas, so a misspelt key is underlined as you type and hovering a key explains it.
- `pipewright explain <key>` says what any key means, its default, the values it allows and whether it may differ per environment.
- `pipewright validate` checks every use case against the schemas, the files it names, the references between sections,

the override rules and the platform's policies. It needs the manifest but no cloud access.
- `pipewright doctor` checks the use case against the live environment: that the identities exist, the tables can be created, the pipeline compiles and the permissions are in place. Each failure says who to ask.
- **The release** builds and pins everything, then verifies it before it goes live. Chapter 7 is about this step.

2.7 Every setting has a default

The retailer's `usecase.yaml` is short, and that can mislead. It names a tenant, a pipeline, a model, a trigger and one gate, but the use case it describes has far more settings than that: how the data is split, what the champion must beat, whether drift is watched, how many replicas serve, who must approve a deployment. Each one the team leaves out has a default, and the defaults are part of what runs.

Two commands make them visible. `pipewright resolve` prints the whole use case as one environment will run it, every key with its value, the defaults filled in:

```
pipewright resolve --env test --usecase lapse-risk
```

and `pipewright explain` says what one key means, what it defaults to and whether it may differ per environment:

```
$ pipewright explain retraining.cooldown_days
retraining.cooldown_days

Days after an approved retrain during which new alerts are recorded
but open no case.

type:      integer
limits:    at least 0
default:   7
per environment: no, the same in every environment
```

Leaving a key out is a decision too, and a reviewer should be able to see it. So the book's repository keeps one file with every setting of the lapse model written out, `ch05/reference/lapse-risk-all-settings.yaml`, generated from `pipewright resolve` and checked with `pipewright validate`. Each value is marked as the team's own, as a default, or as set for one environment only, and the file ends with the settings the use case does not use at all. Appendix B prints it as it stands at the end of Chapter 5. Read it once, from top to bottom, and you know everything a use case can say.

Since Pipewright 1.2.0. The examples you start a use case from are plain YAML with readable sample values, and the `churn` example writes every setting it leaves out as a `# default` comment beside the keys it sets, kept true by a test against Pipewright's own defaults. `minimal` stays minimal. Each example has a README saying what it shows.

2.8 Why this scales

The retailer has one model today and five by the end of the book. Configuration is what lets that happen without a platform team of five.

Every change is a pull request. When the data scientist wants production to retrain weekly, the diff is the five lines of `prod.yaml` above, and the person approving it does not need to read Python to know what will happen. When an analyst raises a gate, the review shows the old bar and the new one.

Every check runs in CI. Pipewright can write the workflows: validate, test and doctor on every pull request, then release after a merge. A configuration that breaks a rule never reaches an environment.

And the platform reads the same files people review. The launcher knows when to run a use case because the triggers say so. The table step knows what to create because the schemas say so. Nothing is configured twice, in a file and in a console, so the two can never disagree.

Thinking it through

On defaults. A default is a decision someone else made for you. Most are sensible, and you should still know them: read the resolved use case once before its first release, and again when you upgrade Pipewright, because a new version may add settings with defaults of their own.

On who has the last word. The platform's policies win over the use case. That only works if the people who set policies and the people who build models talk: a policy nobody explained becomes a wall to climb, not a rule to keep.

On YAML that grows logic. When a team starts wanting conditions in its configuration, it usually means a decision is hiding in the wrong place. Move the logic to code, test it, and keep the configuration to facts.

Try it: one use case, two environments

This continues the `retailer-mlops` folder from Chapter 1, and still needs no cloud account: the manifests are local files, as a platform team would publish them. The book's repository has one complete folder per chapter; `ch02/` holds the use case as this chapter leaves it, pointed at the book's own test environment.

1. Give the use case the retailer's data contract. Replace `usecases/lapse-risk/config/data.yaml` with the columns the lapse model will learn from, which Chapter 4 builds from the transactions:

```
data:
  contract:
    columns:
      - { name: customer_id,       type: string,  required: true }
      - { name: snapshot_date,     type: date,    required: true }
      - { name: recency_days,      type: integer, min: 0 }
      - { name: orders_12m,        type: integer, min: 0 }
      - { name: spend_12m_gbp,     type: float,   min: 0 }
      - { name: distinct_products, type: integer, min: 0 }
      - { name: country,           type: string }
      - { name: lapsed,            type: integer, allowed: [0, 1] }
  label: { column: lapsed }
  split: { strategy: time, time_column: snapshot_date }
features:
  exclude: [customer_id]
```

2. Describe two environments. Create `environments/test.yaml`:

```
apiVersion: pipewright/v1
kind: Environment
metadata: { name: test }
spec:
  manifest: test-manifest.json
```

and `environments/test-manifest.json`, the manifest a platform team would publish:

```
{
  "apiVersion": "pipewright/v1",
  "environment": "test",
  "blueprint": "private-managed@1.0.0",
  "orchestrator": "vertex",
  "project": "retail-ml-test",
  "region": "europe-west2",
  "artifact_registry": {
    "docker": "europe-west2-docker.pkg.dev/retail-ml-test/ml-
images",
    "kfp": "https://europe-west2-kfp.pkg.dev/retail-ml-test/ml-
pipelines"
  },
  "artifacts_bucket": "retail-ml-test-artifacts",
  "landing_bucket": "retail-ml-test-landing",
  "launcher": {
    "url": "https://launcher.retail.example",
    "service_account": "ml-launcher@retail-ml-
test.iam.gserviceaccount.com"
  },
  "tenants": {
    "retail": {
      "runner_service_account": "ml-runner-retail@retail-ml-
test.iam.gserviceaccount.com",
      "bigquery_dataset": "retail",
      "landing_prefix": "incoming/retail/"
    }
  }
}
```

Make the same pair for production, `prod.yaml` and `prod-manifest.json`, with `prod` in place of `test`.

3. Validate both.

```
pipewright validate --env test
pipewright validate --env prod
```

Each prints `OK lapse-risk` and `1 valid, 0 with problems`.

4. Add the overrides from Section 2.5: `usecases/lapse-risk/environments/test.yaml` with `promote_model: auto`, and `prod.yaml` with the weekly schedule. Then compare what each environment runs:

```
pipewright resolve --env test --usecase lapse-risk > test.txt
pipewright resolve --env prod --usecase lapse-risk > prod.txt
diff test.txt prod.txt          # on Windows: fc test.txt prod.txt
```

The only differences are the ones you meant: the runner's identity (from each manifest), the weekly schedule, and who makes a model the champion.

5. Break the rules on purpose. Add a stricter gate to `prod.yaml`:

```
evaluation:
  gates:
    - { metric: auc, min: 0.9 }
```

and run `pipewright validate --env prod` again: it refuses, and lists what may vary. Remove the lines. Then add `"policies": {"promote_model": "manual"}` to `prod-manifest.json`, put `approvals: { promote_model: auto }` into `prod.yaml`, and validate once more: the platform's policy wins. Undo both changes before Chapter 3.

In short

- A setting buried in code is a decision nobody can review, that drifts between environments and that the platform cannot read.
- A setting belongs in configuration if a reviewer would approve a change to it, if it can differ per environment, or if the platform must read it. How the model learns stays in code.
- A use case is one folder: `usecase.yaml` says what it is, `config/` holds the decisions one concern per file, the code stays the team's own.
- The platform team's environment manifest supplies projects, identities, datasets and policies. The use case refers to them by name and never types a project ID.
- Only a fixed list of keys may differ between environments. The gates, the data contract and the features never do, so a model that passed in test passed the test production would set.
- Five checks, from the editor to the release, catch mistakes before anything runs.
- Every key a use case leaves out has a default. `pipewright resolve` shows them all, `pipewright explain` explains any one, and Appendix B lists every setting of the lapse model.

Next: Chapter 3. Data contracts and quality gates. The data contract you just wrote, put to work: what happens when a Monday's data breaks it.

Chapter 3. Data contracts and quality gates

Stop a bad Monday before it trains a model

The lapse model now has a home and a schedule. Every Monday a file of customer snapshots lands in a bucket, and the pipeline trains on it. For six weeks nothing goes wrong. Then, one Monday, the export that builds the file is changed by someone in another team. The change is small: a filter that left cancelled invoices out of the totals is dropped. Nobody tells the ML team, because nobody thinks they need to.

The file still has the same columns, in the same order. It still has a row for every active customer. It loads without an error. But 48 customers now have negative spend, because their refunds were larger than their purchases, and 1,626 have more orders than they really placed, because each cancellation is counted as an order. A model trained on it learns that some customers spend less than nothing, and scores the retailer's best customers on numbers that are wrong.

This chapter is about the promise a file makes before it is allowed to train a model, and the gate that holds it to that promise.

3.1 Data breaks quietly

Code that is wrong usually fails loudly: a test goes red, a step crashes, someone is paged. Data that is wrong usually does not. The pipeline reads the file, trains, evaluates and reports a score, and every step succeeds. The retailer's bad Monday is typical in three ways.

The cause is upstream. The people who changed the export did nothing wrong by their own lights. They own the export; the model is someone else's concern. Most data problems start in a system the ML team does not own.

The shape survives. A check that only asks whether the file has the right columns passes. The problem is in the values, and only in some of them.

The damage arrives later. A model trained on bad data may still pass its gates, because it is evaluated on data from the same bad file. The harm shows weeks later, when its scores lead to the wrong offers.

The cure is to write down what good data looks like, in one place, and check every file against it before anything learns from it. That written promise is a **data contract**.

3.2 What a contract says

A data contract is a short list of rules a file must keep. Each rule is a fact the team already knows about its data but has never written down: a customer ID is never empty, a customer appears once per snapshot, spend cannot be negative, the label is 0 or 1.

In Pipewright the contract lives in `config/data.yaml`, the file Chapter 2 created. These are the kinds of rule it can state:

Rule	Applies to	What it catches
`min_rows`	the file	a file that is empty or cut short
`unique`	a set of columns	rows that appear twice
`allow_extra_columns: false`	the file	a column nobody agreed to, which may be one nobody should see
`required`	a column	a column that has gone missing
`type`	a column	text where a number should be, dates that are not dates
`min` , `max`	a column	values outside what is possible
`allowed`	a column	a value outside a fixed list, such as a label that is not 0 or 1
`max_null_fraction`	a column	more empty values than the team will accept

For the retailer, the contract says what each column of a customer snapshot must be:

```
data:
  contract:
    min_rows: 1000
    unique: [customer_id, snapshot_date]       # one row per
customer per snapshot
    allow_extra_columns: false
    columns:
      - { name: customer_id,       type: string,  required: true,
max_null_fraction: 0 }
      - { name: snapshot_date,     type: date,    required: true,
max_null_fraction: 0 }
      - { name: recency_days,      type: integer, min: 0, max: 365 }
      - { name: orders_12m,        type: integer, min: 1 }
      - { name: spend_12m_gbp,     type: float,   min: 0 }
      - { name: distinct_products, type: integer, min: 1 }
      - { name: country,           type: string,  max_null_fraction:
0.01 }
      - { name: lapsed,            type: integer, allowed: [0, 1],
max_null_fraction: 0 }
  label: { column: lapsed }
  split: { strategy: time, time_column: snapshot_date }
features:
  exclude: [customer_id, country]
```

Every number in it has a reason. A snapshot only includes customers who bought in the twelve months before it, so `recency_days` is at most 365 and `orders_12m` at least 1. Spend is the total of what a customer paid, so it is never below zero. The label must be 0 or 1 and never empty, because a model cannot learn from a row whose answer is unknown. `country` may be empty in one row in a hundred, because the retailer's records are not perfect and the model does not use it; a file with more empty countries than that says something else has gone wrong.

Two things make this a contract and not just a check.

Both sides can read it. The team that owns the export can open `data.yaml` and see exactly what the model relies on. When they plan a change, they know what it must keep. A rule in a notebook cell is a private hope; a rule in a reviewed file is an agreement.

It is the same everywhere. As Chapter 2 showed, the data contract is one of the sections that may not differ between environments. Production cannot quietly accept data that test would refuse. And a release writes the contract into its own record, so every run says which rules its data was held to.

3.3 The contract as a gate

A contract is only useful if something enforces it, at the moment it matters: after the data arrives and before anything learns from it. In the retailer's training pipeline, that is the second step (Figure 3.1).

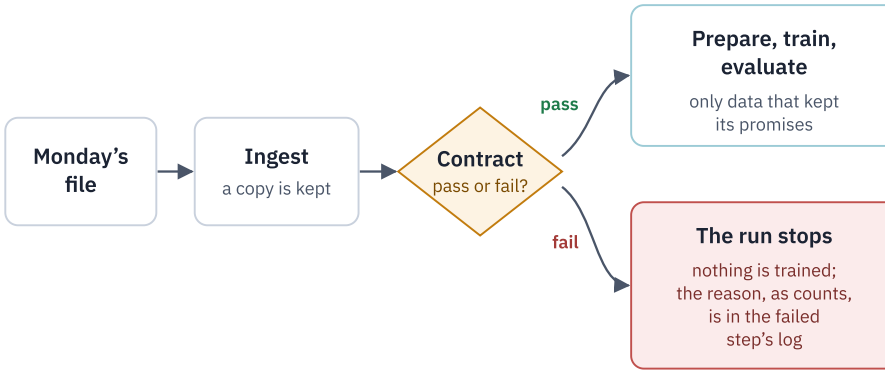


Figure 3.1 The contract as a gate. Every file is ingested, and a copy kept, before the contract is checked. Only data that keeps its promises reaches preparation and training; a file that breaks one stops the run, with the reason in the failed step's log.

Ingest reads the file and appends it, unchanged, to a raw table, with three columns of its own: the run that read it, the file it came from,

and when. It does this before the check, on purpose. Section 3.6 explains why.

Validate checks the data against the contract. It checks every rule, not just the first that fails, and reports all of them together. If any rule is broken, the step fails, and with it the run.

Everything after it (preparing features, training, evaluating, registering) runs only when the contract passes. A file that breaks a rule never reaches a model, so it can never become one.

The report a failed check writes is made of counts, never of values. It says "48 values below 0", not which customers or how much. This matters more than it seems. A pipeline's logs are read by many people and kept for a long time, and the data that breaks a contract is often personal. A check that printed the offending rows would copy customer data into every log it touched.

> **On Google Cloud: Agent Platform Pipelines.** Agent Platform (formerly Vertex AI) is Google Cloud's platform for machine learning. Its Pipelines service (formerly Vertex AI Pipelines) runs pipelines written with Kubeflow Pipelines v2: each step a container, started on a machine sized for it and stopped when it ends.
>
> *Why it matters.* A pipeline turns training from a notebook someone runs into steps that run the same way every time, with each step's inputs, outputs and logs recorded. You pay only while steps run.
>
> *In Pipewright.* Pipewright compiles each use case's pipelines at release time and starts every run on this service, as the team's own identity, with labels that tie the run to its use case and release.
>
> Read more: https://docs.cloud.google.com/gemini-enterprise-agent-platform/machine-learning/pipelines/introduction

On Google Cloud: Cloud Run. Runs a container as a web service that scales with its requests, down to none when nothing is calling it.

Why it matters. Something has to turn "a file arrived" or "it is Monday" into a pipeline run, check that the request is allowed, and never start the same run twice. That is a small, always-available service.

In Pipewright. The launcher, shared by every team in an environment, runs on Cloud Run. It receives files, schedules, API calls and run requests, and starts the right release's pipeline.

Read more: https://docs.cloud.google.com/run/docs

On Google Cloud: Pub/Sub. Google Cloud's messaging service: publishers send messages to a topic, and every subscription to that topic receives them.

Why it matters. Messages let services react to each other without calling each other directly. A file landing, a run finishing or a decision being made can reach whoever needs it, now or later.

In Pipewright. A landing bucket notifies a topic when a file arrives, and the launcher receives it. Pipewright also publishes lifecycle events (a run started, a model approved) on a topic, which Chapter 17 keeps, and private environments send run requests over one.

Read more: https://docs.cloud.google.com/pubsub/docs/overview

3.4 A Monday that passes

Before breaking anything, the retailer trains on a good file: 62,689 snapshots of 5,251 customers, one every month from June 2010 to September 2011, each with whether the customer bought again in the 90 days after it. For now a short script builds them from the

transactions; Chapter 4 explains how, and why each snapshot may only use what was known on its date.

The file lands in the tenant's landing folder, and the launcher starts a training run. The run's steps, as Agent Platform reports them, all succeed:

```
lapse-risk-r96bed78b63a1   SUCCEEDED   09:42 → 09:59
  ingest-data      SUCCEEDED
  validate-data    SUCCEEDED
  prepare-data     SUCCEEDED
  train-model      SUCCEEDED
  evaluate-model   SUCCEEDED
  register-model   SUCCEEDED
  record-metrics   SUCCEEDED
```

Each line is a step: a container, started from the release's image, that reads what the steps before it handed over and hands on what it made. Agent Platform Pipelines (formerly Vertex AI Pipelines) runs the steps in order, and in parallel where it can, and records every artifact passed between them. The retailer's pipeline is the short form. Figure 3.2 is the full one from Pipewright's reference example, as the console draws it, redrawn: it adds tuning and a comparison with the model that serves now. You meet each step in the chapter its label names.

The validate step's log says, simply, `data contract passed: 62689 rows`. Seventeen minutes after the file landed, the model is in the registry, with the run that made it and the score it reached:

```
$ pipewright model list --env test --usecase lapse-risk
lapse-risk-classifier (test)
  v1  DEFAULT  2026-10-27 09:58:45  run r96bed78b63a1  auc=0.7845
[gates pass]
```

An AUC of 0.78 clears the gate of 0.7 that Chapter 2 set. Chapter 10 is about what a gate should be; for now it is enough that the model had to earn its place.

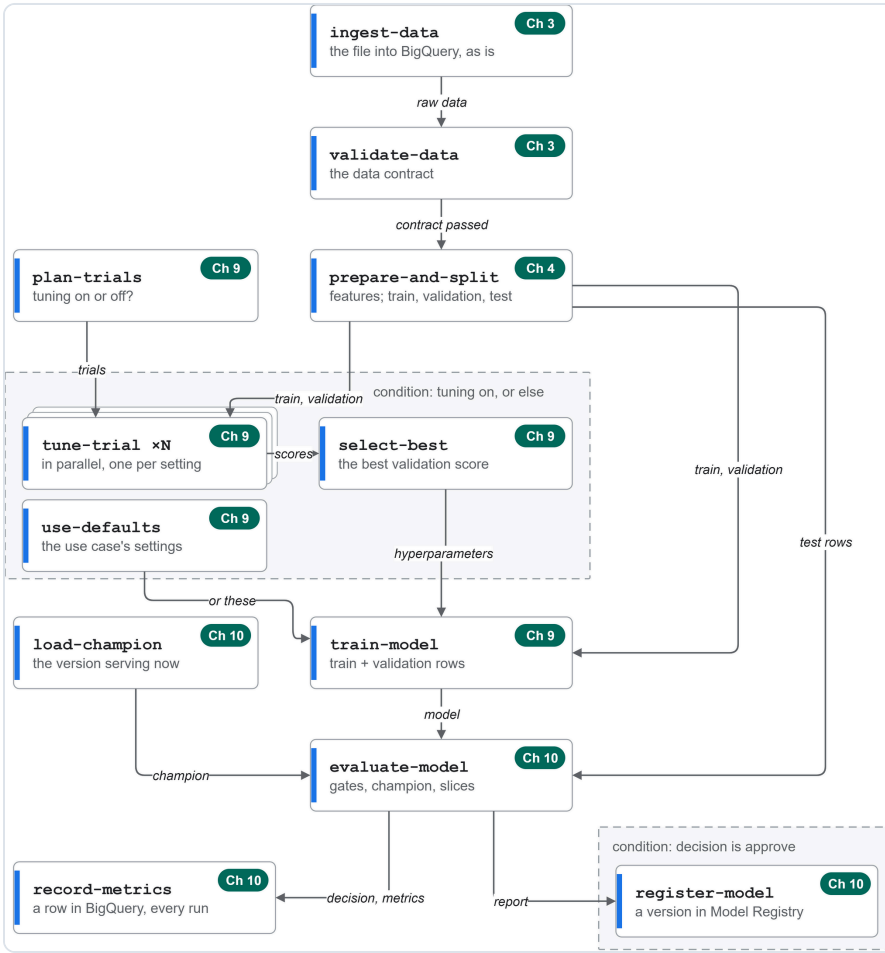


Figure 3.2 The reference training pipeline. Each box is one step; the words on an arrow are the artifacts it passes on (data, a model, metrics), which Agent Platform records, so a model can be traced to the rows that made it. A dashed box is a condition: tuning runs only when it is on, or else the use case's settings are used; only an approved model is registered. Its metrics are recorded on every run, approved or not.

3.5 A Monday that fails

Now the bad Monday. The export without its filter produces 4,366 rows, one for each customer active at the 1 September 2011 cut-off. Its columns are right, its row count is fine, every customer appears once and no label is missing. Only the values have changed.

The file lands, and a run starts. It fails 55 seconds later:

```
lapse-risk-r6396609ab319-snapshots-bad-monday   FAILED   10:00:51 →
10:01:46
  ingest-data      SUCCEEDED
  validate-data    FAILED
  prepare-data     NOT_TRIGGERED
  train-model      NOT_TRIGGERED
  evaluate-model   NOT_TRIGGERED
```

The failed step's log says why:

```
ValueError: data contract failed:
  spend_12m_gbp: 48 values below 0
```

Nothing was prepared, trained or registered. Version 1 is still the model in the registry, untouched. The run cost less than a minute of compute, and the reason fits on one line: which column, which rule, how many rows. The team that owns the export can be told exactly what broke without anyone sending them a single customer's record.

Notice what a good refusal looks like, because you will write the rules that produce it. It stops early, before anything costly. It names the column and the rule. It counts, and never quotes a value. If your data checks today print the offending rows into a log, they are leaking the data they protect.

> **Pause and think.** What happens today when a file your model depends on arrives broken? Does anything stop, or does the model learn from it? And who would be told, in what words?

3.6 The copy you keep

The bad file was ingested before it was checked, so its rows are in the raw table, next to the good file's, each marked with the run that read it:

```
SELECT run_id,
       COUNT(*)                    AS row_count,
       COUNTIF(spend_12m_gbp < 0)  AS negative_spend
FROM   pw_retail.lapse_risk_raw_customers
GROUP  BY run_id
```

```
run_id          row_count   negative_spend
r6396609ab319        4366               48
r96bed78b63a1       62689                0
```

Keeping a copy of a file the model refused may look odd. It is the point. When the export team asks what was wrong, the answer is a query away, on the exact rows that arrived, not on a file someone may since have fixed or deleted. When the same problem happens again in three months, the two copies can be compared. And when an auditor asks why the model was not retrained that week, the run, its log and its data are all still there.

The raw table is append-only and partitioned by the day it was written, so it is cheap to keep and simple to expire. How long it keeps data is a setting, `outputs.bigquery.retention_days`, and one of the few that may differ between environments, because test and

production often have different rules for how long data may be kept.

3.7 What a contract cannot catch

The contract stopped the bad Monday because of 48 customers with negative spend. It said nothing about the other problem in the same file, which was larger (Figure 3.3).

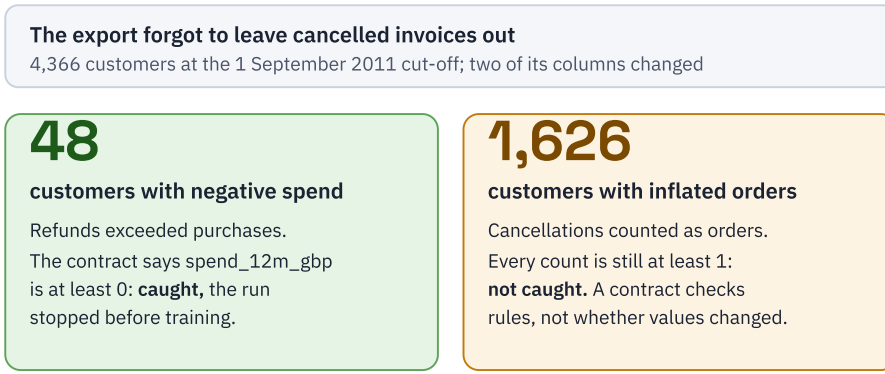


Figure 3.3 The bad Monday. Negative spend broke a rule and stopped the run. Inflated order counts broke none: every count was still at least 1, and a contract checks rules, not whether values have changed.

Counting cancellations as orders gave 1,626 customers more orders than they placed. But every one of those counts was still a whole number of at least 1, so every one kept the rule. Had no customer's refunds exceeded their purchases, the file would have passed, and the model would have learned that the retailer's customers order more often than they do.

A contract checks that each value is possible. It cannot check that a value is right, and it does not compare this week's file with last week's. That needs a second kind of check: one that knows what the data usually looks like and notices when it moves. Chapter 14 is

about that kind of check, drift monitoring, which compares each new batch with the data the model was trained on.

The two work together. The contract is cheap, strict and certain: a rule is kept or it is broken, and a broken rule stops the run. Monitoring is statistical and needs judgement: a shift may be a broken export or a real change in customers, such as the Christmas peak in Chapter 1. A good platform uses the contract to stop what is impossible and monitoring to question what is unusual.

Writing a good contract is a habit more than a skill. Start with what you know is impossible, as the retailer did. Add a rule each time something gets through that should not have; the inflated orders, for example, suggest a rule the retailer could add once it knows its customers: a `max` on `orders_12m`. Keep each rule something both teams agree is true, and the contract stays short enough to read.

> **Pause and think.** Write down three things you know are impossible in your most important input: a negative amount, a date in the future, a key that appears twice. Are any of them checked before a model learns from the data?

3.8 Who finds out

A run that stops for bad data has done its job, but someone still has to fix the data. In Pipewright 1.1.0 the run fails like any other failed pipeline run: Agent Platform shows it as failed, the failed step is marked, and its log holds the reason. Pipewright raises alerts for monitoring, retraining and evaluation, and Chapter 14 sets them up; it does not raise one of its own for a failed run.

So the platform team decides how failures reach people, once, for every use case. A common choice on Google Cloud is an alert in

Cloud Monitoring on the pipeline service's logs, filtered to failed runs, which sends a message to the team's channel. Because every use case runs through the same pipelines service, one alert covers them all.

Since Pipewright 1.2.0. A failed run is an event of its own. Each hour the launcher looks for runs that ended failed or cancelled and records one `run.failed` event per run, with the failed steps and their first error line, on Agent Platform and on Kubeflow alike. A use case that lists `run.failed` in `monitoring.alerts.events` gets an email for it, so the team that owns the data hears about a refused file without a log-based alert of its own:

```
monitoring:
  alerts:
    events: [run.failed]            # an email when a run fails or
is cancelled
```

Whatever carries the news, the message should lead to the run, and the run to the log line. The ML team should not need to read the data to know what broke, and the export team should not need to see a model to know what to fix.

> **On Google Cloud: Cloud Logging and Cloud Monitoring.** Cloud Logging collects the logs of every Google Cloud service and every pipeline step in one place. Cloud Monitoring watches metrics and logs and raises alerts, by email or to a channel, when a condition you set is met.
>
> *Why it matters.* A model that fails quietly does harm for longer. Logs answer "what happened", and alerts make sure someone hears about it without having to look.
>
> *In Pipewright.* Every step writes structured logs labelled with the use case and run. `pipewright alerts sync` turns a use case's alerts into Cloud Monitoring policies that email its team (Chapter 14).
>
> Read more: https://docs.cloud.google.com/logging/docs/alerting/log-based-alerts

Thinking it through

On who writes the contract. The rules describe another team's data. Written alone, they become the ML team's assumptions; written together, they become an agreement. The best contracts are reviewed by the people who produce the data.

On strictness. A contract that refuses too much gets switched off. Start with what is impossible, not with what is unusual, and leave the unusual to monitoring.

On the copy you keep. Keeping refused data costs a little storage and buys the ability to explain every refusal later. Decide how long you keep it as deliberately as you decide the rules.

Try it: a good file and a bad Monday

This continues from Chapter 2, and now uses the cloud. Build it as you read, starting from the `ch02/` folder of the book's repository, or

open `ch03/`, which holds this chapter's finished code, and run it as it is. You need a Google Cloud project with a Pipewright environment, built as Appendix A describes. The book's project is called `pw-ebook`; use your own project's names where the commands show it. Everything here costs pennies: two short pipeline runs and a few megabytes in BigQuery, and nothing is left running.

1. Build the snapshots. Download the dataset from https://archive.ics.uci.edu/static/public/502/online+retail+ii.zip, unzip it into the repository folder, and run the script that ships with the repository:

```
pip install pandas openpyxl
python tools/make_snapshots.py online_retail_II.xlsx
```

It writes two files into `usecases/lapse-risk/data/`: `snapshots.csv`, the good file, and `snapshots_bad_monday.csv`, the 1 September cut-off as the broken export would have made it. It prints what it wrote:

```
snapshots.csv: 62,689 rows, 5,251 customers, 16 cut-offs, lapsed
rate 57.8%
snapshots_bad_monday.csv: 4,366 rows, 48 with negative spend, 1626
with a different order count
```

2. Write the contract. Replace `usecases/lapse-risk/config/data.yaml` with the contract in Section 3.2, and add the schema of the raw table, `usecases/lapse-risk/schemas/raw_customers.json`, from the repository: the eight columns of the contract plus `run_id`, `source_uri` and `ingested_at`. Then check the use case against the environment:

```
pipewright validate --env test
pipewright doctor --env test
```

3. Create the tables and release. The first command creates the raw and metrics tables; the second builds the components, pins them and makes this release the live one:

```
pipewright tables sync --env test
pipewright release --env test
```

The release ends with what it verified, and the contract it wrote:

```
contract: gs://pw-ebook-pw-artifacts/contracts/lapse-
risk/06f161bf549f.json
  verified: contract v3 is valid
  verified: component image exists (by digest)
  verified: training template churn-training/sha256:b4c2e035103a...
matches the contract
live: gs://pw-ebook-pw-artifacts/contracts/lapse-risk/current.json -
> 06f161bf549f (first release)
```

4. Drop the good file. Copy it into the tenant's landing folder:

```
gcloud storage cp usecases/lapse-risk/data/snapshots.csv \
    gs://pw-ebook-pw-landing/incoming/retail/snapshots-2011-09-
01.csv
```

Open **Pipelines** in the Agent Platform console and watch the run. It takes about 17 minutes. When it succeeds, list the model:

```
pipewright model list --env test --usecase lapse-risk
```

On a brand new project, the very first run can fail at once with a message that the *Vertex AI Service Agent* cannot read Artifact Registry. Google creates that identity when the project first uses pipelines, and its permissions take a few minutes to arrive. Wait five minutes and start the run again by hand:

```
pipewright run --env test --usecase lapse-risk \
    --source-uri gs://pw-ebook-pw-landing/incoming/retail/snapshots-
2011-09-01.csv --run-key retry-1
```

5. Drop the bad Monday.

```
gcloud storage cp usecases/lapse-risk/data/snapshots_bad_monday.csv
\
    gs://pw-ebook-pw-landing/incoming/retail/snapshots-bad-
monday.csv
```

Within a minute or so the run fails at `validate-data`. Open the failed step, then its logs, and find the two lines from Section 3.5. Run `pipewright model list` again: still only version 1.

6. Read the copy. Run the query from Section 3.6 in the BigQuery console, with your tenant's dataset. Both files are there, and only one has negative spend.

7. Make the contract catch more. Look at the distribution of `orders_12m` in the good file, choose a `max` you believe, and add it to the contract. Run `pipewright validate --env test`. You do not need to release or run again: the point is that the next rule is a one-line change that both teams can review.

8. Leave it costing nothing. Nothing is deployed, so there is no endpoint to remove, and the launcher scales to zero by itself. Confirm that no run is still going in the **Pipelines** page. What

remains is a few megabytes in BigQuery and Cloud Storage, which cost pennies a month.

In short

- Data breaks quietly: the cause is upstream, the shape survives, and the damage shows weeks later.
- A data contract writes down what good data looks like, in rules both teams can read: rows, uniqueness, columns, types, ranges, allowed values and how many may be empty.
- The contract is the same in every environment, and each release records the one its runs are held to.
- In the pipeline the contract is a gate. A file that breaks a rule stops the run before anything learns from it, and the reason is given as counts, never values.
- Every file is kept, passed or refused, in an append-only raw table, so a refusal can be explained from the rows that arrived.
- A contract checks that values are possible, not that they are right. Monitoring, in Chapter 14, catches what moves.
- Tell people when a run fails: since Pipewright 1.2.0, `run.failed` in `monitoring.alerts.events`; before it, one platform alert on failed pipeline runs for every use case.

Next: Chapter 4. Features without leakage. Where the snapshots come from, and why a feature may only use what was known on its date.

Chapter 4. Features without leakage

Train only on what was known at the time

The retailer's analyst has an idea for a better lapse model. Why rebuild a customer's history at every cut-off, as the snapshot file does, when the data warehouse already has a view called `customer_latest`, with every customer's recency, orders and spend, refreshed each night? Join it to the list of customers and their labels, and the training data is ready in one query.

The model trained that way scores an AUC of 0.852, far above the 0.776 of the model trained on snapshots. The analyst is pleased. The gate of 0.7 is cleared with room to spare. And when the model goes live, it is worse than the one it replaced: on the data it really sees, its AUC is 0.722.

Nothing in the code was wrong. The model learned from values that did not exist yet at the moment its predictions were meant for. This is **leakage**, the most common way a model looks better in evaluation than it will ever be in use, and this chapter is about making it impossible.

4.1 The answer hidden in the question

The lapse model answers one question: *on this date, given what we know about this customer, will they buy again in the next 90 days?* The words that matter are "on this date". Every training row stands for a moment, its **cut-off**, and must look exactly as the world looked at that moment.

The `customer_latest` view breaks that. Take a customer whose row has a cut-off of 1 September 2011 and who bought again three weeks

later. The view, computed in December, already contains that purchase. The model sees a recent purchase, sees the label "did not lapse", and learns a rule that is true in hindsight and useless in advance: customers who bought recently, *after the cut-off*, did not lapse. It has learned to read the answer from the question (Figure 4.1).

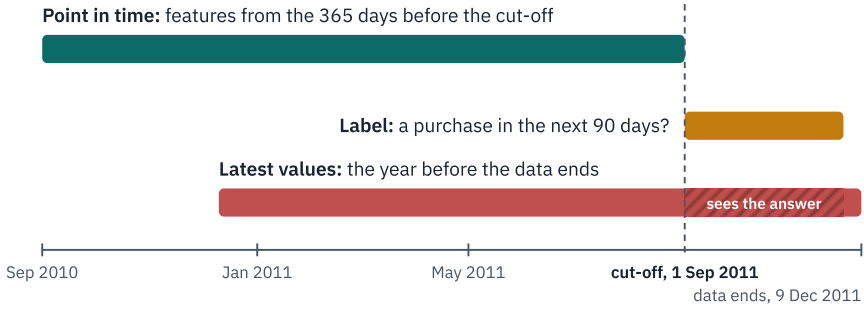


Figure 4.1 The windows around one cut-off. Point-in-time features come from the year before it, the label from the 90 days after. Features from a latest-values view come from the year before the data ends, which contains the label window: the model sees the answer.

Measured honestly, the damage is plain. The same model, trained the same way, on the same customers and labels, with only the features changed:

Features	AUC on the later cut-offs
Point in time, as the snapshots compute them	0.776
Latest values, evaluated as they were trained	0.852
Latest values, scored on what production knows at each cut-off	0.722

The second line is the number the team would have seen and believed. The third is what the business would have got. The leaky model is not just overconfident; it is worse than the honest one,

because it learned to lean on a signal that is never there when it matters.

Leakage has many disguises, and every one of them is a value from after the cut-off:

- **A "latest" view or table**, as here.
- **A table updated in place.** A customer's segment is overwritten each night, so last year's rows see this year's segment.
- **A timestamp of what a value describes, not when it was known.** February's total, computed on 3 March, is stamped 28 February, so a cut-off of 1 March sees it two days early.
- **Something caused by the outcome.** A "win-back email sent" flag is set only for customers who had already lapsed.

No gate catches any of them, because a leaky model really does score well on leaky data. Leakage has to be prevented where the training data is put together.

So be suspicious of a model that scores better than you expected. When a result surprises you upwards, ask before you celebrate: could any feature have known the answer? A gain that cannot be explained is more often leakage than insight.

> **Pause and think.** Take the features of a model you know. For each one, when did its value become known, and could it have changed after the moment the model predicts for? Which of your tables are updated in place?

4.2 Point in time

The rule that prevents it fits in one sentence: **each training row gets, for each feature, the newest value that was known at or before**

its cut-off, and never a value from after it. A join that follows the rule is called a **point-in-time join**.

It needs two kinds of table, and the difference between them is the heart of this chapter.

An **entity table** says who and when: one row per customer per cut-off, with the label. It says nothing about the customer's behaviour. For the retailer, `lapse_risk_entities` has 62,689 rows: every customer who bought in the year before each of 16 monthly cut-offs, with whether they bought again in the 90 days after it.

A **feature table** says what was true, and since when. Each row has the entity, a timestamp saying when the values became known, and one column per feature. A value is never changed; when it changes, a new row is added. So the table holds the whole history, and any moment in the past can be looked up.

Figure 4.2 shows the join for one real customer. The feature table has a row for each month. For the cut-off of 1 September, the join takes the row of 1 September, the newest at or before it. The row of 1 October is never read, which matters: its recency of 10 days is the trace of a purchase in September, which is the label.

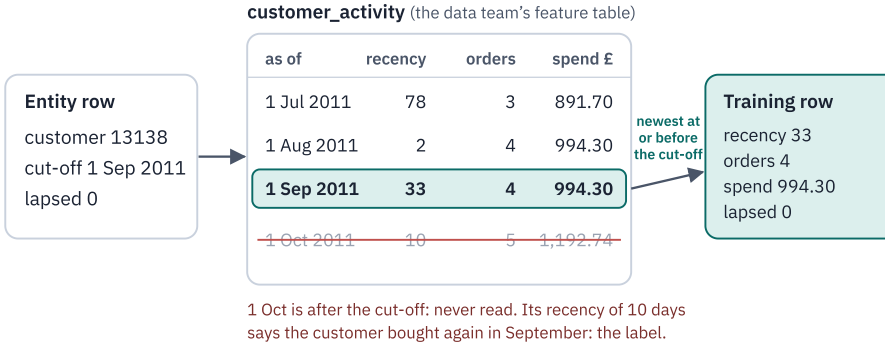


Figure 4.2 A point-in-time join for customer 13138 at the 1 September 2011 cut-off. The join takes the newest feature row at or before the cut-off. The 1 October row would have told the model the answer.

4.3 A feature table, built once

The retailer's features belong to the data team, not to the lapse model. Recency, orders and spend are as useful to the customer segments of Chapter 8 and the checkout offer of Chapter 13 as they are here. If every model computed them in its own query, there would soon be three definitions of "spend", each slightly different, and nobody would know which model used which.

So the data team publishes one feature table, `customer_activity`, and every model reads it. The rules for a feature table are few, and each one protects the point-in-time join:

Rule	Why
An **entity column**, here `customer_id`	how rows are matched to the entity table
A **timestamp column**, `feature_timestamp`, saying when the values became known	the cut-off is compared with it
One column per feature, with the type the data contract expects	a feature is a column
A new row each time values change; never an update in place	an update destroys the history old rows need
One row per entity per timestamp	two rows with the same timestamp make the choice arbitrary

The data team computes the table once a month, in the first minutes of the 1st, from the 365 days before it. Each row's `feature_timestamp` is midnight at the start of that month, the moment its values are true as of. The SQL lives in the data team's folder, `data-team/sql/`, where it is reviewed like any code:

```
SELECT
  customer_id,
  @as_of                                          AS
feature_timestamp,
  TIMESTAMP_DIFF(@as_of, MAX(invoice_date), DAY) AS recency_days,
  COUNT(DISTINCT invoice)                         AS orders_12m,
  ROUND(SUM(quantity * price), 2)                 AS spend_12m_gbp,
  COUNT(DISTINCT stock_code)                      AS
distinct_products
FROM `pw_retail.retail_transactions`
WHERE NOT STARTS_WITH(invoice, 'C') AND quantity > 0    -- purchases
only
  AND invoice_date <  @as_of                            -- nothing
from the month it is for
  AND invoice_date ⩾ TIMESTAMP_SUB(@as_of, INTERVAL 365 DAY)
GROUP BY customer_id
```

Two details make it safe to run for years. The monthly job only ever appends, with an `INSERT`. And it begins with an assertion that the month is not already there, so a job that runs twice by mistake stops instead of adding a second row for every customer:

```
customer_activity_month.sql: Query error: customer_activity already
has this month at [5:1]
```

A table that only starts today has no history, and a model needs history to train on. So the first time, the same query runs for every past month, each computed only from the data that existed then: a **backfill**. Stamping backfilled rows with today's date would be the third disguise of leakage in Section 4.1, so each row gets its own month. After the backfill, `customer_activity` has 84,321 rows: 24 months, from January 2010 to December 2011.

Any tool that keeps these rules can build a feature table. Many teams use Dataform, which runs inside BigQuery with nothing to host, or dbt, whose snapshots record a new row each time a value changes. What matters is not the tool but that the table is tested, reviewed and owned.

4.4 The use case reads it

The ML team's side is a few lines in `config/data.yaml`, under the contract it already has:

```
  features_from:                        # the run's source is the
entity table
    entity: { columns: [customer_id], time_column: snapshot_date }
    sources:
      - table: bq://pw_retail.customer_activity
        features: [recency_days, orders_12m, spend_12m_gbp,
distinct_products]
```

`entity` says which columns identify a row and which holds its cut-off. Each source names a feature table and the features to take from it. A use case can read from several, each with its own owner.

Three things change with it.

The run's source is the entity table. Training no longer starts from a file of finished rows; it starts from `bq://pw-ebook.pw_retail.lapse_risk_entities`, and Pipewright adds the features. A file dropped into the landing folder cannot be an entity table, so `validate` refuses to keep the landing trigger:

```
triggers.landing delivers gs:// files, but with data.features_from a
run's source is the entity table (bq://);
use a schedule or `pipewright run --source-uri bq://...`
```

The contract still applies, to the joined rows. Every column the model sees is in the contract, whichever table it comes from. The bad Monday of Chapter 3 would be stopped just the same if it arrived through the feature table.

Everything is checked before it runs. `validate` refuses a feature that is also the label, the entity or the cut-off column, and a feature that appears twice. `doctor` checks the feature table exists and has what the use case expects:

```
Feature tables of lapse-risk
  OK    pw-ebook.pw_retail.customer_activity: table, 4 features as
of feature_timestamp
```

4.5 The run

The team releases the change and starts a run from the entity table:

```
pipewright run --env test --usecase lapse-risk --source-uri bq://pw-
ebook.pw_retail.lapse_risk_entities
```

The run's first step now does what the snapshot script used to do. It reads the entity table, runs the point-in-time join against `customer_activity` as the tenant's runner, and checks the joined rows against the contract. Nine minutes later the run has succeeded, and the registry has a second version:

```
$ pipewright model list --env test --usecase lapse-risk
lapse-risk-classifier (test)
  v2  DEFAULT  2026-10-27 10:32:00  run r3afbf247999e, auc=0.7855
[gates pass]
  v1           2026-10-27 09:58:45  run r96bed78b63a1, auc=0.7845
[gates pass]
```

The two scores are almost the same, and that is the result to want. The rows the run read are in the raw table, as in Chapter 3, and compared with the snapshot file that trained version 1 they match exactly: 62,689 rows, every value of every column. The features are the same; what changed is where they come from and who owns them. The snapshot file was a one-off made by a script on a laptop. The feature table is a governed table the data team keeps up to date, which any model can read, at any cut-off, with the same guarantee.

The run also records what it read. Its ingest step writes a `data.features.resolved` event to the run's log, with the entity table, each feature table and its features, the cut-off column, how many rows found a match, and the BigQuery job that ran the join. Laid out for reading, it says:

```
pipewright event data.features.resolved
  release        94882967a9f9
  run            r3afbf247999e
  entity_table   pw-ebook.pw_retail.lapse_risk_entities
  cutoff         snapshot_date
  table          pw-ebook.pw_retail.customer_activity
  time_column    feature_timestamp
  features       recency_days, orders_12m,
                 spend_12m_gbp, distinct_products
  rows           62689 (62689 matched)
  query_job      0cc1c2f6-c4a7-41a2-a715-02512c041c1f
```

When someone asks, a year from now, which features version 2 was trained on and as of when, this is the answer. Chapter 17 follows records like it into the model's full lineage.

4.6 Features defined once, owned by someone

The point-in-time join solves leakage. The feature table solves something just as costly: the same feature written many times.

Figure 4.3 shows the split of ownership the retailer ends up with.

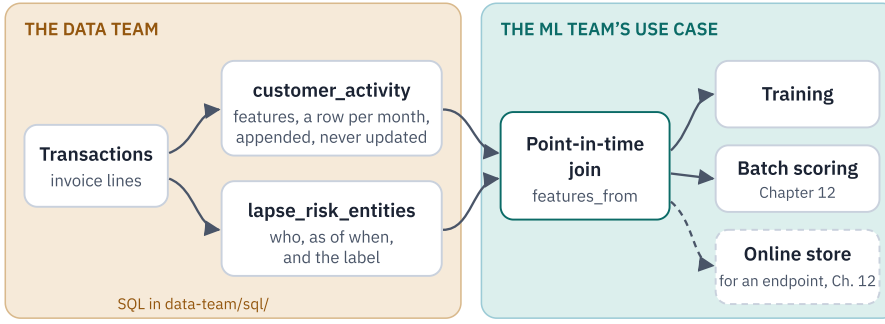


Figure 4.3 Who owns what. The data team turns the transactions into a feature table and an entity table. The use case names them, and Pipewright joins them point in time for training and, later, batch scoring. An online store can serve the same features to an endpoint.

The data team owns the features: how they are computed, their history, their tests, and the erasure of a customer's rows when the customer asks. The ML team owns the use case: which features it reads, its contract, its gates. Neither has to read the other's code to do its job, and a change on one side is a reviewed change to a file the other can see.

The platform team has a say too. The environment manifest can list the datasets each tenant may read features from, so a use case cannot quietly join a table nobody approved, and it can require that every feature in an environment comes from a feature table. In this book's environment the feature tables sit in the tenant's own dataset, to keep the setup short; in a larger organisation they live in the data team's dataset, and the manifest lists it.

Pause and think. How many times is "a customer's spend over the last year" computed in your organisation, and do the copies agree? If a customer asked to be forgotten, how many places would have to remove them?

Google Cloud also offers a registry for features, the **Feature Store** in Agent Platform. A team registers a feature group once, pointing at a BigQuery table, and gives each feature a name, a description and an owner. A use case can then name a feature group instead of a table, and Pipewright looks the table and columns up in the registry at every run, so a group its owners move is followed without a change to the use case. The join is the same point-in-time query either way. Use feature groups when your organisation registers its features centrally, and feature tables otherwise.

On Google Cloud: Agent Platform Feature Store. A registry of features on top of BigQuery: feature groups say which table holds an entity's features and which columns they are. It can also sync the latest values into an online store for fast lookups.

Why it matters. Features are reused across models, so their definitions should be in one place, with an owner. A central registry lets a model name a feature instead of copying the query that makes it.

In Pipewright. A use case reads features from a feature table or a feature group; Pipewright resolves either to the same point-in-time join. Chapter 13 serves the same features from the online store.

Read more: https://docs.cloud.google.com/gemini-enterprise-agent-platform/machine-learning/featurestore/latest/overview

4.7 The same features online

Training and batch scoring read features from BigQuery, which is right for thousands of rows at a time and too slow for one customer at the checkout. A model behind an endpoint needs a customer's features in milliseconds, and the classic mistake is to compute them a second time, in the application, with code that drifts away from the SQL the model was trained on. The model then sees, in production, numbers that are slightly different from the ones it learned from. This is **training-serving skew**, and it is leakage's quieter cousin: the evaluation was honest, but it was an evaluation of a different system.

The cure is the same as for every other duplication in this chapter: one definition, read everywhere. An **online store** keeps the latest value of each feature, synced from the same feature table, and the endpoint reads it by customer ID. Chapter 13 builds this for the retailer's checkout offer, where the latest value is exactly what is wanted, because the prediction is being made now.

Thinking it through

On cost. A feature table with full history costs more storage than a latest-values view. It is almost always worth it: the cheaper table is the one that teaches your models to cheat.

On ownership. Point-in-time features need a data team that keeps history and never changes a value in place. If your data team does not work that way yet, the conversation about why is worth more than any model this chapter builds.

On honest numbers. The leaky model's 0.852 was a real number from a real evaluation. It was simply the answer to a different question. When you report a model's score, say what it was measured on, as of when.

Try it: from transactions to features

This continues from Chapter 3, on the same Google Cloud environment. Start from the `ch03/` folder, or open `ch04/`, which holds this chapter's finished code; run every command from inside the folder. It costs pennies: a few queries over 40 MB, and one training run of about ten minutes. The tools read your Application Default Credentials, so sign in once with `gcloud auth application-default login` if you have not.

1. Prepare the transactions. With the dataset unzipped as in Chapter 3:

```
pip install pandas openpyxl scikit-learn
python tools/make_transactions.py online_retail_II.xlsx
```

```
transactions.csv: 824,364 lines, 5,942 customers, 2009-12-01
07:45:00 to 2011-12-09 12:50:00
```

2. See leakage for yourself. Before any cloud work, train the model three ways on your laptop:

```
python tools/leakage_demo.py
```

```
point in time, evaluated as trained:            AUC 0.776
latest values, evaluated as trained:            AUC 0.852
latest values, scored on what production knows: AUC 0.722
```

3. Be the data team. Load the transactions into BigQuery and build the two tables. Use your own project ID:

```
python tools/data_team.py --project pw-ebook load
python tools/data_team.py --project pw-ebook sql
customer_activity_backfill.sql lapse_risk_entities.sql
```

```
pw-ebook.pw_retail.retail_transactions: 824,364 rows
customer_activity_backfill.sql: created customer_activity,
38,023,831 bytes processed
lapse_risk_entities.sql: created lapse_risk_entities, 38,160,459
bytes processed
```

The backfill stops at November 2011. Run the monthly job for December, then run it again to see it refuse:

```
python tools/data_team.py --project pw-ebook sql
customer_activity_month.sql --as-of "2011-12-01 00:00:00"
```

4. Point the use case at the features. Add the `features_from` block from Section 4.4 to `config/data.yaml`, and remove the `landing` line from the triggers in `usecase.yaml`. Then:

```
pipewright validate --env test
pipewright doctor --env test
```

Put the `landing` line back for a moment and validate again, to see the refusal from Section 4.4. Then remove it.

5. Release and run.

```
pipewright release --env test
pipewright run --env test --usecase lapse-risk \
    --source-uri bq://pw-ebook.pw_retail.lapse_risk_entities
```

Watch the run in the **Pipelines** page. When it succeeds, `pipewright model list --env test --usecase lapse-risk` shows version 2 beside version 1 from Chapter 3.

6. Check the join. The raw table keeps the rows this run read, features included. In the BigQuery console, count them by run: the new run has 62,689 rows, as many as the snapshot file. In **Logs Explorer**, search for `data.features.resolved` to see the record from Section 4.5.

7. Leave it costing nothing. As in Chapter 3: no endpoint is deployed, the launcher scales to zero, and the tables hold less than 100 MB, which costs pennies a month.

In short

- Leakage is training on a value that was not known at the cut-off. A leaky model scores well in evaluation and badly in use, and no gate can tell.
- A point-in-time join gives each training row the newest value of each feature known at or before its cut-off, and never one from after it.
- It needs an entity table (who, as of when, the label) and feature tables (what was true, and since when).
- A feature table has an entity column, a timestamp of when values became known, a column per feature, and a new row for every change. A backfill computes each past period from the data that existed then.
- The data team owns the features; the use case names them in `data.features_from`; the platform decides which datasets may be read. A feature is defined once and read everywhere.
- Online, the same features are served from an online store, so production sees the numbers the model learned from.

Next: Chapter 5. Reproducible runs. A label that needs 90 days to be known, a run that can be replayed exactly, and what to do when the data arrives late.

Chapter 5. Reproducible runs

Labels that are allowed to mature, and data you can read again

A year after the lapse model goes live, an auditor asks the retailer a simple question about one of its versions: *show us what it was trained on.* The team looks. The feature table has gained twelve months of rows. The purchases table has been reloaded twice, once to fix a currency bug. The entity table is rebuilt every month. Each table, read today, gives a different answer from the one the model saw, and BigQuery's time travel only reaches back seven days.

The honest answer is "we can tell you roughly". It is not good enough for an auditor, and it is not good enough for the team either: when a new version scores worse than the old one, they cannot tell whether the code got worse or the data changed, because they cannot train the old code on the old data again.

This chapter fixes two things that make runs hard to reproduce. The first is the label, which for the lapse model takes 90 days to become true, and is easy to get wrong while it is still on its way. The second is the data itself, which keeps changing under every run that reads it.

5.1 A label that takes 90 days

In Chapter 4 the label came ready-made in the entity table: a query decided, for each customer and cut-off, whether they bought again within 90 days. That works for cut-offs far enough in the past. It goes wrong for the recent ones.

Take the customers active on 1 December 2011. Their 90 days end at the start of March 2012, and the data ends on 9 December. A query

that asks “did they buy within 90 days?” finds, for most of them, no purchase yet, and says they lapsed. Run on the data as it stands, it labels 86.6% of the December cohort as lapsed, against a usual rate of 50 to 63%. The customers have not lapsed. Their window simply has not closed.

A row whose label is not yet known is not a 0. Training on it teaches the model that recent customers leave, which is exactly backwards for a retailer whose busiest weeks are the ones just before Christmas. The rule that prevents it: **a row may only be labelled once its whole window has passed.** Until then it is still maturing, and it is left out, 1s and 0s alike.

If your label is anything that takes time to happen, a churn, a default, a return, a claim, you have this problem whether you see it or not. The tell-tale sign is a recent cohort that looks much worse than the older ones.

5.2 Labels from events

The cleanest way to keep that rule is to stop storing labels at all. A label is a fact about the future of a row, and the facts it is made of are events: here, purchases. So the data team publishes the events, and the label is worked out at each run, from the events known at that moment.

The data team’s `customer_purchases` table has one row per purchase: the customer, the invoice and when it happened. The entity table loses its label column, and gains a row for every month up to the latest: 75,706 rows over 19 cut-offs, from June 2010 to December 2011. The use case says where the label comes from:

```
  label: { column: returned }            # 1: a purchase within 90
days of the cut-off
  label_source:
    table: bq://pw_retail.customer_purchases
    time_column: purchased_at
    window_days: 90
```

At each run, a row is labelled 1 when the customer has a purchase within 90 days after its cut-off, and 0 when they do not. And every row whose 90 days end after the newest purchase in the table is dropped.

A label source counts what happened, so the label is now `returned`, the opposite of `lapsed`: 1 when the customer came back. The model is the same model with its scores turned round. A lapse risk is one minus its score, and its AUC is the same either way.

5.3 The anchor, and data that arrives late

Notice what the window is measured against: not today, but the **newest event in the table**, called the anchor. The difference matters whenever data arrives late, which it always does.

Suppose the nightly load of purchases has fallen behind, and the table holds purchases only up to 8 November 2011, while the calendar says 9 December. A label anchored on today would treat the 1 September cohort as complete, since its 90 days ended on 30 November. But the purchases from 9 to 30 November have not arrived. Customers who came back in those weeks would be labelled as lapsed: on the retailer's data, 334 of the 4,335 customers in that cohort, which moves its lapse rate from a true 50.5% to 58.2%.

Anchored on the newest purchase, the same run sees that the 1 September window ends after 8 November, and drops those rows

instead. Late data can make a run smaller; it can never make a label wrong. Figure 5.1 shows which cut-offs each run keeps.

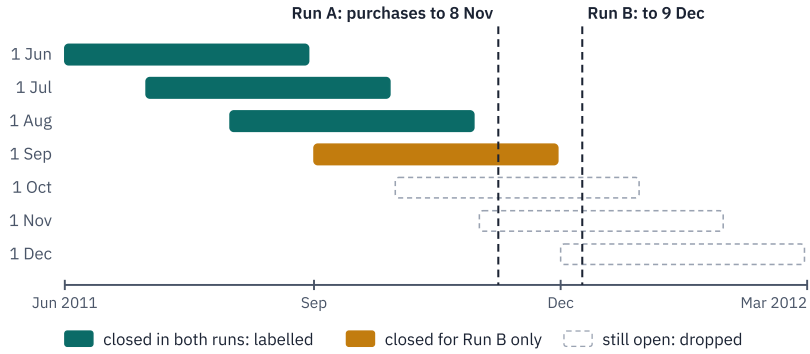


Figure 5.1 Each monthly cut-off has a 90-day label window. A run labels only the rows whose window has closed by the newest purchase it can see. Run A, with purchases loaded to 8 November, stops at August; Run B, a month later, also labels September. Later cut-offs are still maturing in both.

The retailer runs it that way. With the purchases loaded up to 8 November, it starts Run A from the entity table, exactly as in Chapter 4. The ingest step works the labels out first, and its log says what it found:

```
labels from pw-ebook.pw_retail.customer_purchases:
  58354 of 75706 rows have a closed 90-day window
  (anchor 2011-11-08T17:42:00+00:00), 24333 positive
```

Of the 75,706 rows, 17,352 were still maturing and were left out: every cut-off from September on. The 58,354 that remain are the 15 cut-offs from June 2010 to August 2011, and the feature join then runs on those alone. The same numbers go into a `data.labels.resolved` event with the run, so the anchor each model was trained against is on record.

> **Pause and think.** How long does it take for the outcome your model predicts to become known? What does your training data do with the rows whose outcome is still on its way: leave them out, or count them as a "no"?

5.4 Every run keeps what it read

The label is now right at every run. But the runs still read moving tables: a run in December reads a different purchases table from a run in November, and next year's backfill may change last year's features. To reproduce a run, the team needs the rows it read, not the tables it read them from.

So the ingest step keeps them. With one more line in `config/data.yaml`, every training run writes the exact rows it read, after the label and the feature join, into a snapshot table, each row tagged with the run:

```
  snapshot_table: lapse_risk_snapshots   # exactly what each run
read
```

The snapshot table is declared with the use case's other tables, with the contract's columns and three of its own, `run_id`, `source_uri` and `ingested_at`, and `pipewright tables sync` creates it. A run's `data.snapshot.written` event records how many rows it wrote and where. Nothing else changes: the run validates, trains and evaluates the rows it read, as before.

Run A finishes in six minutes and registers version 3, with an AUC of 0.7956. A month later, in the story, the missing purchases arrive. The data team reloads the table, now up to 9 December, and the team starts Run B from the same entity table. The anchor moves to

the newest purchase, the September window closes, and Run B labels 62,689 rows: Run A's 58,354 and the 4,335 of the September cohort. It registers version 4, with an AUC of 0.7836.

The snapshot table now holds both runs, side by side:

```
run_id          source                      rows   returned
rf76b5c548080   lapse_risk_entities        58354      0.417
rbcb69cdc615a   lapse_risk_entities        62689      0.422
```

(The source column is shortened here; the table holds the full `bq://` path.) Same source, different rows, because the source changed between the runs. Without the snapshot, Run A's rows would already be gone: the purchases table that produced them no longer exists in that form.

From a model back to its data

The snapshot also answers the question an audit starts with: which data made this model? You can follow it in four steps, and each step is a record Pipewright already keeps:

Step	Where it is kept	What it tells you
The model version	Model Registry	the run that trained it (its `run_id` label)
The run	the snapshot table, rows tagged with that `run_id`	the exact rows it trained and tested on, label included
The rows	the same rows' `source_uri`, and the run's data events	the entity table, the feature tables read at each row's cut-off, the label source with its window and anchor
The sources	your data team's tables	where those came from, under their own lineage

So you never need to ask “what did the table look like on 8 November?”. The rows are kept, under the run’s name, as they were. Chapter 17’s `pipewright lineage` walks this chain for you, from a version to its run, release, rows and sources, in one command.

The model list also shows something worth a pause. When Run A was evaluated, Pipewright compared it with the champion of the time, version 2 from Chapter 4, on Run A’s test rows. Version 2 scored 0.2010. It is not a bad model: it was trained to predict `lapsed`, and the test rows now say `returned`, so its scores point the other way, and 0.2010 is one minus 0.7990. A champion comparison is only meaningful when the label means the same thing for both models. When a label changes its meaning, as it did here, treat the next version as a new start and say so in the change that makes it.

5.5 Replaying a run

With the rows kept, reproducing a run is one command:

```
pipewright run --env test --usecase lapse-risk --replay <run_id>
```

A replay is a training run whose source is the earlier run’s snapshot. It reads no entity table, no feature table and no purchases. It goes through the same steps, the contract included, and registers a model like any other run (Figure 5.2).

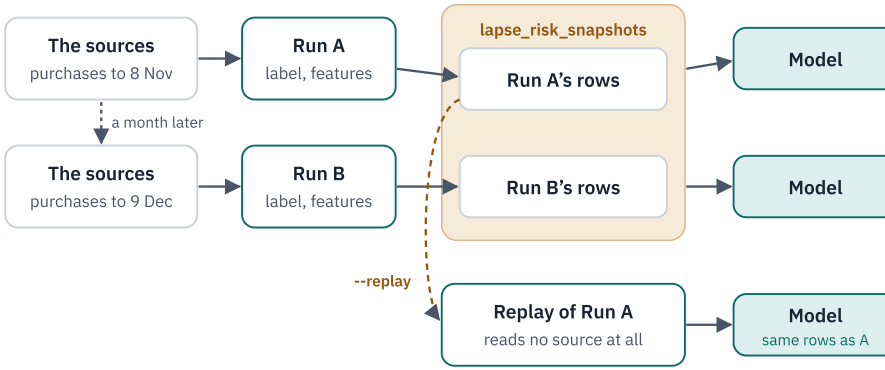


Figure 5.2 Run A reads the sources as they are on 8 November and keeps its rows in the snapshot table. A month later the sources have changed and Run B reads more. A replay of Run A reads only Run A's rows, so it trains on exactly the same data.

The retailer replays Run A, a month after it ran, with the purchases table already changed:

```
pipewright run --env test --usecase lapse-risk \
    --replay rf76b5c548080
```

The replay's ingest step writes a `data.snapshot.replayed` event naming the run it read, and its own snapshot, so the table now shows three runs:

```
run_id          source                        rows   returned
rf76b5c548080   lapse_risk_entities          58354      0.417
rbcb69cdc615a   lapse_risk_entities          62689      0.422
r1c29b0c83e14   snapshot://rf76b5c548080     58354      0.417
```

The replay read exactly Run A's rows, although the sources no longer hold them. It split them the same way, 45,353 for training and 13,001 for testing, and its model scored an AUC of 0.7951 against Run A's 0.7956.

Two things in that result are worth understanding. The first is that the scores are close but not identical: the data is the same, the training is not quite. Section 5.6 explains why. The second is that the replay registered no model. Its evaluation compared it with the champion, which was Run A's own model, version 3, and it did not beat it:

```
champion: auc 0.7951 does not beat 0.7956 by 0
  (difference -0.000485; champion version 3)
```

That is the right outcome. A replay is a check, not a new model: it answers "do we get the same thing again?", and the evaluation record answers it, with the numbers, without adding a copy of a model that already exists.

> **Pause and think.** If an auditor asked tomorrow which rows trained the model you use most, could you produce them, or only the tables they came from, as those tables are today?

5.6 What reproducible means

A run is reproducible when everything it depended on can be had again. There are three things, and this chapter has added the last.

What	Kept by	Chapter
The code, the images and the pipeline	the release, pinned by digest and never changed	7
The configuration: contract, features, label window, gates	the release's contract, written once	2, 7
The data	the run's snapshot, replayed with `--replay`	5

A replay uses the release that is live when it runs. With the same release, it reproduces the run. With a new release, it does something just as useful: it trains the new code on the old data, so the team can tell whether a change in a model's score came from the code or from the data. That is the question the retailer could not answer at the start of this chapter.

Two cautions. First, a replay reproduces the data exactly, but training is only as repeatable as the training code. The retailer's code fixes its random seeds, and the replay still differed in the fourth decimal place. The reason is order: BigQuery returns rows in no fixed order, and the training code shuffles them from the order it receives, so the network saw the same rows in a different sequence. Sorting the rows by their key before shuffling makes the result exact. Training on many machines, or on GPUs with non-deterministic kernels, adds differences of the same small kind. A replay that lands within a rounding of the original has done its job; one that lands far away says something changed.

Since Pipewright 1.2.0. Each run's `data.snapshot.written` event also carries a fingerprint of the rows it read, a SHA-256 over them in a fixed order, so you can tell two runs read the same data without opening either. A replay of a run on 1.2.0 showed the same rows and the same fingerprint as the run it replayed:

```
run             source                      rows   fingerprint
rb059d0a18fe6   lapse_risk_entities       43016
sha256:06a38fff406b83a5...
r1d6b5a6d99c6   snapshot://rb059d0a18fe6  43016
sha256:06a38fff406b83a5...
```

The example's prepare step in 1.2.0 also sorts the rows by the contract's unique key before splitting, which removes the difference

in the fourth decimal place: the same rows, in the same order, split the same way.

Second, a snapshot is a copy of customer data, so it falls under the same rules as the original. Its retention is set like any of the use case's tables (`outputs.bigquery.retention_days`, which may differ per environment), and `pipewright data erase` removes a customer's rows from it along with the use case's other tables, when the customer asks. Keep snapshots as long as your auditors need, and no longer.

Thinking it through

On cost. A snapshot keeps a copy of every run's rows. For the retailer that is a few megabytes a run; for a model trained on billions of rows it is real money. Keep snapshots for the runs that produced models people use, and expire the rest.

On privacy. A snapshot holds customer data, so it needs the same care as the source: a retention period, erasure on request, access as narrow as the source's. Reproducibility is not a reason to keep personal data forever.

On "same data". Two runs can read the same tables and still not read the same rows, because the tables moved in between. When you compare two models, compare what they read, not where they read it from.

Try it: labels that mature, and a run you can replay

This continues from Chapter 4, on the same environment. Start from the `ch04/` folder, or open `ch05/`, which holds this chapter's finished code; run every command from inside the folder. It costs

pennies: a few queries and three training runs of about ten minutes each.

1. Be the data team again. Rebuild the entity table without its label, and load the purchases as if the nightly load had stopped on 9 November 2011:

```
python tools/data_team.py --project pw-ebook sql
lapse_risk_entities.sql
python tools/data_team.py --project pw-ebook sql
customer_purchases.sql --loaded-until "2011-11-09 00:00:00"
```

2. Change the use case. In `config/data.yaml`, rename `lapsed` to `returned` in the contract and the label, and add the `label_source` and `snapshot_table` lines from Sections 5.2 and 5.4. Add the snapshot table to the outputs in `usecase.yaml`:

```
        lapse_risk_snapshots: { schema: schemas/snapshots.json,
partition: { field: ingested_at, type: DAY } }
```

`schemas/snapshots.json` is in the repository. Then validate, create the table and release:

```
pipewright validate --env test
pipewright tables sync --env test
pipewright release --env test
```

3. Run A, in November.

```
pipewright run --env test --usecase lapse-risk \
    --source-uri bq://pw-ebook.pw_retail.lapse_risk_entities --run-
key november
```

When it has finished, open the ingest step's log and find the line that starts `labels from`: how many rows had a closed window, and the anchor.

4. A month passes. Load every purchase:

```
python tools/data_team.py --project pw-ebook sql
customer_purchases.sql --loaded-until "2011-12-10 00:00:00"
```

5. Run B, and a replay of Run A. Find Run A's run ID in `pipewright model list --env test --usecase lapse-risk` (the run behind version 3). Then start both runs together, as the book did, so both are compared with version 3:

```
pipewright run --env test --usecase lapse-risk --source-uri bq://pw-
ebook.pw_retail.lapse_risk_entities --run-key december
pipewright run --env test --usecase lapse-risk \
    --replay <Run A's run id>
```

6. Compare. List the models again: the replay registered nothing, and the ingest and evaluation logs of its run say why. Then count the snapshot rows by run in the BigQuery console:

```
SELECT run_id, COUNT(*) AS row_count, ROUND(AVG(returned), 3) AS
returned_rate
FROM   pw_retail.lapse_risk_snapshots
GROUP  BY run_id
```

7. Leave it costing nothing. No endpoint, nothing running, the launcher at zero; the snapshot table adds about 10 MB.

In short

- A label that takes time to be known is not a 0 while it waits. A row may only be labelled once its whole window has passed; until then it is maturing, and left out.
- Keep the events, not the labels. With `data.label_source`, each run works the label out from the events it can see, within the window after each row's cut-off.
- The window is measured against the newest event, not today, so late data makes a run smaller but never makes a label wrong.
- With `data.snapshot_table`, every run keeps the exact rows it read, label and features included.
- From any model version you can reach its data: the version names its run, the run's rows are in the snapshot, and the rows name their sources. Since 1.2.0 a fingerprint says whether two runs read the same rows.
- `pipewright run --replay` trains again on exactly those rows, whatever has happened to the sources since. With a new release, it tells a change in the code from a change in the data.
- Snapshots are customer data: give them a retention period and erase from them on request.

Next: Chapter 6. Synthetic data for MLOps. When the real rows cannot leave their environment, a copy that can, and how to prove it is both useful and safe.

Chapter 6. Synthetic data for MLOps

When the real rows cannot leave, a copy that can

The retailer's marketing agency wants to help. It runs win-back campaigns for a dozen online shops, and it would like to build its own model of which customers are about to lapse, to compare with the retailer's. It asks for the training data: the 62,689 customer snapshots from Chapter 5.

The answer is no, and it is the right answer. Each row describes a real customer: how recently they bought, how often, how much they spent, where they live. A customer ID that looks harmless can be matched back to a person by anyone who holds the retailer's other records, and the agency holds some of them. The data protection team will not let customer histories leave the company for a modelling exercise, and no contract clause makes that risk go away.

The same wall stands in front of many everyday jobs. A test environment needs realistic data, but copying production data into it puts personal data where more people can see it. A new joiner needs rows to learn the pipeline with. A supplier needs a sample to integrate against. Every time, the choice seems to be between real data and no data.

This chapter is about a third choice: a synthetic copy, learned from the real data inside the environment, that keeps its patterns but none of its people, and that is checked before anyone is allowed to use it.

6.1 What a synthetic copy is, and what it is not

A synthetic copy is a set of rows made by a model that has learned what the real rows look like. It has the same columns and, if it is good, the same shapes: the same spread of spend, the same link between recency and coming back, the same share of customers in each country. But no row in it is a real customer. The customer IDs are new, and the values are drawn from what the model learned, not copied from any one person.

Two things are easy to get wrong.

Synthetic is not anonymous by default. A generator that has memorised its training rows can reproduce them, or come close enough that a real customer can be picked out. Privacy has to be measured, every time, not assumed.

Synthetic is not automatically useful. A generator can produce rows that look right one column at a time but have lost the relationships between columns. Rows like that fool a glance and mislead a model. Usefulness has to be measured too.

So a synthetic copy is only as good as the checks it passes. Without them it is a liability with a reassuring name.

6.2 Two kinds of synthetic data

Pipewright offers two tiers, for two different jobs.

The contract tier uses no real data at all. It writes rows from the data contract of Chapter 3 and a few rules: every column the right type, every value within its range, the label 0 or 1. It runs on a laptop in a second:

```
pipewright synthetic generate --env test --usecase lapse-risk --out
rows.csv --rows 5
```

```
customer_id,snapshot_date,recency_days,orders_12m,spend_12m_gbp,dist
inct_products,country,returned
SYN-00000001,2011-04-24,125,95,3.1783,92,alpha delta zephyr orbit,1
SYN-00000002,2010-07-28,114,70,9.3695,84,echo nova bravo,1
SYN-00000003,2010-06-13,71,12,23.2661,90,nova nova,1
```

Every row keeps the contract, and none of them looks like a customer. A customer who placed 95 orders and spent £3.18 does not exist, and nor does a country called “alpha delta zephyr orbit”. That is the point: the contract tier has seen nothing, so it can leak nothing. It is the right data for continuous integration, for demonstrations and for testing that a pipeline runs. It is the wrong data for anything that learns.

The learned tier is the realistic copy. An engine learns from the real rows, inside the environment where they already live, and generates new ones. Then Pipewright scores what it generated and decides whether anyone may have it. The rest of this chapter is about this tier.

6.3 Switching it on

Synthetic data is off until a use case asks for it. The lapse model’s request is one file, `config/synthetic.yaml`:

```
synthetic:
  enabled: true
  purpose: sharing                        # it leaves the
organisation: the stricter privacy gates
  tier: learned                           # an engine learns from the
real rows, inside this environment
  source_uri: bq://pw_retail.lapse_risk_synthetic_source   # one
training run's labelled rows
  output_table: lapse_risk_synthetic
  rows: 20000
  engine: { type: mostlyai, model: MOSTLY_AI/Small,
max_training_minutes: 15 }
  columns:                                # decided by rules, never
learned
    customer_id: { rule: key, prefix: "SYN-" }

data_protection:
  personal_data_columns: [customer_id]   # never learned, never
copied
```

Three choices in it deserve a word.

The purpose sets the bar. `test-data` is for copies that stay inside the organisation. `sharing` is for copies that leave it, and its privacy gates are stricter. The agency is outside, so the lapse model uses `sharing`.

Keys and personal data are never learned. A column named in `columns` gets its values from a rule, not from the model. `customer_id` becomes `SYN-00000001`, `SYN-00000002` and so on. Any column listed as personal data that has no rule becomes a new key or is left empty. The engine never sees the real values.

The source is a run's rows, not a moving table. The view the source names returns the customers, cut-offs and labels that one training run read, from its snapshot in Chapter 5, and Pipewright joins the

features from the data team's feature table point in time, exactly as training does. So the synthetic copy can be traced to a specific, reproducible set of real rows. (The synthetic pipeline in Pipewright 1.1.0 reads its source like training data but does not apply a label window, which is one more reason to point it at rows that are already labelled.)

Since Pipewright 1.2.0. The synthetic pipeline applies `data.label_source` as training does, label window and anchor included, so `synthetic.source_uri` may also be the entity table itself. Pointing it at one run's snapshot is still the better choice when you want the copy traced to a specific run.

`pipewright doctor` confirms where every column will come from:

```
Synthetic data of lapse-risk
  INFO  purpose sharing, tier learned: engines: mostlyai;
`pipewright synthetic gates` lists the gates
  OK    source pw-ebook.pw_retail.lapse_risk_synthetic_source: 4
contract columns (+4 from feature tables)
```

6.4 Gated, never trusted

A synthetic copy is published only if it passes every gate of its purpose. `pipewright synthetic gates` lists them:

```
lapse-risk in test: purpose sharing, tier learned
  fidelity        min 0.85
  tstr_drop       max 0.08
  exact_copies    max 0
  dcr_ratio       min 1.0
  closer_than_p5  max 0.05
  singling_out    max 0.2
  linkability     max 0.05
  inference       max 0.25
```

They answer three questions.

Question	Gate	In plain words
Does it look like the real data?	`fidelity`	how closely each column and each pair of columns match the real distributions, from 0 to 1
Is it useful?	`tstr_drop`	train a model on synthetic rows, test it on real ones it never saw, and compare with a model trained on real rows: how much AUC is lost
Is it safe?	`exact_copies`	how many synthetic rows are identical to a real one: none allowed
	`dcr_ratio`, `closer_than_p5`	whether synthetic rows sit closer to the training rows than real, unseen rows do, which is the sign of copying
	`singling_out`, `linkability`, `inference`	three attacks run against the copy: can an attacker pick out one real customer, link two records of the same customer, or work out a customer's label?

The usefulness test has a memorable name, **train on synthetic, test on real**. It is the one that matters most to the agency: a copy that looks right but trains a worse model is no use to them.

The attacks are worth a second look. They are not theoretical: the same kinds of attack have been used to re-identify people in data sets that were published as anonymous. Measuring them before release, against real rows the engine never saw, turns “we think it is safe” into a number a data protection officer can sign off.

Pipewright works out every gate itself before it publishes anything, whatever the engine reports, and a team may make the privacy gates stricter but never looser.

If someone offers you “anonymised” or “synthetic” data, these are the questions to ask them: against which real rows was it tested, by which attacks, and what were the numbers? A copy without those numbers is a promise, not evidence.

6.5 The run

The team starts the synthetic pipeline like any other:

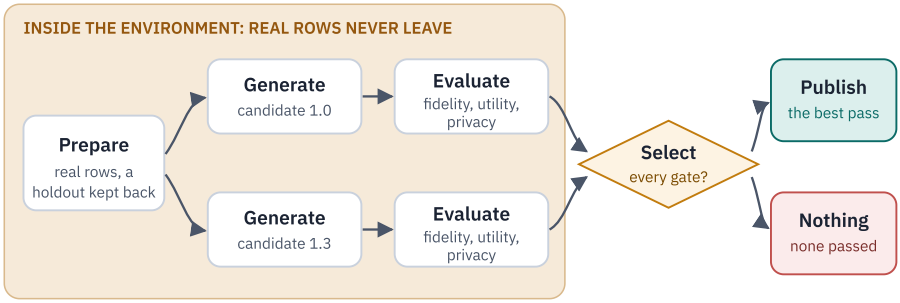
```
pipewright run --env test --usecase lapse-risk --pipeline synthetic
```

It runs in five steps (Figure 6.1). **Prepare** reads the real rows inside the environment, and keeps a holdout back that the engine never sees:

```
synthetic source: 62689 rows, 12538 held out, learning
  ['recency_days', 'orders_12m', 'spend_12m_gbp',
'distinct_products',
   'country', 'returned']
```

`customer_id` and `snapshot_date` are not in that list: their values come from rules. Then two candidates are **generated** and **evaluated** side by side. Both use the same engine and the same training; they differ only in how boldly they sample from what it learned.

`default` samples at temperature 1.0, as the engine learned; `spread` samples at 1.3, which pushes rows a little further from anything it saw. **Select** applies the gates, and **publish** writes the winner, or nothing.



Rules, not learning, give keys and personal data new values: customer_id becomes SYN-00000001.
Only the published table may leave: the generator and the real rows stay inside.

Figure 6.1 The synthetic pipeline. The real rows and the generator never leave the environment; two candidates are generated and scored side by side; the best one that passes every gate is published, and if none passes, nothing is.

Seventeen minutes later the verdict is in the run's `synthetic.evaluated` event:

```
candidates:
  default: closer_than_p5 0.057 is above 0.05
           dcr_ratio 0.9151 is below 1
  spread:  passed
decision: release, chosen: spread
```

The candidate that copied the real data most faithfully was refused. Its fidelity was excellent, but its rows sat slightly too close to the rows it had learned from: closer, on average, than real customers it had never seen. That is what memorising looks like, and for data leaving the company it is not allowed. The bolder candidate passed every gate:

Gate	Limit (sharing)	`default`	`spread`
fidelity	at least 0.85	0.981	0.932
TSTR drop	at most 0.08	0.020	0.018
exact copies	0	0	0
distance ratio (`dcr_ratio`)	at least 1.0	**0.915**	1.468
closer than real (`closer_than_p5`)	at most 0.05	**0.057**	0.029
singling out	at most 0.20	0.104	0.074
linkability	at most 0.05	0.013	0.007
inference	at most 0.25	0.088	0.108

The usefulness numbers are the agency's answer. A model trained on the real rows scores an AUC of 0.777 on the real holdout; the same model trained on the synthetic copy scores 0.759 on the same holdout. The copy costs 0.018 of AUC, well inside the 0.08 allowed.

Pipewright published `spread`: 20,000 rows in `lapse_risk_synthetic`, with a `synthetic.generated` event recording where they went. They look like the retailer's customers without being any of them:

customer	cut-off	recency	orders	spend £	products	country	returned
SYN-00000001	2011-04-24	348	1	203.66	12	United Kingdom	0
SYN-00000002	2010-07-28	187	1	973.57	23	United Kingdom	1
SYN-00000003	2010-06-13	29	4	223.54	10	United Kingdom	0

Put beside the real rows, the copy shows the price of privacy too:

	Real rows	Synthetic copy
Came back within 90 days	42.2%	45.9%
Median spend, last 12 months	£643	£725
Median orders, last 12 months	2	2
Customers in the United Kingdom	91.8%	80.9%

The bolder sampling that made the copy safe also spread it out: fewer British customers, a little more spend. For the agency's purpose, a model of who comes back, that is an acceptable price, and the TSTR gate says so in numbers. For a purpose that depends on the exact country mix, it would not be, and the team would set a stricter fidelity gate and accept that some runs publish nothing. A gate that sometimes says no is working.

> **Pause and think.** Who outside your team asks for real data today: a supplier, an agency, a new joiner's laptop? For each, what is the smallest set of properties the data must keep to be useful to them?

6.6 Where synthetic data fits

The agency's copy is the obvious use, but most of the value of synthetic data in MLOps is quieter.

Need	Tier	Why
CI and unit tests for pipeline code	contract	shape and ranges matter, realism does not, and nothing real may be in a repository
A test environment with realistic data	learned, `test-data`	developers see behaviour like production without seeing production
A supplier, an agency, a hackathon	learned, `sharing`	the copy leaves the organisation, so the strictest gates
More examples of a rare case	neither, on its own	a generator learns the common patterns best; rare cases need care and their own checks

Two limits are worth knowing. A synthetic copy can only be as good as what it learned from: biases in the real data come through. And a copy is a snapshot of patterns at one moment, so when customers change, the copy should be made again from newer rows, which is one schedule line away.

Pipewright can also learn several related tables together (`mode: relational`): an entity table with its feature tables and event histories, keeping keys consistent between them and the timing of each customer's history against their cut-offs. That is the right tool when the agency wants the transactions too, not only the snapshots.

In Pipewright Enterprise, releasing a synthetic copy into another environment needs the approval of named groups, the policy can differ per environment, and each release has its evidence. The open-source core in this chapter generates, gates and publishes the copy inside one environment.

> **Pause and think.** Which of your tests or development environments use a copy of real customer data today? Would a contract-tier copy, with the right shape and no real values, do the same job?

Thinking it through

On who signs off. Privacy gates turn a judgement into numbers, but someone still decides which numbers are good enough. In most organisations that is a data protection officer, not the ML team; bring them in before the first copy, not after.

On fidelity and privacy. The two pull against each other: the closer a copy is to the real data, the more it can leak. Decide which matters more for each purpose, and write that choice into the gates.

On what a copy cannot carry. A generator learns the common patterns. If the use you have in mind depends on rare cases, check those cases in the copy explicitly; the overall numbers will not show their absence.

Try it: a copy you could share

Start from the `ch05/` folder of the book's repository, or open `ch06/`, which holds this chapter's finished code; run every command from inside the folder. It costs pennies: one view, a few megabytes in BigQuery and one synthetic run of about 17 minutes.

1. Try the contract tier on your laptop. Add `config/synthetic.yaml` from Section 6.3 and validate, then:

```
pipewright synthetic generate --env test --usecase lapse-risk --out
rows.csv --rows 5
```

Look at the rows: right shape, no realism.

2. Point the copy at one run's rows. Put the run ID of Run B from Chapter 5 into `usecases/lapse-risk/sql/synthetic_source.sql` and create the view:

```
python tools/data_team.py --project pw-ebook sql usecases/lapse-risk/sql/synthetic_source.sql
```

3. Add the output table and release. Add `lapse_risk_synthetic` with `schemas/synthetic.json` to the outputs in `usecase.yaml`, then:

```
pipewright tables sync --env test
pipewright doctor --env test
pipewright synthetic gates --env test --usecase lapse-risk
pipewright release --env test
```

The release builds two more images, the engine and the evaluation, and pins them by digest like the component image.

4. Run the synthetic pipeline.

```
pipewright run --env test --usecase lapse-risk --pipeline synthetic
```

With no `--source-uri`, the synthetic pipeline reads `synthetic.source_uri`.

5. Read the verdict. In Logs Explorer, search for `synthetic.evaluated`: each candidate's scores and the gates. If a candidate was published, query the output table and compare a few distributions with the real rows.

6. Leave it costing nothing. The synthetic pipeline's jobs end with the run; nothing keeps running.

In short

- A synthetic copy is rows made by a model that learned the real data. It keeps the patterns, not the people, but only if that is measured.
- The contract tier writes rows from the data contract alone: right for CI and demonstrations, wrong for anything that learns.
- The learned tier learns inside the environment, gives keys and personal data new values by rule, and keeps a real holdout back to test against.
- Every copy is scored for fidelity, usefulness (train on synthetic, test on real) and privacy (copies, distances and three attacks). The best candidate that passes every gate is published, or nothing is.
- Point the copy at a run's snapshot, so it traces to a known set of real rows.

Next: Chapter 7. Pipelines that release like software. What a release pins, why a contract is written once, and how a bad release is rolled back in one command.

Chapter 7. Pipelines that release like software

Pin what runs, write it down once, and go back in one command

On a Thursday afternoon a data scientist at the retailer tidies up the shared code the lapse model uses. A helper called `impute`, which fills in missing values, gets a clearer name, `fill_missing`. They update the scoring step that calls it, run the notebook once, and everything works. They commit the change and release it.

The next training run fails five minutes in. One more step, `prepare`, still asks for the old name:

```
ImportError: cannot import name 'impute' from 'mlkit'
(/app/components/mlkit.py)
```

It is the most ordinary bug there is, and it raises the questions this chapter answers. What exactly was released? Can the team get back to the version that worked, now, without rebuilding anything? And could the bug have been stopped before it was released at all?

7.1 What “the pipeline” really is

When a team says “the lapse pipeline”, it means several things that change separately: the team’s Python code; the libraries it needs; the container image they are built into; the pipeline definition that says which step runs after which; and the configuration from Chapters 2 to 6, the contract, features, label window, gates. Change any one of them and the pipeline is different.

Most teams run "whatever is on the main branch" or "the latest image". Both are moving targets. The image tagged `latest` today is not the one tagged `latest` last week, and nobody can say afterwards which one a model was trained with. Software engineering solved this long ago: a **release** is a fixed, named set of exactly what runs, and a deployment says which release is live. Pipewright applies the same idea to ML pipelines.

7.2 What a release pins

`pipewright release` turns one commit of the repository into one release, in five steps (Figure 7.1):

1. **Tables.** It creates or updates the use case's tables, only ever adding to them (Chapter 3).
2. **Images.** It builds the component image from the team's code and requirements, and for Chapter 6 the synthetic engine and evaluation images, and pushes each to the environment's registry. From then on each image is named by its **digest**: a fingerprint of its exact contents, such as `sha256:e4e27f97...`. A tag can be moved to another image; a digest cannot.
3. **Templates.** It compiles each pipeline into a template that refers to the images by digest, and uploads it, itself named by digest.
4. **The contract.** It writes a **release contract**: one file that pins the images, the templates, the commit and the whole resolved configuration of Chapter 2.
5. **Verify, then go live.** It checks that every image and template the contract names exists and matches, and only then moves the environment's **live pointer** to the new release.

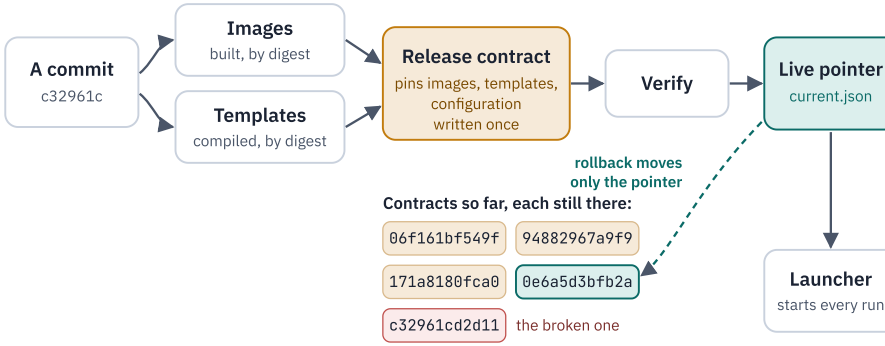


Figure 7.1 A release. A commit becomes images and templates pinned by digest in a contract that is written once. After verification the live pointer moves to it, and the launcher starts every run from the release the pointer names. A rollback moves only the pointer.

The release takes its name from the commit: the data scientist's commit `c32961c...` became release `c32961cd2d11`. A release made from uncommitted changes gets `-dirty-` and a time added to its name, so it can never pass for a clean one. The contract records where it came from:

```
release_id:       c32961cd2d11
source_commit:    c32961cd2d1174de2191dc0a6fd4a3dea584416e
component_image:  .../lapse-risk-components@sha256:34d30482...
template_uri:     .../pw-pipelines/churn-training/sha256:3535149b...
synthetic:        the engine and evaluation images, by digest
static_parameters:
  ml_config:      the contract, features, label window, gates, ...
```

The launcher never reads the repository. When a schedule fires, a file lands or someone calls `pipewright run`, it reads the live pointer, opens the contract it names, and starts the pinned template with the pinned images and configuration. What runs is always exactly a release.

> **On Google Cloud: Artifact Registry.** Google Cloud's store for container images and packages, including compiled Kubeflow pipeline templates. Each image has a digest, a fingerprint of its exact contents.
>
> *Why it matters.* Rerunning a model a year later means running the same code. A registry that keeps every version, named by its digest, makes "the same code" something you can prove.
>
> *In Pipewright.* A release pushes the team's component image and the compiled pipelines here, and the release contract records their digests. Every run of that release uses exactly those.
>
> Read more: https://docs.cloud.google.com/artifact-registry/docs

7.3 Written once

A contract is written once and never changed. Pipewright writes it with a condition that the file does not exist yet, so a second release can never overwrite the first, even by accident or in a race between two releases.

Releasing the same commit again is safe. Pipewright sees that the images and templates would be the same, reuses them, and changes nothing:

```
release lapse-risk → test as 7441e15ebd48
component image 7441e15ebd48-3e52c914fb51 already pushed: reusing it
contract 7441e15ebd48 already published with these images and
templates: reusing it
live: .../current.json → 7441e15ebd48 (was 7441e15ebd48)
```

If the same commit would build something different, because a library or Pipewright itself has changed, the release gets a new name instead of replacing the old contract. Every model ever

trained can therefore point at a contract that still says exactly what trained it. Chapter 17 follows those links from a model back to its code and data.

7.4 A release that passes every check, and still breaks

Here is the data scientist's release, as it went live:

```
contract: gs://pw-ebook-pw-artifacts/contracts/lapse-
risk/c32961cd2d11.json
  verified: contract v3 is valid
  verified: component image exists (by digest)
  verified: synthetic engine image exists (by digest)
  verified: synthetic evaluation image exists (by digest)
  verified: training template churn-training/sha256:3535149b3c77...
matches the contract
  verified: synthetic template synthetic/sha256:0f36d9877950...
matches the contract
live: .../current.json → c32961cd2d11 (was 0e6a5d3bfb2a)
```

Every check passed, because every check was about the release itself: the contract is well formed, the images exist, the templates are the ones named. None of them can know that one line of Python asks for a function that no longer exists. The first training run found out instead:

```
lapse-risk-rf3e7aa38e1ca   FAILED   16:39 → 16:44
  ingest-data      SUCCEEDED
  validate-data    SUCCEEDED
  prepare-data     FAILED
  train-model      NOT_TRIGGERED
```

A release proves that what runs is what was built. It does not prove that what was built is right. That needs the team's own tests, in Section 7.6.

Keep the two questions apart in your own reviews. "Is this the code we meant to ship?" is the release's job, and it can be checked by a machine. "Is this code right?" is your tests' job, and no release process answers it for you.

7.5 Rollback in one command

Until the code is fixed, every run will fail the same way: the weekly schedule, a landing file, a retrain. The team does not need to fix the code first. It moves the live pointer back to the release that worked:

```
pipewright rollback --env test --usecase lapse-risk
```

```
lapse-risk in test: c32961cd2d11 → 0e6a5d3bfb2a (rollback)
```

It takes about six seconds. Nothing is rebuilt: the earlier release's images and templates are still there, pinned in its contract, and Pipewright verifies them again before it moves the pointer. `--to` names any earlier release instead of the previous one. `pipewright status` shows the history:

```
lapse-risk (test): live 0e6a5d3bfb2a since 2026-10-30T16:46:00+00:00
    contract v3, pipelines: training, synthetic
    previous c32961cd2d11  (2026-10-30T16:39:24+00:00)
    previous 171a8180fca0  (2026-10-27T10:40:29+00:00)
    previous 94882967a9f9  (2026-10-27T10:23:13+00:00)
    previous 06f161bf549f  (2026-10-27T09:38:54+00:00)
```

The broken release is not deleted. It stays in the history and its contract stays in the bucket, because a failed run points at it, and that record should survive.

Rollback is for the pipeline: the code and configuration that train and score. The model a pipeline produced has its own controls, the champion in the registry (Chapter 10) and what an endpoint serves (Chapter 13), and they are rolled back separately.

> **Pause and think.** How long would it take your team to put back last week's pipeline after a bad change, and would you be sure it was exactly last week's? Who is allowed to do it at night?

7.6 Catching it before the release

The bug took five minutes of compute and a failed run to find. A test finds it in a second. The retailer adds one to the use case: a short script that reads every component and checks that each name it imports from the team's own modules exists. `usecase.yaml` says how to run it:

```
tests:
  command: python tests/check_imports.py
```

and `pipewright test` runs it for every use case. Against the data scientist's commit:

```
RUN   lapse-risk: python tests/check_imports.py  (in
...\usecases\lapse-risk)
prepare.py:27: mlkit has no impute
FAIL  lapse-risk: exit 1
```

With the import fixed, the same test passes, and the fix goes out as a new release:

```
imports OK in 14 component files
OK    lapse-risk: exit 0
```

```
live: .../current.json → 7441e15ebd48 (was 0e6a5d3bfb2a)
```

The next training run on it succeeds in seven minutes and registers version 5:

```
lapse-risk-classifier (test)
  v5  DEFAULT  2026-10-30 16:53:59  run r16de39249507, auc=0.7847
[gates pass]
  v4           2026-10-27 10:53:22  run rbcb69cdc615a, auc=0.7836
[gates pass]
```

`pipewright release` itself does not run the tests. The place for them is before the release: on the laptop with `pipewright test`, and in CI, where Chapter 16 makes every pull request run them, so a failing test stops a change from being merged at all. A test of this kind is not a substitute for real unit tests of the team's code; it is the cheapest net under the commonest mistake.

Since Pipewright 1.2.0. A release can run the tests itself, before it builds anything, so a broken commit never becomes a release at all, and a run that fails anyway is reported (Chapter 3):

```
  tests:
    command: python tests/check_imports.py
    on_release: true                # pipewright release runs it
first, and stops on a failure
    environment: components         # with the components' own
requirements installed (numpy, pandas)
```

`environment: components` matters as soon as a test imports a component: the test then runs in an environment built from `components/requirements.txt`, as in the component image, and not in Pipewright's own.

7.7 Why this matters beyond one bug

Releases pinned by digest, contracts written once and a pointer that can move back give the retailer three things it did not have.

Repeatability. A run can be repeated with exactly the same code and configuration, and combined with the snapshot of Chapter 5, on exactly the same data.

Speed of recovery. A bad release costs one failed run and a six-second rollback, not an afternoon of rebuilding the old version from memory.

An answer to "what ran?" For any run, any model and any day, there is a contract that says, and it cannot have been edited since.

The next chapter shows that none of this depends on writing Python. The retailer's analysts build models in SQL, and their releases work the same way.

> **Pause and think.** For the model you rely on most, could you say which code trained the version in use today, down to the library versions? If not, where would you have to look, and how sure would the answer be?

Thinking it through

On "latest". Every tag that moves, `latest` on an image, a branch name, an unpinned library, is a decision to let something else choose what runs. Pin by digest where it matters, and know where you have chosen not to.

On rollback. A rollback is only as good as the thing it returns to. Keep old releases runnable, and test a rollback before you need one, the way you would test a backup.

On tests. The cheapest test is the one that catches the commonest mistake. Start with an import check and a contract check, and add tests where your failures actually happen.

Try it: break a release and roll it back

Start from the `ch06/` folder of the book's repository, or open `ch07/`, which holds this chapter's finished code. It costs pennies: two short training runs.

1. See what is live.

```
pipewright status --env test
```

2. Break it on purpose. In `usecases/lapse-risk/components/mlkit.py`, rename `def impute(` to `def`

`fill_missing(`, and change the two lines of `score.py` that use it, but not `prepare.py`. Commit, then:

```
pipewright release --env test
pipewright run --env test --usecase lapse-risk \
    --source-uri bq://pw-ebook.pw_retail.lapse_risk_entities --run-
key broken
```

Watch the run fail at `prepare-data`, and find the `ImportError` in the step's log.

3. Roll back.

```
pipewright rollback --env test --usecase lapse-risk
pipewright status --env test
```

4. Add the test. Copy `usecases/lapse-risk/tests/check_imports.py` from `ch07/`, add the `tests` block to `usecase.yaml`, and run `pipewright test --env test`: it names the line that breaks.

5. Fix and release. Change `prepare.py` to import and call `fill_missing`, run `pipewright test` again, commit and release. Release the same commit a second time and read what it reuses.

6. Leave it costing nothing. Nothing keeps running after a training run.

In short

- A pipeline is code, libraries, images, a pipeline definition and configuration. A release pins all of them.
- Images and templates are named by digest, which cannot be moved the way a tag can.

- A release contract is written once and never changed; releasing the same commit again reuses it.
- The launcher runs only what the live pointer names, so what runs is always exactly a release.
- A release proves that what runs is what was built, not that the code is right. Tests do that, before the release.
- `pipewright rollback` moves the pointer back in seconds, without rebuilding anything.

Next: Chapter 8. Models for SQL teams. The retailer's analysts build customer segments and a demand forecast in BigQuery ML, released, gated and approved like any other model.

Chapter 8. Models for SQL teams

BigQuery ML in production, with the same gates as everything else

The retailer's analysts know their customers and their products better than anyone. They live in BigQuery, they write SQL every day, and they have two questions the lapse model does not answer. Marketing wants to know what kinds of customer the retailer has, so it can speak to each kind differently. The buyers want to know how many of each best-selling product to order for the weeks ahead.

Neither question needs a neural network. Both can be answered inside BigQuery with BigQuery ML, which trains models with a SQL statement. What the analysts lack is not a way to train a model, but a way to put one into production that the rest of the company will trust: tested data, a bar the model must clear, a record of what it learned from, a person who approves it, and someone watching it afterwards.

This chapter gives the analysts the same governed path as the ML team, without asking them to write Python.

8.1 The SQL team's path

BigQuery ML trains a model with `CREATE MODEL` and uses it with `ML.PREDICT`, both plain SQL, and the data never leaves BigQuery. For an experiment that is all you need. For production, a model needs everything Chapters 2 to 7 built, and it is tempting to rebuild that in scheduled queries: one to refresh the data, one to retrain, one to score, and a spreadsheet of which model is which.

> **On Google Cloud: BigQuery ML.** Machine learning inside BigQuery: `CREATE MODEL` trains a model from a query, and `ML.PREDICT`, `ML.EVALUATE` and `ML.FORECAST` use it, all in SQL.
>
> *Why it matters.* Many useful models (segments, forecasts, simple classifiers) do not need Python or a training machine. Teams who know SQL can build them where the data already is.
>
> *In Pipewright.* A use case with a `bqml` section has no pipeline code: Pipewright's own pipelines submit the team's SQL as BigQuery jobs, and the model goes through the same gates, registry and monitoring as any other.
>
> Read more: https://docs.cloud.google.com/bigquery/docs/bqml-introduction

Pipewright takes the other route. A use case with a `bqml` section has no pipeline code at all: Pipewright provides the training and scoring pipelines, and their steps submit BigQuery jobs. The analysts write YAML and SQL, and the model goes through the same steps as the lapse model (Figure 8.1).

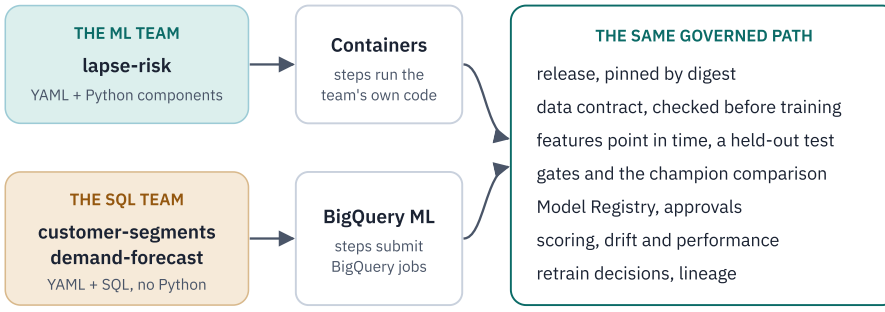


Figure 8.1 Two teams, one path. The ML team's steps run its Python in containers; the SQL team's steps submit BigQuery jobs. Both go through the same release, contract, gates, registry, approvals and monitoring.

They start a use case the way the ML team did, from an example:

```
pipewright new-usecase --name customer-segments --tenant retail \
    --owner sql-team --from bqml
pipewright new-usecase --name demand-forecast --tenant retail \
    --owner sql-team --from bqml
```

8.2 Customer segments

The segments come from the same feature table as the lapse model, `customer_activity` from Chapter 4, read point in time. The entity table is the customers who bought in the year before 1 December 2011: 4,298 of them, built by one SQL file in the use case. There is no label: segments are found, not predicted. The use case asks BigQuery ML for four of them with k-means, which groups customers by how close their numbers are:

```
spec:
  bqml:
    model_type: kmeans
    options: { num_clusters: 4, kmeans_init_method: KMEANS++ }
    transform: sql/transform.sql

  evaluation:
    gates:
      - { metric: davies_bouldin_index, max: 1.5 }
```

and in `config/data.yaml`:

```
data:
  label: { task: clustering }
  split: { strategy: random, test_fraction: 0.2 }
  features_from:
    entity: { columns: [customer_id], time_column: snapshot_date }
    sources:
      - table: bq://pw_retail.customer_activity
        features: [recency_days, orders_12m, spend_12m_gbp,
distinct_products]
features:
  exclude: [snapshot_date]
```

The gate needs a word. A clustering has no right answer to check against, so it is measured on its shape. The **Davies-Bouldin index** compares how spread out each segment is with how far apart the segments are. Compact segments far apart score low; overlapping ones score high. The team chose 1.5 as its bar, and set it before training anything.

Two refusals and what they taught

The first run did not reach the gate. It failed in its evaluation step with `Invalid cast from DATE to FLOAT64`: Pipewright 1.1.0 had counted the cut-off date, `snapshot_date`, as one of the features. With a split by time, as in the lapse model, the cut-off is left out automatically; with a random split it is not. One line fixed it, the `features.exclude` above. A cut-off says when, not who, and it should never be a feature.

Since Pipewright 1.2.0. A BigQuery ML use case leaves the entity's cut-off column out of its default features, and out of the default drift features, so `customer-segments` trains without the `features.exclude` line. `validate` warns where an entity's key column would still be a feature, and on the container path warns about both.

The second run reached the gate and was refused:

```
decision: reject
reasons:  gate davies_bouldin_index: 4.2792 is above the maximum 1.5
```

The segments it found show why:

Segment	Customers	Average spend, 12 months
1	4,264	£1,485
2	27	£49,599
3	4	£204,171
4	3	£71,962

That is not four kinds of customer. It is everybody, plus three tiny groups of the retailer's biggest buyers. Spend, orders and product counts have long tails: a few customers spend a hundred times what most do, and k-means, which works on distances, spent its segments on them. The gate was right to refuse it.

The cure is a log scale, which makes going from £200 to £2,000 count as much as going from £2,000 to £20,000. The team put it where BigQuery ML applies it inside the model, in the use case's `TRANSFORM` clause, `sql/transform.sql`:

```
LN(1 + recency_days)  AS recency_days,
LN(orders_12m)        AS orders_12m,
LN(1 + spend_12m_gbp) AS spend_12m_gbp,
LN(distinct_products) AS distinct_products
```

A transform in the model is applied the same way when it trains, when it scores and behind an endpoint, so the tables keep the real

values and nobody has to remember to apply it. (The team first tried it in `prepare.sql`, the other place SQL can go. In Pipewright 1.1.0 the data contract of a BigQuery ML use case checks the rows after `prepare.sql`, so the logs broke the contract's own rules, `orders_12m: values below 1`. A transformation the model needs belongs in `TRANSFORM`.)

Since Pipewright 1.2.0. `bqml.contract_on: source` checks the contract on the rows as read, before `prepare.sql`, as the container path does. Preparation may then rescale or derive columns freely: in a 1.2.0 run, `prepare.sql` multiplied recency by ten, past the contract's maximum of 365, and the run passed, because the contract checked the rows before preparation. `TRANSFORM` is still the place for what the model needs at serving time.

The third run passed, with a Davies-Bouldin index of 1.481, and was registered as version 1. Its segments are ones marketing can use:

Segment	Customers	Median spend	Median orders	Days since last order
Loyal regulars	639	£3,886	10	7
Steady customers	1,334	£1,315	4	30
One-order customers	1,576	£416	1	82
Fading	749	£158	1	174

The model does not name them; people do, from numbers like these. The names belong in the model card, and the numbers in the evaluation record the run keeps.

> **Pause and think.** Who in your organisation builds models in SQL today, perhaps without calling them models? Do their scheduled queries get the same review, gates and records as the data science team's code?

8.3 A demand forecast

The buyers want weekly units for the products in the most orders. A forecast is a different kind of model: there are no features and no customers, just a series of numbers over time, one series per product. The use case says what the series is and how far ahead to look:

```
spec:
  bqml:
    forecast:
      horizon: 8                          # scoring forecasts the
next eight weeks
      id_columns: [stock_code]            # one series per product
      confidence_level: 0.9

  evaluation:
    gates:
      - { metric: mape, max: 0.5 }        # off by at most half, on
average, on the held-out weeks
```

and `config/data.yaml` names the series, `units` over `week_start`, with `task: forecast`. Pipewright fits one `ARIMA_PLUS` model per product, a classical method that learns a series' trend and its weekly and yearly patterns. To test it honestly, it holds back the latest fifth of the weeks, forecasts them from the weeks before, and compares the forecast with what was really sold. Only then does it refit on every week and register the model.

The series is the model

The SQL that builds the series, `sql/weekly_units.sql`, turned out to be most of the work, and every line of it is a decision about the business.

Which products? The first version took the ten products with the most units sold. Two of them were there by accident: a single order of 74,215 storage jars that was cancelled the same day, and 80,995 paper birds ordered in the data's last week. Ranking by the number of orders instead of units, and leaving out purchases that a cancellation of the same size reversed, gave ten products that genuinely sell every week.

Every week, even empty ones. A series with gaps is not a series. Every product gets every week, zero when nothing sold, so the Christmas closures appear as the zeros they were.

No part weeks. The data ends on Friday 9 December 2011. That last week is left out, because five days of sales would look like a collapse in demand.

Sales are not demand

Even so, the first forecast was refused:

```
decision: reject
reasons:  gate mape: 0.5052 is above the maximum 0.5
```

Pipewright 1.1.0 records the forecast's error over all products together, so the team asked the run's test table which product it came from. (Since Pipewright 1.2.0 the evaluation record holds every series' metrics, and a gate may apply per series with `per_series: true`; the refusal then names the products, as it did on a later run: `gate mape per series: 21212 0.6164 is above the`

`maximum 0.5`.) Nine of the ten were off by 28 to 62% on average. The tenth, a party bunting, was off by 125%. Its monthly sales show why:

```
2010:  Apr 1319  May 2179  Jun 752  Jul 6  Aug 3  Sep 1004
2011:  Apr 1485  May 3213  Jun 1577  Jul 2249  Aug 1435
```

Bunting sells all summer, except in July and August 2010, when it sold almost nothing. Nobody stopped wanting bunting that summer; the retailer most likely ran out of it. A forecast trained on sales learns that bunting collapses every July, and the next July it was badly wrong.

Sales are what happened; demand is what would have happened if the shelf had been full. Every demand forecast trained on sales inherits every stock-out. The team did not lower the gate to let the forecast through. They made a reviewed change to the series instead: the bunting is left out, with the reason in a comment in the SQL, until its stock-out weeks can be marked, and the next product by orders takes its place. The second forecast passed:

```
decision: approve      mape 0.434   mae 197.6 units a week   22
weeks held out
demand-forecast (test)
  v1  DEFAULT  2026-10-30 17:13:00  run ra6179fa33d26, mae=197.5556
[gates pass]
```

The scoring pipeline then forecast the next eight weeks of every product into `demand_forecast_scores`, with a 90% interval. For the retailer's best seller, the white hanging heart T-light holder:

Week from	Forecast	90% interval
5 Dec 2011	1,441	1,069 to 1,814
12 Dec 2011	800	399 to 1,201
19 Dec 2011	165	−235 to 566
26 Dec 2011	−68	−469 to 333

The model has learned the Christmas run-down and the closure from the two Christmases before it. It has not learned that units cannot be negative: ARIMA knows nothing about shelves. Whoever uses the forecast should floor it at zero, and the scoring SQL is the place to do it.

A forecast's average error hides its worst products, and the worst products are often the ones that matter: the seasonal ones, the ones that stock out. When you set a gate on a forecast, ask what a bad week for one product costs, not only what the average is.

> **Pause and think.** Which of your forecasts or scores is used by a person who can see it is impossible, a negative stock or a probability above one, and which is fed straight into another system that cannot?

8.4 Everything else is the same

Both models went through the same path as the lapse model, which is the point of the chapter:

- **Released and pinned.** `pipewright release` pinned each use case's pipelines and its SQL files, so the SQL a model was trained with is part of its release.

- **Checked before training.** The data contract ran in SQL on every run.
- **Gated.** Both were refused before they were accepted, and both refusals were right.
- **Registered and approved.** Each passing model is a new version in Model Registry, with the same approval rules as the lapse model; `pipewright model approve` and `deploy-model` work the same way.
- **Scored and watched.** The scoring pipeline writes segments and forecasts to tables, and monitoring can watch the segments' features drift (Chapter 14).
- **Cheap.** Every BigQuery job has a byte limit, 100 GiB unless the platform sets less; the segments and the forecast billed less than a gigabyte between them.

BigQuery ML has its own limits. It offers a fixed set of model types, from linear and boosted trees to k-means and ARIMA; it does not tune hyperparameters through Pipewright yet; and a forecast is not served online. Within those limits, the analysts ship models as the ML team does, from YAML and SQL alone.

Thinking it through

On choosing the tool. BigQuery ML is the right first choice when the data is already in BigQuery and a standard model type fits. Reach for Python when the model needs something the SQL types cannot do, not because Python feels more serious.

On one path. The reason to put SQL models through the same release, gates and approvals is not tidiness. It is that an auditor, or your successor, should not need to know which team built a model to know how it was governed.

On refusals. Both models here were refused before they were accepted, and both refusals taught the team something about its

data. Treat a refused run as information, not as an obstacle to work around.

Try it: segments and a forecast in SQL

Start from the `ch07/` folder of the book's repository, or open `ch08/`, which holds this chapter's finished code. It costs pennies: a few small BigQuery jobs and five short pipeline runs.

1. Start the two use cases with `pipewright new-usecase --from bqml`, then replace their YAML and SQL with the files in `ch08/usecases/customer-segments/` and `ch08/usecases/demand-forecast/`.

2. Build the two source tables.

```
python tools/data_team.py --project pw-ebook sql \
    usecases/customer-segments/sql/customers.sql usecases/demand-
forecast/sql/weekly_units.sql
```

3. Release and train.

```
pipewright tables sync --env test
pipewright release --env test --usecase customer-segments
pipewright release --env test --usecase demand-forecast
pipewright run --env test --usecase customer-segments \
    --source-uri bq://pw-ebook.pw_retail.segment_customers
pipewright run --env test --usecase demand-forecast \
    --source-uri bq://pw-ebook.pw_retail.weekly_product_units
```

4. Repeat the refusals. Remove `transform:` from the segments use case and run again: the gate refuses lopsided segments. Remove the bunting line from `weekly_units.sql`, rebuild the table and run the forecast again: the gate refuses it by half a percent.

5. Score.

```
pipewright run --env test --usecase demand-forecast --pipeline
inference \
    --source-uri bq://pw-ebook.pw_retail.weekly_product_units
```

and read `demand_forecast_scores` in BigQuery.

6. Leave it costing nothing. Nothing keeps running. Before you ever delete the dataset, drop the BigQuery ML models first (`DROP MODEL`), which also removes them from Model Registry: deleting a dataset with models in it leaves registry entries behind that cannot be removed.

In short

- With a `bqml` section, a use case has no pipeline code: Pipewright's pipelines submit BigQuery jobs, and the SQL team writes YAML and SQL.
- Segments are gated on their shape (Davies-Bouldin), forecasts on their error on held-out weeks (MAPE, MAE).
- A cut-off says when, not who: it is never a feature.
- Skewed customer numbers need a log scale before clustering; put transformations the model needs in `TRANSFORM`.
- The series is the model: which products, every week, no part weeks, no reversed orders.
- Sales are not demand: a stock-out looks like nobody wanting the product.
- A gate that refuses is information. Fix what the model learns from; do not move the bar.

Next: Chapter 9. Training at scale. When one machine is not enough: Ray steps, compute set in configuration, and a cluster that exists only for the length of a run.

Chapter 9. Training at scale

Compute set in configuration, and a cluster that lives for one step

The lapse model trains in about seven minutes on a small machine, and for now that is plenty. But the data scientist has a list of things to try: a bigger network, more training passes, and above all a search over the model's settings, its **hyperparameters**: how many hidden units, how fast it learns, how much it forgets on purpose to avoid memorising. Each combination is a separate training run, and the search wants eight of them, then perhaps eighty.

On a laptop that is an afternoon. In a pipeline it is a question of compute: how big a machine each step gets, how many things run at once, and who pays for the machines that sit idle in between. This chapter answers those questions in configuration, and then measures, honestly, what the extra compute buys.

9.1 Compute is configuration

How much machine a step needs is a decision that differs between environments and that the platform team cares about, so by Chapter 2's test it belongs in configuration, not in code. Each step can be sized by name:

```
steps:
  train-model: { cpu: 4, memory: 16G }
```

Pipewright applies this to the compiled pipeline, so the training code does not change, and Agent Platform picks a machine that fits. The same block can ask for a GPU (`gpu: { type: NVIDIA_L4, count: 1 }`), retry a flaky step a set number of times (`retries`), or stop a

step from reusing a cached result (`cache: false`). `steps` is one of the keys that may differ per environment, so test can train on a small machine and production on a large one without the gates or the data changing.

Since Pipewright 1.2.0. The platform team can cap GPUs per environment, which types and how many per step or per endpoint, under `policies.accelerators` in the manifest, and `validate` refuses a step or a deployment that asks for more; `model.serving.accelerator` serves a model on a GPU within the same limits. A use case may also name its own base image, a CUDA image for GPU training for example, with `components.base_image` , as long as the platform's list of approved images allows it.

> **On Google Cloud: Machines for training.** Each pipeline step runs on a machine Agent Platform starts for it. The machine type sets its processors and memory, and a step can add GPUs.
>
> *Why it matters.* The right machine is a trade-off of time against money. Too small and training is slow or runs out of memory; too large and every run pays for capacity it does not use.
>
> *In Pipewright.* `steps` in the configuration sizes each step by name, per environment, and the platform's policies cap what a team may ask for. The training code does not change.
>
> Read more: https://docs.cloud.google.com/gemini-enterprise-agent-platform/machine-learning/training/configure-compute

9.2 Searching for better settings

Tuning is switched on in `usecase.yaml` . The team gives the defaults, the ranges to search and how many trials to try:

```
  training:
    hyperparameters: { hidden_units: 32, epochs: 40, learning_rate:
0.003, dropout: 0.1 }
    tuning:
      enabled: true
      max_trials: 8                     # the first trial is always
the defaults
      parallel_trials: 4                # how many run at once
      search_space:
        learning_rate: { type: double, min: 0.0005, max: 0.02,
scale: log }
        hidden_units:  { type: discrete, values: [16, 32, 64] }
        dropout:       { type: double, min: 0.0, max: 0.3 }
```

`pipewright validate` immediately refused it, for a good reason:

```
training.tuning: trials are scored on a validation split; set
data.split.validation_fraction
```

Trials must be compared on data the final model is not tested on. If the trials were scored on the test set, the search would pick whichever settings happen to suit those rows, and the test score would no longer be an honest estimate. So the split gains a validation slice, just before the test period in time:

```
  split: { strategy: time, time_column: snapshot_date,
validation_fraction: 0.1 }
```

The training pipeline then plans the eight trials, runs them four at a time as separate pipeline tasks, picks the best on validation, and trains the final model with it. The run took nine minutes, and every trial's score is in the run:

Hidden units	Learning rate	Dropout	Validation AUC
64	0.0067	0.06	**0.7938**
32	0.003	0.10	0.7935 (the defaults)
32	0.0044	0.13	0.7932
64	0.00055	0.07	0.7929
16	0.00071	0.13	0.7928
64	0.00069	0.20	0.7926
16	0.0076	0.07	0.7918
16	0.0077	0.19	0.7904

Then the final model met the champion from Chapter 7:

```
champion: auc 0.7833 does not beat 0.7847 by 0 (difference
-0.001399; champion version 5)
```

9.3 What the extra compute bought

Almost nothing, and that is the most useful result in this chapter.

Every setting in the search scored between 0.790 and 0.794 on validation. The defaults came second. The best trial beat them by three ten-thousandths. On the test period, the tuned model did slightly worse than the champion, and the champion comparison of Chapter 10 rightly kept the champion.

The reason is the data, not the search. The lapse model has four features: recency, orders, spend and products. With four numbers per customer there is only so much any model can learn, and a small network learns most of it with its default settings. More trials, more machines or a bigger network will not find signal that is not in

the features. Chapter 4's feature table is where the next real improvement lies: how the customer's orders are spread over the year, what they buy, whether they responded to the last campaign.

So before scaling anything, measure what scale buys. Run a small search first. If the spread of scores across it is tiny, as here, the money is better spent on features. If it is wide, a bigger search, and the machines for it, are worth paying for. Pipewright makes either answer cheap to get: the trials, their settings and their scores are part of every run.

> **Pause and think.** When did your team last make a model better by giving it more compute, and when by giving it better data? Which of the two does your budget make easier to ask for?

9.4 When one machine is not enough: Ray

Some teams do need scale: a search over hundreds of combinations, a model that takes an hour to train, data that does not fit in one machine's memory. Running hundreds of pipeline tasks is slow, because each one starts its own machine. **Ray** is an open-source framework that spreads Python work across a cluster: one head machine that coordinates, and workers that do the work. Agent Platform runs Ray clusters for you.

On Google Cloud: Ray on Agent Platform. Agent Platform creates and runs Ray clusters for you: a head node and workers, on machine types you choose, in your project and network.

Why it matters. Ray spreads Python work across many machines, which suits large searches and models that do not fit on one machine. A managed cluster saves running Ray yourself, but it bills while it exists.

In Pipewright. A Ray step gets a cluster of its own, created for it and deleted after it, on every run. On private GKE, KubeRay plays the same part (Appendix C).

Read more: https://docs.cloud.google.com/gemini-enterprise-agent-platform/machine-learning/open-source/ray

The catch with clusters is that they cost money while they exist, whether anyone is using them or not. Pipewright's rule is that a pipeline never uses a cluster someone left running: **each Ray step of each run gets its own cluster, created for it and deleted after it** (Figure 9.1).

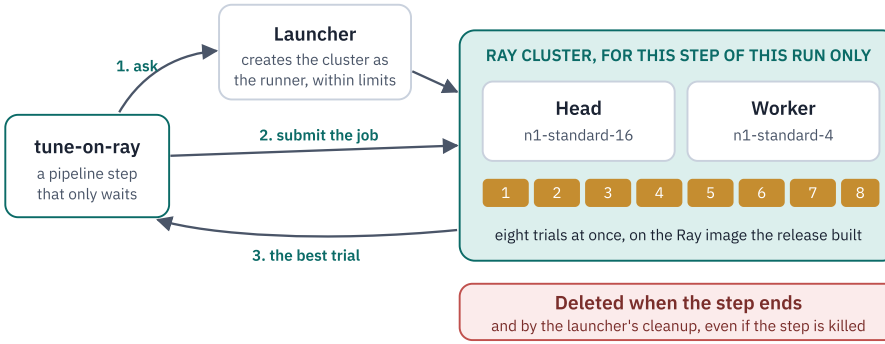


Figure 9.1 A Ray cluster for one step of one run. The step asks the launcher, which creates the cluster as the tenant's runner from the live release and within the platform's limits. The trials run at once on the cluster, the best comes back, and the cluster is deleted when the step ends, or by the launcher's cleanup if the step is killed.

Switching the lapse model's tuning to Ray is three changes. The pipeline file becomes one with a `tune-on-ray` step, taken from Pipewright's Ray example; the Ray packages go into the requirements; and the step asks for its cluster:

```
  pipelines:
    training:
      file: training_pipeline_ray.py
      function: churn_ray_training_pipeline
  components:
    requirements: components/requirements.txt   # with the Ray
client
    ray_requirements: ray-requirements.txt      # for the cluster's
nodes

  steps:
    tune-on-ray:
      ray:
        head_machine: n1-standard-16
        worker_machine: n1-standard-4
        workers: 1
        max_minutes: 60
```

`pipewright release` then builds one more image, the Ray image the cluster runs, with the team's code and packages on the platform's Ray base image, and pins it by digest like every other image:

```
verified: Ray image exists (by digest), for tune-on-ray
```

The step's code asks for the cluster and submits the tuning job to it; Pipewright's runtime does the rest:

```
with ray_cluster(ml_config, "tune-on-ray", run_id) as cluster:
# created by the launcher, deleted after
    ray_job(ml_config, cluster, f"python -m ray_tune_job
{trials.uri}", run_id=run_id)
```

The run took thirty minutes, and its events tell where they went:

Time	What happened
17:58	The run starts: ingest, validate, prepare, as always.
18:03	The launcher creates the cluster: one `n1-standard-16` head, one `n1-standard-4` worker, the Ray image by digest.
18:19	The cluster is ready, and the tuning job is submitted.
18:24	Eight trials are done; the best comes back, and the cluster is deleted.
18:28	The final model is trained and evaluated.

```
8 trials on 1 Ray node(s); best auc=0.794586 {'hidden_units': 64,
'learning_rate': 0.00664561,
  'dropout': 0.0596513, ...}
champion: auc 0.7833 does not beat 0.7847 by 0 (difference
-0.001399; champion version 5)
```

The answer is the same as Section 9.2's: the same best settings, the same final model, the same AUC to four places, and the same decision to keep the champion. Two quite different ways of running the search agreeing exactly is itself worth knowing.

The run also has two lessons about paying for scale.

Starting the cluster took longer than using it. Sixteen minutes passed between asking for the cluster and the first trial; the trials themselves took five. For eight small trials, the pipeline's own tasks

were faster (six minutes for all eight) and cheaper. A cluster per run pays off when the work on it is long: hundreds of trials, or trials that each take an hour.

The worker did nothing. Each trial asks Ray for one CPU, and the 16-vCPU head had room for all eight at once, so Ray never needed the worker. The team paid for a machine for twenty minutes to stand idle. For this search, `workers: 0` would have done the same work for less. Size a cluster from what the trials ask for, and read the line Pipewright logs, `8 trials on 1 Ray node(s)`, after the first run.

Afterwards there is nothing left running. The cluster was deleted when the step ended, and the launcher's cleanup would have deleted it after `max_minutes` if the step had been killed.

What the platform team decides

The cluster is the platform team's money, so the platform team sets its limits in the environment manifest: whether Ray is on at all, which machine types and accelerators clusters may use, how many workers, and how many minutes a cluster may live. A Ray step that asks for more is refused at `validate`. The launcher creates every cluster as the tenant's runner identity, with the environment's encryption key and network, and labels it with the run.

Ray also needs a few permissions that Pipewright's starter environment does not grant, because Ray is optional there: the launcher must be allowed to create and delete clusters, and each tenant's runner must be allowed to act as itself. For this book they were added by hand:

```
P=pw-ebook                                  # your project
LAUNCHER=pw-launcher@$P.iam.gserviceaccount.com
RUNNER=pw-retail-run@$P.iam.gserviceaccount.com
R=aiplatform.persistentResources
PERMS=$R.create,$R.delete,$R.get,$R.list

gcloud iam roles create pipewrightRayClusters \
    --project $P --permissions $PERMS
gcloud projects add-iam-policy-binding $P \
    --member serviceAccount:$LAUNCHER \
    --role projects/$P/roles/pipewrightRayClusters
for ROLE in roles/iam.serviceAccountUser \
            roles/iam.serviceAccountOpenIdTokenCreator; do
  gcloud iam service-accounts add-iam-policy-binding $RUNNER \
      --member serviceAccount:$RUNNER --role $ROLE
done
```

Pipewright Enterprise's Terraform makes these grants itself.

Ray on Agent Platform, or on GKE

Ray on Agent Platform has limits worth knowing before choosing it:

- **A head of at least 16 vCPUs and more than 18 GB of memory.** With less, the cluster starts but never answers its job API; `validate` checks both.
- **A new cluster for every run,** which takes minutes to start before any trial runs. There is no long-lived cluster on Agent Platform.
- **No Ray inside a VPC Service Controls perimeter.** A step reaches its cluster through Agent Platform's dashboard connection, which an enforced perimeter blocks.

Organisations that run inside a perimeter, which is most regulated ones, run Ray on private GKE instead, where Pipewright creates a

KubeRay cluster inside the Kubernetes cluster for each run, or gives a team a standing cluster whose workers scale down to zero. The step's code is the same on both; only the orchestrator and who owns the cluster change. Appendix C compares the two in more detail.

> **Pause and think.** Who in your organisation would notice a cluster left running over a weekend, and how soon? What stops a training job from asking for the largest machine available?

Thinking it through

On scale. Scale is a cost you choose, not a property of serious work. Most of the models a business needs train in minutes on one machine; reach for a cluster when a measurement says you need one.

On limits. Compute limits in configuration are a kindness to the ML team, not a restriction: they turn a surprise on next month's bill into a refusal today, with a reason.

On where Ray runs. If you work inside a perimeter, decide early whether Ray on Agent Platform or on GKE fits your rules; moving later changes who runs the cluster, not the code.

Try it: tune, then tune on Ray

Start from the `ch08/` folder of the book's repository, or open `ch09/`, which holds this chapter's finished code. Tuning in the pipeline costs pennies; the Ray run costs well under a pound, most of it the 16-vCPU head for about twenty minutes.

1. Size a step and tune in the pipeline. Add the `steps` and `training` blocks from Sections 9.1 and 9.2 to `usecase.yaml`, run `pipewright validate --env test` and add the validation split it asks for. Release and run, then read every trial's settings and score in the run's `tune-trial` tasks.

2. Grant what Ray needs. On your own environment, run the `gcloud` commands in Section 9.4 with your project and identities.

3. Switch to Ray. Copy `training_pipeline_ray.py`, `components/ray_tune.py`, `components/ray_tune_job.py`, `components/requirements.txt` and `ray-requirements.txt` from `ch09/`, point `pipelines.training` at the Ray pipeline, and add the `tune-on-ray` step. Run `pipewright test --env test` to check the imports, then release (the first Ray image takes a while to build) and run.

4. Watch the cluster. While the run is in its `tune-on-ray` step, open **Ray on Agent Platform** in the console: the cluster is there, labelled with the run. When the step ends, it is gone.

5. Leave it costing nothing. The cluster deletes itself. If you ever kill a run, the launcher's cleanup deletes the cluster after its `max_minutes`.

In short

- How much machine a step gets is configuration (`steps.<name>`): CPU, memory, GPU, retries, caching. It may differ per environment; the gates and data may not.
- Tuning needs a validation split, so the test set stays an honest estimate. Every trial's score is part of the run.
- Measure what scale buys before paying for it. Here the whole search spread over 0.004 of AUC, and the champion was kept: the next gain is in features, not compute.

- A Ray step gets a cluster for that step of that run only, created by the launcher within the platform's limits and always deleted.
- Ray on Agent Platform needs a 16-vCPU head, starts a cluster per run and does not work inside a VPC Service Controls perimeter; inside one, Ray runs on private GKE with the same step code.

Next: Chapter 10. Evaluation gates. The bar a model must clear, the champion it must beat, the slices it must not fail, and whether it treats customers fairly.

Chapter 10. Evaluation gates

The bar a model must clear, and who decides where it sits

Every chapter so far has ended with a model being registered or refused, and every time the decision came from a gate the team wrote down in advance. This chapter is about those gates: what they should measure, how strict they should be, and what to do when one says no.

It matters more than it seems. A model that reaches the registry is trusted by everyone downstream. The marketing team that sends the win-back emails does not read evaluation records; it assumes that a registered model is a good one. The gates are the only place where "good" is defined, so whoever writes them is deciding, on behalf of everyone else, what the model must be.

10.1 Four questions before a model is trusted

A new lapse model has to answer four questions, all on the same held-out test rows that no step of training has seen (Figure 10.1).

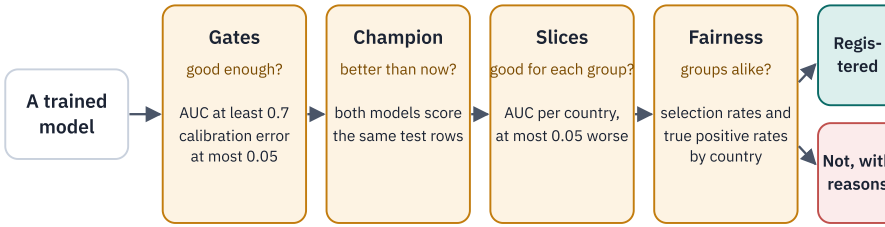


Figure 10.1 Four checks between a trained model and the registry, on the same held-out test rows: absolute gates, the champion, slices and fairness. A model that passes all four is registered; one that fails any is not, and the reasons are recorded.

Is it good enough? Absolute gates set a floor that holds whatever the champion is. The retailer's are an AUC of at least 0.7, and a calibration error of at most 0.05: when the model says 70%, about 70% of those customers should really come back. A model can rank customers well and still be badly calibrated, and calibration matters as soon as someone uses the score as a probability, to budget a campaign for example.

Is it better than the one we have? The champion comparison you met in Chapter 5: the new model and the current default both score the same test rows, and the new one must win on the primary metric.

Is it good enough for everyone? A model can be good on average and poor for a group. **Slices** measure it per group, and a **slice gate** limits how much worse any group may be.

Does it treat groups alike? Fairness measures compare what the model does to each group: who it selects, and whom it gets right.

The use case says all of it in one block:

```
  evaluation:
    primary_metric: auc
    decision_threshold: 0.5
    gates:
      - { metric: auc, min: 0.7 }
      - { metric: calibration_error, max: 0.05 }
    champion: { enabled: true, metric: auc, min_improvement: 0.0 }
    slices: [country]
    slice_gates:
      - { metric: auc, max_gap: 0.05, min_rows: 100 }
    fairness:
      attributes: [country]              # measured, never trained
on
      min_rows: 100
      gates:
        - { metric: demographic_parity_difference, max: 0.1 }
        - { metric: equal_opportunity_difference, max: 0.1 }
```

Notice `country`. The lapse model has never been trained on it: Chapter 3's `features.exclude` keeps it out. That is exactly why it is worth measuring by. A model can treat groups differently without ever seeing the column that names them, through the features that go with it, and the only way to know is to look.

Notice too `min_rows`. A slice of twelve customers is too small to judge: one or two wrong predictions swing its numbers either way. Groups below the limit are listed in the record, but not measured or gated.

On Google Cloud: Model Registry. Agent Platform's catalogue of models: each model has versions, each version its files, labels and a default (the champion), and any version can be deployed.

Why it matters. Without a registry, "which model is in production, and where did it come from?" has no reliable answer. A version that records its run is the start of every audit.

In Pipewright. Only a model that passes its gates is registered, labelled with its run, release and evaluation. Promotion copies a version between environments, and deployment always names a version.

Read more: https://docs.cloud.google.com/gemini-enterprise-agent-platform/machine-learning/model-registry/introduction

10.2 The first run

The team released the change and trained. The absolute gates passed: AUC 0.7847, calibration error 0.042. The slices passed too. Only three countries had 100 or more test rows, and none was more than 0.05 below the overall AUC:

Country	Test rows	AUC	Calibration error
United Kingdom	11,862	0.784	0.042
Germany	223	0.754	0.126
France	203	0.841	0.143

The fairness gates did not:

```
fairness gate demographic_parity_difference: country is 0.1734,
  above the maximum 0.1
fairness gate equal_opportunity_difference: country is 0.1192,
  above the maximum 0.1
```

And there was a twist. The model this run trained was identical to the champion, version 5, to the last decimal: the same code, the same data and the same seed give the same model, which is Chapters 5 and 7 working as intended. So the refusal was not really about a new model. The model the retailer already used had the same gaps, and nobody had measured them, because the gates did not exist when it was registered. `pipewright model list` now shows exactly that, judging every version against the gates in force today:

```
v5   2026-10-30 16:53:59  run r16de39249507, auc=0.7847
     [gates FAIL: slice gate auc: the evaluation record has no
slices]
```

Since Pipewright 1.2.0. `pipewright model check --version 5` re-checks one version against today's gates and says, gate by gate, what passes, what fails and what its record cannot answer, so the question "does the model we use still meet today's rules?" is one command, not a reading of `model list`. Each evaluation also measures the champion's slices and fairness on the same test rows as the candidate, so the next comparison has numbers on both sides.

> **Pause and think.** Which of the models your organisation uses today were registered before anyone measured how they treat different groups? If you measured one tomorrow and it failed, what would happen next, and who would decide?

10.3 Reading what the model does

The fairness record gives, for each measured country, what the model did at the decision threshold. Remember that the label is `returned`: a positive prediction means "this customer will come back", and those customers do not get the win-back offer.

Country	Predicted to return	Returners recognised (true positive rate)	Customers who really returned
United Kingdom	29.7%	52.3%	41.5%
Germany	47.1%	64.2%	60.0%
France	42.9%	62.5%	57.5%

Two different things are going on, and the two fairness measures see them differently.

Demographic parity compares the first column: how often each group is selected. Germany is predicted to return 47% of the time and the UK 30%, a gap of 0.17. But look at the last column: German customers really do come back more often, 60% against 42%. A model that selected every country at the same rate would be wrong, sending win-back offers to Germans who were coming back anyway and missing British customers who were not.

Equal opportunity compares the middle column: of the customers who really came back, how many did the model recognise? Here the

UK is genuinely behind, 52% against 62 to 64%. A British customer who is going to come back is more likely than a German one to be taken for a lapser, and sent a discount they did not need.

The team tried the obvious repair before anything else: give the model a flag for UK customers. In a quick local test it changed nothing; the gap was 0.105 with the flag and 0.097 without. The difference is in how customers behave, not in what the model can see.

10.4 When a gate says no

This is the moment that defines how your gates work in practice, so it is worth slowing down.

When a gate fails, ask first whether the model is wrong or the question is. In Chapter 8, the forecast failed a gate that was plainly right: a forecast off by half is a bad forecast. The data was wrong, the team fixed the data, and the bar stayed where it was. Lowering it would have hidden a real problem behind a passing grade.

Here, one of the two questions was wrong. Demographic parity asks every country to be selected equally, and for a model of who comes back, when countries genuinely come back at different rates, that is not a fair outcome. It is an inaccurate one. The gate was not too strict; it measured something that should not be equal in this use.

Choosing a fairness measure is a judgement about harm, and it belongs to the people accountable for that harm, not to the person whose model is being judged. For the retailer, the question went to its data protection lead, with one fact on the table: what happens to a customer when the model is wrong? Here, a British customer who would have come back anyway gets a 10% discount email. The retailer loses a little margin; the customer loses nothing. For a loan, a job application or a medical decision, the same gap would mean

something very different, and the measure and the limit would be too.

The decision was written into the use case, where every later reader will find it:

```
    fairness:
      attributes: [country]              # measured, never trained
on; all three measures are recorded
      min_rows: 100
      gates:
        # Reviewed with data protection after the first measurement:
the model chooses who gets a win-back
        # discount. Customers abroad genuinely come back more often,
so equal selection rates would send offers
        # to the wrong people: measured, not gated. What matters is
that returners are recognised alike in every
        # country; a miss costs the retailer a discount, not the
customer, so the gap may be up to 0.15.
        - { metric: equal_opportunity_difference, max: 0.15 }
```

Four things keep a change like this honest, and you should insist on all of them.

1. **It is a reviewed change.** It goes through a pull request like any other, and the reason is in the file, not in someone's memory.
2. **The people accountable for the harm make it,** not the person whose model is being judged.
3. **The limit comes from the harm, not from the score.** The team set 0.15 because of what a miss costs. Had the model measured 0.16, the answer would have been to keep the gate and improve the model, not to move the limit to 0.17. If you could not have written the reason down before you saw the number, it is not a reason.

4. **Nothing is hidden.** Demographic parity is still measured and recorded for every model; it is simply no longer a reason to refuse one.

10.5 The second run

With the reviewed gate, the same data and the same code trained the same model again, and this time it passed every check. It was registered as version 6 and became the default:

```
lapse-risk-classifier (test)
  v6  DEFAULT  2026-10-30 18:52:53  run rc4d0bacee32d, auc=0.7847
[gates pass]
  v5           2026-10-30 16:53:59  run r16de39249507, auc=0.7847
[gates FAIL: slice gate auc: ...]
```

Version 6 predicts exactly what version 5 did. What changed is what the retailer knows about it: its slices, its calibration per country and its fairness measures are in its evaluation record, and the decision about which fairness measure applies, and why, is in the use case. If a regulator, a journalist or a customer asks how the model treats people in different countries, there is now an answer, with numbers, and the name of the decision behind it.

The gap the UK customers face has not gone away. It is now known, it is within a limit someone accountable chose, and it is a target for the next chapter, which is about better features.

> **Pause and think.** Think of one model you know well. If it made a wrong prediction about someone, what would actually happen to them? Would you choose demographic parity, equal opportunity or something else to measure it, and what gap would you accept?

10.6 Choosing your gates

A few rules of thumb, from this chapter and the ones before it:

- **Set gates before you train.** A bar set after seeing the score is a description, not a gate.
- **Gate what someone relies on.** If the scores are used as probabilities, gate calibration. If a campaign targets the top tenth, gate precision there.
- **Slice by what you are accountable for.** Countries, channels, customer types, regions. If a group would complain, measure it.
- **Size slices honestly.** Below a hundred or so rows, a slice's numbers are noise; group small categories together ("rest of Europe") rather than pretending to measure them.
- **Measure fairness on attributes the model does not see.** That is where unfairness hides.
- **Make the champion earn its place.** `min_improvement` stops a new model that is better by a rounding error from replacing one that people already trust.
- **Expect gates to say no.** In this book, nearly as many runs have been refused as accepted. Every refusal taught the team something. A gate that never refuses is not doing its job.

Since Pipewright 1.3.0. A platform team can set the least each gate may ask in an environment, in the manifest: `policies.evaluation.min_gates: { auc: 0.70 }`. A use case with

no such gate, or a weaker one, is refused by `validate` like any policy, so a bar cannot be lowered until it passes. `validate` also warns about choices that pass and still mislead, such as a random split on rows ordered in time, or a date used as a feature (Chapter 20).

Thinking it through

On whose measure it is. The retailer's data protection lead made the call, but in many organisations nobody owns fairness for a given model. If yours is one, the first step is not choosing a metric but naming a person.

On small groups. Thirty-seven of the retailer's forty countries were too small to measure. If you serve many small groups, they may be exactly the customers a model treats worst, and the least likely to be seen. Grouping them, or collecting enough data to measure them, is part of the work.

On models already in use. The most important finding of the chapter was about the model already in production. New gates apply to new models; check the old ones too.

Try it: gates that say no

Start from the `ch09/` folder of the book's repository, or open `ch10/`, which holds this chapter's finished code. It costs pennies: two training runs of about nine minutes.

1. Add the gates. Put the `evaluation` block from Section 10.1 into `usecase.yaml` and run `pipewright validate --env test`. `pipewright explain evaluation.fairness` describes each measure.

2. Train, and read the refusal.

```
pipewright release --env test --usecase lapse-risk
pipewright run --env test --usecase lapse-risk \
    --source-uri bq://pw-ebook.pw_retail.lapse_risk_entities --run-
key gates
```

Query `lapse_risk_model_metrics` for the run's `decision_reasons`, and read the per-country numbers in the evaluate step's `report` artifact.

3. See the champion judged. Run `pipewright model list --env test --usecase lapse-risk` and read what it says about the version that is still the default.

4. Make the decision, and write it down. Decide, for your own reasons, which fairness measure fits this use and what gap you accept. Put it in the `fairness` block with a comment saying why, commit it, release and run again.

5. Leave it costing nothing. Nothing keeps running.

In short

- Gates are where "good" is defined for everyone who will rely on a model. Set them before you train.
- A model is checked four ways on the same test rows: absolute gates, the champion, slices and fairness.
- Measure fairness on attributes the model never sees; groups too small to measure are listed, not judged.
- Demographic parity asks for equal selection; equal opportunity asks for equal recognition. Which one fits depends on what a wrong prediction does to someone.
- When a gate fails, ask whether the model is wrong or the question is. Fix the model or its data; change the question

only in the open, with the people accountable for the harm, for a reason that does not depend on the score.
- New gates judge new models. Check the ones already in use.

Next: Chapter 11. Features that earn their place. The gates now say what better means. The next chapter finds features that make the lapse model better, and lets the gates decide which ones stay.

Chapter 11. Features that earn their place

What a feature is, how to find one, and how to know it helps

Chapter 9 ended with an uncomfortable finding. Eight trials, a cluster and an afternoon of compute moved the lapse model's AUC by less than the noise between random seeds. The model was not short of settings. It was short of information: four numbers per customer, recency, orders, spend and products, say only so much about whether someone will come back.

The next real improvement was always going to come from features. This chapter is about finding them, and above all about choosing them, because the hardest part of feature engineering is not having ideas. It is knowing which ideas to keep.

11.1 What a feature is

A feature is a fact about the thing you are predicting for, known at the moment of the prediction, that changes the answer. Each part of that sentence rules something out.

A fact about the customer. Not about the retailer, the weather or the economy, unless those differ between customers.

Known at the moment of the prediction. Chapter 4's rule: a feature computed from anything after the cut-off is leakage, however well it scores.

That changes the answer. A fact that does not help the model tell lapsers from returners is not a feature; it is cost. It has to be computed, stored, checked by the contract, monitored for drift and

explained to anyone who asks why a customer was scored as they were.

You will find more facts than you can use. A transaction table alone gives you hundreds: counts and sums over every window, ratios of each to each, trends, gaps, shares of every category. The question is never "what could we compute?" It is "what would a person who knows these customers want to know?"

11.2 Where good candidates come from

Three places, in this order.

The business question. The lapse model asks whether a customer will buy again within 90 days. Ask the people who run the win-back campaigns what makes them worry about a customer. They will tell you things the data team has not thought of: "they used to order every month and they've gone quiet", "they returned a lot last time", "they only buy at Christmas".

How people behave. Customer behaviour has a few dimensions that matter almost everywhere: how recently, how often, how much, and how those are changing. The lapse model had the first three. It had nothing about change, and nothing about each customer's own rhythm.

What already exists. Before building a feature, look in the feature tables (Chapter 4). Somebody may already compute it, tested and owned, and another copy of the same idea under a different name is how definitions drift apart.

From those three places the retailer's team listed seven candidates, each computable from the transactions as of each cut-off:

Candidate	The idea behind it
Average days between orders	each customer’s own rhythm
Months active in the year	regular or occasional
Share of spend in the last 3 months	spending rising or falling
Orders in the last 3 months	recent activity
Returns rate	returns as a sign of dissatisfaction
Days since first order	how long they have been a customer
Share of Christmas items	seasonal buyers

11.3 Measure before you build

Building a feature properly means a column in the data team’s feature table, a backfill of its history, a line in the contract, monitoring and a release. That is too much work to do for an idea that might not help. So test the ideas cheaply first, on a laptop, point in time, on the same time split the model is judged on.

The team added each candidate on its own to the four existing features, trained a quick model and measured the test AUC. Then it trained the same model with three different random seeds, to see how much the score moves by chance alone: about 0.002 either way (Figure 11.1).

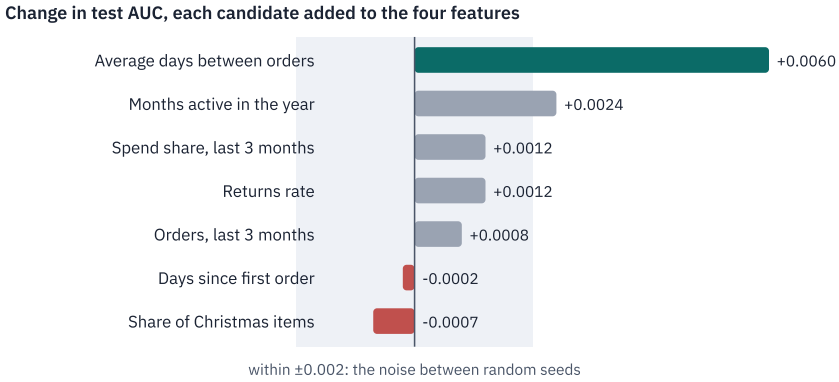


Figure 11.1 The change in test AUC when each candidate is added to the four existing features. Only one is clearly above the noise between random seeds, shaded. Two make the model worse.

One candidate stood out. **Average days between orders** added 0.006, three times the noise. Everything else was inside the noise, and two candidates made the model slightly worse. All seven together scored only 0.0007 more than the four plus the winner, which is nothing at all for six more features to build and run.

The winner makes sense once you see it. Recency, "days since the last order", means very different things for different customers. A customer who orders every 30 days and has not ordered for 60 is overdue. A customer who orders twice a year and has not ordered for 60 days is perfectly normal. The rhythm tells the model what recency means for each customer.

> **Pause and think.** Which of your model's features would a person who knows your customers find obvious, and which would surprise them? What is the one fact they would want that the model does not have?

11.4 Build it once, properly

The feature belongs to the data team, so it goes into their feature table, `customer_activity`, as one more column, computed the same way for every month:

```
  -- the customer's rhythm: average days between their orders in the
year, empty with one order
  ROUND(SAFE_DIVIDE(TIMESTAMP_DIFF(MAX(p.invoice_date),
MIN(p.invoice_date), SECOND) / 86400,
                    COUNT(DISTINCT p.invoice) - 1), 1)  AS
avg_days_between_orders
```

Three rules from Chapter 4 decide how.

Add, do not rename or retype. Other models read this table. A new column changes nothing for them.

Backfill the history honestly. A model needs the feature for every past cut-off, each computed only from what was known then. The team rebuilt the table's history with the new column, then let the monthly job carry on adding a month at a time.

Say what empty means. A customer with one order in the year has no rhythm. The column is empty for them: 38.5% of rows. The contract says so, so an empty value is expected rather than an error:

```
      - { name: avg_days_between_orders, type: float, min: 0,
max_null_fraction: 0.5 }
```

On the use case's side, the change is one more name in the list of features it reads:

```
        features: [recency_days, orders_12m, spend_12m_gbp,
distinct_products, avg_days_between_orders]
```

`pipewright tables sync` added the column to the use case's own tables, an additive change, and `doctor` confirmed the feature table now offers five features.

11.5 The run: better, and not only better

The new model trained in seven minutes and met every gate. The champion comparison let it through:

```
champion: auc 0.7862 vs 0.7847 (champion version 6)
lapse-risk-classifier (test)
  v7  DEFAULT  2026-10-30 19:11:57  run r41d1e2a0888d, auc=0.7862
[gates pass]
```

Two things about that result are worth your attention.

The gain was smaller than on the laptop. The quick test showed 0.006; the real model gained 0.0015. The difference is how each model handles an empty value. The quick test used gradient-boosted trees, which treat "empty" as information in its own right: a customer with no rhythm is a one-order customer, and those come back far less often (22%, against 54% of the rest). The lapse model is a small neural network, and its preparation step fills empty values with the median, so every one-order customer gets a typical 47-day rhythm they do not have. A feature's worth depends on the model that uses it, and on what happens to its missing values on the way in. If you test features with one kind of model and train with another, expect the numbers to differ, and decide on the real run.

Better on average, a little worse for some. The fairness record from Chapter 10 shows what the new feature did to each country:

	Version 6	Version 7
AUC overall	0.7847	0.7862
AUC, Germany	0.754	0.765
UK customers who came back, recognised	52.3%	50.2%
Equal opportunity gap (limit 0.15)	0.119	0.141
Calibration error (limit 0.05)	0.042	0.048

The rhythm helped the model with German customers and slightly hurt it with British returners. The gap the team agreed to watch in Chapter 10 grew, and is now close to its limit; so is calibration. Without the gates, nobody would know. With them, the team can see that its next feature has very little room to make the UK gap worse, and that the gap is where its next piece of work should go.

> **Pause and think.** Your last model improvement: did you check whether it was better for everyone, or only on average? Who would have noticed if it was worse for some of your customers?

11.6 Feature tables, feature groups or an online store

Chapter 4 showed how Pipewright reads features. With a real feature now added, it is worth stepping back and asking which of the three ways to hold features fits which problem.

You have...	Use	Because
Features one use case needs, owned by the team that uses them	a **feature table** in BigQuery, read point in time	the simplest thing that keeps history and stops leakage
Features several use cases share, owned by a data team	the same **feature table**, in the data team's dataset, listed for each tenant	one definition, one owner, one backfill
Many teams and many features, and a need to find, describe and govern them	**feature groups** registered in Agent Platform's Feature Store	names, descriptions and owners in a registry; Pipewright reads the same table through it
A model behind an endpoint that needs a customer's features in milliseconds	an **online store**, synced from the same feature table	the same definition served fast, so production sees what training saw (Chapter 13)

The lapse model's rhythm feature now lives in a shared feature table, and the customer segments of Chapter 8 can read it too, without anyone writing it again. If the retailer grows to dozens of features and several data teams, the next step is to register them as feature groups, so that people can find them by name and see who owns them. The checkout offer of Chapter 13 will need the online store. None of these choices changes the use case's YAML much; each changes who owns the feature and how fast it can be read.

11.7 Choosing, not collecting

A last word on restraint. It is tempting, once features are cheap to compute, to add all of them and let the model sort them out. Resist it.

Every feature you keep has to be computed every month, backfilled whenever its definition changes, checked by the contract, watched for drift (Chapter 14), explained when someone asks why they were

scored as they were, and kept correct for as long as the model lives. A feature that adds nothing measurable adds all of that cost and some risk: one more input that can break, drift or leak.

So keep a feature only when it earns its place on the same test the model is judged on, by more than the noise, without making the model worse for a group you are accountable for. Take the rest off the list. The retailer's team built one feature out of seven candidates, and that one is the one that improved the model.

Thinking it through

On where the next gain is. The UK gap is now the team's clearest problem. A feature that helps the model recognise British returners, perhaps one about the products they buy, would be worth more than another point of overall AUC. What would you look for in your own data to close a gap like that?

On missing values. The lapse model throws away the information in "empty" by filling it with the median. A team could add an explicit flag, or change how the model prepares its inputs. When did you last check what your pipeline does with an empty value?

On ownership. The new feature is now part of the data team's table, which every model can read. Who should approve a change to its definition once three models depend on it?

Try it: one feature, measured and built

Start from the `ch10/` folder of the book's repository, or open `ch11/`, which holds this chapter's finished code. It costs pennies: a feature table rebuild and one training run.

1. Measure the candidates on your laptop. The script computes the seven candidates as of each cut-off, adds each in turn to the four the model has, and prints the test AUC on the time split:

```
pip install pandas openpyxl scikit-learn
python tools/candidate_features.py online_retail_II.xlsx
```

Which ones are above the noise? Add a candidate of your own to the script and measure it the same way.

2. Add the winner to the feature table. Put the `avg_days_between_orders` column into `data-team/sql/customer_activity_backfill.sql` and `customer_activity_month.sql`, then rebuild the history and add the latest month:

```
python tools/data_team.py --project pw-ebook sql
customer_activity_backfill.sql
python tools/data_team.py --project pw-ebook sql
customer_activity_month.sql \
    --as-of "2011-12-01 00:00:00"
```

3. Read it in the use case. Add the column to the contract and to `features_from`, and to the snapshot table's schema. Then:

```
pipewright validate --env test
pipewright tables sync --env test
pipewright doctor --env test --usecase lapse-risk
pipewright release --env test --usecase lapse-risk
pipewright run --env test --usecase lapse-risk \
    --source-uri bq://pw-ebook.pw_retail.lapse_risk_entities --run-
key rhythm
```

4. Read the verdict, all of it. `pipewright model list` shows whether the champion changed. Then compare the fairness and calibration numbers in the evaluate step's report with the previous version's. Better on average is not the end of the question.

5. Leave it costing nothing. Nothing keeps running.

In short

- A feature is a fact about the customer, known at prediction time, that changes the answer. Anything else is cost.
- Find candidates from the business question, from how people behave (how recently, how often, how much, how it is changing) and from the feature tables you already have.
- Measure candidates cheaply first, on the model's own test split, against the noise between seeds. Keep only what is clearly above it.
- Build the keeper once, in the owner's feature table: add, do not rename; backfill history honestly; say what empty means.
- A feature's worth depends on the model and on how it handles missing values. Decide on the real run.
- Check a better model for everyone, not only on average: the gates and fairness records exist for exactly that.
- Feature tables for history without leakage, feature groups to find and govern features at scale, an online store to serve them fast.

Next: Chapter 12. Promotion between environments. Version 7 is the best lapse model the retailer has had. The next chapter takes it to production, and makes sure that what was tested is exactly what ships.

Chapter 12. Promotion between environments

What was tested is exactly what ships

Version 7 of the lapse model is the best the retailer has had, and the marketing team wants it scoring real customers. Until now everything in this book has happened in one environment, test. Production is a different project, run by the platform team, with different people allowed to change it, and the retailer's customers on the other end.

The question for this chapter sounds simple: how does the model the team tested get into production? The common answer is "we rebuild it there": run the same pipeline in production, from the same branch, and trust that the result is the same. It usually is. When it is not, a library has moved, an image tag points somewhere new, a setting differs, and nobody finds out until the production model behaves differently from the one everybody approved.

Pipewright takes the other answer. Production receives the tested release itself, and the tested model itself, copied and checked, never rebuilt.

12.1 Why two environments

Test is where the team experiments. Models are trained, refused and retrained there; gates change; releases come and go. Production is where the retailer relies on the result. Keeping them apart, in separate Google Cloud projects, gives three protections:

- **Separate permissions.** The data scientists can do anything in test. In production, the platform team decides who can change what.

- **Separate data and cost.** A runaway experiment cannot touch production data or its budget.
- **A deliberate step between them.** Nothing reaches production by accident: someone promotes it.

The retailer's production environment was built the same way as test in Chapter 3, from Pipewright's starter, in a project of its own, `pw-ebook-prod`. Its platform policy says that a person makes a model the champion there and a person deploys it. In the repository, production gets an environment file of its own, with one line that matters:

```
apiVersion: pipewright/v1
kind: Environment
metadata: { name: prod }
spec:
  manifest: gs://pw-ebook-prod-pw-
artifacts/pipewright/environment.json
  promotion: { from: test }             # releases arrive here only
from test
```

and the use case's production overrides say the same as the platform's policy:

```
# usecases/lapse-risk/environments/prod.yaml
approvals:
  promote_model: manual
  deploy_model: manual
```

Everything else is as in test: the contract, the features, the gates. Chapter 2 made sure those cannot differ.

12.2 Production needs its data too

Promotion moves code and models. It does not move data, and it should not: production has its own.

`pipewright doctor --env prod` made that plain on its first run:

```
Feature tables of lapse-risk
  FAIL  pw-ebook-prod.pw_retail.customer_activity: Not found
```

The model reads its features from the data team's feature table, and in production that table did not exist yet. In a real organisation the data team runs its jobs in every environment, against the data that lives there. For the book, the data team ran the same SQL in production as in test: the transactions, `customer_activity` with all five features, the entity table and the purchases. The same files, so the same definitions, which is the point of keeping them in the repository.

> **Pause and think.** Which tables does your model need in production that nobody has listed? If your test environment were deleted tomorrow, could production still run, and score, on its own?

12.3 The rules of promotion

`pipewright promote` takes a release from the environment it was tested in to the next one. Four rules decide whether it may.

Only from the named source. Production says `promotion: { from: test }`, so a release can reach it only from test.

From the release's own commit. Promotion runs from a clean checkout of the commit the release was built from, so the configuration production resolves is the configuration that was tested.

With a model that release trained. The model being promoted must have been trained by the release being promoted. That rule shaped the retailer's day. Adding the production environment file was itself a change to the repository, so it made a new release. The team released it to test, trained once more, and got version 8: the same model as version 7, now trained by the release it would be promoted with. Only then could it promote. The rule sounds fussy; it is what guarantees that the code in production and the model in production belong together.

Under production's rules. The use case is resolved under production's manifest first, with its approval policies and limits. A refusal there stops the promotion before anything is copied.

12.4 Promotion

With the release live in test and version 8 its default model, the team promoted (Figure 12.1):

```
pipewright promote --from test --to prod --usecase lapse-risk --
approve
```

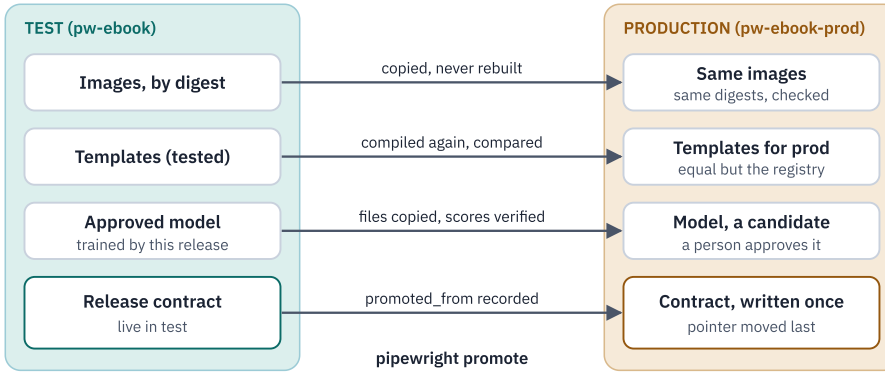


Figure 12.1 Promotion copies the tested release and its model from test to production: images by digest, templates compiled for production and compared with the tested ones, the model's files verified and registered as a candidate, the contract written once and the live pointer moved last. Nothing is rebuilt.

It took two and a half minutes:

```
promote lapse-risk release 4671cbcae0a4: test → prod (model)
copying lapse-risk-components@sha256:79c1a7bc...
  from .../pw-ebook/pw-images  to .../pw-ebook-prod/pw-images
copying lapse-risk-synthetic@sha256:6075a0db...
copying lapse-risk-synthetic-evaluation@sha256:e5dde293...
model files: 7 copied to gs://pw-ebook-prod-pw-artifacts/...,
  contents verified
model: lapse-risk-classifier v8 (test) → v1 (prod), a candidate
live in prod: 4671cbcae0a4 (first release)
  verified: component image exists (by digest)
  verified: training template churn-training/sha256:1672ce02c582...
```

Each line is a promise kept.

- **The images were copied, not rebuilt.** Each arrives in production's registry with the same digest it had in test, so the code that runs in production is byte for byte the code that was tested.

- **The templates were compiled for production and compared.** They differ from test's only in the registry address. If anything else changed, an override that altered a pipeline's shape for example, promotion would stop and ask for `--accept-template-change`, which is recorded.
- **The model's files were copied and verified.** Test's version 8 became production's version 1, registered as a candidate.
- **The contract was written once, and the pointer moved last.** Until the last step, production went on running whatever it ran before; if anything had failed half-way, nothing would have gone live.

Production's model list records where its model came from:

```
lapse-risk-classifier (prod)
  v1  DEFAULT  2026-10-30 19:42:13
      promoted from test v8 (release 4671cbcae0a4)  [gates pass]
```

A model's first version in Model Registry becomes the default by itself, so the policy that a person makes a model the champion was met with `--approve` on the promotion, and recorded with who gave it. From the second version on, a promoted model arrives as a candidate, and someone runs `pipewright model approve --env prod ... --approve` to make it the champion.

12.5 Safe to repeat, easy to undo

Running the same promotion again does nothing, quickly:

```
lapse-risk release 4671cbcae0a4 is already live in prod:
nothing to do
```

That matters more than it seems. Promotions run from CI (Chapter 16), and CI jobs get retried. A step that is safe to repeat cannot be harmed by a retry.

And if something goes wrong in production, Chapter 7's rollback works there too: `pipewright rollback --env prod` moves production's pointer back to its previous release in seconds, without rebuilding or re-promoting anything.

> **Pause and think.** How does code reach production in your organisation today: is it the artefact that was tested, or something built again from the same source? If it is rebuilt, what would tell you the two were different?

12.6 What promotion does not do

Promotion moves a release and a model. Three things it deliberately leaves alone.

It does not move data. Production's tables are production's. The data team's jobs, and the use case's own tables from `pipewright tables sync --env prod`, are set up there in their own right.

It does not serve. The model is the champion in production's registry, but nothing is calling it yet. Putting it on an endpoint, carefully, is the next chapter.

It does not retrain. With `model.promotion: model`, as here, production never trains its own lapse model; it receives the tested one. A team can choose `model.promotion: retrain` instead, so that production trains its own candidate on its own data; the SQL team's forecast in Chapter 8 does that, because a forecast belongs to the data it was fitted on.

Thinking it through

On who promotes. Here the data scientist promoted and approved in one command. In most organisations those should be two people: the team that built the model asks, and someone accountable for production approves. Chapter 16 puts that approval in front of the right people.

On environments you do not have. Many teams have only test and production. Some add a staging environment with production-like data for a final check. Pipewright's promotion works the same between any two named environments; the question is how many steps your risk deserves.

On the cost of the rule. Retraining in test after changing an environment file can feel like ceremony. It is the price of being able to say, for any model in production, exactly which code trained it and which code runs around it.

Try it: promote to production

Start from the `ch11/` folder of the book's repository, or open `ch12/`, which holds this chapter's finished code. You need a second environment: build one from the starter in its own project, as you did for test. Promotion itself costs pennies; production's environment costs pennies a month while idle.

1. Describe production. Add `environments/prod.yaml` with `promotion: { from: test }` and the manifest your production environment printed, and `usecases/lapse-risk/environments/prod.yaml`. Validate both environments:

```
pipewright validate --env test
pipewright validate --env prod
```

2. Give production its data. Run the data team's jobs in the production project, then create the use case's tables there and check:

```
python tools/data_team.py --project pw-ebook-prod load
python tools/data_team.py --project pw-ebook-prod sql \
    customer_activity_backfill.sql lapse_risk_entities.sql
python tools/data_team.py --project pw-ebook-prod sql
customer_activity_month.sql \
    --as-of "2011-12-01 00:00:00"
python tools/data_team.py --project pw-ebook-prod sql
customer_purchases.sql \
    --loaded-until "2011-12-10 00:00:00"
pipewright tables sync --env prod
pipewright doctor --env prod --usecase lapse-risk
```

(`doctor` still reports the synthetic source of Chapter 6 missing in production; the agency's copy is made in test.)

3. Release, train and promote. Commit, release to test and train there, so that the model was trained by the release you promote. Then:

```
pipewright promote --from test --to prod --usecase lapse-risk --
approve
pipewright model list --env prod --usecase lapse-risk
pipewright status --env prod
```

4. Repeat it. Run the promotion again and read what it does.

5. Leave it costing nothing. Nothing is deployed. Both environments' launchers scale to zero.

In short

- Test is for trying; production is for relying. Keep them in separate projects, with a deliberate step between.
- Production needs its own data: the data team's jobs run in every environment, from the same SQL.
- `pipewright promote` copies the tested release and model: images by digest, templates compared, model files verified, the contract written once and the pointer moved last. Nothing is rebuilt.
- Promotion only comes from the named source, from the release's own commit, with a model that release trained, under the destination's rules.
- A person makes the model the champion in production; promotion is safe to repeat and rollback works there too.
- Promotion does not move data, serve or retrain; each of those is its own decision.

Next: Chapter 13. Serving, canaries and A/B tests. The champion is in production's registry. The next chapter puts it on an endpoint, lets it take a little traffic first, and explains how to tell whether a new model is better for the business, not only for the metrics.

Chapter 13. Serving, canaries and A/B tests

Putting the model to work, a little at a time

The lapse model is the champion in production's registry, and so far it has done nothing for the retailer. A model earns its keep only when its answers reach the people who act on them: here, the marketing team that runs the win-back campaigns.

There are two ways to get a model's answers to people, and the first question of this chapter is which one you need. Most teams reach for an endpoint, a model answering requests over the network, because that is what "deploying a model" sounds like. Most models do not need one.

13.1 Batch or online

Batch scoring runs the model over many rows at once, on a schedule, and writes the answers to a table. Nobody waits for it. It is cheap, because the compute exists only while the job runs, and it is easy to check, because every answer is in a table you can query.

Online serving keeps the model running behind an endpoint, so an application can ask about one customer and get an answer in milliseconds. It is the right choice when the answer is needed during an interaction: a checkout page deciding which offer to show, a call-centre screen, a fraud check before a payment. It costs money every hour it is running, whether anyone calls it or not.

The question that decides is: *when is the answer needed, and how fresh must the inputs be?*

The lapse model's answer is needed once a week. The marketing team plans each campaign from a list: the customers most at risk, with enough of their history to choose an offer. Its inputs, the data team's monthly features, change once a month. That is batch scoring, and nothing about it needs an endpoint.

The retailer does have an online need. The web team wants a checkout offer: when a customer at risk reaches the checkout, show a discount on their next order. That decision has to be made while the page loads, from what the customer has done up to today. So this chapter builds both: the weekly list first, then an endpoint for the checkout, put into service carefully, and finally fed from an online store so the checkout only has to send a customer ID.

> **Pause and think.** Think of a model you have deployed, or want to. When is its answer actually used, and by whom? If it were a table refreshed every night instead of an endpoint, what would anyone lose?

13.2 The weekly list

The scoring pipeline is the training pipeline's partner. It takes the customers to score, joins their features point in time exactly as training did (Chapter 4), fills empty values with the medians the model was trained with, scores them with the champion and writes the answers to a table. The use case names it next to training:

```
  pipelines:
    training:  { file: training_pipeline.py,  function:
churn_training_pipeline }
    inference: { file: inference_pipeline.py, function:
churn_inference_pipeline }
```

and says where the answers go:

```
# usecases/lapse-risk/config/operations.yaml
inference:
  output_table: lapse_risk_scores        # one row per customer
  decision_threshold: 0.5
```

Who is scored. The data team keeps a view of the customers to score each week: everyone active at the latest cut-off, as of that cut-off. It has no label, because nobody knows yet who will come back.

```
-- usecases/lapse-risk/sql/scoring_customers.sql
CREATE OR REPLACE VIEW `pw_retail.lapse_risk_scoring` AS
SELECT customer_id, snapshot_date, country
FROM `pw_retail.lapse_risk_entities`
WHERE snapshot_date = (SELECT MAX(snapshot_date) FROM
`pw_retail.lapse_risk_entities`)
```

Name the answer honestly. The model was trained to predict whether a customer *returns*: the label is 1 for a customer who buys again within 90 days. Its raw output is therefore the probability of returning, and the scoring step writes it under that name, `return_probability`. The marketing team thinks in risk, so the step also writes `lapse_risk`, which is one minus it. A column called `churn_probability` that actually held the chance of staying would be read the wrong way round by the first person who did not train the model. Naming is cheap; a campaign aimed at the wrong end of the list is not.

`pipewright tables sync` created the table, partitioned by the time of scoring, and the team ran the pipeline:

```
pipewright run --env test --usecase lapse-risk --pipeline inference
\
    --source-uri bq://pw-ebook.pw_retail.lapse_risk_scoring
```

It took four and a half minutes and scored 4,298 customers:

Lapse risk	Customers
0.8 and over	735
0.6 to 0.8	1,674
0.4 to 0.6	778
under 0.4	1,111

Every row records the model version that produced it, the run and the time:

```
customer_id  return_probability  predicted_return  lapse_risk
model_version_id  run_id
15098        0.006              0                 0.994       9
r9d3d079702c4
```

That last part matters more than it looks. When a customer asks why they received a win-back offer, or a manager asks why last month's list was different, the answer starts with which model scored them. Chapter 17 follows that thread all the way back to the data.

The 735 customers at 0.8 and over are the campaign's first list. The decision threshold of 0.5 only decides the `predicted_return` column; the marketing team ranks by risk and chooses how far down the list its budget goes.

In real use, the pipeline runs on a schedule rather than by hand: once a week, after the data team's jobs. Chapter 15 sets that up, along with the retraining around it.

13.3 An endpoint

For the checkout offer, the model goes behind an endpoint. The use case says how it is served:

```
  model:
    name: lapse-risk-classifier
    serving:
      enabled: true
      machine: n1-standard-2
      replicas: { min: 1, max: 2 }
      canary_percent: 10                # a new version first takes
a tenth of the traffic
```

Two things have to be true before that is allowed. The platform team must have switched on online serving for the retail tenant in each environment's Terraform (one setting, `online_serving = true`, applied by the platform team); until it did, `pipewright validate` refused the file with *tenant 'retail' is not enabled for online serving in 'test'*. And the model being deployed must meet today's gates: `deploy-model` checks them again, so a model refused by the gates cannot be served even by someone who approves it by hand.

Then one command:

```
pipewright deploy-model --env test --usecase lapse-risk --approve
```

```
model lapse-risk-classifier default version v8 → endpoint lapse-
risk-endpoint (n1-standard-2, replicas 1-2)
evaluation gates: pass
deploying lapse-risk-classifier v8 to lapse-risk-endpoint on n1-
standard-2 with 100% of traffic
live: lapse-risk-endpoint serves lapse-risk-classifier v8
```

The first deployment took 24 minutes, most of it Google Cloud starting a machine and loading the model. Later changes to the same endpoint are quicker.

What it costs. With `min: 1`, one machine runs all the time, answering or not. An n1-standard-2 is small, but it is billed by the hour, around the clock, for as long as the model is deployed. Over a month that is a real line on the bill for a test environment that nobody is calling at night. Batch scoring costs nothing between runs. This is the practical reason to choose batch whenever the need allows, and to undeploy test endpoints when you finish with them.

Since Pipewright 1.2.0. Two things change here. `deploy-model` ends by saying how to call the model: the endpoint ID, the predict URL, the permission a caller needs and a sample request with the features in the model's order (`pipewright model endpoint` prints the same later). And an endpoint may scale to zero when nobody calls it, with `replicas: { min: 0 }` and `scale_to_zero: true` under `model.serving`: it then costs nothing at night, and the first call after a quiet spell waits while a machine starts (on 1.2.0, the first prediction came back in about six seconds). Repeating a deploy is safe too: it says the version is already served and does nothing.

On Google Cloud: Agent Platform endpoints. An endpoint is a stable address that serves online predictions from one or more deployed model versions, each on machines kept running for it, with a share of the traffic each.

Why it matters. Some answers are needed in a request, not overnight. An endpoint keeps a model ready to answer in milliseconds, and splitting traffic lets a new version prove itself on a little of it first.

In Pipewright. `pipewright model deploy` deploys a registered version, and a canary sends it a share of the traffic. On private GKE, KServe serves instead (Appendix C).

Read more: https://docs.cloud.google.com/gemini-enterprise-agent-platform/machine-learning/predictions/get-online-predictions

13.4 Calling it

An application calls the endpoint over REST, with the model's features in the order the model was trained on. That order is recorded with the model, in its `features.json`:

```
["recency_days", "orders_12m", "spend_12m_gbp", "distinct_products",
"avg_days_between_orders"]
```

```
curl -X POST -H "Authorization: Bearer $(gcloud auth print-access-
token)" \
  -H "Content-Type: application/json" \
  https://europe-west2-aiplatform.googleapis.com/v1/projects/pw-
ebook/locations/europe-west2/endpoints/ENDPOINT_ID:predict \
  -d '{"instances": [[12, 9, 640.5, 48, 31.2], [210, 1, 35.0, 4,
47.5]]}'
```

```
{
  "predictions": [0.7411, 0.1426],
  "modelDisplayName": "lapse-risk-classifier",
  "modelVersionId": "8"
}
```

The first customer ordered 12 days ago, nine times in the year, every month or so: a 74% chance of coming back. The second ordered once, 210 days ago: 14%. The answer, like the weekly list's rows, names the version that gave it.

The empty value. Remember from Chapter 11 that `avg_days_between_orders` is empty for a customer with one order: 38.5% of them. Batch scoring fills an empty value with the training median, 47.5 days, before the model sees it. The endpoint does not. Send it an empty value and it fails:

```
{"instances": [[210, 1, 35.0, 4, null]]}
```

```
{"code": 503, "type": "InternalServerException", "message":
"Prediction failed"}
```

This is the most common way online models go wrong, and it has a name: **training/serving skew**. The model sees one thing in training and batch scoring, and another thing online, because the preparation around the model was not the same in both places. Here it fails loudly, which is the better case. The worse case is a caller who "fixes" the error by sending zero instead: a customer with no rhythm then looks like one who orders every day, and the model answers with confidence, wrongly, and nobody notices.

So while callers send the features, a caller must prepare each row exactly as batch scoring does: the features in the model's order, empty values filled with the model's own training medians. (Since Pipewright 1.2.0 the endpoint does that filling itself, for a model trained with the 1.2.0 example's components: training packages the medians with the model, and an empty value is filled as batch scoring fills it. `model.serving.fill_missing: false` keeps the refusal, for a team that would rather be told.) Section 13.6 shows a caller that does, and section 13.7 takes the job away from callers altogether.

> **Pause and think.** Pick one input to a model you know. Follow it from the raw data to the model in training, then from the raw data to the model in production. Are the two paths the same code? If not, who would notice when they drift apart?

13.5 A canary

Version 9 is now the champion in test. Replacing version 8 on the endpoint all at once would be quick, and if anything about version 9 were wrong online, every checkout would see it at once. A **canary** limits the damage: the new version takes a small share of the traffic first, while the old one keeps the rest, and someone watches before going further (Figure 13.1).

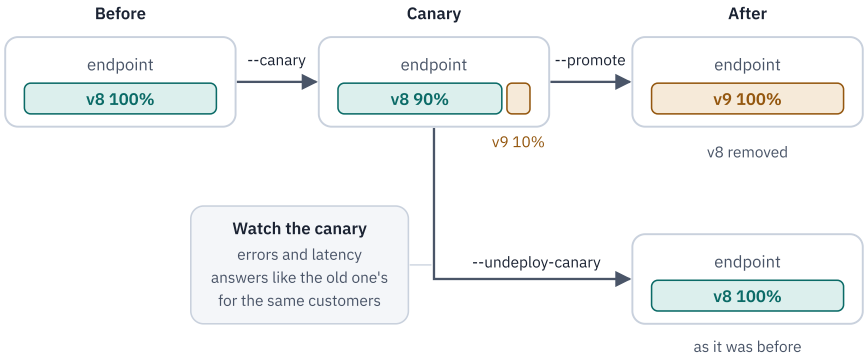


Figure 13.1 A canary. The new version joins the endpoint with a tenth of the traffic while the current one keeps the rest. If it behaves, promote moves all traffic to it and removes the old one; if not, undeploying the canary leaves the endpoint as it was.

```
pipewright model approve --env test --usecase lapse-risk --version 9
--approve
pipewright deploy-model --env test --usecase lapse-risk --canary --
approve
```

```
model lapse-risk-classifier default version v9 → endpoint lapse-
risk-endpoint; replaces v8
evaluation gates: pass
deploying lapse-risk-classifier v9 to lapse-risk-endpoint on n1-
standard-2 with 10% of traffic
canary: v9 serves 10%, v8 the rest. Watch it, then `pipewright
deploy-model --promote` (or --undeploy-canary).
```

The team sent the endpoint a hundred requests. Ninety were answered by version 8 and ten by version 9, each answer naming its version. (They gave the same probabilities: version 9 is version 8 retrained on the same data by the chapter's release, as in Chapter 12.)

What to watch. A canary is a safety check, not an experiment. It asks one question: *does the new version work online?* The signals are operational: errors, latency, and whether its answers look like the old version's for the same kind of customer. Ten per cent of a few thousand checkouts a day is enough to see a version that crashes, answers slowly or scores everyone at 0.99. It is not enough to tell whether the new version wins back more customers; that is a different question, and the next section is about it.

Then one of two commands:

```
pipewright deploy-model --env test --usecase lapse-risk --promote --
approve        # all traffic to v9
pipewright deploy-model --env test --usecase lapse-risk --undeploy-
canary --approve  # give up on v9
```

The team promoted. It took seven seconds: the traffic moved to version 9 and version 8 was taken off the endpoint.

```
undeploying 4197700099360423936 (v8)
live: lapse-risk-endpoint serves lapse-risk-classifier v9
```

13.6 Production

Production's rules, set in Chapter 12, say that a person deploys. Without the approval, the command refuses:

```
error: approvals.deploy_model is manual in prod: re-run with --
approve
```

With it, production's endpoint started serving its champion, version 1, the model promoted from test:

```
pipewright deploy-model --env prod --usecase lapse-risk --approve
```

```
deploying lapse-risk-classifier v1 to lapse-risk-endpoint on n1-
standard-2 with 100% of traffic
live: lapse-risk-endpoint serves lapse-risk-classifier v1
```

In a real organisation the approval is not the data scientist's own flag. Chapter 16 puts it in front of the people accountable for production, in the pull request and the CI pipeline.

A careful caller. The chapter's code includes a small caller, `tools/score_customer.py`, that does what the checkout's code would have to do. It reads the customer's latest features from the data team's table, reads the order of the features and the training medians from the files of the model the endpoint is serving, fills what is empty, and calls the endpoint over REST. Against production, for a regular customer and for a customer with one order:

```
customer 12347, features as of 2011-12-01:
  recency_days                    30.0
  orders_12m                       6.0
  spend_12m_gbp                 4796.97
  distinct_products              100.0
  avg_days_between_orders         65.6
return probability 0.707, lapse risk 0.293 (model version 1)

customer 12350, features as of 2011-12-01:
  recency_days                   301.0
  orders_12m                       1.0
  spend_12m_gbp                  334.4
  distinct_products               17.0
  avg_days_between_orders         47.5  (empty: the training median)
return probability 0.168, lapse risk 0.832 (model version 1)
```

The important line is the one that reads the medians *from the model being served*. Medians typed into the application's code would be right for this version and silently wrong for the next.

It works, and it is still the wrong design for a checkout. The caller queries BigQuery for every customer, which takes a second or more, too slow while a page loads. And it puts a copy of the model's preparation, the feature order and the filling of empty values, in someone else's code, where it will drift. The next section removes both.

13.7 The checkout offer: features from an online store

Chapter 4 promised the cure for this: an **online store** that keeps the latest value of each feature, synced from the same feature table training reads, so the endpoint can look a customer up by ID in milliseconds. The checkout then sends only the customer ID, and the model's inputs are prepared in one place.

> **On Google Cloud: Feature Store online serving.** An online store keeps the latest value of each feature, synced from BigQuery, and answers a lookup by entity ID in milliseconds. Pipewright's online stores use the Bigtable serving type.
>
> *Why it matters.* A caller at a checkout knows the customer, not their 90-day spend. Looking features up at request time keeps the caller simple and serves the same definitions training used.
>
> *In Pipewright.* The platform team creates the store; a use case names an online view of its feature group and sets `features: online`, and the endpoint reads the store on each request (Chapter 19 measures the cost).
>
> Read more: https://docs.cloud.google.com/gemini-enterprise-agent-platform/machine-learning/featurestore/latest/online-serving-types

It takes three owners, each doing their part.

The data team registers its features. Agent Platform's Feature Store keeps a registry of feature tables, called **feature groups** (Chapter 11's table). The data team registers `customer_activity` as one, with a description of each feature, from a file in the repository:

```
python tools/data_team.py --project pw-ebook feature-group
customer_activity
```

```
feature group customer_activity: created over
pw_retail.customer_activity
  5 features: recency_days, orders_12m, spend_12m_gbp,
distinct_products, avg_days_between_orders
```

The platform team creates the store. An online store is infrastructure with a standing cost, so it belongs in the platform's Terraform. The retailer's platform team added one for the retail tenant, with a **feature view** that serves the five features from the data team's group and syncs the latest values every morning:

```
tenants = {
  retail = {
    online_serving  = true
    online_features = {
      views = {
        customer_activity = {
          feature_groups = {
            customer_activity = ["recency_days", "orders_12m",
"spend_12m_gbp", "distinct_products",
                                 "avg_days_between_orders"]
          }
        }
      }
    }
  }
}
```

The apply created the store, its view and a grant that lets the store's sync read the tenant's dataset, and wrote the store's name into the environment manifest. (The view's first sync, started the moment it was created, failed: the grant had not yet reached every part of Google Cloud. A sync a minute later finished in 16 seconds.)

The ML team switches the endpoint to online features. Two lines in the use case: which view serves the features training read, and that the endpoint takes customer IDs:

```
# config/data.yaml
      - table: bq://pw_retail.customer_activity
        features: [recency_days, orders_12m, spend_12m_gbp,
distinct_products, avg_days_between_orders]
        online_view: customer_activity     # the same features,
latest values, in the online store

# usecase.yaml
    serving:
      enabled: true
      features: online                     # callers send a customer
ID; the endpoint reads the online store
```

`pipewright doctor` checks the store and the view's last sync:

```
Online features of lapse-risk
  OK    online store: pw_retail_online (Bigtable)
  OK    view customer_activity: last sync 2026-10-30T21:09 (0 h
ago); 5 features
```

How the model reads the store is packaged with the model when it is trained, so the team released and trained once more. Version 10 is the same model as version 9, now carrying its online lookup. Deployed, it answers a customer ID:

```
{"instances": ["12347", "12348", "12350"]}
```

```
{
  "predictions": [
    0.7065,
    0.5140,
    {"error": "the online store has no value of
avg_days_between_orders for this entity"}
  ],
  "modelVersionId": "10"
}
```

Three things to read in that answer.

The same answer as before. Customer 12347 scores 0.707, as when the careful caller sent the features by hand: the online store holds the same latest values as the feature table, because it is synced from it. That is the whole point: one definition, read in training, in batch and online.

It is fast enough. Repeated calls took about 0.3 seconds from a laptop in the UK, round trip, network included. The checkout's own servers, in the same Google Cloud region, would see less. A BigQuery query per customer would have taken several times that.

One customer was refused. Customer 12350 has one order, so their rhythm is empty, and the endpoint will not guess. It answers the others and says exactly what is missing for this one, without leaking the customer's values into a log. This is the same gap as section 13.4's 503, now polite: batch scoring fills an empty rhythm with the training median, and the online endpoint does not. For the retailer, more than a third of customers have one order, and many of those are exactly the customers the checkout offer is for.

The team had two honest choices. It could leave one-order customers out of the offer until the endpoint fills empty values as batch scoring does (a change Pipewright 1.2.0 makes, Chapter 19),

or it could change the data team's definition so that "empty" never reaches the model. It chose the first: an offer that is missing for some customers is a smaller harm than a feature whose meaning differs between training and serving. Whichever you choose, choose it knowingly: the worst outcome is not a refused request, it is a silently wrong one.

An unknown customer ID gets an answer too, not a crash: `the online store has no entity with this ID`. A new customer, signed up this morning, is not in the store until the next sync, and the checkout must treat "no answer" as "no offer", not as an error page.

Production, and a lesson. Production got its own online store, from the same Terraform, and the data team registered its feature group there too. The release and version 10 were promoted as in Chapter 12; version 10 became production's version 2, a person approved it and deployed it:

```
model lapse-risk-classifier v10 (test) → v2 (prod), a candidate
lapse-risk-classifier: default v1 → v2 (approved by ...)
live: lapse-risk-endpoint serves lapse-risk-classifier v2
```

and every request failed:

```
{"predictions": [{"error": "the online store answered HTTP 403 (view
customer_activity)"}, ...]}
```

A 403 means "not allowed". Production's endpoint was asking to read a store it had no right to read, and the reason was in the model's own files. When version 10 was trained in test, the training step wrote down how the model reads its features, including *which store*: test's. Promotion copied the model exactly, as it should, and the address of test's store came with it. Production's endpoint was trying to read test's features.

It failed safely, because production's identity cannot read anything in the test project. Had it been allowed, the endpoint would have answered every request, with test's feature values, and nothing would have looked wrong. This is a gap in Pipewright 1.1.0, recorded for its next version: the store an endpoint reads will come from the environment it is deployed in, never from the files of the model. Pipewright 1.2.0 does exactly that: promotion gives the model the destination's store, and `deploy-model` and `doctor` refuse a model that would read another environment's. Chapter 19 shows it working in production.

The general lesson is worth more than the bug. **Anything that names an environment, a project, a bucket, a store, must be supplied by the environment at deploy time, never baked into the artefact you promote.** Promotion's whole promise, Chapter 12's "exactly what was tested", depends on the artefact being the same everywhere; the only things that may differ are the addresses around it. When you build your own serving, ask of every file you package: would this be true in production too?

Production's checkout offer waits for that fix. Until then, production scores its customers in batch every week, which is what the marketing team needs, and the endpoint stays empty.

The online store has a standing cost. Bigtable is billed every hour the store exists, with at least one node, whether anyone calls it or not. It cannot scale to zero. For the book, the store was destroyed at the end of the day, by taking the `online_features` block out of the Terraform variables and applying the plan (since 1.2.0 the starter has that block itself, `tenants.<team>.online_features`); it returns in Chapter 19, when the checkout is built. A real checkout keeps it, and its cost belongs in the business case for the offer.

13.8 A/B tests: better for the business

A canary tells you the new model works. It cannot tell you the new model is *better*, and better is what the business pays for. Offline metrics, the AUC and calibration of Chapter 10, measure how well a model ranks customers on historical data. The marketing team cares about something else: how many customers the campaign brings back, and at what cost. Those can disagree. A model that ranks slightly better might put its extra accuracy among customers who were coming back anyway, and win nobody new.

An **A/B test** answers the business question directly. Customers are split at random into groups, each group gets the decisions of one model, and the team compares what happens to them. Five rules make the comparison honest.

One question, decided in advance. For the lapse model: "Does a campaign aimed by version B bring back more customers per pound than one aimed by version A?" Write down the metric, here customers who buy again within 90 days per pound of offer, before the test starts. A metric chosen after seeing the results will always find a winner.

Guardrails. Metrics that must not get worse even if the main one improves: unsubscribes, complaints, the discount spent per customer won back. A model that wins back more customers by sending every offer twice has not won.

Sticky, random assignment. Each customer belongs to one group for the whole test, decided by a hash of their ID, not by who reaches the checkout first or which region they live in. Otherwise the groups differ in ways other than the model, and the difference you measure is not the model's.

Enough customers, for long enough. Decide the size of the difference worth detecting, then work out how many customers

each group needs to see it. For a 90-day outcome, the test lasts at least 90 days after the last customer is targeted, however eager everyone is. Stopping the moment one side looks ahead is the most common way A/B tests lie.

Keep a group that gets nothing. For a win-back campaign, the most important comparison is not model A against model B but campaign against no campaign. A small held-out group that receives no offer shows how many customers come back on their own. Without it, a campaign takes credit for customers it never changed.

That last rule points at something deeper. The lapse model predicts who will *not come back*. The campaign needs to know who *would come back if offered something*. They are not the same customers. A customer at 0.99 risk may be gone whatever you send them; one at 0.6 may be exactly the person a good offer turns round. Models that predict the effect of an action, rather than the outcome, are called **uplift models**, and they are trained on the results of tests like this one. The A/B test is not only a way to choose between models: it is how a team collects the data to build a better kind of model.

A person decides the outcome. The test produces evidence; whether a small gain is worth a new model in production, given its cost and its fairness record, is a judgement for the people accountable for the campaign.

> **Pause and think.** For the model you thought about at the start of the chapter: what is the business result it is supposed to improve, in one sentence? How would you know, a year from now, whether it did?

13.9 Leaving it costing nothing

Endpoints cost money while models are deployed. When a test is over, undeploy:

```
pipewright deploy-model --env test --usecase lapse-risk --undeploy --approve
```

```
undeployed all models from lapse-risk-endpoint (the endpoint itself costs nothing)
```

The empty endpoint stays, with its address, so the next deployment reuses it and the checkout's configuration does not change. The retailer's team undeployed both endpoints at the end of the day and turned off both online stores; the weekly list needs none of them.

Thinking it through

On batch first. Every endpoint you run is a machine to pay for, monitor and keep in line with training. Start every new model as a batch job, and move to an endpoint only when someone can name the moment the answer is needed and why a table refreshed overnight is too late.

On preparation. The 503 and the refused customer in this chapter came from the same gap: the model's inputs are prepared one way in batch and another way online. The online store closed half of it, one definition of each feature read everywhere; the empty value is the half that remains. The most reliable fix is to have only one preparation, packaged with the model and used in training, in batch and online alike. Where in your system is input preparation written twice?

On what the model predicts. The lapse model ranks customers by risk. The campaign wants to know whom an offer will change. Before the next model, ask whether the question it answers is the question the business is asking.

Try it: score in batch, serve with a canary, then by customer ID

Start from the `ch12/` folder of the book's repository, or open `ch13/`, which holds this chapter's finished code. Batch scoring costs pennies. An endpoint and an online store are billed for every hour they run, so finish step 6 in the same sitting.

1. Ask the platform team for online serving. In the starter, set `online_serving = true` for your tenant in each environment's Terraform variables, plan, review and apply. Add the inference pipeline and its output table to the use case, and the `serving` block with `features: request` for now, and check:

```
pipewright validate --env test
pipewright tables sync --env test
```

2. Score in batch. Create the scoring view, release, train once so the champion belongs to this release, then score:

```
python tools/data_team.py --project pw-ebook sql usecases/lapse-
risk/sql/scoring_customers.sql
pipewright release --env test --usecase lapse-risk
pipewright run --env test --usecase lapse-risk \
    --source-uri bq://pw-ebook.pw_retail.lapse_risk_entities --run-
key serve
pipewright run --env test --usecase lapse-risk --pipeline inference
\
    --source-uri bq://pw-ebook.pw_retail.lapse_risk_scoring
```

Query `lapse_risk_scores`. How many customers are at 0.8 and over? Which model version scored them?

3. Deploy and call. Deploy the champion, then score a regular customer and one with a single order:

```
pipewright deploy-model --env test --usecase lapse-risk --approve
python tools/score_customer.py --project pw-ebook --customer 12347
python tools/score_customer.py --project pw-ebook --customer 12350
```

Try the `curl` call of section 13.4 with `null` in place of the last value, and read the error.

4. A canary. Approve the previous version as the champion and deploy, then approve the newer one and deploy it as a canary. Call the endpoint a hundred times and count `modelVersionId` in the answers. Then promote:

```
pipewright deploy-model --env test --usecase lapse-risk --canary --
approve
pipewright deploy-model --env test --usecase lapse-risk --promote --
approve
```

5. Serve by customer ID. Register the feature group, add the `online_features` block of section 13.7 to the starter's variables (the book's repository has the Terraform file the starter needs, `platform/online_features.tf`), plan, review and apply. Then set `online_view` and `features: online`, release, train, deploy and call with IDs:

```
python tools/data_team.py --project pw-ebook feature-group
customer_activity
pipewright doctor --env test --usecase lapse-risk
pipewright release --env test --usecase lapse-risk
pipewright run --env test --usecase lapse-risk \
    --source-uri bq://pw-ebook.pw_retail.lapse_risk_entities --run-
key online
pipewright deploy-model --env test --usecase lapse-risk --approve
```

Send `{"instances": ["12347", "12350"]}` and read both answers.

6. Leave it costing nothing. Undeploy, then take the `online_features` block out of the variables, plan and apply: the plan destroys the store, its view and its grant.

```
pipewright deploy-model --env test --usecase lapse-risk --undeploy -
-approve
```

In short

- Choose batch or online by when the answer is needed. Most models are batch: a table, refreshed on a schedule, cheap and easy to check.
- The scoring pipeline joins features point in time, fills empty values as training did, and writes every answer with the model version that gave it. Name the score for what it is.
- An endpoint costs money every hour a model is on it. `deploy-model` checks the gates again, and production needs a person's approval.
- Training/serving skew is the classic online failure: the input prepared one way in training and another way online. Read the model's own feature order and medians from its files; never type them into the application.

- A canary asks whether the new version works online: a small share of traffic, watched, then promoted or abandoned.
- An online store serves the feature table's latest values by ID, so the caller sends only a customer ID and the model's inputs are defined once. It has owners on three teams, and a cost every hour it exists.
- Never bake an environment's addresses into a model you promote: the environment supplies them when it deploys.
- An A/B test asks whether it is better for the business: one metric decided in advance, guardrails, sticky random groups, enough time, and a group that gets nothing. A person decides.
- Undeploy when you are done.

Next: Chapter 14. Monitoring drift and performance. The model is scoring every week. The next chapter watches its inputs and its answers, and notices when the customers it scores no longer look like the ones it learned from.

Chapter 14. Monitoring drift and performance

Noticing when the world, or your data, moves

The lapse model scores the retailer's customers every week. Nobody looks at every score, and nobody should have to. But a model is a bet that the customers it scores tomorrow behave like the customers it learned from, and that bet can go bad quietly. Nothing crashes. The scores keep arriving, a little more wrong each month, and the first sign is a campaign that stops working.

Monitoring is how you find out first. This chapter switches it on, replays seven months of it in an afternoon, and follows the first alert to where it leads, which turns out not to be where anyone expected.

14.1 What can go wrong after deployment

Three different things, each with its own signal.

The inputs move. The customers being scored stop looking like the ones in training: a new market opens, a promotion changes how often people buy, a data feed starts sending zeros. This is **feature drift**, and it can be measured the moment the scores are made, by comparing each feature's distribution with training's.

The answers move. The model's scores themselves shift: suddenly everyone looks at risk, or nobody does. This is **prediction drift**. Like feature drift, it is measurable at once.

The answers stop being right. The relationship between the inputs and the outcome changes: the same recency that meant "coming back" last year means "gone" this year. This is the one that matters

most, and it is the slowest to see, because you need the outcomes, and for the lapse model an outcome takes 90 days to arrive. This is **performance monitoring**.

Drift is an early warning; performance is the verdict. A good monitoring setup has both: drift to notice that something changed, performance to know whether it hurt.

14.2 Measuring drift

The common measure of drift, and Pipewright's, is the **population stability index (PSI)**. It splits a feature into bins using training's distribution, then compares the share of rows in each bin now with the share in training. Zero means identical. A rough, widely used reading:

PSI	Reading
below 0.1	no meaningful change
0.1 to 0.2	a moderate shift: worth a look
above 0.2	a significant shift: investigate

The training step already stores what PSI needs: each feature's distribution in training, and the distribution of the model's scores, in `training_stats.json` next to the model. So monitoring is a few lines in the use case:

```
# usecases/lapse-risk/config/operations.yaml
monitoring:
  output_table: lapse_risk_monitoring    # one row per scoring run
  drift: { enabled: true, method: psi, threshold: 0.2 }   # each
feature against its training distribution
  prediction_drift: { enabled: true, threshold: 0.2 }     # the
scores themselves against training's
  alerts:
    notify: [ml-team@example.com]        # your team's address; a
distribution list works
    events: [monitoring.alert]
    severity: WARNING
```

Every scoring run now ends with a monitoring step. It computes the PSI of each feature and of the scores, writes one row to `lapse_risk_monitoring`, and raises a `monitoring.alert` event when anything crosses its threshold.

Alerts by email. `pipewright alerts sync` turns the `alerts` block into a Cloud Monitoring alert policy that emails the addresses listed whenever the event is raised:

```
Alerts for lapse-risk (test)
  notify: ml-team@example.com
  DONE  email channel ml-team@example.com: created
  DONE  monitoring.alert: created
```

Put your team's address, or a distribution list, in `notify`. The book's repository uses `ml-team@example.com`, an address reserved for examples that never delivers mail; with it, an alert still fires and still shows in Cloud Monitoring's incidents, but nobody is emailed. The retailer's team tested the email path with a real address first, saw the alert arrive, and then put the list's address in.

14.3 Seven months in an afternoon

A monitoring setup you have never seen fire is a monitoring setup you do not understand. The team wanted to know what it would have said had it been running all year, so it replayed history: one scoring batch per month, from June to December 2011, each the customers active at that month's cut-off with their features as of then. The data team made them with one SQL script (`sql/backtest_scoring.sql`), and the team scored each:

```
for m in 06 07 08 09 10 11 12; do
  pipewright run --env test --usecase lapse-risk --pipeline
inference \
      --source-uri bq://pw-
ebook.pw_retail.lapse_risk_scoring_2011_$m --run-key backtest-
2011-$m
done
```

Seven runs, in parallel, nine minutes, pennies. The monitoring table told this story (Figure 14.1):

Scored for	Customers	Recency PSI	Largest other feature PSI	Prediction PSI	Alert
June 2011	4,338	0.196	0.004	0.028	no
July	4,326	0.169	0.004	0.034	no
August	4,337	**0.260**	0.006	0.046	**drift**
September	4,335	**0.228**	0.008	0.064	**drift**
October	4,370	**0.202**	0.018	0.062	**drift**
November	4,349	0.125	0.022	0.037	no
December	4,298	0.129	0.013	0.027	no

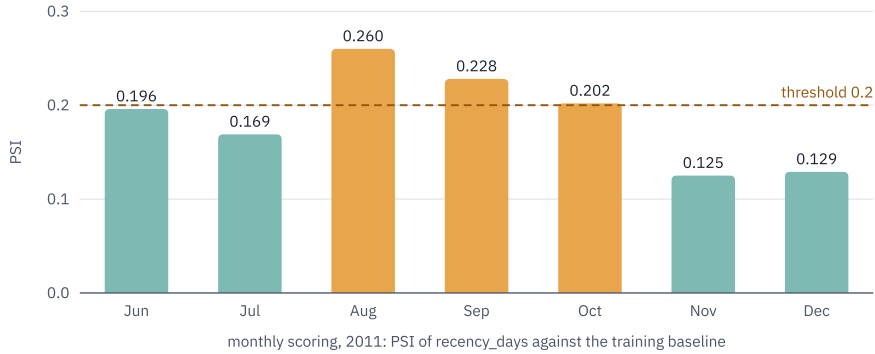


Figure 14.1 The PSI of recency_days in each month's scoring, against the threshold of 0.2. Every other feature stayed below 0.03. Recency was above the threshold from August to October and close to it all year.

One feature moved; four did not. Recency, the days since a customer's last order, was close to the threshold all year and crossed it from August to October. Orders, spend, products and rhythm hardly moved at all. The scores themselves moved little: prediction PSI never passed 0.07.

Three alerts, and the emails arrived. Now what?

> **Pause and think.** Before you read on: one feature drifting, the others steady and the model's scores barely changing. What would you check first? What would make you retrain, and what would make you do nothing?

14.4 Following an alert

An alert is a question, not an answer. The first thing to look at is what the feature actually did. The team looked at recency at every cut-off, training's and scoring's alike:

Cut-off	Median recency (days)	Longest recency (days)	Share over 180 days
June 2010	46	181	0.5%
September 2010	72	273	15%
December 2010	47	364	18%
March 2011	105	364	22%
June 2011	109	364	38%
September 2011	111	364	36%
December 2011	48	364	20%

Two patterns are in that table, and only one of them is about customers.

The season. Recency is short in December, when everyone is buying gifts, and long by late summer, when the occasional customers have gone quiet. That is real behaviour, and a model should see it. It explains why recency drifts most in August to October and least in December.

The data. Look at the longest recency. In June 2010 no customer could be more than 181 days from their last order, because the transactions start on 1 December 2009: there were only 182 days of history to look back over. Every feature claims to describe “the year before the cut-off”, and for every cut-off before December 2010, it did not. The first six months of training rows had a truncated year.

That is the real finding. The baseline monitoring compares against came from training, and training was full of rows whose recency

could not be long. So ordinary customers in 2011, with ordinary long gaps, looked unusual. The alert was right that something was wrong; what was wrong was the training data, not the world.

Nothing had checked for it. The contract (Chapter 3) checks that each value is possible, and a recency of 46 days is perfectly possible. The point-in-time join (Chapter 4) checks that no feature comes from after the cut-off, and none did. The flaw was in what the features *could not* see: the history before the data began. It took monitoring, comparing the model's view of the world with the world, to show it.

14.5 The fix, and what it cost

The data team owns the entity table, so the fix is theirs, and it is one line: start the cut-offs in December 2010, the first with a full year of history behind it.

```
-- Cut-offs start in December 2010, the first with a full year of
transactions behind it (they start on
-- 2009-12-01). Earlier cut-offs had features over less than the 365
days they claim.
  FROM UNNEST(GENERATE_DATE_ARRAY('2010-12-01', '2011-12-01',
INTERVAL 1 MONTH)) AS d
```

The team released and trained on the corrected data. The evaluation gates refused the result:

```
decision reject: ['gate calibration_error=0.1327 above max 0.05',
                  'champion: auc 0.7843 vs 0.7796 (champion version
10)']
```

Read both halves. The new model *ranks* customers better than the champion, on the same test rows: 0.784 against 0.780. But its

probabilities are badly calibrated, and Chapter 10's gate says a score must mean what it says.

The reason is the season again. With the truncated months gone, only ten cut-offs have both a full year of history and a closed 90-day window: December 2010 to September 2011. The time split trains on the earlier eight and tests on the latest two, August and September. And the share of customers who come back rises sharply in autumn, because the 90 days after a late-summer cut-off include Christmas:

Cut-off	Came back within 90 days
December 2010 to July 2011	33% to 39%
August 2011	43%
September 2011	50%

A model that has never seen an autumn predicts autumn's returns too low. The old training data had an autumn, in 2010, but with features that could not see a full year. One version had seasons and flawed features; the other has honest features and no full season.

The team kept version 10 as the champion. The gates did exactly their job: they stopped a model that would have under-stated every autumn customer's chance of coming back, however well it ranked them. The data fix stays, because it is right, and the real cure is time: from the next December, the retailer has a full year of complete cut-offs, and a model trained then sees every season with honest features. Until then the team knows why recency drifts, has written it down, and watches the other signals, above all performance.

14.6 Performance: the verdict

Drift says something changed. Only the outcomes say whether the model is still right. For the lapse model, an outcome is whether a scored customer bought within 90 days, so the verdict on a week's scores arrives three months later.

The data team publishes the outcomes as they become known, in a view with the same definition as the training label: a purchase at or after the cut-off and before the cut-off plus 90 days (`sql/lapse_risk_outcomes.sql`). Its window is the latest that has closed: September 2011's customers, whose 90 days ended in early December.

```
  performance:                          # once outcomes are known:
the model measured on what customers did
    enabled: true
    labels: { table: lapse_risk_outcomes, key: customer_id, label:
returned }
    metric: auc
    min: 0.7                            # below this the model no
longer ranks customers usefully
```

The team scored September 2011 again with performance on. The monitoring row:

```
scored for 2011-09   performance auc 0.7695 on 4,335 labelled
customers   (min 0.7)
```

Version 10 still ranks customers well on outcomes it has never seen. Then the team scored the next batch, December, and the monitoring row said something remarkable:

```
scored for 2011-12   performance auc 0.9554 on 4,335 labelled
customers
```

A model does not jump from 0.77 to 0.96 in a week. The number was wrong, and the reason is worth understanding, because it is the most dangerous kind of wrong: it reassures.

Pipewright 1.1.0 joins the outcomes to each customer's *latest* score. After December was scored, the latest score of every customer was December's, so September's outcomes were being compared with December's scores. And December's features were measured on 1 December, *after* the September-to-November window: a customer who came back in October has a short recency in December. The December scores already knew the outcome. That is leakage (Chapter 4), arriving by the back door of monitoring.

The cure is the same as in Chapter 4: every join on outcomes must be on the entity *and* the time. A score has to record the cut-off it was made for, and an outcome has to be matched to the score for that cut-off, never a later one. Pipewright's next version does this. Until then, the team did it itself, in two small changes it owns:

The scores record their cut-off. The scoring step is the team's own component. It now writes the entity's cut-off next to each score, in a new column `scored_for`, which `pipewright tables sync` added to the table as an additive change.

Performance is measured on customer and cut-off. A short query, `sql/performance.sql`, joins each score to the outcome of its own cut-off and computes the AUC per scoring run. Pipewright's built-in check is switched off, with a comment saying why, so nobody switches it back on by mistake.

After rescoring both months, the query gives:

Scored for	Customers	Came back	Predicted, on average	AUC
September 2011	4,335	49.5%	38.7%	0.770

September is measured on its own outcomes, and December does not appear at all: its outcomes are not known yet, which is the truth.

Since Pipewright 1.2.0. The built-in check joins on the cut-off too, when the outcomes say which cut-off they belong to:

```
    labels: { table: lapse_risk_outcomes, key: customer_id, label:
returned, time_column: scored_for }
```

The scores must carry `scored_for`, as the team's did. On 1.2.0, with the team's components brought up to date, the built-in check gave the same 0.769 on 4,334 customers as the team's own query, where the old join had given 0.955. That last condition is the catch: the join lives in the team's monitor step, copied from the example before 1.2.0, so an existing use case gets the fix only by updating that file. `pipewright validate` warns when `time_column` is set but no component reads it, and Pipewright's changelog lists which of the example's files carry which 1.2.0 fix. BigQuery ML use cases get the cut-off join from Pipewright itself.

The table holds one more finding, and it brings the chapter full circle. Version 10 predicted, on average, that 38.7% of September's customers would come back. 49.5% did. The champion under-states autumn returns too, by eleven points; its ranking is good, its probabilities are not. That is the same seasonal problem that made the gates refuse the corrected model in section 14.5, now seen in the model in use, on real outcomes. The marketing team ranks by risk, so the campaign list is still right; anyone reading the probabilities as forecasts of how many will return would be wrong. It is a known

limit of the model in use, to be stated wherever its probabilities are shown, until a full year of honest data lets the team fix it.

> **Pause and think.** How does your monitoring know which prediction an outcome belongs to? If you scored the same customer twice before the outcome arrived, which score would it judge?

Thinking it through

On thresholds. 0.2 is a convention, not a law. A feature that drifts with the seasons will cross it every year; a feature that matters a great deal to the model may deserve a lower one. Look at how much each feature matters (Chapter 10's importances) before deciding how loudly it should alarm.

On alerts nobody acts on. Three alerts that everyone agrees to ignore teach a team to ignore the fourth. When an alert is explained, write the explanation down where the next person will find it, and fix the cause if you can: here, by correcting the training data and waiting for a full season.

On history. Does every row of your training data have the full history its features claim? Think of the first months after a system went live, a market opened or a product launched: features over "the last year" quietly mean less there.

On what monitoring cannot see. Every check in this chapter compares the present with the training data. If the training data is wrong, monitoring can only tell you that the present looks different from something wrong. Here, that was enough to find the flaw. It will not always be.

Try it: switch monitoring on and replay history

Start from the `ch13/` folder of the book's repository, or open `ch14/`, which holds this chapter's finished code. It costs pennies: a release, seven short scoring runs and one training run.

1. Switch drift monitoring on. Add the `drift`, `prediction_drift` and `alerts` blocks to `config/operations.yaml`, with your own address in `notify`, then release and sync the alerts:

```
pipewright validate --env test
pipewright release --env test --usecase lapse-risk
pipewright alerts sync --env test --usecase lapse-risk --dry-run
pipewright alerts sync --env test --usecase lapse-risk
```

2. Replay seven months. Create the monthly scoring views and score each:

```
python tools/data_team.py --project pw-ebook sql usecases/lapse-
risk/sql/backtest_scoring.sql
for m in 06 07 08 09 10 11 12; do
  pipewright run --env test --usecase lapse-risk --pipeline
inference \
      --source-uri bq://pw-
ebook.pw_retail.lapse_risk_scoring_2011_$m --run-key backtest-
2011-$m
done
```

Query `lapse_risk_monitoring`, ordered by `source_uri`. Which features drifted, and in which months? Did the alert email arrive?

3. Follow the alert. Query recency by cut-off in `customer_activity`, as in section 14.4. Find the longest recency at each cut-off before December 2010 and explain it.

4. Fix the data, and read the verdict. Start the entity table's cut-offs at December 2010, rebuild it, release and train. Read why the gates refused the new model.

5. Measure performance on the right scores. Create the outcomes view, record `scored_for` in the scoring step and its schema (the `ch14/` folder has both), sync the tables, release, and score September and then December 2011:

```
python tools/data_team.py --project pw-ebook sql usecases/lapse-
risk/sql/lapse_risk_outcomes.sql
pipewright tables sync --env test
pipewright release --env test --usecase lapse-risk
for m in 09 12; do
  pipewright run --env test --usecase lapse-risk --pipeline
inference \
      --source-uri bq://pw-
ebook.pw_retail.lapse_risk_scoring_2011_$m --run-key perf-2011-$m
done
```

Then run `sql/performance.sql` in BigQuery. Which runs does it measure, and why is December missing? If you want to see the trap for yourself, switch `performance.enabled` on, release, score December again and read its monitoring row.

6. Leave it costing nothing. Nothing keeps running. The alert policy costs nothing until it fires.

In short

- After deployment, three things can go wrong: the inputs move (feature drift), the answers move (prediction drift), and the answers stop being right (performance). Watch all three.
- PSI compares each feature's distribution now with training's. Below 0.1 is stable, above 0.2 deserves a look.

- `monitoring` in the use case adds the checks to every scoring run, writes one row per run and raises an event; `pipewright alerts sync` emails it.
- Replay history before you rely on monitoring, so you know what it says and what is normal.
- An alert is a question. Here it led back to training data with less history than its features claimed.
- A fix can cost something else. The gates caught it, and the champion stayed.
- Drift is the early warning; performance, measured on real outcomes, is the verdict.
- Join outcomes to the score made for the same cut-off, on entity and time. A later score already knows the outcome, and a performance number that looks too good is leakage.

Next: Chapter 15. Retraining with a human in the loop. Monitoring says something changed. The next chapter turns an alert into a case, puts the evidence in front of a person, and lets that person decide whether to retrain.

Chapter 15. Retraining with a human in the loop

An alert opens a case; a person decides

Chapter 14 ended with a team that understood its alerts. Recency drifts every autumn, for a reason the team had traced and written down, and a model trained on the corrected data was refused because ten months of data cannot show it a whole year. All of that lived in one person's head and one chapter of a book. The next Monday, the scoring runs again, the alert fires again, and someone new has to decide what to do about it.

This chapter is about that decision: who makes it, what they see, and what happens next. The answer most teams reach for first is automatic retraining: drift detected, model retrained, new model deployed. It sounds like good engineering. It is how a team ends up with a production model that nobody chose.

15.1 Why not retrain automatically

Retraining is not a fix for drift. It is one possible response, and often not the right one.

The data may be wrong. If drift comes from a broken feed, a changed definition or a bug, retraining teaches the model the bug. Chapter 14's drift came from the training data itself; retraining on the same data would have changed nothing.

The change may be seasonal. Customers who behave differently in December will behave like that again next December. A model retrained on the season alone forgets the rest of the year.

The new model may be worse. Retraining on less data, newer data or different data produces a different model. Chapter 14's retrained model ranked customers better and gave probabilities that were badly wrong.

Someone is accountable. A model that decides who receives an offer, a price or a refusal has an owner. "The pipeline retrained it" is not an answer that owner can give anyone.

So the alert should not retrain. It should open a **case**: a record of what was seen, put in front of a person with the authority to decide, with everything they need to decide well, and with the decision recorded.

15.2 Running every week

First, the scoring has to run without anyone remembering to run it. The use case says when:

```
  triggers:
    api: true
    schedules:                              # `pipewright schedules
sync` makes each a Cloud Scheduler job
      - name: weekly-scoring
        pipeline: inference
        cron: "0 6 * * 1"                   # every Monday at 06:00,
after the data team's overnight jobs
        timezone: Europe/London
        source_uri: bq://pw_retail.lapse_risk_scoring   # this
week's customers, as of the latest cut-off
```

```
Schedules for lapse-risk (test)
  DONE  pw-lapse-risk-weekly-scoring: create 0 6 * * 1
(Europe/London)
```

Every Monday morning, Cloud Scheduler asks the launcher to run the scoring pipeline, which scores this week's customers, writes the list, and runs the monitoring of Chapter 14. The time zone matters: "06:00" in London moves by an hour twice a year against UTC, and a schedule written in UTC would run at 07:00 all summer.

> **On Google Cloud: Cloud Scheduler.** A managed cron: a job sends a request or a message on a schedule you set, such as every Monday at 06:00 in a given time zone.
>
> *Why it matters.* Scoring, monitoring and retraining checks happen on a calendar, and a calendar that lives on someone's laptop stops when the laptop does.
>
> *In Pipewright.* `pipewright schedules sync` turns a use case's `schedules` into Cloud Scheduler jobs that ask the launcher for a run; the hourly retrain tick and the weekly digest are jobs too. On private GKE they can be Kubernetes CronJobs instead.
>
> Read more: https://docs.cloud.google.com/scheduler/docs

15.3 The policy

What an alert does is a policy, and it lives next to the monitoring that raises it:

```
retraining:
  mode: propose                          # off: alerts only |
propose: a case for a person | candidate: trains, never promotes
  when: [drift]                          # what opens a case
  cooldown_days: 7                       # after an approved
retrain, alerts are recorded but open no new case
  source_uri: bq://pw_retail.lapse_risk_entities   # what a retrain
would train on; the approver may choose another
  decisions:
    notify: [ml-team@example.com]        # emailed the case and how
to decide; email never approves
    escalate_after_days: 3
    escalate_notify: [ml-leads@example.com]
  approvals:
    required: [ml-leads]                 # the approver group the
platform registered
    separate_from_requester: true        # whoever asks cannot also
approve
  snooze: { max_days: 30, reopen_when_worse_by: 0.1 }
```

Each line answers a question a team would otherwise argue about later.

Mode. Three choices. `off` raises alerts and nothing else. `propose`, the retailer's choice, opens a case for a person. `candidate` goes one step further and trains a candidate model straight away, so the person decides with the new model's results in hand; it still never becomes the champion by itself.

Who decides. `ml-leads` is an approver group the platform team registered in the environment (Chapter 3), with its members. Only a member can approve, and `separate_from_requester` means the person, or service, that asked cannot approve its own request.

What happens if nobody decides. The case is emailed when it opens. After three days without a decision it escalates to the leads'

list. An open case does not go away by being ignored.

What a decision lasts. After an approved retrain, a week of cooldown: new alerts are recorded against the case but open no new one, so the new model has time to settle. A snooze lasts at most 30 days, and ends early if the drift gets worse by more than 0.1.

`pipewright alerts sync` turns the `decisions` block into emails for every step of a case: proposed, approved, declined, snoozed, escalated, and a weekly digest. As in Chapter 14, put your team's real addresses in; the book uses `example.com` addresses, which never deliver. The retailer's team tested the path with a real address first.

15.4 A case

The Monday scoring of a drifting month ran, and the monitoring row ended differently from Chapter 14's:

```
  monitored (UTC)      model    rows  max PSI pred PSI  alert
drifted / retrain
  2026-10-30T23:15:31  v10      4337    0.260    0.046  YES
recency_days / proposed (drift)
```

A case was open, and an email said so. `pipewright retrain list` shows it, and `pipewright retrain show` gives the person deciding a **decision pack**:

```
Case rc-20261030-0a0549: lapse-risk (test), drift: open.
Champion: version 10, trained 2026-10-30.
Drifted: recency_days 0.260.
Prediction PSI 0.046.
Worst PSI, recent runs: 0.26, 0.23, 0.13, 0.23, 0.13, 0.26.
Data checks passed (4337 rows; most nulls avg_days_between_orders
35.6%).
Retraining would use: bq://pw_retail.lapse_risk_entities.
Approvers: ml-leads.

Options:
  approve   pipewright retrain approve --case rc-20261030-0a0549 --
reason ... [--source-uri <window>]
  decline   pipewright retrain decline --case rc-20261030-0a0549 --
because <reason> --reason ... [--days N]
  snooze    pipewright retrain snooze  --case rc-20261030-0a0549 --
because <reason> --reason ... --days N
  reasons   data-problem (fix the data first), seasonal, no-harm,
other
  also      recalibrate inference.decision_threshold, or roll back:
pipewright model approve <version>
```

Read it the way an approver would. It answers three questions before anyone has to ask them.

Is it real? One feature drifted, the scores barely moved, and the drift has come and gone over recent runs: 0.26, 0.23, 0.13. That is a pattern, not a sudden break.

Is it the data? The data checks passed: the batch is complete and its empty values are where they are expected.

What would retraining do? It would train on the entity table, the corrected one from Chapter 14, and the approver can point it at another window.

And it names the decline reasons a team needs most: `data-problem`, `seasonal`, `no-harm`. Two of the three described this case exactly.

> **Pause and think.** You are the approver. You have the decision pack and you have read Chapter 14. Approve, decline or snooze, and what reason would you write, knowing the next person to read it may be an auditor a year from now?

15.5 Approving, and what the gates said

The retailer's lead wanted to see for themselves whether a model on the corrected data could pass now, and approved, with a reason:

```
pipewright retrain approve --env test --usecase lapse-risk --case
rc-20261030-0a0549 \
    --reason "recency drift on the corrected entity table: see
whether a model on full-year cut-offs now passes the gates"
```

```
approve for case rc-20261030-0a0549 sent as approver group ml-leads
recorded: case rc-20261030-0a0549 is approved
training is submitted by the launcher; `pipewright retrain show`
shows the job
```

The approval was recorded with the approver's name and reason, and the launcher submitted the training run, under the release that is live, on the source the case named.

Five minutes later the training run had finished, and the evaluation step had decided:

```
decision reject: ['gate calibration_error=0.1327 above max 0.05',
                  'champion: auc 0.7843 vs 0.7796 (champion version
10)']
```

The same verdict as in Chapter 14, to the fourth decimal place: the same data and the same seed give the same model. Approving a retrain does not approve the model it produces. The gates of Chapter 10 still stand between the training run and the registry, and no person's approval goes around them. Version 10 remained the champion.

That part worked as it should. What the case said afterwards did not:

```
State: approved  mode propose
Training: retrain submitted job lapse-risk-r48159fad5c41-retrain on
bq://pw_retail.lapse_risk_entities
History:
  2026-10-30T23:15:04+00:00  proposed   pw-retail-run@...
  2026-10-30T23:16:34+00:00  approve    ... [ml-leads] recency drift
on the corrected entity table: ...
```

"Approved", "submitted", and nothing more. The case did not record that the gates had refused the result, and no email said so. An approver reading it would believe a new model was on its way. Pipewright 1.1.0 leaves that step to you: the refusal is in the training run's evaluation, its `training.evaluated` event and `pipewright model list`, which shows no new version. Its next version writes the outcome onto the case, emails it, and lets the case be approved again or declined. Until then, an approver's job does not end at "approve": check the run's verdict.

Since Pipewright 1.2.0. A refused retrain closes the loop by itself. The case records the gates' reasons, re-opens for a decision, and

emits `retrain.refused`, which the decision recipients are emailed. On 1.2.0 an approved drift case whose training failed the same calibration gate came back like this:

```
  rc-20261031-0b297b   open   drift   refused
retrain.refused: gate calibration_error=0.1070 above max 0.05
  next: pipewright retrain approve --case rc-20261031-0b297b --
source-uri ... (another window), or decline
```

The approver now sees what the gates saw, without going to look.

The cooldown. The next scoring run drifted too, as everyone now expected. It opened no new case. The launcher's log gave the reason:

```
retrain proposal for lapse-risk: {'status': 'no-case',
  'detail': 'no case: cooldown (7 days) after the retrain approved
2026-10-30T23:16:34+00:00'}
```

That is the cooldown doing its job: after an approved retrain, a week of quiet, so a new model is not judged on its first days. Here it also shows the cost of the gap above: the retrain was refused, there is no new model to protect, and the cooldown silences a week of alerts all the same. The monitoring row is less precise than the log, too; it says `proposed (drift)`, because the scoring run only knows it asked. When you need to know what happened to an alert, read the case and the launcher's log, not the row.

Declining, and snoozing. The other two answers are just as much a decision, and they are recorded the same way. Had the lead declined, the command would have carried the reason the decision pack offered:

```
pipewright retrain decline --env test --usecase lapse-risk --case
<case> --because seasonal \
    --reason "recency drifts every autumn (Chapter 14); a full year
of complete cut-offs arrives in December" --days 30
```

A declined case stays declined for its `--days`, then reopens if the drift is still there; it reopens early if the drift gets worse by more than `snooze.reopen_when_worse_by`. A snooze is the same with a shorter horizon, for "not now, ask me again". Either way, the next person to look finds not just "no retrain", but who decided, why, and until when.

Thinking it through

On the cost of a person. A case waits for someone, and that is slower than a pipeline. It is meant to be. The question is not whether a person is slower but whether this decision deserves one; for a model that decides who gets an offer, it does.

On reasons. A decision without a reason cannot be reviewed. Free text is better than nothing; a short list of reasons, like `seasonal` and `data-problem`, is better still, because a year later someone can count them.

On `candidate`. If your approvers keep asking "but would a new model be better?", switch to `candidate` mode, so every case arrives with the answer. It costs a training run per case.

Try it: a case and a decision

Start from the `ch14/` folder of the book's repository, or open `ch15/`, which holds this chapter's finished code. It costs pennies: two scoring runs and one training run. The weekly schedule costs pennies a week while it exists.

1. Schedule the scoring. Add the `schedules` block, release, and sync:

```
pipewright release --env test --usecase lapse-risk
pipewright schedules sync --env test --usecase lapse-risk --dry-run
pipewright schedules sync --env test --usecase lapse-risk
```

2. Write the policy. Add the `retraining` block, with your own addresses in `notify`, and an approver group your environment registered in `approvals.required`. Release and sync the emails:

```
pipewright release --env test --usecase lapse-risk
pipewright alerts sync --env test --usecase lapse-risk
```

3. Open a case. Score a month that drifts:

```
pipewright run --env test --usecase lapse-risk --pipeline inference
\
    --source-uri bq://pw-ebook.pw_retail.lapse_risk_scoring_2011_08
--run-key case-2011-08
pipewright retrain list --env test --usecase lapse-risk
pipewright retrain show --env test --usecase lapse-risk
```

4. Decide. Approve, decline or snooze, with a reason. Then read the case again, and its history.

5. Leave it costing nothing. Take the `schedules` block out and run `pipewright schedules sync` again: it deletes the job. Or keep it, and let the case collect a week of evidence.

In short

- Drift is not a reason to retrain by itself. The data may be wrong, the change seasonal, the new model worse.
- An alert should open a case for a person, with the evidence to decide: is it real, is it the data, what would retraining use.
- `retraining` in the use case is the policy: the mode, what opens a case, who approves, escalation, cooldown and snooze.
- The approver group is the platform's; whoever asks cannot approve.
- Every decision is recorded with a name and a reason, and the case keeps its history.
- Approving starts training under the live release; the gates still decide whether the result becomes the champion.

Next: Chapter 16. CI/CD and approvals in GitHub. So far every release and every decision has been a command someone typed. The next chapter puts them in a pipeline, in front of the right people, with production approvals where they belong.

Chapter 16. CI/CD and approvals in GitHub

The same steps, every time, by an identity everyone can see

Every release, promotion and deployment in this book so far has been a command someone typed on their own laptop, signed in as themselves, with whatever version of every tool they happened to have. That is how a team starts, and it is fine while the team is one person. It stops being fine the day two people release the same use case in the same week, or an auditor asks who put a model into production and how.

This chapter moves the work into GitHub Actions. Not because a pipeline is faster than a person, but because it is the same every time, it runs as an identity whose permissions are written down, and it leaves a record of who asked for what. It also turned out to be the chapter in which the most things went wrong on the way, each of them a thing a laptop had been quietly hiding.

16.1 What runs where

Four kinds of work, each with its own trigger:

When	What runs	Against	Changes anything?
A pull request is opened or updated	validate, unit tests, live `doctor`, dry runs of tables and alerts	test	no
A pull request is merged to `main`	tables sync, release, alerts sync	test	yes, test only
A person starts it	promote a tested release (and model)	test to production	yes, production
A person starts it	operate: run a pipeline, approve a model, deploy, roll back	the chosen environment	yes

Two principles sit under that table. **Nothing reaches production on a merge**: production only receives what was tested, by promotion (Chapter 12), when a person asks. And **the pipeline only does what a person could do with the same commands**: there is no CI-only path, so anything CI does can be reproduced, and explained, by hand.

16.2 An identity without keys

A workflow that releases needs to act in Google Cloud. The old way is a service account key, a JSON file stored as a GitHub secret. It never expires, it works from anywhere, and it is the credential most often found leaked in public repositories.

The better way is **Workload Identity Federation**. GitHub gives every workflow run a short-lived signed token saying which repository, branch and environment it is. Google Cloud is told to trust tokens from GitHub, but only from one repository, and to exchange them for a few minutes' access as one service account. No key exists to leak.

The retailer's platform team set it up in each project:

- a **workload identity pool** `github` and a **provider** that accepts GitHub's tokens only when `assertion.repository == 'fluxorsoft/pw-ebook'`;
- a **service account** `pw-github-ci`, which only that repository's workflows may act as;
- the service account added to the environment's **operators** in the starter's Terraform, which gives it exactly what a person who releases and operates gets, and nothing more;
- in production, read access to test's images, artifacts and models, because promotion copies from test.

Each GitHub environment, `test` and `prod`, holds the provider's name and the service account's address as variables, with the Pipewright version to install. None of them is a secret.

> **On Google Cloud: Workload Identity Federation.** Lets software outside Google Cloud, such as a GitHub Actions workflow, act as a service account by exchanging its own short-lived token. No key file is involved.
>
> *Why it matters.* A service account key in a CI secret is a password that never expires and can be copied. Federation removes it: the trust is between GitHub and Google Cloud, scoped to one repository.
>
> *In Pipewright.* `pipewright ci init` writes workflows that sign in this way, so releases and promotions run as the CI account and are recorded as such.
>
> Read more: https://docs.cloud.google.com/iam/docs/workload-identity-federation

16.3 Generating the workflows

Pipewright writes the workflows:

```
pipewright ci init --environments test,prod
```

```
wrote .github/actions/pipewright-setup/action.yml
wrote .github/workflows/pipewright-pr.yml
wrote .github/workflows/pipewright-release.yml
wrote .github/workflows/pipewright-operate.yml
wrote .github/workflows/pipewright-decisions.yml
wrote .github/workflows/pipewright-gate.yml
```

Generated files are a starting point you own, not a black box, and the team read them before committing. Two things did not fit the retailer.

The release workflow released to production. On every merge it ran `pipewright release --env prod`, building in production what had just been built in test. That breaks Chapter 12's promise that production receives exactly what was tested. The team removed the production job and added a small workflow of its own, `pipewright-promote.yml`, started by hand, that runs `pipewright promote --from test --to prod` in the `prod` GitHub environment. Pipewright's next version generates it. (Since Pipewright 1.2.0, `ci init` reads `promotion: { from: test }` in `environments/prod.yaml`, writes no release job for production and writes `pipewright-promote.yml` instead, which checks out the commit the tested release was built from before it promotes.)

The decisions workflow ran every fifteen minutes. It polls for pending decisions (Chapter 15) and opens approval gates for them. Every fifteen minutes, all day, is about three thousand runs a month, more than a private repository's free Actions minutes. The team left it out; `pipewright decisions pending` answers the same question when someone asks. (Since Pipewright 1.2.0 it runs hourly

in working hours by default, and `ci init --decisions-cron` sets your own.)

16.4 The first push

The team pushed the repository and the release workflow started: one job per use case, in test. Two released in under two minutes. The lapse model failed:

```
error: kfp version: `kfp==2.17.0` in requirements.txt does not match
Pipewright's compiler (kfp 2.18.0):
pin `kfp==2.18.*`
```

The runner had installed exactly `pipewright-mlops[gcp]==1.1.0`, and with it the newest compatible version of every dependency. Since the team's laptops were set up, a new minor version of the Kubeflow Pipelines SDK had come out. Pipewright refuses to compile a pipeline with a different SDK version from the one inside the component image, rightly: that mismatch is how pipelines that pass locally fail in the cloud.

The lesson is older than machine learning. **Pinning your own tool is not pinning your toolchain.** The fix is a constraints file that holds CI to the versions the releases were tested with, used by the setup step:

```
# .github/pipewright-constraints.txt
# kfp must match the minor version pinned in each use case's
components/requirements.txt.
# Change both together, in one pull request.
kfp==2.17.*
```

That fix went in as the team's first pull request, which is where the next two surprises were waiting.

Since Pipewright 1.2.0. `ci init` writes `.github/pipewright-constraints.txt` with the kfp Pipewright was tested with, and the setup step installs with it. The package itself also stops at that kfp: on the day 1.2.0 was released, kfp 2.18 came out, and a fresh install took it, exactly as the team's runner had. The release's own checks caught it before anything was published, and 1.2.0 shipped with kfp held below 2.18.

> **Pause and think.** If you set up a new laptop today and installed your team's tools from scratch, would you get the versions your last release was built with? How would you know?

16.5 A pull request

The pull request checks ran against test: validate, the use case's unit tests, a live `doctor` that compiles every pipeline, and dry runs of every change a release would make. Two of them failed.

The unit tests could not import `numpy`. The lapse model's test (Chapter 7) imports every component module, so it needs the components' own dependencies: pandas, numpy, PyTorch. On a data scientist's laptop they are installed. On a fresh runner, only Pipewright was. The team made the test step install the components' requirements first, with the same constraints file, so the test runs against what the image will contain. (Since Pipewright 1.2.0, `tests.environment: components` does this without a workflow change, as Chapter 7 shows.)

`doctor` could not read the buckets. It could not read the launcher, its subscription or its topics either:

```
  FAIL  artifacts bucket pw-ebook-pw-artifacts: no permission to
read it
  FAIL  landing bucket pw-ebook-pw-landing: no permission to read it
  FAIL  launcher service pw-launcher: no permission to read it
```

The operators' roles let them write release contracts and publish run requests, but not read a bucket's own settings or a service's description. Every person who had run `doctor` so far was the project's owner, so nobody had noticed. The platform team added three read-only roles for operators to the starter's Terraform, planned, reviewed and applied.

Then everything was green:

```
✓ check (customer-segments) in 1m4s
✓ check (lapse-risk) in 2m7s
✓ check (demand-forecast) in 58s
```

and a person reviewed the pull request and merged it. That last step is not a formality. The checks say the change is valid; only a person can say it is wanted.

Look back at what the three failures had in common. Each was something a laptop had been hiding: a newer library, a missing package, a missing permission. Each was fixed in the repository or in the platform's Terraform, where the next person will find it, not on one machine. That is most of what CI is for.

Since Pipewright 1.3.0. The pull request workflow from `ci init` gains a `review` job: it writes the review bundle (what a release would change in Google Cloud, the settings that moved, the checks and the author's judgement calls) to the run's summary, and once the repository has a context pack for coding agents, fails when it is stale (Chapter 20).

16.6 Merging releases to test

The merge started the release workflow. For each use case, in test: create or update its tables, release (build the image, compile and upload the templates, write the contract, verify, move the pointer), then sync its alerts.

```
✓ release-test (demand-forecast) in 1m38s
✓ release-test (customer-segments) in 1m44s
✓ release-test (lapse-risk) in 9m20s
live: gs://pw-ebook-pw-artifacts/contracts/lapse-risk/current.json -
> 89e75bdfad29 (was e422180bf066)
```

The lapse model takes longest because its component image, with PyTorch, is built on the runner; the two BigQuery ML use cases have no image of their own to build. Schedules were left alone: the `test` environment's variables say `PIPEWRIGHT_SCHEDULES: off`, because the book's environment does not keep the weekly job running.

One line in the log was easy to miss:

```
event release.live not published to projects/pw-ebook/topics/pw-
events: 403
```

The release succeeded, but the event announcing it was refused: operators were allowed to publish run requests, not lifecycle events. Every release made by CI would have been missing from the event log that Chapter 18 builds the audit trail from. The platform team added the permission. A lost event should be louder than a log line; that is in Pipewright's next version too. (Since Pipewright 1.2.0 a command that could not publish a lifecycle event ends with a warning naming it, and the starter grants operators the right to publish them.)

16.7 Promotion, and who approves it

Production should only ever change because a person decided it should. GitHub's way of saying that is an **environment with required reviewers**: a job that targets `prod` waits until one of the named people approves it in GitHub, and GitHub records who did and when.

On GitHub's free plan, required reviewers are only available on public repositories. The retailer's repository is private, and the platform team tried to add a reviewer:

```
Failed to create the environment protection rule. Please ensure the
billing plan supports the required reviewers
protection rule. (HTTP 422)
```

So production's gate is in two other places, both of which hold on any plan.

Who can start the promotion. Only people with write access to the repository can start a workflow by hand, and GitHub records who started each run.

Production's own policy. Production's environment says that a person makes a model the champion and a person deploys it (Chapter 12). The promote workflow only passes `--approve` when the person starting it ticks `approve_model`, and `deploy-model` in production refuses without it.

On GitHub Team or Enterprise, add required reviewers to `prod` as well; the workflow already targets that environment.

The lapse model could not be promoted. Promotion carries the model trained by the release being promoted, and every retrain on today's data is refused by the gates (Chapters 14 and 15), so the

newest release in test has no model of its own. That is the rules working, not failing. The SQL team's forecast was different: with `model.promotion: retrain`, promotion copies the release, and production trains its own forecast on its own data (Chapter 8). The production data team built `weekly_product_units` in production, and the team started the promote workflow:

```
promote demand-forecast release 89e75bdfad29: test → prod (retrain)
copying .../pw-ebook/pw-images/demand-forecast-
components@sha256:27c50f72...
  → .../pw-ebook-prod/pw-images/demand-forecast-components
live in prod: 89e75bdfad29 (first release)
  verified: contract v3 is valid
  verified: component image exists (by digest)
  verified: training template bqml-training/sha256:3fa3d375...
matches the contract
  verified: inference template bqml-scoring/sha256:6f8f189d...
matches the contract
```

It ran as production's CI account, reading test's registry and writing production's, with no key anywhere. Then the operate workflow, in production, started production's own training run:

```
action: run   environment: prod   use case: demand-forecast
pipeline: training
source: bq://pw-ebook-prod.pw_retail.weekly_product_units
```

Ten minutes later production had its own forecast, trained on its own data, under the release that was tested:

```
demand-forecast (prod)
  v1    DEFAULT    2026-10-31 00:25:15  run r8dd183e56a8a,
mae=197.5556  [gates pass]
demand-forecast (test)
  v1    DEFAULT    2026-10-30 17:13:00  run ra6179fa33d26,
mae=197.5556  [gates pass]
```

The same error to four decimal places, because production's data team loaded the same transactions as test's. In a real organisation production's data would differ, and so would its forecast; that is the reason this use case retrains rather than copies.

Notice `DEFAULT` on production's line. Production's policy says a person makes a model the champion, and nobody did: Model Registry makes a model's first version its default automatically. For a promoted model, Chapter 12's `--approve` covered that; a model production trains itself never passes through promotion's approval. Pipewright's next version holds a first version back until a person approves it. Until then, under a manual policy, look at a new model's first version yourself, and record your approval with `pipewright model approve` before anything uses it.

Since Pipewright 1.2.0. Under a manual policy a first version is labelled not approved: `model list` and `doctor` say so, and `deploy-model` refuses it until `pipewright model approve` records the approval. The label is set by the register step, so a use case whose components were copied before 1.2.0 needs the example's `register.py`; `validate` warns until it has it.

The promotion run itself ended in red, on its last step: it listed production's models, and production had none yet, because the forecast is trained there after promotion. A second pull request made that listing informational. A red run that only means "nothing to list yet" teaches people to ignore red runs.

> **Pause and think.** Who in your organisation can change what runs in production, and could you list them from a system rather than from memory? If one of them left tomorrow, what would you need to change?

Thinking it through

On generated files. `pipewright ci init` saved an afternoon and still needed reading. Treat anything generated as code someone else wrote for you: review it before it runs with your permissions.

On "works on my machine". Three of this chapter's failures came from differences between a laptop and a clean runner. A pipeline that runs from nothing every time is the cheapest way to find them, and the fix belongs in the repository, not on the runner.

On the approval you cannot buy. Required reviewers are a feature of a paid plan for private repositories. The retailer's gate still holds without them, because Pipewright's production policy refuses to promote a champion or deploy without an explicit approval. Decide which layer is your real control, and do not rely on one you do not have.

Try it: CI/CD for your repository

Start from the `ch15/` folder of the book's repository, or open `ch16/`, which holds this chapter's finished code. Make it the root of a GitHub repository of your own; GitHub Actions minutes are free up to a monthly limit on private repositories, and the runs cost pennies in Google Cloud.

1. An identity for CI. In each environment's project, create a workload identity pool and a GitHub provider limited to your repository, a service account, and the binding that lets your

repository act as it. The book's repository has the script the retailer's platform team used, `platform/github-ci.sh`. Add the service account to `operators` in the starter's variables, plan, review and apply. In production, give it read access to test's images, artifacts bucket and models.

2. GitHub environments. Create `test` and `prod` in the repository's settings, each with the variables `GCP_WIF_PROVIDER`, `PIPEWRIGHT_SERVICE_ACCOUNT`, `PIPEWRIGHT_PIP_SPEC` (`pipewright-mlops[gcp]==1.2.0`) and `PIPEWRIGHT_SCHEDULES` (`off` if you do not want CI to create schedules). On a paid plan, add required reviewers to `prod`.

3. The workflows. Generate them, then make the changes of section 16.3, or copy `ch16/.github/`:

```
pipewright ci init --environments test,prod
```

4. A pull request. Change something small, open a pull request, and read every check. Fix what fails in the repository, not on the runner.

5. Merge, then promote. Merge, watch the release, then start `pipewright-promote` for a use case whose release and model were tested in test.

In short

- CI/CD makes releases the same every time, run by an identity whose permissions are written down, with a record of who asked.
- Pull requests check against test and change nothing; merges release to test; production changes only by promotion, when a person starts it.

- Workload Identity Federation gives a workflow a few minutes' access as a service account, for one repository, with no key to leak.
- Read generated workflows before they run with your permissions. Here they would have rebuilt in production.
- Pin the toolchain, not only your tool: a constraints file holds CI to the versions releases were tested with.
- A clean runner finds what a laptop hides: missing packages, missing permissions, newer libraries.
- Required reviewers need a paid plan on private repositories; production's own approval policy holds on any plan.

Next: Chapter 17. Lineage: every model's full story. Releases, runs, models and promotions now leave records in several places. The next chapter follows one production model back through all of them, to the rows it learned from.

Chapter 17. Lineage: every model's full story

From one customer's score back to the rows it learned from, and forward to who decided

A customer writes to the retailer. They received a "we miss you" offer, and they want to know why the retailer thinks they are leaving, and what it knows about them. Around the same time, an internal auditor asks the ML team a more general version of the same question: for the model in production, show us what it is, where it came from, who approved it, and whether you could build it again.

Both questions are about **lineage**: the chain of records that connects an answer to the model that gave it, the model to the code and data that made it, and all of it to the people who decided. This chapter follows that chain for one customer, finds the link that was missing, and tests the hardest claim of all: that a model can be reproduced.

17.1 From a score to a model

The weekly list (Chapter 13) records, next to every score, the model version and the run that produced it, and since Chapter 14 the cut-off it was made for. For customer 15098:

```
scored_for   lapse_risk  model_version_id  run_id
2011-09-01   0.953       10                r887f360a7d7c
2011-08-01   0.822       10                rd0172c413ba7
```

So the answer to "why me?" starts with "version 10 of the lapse model, in the scoring run of 1 September". That is not yet an

explanation, but it is the only possible start of one: without it, every later step is a guess.

17.2 From a model to everything that made it

`pipewright lineage` follows a model version back through every record Pipewright keeps. For production's lapse model:

```
pipewright lineage --env prod --usecase lapse-risk
```

```
lapse-risk-classifier v2 in prod (default), created 2026-10-30
21:29:29
  promoted from test v10: `pipewright lineage --env test --usecase
lapse-risk --version 10` goes on
  run      r6e3c5bd055c8  (recorded 2026-10-30T21:20:41)
  release  5708ac83eaa2  commit
5708ac83eaa2bc0eba77f1ced6764e0a6b6fce4a
           template training: sha256:12341cce2d0c...
           image .../pw-ebook-prod/pw-images/lapse-risk-
components@sha256:9223f314ab9d...
  config   ml_config sha256 6c38f878ce03...
  data     bq://pw-ebook.pw_retail.lapse_risk_entities
  rows     snapshot pw-ebook.pw_retail.lapse_risk_snapshots run
r6e3c5bd055c8: 62689 rows
  sources  entity table pw-ebook.pw_retail.lapse_risk_entities
           features bq://pw_retail.customer_activity: recency_days,
orders_12m, spend_12m_gbp,
             distinct_products, avg_days_between_orders
           labels bq://pw_retail.customer_purchases, window 90 days
  eval     approve: auc 0.78617 vs champion v9 0.78617
(gs://.../evaluation.json)
```

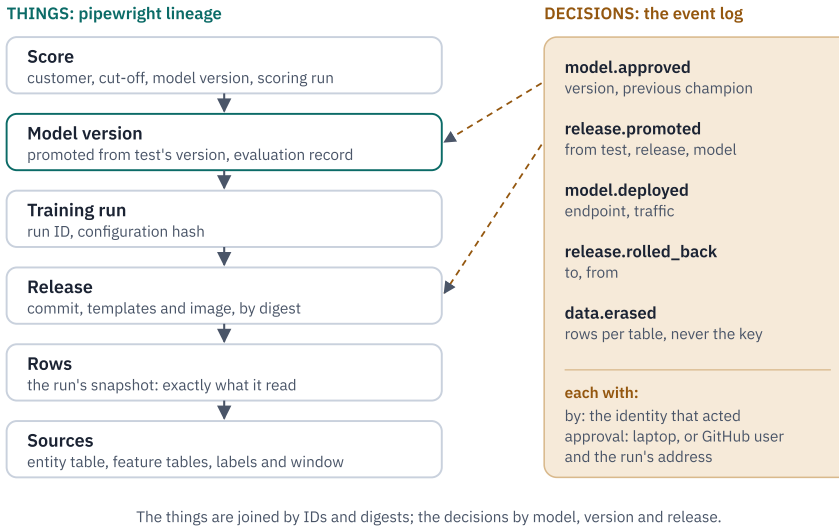


Figure 17.1 The chain behind one score: the score names the model version and run; the run names its release, whose commit, templates and image are pinned; its snapshot holds the exact rows; the rows came from the entity, feature and label tables; the evaluation record says why it passed. Decisions about the model are events, kept in the event log.

Read it from the top, the way an auditor would.

Where it came from. Production's version 2 was promoted from test's version 10, and the next command follows it there. Test's lineage shows the same run, the same commit, the same configuration hash, and the same image digest, `sha256:9223f314...`: the image in production is byte for byte the one that was tested, copied, not rebuilt (Chapter 12). The templates have different digests, because they were compiled again for production's registry and compared.

What code. The release names a commit. `git show 5708ac8` in the team's repository shows exactly what changed in it, and the release contract pins the templates and image that commit produced.

What data. The run kept its rows: 62,689 of them, in the snapshot table under its run ID (Chapter 5), with the tables they were read

from, the features taken from each and the label's window.

Why it passed. The evaluation record, with every metric, slice and fairness measure of Chapter 10, is a file next to the model.

The same command works for production's forecast, which production trained itself (Chapter 16), and leads to production's own run, release and rows.

17.3 Which training runs learned from this customer?

Lineage also runs the other way. Customer 15098's rows in the snapshot table answer the question directly:

```
SELECT run_id, COUNT(*) AS rows, MIN(snapshot_date) AS first,
MAX(snapshot_date) AS last
FROM pw_retail.lapse_risk_snapshots
WHERE customer_id = '15098'
GROUP BY run_id
```

Fifteen training runs had this customer in their data, three rows each, for the cut-offs of July, August and September 2011. One of them, `r6e3c5bd055c8`, trained the model in production. When the customer asks "what did you use about me?", the team can answer precisely: these three rows, these five features, in these runs, for this model.

> **Pause and think.** For a model you run today: if a customer asked which of your models learned from their data, how long would it take you to answer, and how sure would you be?

17.4 Who decided?

The chain above is all about things: runs, commits, images, rows. The auditor's second question is about people: who approved this model for production, who promoted it, who put it on an endpoint? Pipewright records each of those as an **event** with the decision's details and how it was approved: from a laptop, or from a GitHub Actions run, with the GitHub user and the run's address.

The team went to find those events, and found nothing. Production's logs held only what the pipeline runs had logged themselves. Chapter 12's promotion, Chapter 13's approval and deployments, and Chapter 16's CI promotion had all been published to the environment's events topic, and that was where the trail ended: nothing subscribed to the topic, and Pub/Sub discards a message nobody subscribes to. Every decision had been announced, and none of it had been kept.

The platform team added the missing piece to the starter's Terraform, ported from Pipewright Enterprise: a BigQuery dataset of evidence, an append-only `events` table, and a subscription that writes every event on the topic into it, readable by the operators. From then on, every decision was kept. The ones before it were gone; no tool can recover a message that was never stored.

The next decision was made from GitHub Actions: the team made version 9 the champion in test through the operate workflow, and then version 10 again. The event log:

```
01:01:53 model.approved  version 9, previous 10
         by: pw-github-ci@pw-ebook.iam.gserviceaccount.com
         approval: via github, actor pipewright-mlops,
                   run_url https://github.com/fluxorsoft/pw-
ebook/actions/runs/41153204666, reviewers []
01:03:17 model.approved  version 10, previous 9
         ...
```

Both halves of the answer are there. `by` is the identity that acted, the CI account. `approval` is the person behind it: the GitHub user who started the run, with a link to the run itself. `reviewers` is empty, honestly: the free plan has no required reviewers (Chapter 16). A rollback made from a laptop says `via: outside-ci`, with the person's own account.

Two parts of the story are still missing in Pipewright 1.1. `pipewright lineage` shows the things, not the decisions; joining the two is left to you, or to Fleet View in Pipewright Enterprise. And a release made by CI on a merge records the CI account, not the person who merged. Pipewright's next version adds both. Until then, the event log is the place to look, and it is only there if someone subscribed it.

Since Pipewright 1.2.0. `pipewright lineage` reads the decisions from the event log, when the environment's manifest names it (`observability.evidence_dataset`, which the starter now writes), and lists them under the version:

```
  decided  2026-10-31 00:43:56  rolled back to release 5708ac83eaa2,
by ops@example.com (outside CI)
           2026-10-31 01:03:17  v10 made the champion, by pw-github-
ci@... (GitHub, run .../actions/runs/...)
           2026-10-31 04:23:19  v10 deployed to lapse-risk-endpoint
(100% of traffic), by ops@example.com
```

A release made by CI also records the person behind the CI account on the release pointer, so `pipewright status` says who merged, not only which account released.

> **Pause and think.** Where are your decisions recorded: the approvals, deployments and rollbacks? Would the record survive the person who made it leaving, or the tool that announced it having no listener?

> **On Google Cloud: BigQuery subscriptions.** A Pub/Sub subscription that writes every message on a topic straight into a BigQuery table, with no code in between.
>
> *Why it matters.* Decisions about models (approved, promoted, rolled back) are evidence. Writing them to an append-only table as they happen gives you a record you can query, instead of one you reconstruct later.
>
> *In Pipewright.* The platform team points a BigQuery subscription at Pipewright's events topic. The result is the event log that `pipewright lineage` reads for who decided what, and when.
>
> Read more: https://docs.cloud.google.com/pubsub/docs/bigquery

17.5 Could you build it again?

The hardest claim in lineage is reproducibility: with the same data, code and image, you get the same model. The team tested it on production's model, replaying its training run on its exact rows (Chapter 5):

```
pipewright run --env test --usecase lapse-risk --replay
r6e3c5bd055c8
```

	Original run	Replay
Rows	62,689	62,689, the same rows
AUC	0.78617	0.785136
Fairness gap (limit 0.15)	inside	0.1517: refused

Close, not equal, and the difference was enough to change a gate's decision. The team suspected the image, since the replay ran under a newer release built by CI, and rolled test back to the original release to replay again. Same answer, to six digits: 0.785136. Two replays agreed with each other exactly, and both disagreed with the original.

Chapter 5 had already named the cause and left it unfixed: BigQuery returns rows in no fixed order, and the training code shuffles them, with a fixed seed, from whatever order it received. The original run read its rows from a join; the replays read them from the snapshot table, in another order. Same rows, same seed, different sequence, slightly different network.

The fix is one sort, in the team's own preparation step, by the contract's unique key:

```
    # Rows arrive from BigQuery in no fixed order, and training
shuffles them from the order it receives. Sorting by
    # the contract's unique key makes training, and so a replay,
repeat exactly.
    key = [c for c in (data["contract"].get("unique") or []) if c in
df.columns]
    if key:
        df = df.sort_values(key, kind="stable")
```

The team released it, trained a fresh run, and replayed that run:

	Run A, fresh	Run B, replay of A
AUC	0.782421	0.782421
Calibration error	0.1070	0.1070
Decision	refused: calibration	refused: calibration, the same reasons

Exactly the same. (Both were refused by the calibration gate, for Chapter 14's seasonal reason; for this test, the verdict does not matter, only that it repeats.) Since Pipewright 1.2.0 the example's prepare step carries this sort, and each run's snapshot event carries a fingerprint of its rows (Chapter 5), so "same rows" is a record, not a belief: a replay and its original show the same fingerprint. "Within a rounding" had been accepted in Chapter 5 as good enough. It stops being good enough the day a gate sits inside the rounding.

17.6 Forgetting a customer

Customer 15098 then asked the retailer to delete their data. The use case declares which column identifies a person:

```
data_protection:
  personal_data_columns: [customer_id]
  erasure: { key_column: customer_id }
```

and `pipewright data erase` finds and deletes their rows in every table the use case owns, with a dry run first:

```
Erasing 1 key(s) on customer_id for lapse-risk (test)
  DEL   lapse_risk_raw_customers: 7 row(s) deleted
  DEL   lapse_risk_snapshots: 55 row(s) deleted
  OK    lapse_risk_synthetic: 0 row(s) deleted
  DEL   lapse_risk_scores: 14 row(s) deleted
not covered: landing files and pipeline artifacts in GCS (platform
retention), and models trained on this data:
retrain to remove its influence
```

The last line is the important one, and it is honest. Erasure removes the customer's rows. It does not remove what the models already learned from them, and it does not reach the data team's tables, `customer_activity` and the transactions, which the data team must erase from in its own way.

It also changes what lineage can promise. The team replayed run A once more, after the erasure:

	Run A	Replay B, before erasure	Replay C, after erasure
Rows read	43,016	43,016	43,013
AUC	0.782421	0.782421	0.782352
Calibration error	0.1070	0.1070	0.1100

Three rows fewer, the customer's three cut-offs, and a model that differs in the fifth decimal place. The replay did exactly what it

should: it trained on what the snapshot still holds. And the snapshot no longer holds what run A trained on.

Reproducibility and erasure pull against each other, and the law decides which wins: once a person has asked to be forgotten, the snapshot no longer holds what the run trained on, and nobody can reproduce that run exactly again. Lineage records that too: the erasure is an event, with the number of rows, and the replay's own snapshot says how many rows it read.

Thinking it through

On what lineage is for. Most of the time nobody reads it. It is for the day someone asks, and on that day it has to be complete, because a chain with one missing link answers nothing. The missing link here was not in the model's records but in the platform: a topic nobody listened to.

On exactness. A replay that lands within a rounding sounds good enough until a gate, a threshold or a court sits inside the rounding. Exactness is cheap when you design for it, a sort in the right place, and very expensive to add after someone has asked.

On erasure and models. Deleting a person's rows does not delete their influence on a model already trained. Most regulators accept retraining on the next schedule as the remedy; some cases demand it at once. Know which applies to you before the first request arrives.

Try it: follow one customer

Start from the `ch16/` folder of the book's repository, or open `ch17/`, which holds this chapter's finished code. It costs pennies: three training runs.

1. From a score to a model. Pick a customer from `lapse_risk_scores` and note the model version and run of their latest score. Then:

```
pipewright lineage --env test --usecase lapse-risk --version
<version>
pipewright lineage --env prod --usecase lapse-risk
```

2. Which runs learned from them. Run the snapshot query of section 17.3 for your customer.

3. Keep the decisions. If your environment's events topic has no subscription, add the starter's `evidence.tf` (in the book's repository, `platform/`), plan, review and apply. Then make a decision and find it in `<prefix>_evidence.events`.

4. Replay, then make it exact. Replay the run behind your champion and compare the numbers. Add the sort to `components/prepare.py`, release, train, and replay that run: the numbers must match exactly.

5. Forget a customer. Add `erasure: { key_column: customer_id }`, then:

```
pipewright data erase --env test --usecase lapse-risk --key
<customer>
pipewright data erase --env test --usecase lapse-risk --key
<customer> --confirm
```

and replay the run again.

In short

- Lineage connects an answer to its model, the model to its code and data, and all of it to the people who decided.

- Every score names its model version, run and cut-off; `pipewright lineage` follows a version back to its release, commit, image, configuration, exact rows, sources and evaluation.
- The snapshot table answers the reverse question: which runs learned from a given person.
- Decisions are events. If nothing subscribes to the events topic, they are announced and lost: keep them in BigQuery.
- In CI, a decision records both the identity that acted and the person behind it, with a link to the run.
- Reproducing a model needs the same rows in the same order: sort by a key before training, and a replay is exact.
- Erasure removes a person's rows from the use case's tables, not their influence on trained models, and ends exact reproduction of the runs that used them.

Next: Chapter 18. Evidence, audit and running a fleet. One model's story is now complete. The next chapter puts the evidence of every model in one place, for the auditor who asks about all of them at once, and for the platform team that runs dozens.

Chapter 18. Evidence, audit and running a fleet

Every model, every environment, on one page

Chapter 17 answered questions about one model. The auditor's next visit is different. They do not want one model's story; they want all of them, at once, on paper: what is running where, who decided it, how each was tested, what watched it afterwards, and what it cost. The platform team, meanwhile, asks the same questions every Monday for a less formal reason: three use cases are easy to keep in your head, thirty are not.

This chapter builds the answer with what every Pipewright environment already has: the release pointers, the model registry, each use case's own tables, the pipeline jobs and, since Chapter 17, the event log. It ends with a one-page fleet report, a script in the book's repository that you can run against your own environments, and with an honest list of what an auditor will ask that the open-source platform does not answer for you.

18.1 What an auditor asks

Audits differ by industry, but the questions about models are remarkably constant. Five cover most of them.

The question	Where the answer lives
What is running, in which environment, since when?	each use case's release pointer (Chapter 7), the model registry
How was each model tested, and did it pass?	the evaluation record and the metrics table (Chapter 10)
Who decided: approved, promoted, deployed, rolled back?	the event log (Chapter 17)
What has watched it since, and what did it find?	the monitoring table and its alerts (Chapter 14)
What does it cost?	pipeline jobs, and Cloud Billing by label

Every row of that table is already recorded somewhere. The work of an audit is not producing evidence; it is gathering it from five places, quickly, and showing it in a form someone who did not build the system can read.

18.2 Evidence that is worth something

Not every record is evidence. Four properties separate the two, and the book has been building them chapter by chapter.

Written at the time, by the system. A spreadsheet filled in before the audit is a reconstruction. The release pointer, the evaluation record and the event log were written by the commands themselves, at the moment they acted.

Hard to change afterwards. Release contracts are written once (Chapter 7); the artifacts bucket keeps every version of the pointers; the event log is append-only, and only Pub/Sub writes to it. None of them is tamper-proof against the project's owner, but none of them can be quietly edited by the people whose work they record.

Joined by identifiers, not by memory. Every score names its model version; every version names its run and release; every event names its use case, environment and release. The chain of Chapter 17 holds because nobody has to remember which run went with which model.

Complete, or honestly incomplete. The decision trail before Chapter 17 was not kept. An audit pack that does not say so is worse than one that does: the gap is part of the evidence.

18.3 One page for the fleet

The book's repository has a small script, `tools/fleet_report.py`. It reads every environment in `environments/` and every use case in `usecases/`, gathers the five answers above from the records Pipewright keeps, and writes one self-contained HTML page with charts. It changes nothing, and with `--mask` it replaces every email address with `...` for sharing.

```
python tools/fleet_report.py --out fleet.html --mask
```

```
wrote fleet.html: 3 use cases in 2 environments
```

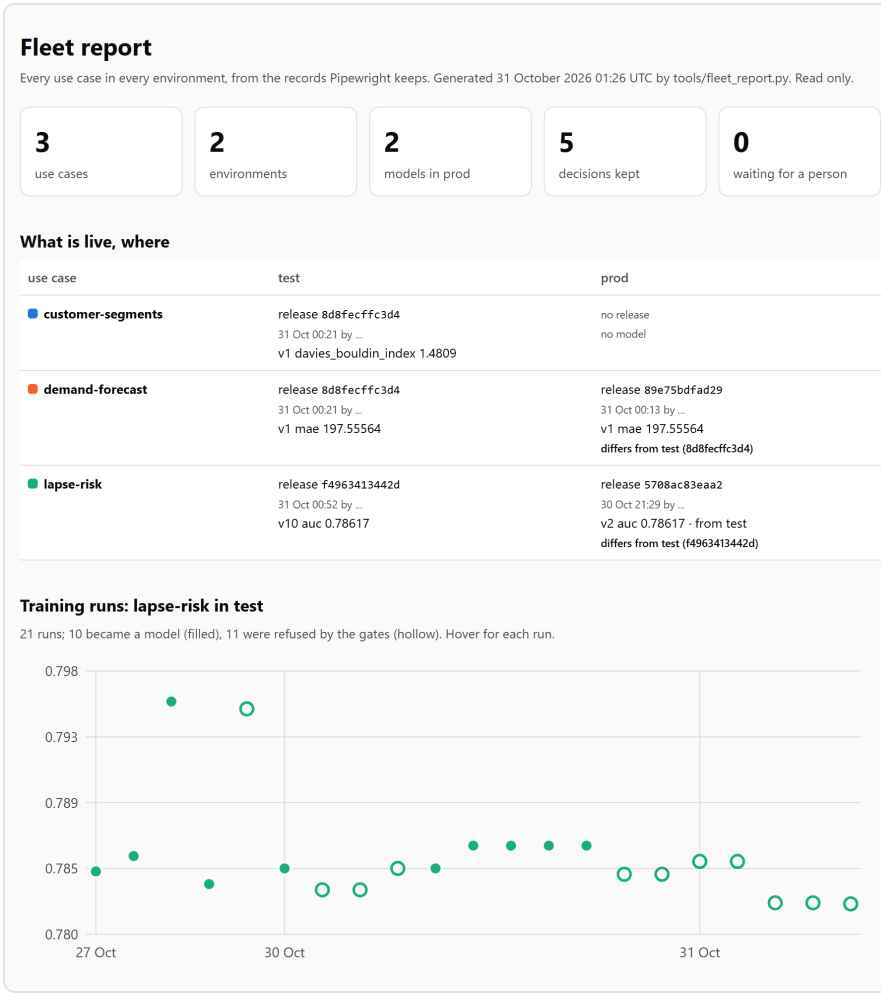


Figure 18.1 The top of the fleet report for the retailer's two environments: the count of use cases, environments, models in production, decisions kept and decisions waiting; what is live where, with each champion; and every training run of the lapse model, kept or refused. Email addresses are masked.

Read it the way the platform team does on a Monday.

The tiles. Three use cases, two environments, two models in production (the lapse model and the forecast; the segments are only in test), five decisions kept, nothing waiting for a person. The fourth

number is small for a reason the page itself explains further down: the event log only exists since Chapter 17.

What is live, where. One row per use case, one column per environment: the live release, when and by whom it was moved, and the champion with its metric. Two cells are marked in bold: production runs older releases than test, for both use cases it has. That is not a fault: production receives releases only by promotion, and nobody has promoted since. But it is exactly the thing a fleet view is for. A test environment three releases ahead of production is either a deliberate pause or a forgotten promotion, and the page makes someone ask which.

Training runs. Every training run of the lapse model, oldest first, its AUC, filled when it became a model and hollow when the gates refused it: 21 runs, 10 kept, 11 refused. The shape tells the story of the last five chapters without a word: steady runs, a better feature (Chapter 11), then a cluster of refusals as the data was corrected (Chapter 14) and replays were tested (Chapter 17). A model whose runs are all filled is either very stable or very unguarded.

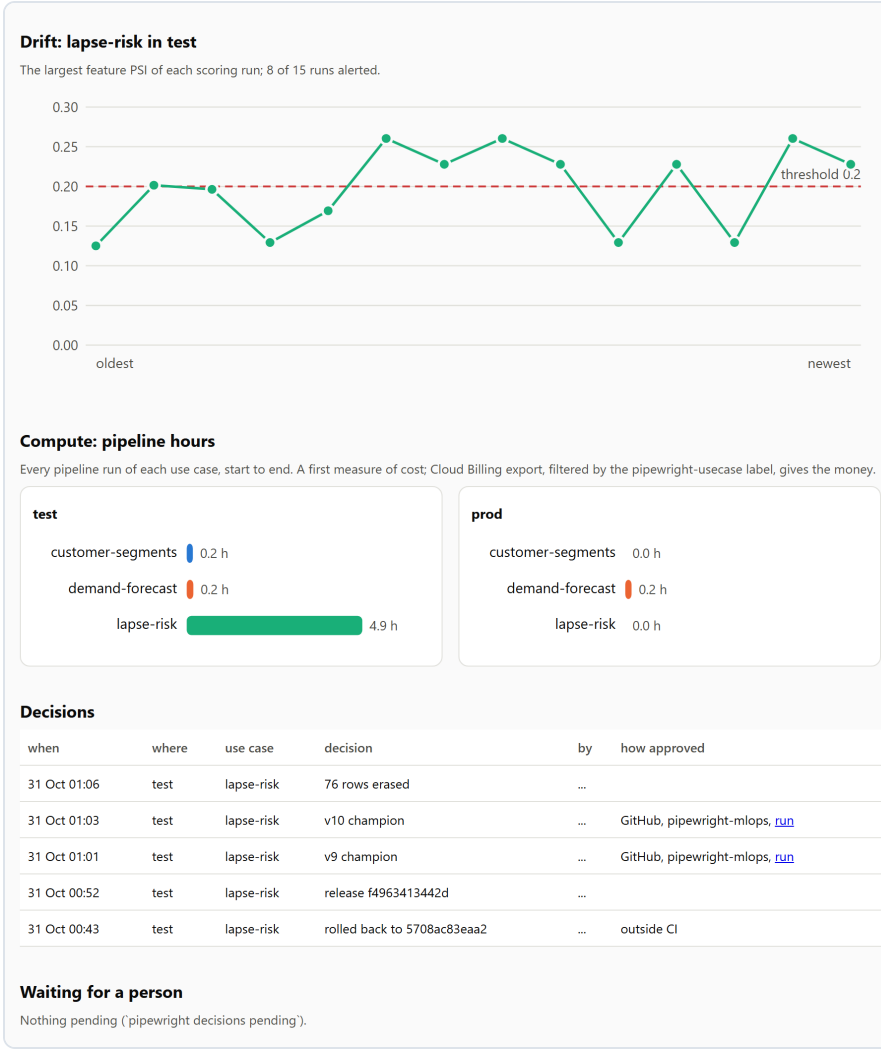


Figure 18.2 The rest of the fleet report: the largest feature PSI of each scoring run of the lapse model against the drift threshold; pipeline hours per use case in each environment; and the decision log, with who acted and how each decision was approved.

Drift. Every scoring run's largest feature PSI against the threshold: 8 of 15 runs alerted, all of them on the recency drift Chapter 14 traced to the training data. On a fleet page, a use case that alerts every week stands out at once, which is the point: the question is not "did it alert?" but "has anyone looked?".

Compute. Pipeline hours per use case, in each environment: the lapse model used 4.9 hours in test, the SQL team's two use cases a fifth of an hour each. This is the first cost question, "which use case uses the most?", and pipeline hours answer it well enough for a fleet of this size. The money itself comes from Cloud Billing: export the billing account's costs to BigQuery. Every pipeline run carries the labels `pipewright-usecase` and `pipewright-release`, with a billing ID Vertex AI adds for exactly this purpose, so a query by label gives each use case's pipeline bill. Endpoints and online stores need the same labels from whoever creates them.

Decisions. The event log, newest first: the erasure of Chapter 17, the two champion changes made through GitHub Actions with the user and a link to the run, the release made from a laptop, the rollback "outside CI". The `by` column is masked here; in the team's own copy it names the account that acted.

> **Pause and think.** If someone asked you this morning for a list of every model in production, with who approved it and when it was last checked, how would you produce it, and how long would it take? Would you trust the answer?

18.4 Building a report like this

The script is short, and worth reading, because it shows that the evidence is ordinary data in ordinary places. For each environment it:

1. reads the environment's manifest, which names its project, region, buckets and datasets;
2. reads each use case's release pointer, `contracts/<use case>/current.json`, for the live release and who moved it;

3. asks `pipewright lineage --json` for the champion, its metric and where it came from;
4. queries the use case's metrics and monitoring tables for its training and scoring history;
5. lists the project's pipeline jobs and adds up their hours per use case;
6. queries the event log, and `pipewright decisions pending --json` for what waits.

Two of those steps read files and tables directly where a command would have been cleaner: `pipewright status`, `model list` and `monitor status` print text, not JSON, in Pipewright 1.1.0, and the SQL team's use cases do not write their training runs to a metrics table. Pipewright's next version adds both. Until then the script reads the same records the commands do.(Since Pipewright 1.2.0, `status`, `model list`, `monitor status` and `retrain list` take `--json`, with stable field names, and a BigQuery ML use case with `bqml.metrics_table` writes each training run, approved or refused, to the same metrics table the container models use.)

Making the charts honest took more care than gathering the data. Three rules from the chart's code are worth borrowing for any fleet view:

Each use case keeps its colour. The lapse model is the same green in every chart, so nobody has to re-read a legend. Colour never means "good" or "bad"; that is what the filled and hollow markers, and the threshold line, are for.

Runs are plotted in order, not on a calendar. Training happens in bursts days apart; on a time axis, twenty runs collapse into two blobs.

Every mark answers a question on hover. The run's time, metric, decision and run ID, the batch and version of a scoring run, so the page leads straight back to `pipewright lineage`.

18.5 What the open-source platform does not answer

An honest audit section ends with the gaps, and the open-source starter has three worth knowing.

Private by identity, not by network. The starter's buckets enforce public access prevention, and every call to the launcher needs an authorised identity. But the launcher, the endpoints and the APIs are reachable from the internet, and data is encrypted with Google's keys, not yours. Regulated organisations usually require a service perimeter (VPC Service Controls), private endpoints and customer-managed keys. Those are platform decisions, made in Terraform, outside the use cases; the use cases do not change.

Evidence that you can hand over. The fleet report is a view of live records, regenerated on demand. An auditor often wants a pack per model version, frozen, signed and stored where nobody can change it. You can build that from the same sources: lineage, evaluation record, events, written to a bucket with a retention lock.

Many teams, one view. The script reads one repository's use cases. A platform with many teams and many repositories needs the view assembled from every environment's records, not from one checkout.

Pipewright Enterprise covers all three: a private-by-default environment blueprint, write-once evidence packs per model version, and Fleet View across teams. This book's subject is the open-source path, and everything in it works without them.

> **Pause and think.** Which of these gaps matter for your organisation today: the network perimeter, frozen evidence packs, or a view across teams? Which would an auditor ask about first?

Thinking it through

On evidence as a by-product. The cheapest evidence is the kind nobody had to produce. Every record in this chapter was written by a command doing its job. If your audit preparation involves people filling in documents, ask which command should have written it.

On reading a fleet. A fleet page is not a dashboard to admire; it is a list of questions. Why is production behind test? Why does this model alert every week? Why did those runs fail? A good page makes the questions obvious and the answers one command away.

On cost. Pipeline hours find the expensive use case; Cloud Billing by label prices it. Endpoints and online stores cost more than pipelines, every hour they exist, and appear in neither chart until someone switches them on. Check those first.

Try it: your own fleet report

Start from the `ch17/` folder of the book's repository, or open `ch18/`, which holds this chapter's finished code. It costs nothing beyond a few small queries.

1. Run it. From the repository root, with read access to each environment:

```
python tools/fleet_report.py --out fleet.html --mask
```

Open `fleet.html`. Which cells are marked in bold, and why?

2. Read one chart. Hover over the hollow points of a training chart and look up one refused run with `pipewright lineage`. Why was it refused?

3. Find a decision. Make one decision, approve a model version or roll back a release, run the report again, and find it in the decision log with how it was approved.

4. Price a use case. If your billing account exports to BigQuery, query it for the `pipewright-usecase` label and compare the money with the pipeline hours.

In short

- An audit asks five things: what runs where, how it was tested, who decided, what watched it, what it cost.
- Pipewright already records all five: release pointers, the registry, metrics and monitoring tables, pipeline jobs, and the event log of Chapter 17.
- Evidence is worth something when it is written at the time by the system, hard to change, joined by identifiers, and honest about its gaps.
- `tools/fleet_report.py` gathers it into one page with charts: what is live where, training runs kept and refused, drift, compute, decisions.
- A fleet page is a list of questions: production behind test, a model that alerts every week, a run of refusals.
- Pipeline hours find the expensive use case; Cloud Billing, by the pipeline runs' `pipewright-usecase` label, prices it.
- The open-source starter is private by identity, not by network; perimeters, frozen evidence packs and cross-team views are platform decisions beyond it.

Next: Chapter 19. From endpoint to app. The book ends where the retailer's customers are: a small web app that calls the lapse model at the checkout, and what it takes to put it in front of the public safely.

Chapter 19. From endpoint to app

The model meets the customer

Everything in this book so far has happened behind the retailer's walls: tables, pipelines, releases, gates, decisions. This chapter goes to the place the models were built for: the moment a customer reaches the checkout, and the website has a fraction of a second to decide whether to show them a "welcome back" offer.

That decision is not the model's. The model says how likely the customer is to stop buying. The business decides what to do about it: offer a discount above a certain risk, for instance, and not below. The app's job is to ask the model, apply the rule, and fail safely when it cannot ask.

19.1 What the checkout needs

The web team's requirements, written down before any code:

- **Ask with a customer ID.** The checkout knows who the customer is, nothing about their purchase history.
- **Answer before the page loads.** Hundreds of milliseconds, not seconds.
- **Apply the business's rule.** Offer when the lapse risk is at least 0.6; the marketing team owns the number and can change it without a new model.
- **Fail safe.** If the model cannot be asked, show no offer. A missing discount costs little; an error page at the checkout costs a sale.
- **Say why.** Support staff must be able to see why a customer did or did not get the offer.

19.2 Three ways to answer

Chapter 13 built the pieces for three ways to use the lapse model, and the checkout can use any of them.

The weekly list. Every Monday the scoring pipeline writes every active customer's lapse risk to `lapse_risk_scores` (Chapters 13 and 15). The app looks the customer up: one query, no endpoint, no model running. The score is as fresh as last Monday's run, which for a model whose features change once a month is fresh enough.

A fresh score, with the features in the request. The app reads the customer's features as of today from the data team's feature table, prepares them exactly as training did, and calls the lapse model's endpoint over REST with the prepared values. The score reflects today's features, but an endpoint costs money every hour it is deployed (Chapter 13), and the app now holds a copy of the preparation.

A fresh score, with the features in an online store. The app sends the customer ID alone. The endpoint reads the customer's features from an online store, a copy of the feature table kept for fast lookups by ID (Chapter 13), and prepares them itself, with the medians it was trained with. The app holds no preparation at all, but now two things cost money every hour: the endpoint and the store.

	Weekly list	Features in the request	Features in the online store
What the app sends	nothing: one lookup in BigQuery	the prepared features	the customer ID
Freshness	the last weekly run	today's feature table	the store's last sync (daily here)
Running cost	pennies a week	the endpoint, every hour	the endpoint and the store, every hour
Preparation	once, in the scoring pipeline	a copy in the app: the risk of skew	in the model only: no copy to drift
Speed	one BigQuery lookup	a BigQuery lookup, then the call	the call alone, the store answers in milliseconds
Says which features were filled	not needed	yes, the app filled them	no, the endpoint fills them silently
Can fail because	the customer is new since Monday	the endpoint is down, a feature is odd	the endpoint or store is down, the ID is not in the store
Platform work	none	none beyond the endpoint	the store, its views and their sync, per environment

None of them wins everywhere. The weekly list is cheapest and simplest, and the right start. The request path adds freshness with nothing new to run, at the price of a second copy of the preparation, which is exactly where training and serving drift apart. The online store removes that copy and the slow lookup, at the price of a store that bills every hour and a sync that someone must watch. The retailer's checkout uses the weekly list and asks for a fresh score only when it has a reason to; Chapter 13's advice holds at the very end of the book: batch first, and an endpoint when someone can

name why last Monday is too old. Section 19.5 shows the online store working end to end, after the upgrade to Pipewright 1.2.

19.3 The service

The checkout does not call the model; it calls a small service the ML team owns, `app/main.py` in the book's repository, written with FastAPI. Its one useful endpoint:

```
GET /api/offer/{customer}?fresh_score=true|false
```

answers with the decision and everything behind it:

```
{
  "offer": true, "offer_at": 0.6, "based_on": "fresh",
  "weekly": {"lapse_risk": 0.832, "model_version": "1", "scored_at":
"2026-10-31T01:42:35"},
  "fresh":  {"available": true, "lapse_risk": 0.832,
"model_version": "1", "features_as_of": "2011-12-01",
             "filled_with_median": ["avg_days_between_orders"],
"millis": 267}
}
```

Four details in the code carry the lessons of earlier chapters.

The preparation comes from the model being served. Before a fresh call, the service asks the endpoint which model version it serves and reads that version's `features.json` and `training_stats.json`: the feature order and the training medians. An empty rhythm is filled with the median, exactly as batch scoring does (Chapter 13), and the answer says which features were filled. When a new version is deployed, the service follows it without a change.

The rule is configuration. `OFFER_AT=0.6` is an environment setting of the service. The marketing team changes it; the model does not move.

It fails closed. A customer with no score gets no offer. An endpoint that is not deployed or answers with an error gives `"available": false` and a reason, and the decision falls back to the weekly score.

It logs decisions, not customers. The service records that a decision was made and on what basis, never the customer's ID or features. The scores table and the event log already say what the model said, under their own access rules.

There is also a one-page front end, for support staff and for the book (Figure 19.1). A link can carry the question, `/?customer=12350&fresh=1`, so a support ticket can point straight at an explanation.

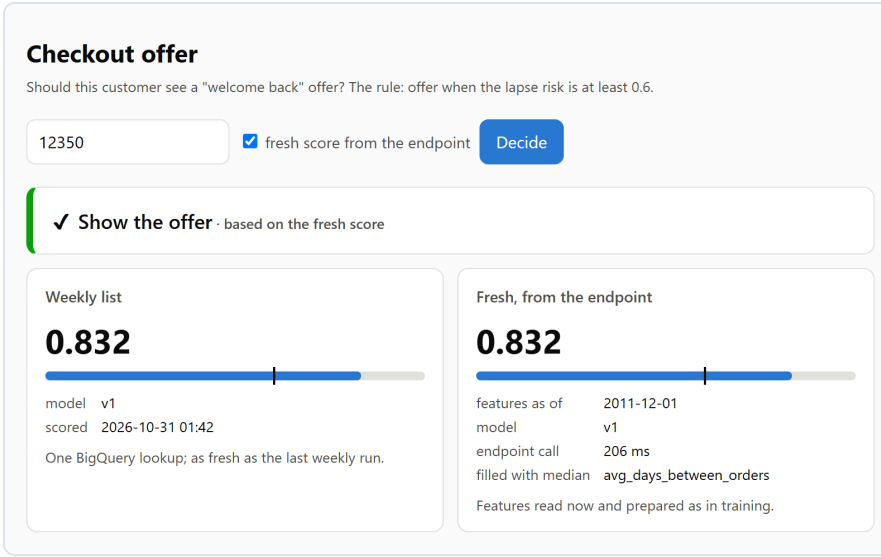


Figure 19.1 The checkout offer service for a one-order customer: the decision and its basis, the weekly score and a fresh score from production's endpoint, side by side, with the offer threshold marked on each bar. The fresh score filled the empty rhythm with the training median, and says so.

19.4 Production, through the pipeline

The service talks to production, so production had to be ready, and every step went through Chapter 16's workflows.

Which model. Production's champion was version 2, the model that carries test's online-store address (Chapter 13): it cannot serve requests. Version 1, promoted from test before online features existed, served correctly over REST in Chapter 13. The team made version 1 the champion again through the operate workflow, and the event log recorded it with the GitHub user and the run.

The weekly list. Production's data team created the scoring view, and the operate workflow started production's scoring: 4,298 customers scored by version 1.

The endpoint. The operate workflow deployed version 1 to production's endpoint, with production's manual approval given in the workflow's input; it took 16 minutes.

Then the service ran against production:

Customer	Weekly	Fresh	Filled with median	Offer
12347, six orders this year	0.293	0.293	none	no
12348, four orders	0.486	0.486	none	no
12350, one order, 301 days ago	0.832	0.832	`avg_days_between_orders`	yes

One query on the weekly list put the rule in proportion: at 0.6, 2,409 of the 4,298 customers would see the offer, more than half of them. That is not a question for the model. It is the marketing team's: can the budget afford a discount for half the customers at the checkout, and would a higher threshold, or Chapter 13's A/B test with a group that gets nothing, spend it better?

The fresh and weekly scores agree, as they should: the features have not changed since Monday, and the same model version produced both. The day they disagree, something has moved, and the service shows both side by side so someone notices.

The timing. The model itself answered in about 150 milliseconds once warm, and the first call of the day took nearly a second. A whole request took about half a second on the weekly path and 1.3 seconds with a fresh score, almost all of it BigQuery lookups, which are built for scanning millions of rows, not for finding one in a hurry. That is too slow for a checkout, and it is the honest case for what comes next: keep the weekly list in a fast cache, or move the features to the online store, which the next section does.

> **Pause and think.** Where does the business rule live in your systems, the threshold that turns a score into an action? Could the people who own it change it without asking the people who own the model?

19.5 The online store, with Pipewright 1.2.0

Chapter 13 built an online store and took it down again, for two reasons Pipewright 1.1.0 could not get past. A model promoted to production still read test's store, because the store's address travelled inside the model's files. And a customer with one order, whose `avg_days_between_orders` is empty, was refused with an error: the endpoint had no training median to fill it with. Pipewright 1.2.0 fixes both, so this section upgrades and turns the store back on.

The platform turns the store on. In the starter, the tenant's `online_features` block in `terraform.tfvars` comes back (Chapter 13 commented it out to save cost), in test and in production: a

Bigtable store, the `customer_activity` view over the data team's feature group, and its sync. `pipewright doctor` then checks it:

```
Online features of lapse-risk
  OK    online store: pw_retail_online (Bigtable)
  OK    view customer_activity: last sync 2026-10-31T08:41 (0 h
ago); 5 features
```

The use case asks for it. One line, the one Chapter 14 switched off:

```
model:
  serving:
    features: online          # the endpoint reads the features by
customer ID
```

The model must be trained again. What fills an empty feature is the serving code packaged with the model, and the medians it needs are written by training. A model trained under 1.1.0 has neither, so the fix arrives with the next model trained with the 1.2.0 example's `train.py`, `register.py` and `torchserve_handler.py`. Here the book meets its own gates: on the corrected data the lapse model fails the calibration gate (Chapter 14). To measure the online store at all, a scratch copy of the repository relaxed the gate to 0.12 and trained version 11; the book's repository keeps 0.05, and every environment went back to its champion afterwards.

Asking by ID. Version 11 in test, asked about three customers in one call:

Customer	Lapse risk	What happened
12347, a regular	0.286	read from the store
12346, one order	0.920	the empty rhythm filled with the training median, by the endpoint
99999999, unknown	none	an error for that ID only: “the online store has no entity with this ID”

The unknown ID matters as much as the one-order customer: under 1.1.0 one bad ID failed the whole call, so a checkout that batched requests lost every answer. Now each customer gets its own answer or its own reason.

Promotion gives production its own store. `pipewright promote` copied version 11 to production as version 3, and `doctor` in production says whose store it reads:

```
  OK    champion's online store: v3 reads this environment's store
```

Production answered the same scores from its own store. The model promoted under 1.1.0, production’s version 2, is refused rather than served wrongly:

```
online store: FAIL
  the model would read the online store projects/pw-
ebook/.../pw_retail_online (serving.json in its files),
  not prod's projects/pw-ebook-prod/.../pw_retail_online: it was
trained or promoted in another environment.
  Promote it again with Pipewright 1.2.0 or later, which gives a
promoted model the destination's store, or retrain it here
```

The service follows. `FEATURES=online` switches the checkout service to the third way: it sends the customer ID and skips its own feature lookup and preparation. The card on the page says the features were filled "by the endpoint, not reported": the price of moving the preparation into the model is that the app can no longer say what was filled, so the model's own logs and the monitoring table (Chapter 14) are where that question is answered now.

Against production, with version 3 deployed and the store synced:

Customer	Weekly list	Fresh, from the online store	Model call
12347, a regular	0.293, no offer	0.286, no offer	122 ms
12350, one order	0.832, offer	0.838, offer	128 ms
99999999, unknown	no score, no offer	"no entity with this ID", no offer	none

The fresh and weekly scores differ a little because they come from different versions: the weekly list was scored by version 1, the endpoint serves version 3. The timing is the point. The model call, with the store read inside it, took about 125 milliseconds, and the fresh score added about 0.3 seconds to a request; on the request path in section 19.4 it added about 0.8 seconds, nearly all of it BigQuery lookups. The rest of each request is the weekly lookup, which the online path does not change: that is the next thing to move into a cache.

Then the store came down again, in both projects, the same way it went up: the `online_features` block commented out, a plan that destroys only the store, its view and the sync grant, and an apply.

19.6 In front of the public

The service ran on a laptop, against production, for this chapter. Putting it in front of real customers is a platform decision, and four pieces of advice hold whatever platform you choose.

Never expose the model's endpoint. The endpoint takes raw features and returns raw probabilities; anyone who can call it can probe the model. The checkout's back end calls the offer service; the offer service calls the endpoint; the browser calls neither.

Give the service its own identity. On Cloud Run, deploy it with a service account of its own that can read the two tables, read the model's files and call the endpoint, and nothing else, and do not allow unauthenticated calls: the checkout's back end calls it with its own identity. The book's repository has the `Dockerfile`; the service account and its roles belong in the platform's Terraform, next to the starter.

Put a gateway in front when it faces the outside. If partners or a mobile app call the service directly, an API gateway such as **Apigee** gives it keys per caller, quotas and rate limits, and a record of who called how often, without changing its code.

Put a shield in front of anything public. A public load balancer with **Cloud Armor** filters abusive traffic, limits rates per client and blocks known attack patterns before they reach the service. A checkout is a target, and an offer service that hands out discounts is an interesting one.

None of this is specific to machine learning, which is the point: by the time a model reaches customers, it should be an ordinary, well-behaved service.

> **On Google Cloud: Apigee and Cloud Armor.** Apigee is Google Cloud's API management: keys, quotas, rate limits and analytics in front of your services. Cloud Armor is its web application firewall and DDoS protection, on a load balancer.
>
> *Why it matters.* A model endpoint is not built to face the public. Something in front of it must decide who may call, how often, and what to refuse, before a request costs you a prediction.
>
> *In Pipewright.* Pipewright stops at the endpoint and the service that calls it. What faces the public is the platform team's choice; these are the usual ones on Google Cloud.
>
> Read more: https://docs.cloud.google.com/apigee/docs

19.7 Leaving it costing nothing

The operate workflow undeployed production's endpoint, and the service went back to answering from the weekly list alone, which is how it behaves when the endpoint is away: the fresh card says `lapse-risk-endpoint serves no model`, and the decision falls back to the weekly score. Nothing else was left running.

One cost does not show on any running service: the images. Every release builds and pushes its component images, and nothing removes them, so after nineteen chapters the test registry held 77 images and several gigabytes. Each is a few pence a month, and they add up quietly. A blanket cleanup rule would be the wrong fix, because rollback and replay run the images of earlier releases.

Since Pipewright 1.2.0. `pipewright images prune --env test` lists the images no release needs, keeping every image a live or earlier release pins and the newest few of each, and deletes them with `--apply`; `doctor` reports how much the registry holds. In the book's test environment it removed nine images, about 4.8 GB, and found

nothing more on a second look. The starter also deletes untagged images after 30 days (`image_retention_days`).

Thinking it through

On the last mile. Most of the risk in this book was upstream: data, features, gates, promotion. Most of the value is here, in the moment a person sees the result. A model nobody acts on is a report; a model acted on badly is a liability. The service is where the two meet, and it deserves the same care.

On what to build first. The weekly list answered every customer in this chapter correctly, for pennies. The endpoint added freshness nobody needed yet, at an hourly cost. Build the cheap path, measure where it falls short, and add the expensive one for that reason.

On the book's model. The lapse model ends the book refused by its own gates on corrected data, waiting for a full year of honest history, while an earlier version serves customers well. That is not a failure; it is what a governed system looks like when it works: every decision recorded, every limit known, nothing shipped because someone was in a hurry.

Try it: the checkout offer service

Start from the `ch18/` folder of the book's repository, or open `ch19/` , which holds this chapter's finished code. The weekly path costs pennies; the endpoint costs money every hour it is deployed, so undeploy it in the same sitting.

1. Run the weekly scoring in the environment you want to serve (Chapter 13), so `lapse_risk_scores` has this week's list.

2. Start the service from the repository root, with your Application Default Credentials:

```
pip install -r app/requirements.txt
PROJECT=<your project> uvicorn app.main:app --port 8080
```

Open `http://localhost:8080/?customer=12350` and then `?customer=12347`. Change `OFFER_AT` and restart: which customers change?

3. Add a fresh score. Deploy the champion (Chapter 13), tick "fresh score from the endpoint", and compare the two cards. Which features were filled with the median, and why?

4. Ask by ID alone (optional, Pipewright 1.2.0). Turn on the online store (Section 19.5), set `model.serving.features: online`, train and deploy, then start the service with `FEATURES=online`. Ask about 12346: what does the card say about filled features now, and where would you look to find out?

5. Break it on purpose. Undeploy the endpoint and ask again: the service should fall back to the weekly score. Ask about a customer who is not on the list: no offer.

6. Leave it costing nothing. Undeploy, and if you turned the store on, comment `online_features` out again and apply: Bigtable bills every hour.

```
pipewright deploy-model --env <env> --usecase lapse-risk --undeploy
--approve
```

In short

- The model gives a probability; the business rule turns it into an action. Keep the rule outside the model, owned by the business.

- The checkout asks a small service the ML team owns, never the model's endpoint directly.
- Three ways to answer: the weekly list, cheap and as fresh as the last run; a fresh score with the features in the request, current but with a copy of the preparation in the app; a fresh score from the online store, by ID alone, with the preparation in the model and a store that bills every hour. Start with the weekly list.
- Pipewright 1.2.0 makes the online store usable in production: a promoted model reads the destination's store, an empty feature is filled as in training, and an unknown ID fails alone.
- A fresh score must be prepared exactly as training did: the served model's feature order and medians, read from its own files.
- Fail closed: no score, no offer, and say why.
- Measure before you optimise: here the model took 150 milliseconds and the lookups took the rest.
- In front of the public: its own identity, no unauthenticated calls, a gateway for outside callers, a shield for public traffic.

Next: Chapter 20. Building with a coding agent. The path is built. The last chapter shows how a coding agent can help a team walk it faster, from one sentence of intent to a pull request, without seeing a row of the retailer's data.

Chapter 20. Building with a coding agent

The agent writes, Pipewright checks, a person decides

By now the retailer's ML team writes a new use case the way the book has taught: a data contract, features from the catalogue, a time split, gates, monitoring, a pull request. It takes an afternoon, most of it spent looking things up: which dataset, which feature group, what the platform requires, what the last use case did.

That afternoon is exactly what coding agents are good at. Claude Code, Gemini CLI and their peers read a repository, write files, run commands and fix what fails. Pointed at an MLOps repository, they can draft a use case in minutes. They can also guess a dataset name, ask for "ten rows so I understand the table", split monthly snapshots at random, and open a pull request nobody can check. This chapter shows how Pipewright 1.3.0 lets you use an agent for the first kind of work without the second.

20.1 What goes wrong without help

An agent with a shell and your credentials is you, typing very fast. Three things follow, and none of them is the agent's fault.

Rows end up in a model's context. Reading a few rows is the natural way to understand a table, so it is the first thing an agent tries. With your access, nothing stops it, and the rows travel to whichever model the agent uses. For the retailer's customer history, that is personal data leaving the platform.

The configuration is plausible and wrong. The agent has never seen your environment. It writes YAML that looks like every use case it was trained on: a stratified split on rows ordered in time, a date as a

feature, a feature computed in the use case that the data team already owns. Much of it validates. Some of it misleads, in the ways Chapters 4, 10 and 11 spent pages on.

Nobody can say what it saw. When a risk team asks whether customer data reached an AI model, the honest answer today, in most organisations, is that nobody knows.

The answer is not to trust the agent more. It is to put the limits where the agent cannot move them.

20.2 Three limits that do not depend on the agent

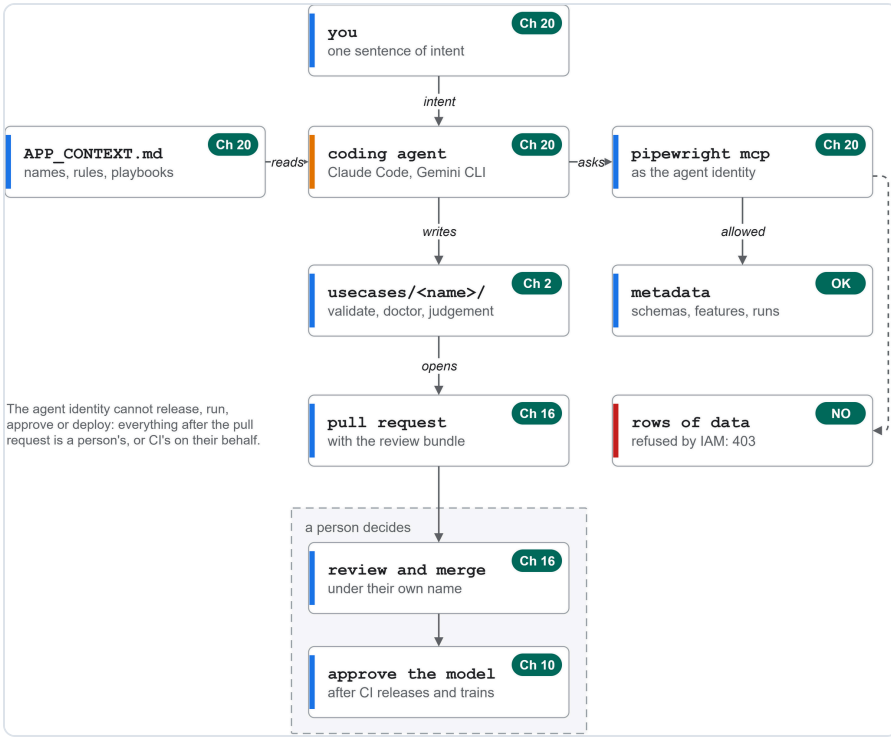


Figure 20.1 Building with a coding agent. The agent reads the context pack and asks Pipewright's tools, which act as its own identity: metadata is allowed, rows are refused by Google Cloud itself. The agent writes the use case and opens a pull request; everything after that is a person's decision, or CI's on their behalf.

An identity of its own. The agent's tools run as a service account that can read metadata and nothing else: schemas, row counts, feature definitions, runs, models. Google Cloud's IAM refuses table data to that account on every route, whether the agent goes through Pipewright's tools, through `bq`, or through anything else.

Context it cannot get wrong. A generated file, `APP_CONTEXT.md`, tells the agent the environment's names, rules and use cases. It is built

from the manifest, stamped, and checked in CI, so it cannot quietly go out of date.

A person in the loop. The agent's identity cannot release, run, approve or deploy. Its work arrives as a pull request, with a review bundle that says what would change and why. The book's whole path, from Chapter 7's releases to Chapter 16's approvals, is still the only way to production.

Look at where each limit lives: in IAM, in a file CI checks, in GitHub's branch protection. None of them lives in a prompt. You will hear advice to tell an agent "never read customer data"; keep that instruction, and do not rely on it.

> **Pause and think.** If a developer in your organisation pointed a coding agent at your data platform tomorrow, what could it read, as whom, and where would you find out afterwards?

20.3 An identity for the agent

The starter creates the agent's identity with one setting per team, in `terraform.tfvars`:

```
tenants = {
  retail = {
    members          = ["user:ana@example.com"]
    agent_identity   = true                          # pw-retail-
agent: metadata only
    feature_datasets = ["pw-ebook.pw_features"]      # the data
team's feature tables
  }
}
```

The account, `pw-retail-agent`, gets metadata roles on the team's dataset and its feature datasets, read access to the manifest and release contracts, and viewer roles for runs, models, schedules and alerts. The team's members may impersonate it. The manifest names it, so every Pipewright tool knows who to act as.

> **On Google Cloud: Service account impersonation.** A person with the Token Creator role on a service account can obtain short-lived credentials for it and act as it, without a key file.
>
> *Why it matters.* The agent gets the account's narrow permissions, not the person's broad ones, and Cloud Audit Logs record both names: the account as the principal, the person in its delegation chain.
>
> *In Pipewright.* The tool server impersonates the team's agent account for every call, and `doctor`, run for an agent, checks the account cannot release, run, read table data, upload or deploy.
>
> Read more: https://docs.cloud.google.com/iam/docs/service-account-impersonation

Before trusting it, the book tried to get round it. In the test project, acting as `pw-retail-agent`, a script asked for the entity rows with a query, the feature values with a query, an average of the label, the same two tables through BigQuery's table data API, and a listing of the landing bucket. Six attempts, six refusals, every one a 403 from Google Cloud. The table's metadata came back. That is the point of an identity: it holds whatever the agent tries.

The first run of `doctor` as the agent found a gap of its own: thirteen failures, because the account could not read the platform's buckets, registries and services *as configuration*, which `doctor` checks. The starter now grants a small custom role for exactly those reads; the

second run passed with no failures, and the account still could not read a row.

20.4 The context pack

```
pipewright context build --env test
```

```
wrote APP_CONTEXT.md
wrote AGENTS.md
wrote CLAUDE.md
wrote GEMINI.md
```

`APP_CONTEXT.md` holds what an agent must not guess: the team `retail`, its dataset `pw_retail`, its feature dataset, the environment's approval policies, the rules every use case is held to (time splits for time-ordered rows, personal data never a feature, features from the catalogue), every use case with where its features come from, and four playbooks in `.pipewright/playbooks/`: onboard a use case, add a drift alert, debug a failed run, change a gate. The three pointer files are the ones agents look for when they start; each gets a short marked block pointing at the pack, and anything else you wrote in them is left alone.

The pack is stamped with a fingerprint of Pipewright's version, the manifest and every use case file. `pipewright context check --env test` fails when any of them has changed since, and the pull request workflow from Chapter 16 runs it. An agent working from last month's picture of the environment is worse than one with no picture, because it is confident.

20.5 The tools

```
pip install "pipewright-mlops[gcp,mcp]==1.3.0"
pipewright mcp --env test --tenant retail --print-config
```

```
{ "mcpServers": { "pipewright": { "command": "pipewright",
    "args": ["mcp", "--env", "test", "--tenant", "retail"] } } }
```

That goes in `.mcp.json` at the repository root for Claude Code, or in `.gemini/settings.json` for Gemini CLI. The Model Context Protocol (MCP) is the common way agents call tools; `pipewright mcp` is a tool server the agent starts on your machine. Its tools are Pipewright's own commands, read-only: `validate`, `doctor`, `plan`, `inspect_table`, `list_features`, `status`, `runs`, `model_list`, `lineage`, `bootstrap_usecase`, `review_bundle` and a few more.

Here is what the agent learned in the book's test project, from a small entity table made for this chapter (one row per customer and month, with a `churned` flag) and the data team's `customer_activity` feature group:

```
> environment
acting_as  pw-retail-agent@pw-ebook.iam.gserviceaccount.com
data rules metadata and platform configuration only; rows never; no
aggregates

> list_features
feature group customer_activity  recency_days, spend_90d, orders_90d
(owner: data-team)
rule  Use these through data.features_from. A missing feature goes
back to the data team;
      do not compute it in the use case.

> inspect_table pw_retail.retail_entities
customer_id STRING, snapshot_date DATE, churned INTEGER
rows 16000, partitioned by snapshot_date

> sample_rows pw_retail.retail_entities
refused: rows are never sent to an AI agent (data class: rows)
```

Then the draft. `bootstrap_usecase` wrote a use case folder from the table's schema and the chosen feature group, with notes for the author that an agent should never skip:

```
> bootstrap_usecase churn-daily  table=retail_entities
label=churned
created usecases/churn-daily
  label values assumed 0/1 (not read: an AI agent sees no data);
check them
  runs read the entity table: pipewright run --source-uri bq://pw-
ebook.pw_retail.retail_entities
  customer_id looks like an identifier: list personal data in
data_protection.personal_data_columns
  features: recency_days, orders_90d, spend_90d
```

Notice what the agent did not do: it did not compute features. They come from the feature group, as of each row's `snapshot_date`, exactly as Chapter 4's point-in-time rule requires. The agent learned the catalogue existed from the tools and the rule from the context pack; neither came from a row of data.

A table description, written by a person, comes back marked as untrusted text. A description cannot instruct the agent. That matters more than it sounds: anyone who can edit a table's description could otherwise write instructions for every agent that reads it.

> **Pause and think.** Which of your tables would you be uncomfortable describing to an outside contractor by schema and row count alone? What does that tell you about where your sensitive data lives?

20.6 Checks for judgement, and the review

`validate` has always refused what is wrong. From 1.3.0 it also warns about what validates and still misleads: rows ordered in time with a random split, a date or timestamp as a feature, a feature named after the label, and a team with feature datasets whose use case takes nothing from them. They are warnings, so nothing that passed before fails. The one rule that refuses is the platform's own: a manifest can set gate minimums per environment, and a use case below them fails like any policy.

```
"policies": { "evaluation": { "min_gates": { "auc": 0.70 } } }
```

These checks are not only for agents. They are the mistakes people make most, the ones this book has warned you about chapter by

chapter, now caught in the place every change passes through.

The agent finishes with the review bundle:

```
pipewright review --env test --base main
```

```
## Review bundle: churn-daily (test)
### churn-daily (new)
What a release would change in Google Cloud
- + table churn_daily_model_metrics in pw_retail
- + table churn_daily_raw_customers in pw_retail
- = no IAM, network or project changes (Pipewright never makes them)
Checks      validate: passes, no warnings
Features    from customer_activity (recency_days, orders_90d,
spend_90d)
### Judgement calls
```

The last section is the agent's to fill in, and it is the most useful part of the pull request. In a session with Claude Code on another environment, the agent wrote judgement calls nobody had asked for: that a daily schedule on monthly snapshots would mostly retrain on the same rows, that the label was assumed to be 0 and 1 because it had never seen a value, and that it had listed `customer_id` as personal data. A reviewer cannot weigh a decision nobody wrote down. Ask your agent for this section every time, and read it first.

20.7 What the record shows

Every call the tools made is in Cloud Audit Logs, written by Google Cloud, not by the agent or by you. One entry from the session above, shortened:

```
principalEmail   pw-retail-agent@pw-ebook.iam.gserviceaccount.com
delegation       ...  (the engineer who impersonated it)
query            SELECT AVG(churned) FROM `pw-
ebook.pw_retail.retail_entities`
permissions      bigquery.jobs.create: granted
                 bigquery.tables.getData: denied
status           Access Denied: Table pw-
ebook:pw_retail.retail_entities: User does not
                 have permission to query table ...
```

The account may start a query job, which is how it dry-runs SQL for `plan`; it may not read the table the query touches. Even an average, a single number, is refused, because an average is computed from rows.

Refused attempts and queries are always recorded; with Data Access audit logs on (`agent_audit_logs = true` in the starter, billed by volume), every metadata read is too. Calls through Pipewright's tools also carry the user agent `pipewright-mcp/1.3.0 agent=<name>` and BigQuery job labels, so you can tell them from anything else the account did. On your machine, `.pipewright/mcp-log.jsonl` lists every call for your own reading; the audit log is the record.

Pipewright Enterprise. For organisations that must prove it to someone else, Enterprise adds a hosted tool server per team inside the perimeter, as an identity nobody can impersonate; aggregates such as a label's balance, only in groups above a minimum size the platform sets; every tool call as a row in the evidence dataset; an evidence pack control that shows no agent identity read a row of table data; certified agent configurations checked in CI; and an AI agents page in Fleet View. getpipewright.dev has the details and a recorded session with Claude Code.

Thinking it through

On speed. An agent turns an afternoon into minutes. It does not turn a wrong decision into a right one, and it makes wrong decisions faster too. The limits in this chapter do not slow the agent down; they make its speed safe to accept.

On where limits live. Every limit here is enforced by something the agent cannot edit: IAM, a stamped file checked in CI, branch protection, a person's approval. When you add your own rules for agents, ask the same question of each: if the agent ignored this instruction, what would stop it?

On the review. The review bundle and its judgement calls are as useful for people as for agents. A team that asks every pull request, human or not, to say what it would change and which decisions it made will review faster and argue less.

Try it: an agent on your own repository

Start from the `ch19/` folder of the book's repository. You need Pipewright 1.3.0, a coding agent that speaks MCP (the steps use Claude Code), and an environment from Appendix A. It costs nothing until you train.

1. Give the team an agent identity. In the starter's `terraform.tfvars`, set `agent_identity = true` for your team, and `feature_datasets` if the data team has feature tables. Plan, read the plan, apply.

2. Build the context pack and connect. Run `pipewright context build --env test`, put the output of `pipewright mcp --env test --tenant <team> --print-config` in `.mcp.json`, and commit both on a new branch. Start Claude Code in the repository and type `/mcp`: is **pipewright** connected?

3. Ask what it can see. Try this prompt:

```
Using only the pipewright tools, tell me what you can see in test.
List the feature
catalogue, describe the lapse_risk_entities table, and then try to
sample five rows
from it. Tell me plainly what was refused.
```

Then ask it to run `bq head` on the same table. What happens, and where would you find the attempt in Cloud Audit Logs?

4. Draft a use case. Ask it to follow `.pipewright/playbooks/onboard-a-use-case.md` for a small new use case of your own, to run `validate` and `doctor`, rebuild the context pack, and finish with `pipewright review` and its judgement calls. Which judgement calls would you have missed?

5. Leave it. Close the agent: the local tool server stops with it. Merge only what you would merge from a colleague.

The tutorial on getpipewright.dev goes further, through the pull request, CI, a training run and a model approval, with a list of things to watch out for on real laptops and networks.

In short

- A coding agent with your credentials is you, typing fast: it can read rows, write plausible wrong configuration, and leave no record of what it saw.
- Put the limits where the agent cannot move them: an identity that reads metadata only, a context pack checked in CI, and a person who approves every change.
- The starter creates the agent identity per team; IAM refuses table data to it on every route, and Cloud Audit Logs record it with the engineer who impersonated it.

- `pipewright context build` writes `APP_CONTEXT.md` and four playbooks; `context check` fails in CI when it is stale.
- `pipewright mcp` gives the agent Pipewright's read-only commands as tools: the catalogue, schemas, runs, drafting, checks and the review. Rows are always refused.
- `validate` warns about the mistakes that validate and still mislead, and a platform can set gate minimums that refuse.
- The review bundle ends with the author's judgement calls. Read them first.
- Pipewright Enterprise makes it provable: a hosted server, aggregates under rules, audit rows as evidence, and EV-16. Explore it, and the tutorial, at getpipewright.dev.

Appendix A. Your Google Cloud project

From Chapter 3 on, every **Try it** runs on Google Cloud. This appendix sets up the project those sections assume, says where the book's code is, and says what each part costs, so you can follow the book for pennies and know before you run something whether it will cost more.

The code

Every chapter's finished code is on GitHub at **github.com/fluxorsoft/pipewright-ebook**, one complete folder per chapter, `ch02/` to `ch19/`. Each folder is the retailer's MLOps repository as it stands at the end of that chapter: open the one for the chapter you are reading and run `pipewright` commands from inside it. `diff -r ch04 ch05` shows exactly what a chapter changed. The book's own project is called `pw-ebook`; wherever a command or a file shows it, use your project's name.

What you need

- **A Google Cloud project with billing enabled,** and permission to create resources in it. Owner is simplest for a project of your own. Use a new project, not one with other work in it: the environment adds service accounts, datasets and policies, and removing it is cleanest when nothing else is there.
- **Tools:** Python 3.11 or later, the Google Cloud CLI (signed in with `gcloud auth application-default login`), Terraform 1.9 or later, and Docker.
- **Pipewright:** `pip install "pipewright-mlops[gcp]==1.3.0"` (Chapter 1); add the `mcp` extra, `"pipewright-mlops[gcp,mcp]==1.3.0"`, for Chapter 20.

The environment

The book's environment is built with **pipewright-starter**, the open-source Terraform at github.com/fluxorsoft/pipewright-starter. It creates what Chapter 2's manifest describes: the buckets, the image and pipeline registries, an identity and a BigQuery dataset per team, the Pub/Sub topics, the launcher on Cloud Run, the event log of Chapter 17, and the manifest itself. Copy `terraform.tfvars.example` to `terraform.tfvars` and set at least these:

```
project_id = "your-project"
region     = "europe-west2"          # the book's region; any works
tenants    = { retail = { members = ["user:you@example.com"] } }
operators  = ["user:you@example.com"]
approver_groups = {
  ml-leads = { members = ["user:you@example.com"], description = "ML
leads" }
}
```

Then two applies, because the launcher's image has to be copied into your registry between them:

```
terraform init
terraform plan -out=env.tfplan && terraform apply env.tfplan
scripts/publish_launcher.sh "$(terraform output -raw
images_repository)"   # prints the image by digest
# set launcher_image in terraform.tfvars to what it printed, then:
terraform plan -out=env.tfplan && terraform apply env.tfplan
terraform output -raw environment_file    # save as
environments/test.yaml in the chapter's folder
```

A few chapters switch on more, each in its own section:

Chapter	What it adds	How
9	Ray on Agent Platform	a few extra permissions, listed in the chapter
12	a production environment	a second project, built the same way, with `promotion: { from: test }`
13, 19	the online store	`tenants.retail.online_features` in `terraform.tfvars`
16	CI in GitHub	`github = { repository = "your-org/your-repo" }`

What it costs

Most of the book costs pennies, because almost nothing in it runs between commands. The launcher scales to zero, and storage, BigQuery and Pub/Sub cost little at this size. What costs money is what runs, and for how long:

What	When it bills	In the book
Storage, BigQuery tables, Pub/Sub	always, by size	pennies a month for every chapter's tables together
A pipeline run	while it runs	pennies for a training run of a few minutes
A Ray run	while its cluster runs	well under a pound for Chapter 9's run, mostly the 16-vCPU head
An endpoint	every hour it is deployed, called or not	the largest cost if you forget one; scale to zero from 1.2.0
The online store	every hour it exists	Bigtable cannot scale to zero: Chapters 13 and 19 create it and remove it the same day
Images in the registry	always, by size	they accumulate with every release; `pipewright images prune` (Chapter 19)
A GKE cluster (Appendix C)	every hour it exists	not used by the chapters themselves

Every **Try it** ends with a step called **Leave it costing nothing**, and it is worth doing every time: undeploy the endpoints, remove the online store when you are done with it, and leave the launcher at zero. Check the current prices on Google Cloud's pricing pages for your region; this table says what bills, not how much.

When you have finished with the book, `terraform destroy` in the starter's folder removes the environment, buckets and datasets with their contents. Delete the project afterwards if you created it for the book.

Appendix B. Every setting of a use case

The retailer's `usecase.yaml` is short because Pipewright fills in a default for every key the team leaves out. This appendix shows what that short file stands for: every setting of the lapse model as it stands at the end of Chapter 5, written out as one complete use case. It is the file `ch05/reference/lapse-risk-all-settings.yaml` in the book's repository, generated from `pipewright resolve --env test --usecase lapse-risk` by `tools/all_settings.py`. As a single `usecase.yaml` it passes `pipewright validate` and resolves to exactly what the split files do.

How to read it:

- **No mark:** the retailer's own value, as written, `${...}` references included.
- `# default`**:** what Pipewright uses when the key is left out. Leave it out and you get this value.
- `# environments/test.yaml`**:** set for the test environment only (Section 2.5).
- `# not set by default`**:** a key that is off unless you give it a value, such as an output table for scores.
- **The closing comments** list the settings the lapse model does not use at all, each with what it is for.

Values the platform supplies, such as the runner's identity, the dataset and the landing prefix, are not here: they come from the environment manifest (Section 2.4). `pipewright explain <key>` explains any key in the file.

Section	What it holds	Chapters
`pipelines`, `components`, `triggers`, `parameters`, `steps`	the code, what starts a run, compute per step	2, 7, 9
`data`, `features`	the contract, the label, the split, feature tables, label windows, snapshots	3, 4, 5, 11
`training`, `evaluation`, `explainability`	hyperparameters, tuning, gates, the champion, slices	9, 10, 11
`registry`, `model`, `approvals`	the champion in the registry, serving, who decides	12, 13, 16
`inference`, `monitoring`, `retraining`	batch scoring, drift and performance, retraining decisions	13, 14, 15
`data_protection`, `outputs`	personal data and erasure, the tables the use case writes	2, 18

```
# Every setting of lapse-risk, as `pipewright resolve --env test` runs it.
# No mark: the team's own value, as written.
# `# default`: what Pipewright uses when the key is left out.
# `# environments/test.yaml`: set for this environment only.
# The platform's values (runner, dataset, landing prefix) are in the manifest.
# Written by tools/all_settings.py. `pipewright explain <key>` explains a key.
apiVersion: pipewright/v1
kind: UseCase
metadata:
  name: lapse-risk
  owner: ml-team
spec:
  tenant: retail
  pipelines:
    training:
      file: training_pipeline.py
      function: churn_training_pipeline
  components:
    dir: components/
    requirements: components/requirements.txt
  framework: pytorch  # default
  triggers:
    # landing:  # not set by default
    schedules: []  # default
    api: true
  parameters:
    bq_dataset: ${resources.bigquery_dataset}
    model_display_name: ${model.name}
```

```
    raw_table: lapse_risk_raw_customers
    metrics_table: lapse_risk_model_metrics
  steps: {}  # default
  data:
    contract:
      min_rows: 1000
      unique: [customer_id, snapshot_date]
      allow_extra_columns: false
      columns:
        - name: customer_id
          type: string
          required: true
          max_null_fraction: 0
        - name: snapshot_date
          type: date
          required: true
          max_null_fraction: 0
        - {name: recency_days, type: integer, min: 0, max: 365}
        - {name: orders_12m, type: integer, min: 1}
        - {name: spend_12m_gbp, type: float, min: 0}
        - {name: distinct_products, type: integer, min: 1}
        - {name: country, type: string, max_null_fraction: 0.01}
        - name: returned
          type: integer
          allowed: [0, 1]
          max_null_fraction: 0
    label:
      column: returned
    split:
      strategy: time
      time_column: snapshot_date
      test_fraction: 0.2  # default
      validation_fraction: 0.0  # default
      seed: 42  # default
    features_from:
      entity:
        columns: [customer_id]
        time_column: snapshot_date
      sources:
        - table: bq://pw_retail.customer_activity
          features: [recency_days, orders_12m, spend_12m_gbp, distinct_products]
    label_source:
      table: bq://pw_retail.customer_purchases
      time_column: purchased_at
      window_days: 90
      key: [customer_id]  # default
      cutoff_column: snapshot_date  # default
    snapshot_table: lapse_risk_snapshots
  features:
    include:  # default
      - recency_days
      - orders_12m
      - spend_12m_gbp
      - distinct_products
    exclude: [customer_id, country]
  training:  # default
    hyperparameters: {}  # default
    seed: 42  # default
```

```
    experiment:  # default
      name: lapse-risk  # default
    tuning:  # default
      enabled: false  # default
      metric: auc  # default
      goal: maximize  # default
      max_trials: 8  # default
      parallel_trials: 4  # default
      search_space: {}  # default
  evaluation:
    primary_metric: auc  # default
    decision_threshold: 0.5  # default
    metrics:  # default
      - accuracy
      - auc
      - calibration_error
      - f1
      - log_loss
      - pr_auc
      - precision
      - recall
    gates:
      - {metric: auc, min: 0.7}
    champion:  # default
      enabled: true  # default
      metric: auc  # default
      min_improvement: 0.0  # default
    slices: []  # default
    slice_gates: []  # default
  explainability:  # default
    enabled: false  # default
    method: permutation  # default
    top_k: 10  # default
  registry:  # default
    set_default: on_approval  # default
  model_card: {}  # default
  inference:  # default
    # output_table:  # not set by default
    decision_threshold: 0.5  # default
  monitoring:  # default
    drift:  # default
      enabled: false  # default
      method: psi  # default
      threshold: 0.2  # default
      features:  # default
        - recency_days
        - orders_12m
        - spend_12m_gbp
        - distinct_products
    prediction_drift:  # default
      enabled: false  # default
      threshold: 0.2  # default
    performance:  # default
      enabled: false  # default
      # labels:  # not set by default
      metric: auc  # default
      # min:  # not set by default
    # output_table:  # not set by default
```

```
    alerts:  # default
      notify: []  # default
      events: [monitoring.alert]  # default
      severity: WARNING  # default
  retraining:  # default
    mode: 'off'  # default
    enabled: false  # default
    when: [schedule]  # default
    cooldown_days: 7  # default
    # source_uri:  # not set by default
    decisions:  # default
      notify: []  # default
      publish: false  # default
      escalate_after_days: 3  # default
      escalate_notify: []  # default
      digest: true  # default
    approvals:  # default
      required: []  # default
      separate_from_requester: true  # default
    snooze:  # default
      max_days: 30  # default
      reopen_when_worse_by: 0.1  # default
  data_protection:  # default
    personal_data_columns: []  # default
    # erasure:  # not set by default
  outputs:
    store: bigquery  # default
    # gcs:  # not set by default
    bigquery:
      tables:
        lapse_risk_raw_customers:
          schema: schemas/raw_customers.json
          partition:
            field: ingested_at
            type: DAY
          # cluster:  # not set by default
        lapse_risk_model_metrics:
          schema: schemas/model_metrics.json
          partition:
            field: evaluated_at
            type: DAY
          # cluster:  # not set by default
        lapse_risk_snapshots:
          schema: schemas/snapshots.json
          partition:
            field: ingested_at
            type: DAY
          # cluster:  # not set by default
      # retention_days:  # not set by default
  model:
    promotion: model  # default
    name: lapse-risk-classifier
    serving:  # default
      enabled: false  # default
      target: vertex  # default
      machine: n1-standard-2  # default
      replicas:  # default
        min: 1  # default
```

```
        max: 1  # default
  approvals:
    promote_model: auto  # environments/test.yaml
    deploy_model: manual  # default
    approvers: []  # default

# Also available, not used by this use case (see `pipewright explain`):
#   bqml (21 keys): BigQuery ML as the training engine
#   components.ray_requirements: Extra packages for the use case's Ray ...
#   data.contract.max_null_fraction: The largest share of empty values ...
#   data.features_from.sources.entity_columns: The table's entity ...
#   data.features_from.sources.feature_group: A feature group registered ...
#   data.features_from.sources.online_view: The feature view in the ...
#   data.features_from.sources.prefix: Added to this table's feature ...
#   data.features_from.sources.project: The project the feature group is ...
#   data.features_from.sources.region: The region the feature group is ...
#   data.features_from.sources.time_column: The table's column saying ...
#   data.label.positive: The value that counts as positive for metrics ...
#   data.label.task: What the model predicts: binary (the default), ...
#   data.label.values: The label's possible values, for example [0, 1]
#   data_protection.erasure.key_column: The column that identifies a data ...
#   environments: Per-environment overrides, by environment name
#   evaluation.fairness (6 keys): Bias measures per attribute on the test ...
#   evaluation.gates.max: The highest value allowed
#   evaluation.slice_gates.max_gap: The largest amount a slice may be ...
#   evaluation.slice_gates.metric: The metric compared per slice
#   evaluation.slice_gates.min_rows: Slices with fewer test rows are not ...
#   include: Glob patterns, relative to the use case folder, of files ...
#   model.serving.canary_percent: The share of traffic a new version gets ...
#   model.serving.features: online: the endpoint takes entity IDs ...
#   model_card.ethical_considerations: Fairness and other risks, and how ...
#   model_card.intended_use: What the model is for and who uses its output
#   model_card.limitations: Known weaknesses, such as data it has not seen
#   model_card.out_of_scope: Uses the model is not suitable for
#   monitoring.performance.labels.key: The column that joins outcomes to ...
#   monitoring.performance.labels.label: The column holding the observed ...
#   monitoring.performance.labels.table: The table of observed outcomes
#   outputs.gcs.external_tables: Whether BigQuery external tables of the ...
#   outputs.gcs.format: The file format
#   pipelines.inference (2 keys): The pipeline that scores new data with ...
#   retraining.loop (1 key): The closed retrain loop: production never trains
#   run_in: The environments this use case runs in, by name (Pipewright ...
#   steps.<name> (15 keys): A step, by its component name with dashes ...
#   synthetic (67 keys): Synthetic data generation
#   tests (1 key): How `pipewright test` and the release check the use ...
#   training.hyperparameters.<name>: A hyperparameter value
#   training.tuning.search_space.<name> (5 keys): One hyperparameter's range
#   triggers.landing.pipeline: The pipeline a landed file starts
#   triggers.landing.suffixes: File name endings that start a run when a ...
#   triggers.schedules.cron: When it runs, in cron syntax, for example "0 ...
#   triggers.schedules.name: The schedule's name, unique in the use case
#   triggers.schedules.pipeline: The pipeline the schedule starts
#   triggers.schedules.source_uri: The gs:// data the scheduled run reads
#   triggers.schedules.timezone: The time zone the cron expression is ...
```

Appendix C. Running it on GKE

Every chapter of this book ran on Google's managed services: pipelines on Agent Platform Pipelines, models on Agent Platform endpoints, Ray on Agent Platform. Some teams run the same work on Google Kubernetes Engine instead, in a cluster of their own. This appendix is a short guide to what changes when you do, and what does not. It is reading, not a walkthrough: the outputs below come from a churn model trained, served and tuned on a private GKE test environment with Pipewright 1.2.0, the day 1.2.0 was released.

> **On Google Cloud: Google Kubernetes Engine (GKE).** Google Cloud's managed Kubernetes: Google runs the control plane, and your workloads run as containers on node pools of machines you choose.
>
> *Why it matters.* Kubernetes is where many platform teams already run everything else. Open-source machine learning tools (Kubeflow Pipelines, KServe, KubeRay) install into it, so the work stays inside the team's own cluster and network.
>
> *In Pipewright.* With `orchestrator: kfp` in the manifest, pipelines run on Kubeflow in a private GKE cluster, as each team's identity through Workload Identity; KubeRay and KServe take Ray's and the endpoints' places.
>
> Read more: https://docs.cloud.google.com/kubernetes-engine/docs, https://www.kubeflow.org/docs/components/pipelines/, https://kserve.github.io/website/, https://docs.ray.io/en/latest/cluster/kubernetes/index.html

Figure C.1 is Figure 1.5 again, for a private GKE environment. Everything outside the blue band is the same; inside it, the cluster runs the pipelines, the Ray clusters and the serving that Agent Platform ran before.

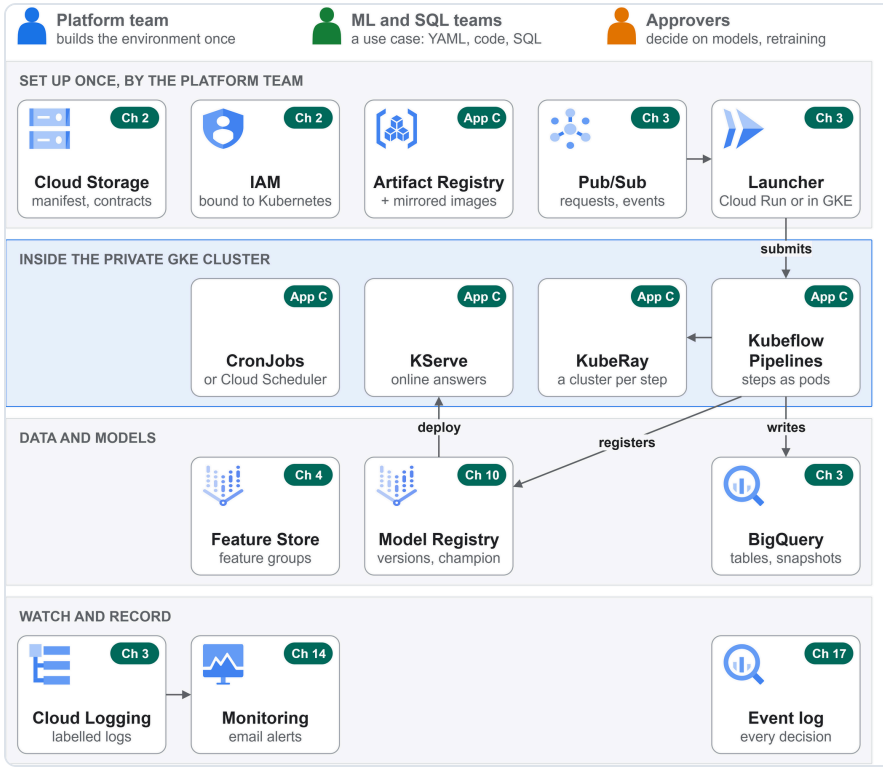


Figure C.1 The same environment with a private GKE cluster (the manifest says orchestrator: kfp). Kubeflow Pipelines runs the same compiled templates as pods, as each team's identity; KubeRay creates a Ray cluster for a step; KServe serves a registered version; schedules can be CronJobs. The launcher runs on Cloud Run or inside the cluster, and Artifact Registry also holds mirrored copies of the images the cluster pulls. The data, the registry and the records stay where they were.

Why a team chooses GKE

- **It already runs Kubernetes.** The platform team has GKE clusters, on-call rotas and tooling for them, and would rather add machine learning to that than run a second kind of platform.
- **It must stay inside a perimeter.** Inside an enforced VPC Service Controls perimeter, Ray on Agent Platform cannot reach its cluster (Chapter 9). On GKE, Ray runs inside the cluster, inside the perimeter.

- **It wants control of the machines.** Node pools sized and priced by the team, GPUs it reserves itself, a cluster that stays up between runs.
- **It wants the open-source pieces themselves.** Kubeflow Pipelines, KServe and KubeRay are open source and run on any Kubernetes, so the skills and the pipelines travel.

The price is that someone now runs the cluster: upgrades, node pools, the Kubeflow and KServe installations, and the images they pull. On Agent Platform, Google does that.

Kubeflow Pipelines

Agent Platform Pipelines runs pipelines written with the Kubeflow Pipelines SDK (KFP). Kubeflow Pipelines on GKE runs **the same compiled templates**: Pipewright compiles a use case's pipeline once, and the environment decides where it runs. Nothing in `training_pipeline.py` changes.

	Agent Platform Pipelines	Kubeflow Pipelines on GKE
Who runs it	Google	your platform team: an installation in the cluster
Each step runs as	a managed job, billed per run	a pod on your node pools
Idle cost	none	the nodes that stay up
Inside a perimeter	yes	yes, and Ray steps too
Run history and UI	the Google Cloud console	Kubeflow's own UI, in the cluster
Errors of a failed step	in the job's details	in the step's pod logs

The last row matters for operations. When a run fails on Agent Platform, the run's record carries the failing step's error. Kubeflow keeps the error in the step's pod logs, so Pipewright's `run.failed` event names the failed step and says where to look:

```
run.failed  churn-gke  failed step: ingest-data
  ingest-data: failed: Kubeflow keeps the error
  in the step's pod logs
```

Kubeflow's UI is not exposed to the network. `pipewright pipelines-ui --env <env>` opens it on your laptop through the cluster's API, as you, read only:

```
churn-gke-r189413a261bc        FAILED
churn-gke-ray-r4604b469a426    SUCCEEDED
churn-gke-r8d1613a188c4        SUCCEEDED
```

KServe

An Agent Platform endpoint serves a model from Model Registry on machines Google manages. **KServe** does the same inside the cluster: an `InferenceService` runs the model's serving container as pods in the team's namespace, with the same container and the same request format.

	Agent Platform endpoint	KServe on GKE
Who runs it	Google	your platform team: KServe in the cluster
Reached by	a Google Cloud URL, with IAM	an address in the cluster, or an internal load balancer in the VPC
Canary traffic split	yes (Chapter 13)	no, in the standard mode: a new version replaces the old with a rolling update
Scale to zero	yes, with Pipewright 1.2.0	no, in the standard mode: at least one pod
Machines	a machine type per endpoint	pods on node pools the team sizes

`pipewright deploy-model` works the same way on both, and ends by saying how to call the model:

```
live: churn-gke serves churn-gke-classifier v1
How to call it:
  service   InferenceService serving-churn/churn-gke
  cluster   POST http://churn-gke-predictor.serving-churn
                   .svc.cluster.local/predictions/model
  VPC       POST http://10.10.0.31/predictions/model
            (internal load balancer, no public address)
  request   features in this order: tenure_months,
            monthly_charges, total_charges, ...
            {"instances": [[0, 0.0, 0.0, 0, 0, 0, 0, 0.0]]}
```

There is no public address. A caller in the cluster or in the VPC reaches it over the network; from outside, `pipewright model predict` goes through the cluster's API, with your identity.

KubeRay

Chapter 9 tuned the lapse model on a Ray cluster that Agent Platform created for the run. On GKE, the **KubeRay** operator creates that cluster inside the Kubernetes cluster, on a node pool kept for Ray, and removes it when the step ends. The step's code is the same; `steps.<name>.ray` in the use case is the same.

	Ray on Agent Platform	KubeRay on GKE
A cluster per run	yes, started for the step	yes, started for the step
A standing cluster	no	yes, per team, with workers that scale to zero
Inside a perimeter	no (Chapter 9)	yes
Smallest useful head	16 vCPUs	what the node pool allows

The churn model's tuning run on KubeRay took nine minutes from submission to a registered version, with no head node to start outside the cluster.

The serving images and the mirror

Every piece above runs container images: Kubeflow's dozen or so services, the KServe controller and its certificate manager, the KubeRay operator, Ray itself, and the serving container your model runs in. On the public internet, the cluster pulls them from their publishers' registries. **A private cluster cannot**: it has no route to the internet. So the platform team copies each image, by its digest, into the environment's own Artifact Registry, a **mirror**, and the cluster pulls only from there. For the test environment above that was 31 images.

Two things follow:

- **An upgrade is a mirror first.** A new Kubeflow or KServe version means new images, copied and pinned before the cluster can use them.
- **Cleaning up must not touch the mirror.** `pipewright images prune` deletes only the old images of packages that your releases use, so the mirrored images stay, whatever their age:

```
Images in .../pw-ent-images: 39 (9.3 GB)
would free up to 0.0 GB
```

What changes in the use case

Very little, and that is the point. Where pipelines run is the environment's choice: the platform team builds a GKE environment, and every use case released there runs on Kubeflow. A use case changes only where it says how to serve:

```
model:
  serving:
    target: kserve           # vertex (default) or kserve
    machine: e2-standard-2   # a pod's size on the nodes
    replicas: { min: 1, max: 2 }   # at least one pod
```

Schedules can become Kubernetes CronJobs, also an environment setting, and `pipewright schedules sync` manages them the same way. Contracts, gates, promotion, lineage and the event log are identical: the rest of this book applies unchanged.

> **Pipewright Enterprise.** The open-source core runs on a GKE environment once one exists: the orchestrator, KServe serving and KubeRay are part of it. Building that environment, the private cluster, the Kubeflow, KServe and KubeRay installations, the image mirror and the perimeter around them, is the platform side. Pipewright Enterprise provides it as Terraform (its private-managed blueprint with `spec.gke`); a team can also build it with its own Terraform.

Appendix D. Google Cloud at a glance

Every Google Cloud service this book uses, grouped as in Figure 1.5, with what it does for you and the chapter that introduces it. Each chapter's **On Google Cloud** boxes say more: what the service is, why it matters for machine learning, and how Pipewright uses it. Agent Platform was formerly called Vertex AI; older articles and answers use that name, and its old documentation addresses redirect to the new ones.

Set up once, by the platform team

Service	What it does for you	Chapter
Cloud Storage	Buckets of files: the manifest, release contracts, landing files, run working files	2
IAM	Who may do what: a service account per team, the launcher and CI, no keys	2
Artifact Registry	Component images and compiled pipelines, pinned by digest	7
Pub/Sub	Messages: landing files, run requests, lifecycle events, approvals	3
Cloud Run	Runs the launcher, which turns files, schedules and requests into runs	3
Workload Identity Federation	GitHub Actions acting as a service account without a key	16

Every run

Service	What it does for you	Chapter
Agent Platform Pipelines	Runs each training and scoring pipeline, step by step, as the team's identity	3
Cloud Scheduler	Scheduled runs, the retrain tick and the weekly digest	15
BigQuery ML	Models trained and used in SQL	8
Agent Platform training machines	The machine, memory and GPUs each step runs on	9
Ray on Agent Platform	A Ray cluster for one step, created and deleted with it	9

Data

Service	What it does for you	Chapter
BigQuery	Tables queried with SQL: each team's data, predictions, metrics and monitoring	2
Agent Platform Feature Store	Registered feature groups over BigQuery, and online stores	4, 13
Feature Store online serving	The latest features by entity ID, on Bigtable	13, 19

Models and serving

Service	What it does for you	Chapter
Model Registry	Model versions, their labels and the champion	10
Agent Platform endpoints	Online predictions, with traffic split for canaries	13
Apigee and Cloud Armor	API keys, quotas and a firewall in front of a public service	19

Watch and record

Service	What it does for you	Chapter
Cloud Logging and Monitoring	Every step's logs, and the alerts that email a team	3, 14
BigQuery subscriptions	Every event on a topic written to a table: the event log	17

Regulated environments and GKE

Service	What it does for you	Chapter
Cloud KMS	Customer-managed encryption keys, in regulated environments	1
VPC Service Controls	A perimeter that keeps data inside the environment	9, C
GKE	Kubernetes for Kubeflow Pipelines, KServe and KubeRay	C

The platform team enables these APIs in each project: `aiplatform`, `artifactregistry`, `bigquery`, `cloudscheduler`, `iam`, `iamcredentials`, `logging`, `monitoring`, `pubsub`, `run` and `storage` (all `.googleapis.com`), plus `cloudkms` for customer-managed keys and `container` for GKE.

Where to read more

The product documentation, checked when this book went to press:

- Cloud Storage: https://docs.cloud.google.com/storage/docs
- BigQuery: https://docs.cloud.google.com/bigquery/docs
- IAM: https://docs.cloud.google.com/iam/docs/overview
- Agent Platform Pipelines: https://docs.cloud.google.com/gemini-enterprise-agent-platform/machine-learning/pipelines/introduction
- Cloud Run: https://docs.cloud.google.com/run/docs
- Pub/Sub: https://docs.cloud.google.com/pubsub/docs/overview
- Cloud Logging and Monitoring: https://docs.cloud.google.com/logging/docs/alerting/log-based-alerts
- Agent Platform Feature Store: https://docs.cloud.google.com/gemini-enterprise-agent-platform/machine-learning/featurestore/latest/overview
- Artifact Registry: https://docs.cloud.google.com/artifact-registry/docs
- BigQuery ML: https://docs.cloud.google.com/bigquery/docs/bqml-introduction
- Agent Platform training machines: https://docs.cloud.google.com/gemini-enterprise-agent-platform/machine-learning/training/configure-compute
- Ray on Agent Platform: https://docs.cloud.google.com/gemini-enterprise-agent-platform/machine-learning/open-source/ray
- Model Registry: https://docs.cloud.google.com/gemini-enterprise-agent-platform/machine-learning/model-registry/introduction
- Agent Platform endpoints: https://docs.cloud.google.com/gemini-enterprise-agent-

platform/machine-learning/predictions/get-online-predictions
- Feature Store online serving: https://docs.cloud.google.com/gemini-enterprise-agent-platform/machine-learning/featurestore/latest/online-serving-types
- Cloud Scheduler: https://docs.cloud.google.com/scheduler/docs
- Workload Identity Federation: https://docs.cloud.google.com/iam/docs/workload-identity-federation
- BigQuery subscriptions: https://docs.cloud.google.com/pubsub/docs/bigquery
- Apigee and Cloud Armor: https://docs.cloud.google.com/apigee/docs
- Cloud KMS: https://docs.cloud.google.com/kms/docs
- VPC Service Controls: https://docs.cloud.google.com/vpc-service-controls/docs
- GKE: https://docs.cloud.google.com/kubernetes-engine/docs

About the author

Rakesh Gupta is the founder of Fluxor Software and the creator of Pipewright. He has spent more than two decades building software for airlines, banks, automotive companies and governments, as a consultant and as a CTO, and the last few years helping companies with data and AI. He lives in London with his wife Shweta and their two children, Rishika and Ayansh. When he is not building software, he likes spending time with his family, trekking and playing tennis.

Data and credits

The running example. Chen, D. (2012). *Online Retail II* [Dataset]. UCI Machine Learning Repository. https://doi.org/10.24432/C5CG6D. Licensed under the Creative Commons Attribution 4.0 International licence (CC BY 4.0). Every figure about the retailer is computed from this dataset by the scripts that accompany the book.

Screenshots come from real runs of Pipewright 1.1.0, 1.2.0 and 1.3.0 in test environments, with identities, project names and environment names replaced.

Pipewright is open source under the Apache License 2.0: github.com/fluxorsoft/pipewright.

Where to go next

- **github.com/fluxorsoft/pipewright-ebook**: the example repository, one complete folder per chapter.
- **getpipewright.dev**: the tutorials that accompany each chapter, and the AI build assistance guide with its list of things to watch out for (Chapter 20).
- **github.com/fluxorsoft/pipewright**: the source, the documentation and the changelog.
- **pypi.org/project/pipewright-mlops**: `pip install pipewright-mlops`.

Most machine learning models never reach production. Many that do are switched off a year later. This book is the path that gets them there and keeps them there.

Follow a real UK online gift retailer, its Christmas peaks, its customers who drift away and its analysts who know SQL better than Python, as it takes five models from notebook to production on Google Cloud: released, approved, watched, retrained and traced the same way.

INSIDE

- Data contracts, point-in-time features and reproducible runs
- Synthetic data that is useful and provably safe to share
- Pipelines on Agent Platform, Kubeflow on GKE, Ray and BigQuery ML
- Promotion, serving, canaries, A/B tests and rollback
- Drift, monitoring and retraining with a person in the loop
- CI/CD in GitHub, with every approval on record
- Lineage and evidence your auditors can read
- An endpoint behind a web app, with Apigee and Cloud Armor advice
- Coding agents that build use cases without seeing a row of data
- A guide to running it all on GKE: Kubeflow, KServe and KubeRay

THE FIVE MODELS YOU WILL BUILD

Customer segments | Demand forecast | Lapse risk | Checkout offer | Synthetic customers

Rakesh Gupta is the founder of Fluxor Software and the creator of Pipewright, the open-source MLOps platform for Google Cloud used throughout this book.

getpipewright.dev