DELTAVAULT

The Data Lakehouse for Everyone

The work every lakehouse assumes you already did. A data lakehouse pays only once someone agrees what the business means and maps the sources to it.

2026

Contents

INTRODUCTION	Read this whichever chair you sit in
CHAPTER 1	The promise and the gap
CHAPTER 2	Why the silver layer fails
CHAPTER 3	Speak before you store
CHAPTER 4	Know your sources
CHAPTER 5	The mapping is the deliverable
CHAPTER 6	Choose an integration shape without a religion
CHAPTER 7	Templates, not handcrafted tables
CHAPTER 8	The metadata contract for AI
CHAPTER 9	Lineage, governance, and trust before go-live
CHAPTER 10	Six steps to your integration layer

Read this whichever chair you sit in

Land your data, and our AI does the rest. Every platform in the market makes you some version of that offer, and every version holds right up to the word “rest”. This book is about everything “the rest” assumes you already did.

None of that work is exotic.

1. Agree what your business means.
2. Find out what your sources actually hold.
3. Record how the two map together.
4. Choose what shape the integrated middle takes.
5. Capture enough of that decision to generate the layer instead of hand-typing it.

None of it happens inside a platform, which is why no platform ships it.

You are already in this book

- Maybe you are the engineer whose silver layer became a rename pass, because nobody would rule on which system wins.
- Maybe you are the architect asked to pick a modeling religion and then defend it for three years.
- Maybe you are the leader whose bronze layer shipped eighteen months ago while the gold layer still carries no weight in the meeting that matters.
- Or maybe you are the business expert answering the same definition question for the third time, in rooms where nobody was writing the answer down.

Four chairs, one problem, and that is exactly what the title argues. “For everyone” means every stack, because no decision in this book belongs to an engine: the questions arrive unchanged on Databricks, on Snowflake, on Microsoft Fabric, and on whatever replaces them. It means every role too, because a data lakehouse does not fail inside anybody’s job. It fails in the gaps between those jobs, where the decisions nobody scheduled get made anyway, at a keyboard, under deadline, unrecorded.

Four things this book is not

Let me say plainly what this book is not, so you can put it down now if you came for one of those things. It is not a platform comparison, and it does not rank the vendors. It is not a modeling religion either: chapter 6 gives Data Vault, third normal form, dimensional-first, and one big table an honest paragraph each, then hands you the factors that should drive the choice rather than a verdict. It is not a gold-layer deep dive, because stars and semantic layers are already the best-written part of the shelf. And it is not a product manual, because every method here runs on a spreadsheet and some discipline, if that is what you have.

How should you read this book?

Read it straight through the first time. Chapters 1 and 2 make the case that the deciding work sits between the layers rather than inside them. From there, Chapters 3, 4, and 5 cover everything that happens before the platform: speak before you store, know what your sources actually hold, and treat mapping as recorded decisions rather than guesswork. Chapters 6, 7, and 8 then build the integration layer, beginning with a shape you chose deliberately, moving to the templates that generate it, and ending with the metadata contract that generation runs on. Chapter 9 turns to trust, which in practice means lineage, governance, and version control. Chapter 10 is Monday morning: six steps, one subject area.

Three boxes carry the argument

Three boxes recur through the chapters, and each one earns its place because the argument keeps reaching for it.

- **The people part:** names who does the work and what the work demands of them, because a missing owner is this book's most common defect.
- **Metadata to capture:** records what a chapter leaves behind as documented fact, and those lists accumulate into the scorecard that chapter 10 hands you.
- **Regardless of stack:** states what stays true on any platform, which is the claim the book is willing to be tested on.

Here is the second one, doing its job.

Metadata to capture. One list, before you start. Write down the decisions your organization has already made about what its data means, and beside each one, note where that decision is written down today. Most teams fill the first column easily, then leave the second reading "in a meeting". That gap is the book.

Beyond the book

This book and the blog at deltavault.ai grew from each other. Several chapters started as posts, beginning with meaning before machinery, and each chapter links back to the posts it draws on. A post argues one point and nothing more, which makes it the thing to send a colleague who will not read sixty pages. Of those, the lakehouse posts sit closest to these chapters. And where a vendor's own page says a thing better than a paraphrase could, the chapter links that instead, starting with the Databricks description of medallion architecture that chapter 1 takes apart.

CHAPTER 1 OF 10

The promise and the gap

In this chapter

- What every lakehouse book and platform page actually documents, and where all of it begins.
- Three receipts that show the industry compressing the work that decides success.
- The bridges map: the four gaps between your sources and a gold layer people trust.
- Why modelers and engineers end up solving the same problem in different rooms.

Every platform makes you the same offer: land your data, and our AI does the rest. This book is about everything "the rest" assumes you already did.

The offer is not a trick. The engines are genuinely excellent, storage is cheap, the file formats are open, and AI now writes working pipeline code in minutes. As far as it goes, the promise holds. What this chapter examines is exactly how far it goes, and the work it quietly hands back to you.

What does every lakehouse book actually document?

Pick up any book about the lakehouse and you will learn a great deal: file formats and table formats, engines and clusters, transaction logs and time travel. Open any vendor's architecture page and you get the storage mechanics of that same picture. Somewhere on every one of those pages sits the same diagram: three boxes labeled bronze, silver, and gold, with confident arrows between them.

Now look at where all of that material begins. The book starts once the data is in the lake. The architecture page starts once the data is in the lake. The diagram's first box is data that has already landed. Everything before that moment, along with everything the arrows between the boxes quietly require, is treated as either finished or trivial.

None of it is wrong. The engines deserve their documentation, and you will need it. But a lakehouse program does not fail because someone misconfigured file sizes. It fails in the parts the shelf and the stack agree not to write about: agreeing what things mean, knowing what your sources actually contain, and deciding what shape the integrated middle should take. You do not have to take that on faith, because three receipts follow.

Watch the deciding work shrink to two pages

A sixty-page primer spends two pages on it. One widely distributed, vendor-sponsored cloud data warehousing ebook, the kind many data leaders keep on their desk, runs about sixty pages: delivery models, evaluation criteria, security, cost, and a six-step getting-started plan. Everything upstream of loading, the entire journey from your source systems to usable data, is compressed into two pages of advice about streamlining the data pipeline. The premise underneath is plain: value begins once your data is inside the platform.

The official implementation guide gives silver one line. Microsoft's implementation page for the medallion architecture in Fabric spends roughly 2,200 words on the pattern, and the silver layer, the place where your data supposedly becomes an integrated, trustworthy asset, gets this job description: "Fix errors, standardize formats, and remove duplicates." Everything after that is storage mechanics: Delta versus Parquet, shortcuts, file sizing, retention windows, and clustering choices. The page answers how to store a silver layer. It never answers how to design one.

The glossary names two methods and teaches neither. Databricks' medallion glossary says the silver layer's data is "matched, merged, conformed and cleansed ('just-enough')", then delivers its entire modeling guidance in two sentences: "From a data modeling perspective, the Silver Layer has more 3rd-Normal Form like data models. Data Vault-like, write-performant data models can be used in this layer." Two methods are named, and neither one is chosen, taught, or sequenced.

Read those verbs again: matched, merged, conformed. Matched on which keys, and agreed by whom? Merged how, when your sources disagree? Conformed to whose definitions, recorded where? Each verb is a project in its own right, and none of them gets a page.

The real work sits on the arrows, not in the boxes

Here is the medallion architecture diagram you have seen a hundred times, redrawn honestly. The three boxes shrink, because the boxes are the easy part now, and the arrows grow, because each arrow carries a question no engine answers for you.

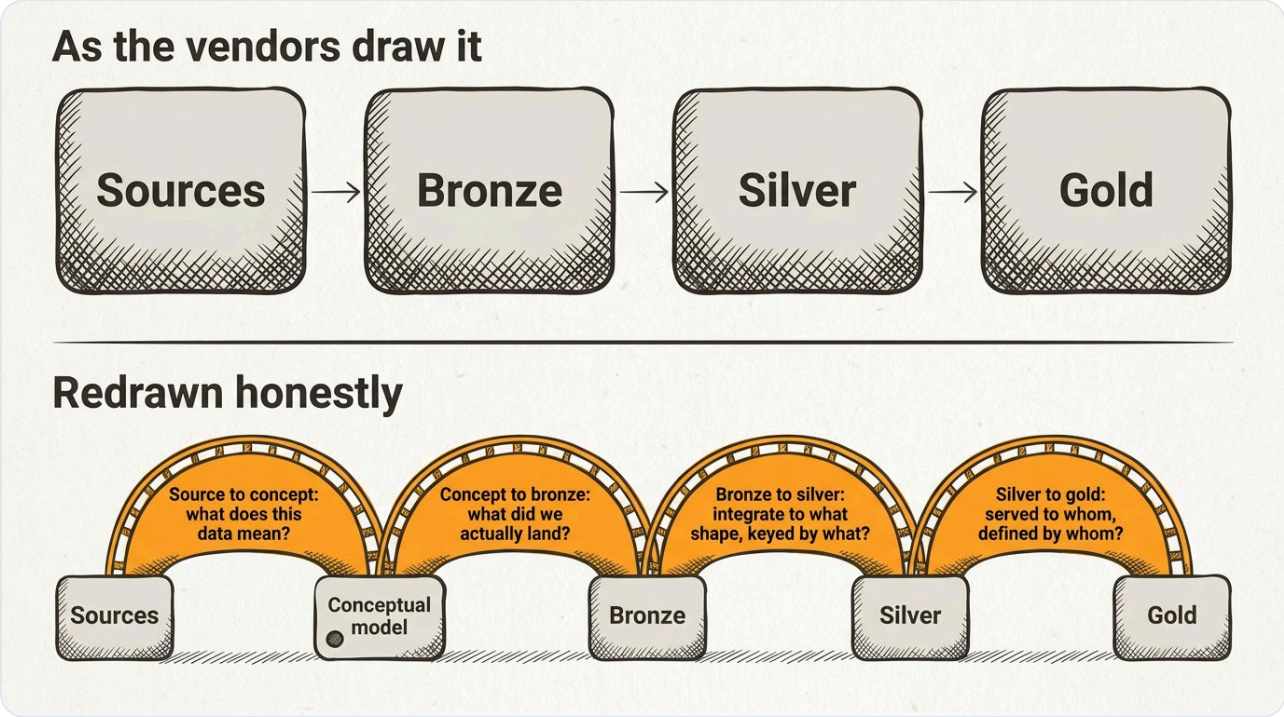

Figure 1.1: the bridges map. The boxes shrink, because the boxes are the easy part now, and the arrows grow, because each arrow carries a question no engine answers for you.

- **Source to concept: what does this data mean?** Before you make a single platform decision, someone has to establish the business language: the entities, the attributes, the relationships, the

names. This is not documentation written after the fact, it is the working vocabulary everything else maps to.

- **Concept to bronze: what did we actually land?** Sources drift, and their documentation rarely matches what the columns hold, so somebody has to know each system well enough to say what a field really contains, which keys really identify a customer, and where the quality problems live, all before that data gets treated as raw truth.
- **Bronze to silver: integrate to what shape, keyed by what?** This is the bridge where programs stall, because matching, merging, and conforming each demand what the verbs assume you already have: a target shape to conform to, agreed keys to match on, and a record of what you decided when the sources disagreed. That is design work, and it is exactly the work the official pages compress into a sentence.
- **Silver to gold: served to whom, defined by whom?** A metric is a business definition wearing a table. When nobody agreed on the definition back at the first bridge, gold becomes the place where every department discovers a different number.

The first three bridges get chapters of their own, and the fourth is why gold sits outside this book's scope. What they share is that none of them belongs to a platform, and that is the point of the map: you can change engines without touching a single bridge question, and you can answer every bridge question without choosing an engine.

The people part. Ask who owns each bridge in your organization today. Source to concept usually belongs to a modeler, if anyone holds it at all. Concept to bronze belongs to whoever wrote the ingestion job, while bronze to silver belongs, by default, to whichever engineer picked up the ticket. Silver to gold lands with the analyst under the most deadline pressure. That leaves four bridges with four accidental owners and no shared record of the decisions between them. The gap in your lakehouse is not a missing feature, it is a missing owner.

Two rooms, one problem

There is a reason the bridges go unowned, and it is older than the lakehouse. Data modelers and data engineers are solving the same problem in different rooms. In one room, modelers settle what a Customer is: the definition, the identifying keys, the relationships that matter. In the other, engineers load customer data through pipelines, schedules, and schemas. The model captures meaning but never touches the platform, and the pipelines deliver data nobody can trace back to an agreed definition.

So the model becomes a poster on the wall, and the warehouse becomes undocumented debt.

Nobody chose this split; the tools imposed it. Modeling tools export diagrams and documents, engineering tools consume code and configuration, and nothing carries a decision from the first room into the second. Every mapping from meaning to data gets re-derived in someone's head, at the keyboard, under deadline, unrecorded. The lakehouse did not create this divide. It raised the price of it, because now an AI sits in the second room, generating code at speed, with no way to see anything that was decided in the first.

A field story (a composite, not a single real company: assembled from the recurring pattern, with no real names or numbers). A retail company ships bronze in eight weeks and celebrates: forty source tables landing daily, dashboards of pipeline counts, green ticks everywhere. Eighteen months later, "silver" turns out to be a rename pass, the same forty tables with cleaner column names, three

competing customer identifiers, and a business that still reconciles revenue in spreadsheets because no two teams agree what an order is. Every metric that made the early demos sat on the source side: systems connected, loads per day, pipelines generated. Not one of them measured whether anything was integrated. The program did not fail at storage; it never started the integration.

Regardless of stack. Swap the platform in that story and nothing changes. On Databricks, on Snowflake, on Fabric, on anything else, the four bridge questions keep their exact names: what does this data mean, what did we actually land, integrate to what shape keyed by what, served to whom and defined by whom. Platform choice changes how you execute the answers. It has never once supplied them.

What this book promises

This is not another platform book, and it is not a modeling religion either. On the question of Data Vault versus third normal form versus dimensional-first versus one big table, the book stays deliberately neutral. It lays out what each shape optimizes for, gives you a decision framework grounded in your sources and your team, and then shows you the contract every shape demands regardless: agreed language, known sources, recorded mappings, explicit keys, captured metadata.

The arc follows those bridges. Chapters 3 through 5 cover the work that comes before the platform: speak before you store, know your sources, and treat mapping as a discipline with recorded decisions. Chapters 6 through 8 build the integrated middle, where you choose a shape without a religion, generate the integration layer from templates instead of handcrafting it, and capture the metadata contract that lets a template engine, a new hire, or an AI do the building. Chapter 9 then closes the trust loop with lineage and governance, and chapter 10 turns all of it into six steps you can start on Monday.

One thread runs through every chapter: the same captured decisions that make your integration layer buildable are exactly what AI needs to help you build it. The platforms are right that AI can do the rest. They are simply quiet about the fact that “the rest” runs on context, and the context is yours to create.

The platforms are ready and the engines are documented, so the missing work is yours. By the end of this book you will know exactly what that work is, who does it, and what to capture along the way.

Why the silver layer fails

In this chapter

- Why bronze always ships and silver always stalls.
- The medallion redrawn as the people each layer actually needs.
- The three-body problem: what a lakehouse does without a conceptual anchor.
- How work routes around missing decisions, and what that costs downstream.

Every stalled lakehouse program tells the same story in the same order. Bronze ships in weeks and the team celebrates: sources connected, tables landing daily, green ticks on the ingestion dashboard. Silver is scheduled next, and silver never quite arrives. It slips a sprint, then a quarter, then quietly shrinks until what ships under the name is not what the diagram promised. When the reckoning comes, the blame lands on the platform or on the engineers, and both verdicts are wrong. This chapter is about the mechanism that stalls the middle layer, because until you can name it, you will keep buying bigger platforms to fix a problem that was never technical.

Bronze is the only layer one discipline can finish alone

Count the people it takes to land a source table in bronze. A data engineer builds the ingestion. Someone who owns the source system explains what the extract actually emits and which field moves when a row changes. A platform engineer handles storage, identity, and compute. All three roles sit inside technology, all three are already on your organization chart, and all three are bookable this week.

Bronze asks nothing of you beyond that. There is no modeling to do, because the source system already made every structural decision, and no metadata to negotiate beyond what the source publishes. There is no meeting where two departments have to agree on anything. A team can land two hundred tables without asking anyone what a single one of them means, and because the work is pure technology, it is exactly what this generation of AI tooling accelerates best. That is why every tool ships source-to-bronze first, and why the early weeks feel so good: ingestion runs, row counts match, the dashboards glow. It feels like a stable era, and stable eras end.

Now count the people it takes to publish a governed revenue number in gold. You need a data modeler for grain and conformed dimensions, a steward who can say which definition is authoritative, and a business owner who can define the measure and defend it when departments disagree. Then you need an analytics engineer to implement it and a semantic model developer to expose it. Half of that list does not report to you, and most of them are paid for a different day job.

That asymmetry, not technical difficulty, decides your running order. Bronze goes first not because it is easiest, but because it is the only layer a single discipline can finish without asking anyone else for a calendar slot.

The diagram nobody draws

Every medallion architecture diagram you have seen is a diagram of technology: three boxes, arrows, and a vendor logo in each corner. Strip the technology out and draw something else instead, the people each layer actually needs.

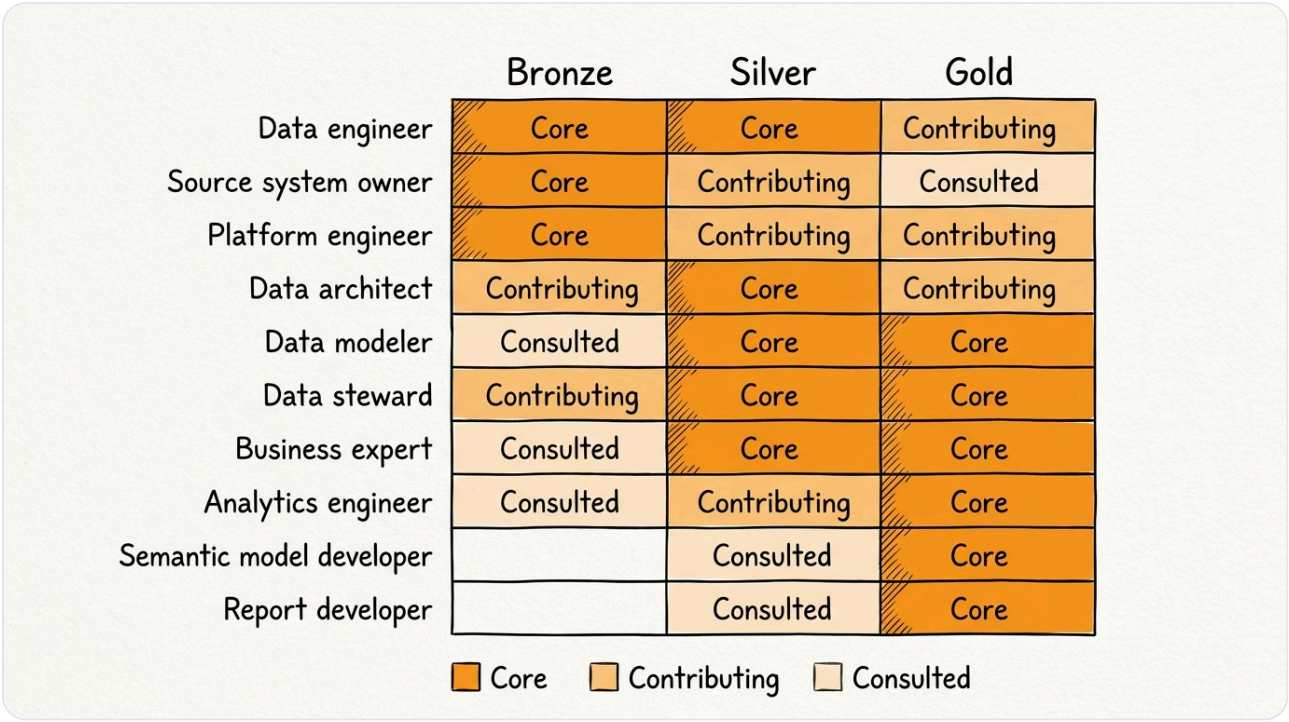

Figure 2.1: the involvement grid. Ten roles down the side, three layers across, weighted by how much of each person's time the layer consumes.

Read it by column, not by row. Bronze is a narrow band of engineering. Gold is wide, but it is predictable, because by the time a number gets published everyone already accepts that it needs a business owner. That leaves the middle column, and it is the one worth staring at, because it is nearly solid. Silver is the only layer where engineering, architecture, modeling, governance, and the business are all core at the same time. That is the diagram you never saw, and it is the one that predicts your delivery dates.

One staffing error hides at the top of the stack too. Ask most teams who owns gold and they name the reporting team, on the reasonable-sounding grounds that gold is where the dashboards read from. But gold is dimensional modeling: fact grain, conformed dimensions, a deliberate strategy for history. The reports live above it, on a semantic layer of measures and business-friendly names. Collapse those two into one box and you have quietly handed the dimensional modeling of your business to whoever built last quarter's dashboard.

Regardless of stack. The grid is the same grid on Databricks, on Snowflake, on Microsoft Fabric. No platform choice adds a steward to your program, and none of them frees an hour of your business expert's calendar.

Why does the silver layer stall when the code is easy?

The usual explanation for a thin silver layer is that the transformations are difficult. They are not. Deduplicating a table, conforming a code list, handling late-arriving records: these are well understood problems with well understood solutions, and your engineers can do every one of them. The Databricks glossary asks for data that is "matched, merged, conformed and cleansed", and every one of those verbs is mechanically easy and organizationally expensive.

What silver actually needs is a decision, and the people who can make it have the least available time. Somebody has to say that billing's account and support's login are the same customer. Somebody has to say which system wins when the two disagree, write that ruling down, and then hold the line the week two departments count differently. None of that is an engineering problem with a technical answer waiting to be found; these are business decisions, and the grid shows exactly who they belong to: the steward who is half of somebody's job, the business expert who gives you an hour every two weeks, the modeler shared across three programs.

Three suns and no anchor

There is a name for a system of massive bodies with nothing to anchor them. In Liu Cixin's novel The Three-Body Problem, a planet orbits three suns, and that single fact makes its sky unpredictable forever, because three bodies pulling on each other produce chaos no formula can tame. Build bronze, silver, and gold without an agreed model of the business and your lakehouse inherits the same physics.

The project was supposed to start before bronze. Somebody was meant to define what the business means by customer, order, and account: the conceptual model, the working vocabulary, the modelers' bridge from chapter 1. That work is the first thing cut, because bronze does not need it. Then silver arrives, and its job sounds mechanical, integrate and conform the data, until you ask the only question that matters: conform it to what? Without an agreed model there is no target shape, so each pipeline improvises its own. Bronze drags silver back toward the source structures, gold drags it toward whatever this quarter's dashboards demand, and every engineer's private guess about what a customer is adds another mass to the system.

The real lesson of the three-body problem is sensitivity, because tiny differences in starting conditions compound until prediction fails. A definition guessed slightly wrong in silver gets joined, aggregated, and re-aggregated until gold serves two revenue numbers to the same meeting, and nobody can say which one is right, because right was never written down. When the chaos becomes unbearable, data teams do what the novel's civilizations do: they rebuild. The second warehouse, the third lakehouse, better technology every time, the same missing model every time, and the same chaos wearing newer file formats. The industry calls that loop replatforming, but it is the same sky.

The rename pass

Here is how the stall looks from inside, because it never looks like a stop. The work routes around the missing decisions. Each pipeline improvises a local answer, because a local answer unblocks the ticket and a meeting does not. On the slide, the architecture still looks like a medallion. Underneath, silver has

quietly become a rename pass over bronze: the same source-shaped structures with friendlier names, and a patchwork of views stitched over the top to gloss over the agreement nobody made.

Name what that leaves you with, which is a replication service. You have copied your operational systems into a lakehouse with better file formats and a fresh coat of medallion vocabulary, and copying is not integrating. The cost lands the first time someone asks a question that spans business units. How many customers do we have, and which of them buy from both product lines? The platform has no answer, because the answer never lived in any system you copied.

None of this is a new disease. Bill Inmon defined a data warehouse in four words: subject-oriented, integrated, time-variant, non-volatile. Three of the four still get built as a matter of course, and integrated is the one the industry quietly let go. Kimball's conformed dimensions tell the same story from the other direction. Those disciplines did not stop being right, they stopped being convenient, and the tooling drifted toward the convenient side, where everything can be inferred from sources and nothing has to be agreed. The industry has started saying this out loud. Practitioner essays now argue that the medallion is a communication device rather than an engineering blueprint, that it is not data modeling at all, and even Databricks' own summit hosted a session asking what is wrong with it. So the diagnosis is spreading. What is missing is a method, and the method is the rest of this book.

Staff the handshakes, record the decisions

The reflex fix is structural: stand up a bronze team, a silver team, and a gold team. That rebuilds the silo the medallion was supposed to dissolve, and now tickets get thrown over two new walls instead of one. Staff the boundaries instead. There are three of them, and each has its own guest list.

- **Source to bronze.** The source owner and the data engineer agree on schema, keys, change capture, freshness, and what happens when the source changes shape without telling anyone.
- **Bronze to silver.** The engineer, the modeler, the steward, and the business expert agree on entities, business keys, source precedence, duplicate handling, and history. This is the highest-interaction boundary in the architecture, and it is also the one most programs never schedule.
- **Silver to gold.** The modeler, the analytics engineer, and the business owner agree on grain, dimensions, measures, and how much history the business actually needs. This handshake runs backwards as often as forwards. "I need customer segment as it was on the order date" is a gold requirement that only silver can satisfy, and with no return path it gets solved inside a report, once, badly, and then solved again next quarter by somebody else.

Metadata to capture. Every decision the bronze-to-silver handshake produces is a recordable fact rather than tribal knowledge: which source records count as the same customer and why, which system wins when two disagree, how duplicates are resolved, how much history is kept and at what grain. Written down where the work can read them, those rulings become the thing silver conforms to. Left in the meeting, they evaporate, and the next pipeline guesses again. Chapter 5 turns this into a working discipline.

The people part. You cannot hire the grid, and the goal was never to hire it. The goal is to spend the scarce hours you get, the steward's afternoon and the business expert's hour, on the decisions only those people can make, and to stop burning those hours on plumbing. Read that way, the involvement grid stops being a hiring plan and becomes a meeting schedule.

So the silver layer fails for one reason wearing many costumes: decisions nobody was scheduled to make get made anyway, by default, at the keyboard, and unrecorded. The fix is not a bigger platform, and it is not more heroic engineering. It is agreeing the business language before you store anything, knowing what your sources actually hold, and recording the mappings between the two, which are chapters 3, 4, and 5. First on the agenda is the question this chapter kept circling: which system wins when two of them disagree about a customer.

Speak before you store

In this chapter

- Why the conceptual model is the first deliverable, on any stack.
- What a usable model contains, what it never contains, and the rules that keep it honest.
- Meetings, documents, and sources: where the model actually comes from.
- Naming as ordered, executable rules rather than a wall poster.

Chapter 2 ended on the question that stalls the middle of every lakehouse: conform it to what? The official guidance never answers. Microsoft's implementation page for the medallion architecture in Fabric asks silver to "fix errors, standardize formats, and remove duplicates" without ever naming the thing it is standardizing toward. This chapter supplies that missing target. Before anything is stored, you agree the business language and write it down: the entities, the attributes, the relationships, and the names. The industry calls that artifact a conceptual model, plenty of teams call it the business model, and both mean the same thing. This book says conceptual model, though the label matters less than the timing. It comes first, it lives outside every platform, and everything else in this book maps into it.

So resist the reflex to file this under documentation. Documentation records a system that already exists, while the conceptual model is what your architecture is made from. The silver layer conforms to it, the mappings in chapter 5 terminate in it, and the metadata contract in chapter 8 serializes it.

Can AI infer your business model from the schemas?

The tempting shortcut is to let the machines write the model. Point an AI at your source schemas, ask it for a business model, and you will get one. What comes back is your source schema with better labels, and that is the only thing it could be, because the schema was the only thing you gave it to read.

That is not a failure of the model. The schema was never trying to explain your business in the first place. A relational schema tells you how an application chose to persist something, not what that something means. You may have three tables with customer in the name: one is a billing account, one is a login, and one is what your sales team actually means by the word. The fact your model needs, which of the three your finance team counts as a customer, was never recorded in any of them. It lives in a glossary, in a requirements document, and in a two-year argument. The semantics that matter most were never columns, and no engine can extract what was never persisted.

So the model has to be spoken before it can be stored. It comes out of people, which sounds expensive until you remember the grid from chapter 2: your business expert's scarce hours get spent either way, either in a scheduled meeting that produces a recorded model, or at the keyboard, one improvised pipeline decision at a time.

A usable model is smaller than you think

Teams put off conceptual modeling because they picture a cathedral: a wall-sized diagram, a year of workshops, a binder nobody opens. A usable model is nothing of the sort. It is modest, boring, and decisive, and it contains six kinds of fact.

- **Entities with definitions.** Each named business concept carries a sentence or two the business is willing to sign. Not the textbook definition of a customer, but the one your organization argues about in meetings.
- **Attributes with business names.** These are the facts the business records about each entity, named in business language. Because the definition sits on the attribute rather than on any column, a renamed source column costs you a mapping, not a meaning.
- **Relationships that read as sentences.** Customer places Order. Order contains Product. Read each one aloud in its direction and it has to be a sentence your business would actually say, because a relationship that fails that test is a misunderstanding you have caught early.
- **Synonyms, recorded rather than resolved.** Real businesses have synonyms, so record which name is the preferred term and keep the rest as registered aliases instead of forcing a winner. When two departments use different words for the same thing, the model already knows.
- **Categories.** Each concept says what kind of thing it is, either an event, something that happens, or a concept, something that simply is, along with which question it answers: who, what, where, when, why, or how. Categories are what let a template, a reviewer, or an AI treat an Order differently from a Customer without being told twice.
- **Business keys.** These record how the business tells one occurrence from another, such as an order number or a membership number. Just as valuable is the honest flag that no such key exists yet.

Notice what is not on the list: data types, indexes, load schedules, history strategies, table designs, platform names. Those belong to later chapters. Let one in and you couple your business language to a system, and the language dies with the system.

Three rules keep it honest

A model with no discipline degrades into noun soup: every capitalized word in the workshop notes becomes an entity, and order status arrives as a thing in its own right. Three rules prevent that, and because they are written down, they are also what stop a machine guessing at the answer later.

- **The sorting rule.** Anything you learn about a concept is identity, relationship, or context, and never two of those at once. Identity is how the business tells occurrences apart; relationship is a verb phrase pointing at another concept; context is a descriptive fact recorded about the concept alone. Run the sort in that order every time, and order status lands as a fact about the Order instead of a concept of its own.
- **The happening test.** A happening earns concept status only when the business names it and identifies it separately: a Payment tracked by a payment reference, a Flight tracked by a flight number. A happening nobody identifies is the relationship between the concepts it involves, and statuses such as placed, confirmed, and shipped are states, never concepts.
- **Evidence for keys.** A business key has to be quoted by the business or clearly implied by what it says. When no identifier is named, flag the gap rather than manufacturing a key out of the

concept's name. A flagged gap is a work item; an invented key is a defect wearing a badge.

These rules are mechanical on purpose. A person in a workshop and a machine reading a transcript can both apply them and reach the same answer, which is what turns model review from a matter of taste into a checklist.

None of this discipline is new. Lawrence Corr's BEAM method runs people-first modelstorming aimed at the dimensional gold layer, John Giles has argued for years that Data Vault work must start from a conceptual model, and Ensemble Logical Modeling, set out in Remco Broekmans's From Stories to Solutions, supplies the identity, relationship, and context separation this chapter leans on. Take the discipline from whichever school your team already reads.

The meeting is the model

The most productive modeling session your team ran this week did not happen in a modeling tool. It happened in a meeting, where a real question finally got settled and the model walked out in five people's heads. The transcript sat there the whole time, filed as minutes, when it was source material.

To see those rules run by actual people, watch a working session. The worked examples in this book come from a demo dataset: Maluga, an outdoor-living retailer whose garden webshop is called Willibald. In one nine-minute meeting, a domain expert, an architect, and an engineer settle the corner of the model where the confusion lives, namely the two lines between Customer and Club Partner that every new reader wants to merge. The ruling is a model in a sentence: one line says Jane is a member of the club, the other says Jane is the person the club put forward to deal with the company. Membership versus role, two relationships answering two different questions. The distinction is load-bearing, because orders placed by a club's contact person are attributed back to the club, so merging the lines sends the revenue report quietly wrong for months.

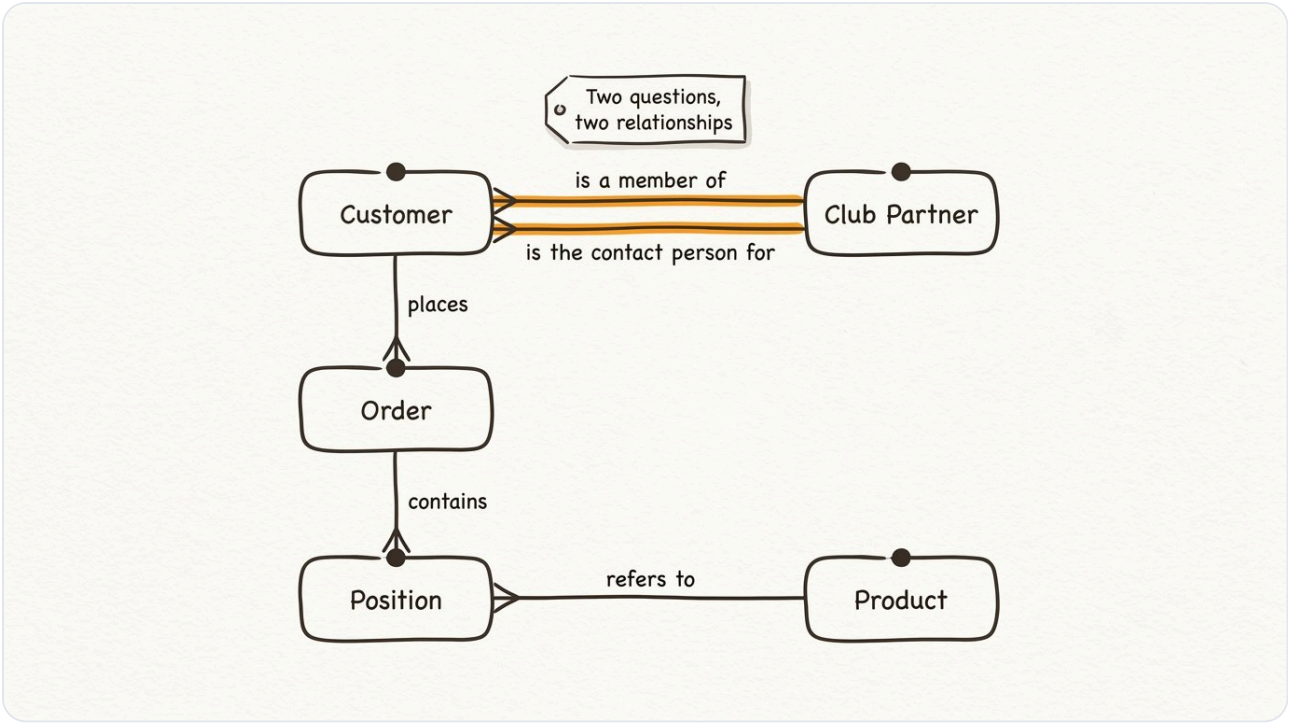

Figure 3.1: the two lines every new reader wants to merge. One says Jane is a member of the club, the other says Jane is the person the club put forward to deal with the company.

The same meeting shows the key discipline holding under pressure. The club record carries three discount fields nobody can define, and instead of inventing names and letting the guess harden into fact, the room logs an open question for the business. That is the flag-the-gap rule, executed by people who have never read a modeling book.

Once your team knows the conversation will drive a conceptual model, you run the conversation differently. You push for precision while the people who hold it are still in the room: is that a role or a membership, and which address are we actually talking about? The transcript stops being minutes and becomes the specification the meeting was always trying to write.

The people part. The meeting needs three chairs filled: a business expert who owns the meaning, someone who knows what the data actually holds, and someone holding the method, running the sorting rule and the happening test in real time. This is where chapter 2's scarce silver hours go, and an hour here saves a sprint of keyboard guessing later.

Documents feed it, sources witness it

The meeting is not the only input. Your organization has already written much of its language down in glossaries, requirements documents, vendor data dictionaries, and domain notes. Treat those artifacts as governed inputs: versioned, and re-read every time the model is worked on, rather than as text pasted once into a prompt. A definition your team argued its way to deserves to be read directly, not guessed at from a column name.

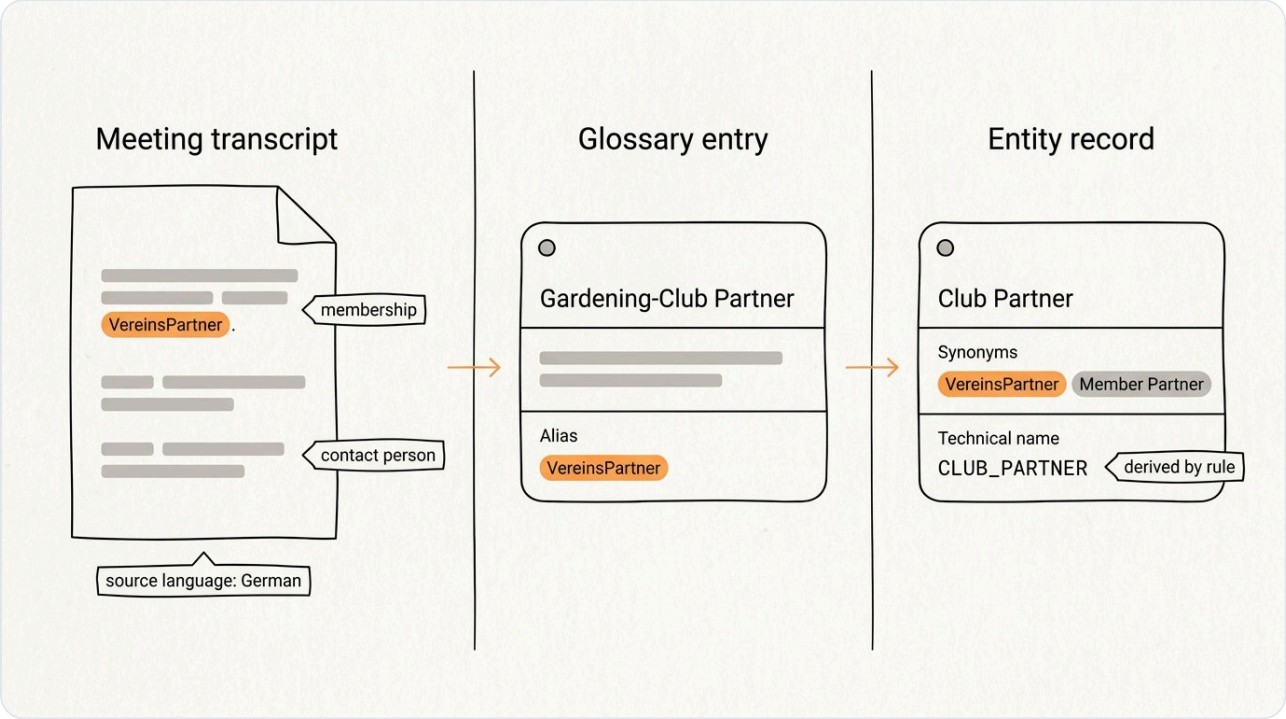

Figure 3.2: the same fact travelling. Panel one, the meeting transcript: club membership distinguished from the club's contact person, in a source whose language is German. Panel two, the glossary: a Gardening-Club Partner entry defined in business language, with the source term VereinsPartner recorded as an alias. Panel three, the entity record: Club Partner, synonyms VereinsPartner and Member Partner registered rather than erased, technical name CLUB_PARTNER derived by rule.

Sources come last, and they arrive as witnesses. Your systems have true things to say about which concepts show up in practice and which identifiers actually get used. Cross-examine them, map what survives into the model, and remember that a witness does not hold the pen. The glossary entry in the figure shows the pattern in miniature: the German source term is kept as a synonym while the concept it names is defined in the business's preferred language. The source contributed evidence, and the business kept the pen.

A naming standard nobody can execute is a poster

Naming is part of the model, not a cosmetic pass at the end. Every concept and attribute ends up carrying at least two names: the business name people say in meetings and the technical name the platform carries, along with the registered synonyms. The naming standard is the bridge between the first two, and most naming standards fail one simple test, which is that they cannot be executed.

An executable standard is a short stack of ordered rules: which term is preferred when synonyms exist, which abbreviations are allowed (a finite approved list, not a habit), and the derivation rules applied in a fixed order, abbreviations first, then casing. The order matters, because a standard whose order lives in two places will eventually hold two versions of it, and at that point your documentation describes a name no generator ever produced. Define the order once and apply it everywhere. Club Partner then becomes CLUB_PARTNER by rule, not by whoever typed it first.

The test is mechanical execution. Hand the standard and ten new attribute names to someone who joined yesterday, or to a machine, and if the technical names come back without a question, you have a

standard. If any step requires judgment, what you have is a poster, and posters do not survive the columns arriving in chapter 4.

The cheapest deliverable you will ever ship

Metadata to capture. Everything this chapter produces is a recordable fact: each entity, its definition, and who signed it, each attribute and its business name, each synonym and the preferred term, each category, each relationship written as a readable sentence, each business key or its flagged absence, and the naming rules in the order they apply. Write them down where the work can read them, because chapter 5's mappings and chapter 8's metadata contract consume exactly this list.

Regardless of stack. Read back through this chapter and count the platform decisions. There are none. No engine was chosen, no file format debated, no license bought. A conceptual model costs a handful of well-run meetings, which makes it the cheapest deliverable in your program, and the longest-lived one. Platforms come and go underneath it, while the agreed definition of a customer outlives them all.

That model is half of the agreement your silver layer needs. The other half is knowing what your sources actually hold, because sources drift, surprise, and occasionally lie. Chapter 4 puts the witnesses on the stand.

Know your sources

In this chapter

- The source register: connections, capabilities, and what each source cannot tell you.
- Metadata import as a reviewed change, because schema drift never stops.
- Profiling and key discovery: observations, held apart from decisions.
- Reading a source honestly: witness, not author.

Chapter 3 closed by promising to put the witnesses on the stand, and this chapter is the cross-examination. The conceptual model records what your business means. Now you find out what your systems actually hold, and the distance between those two answers is exactly the work the silver layer exists to do.

Every data project starts the same way, whether anyone dignifies it with a name or not: by proving you can read the source. Can you connect? Can you list the tables? Can you pull the columns and their types? And, just as important, what can this particular source never tell you? Teams treat all of that as throat-clearing before the real work begins. It is the real work, and the teams that skip writing it down get to do it again every sprint.

A register, not a pile of connection strings

No two sources answer questions the same way. A live database connection can hand you tables, columns, types, primary keys, foreign keys, and views. A nightly file drop gives you column headers and whatever you can infer from the rows. Some sources carry table and column descriptions, but most arrive with those fields empty. Every connection, then, has a capability profile: what it can tell you, what it cannot, and what you will have to learn some other way.

The source register is where that profile lives, one entry per source: how you connect, who owns the system, when extracts arrive, which capabilities the connection supports, and the limits you have already discovered. It is deliberately unglamorous, and a team without tooling still keeps one, because the alternative is rediscovery. The engineer who learned in March that the webshop extract carries dates but no timestamps leaves in June, and in July someone burns two days learning it again. An unwritten source limit is a recurring cost booked as a surprise, every time.

An inventory that survives a rename

Once you can read a source, what you read goes into a catalog: every table and every column, held as an inventory. That inventory only earns its keep if other artifacts can point at it. Mappings reference these columns and rulings attach to them, so every entry needs an identifier that stays stable while everything around it changes.

Names are the wrong identifier for that job. Columns get renamed, tables move between schemas, and the same logical source turns up under different names in development and production. A name-keyed

spreadsheet does not object when any of that happens; every reference into it simply breaks, and it breaks without an error message. So the inventory keys on stable identifiers, and the names hang off them as facts that are free to change. Identifiers carry identity, names are only labels. The idea returns at full strength in chapter 9, when the metadata itself goes under version control; it enters the book here because the inventory is the first artifact the others point at.

Import is a diff you approve

A source inventory would be a one-time chore if sources held still, and they do not. Columns appear, types widen, a table splits in two, and none of it waits for your review meeting. Drift is not an incident; it is the weather.

That is why a catalog refresh is a staged, reviewable change rather than a reload. The import lands as a proposal: these tables are new, these columns changed type, these disappeared. A person reads the diff, approves it, and the catalog moves forward with its history intact. The alternative, truncate-and-reload, fails twice at once. It destroys everything you attached to the inventory, every description, mapping, and ruling keyed to those entries, and it waves drift straight through, importing every surprise as silently as the source produced it. A dropped column should be a decision you make with the diff in front of you, not something you discover downstream when a report goes blank.

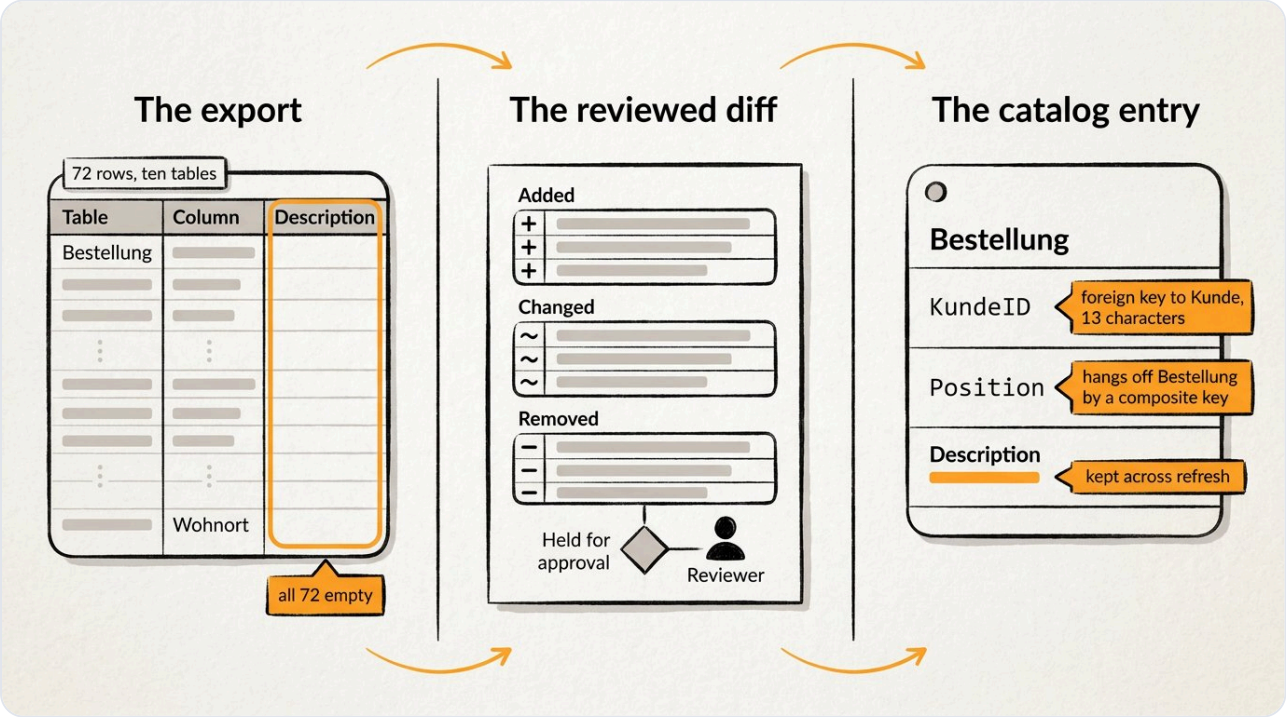

Figure 4.1: the export, the reviewed diff, and the catalog entry. Seventy-two rows arrive honest about structure and empty about meaning; the review is where meaning gets attached, and where it stays.

The worked example runs on Willibald, the garden webshop from chapter 3. Its metadata arrives as exactly the export in the figure: 72 rows describing ten tables, from `Bestellung` (the order header) down to `Wohnort` (a customer's places of residence over time). The file is honest about structure, and it can prove that `Bestellung.KundeID` is a thirteen-character foreign key to `Kunde`, and that `Position` hangs off `Bestellung` by a composite key. It is just as honest about meaning, though in the other direction: the export has a description field for every table and every column, and all 72 of

them arrive empty. The source can tell you precisely how the order data is shaped and nothing at all about what an order is.

What does data profiling tell you that the schema will not?

The inventory says what the source claims about itself. Data profiling checks those claims against the data: null rates, distinct counts, ranges, and top values per column, with candidate keys proposed where values actually run unique and candidate relationships proposed where value sets overlap. None of this is new discipline. Kimball and Caserta gave profiling a named place in the warehouse back in 2004, in their book on extracting, transforming, and loading warehouse data, and it says something about the industry's attention span that a twenty-year-old book is still the last mainstream text to treat it as one. It is also where the model meets the source for the first time.

Profiling earns its keep on the gap between declared and true. The Willibald data ships with exactly the failures the discipline exists to catch: product rows duplicated in the extract, a table whose business key simply is not there, and order numbers from an earlier load period reused in a later one, the kind of key collision that never shows up in a schema and always shows up in a join. A profile run finds each of these as a measured fact, with numbers attached.

What you then do with a measured fact is what decides whether your catalog stays trustworthy, and the essential split is that observations are not decisions. A profile run observing that `Position.PosID` never repeats is evidence; declaring the pair of order and position the key is a ruling. Evidence accumulates automatically and goes stale harmlessly, while rulings are agreed by people and carry authority. Keep the two apart, in separate places with separate lifecycles, and record the promotion: who promoted which observation into which ruling, and when. A catalog that files observations directly as facts has automated its own gullibility, because the profile of one bad extract becomes policy without anyone deciding anything.

The witness tells the truth, but only about itself

There is a deeper limit underneath all of this, and it is the one chapter 3 flagged. A relational schema tells you how an application chose to persist something, not what that something means: your database is not your business. Surrogate keys, junction tables, status codes, audit columns are implementation decisions, made under a deadline, often by a vendor who never met your business. Read a schema as testimony about the application and it is excellent evidence. Read it as testimony about your business and you are quoting a witness on questions it never saw.

Your conceptual model will keep colliding with that limit. The model says a customer has a delivery address, while the source has a customer table, an address table, and a type column carrying values like Shipping. The clean noun on your diagram turns out to be a join and a filter that nobody has written down, and writing it down is precisely the mapping work of chapter 5. Neither side is wrong here. They are simply answering different questions.

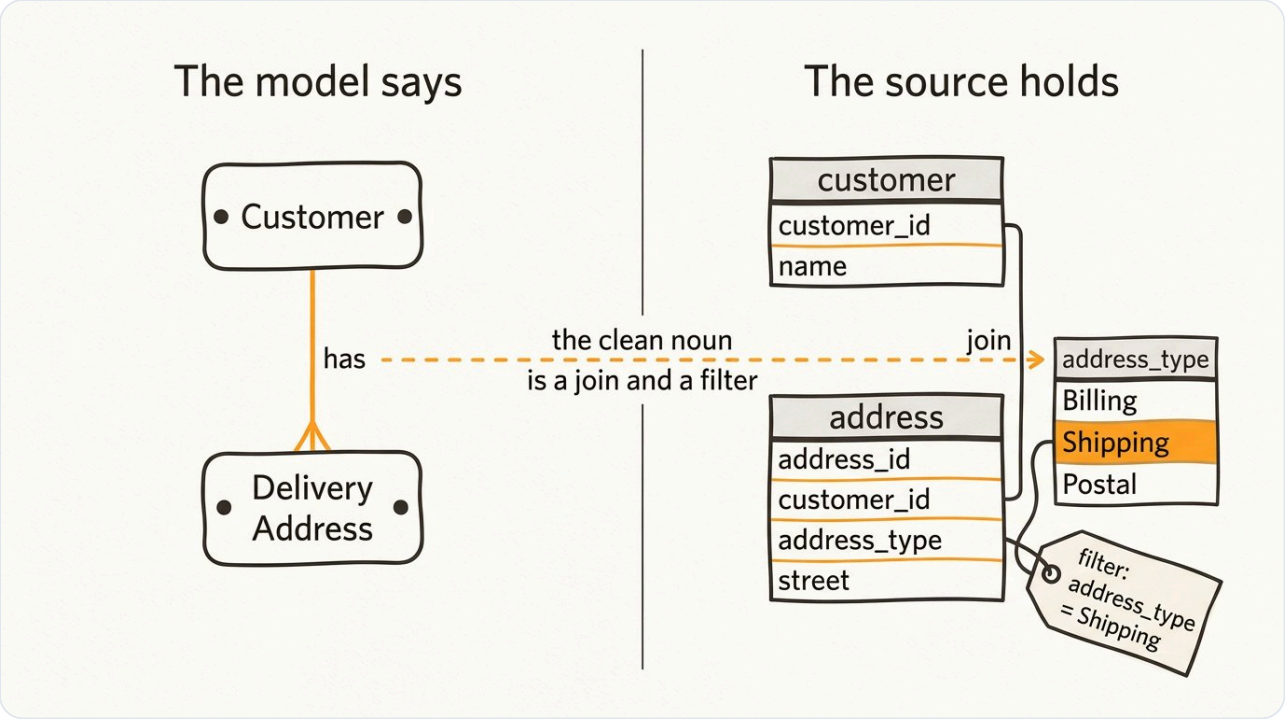

Figure 4.2: the clean noun on your diagram turns out to be a join and a filter.

So the last source-understanding instrument is not a query at all. It is a conversation with the people who run the process, and the Willibald team shows what one is worth. In a twenty-five minute walk through the order tables, field by field, with the person who operates the webshop in the room, the model gains facts no profile run could reach. `Wunschdatum` is the delivery date the customer requested, not a promise the business made, and early delivery is not automatically good, because plants can arrive before the customer is ready for them. The order discount reads as an amount deducted from the total, but the source carries no discount-type column, so the room records the reading and files a validation question instead of letting the guess harden. No order status is stored anywhere either: open and delivered turn out to be derived states, read off the positions and deliveries. The roadshow table is named like an order table and is nothing of the kind, since each row is one product line of a point-of-sale transaction with header values repeated on every row. And nothing in either schema says the webshop customer and the roadshow buyer are the same person; the rule that matches them through their payment card lives in a documented business rule, written by people, invisible to any scan of the source.

Every one of those facts changes the model or the pipeline, and not one of them was a column.

The people part. That meeting only works if the source system owner is in the room, and they are the role data teams schedule last. Chapter 2's grid puts them at the core of bronze and contributing at silver, so book their hours when the source enters the register, not on the day their absence blocks a mapping. The interview itself is a repeatable format: one table at a time, every field named and questioned, with someone writing the answers into the catalog rather than into minutes.

What the stand produced

Metadata to capture. Every output of this chapter is recordable: the register entry per source with its capability profile and limits, the table and column inventory keyed on stable identifiers, each approved

import diff, the observed profiles, the candidate keys and relationships still held as candidates, and the promotion record that moves an observation to a ruling. Taken together, they form the evidence file for every source you touch.

Regardless of stack. Nothing above named a platform. Your register can be a wiki page, your diff can be a pull request, and your profile can be a script plus a table of results. Tooling makes each step cheaper and repeatable, but the discipline is what makes any of them worth doing.

The conceptual model from chapter 3 states what the business means, and the evidence file from this chapter states what the sources actually hold, on the record, with the gaps marked. Chapter 5 is where the two get joined through the mapping that ties each source column to the meaning it feeds. Without the evidence, that mapping is guessing with confidence behind it. With the evidence, the mapping becomes the most valuable document your team produces.

The mapping is the deliverable

In this chapter

- The document every project runs on and nobody has.
- The anatomy of a mapping decision: recorded, reviewable, owned.
- Families: one decision carried across every copy of a column.
- Bridges for the gaps a mapping cannot close, with the lineage kept.

Everything so far has been preparation. Chapter 3 wrote down what your business means; chapter 4 put on record what your sources hold. This chapter joins the two, and everything that follows consumes the join.

What is the document every project runs on?

Every data project runs on a document that does not exist: the one that says which source table means which business entity, which column carries which attribute, and why. It is where the model meets the source and one of them has to give. You already know where that knowledge actually lives: in the head of the engineer who did the last integration, in a spreadsheet half-updated since the workshop that produced it, or nowhere at all. Everyone downstream pays for the vacancy:

- The modeler cannot tell how much of the model is real.
- The engineer re-derives the translation table by table.
- The next project starts the archaeology over.

The document is not missing because nobody thought of it. Kimball and Caserta's 2004 book gave source-to-target mapping a whole chapter of discipline. There is stranger evidence that the document is inevitable: ask a language model to write a requirements document for a source database, as one of the teams behind this book's demo datasets did, and it appends a source-to-business name mapping on its own initiative, for traceability, unprompted. A statistical model trained on the industry's collective writing knows this document has to exist. The industry still does not keep it, because keeping it as a document is exactly what fails. A document goes stale the day after the workshop. What you need is a mapping that lives where the metadata lives.

A model that cannot bind is a poster

Be precise about why the spreadsheet fails, because the usual diagnosis blames people and the fault is not theirs. Your modelers hold what the business means. Your engineers hold what the sources are. Neither side is wrong, and someone still has to translate. The mistake is calling that a people problem. It is a problem of medium: a model that exists as a picture cannot meet a schema, so every comparison happens by hand, and every manual comparison is another place where meaning leaks out. A conceptual model that cannot bind to a discovered schema is a poster.

The two preceding chapters dissolved that problem. Chapter 3 made the conceptual model recorded metadata: entities, attributes, and definitions, each with a stable identity. Chapter 4 did the same for the source estate, giving you an inventory built on identifiers that survive a rename. Once both sides are made of the same material, the correspondence between them stops being a drawing and becomes a join. The mapping is that join, written down: this table is that entity, this column carries that attribute.

Anatomy of a mapping decision

A mapping worth trusting is not a line on a diagram. It is a small record with a fixed shape:

- the physical thing
- the business thing it means
- a confidence in the pairing
- who decided
- when
- why

The why is what every spreadsheet drops and what the next reader needs most. A pairing that looks wrong with no rationale gets silently undone; a pairing that looks wrong with its reasoning attached starts the right argument.

The record is bidirectional. Stand on a source table and ask what it means, or stand on a business entity and ask where it is held: both readings return the same answer because there is only one record. That also settles a scheduling fight that wastes months, because neither side waits for the other. If the modelers arrived first, mapping a table is a lookup into the model they built. If the engineers arrived first, the mapping raises a placeholder entity to be adopted into the model later. Whichever side gets there first, the other's work attaches when it arrives.

The people part. Treat the review as the unit of work, not the match. A proposed pairing, however it was produced, is a question addressed to a person, and disagreement is a first-class outcome: accept it, correct it, or skip it with the reason recorded. All three are decisions, and a recorded no is worth nearly as much as a recorded yes, because it stops the same wrong suggestion returning next quarter. The signature matters just as much. A mapping has a decider the way chapter 4's rulings had one, and the right decider is whoever can answer for the meaning, usually the modeler and the source system owner together, not whoever happened to be running the tool.

One decision, every copy

Source data rarely arrives once. It arrives as a family: a landing copy, a staging copy, a persistent staging copy, all descended from one source table and renamed a little at every hop. Map each copy independently and you multiply the work by the depth of your pipeline. You also create the conditions for the copies to disagree, so the same column means one thing in landing and another in staging, and the contradiction sits unnoticed until an audit finds it.

The discipline is to map the meaning once and let the decision travel the chain. Treat the family as a recorded structure rather than a naming coincidence, and one rule governs all of it: one pairing rule for the family, or none.

Ownership is what makes that cascade safe. A machine-made mapping belongs to the machinery, so it moves with its source, updates when the source's mapping changes, and disappears when the pairing is cleared. A human-made mapping belongs to the person who made it, recorded with a decider and never silently overwritten by a later pass, however confident that pass may be. Without the distinction, a cascade is a hazard: the most careful decision in the catalog can be flattened by the next automated run. With it, automation fills every gap a person has not claimed, and stops at every gap a person has.

When the gap will not close

Some gaps a mapping cannot close. The model says Customer; the source offers three tables, a join nobody recorded, and not a foreign key in sight. This is where projects reach for paint and ship a diagram claiming the source fits the model. The discipline says otherwise: build the bridge, and keep the lineage.

The garden webshop from chapters 3 and 4 gives you the honest version. Willibald's order data arrives from two systems that share a concept and nothing else. The webshop delivers orders as a numerically keyed header table and position table. The roadshow file delivers one row per product line with the header values repeated, and its order identifiers are text with a prefix deliberately unlike the webshop's, so the integrated key has to carry the source system alongside the value. The two systems cannot even agree on who the customer is. Only about a fifth of roadshow buyers give a customer number.

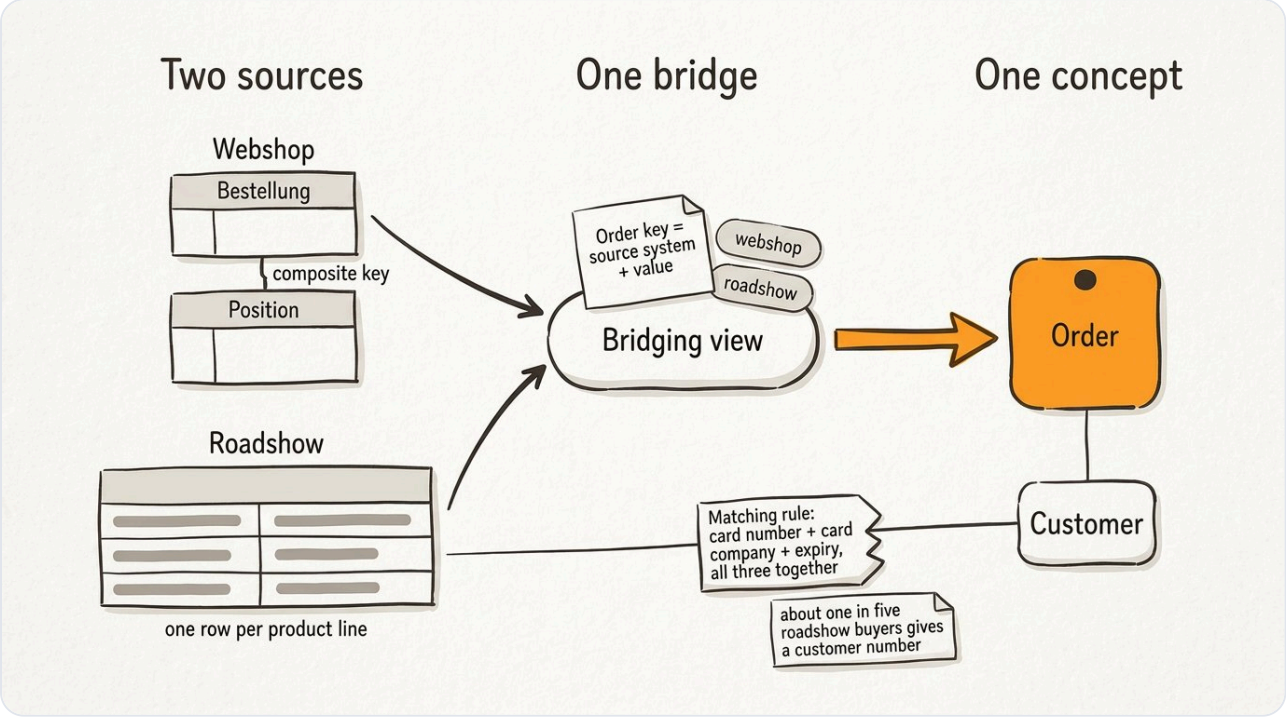

Figure 5.1: two systems that share a concept and nothing else, and the bridge that was built between them.

> **Note:** The documented rule for matching the rest to webshop customers runs through their payment card: card number, card company, and expiry together, never one field alone.

No single mapping records any of that, and pretending otherwise is how catalogs lose their readers' trust. What closes the gap is a built structure. You recover the undeclared joins, define the bridging view that gives the sources the model's shape, and encode the matching rule where it runs rather than on a slide. The mapping discipline still governs the bridge:

- The built structure maps to the entity it realizes.
- Every projected column records exactly which source columns feed it, computed expressions included.
- The lineage is metadata committed alongside everything else, not a report generated after the fact.

A bridge with its lineage kept is a recorded decision like any other. A bridge without it is the next generation's archaeology.

The human is the loop

A mapping program stands or falls on where you put the people. Put them at the end, approving a finished batch of guesses under deadline, and you have a rubber stamp with better branding. Put them at the front, judging each pairing while it is still a proposal, and the work compounds: every accepted mapping becomes evidence for the next, because a table whose sibling is already mapped is no longer a stranger. Your decisions do not exit the process. They become it.

That compounding is what makes the mapping your governance backbone rather than documentation about it. Classify the Email Address attribute once, in the model, and every column mapped to it inherits the classification across every landing table, staging copy, and downstream structure the attribute touches. Policy attaches to meaning, and the mapping is how meaning reaches the physical estate. Chapter 9 takes this up.

The most valuable document your team produces

Metadata to capture. The mapping record holds five kinds of entry:

- Each mapping carries its physical thing, its business thing, a confidence level, the decider, the date, and the rationale.
- Each rejected pairing carries the reason for rejection.
- The family chain records which mappings are machine-owned and which are human-owned.
- Each bridging structure carries its column-level lineage.
- The classifications that flow through the mappings are recorded alongside them.

Regardless of stack. The mapping asks the same questions with or without tooling: what does this table mean, who says so, and what happens when the source changes. Tooling changes who keeps the answers honest, because a cascade will not forget the staging copy and a review queue will not lose the rationale. Still, a team with a spreadsheet and the discipline to record deciders and reasons is ahead of a team with neither.

Here is the payoff, and it is the sharpest test in the book. Integration claims are usually unfalsifiable: a slide saying customer data is integrated cannot be checked. A mapping can. Real integration leaves a record, and the record is a mapping: point to the entity, list the sources bound to it, read who decided

and why. With that in hand, you are ready for the question the industry loves to argue about, which is what shape the integrated data should take. Chapter 6 takes it up with the one answer nobody markets: it depends, and here is how to decide.

Choose an integration shape without a religion

In this chapter

- Data Vault, third normal form, dimensional-first, and one big table, each surveyed fairly enough that its advocates would nod.
- The decision factors that actually separate the shapes.
- What the lakehouse changed about the old arguments.
- The obligations no shape excuses.

Chapter 5 left you holding the record that proves integration: the mapping. The next question is what shape the integrated data should take, and the industry treats that question as a matter of faith. This chapter treats it as a decision, because that is all it is.

Consider the entire official silver-layer modeling guidance from the largest lakehouse platform: "From a data modeling perspective, the Silver Layer has more 3rd-Normal Form like data models. Data Vault-like, write-performant data models can be used in this layer." That is Databricks' medallion glossary quoted in full, two methods named in two sentences, neither chosen and neither taught. So the vendors will not decide this for you, and the religions will decide it for the wrong reasons. The truth sits in between: the choice is real, it is smaller than the arguments make it, and it excuses you from nothing.

Which integration shape should you choose?

Every shape in this survey works, has production estates behind it, and has advocates who are not fools. So the honest question is never which shape is correct. It is what each one optimizes for, and what it charges you in return. The Data Vault posts on the blog go deeper into the shape this chapter can only give a paragraph.

Dimensional-first optimizes for query usability now. Ralph Kimball's stars put facts and dimensions in front of analysts in a shape they can read without a translator, and Lawrence Corr's BEAM showed how to design them with business people in the room rather than after they leave. The cost is hidden in where the integration went: it is deferred into conformed dimensions, so the hardest cross-system agreement arrives late, mart by mart, under delivery pressure. A star estate that never does the conformance work becomes a collection of departmental answers that agree with nobody.

Third normal form optimizes for integrity and one version of the truth. Bill Inmon's atomic warehouse stores every fact once, non-redundantly, integrated before anyone consumes it. That integrity creates friction at both ends. Analysts pay a join tax to answer ordinary questions, and change amplifies, because a new source or a revised relationship reshapes structures that everything downstream already depends on. It also demands enterprise modeling stamina up front, which is exactly the appetite most organizations claim and few fund.

Data Vault optimizes for multi-source agility and audit. Dan Linstedt's hubs, links, and satellites separate business keys from relationships from context, so a second source lands beside the first without reshaping what already exists, and full history is kept by default. Here the cost is scale of a

different kind: every source table becomes several vault tables, the pattern discipline is unforgiving, and the people who hold it are the scarcest skill pool of the four.

One big table optimizes for simplicity now. Wide, denormalized tables are cheap to store on columnar formats and cheap to query, and there is no modeling gate between landing data and asking questions of it. The cost arrives on a delay. The semantics live in transformation code instead of in a model, every consumer re-derives them slightly differently, and governance drifts because there is no agreed structure for it to attach to. It is the fastest route to a first answer and the slowest route to a shared one.

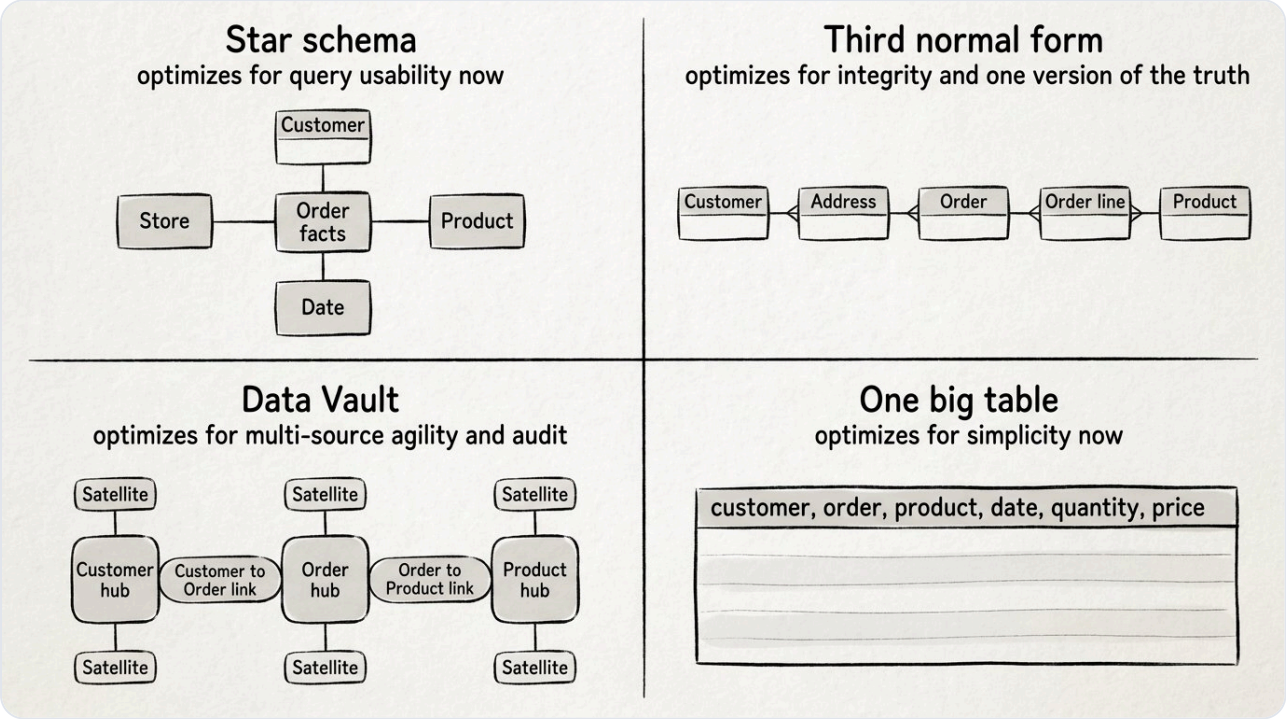

Figure 6.1: the same subject in four shapes. Every one of them works; every one of them charges you differently.

The factors decide, not the faith

Strip the advocacy away and a small set of factors does the actual separating. Weigh them for your estate, in writing, and the decision largely makes itself.

- **Change tolerance.** How often do new sources arrive, and how often do the existing ones change shape? Frequent arrivals favor the shape built for addition, while a stable estate can afford a shape that integrates harder up front.
- **Audit and history obligations.** If regulation or dispute resolution requires you to show what you knew and when, a shape that keeps history by default beats one where history is a per-table design decision someone might skip.
- **Team skills.** The best shape your team cannot staff is worse than the second-best shape it can. That constraint is real, and it is also the factor most often allowed to decide alone, which is precisely what it should not do.
- **Query and compute economics.** Who pays, the load or the read? Join-heavy shapes tax every consumer, wide shapes tax change and governance, and your platform's pricing decides how

much each tax costs.
- **Latency.** The shorter the path from landing to consumption has to be, the fewer transformation stages the shape can afford.
- **Appetite for modeling.** Some organizations will fund the agreement work and others will not, whatever they said at kickoff. Choose a shape whose demands match the appetite you have actually observed, not the one in the slide deck.

Weigh those factors across the practitioner discourse of the last few years and a pattern emerges: comparisons that start as arguments keep ending as hybrids, with an integration shape in silver (Data Vault or third normal form) and dimensional serving in gold. The books teach each shape alone, so the hybrid that practitioners keep converging on is a process nobody's book walks through. That process is what chapters 3 through 9 of this one are.

Figure 6.2: the decision table. Four shapes weighed on the factors that actually differ, so the choice is argued rather than inherited: the decision is yours, and the obligations are not optional.

Factor	Dimensional-first	Third normal form	Data Vault	One big table
Optimizes for	Query usability now	Integrity, one version	Multi-source agility, audit	Simplicity, first answer fast
New source arrives	Conform another dimension	Reshape affected structures	Add tables beside existing ones	Rebuild the wide table
History	Per-dimension design choice	Design choice	Kept by default	Whatever the pipeline kept
Skills market	Widely held	Classical, thinning	Scarcest of the four	None required at first
Read-side cost	Lowest	Join tax	Join tax, absorbed by views	Lowest until drift
Modeling appetite required	Moderate, rising at conformance	High, up front	High, sustained	None, which is the trap

The people part. The architect proposes the shape, and the proposal should read like this chapter: factors weighed, trade-offs named, no religion. The rulings underneath the shape, though, which business keys the estate counts by and what history depth the business owes, belong to the stewards and business experts who can answer for meaning, exactly as chapter 3's rulings and chapter 5's mappings did. Watch, too, for the quiet anti-pattern of a shape chosen purely by hiring history. "We know stars, so stars" is a real factor wearing the costume of a decision. Put team skills in the table as one row, and make the other rows show up to the same meeting.

The lakehouse moved the goalposts

Whole shelves of shape advocacy were written on assumptions a columnar lakehouse quietly deleted. Storage width was expensive on row stores, so shapes split tables narrow. On columnar formats a repeated value collapses to nearly nothing and a query reads only the columns it asks for, so width became cheap. Joins were the counter-argument, and generated read views absorbed them: current views, end-dated views, and point-in-time reconstructions, all produced from metadata rather than

hand-written, so consumers stopped seeing the physical layout. Even change detection, the load mechanic that justified so much structural cleverness, moved into the platform as declarative change data capture that keys, sequences, and versions rows from metadata you already hold.

What survives of the old physics is row count, which remains a real cost worth designing for, along with one hazard that got sharper. In a vault, every satellite boundary is a hash boundary, so moving a column across it makes the histories on both sides incomparable. That makes a physical split the least reversible decision in the model, and the conceptual model is the thing most likely to change. Take those two facts together and you should split less, and never casually.

Every split and structure in your current shape that was justified by row-store physics therefore needs re-deciding, because its justification no longer exists. What remains as a split criterion is business meaning. Sensitivity is a boundary, because a regulatory erasure should touch a small table rather than rewrite years of history. Governance status is a boundary too: one working modeling standard keeps governed, mapped columns in a separate satellite from ungoverned, unmapped ones, so the table boundary itself records a governance fact. A change-rate difference of orders of magnitude is a boundary when profiling measures it, not when a rule of thumb merely feels it. "We always split by source system" is not a boundary. It is an inherited habit whose reasons retired.

One vocabulary correction rescues an entire misreading of the most argued-about shape: raw describes the data, never the structure. A raw vault holds data exactly as the sources delivered it, integrated by business keys the business ruled on. Mechanically decomposing source tables into vault shapes skips the ruling and keeps the ritual.

The obligations no shape excuses

Here is what the religions get wrong, and they get it wrong in both directions: advocates imply their shape provides these things automatically, while skeptics imply that skipping the shape skips the work. Both are false. Every shape in the table carries the same five obligations, and each one is a decision a person makes rather than a property a structure has.

- **Business keys someone ruled on.** A hub is a business key, and a business key is a decision about what the business counts: one Customer, however many systems store one. A conformed dimension needs the identical ruling, and so does a third normal form entity, and so does the grain of a wide table. No shape makes the decision for you; some shapes only make its absence visible sooner.
- **Identity that survives renames.** Chapter 4's inventory of stable identifiers has to hold all the way through the integration layer, because otherwise every physical rename silently orphans your history and your lineage.
- **History with a declared depth.** How far back, at what grain, decided by whom. "The platform keeps versions" is a mechanism, not a policy.
- **The mapping.** Chapter 5's record of what means what is the substance of integration in every shape. A vault without those rulings is source-shaped storage wearing vault vocabulary, and a star without conformance is a dashboard cache.
- **Lineage.** Where each value came from, kept as metadata rather than reconstructed by archaeology. Chapter 9 takes this up.

Once the obligations are met, write the decision down: the shape you chose, the factors you weighed, the date, and the people who decided. One page, kept where the next engineer will look. Estates outlive their architects, and an integration shape with no recorded rationale gets relitigated by every new hire who arrives fluent in a different one.

Metadata to capture. Start with the decision record itself, covering shape, factors weighed, date, and deciders. Underneath it sit the business-key rulings and the declared history depth. Capture every split boundary along with the reason it exists, so the next modeler can tell a meaning boundary from a retired habit.

Regardless of stack. The factor list survives every platform, and only the economics row changes its weights. A different engine reprices joins, width, and history, and touches nothing else in the table. That is exactly why choosing a shape is not a platform decision, and why no platform migration will ever do it for you.

Whichever column of the table you choose, this book supports the choice, because the point was never the pattern. The point is what happens after the choice: an integration layer generated from the model you have been building since chapter 3, rather than hand-written until it drifts away from it. A shape you picked with a written rationale, expressed as structures no human hand-drifts, is the difference between architecture and archaeology. Generating it is chapter 7.

Templates, not handcrafted tables

In this chapter

- Why the integration layer must be generated, never hand-written.
- The five things a template framework consists of, whatever engine renders it.
- Conventions as metadata: naming, system columns, hashes, and bindings declared once.
- What proving a pattern actually requires, and why a green sample is not a green run.

Chapter 6 closed on a promise: whichever shape you chose, the integration layer that expresses it should be generated from the model you have been building since chapter 3. This chapter makes that promise concrete. “Use templates” is advice everyone nods at and almost nobody can evaluate, so the work here is to define the standard behind the advice. By the end you will know what a template framework actually consists of, and you will be able to judge any generation tool against that standard, including the scripts your own team wrote.

The most repetitive code you will ever ship

An integration layer is hundreds of structurally identical objects that differ only in their metadata. Every landing table is the source table plus the same two audit columns. Every staging load is the same change-detection pattern pointed at a different table. Every satellite hangs off its hub by the same key treatment. The shape you chose in chapter 6 does not change this; it only changes which pattern repeats.

Hand-writing that layer produces drift, and the mechanism is worth naming precisely. Notebook twelve starts as a copy of notebook eleven with two edits, one of which gets forgotten. A column rename lands in the table that raised the ticket and misses the two downstream tables nobody remembered. Worse, every hand-written exception is a decision nobody recorded, so the one load that handles deletes differently stays invisible until it disagrees with the other forty at three in the morning. Multiply that by five hundred objects and the estate’s real design lives in the diffs between files, where no reviewer will ever find it.

Generation inverts the failure mode. When every object is rendered from the catalog, the catalog cannot drift from the code, because the code came from the catalog. The mapping you built in chapter 5 stops being documentation that describes the pipelines and becomes the source the pipelines are compiled from. What comes out the other side is a derived artifact, like a compiled binary: you can read it and you can deploy it, but you do not edit it, because the thing you edit is the model.

What is in a template framework, whatever the engine?

The word “templates” undersells what a working framework contains. Whether the engine is a vendor product, a set of dbt macros, a template engine that is yours, or the generator script your team wrote three years ago, the same five parts have to be there, and whichever one is missing will find you.

- **The patterns.** One template per object kind and load style: a landing table, a staging load, a satellite merge. A pattern is the design decision "how we do staging" written down once, rather than expressed forty slightly different ways across forty files.
- **The conventions the patterns read.** Naming rules, system-column definitions, hash definitions, data-type mappings. These are inputs to the patterns, not text buried inside them, and the next section explains why that placement decides everything else.
- **The bindings.** Which pattern applies to which object, decided by metadata such as this table's layer, its entity type, and its load style, never by folder discipline. A binding you can query is a rule, while a folder convention is only a hope.
- **The declared context.** Exactly what a pattern is allowed to know about the world when it renders. This is the contract that makes patterns portable and testable, and it carries enough weight to be the whole subject of chapter 8.
- **The proof.** A way to render any fragment against real metadata, and a way to confirm the platform accepts the result. A sample is not proof; a test is.

Ask those five questions of any tool and the demonstrations sort themselves quickly. Plenty of products will show you patterns. Far fewer can show you the conventions as editable metadata, the bindings as rules, the context as a contract, and the proof as something you can run against your own catalog this afternoon.

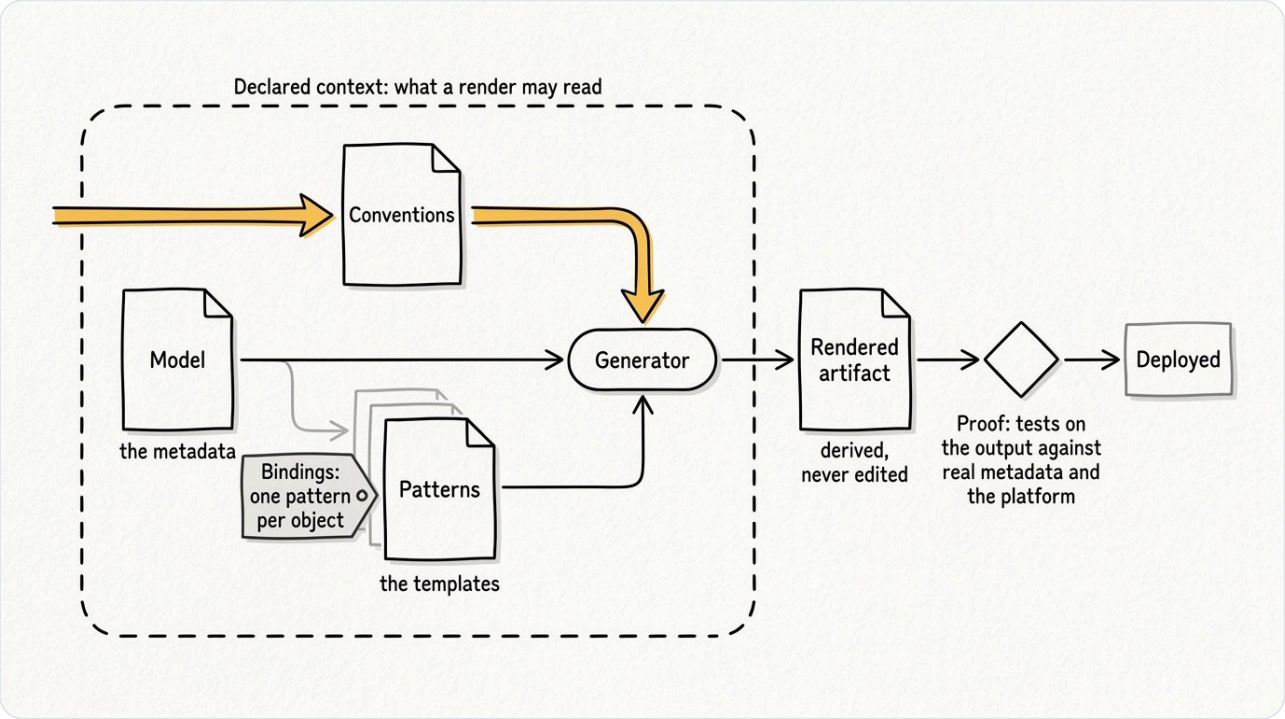

Figure 7.1: the five parts on the pipeline. The conventions live in metadata the patterns read, never in the patterns themselves.

Conventions are metadata, not code

Here is the placement decision that separates a framework from a pile of scripts: the conventions live in metadata the patterns read, never in the patterns themselves.

Take naming. A naming convention stored as metadata is a pattern file a few lines long: a schema pattern built from tokens like the source system, a table pattern like the schema name joined to the table name, and from those few lines every generated name in the estate falls out. Change one line and every name moves together, consistently, in the next render. Store the same convention as code and you get the same string-concatenation logic pasted into every template that names anything, so changing it becomes an archaeology project.

The same rule governs system columns, and here the stakes are higher because the failure is silent. Every layer adds columns the source never had: a load timestamp, a record source, a hash for change comparison, effective dates. A framework declares each of these once, in a catalog, with its role stated as a fact: this column is the load timestamp, this one is the hash difference. The alternative, which real tools actually ship, is inferring a column's role by recognizing its transformation code, and that inference breaks the day someone customizes a transformation. Wrap the load timestamp in a time-zone conversion and the string no longer matches, the column's role evaporates, and nothing errors. Roles are decisions. Store them as decisions, not as patterns to be guessed back out of code.

Figure 7.2: one pattern, three generated names. The source table Kunde, from the garden webshop of chapters 3 to 5, arrives with 12 columns; each layer's name and additions come from declared conventions, not from anyone's typing.

	Generated name	Columns	What this layer adds, by declared role
Source	`dbo.Kunde`	12	The source's own columns, exactly as delivered
Landing	`lnd_dbo_Kunde`	14	Load timestamp, record source
Staging	`stg_dbo_Kunde`	15	Hash difference
Persistent staging	`psa_dbo_Kunde`	18	Effective from, effective to, current flag

Behind the persistent-staging name sit two tokens in a profile file: the schema pattern resolves the source system, and the table pattern joins the schema to the table name. That is the entire convention, and it is smaller than this paragraph. Fifty source tables produce one hundred and fifty generated names from those lines, which makes renaming the estate an edit to the file rather than a campaign.

The people part. The template author is a role, and in most teams nobody holds it, so the patterns end up being whatever the last consultant left behind. Name the owner. Then notice what generation does to change management: a pattern edit that touches five hundred objects is a decision with five hundred objects' worth of consequence, so it deserves the same review a schema change gets. Who approves it belongs in chapter 3's decision rights, not in a merge nobody read. Retiring a pattern demands the same discipline, because a framework must be able to say what still depends on the old one. Retire a pattern that strands even one object and you have quietly converted that object back to hand-maintained code.

Proof or drift

Every generation setup demos beautifully, because a demo renders the vendor's sample. What separates a framework from a demo is what you can prove against your own metadata.

The first level of proof is render stability: the pattern produces the same output for the same input, usually held in place by snapshot tests. That is worth having, and it is also radically oversold, because a

stable render proves only that the template did not change. It says nothing about whether the platform accepts the result. Generated code that the render suite blesses can still fail the day it meets a reserved word, a dialect quirk, or a type your mapping never covered. A green offline suite is not a green run, and a framework that cannot close that gap is asking you to discover platform rejections in production.

The second level is the one to insist on: every fragment of a pattern must be testable in isolation against real metadata, meaning your tables, your columns, and your ugly legacy names. A fictional sample context is engineered to render cleanly, and that is precisely its flaw, because a deliberately tidy fixture can never surface your gaps. The pattern that renders the vendor's fictional Customer table perfectly may still fall over on your forty-column source with the reserved word in position three, and only a render against your catalog will say so before the platform does.

So the acceptance test for any framework, vendor-bought or homegrown, is a five-minute exercise: point one pattern at your worst real table and render it. If the framework cannot do that, its proof story is a sample, and a sample is marketing.

Own your patterns

One more question remains, and it is the one most evaluations skip: whose opinion does the generated code express, and what happens when you disagree?

Platforms are moving faster than any vendor's template library can follow. New load syntax, new table features, and new orchestration primitives arrive on the platform's release cycle, not your vendor's. A fixed, sealed template set is therefore out of date by the day you adopt it, and every month after that you are waiting on someone else's release notes to use capability you already pay your platform for. So the framework question is not whether the shipped patterns are good. It is whether you can open them, customize the one block you disagree with, and still receive the vendor's next improvement to everything you did not touch. Fork the whole file and you own its maintenance forever; change nothing and you ship someone else's opinion. The workable middle is surgical: override the one section, and keep tracking upstream for the rest. Applied that way, generation becomes acceleration that produces your opinion rather than the vendor's.

The wider tooling landscape makes the same point from the opposite direction. The nearest thing the modern stack has to a prescriptive process is dbt Labs' project-structure guide, which is genuinely disciplined but is entirely folder-and-SQL convention inside one tool. Writing on metadata-driven generation does go deeper, yet almost all of it is vendor authored and describes one product's way. That is exactly why you need the five questions in tool-independent form: patterns, conventions, bindings, context, proof. They apply unchanged to dbt macros, to a hand-rolled generator, and to any vendor engine, and they are the difference between evaluating a framework and admiring a demo.

Metadata to capture. The naming rules as pattern files. The system-column catalog with each column's declared role. The hash definitions. The data-type mappings per platform. The pattern bindings, queryable, never inferred from folders. And the customization points: which blocks your organization overrode, and why, so the next engineer can tell your opinion from the vendor's.

Regardless of stack. Nothing in this chapter named an engine, because none of it depends on one. The five framework questions interrogate a homegrown script exactly as hard as they interrogate a product, and a script that answers all five is a better framework than a product that answers two.

Templates close the gap chapter 6 left open: the shape you chose now compiles from the model instead of drifting away from it. In doing so, they raise the question they cannot answer alone. A pattern renders from what it is allowed to read, and everything in this chapter quietly assumed that what it reads is complete, correct, and agreed. So what exactly does the pattern get to know? That contract, the declared context, is chapter 8.

The metadata contract for AI

In this chapter

- Why in-platform AI cannot see what was never captured.
- Context as a governed setting, not a prompt pasted into a chat window.
- The contract, enumerated: seven kinds of metadata generation needs, each phrased as a question to ask any vendor.
- Receipts: prompts under change control, and runs with records.

Chapter 7 ended on a question templates cannot answer for themselves: a pattern renders from what it is allowed to read, so what exactly should it be allowed to read? This chapter writes the answer down as a metadata contract, and that contract reaches well beyond templates. Every generator you will meet produces its output from what it was given to read and nothing else, whether that is the assistant built into your platform, the template engine from chapter 7, or the new hire who starts Monday. Enumerate what the reading list must include and you hold two useful things at once: a specification for the metadata your team should be capturing, and an audit list for every "our AI does it" claim a vendor puts in front of you, including the one writing this book.

The AI cannot see what was never captured

Chapter 1 opened on the offer every platform makes: land your data, and our AI does the rest. The assistants behind that offer are real, and inside their lane they are genuinely good. The lane is the point. An in-platform assistant reads what the platform holds, and what the platform holds is the physical estate: table names, column names, data types, key constraints, and whatever descriptions somebody once remembered to write.

The vendors do not dispute this. As of September 2026, the documentation for the assistants on Databricks, Snowflake, and Microsoft Fabric each says plainly that answer quality depends on expressive names, good descriptions, and the curated context you supply. Each platform then ships a place to author that context: instructions and example queries on one, a hand-written semantic model on another, per-model AI instructions on the third. Give the industry its due, because that is a concession to this book's argument: the model was never the bottleneck, and everyone now agrees the context is. Look closely at where those context surfaces sit, though. Every one of them describes a single platform's physical estate, in that platform's own format, readable by that platform's assistant, authored after the data landed. The right idea has arrived, one tool at a time, at the wrong altitude.

Consider what happens while the deepest context available is the physical estate. Point the assistant at your source schemas, ask it for a business model, and what comes back is your source schema with better labels. That is the only thing it could produce, because the schema was the only thing it was given to read. Ask it to integrate two systems and it cannot, because integration is an agreement between systems and the assistant met your systems one at a time. Nothing in either schema records that the customer number in your sales system and the account identifier in your billing platform

describe the same person. That fact is a decision somebody has to make, and reading either system harder will never surface it.

That is the quiet dependency under the pitch. The AI cannot see what was never captured, and what was never captured is exactly the work of the last five chapters: what your business means, what your sources actually hold, how the two map together, what shape the integrated middle takes, and which conventions govern the build.

Why is business context not an AI prompt?

The workaround everybody reaches for first is the chat window. You paste in the glossary excerpt, then the naming rules, then the warning about the three customer tables, and finally you ask the question. It works once, and that is its whole problem. Pasted context is unversioned, unreviewed, and gone tomorrow. Nobody approved it, nobody can compare Tuesday's paste against Monday's, and when a colleague runs the same task with slightly different text, they get a slightly different answer. That is not governing the AI's input. That is privatizing it, one chat session at a time.

Treat context instead the way you treat any setting that matters: captured once, in your own words, reviewed like any other change, and read by every run rather than retyped for each one. Definitions handled that way compound, which is why an afternoon spent sharpening what your organization means by a customer is the cheapest AI improvement you will ever buy. It lifts every proposal and every review that follows it. A vague frame gives you plausible answers about somebody else's business, while a sharp frame gives you proposals that already speak your language.

The contract, enumerated

So what must generation be able to read? Seven kinds of metadata, and five of them you have already met as chapters. Each item below closes with the question to take into your next vendor demo, and each one is a strand of how we use AI rather than a prompt to be typed.

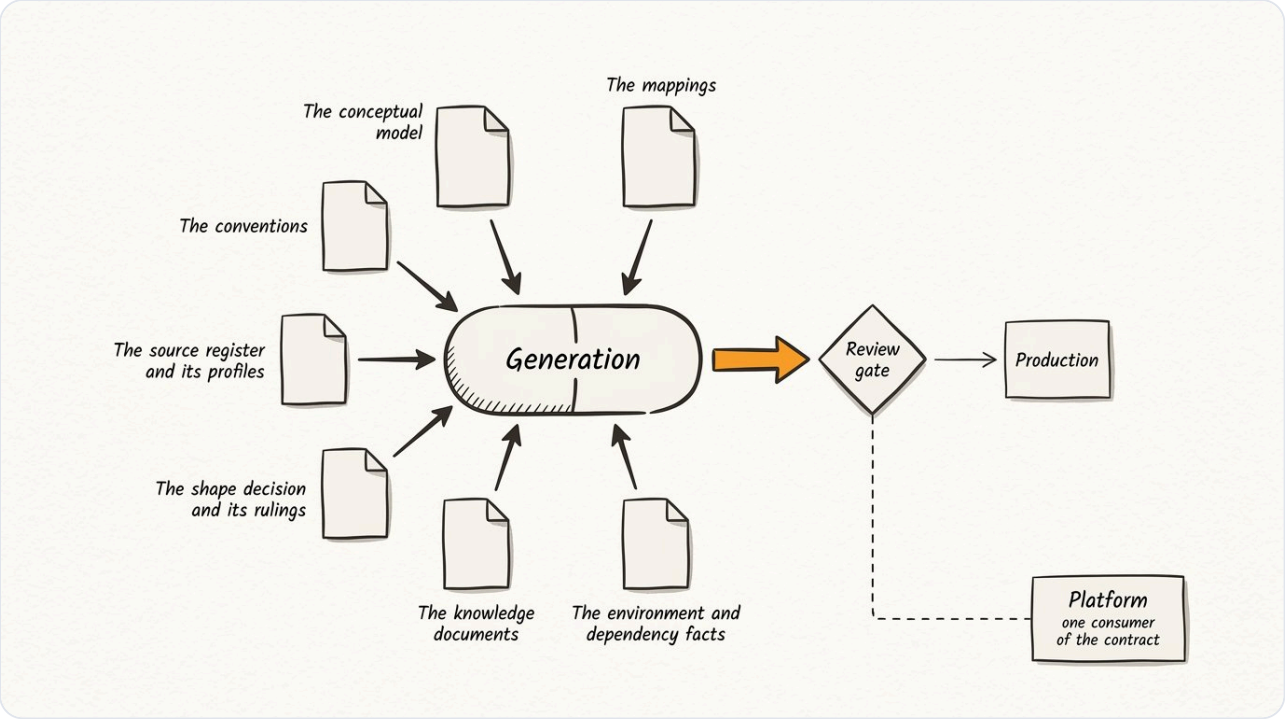

Figure 8.1: the contract, enumerated. Seven inputs, one generation step, a review gate, and the platform downstream of all of it.

- **The conceptual model.** The entities, attributes, definitions, categories, and relationships you built in chapter 3, with their approvals attached. This is what separates a conceptual model from a schema with better labels: what your organization means, expressed in vocabulary a domain expert recognizes, decided by people with the standing to decide. Where does the generator read what my business means by a customer, and who approved that definition?

- **The mappings.** Chapter 5's deliverable: which source column feeds which business attribute, with each pairing carrying its decider, its rationale, and its history. Without the mappings, a generator can reach your sources or your meaning, but never both at once. Can the generator read which column maps to which attribute, who decided each pairing, and which columns nobody has mapped yet?

- **The conventions.** Chapter 7's placement rule, made enforceable: naming rules with their resolution order, system columns with declared roles, hash rules, and the type mappings for each platform, all stored as data the generator reads rather than logic sealed inside it. Are my conventions inputs the generator reads, or opinions compiled into its code?

- **The source register and its profiles.** Chapter 4's observations promoted to decisions: which systems exist, who owns each one, how each delivers its data, and the accepted profile facts about what that data actually does. Does generation know how my sources behave, or only how they are declared?

- **The shape decision and its rulings.** Chapter 6's choice, written down: which integration shape the middle takes, along with the rulings that keep it consistent, including every exception you allowed and the reason you allowed it. Where is the shape decision recorded, and how does the generator learn about the exceptions?

- **The knowledge documents.** The standards, dictionaries, requirements, and meeting transcripts your organization already wrote, extracted and addressable so that a run reads the relevant passage in your words rather than a summary of it. Can I hand over the documents my team already maintains, and will the generator read what they say or a paraphrase?
- **The environment and dependency facts.** The unglamorous item that breaks builds the moment it goes missing: which layers exist, which targets receive which artifacts, and what must already exist before each object loads. Does the generator know which layer it is writing for, on which platform, and in what order?

Score yourself against that list today. For most teams the honest result is a few items captured, a few scattered across wiki pages and people's heads, and at least one that exists nowhere at all. That is not a failing grade; it is the backlog.

Self-sufficiency is the test

How do you know a contract is complete? Borrow the sharpest test template engineering has produced: if an author, human or machine, cannot produce the artifact from the declared context alone, the contract is incomplete, and the gap gets filled by guessing. Chapter 7 showed the miniature version of this failure, a generator that recognizes a column's role by matching its transformation code and then breaks silently the day someone wraps that transformation in a time-zone conversion. The general rule deserves stating on its own: sniffing and inference are contract debt. A generator that quietly reads beyond its declared context is not being resourceful, it is borrowing facts nobody agreed to supply, and every borrowed fact fails without an error message on the day it moves.

Regardless of stack. The contract is identical whatever sits in the generation seat. A language model, a template engine, and a contractor hired for the quarter all need the same seven items, and all three fail the same way without them, by producing something plausible built on guesses. That makes the new-hire framing an honest self-audit: could a competent stranger produce a correct artifact from what your team has actually written down? If the answer is no, the machine cannot either, and unlike the stranger, the machine will not ask follow-up questions on its way to being wrong.

Receipts

If generation is part of your method, it needs the discipline you give any production process, and that discipline produces two artifacts.

The first is the prompt. The instructions a generator runs are where its opinions live, so they belong under change control like any other source file: versioned, readable, testable before publication, and different from yesterday only because somebody decided to change them. If you cannot read the instructions your generator runs, you cannot review its opinions, and you are shipping them anyway.

The second is the run record. Every generation run should leave behind what was asked, what context was resolved for it, what came back, and who approved the result. Three weeks later, when somebody asks why the model proposes a concept nobody remembers discussing, that record is the difference between an answer and a shrug. It is also what makes machine-written changes governable at all,

because a proposal that passes the same permission checks and lands in the same audit log as a human edit means "what did the AI change" has exactly the same answer as "what did your colleague change".

The people part. Neither artifact helps if nobody tends the contract, and the contract does not curate itself. Owning the definitions, the conventions, and the choice of which documents count is steward and architect work, and it is continuous rather than a one-off setup. The review seat is where that work compounds, because a decision made on one proposal becomes context the next run reads. Teams that treat review as a formality get a generator that stays exactly as wrong as it was on day one, while teams that treat it as curation get a generator that improves every week without a single model upgrade.

The contract is buildable this quarter

Notice what is absent from the seven items: not one of them waits on a platform feature or a model release. Every item is metadata, and chapters 3 through 7 showed how each one is captured, whether that is the conceptual model drawn out of conversations, the source register built from profiling, the mappings recorded as deliberate decisions, the shape settled by a reasoned choice, or the conventions fixed by writing the rules down once. The platforms will keep shipping smarter assistants, and every one of them will make the same trade this chapter opened with: it reads what it can see. Widen what is captured and every generator you own improves at once, whichever vendor ships it.

Metadata to capture. Here is the score sheet: the conceptual model with approvals; the mappings with deciders and history; the conventions as declared data; the source register with accepted profiles; the shape decision with its rulings; the knowledge documents, extracted and addressable; and the environment and dependency facts. Then there are the two receipts, which are prompts under change control and a record for every run.

Look at what those receipts quietly demanded: who decided, who approved, and which context fed which output. Capture the facts as the work happens and they cost you nothing, but reconstruct them later and they become a project with a budget. That trade, records written at creation against records rebuilt under audit pressure, is the whole subject of chapter 9.

Lineage, governance, and trust before go-live

In this chapter

- Lineage recorded at creation, and what reconstructing it later costs.
- Governance as measurable completeness with gates, not a policy document.
- Rules with teeth, and the two ways rules go wrong.
- The model under version control: branch, diff, review, commit.

Chapter 8 closed on a trade: records written while the work happens cost you nothing, while the same records rebuilt afterwards become a project with a budget. This chapter takes that trade in its three most expensive forms, which happen to be the three questions every go-live review asks. Where did this number come from? Who is accountable for it? Can you show me what changed and who agreed to it?

Answer them while you build and the review is a formality. Answer them afterwards and you answer under time pressure, in an incident or in front of an auditor, which is the same work at several times the price.

Is data lineage a by-product, or a project?

Record which upstream column produced which downstream column at the moment of derivation and data lineage costs almost nothing: one field, filled in by the step that was creating the column anyway. Reconstruct it later, from SQL and query logs and notebooks nobody has opened since March, and you are paying the expensive alternative every team eventually pays for. You also get a weaker result: an observation of what already ran, usually at table grain, stale from the first pipeline edit.

Lineage written at birth is unglamorous, and that is the argument for it. Here it is, on one column of the demo dataset.

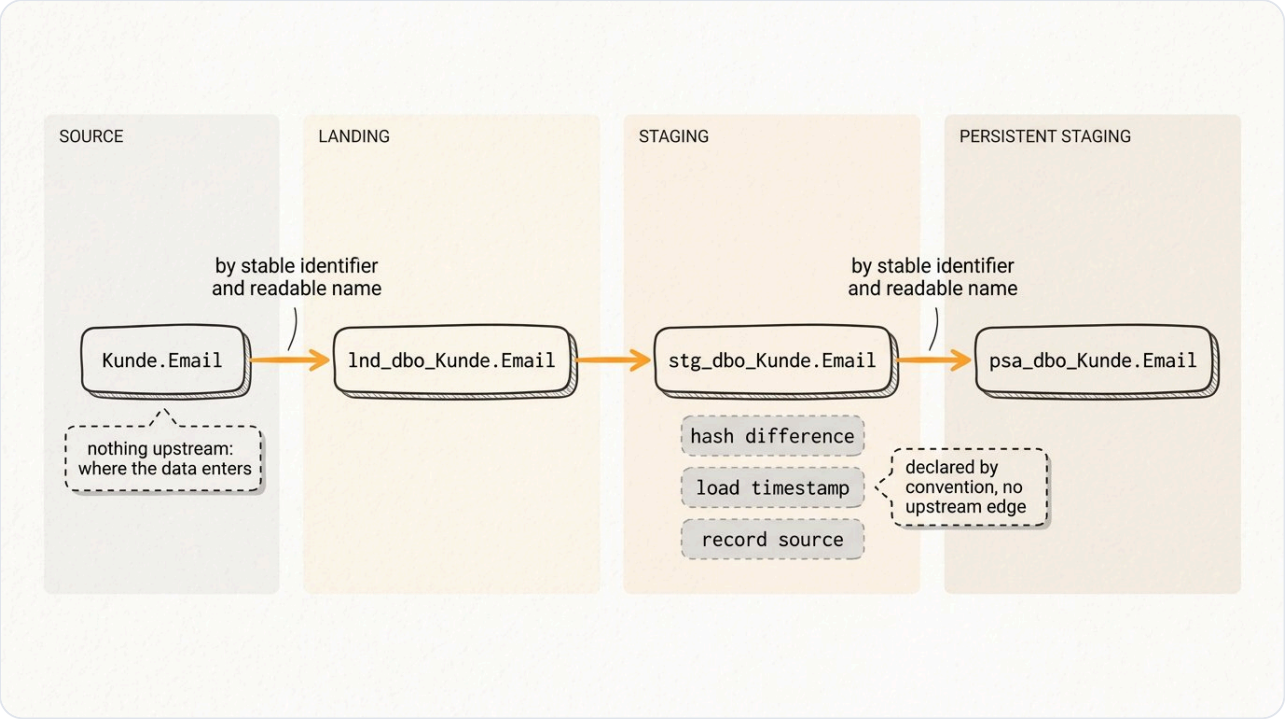

Figure 9.1: one column, four layers, four records that nobody drew by hand.

Four files and four one-line records are all it takes, and "where did this email address come from" becomes a walk rather than an investigation.

Three of the fifteen columns in that staging table name no parent: the hash difference, the load timestamp, and the record source. Those are chapter 7's system columns, and their provenance is a declared convention, so "nothing upstream" is the correct record. What you cannot afford is a column whose record is silent because nobody captured anything.

Two demands separate a real lineage record from a diagram. The first is column grain with expressions resolved: a computed column has to record every column its expression reads, or the trace goes dark exactly where the logic gets interesting. The second follows from chapter 5. Because the mappings terminate in the conceptual model, a governed attribute traces back to every column that populates it, across every system that feeds it. Impact analysis stops being archaeology and becomes a query you run before you change anything.

Governance flows through the mappings

Who is accountable has the same shape as where it came from: capture it once, where the meaning lives, and let it travel. Data governance belongs on the business entity and its attributes, not on the tables and columns. Classify an attribute once as personal data, set its sensitivity, name its owner and its steward, and every column mapped to it inherits all of that, across every landing table, staging copy, and satellite the attribute reaches. Tag the columns instead and you record the same fact several hundred times, in places that will disagree with each other by the second quarter.

Inheritance gives you propagation, and measurement turns that propagation into governance. Declare requirements per property and per asset type: a table must carry an owner and a description, an entity an approved definition and a classification, an attribute its definition. Each one is either present or absent, so completeness becomes a number for a subject area instead of a feeling about one. Most

teams have the policy document and no measurement, which is why the honest answer to “are we governed” is usually a pause.

A requirement that stops nothing is tracking rather than enforcement. Turning it into a gate means naming the action it blocks: generating a build, approving a definition, accepting a mapping, finishing an import. Two rules keep those gates from becoming an obstacle course.

Gate the release, not the thinking. Committing metadata to a branch should never be blocked by a governance requirement, because if a half-governed draft cannot be saved, the work moves into spreadsheets where nothing is measured at all. Block the generation, the approval stamp, and the promotion to the main line, but never the act of writing down what somebody just learned.

You cannot require a fact about something that does not exist yet. A gate that fails an import because the tables it is about to create carry no descriptions blocks the very step that creates them. Let new assets pass, then govern them once there is something to govern. Check the unattended paths as well, since a gate built around a person clicking Accept does not stand in an automatic one.

Rules with teeth, aimed carefully

Requirements answer whether a fact was captured. Validation answers a harder question: whether what was captured makes sense. A business key that no column implements, a system column that has drifted from the convention, a transformation referencing something the target platform does not have.

Treat the rules as a catalog rather than as logic buried in code. Each rule gets a title, criteria in plain words, remediation guidance, a severity, an enforcement level, and a switch, all adjustable without a release. The check itself stays code, because a check is a program. Everything about it is data, because tuning a severity is a governance decision rather than an engineering task. Rules nobody can see or tune get ignored the first week they turn noisy.

Two failure modes are worth designing against, because both are fatal to trust.

Rules that fire on things nobody can action. Render every advisory convention at the same volume as a genuine defect and you get a canvas full of amber icons, which communicates “everything is wrong” rather than anything actionable. The sharpest version of this is the false positive, and its usual cause is a hand-curated reference list. A checker holding a couple of dozen function names, judging expressions written against a platform that documents several hundred, reports every name it lacks as a problem when the expression is correct. The cost is not the wrong warning. It is the engineer who learns the warnings are wrong and stops reading them, including the one that mattered.

Findings that outlive the thing they complained about. A finding is a stored row, and if nothing re-evaluates the rule when the metadata changes, that row becomes the only statement left on a question it can no longer answer. The convention moved, the column was fixed, the severity was retuned, and the finding still says what it said in June. So let anyone ask for a fresh evaluation on demand, sweep the findings a fresh run did not rewrite, and make every remediation report what it actually did. A repair button that claims success without changing anything is worse than no button.

A model you can diff

Software engineering settled the third question fifty years ago. Michael Fagan's 1976 paper on design and code inspections turned code review into a measured practice rather than a courtesy, and the branch-and-pull-request loop that grew up around version control turned it into a reflex. Data teams inherited none of it for their metadata, which is the part everyone has to agree on.

Five things make that loop real, and every one of them is a design choice you can make on any stack.

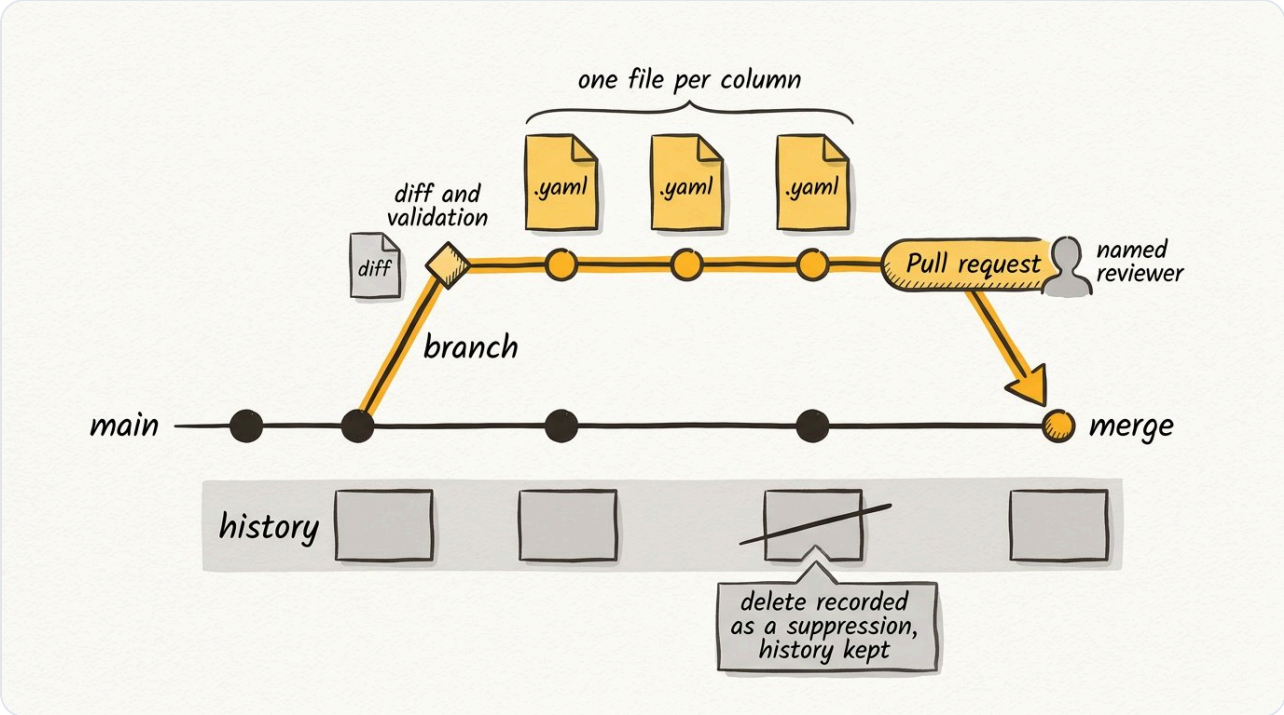

Figure 9.2: the loop software solved fifty years ago, with metadata files where code usually sits.

- **Edits land on a branch, never on the main line directly.** Two people working on different branches never see each other's half-finished thinking, and the main line receives only work that has already been through the loop.
- **Deterministic file paths, at the grain people edit.** The layout mirrors the catalog, one file per column, so a pull request reads like a map of what changed. Write a 200-column table as a single file instead and every one-column edit rewrites the whole thing, leaving a diff nobody reads.
- **Identity that survives a rename.** References carry a stable identifier alongside the readable name, so renaming a table moves every descendant file in one commit while the references keep resolving. Identity carried by name alone breaks the first time a name changes.
- **A review that shows the exact difference.** You get before-and-after per entity, one change revertable without abandoning the rest, and preview paths that match the paths that land.
- **A commit gated by validation, and a history that answers questions.** The history records who changed what, when, and who accepted it, and it has to survive deletion: record a delete as a suppression, or the audit trail leaves with the object.

The alternative is a database nobody can diff, where metadata lives only in application tables, gets updated in place, and cannot say what it looked like in June or show a reviewer a change before it lands. Governance programs built on one end up reinventing half this loop by hand, in change-request forms.

Trust is also tests

The last question a go-live review asks is whether what you built does what you said it does, in production, on a Tuesday. Answering it takes the same metadata again, this time pointed at the data.

Assertions generated from the model cover more ground than hand-written checks usually do. You get a row-count reconciliation between source and target with a tolerance you set, an aggregate reconciliation on a numeric column, grouped so that a total cannot hide a regional gap, a row-count band that catches loads which are suspiciously small or implausibly large, a uniqueness check on a declared key, and a custom assertion for the invariant only your business knows about. Each one declares what a failure means: stop the workflow so nothing downstream runs on bad data, or record a warning worth knowing about but not worth stopping for.

Define them once on the source table, and every workflow step that loads it inherits them. The assertion then travels with the source instead of being copied into every pipeline that touches it, which is the classify-once argument applied to tests, and it leaves the model asserting itself in production.

The people part. Metadata changes need a named reviewer rather than whoever happens to notice the pull request, because a review nobody owns becomes an approval stamp with no reading behind it. Stewards own the rule catalog, since deciding that a naming advisory is a warning rather than an error is really a call about how much noise the team will tolerate. And somebody has to answer the auditor, which is far easier when the answer is a history rather than a memory.

Regardless of stack. Version control, review gates, and lineage written at creation are practices, not platform features: no vendor grants them and none forbids them. A team with text files, a repository, and the discipline to record provenance at derivation has better lineage than a team with an expensive catalog that harvests query logs once a night.

The cheapest time is now

Every item in this chapter is cheap while the work happens and expensive afterwards, which is why it belongs before go-live rather than after it. Provenance is one field at derivation, and a parsing project once you are in the middle of an incident. Classification is one decision on an attribute, or a hundred decisions on columns. History is a by-product of a loop you were going to run anyway, and an impossibility once the edits went straight into a database.

Metadata to capture. Write provenance pointers at creation, at column grain, with expressions resolved. Hold classification and ownership on the business attributes and let them inherit through the mappings. Set governance requirement levels per property and asset type, naming the gates you want and the actions they block. Keep the rule catalog so that each rule carries its severity, enforcement, and remediation guidance. Then keep the audit trail: who changed what, who approved it, and when.

You now have the whole of what chapters 2 through 9 asked you to capture, and a reason to capture all of it. What is missing is a plan for Monday morning, in an order that produces something usable long before it produces everything. That is chapter 10.

CHAPTER 10 OF 10

Six steps to your integration layer

In this chapter

- Six steps in dependency order, scoped to one subject area.
- What each step produces, and who has to be in the room.
- The test that says a step is finished, and the anti-pattern that kills it.
- Why the sixth step feeds the first instead of ending the work.

Nine chapters of argument come down to something you can start on Monday, with the people you have and the platform you already bought. This is not a program and not a migration. It is one subject area and six steps, each one producing what the next consumes, with an integration layer standing at the end of them.

Scope is the first decision you make, and it is the one teams most often get wrong. Pick a single subject area, Customer or Order or Product, and walk it the whole way through. One governed subject in six weeks teaches you more than a charter in six months, and it leaves something running when you are done.

Read the sequence as dependencies rather than calendar slots. Steps 1 and 2 run in parallel: your modelers can agree on what a customer is while your engineers import the source metadata, and neither side waits on the other. Step 3 is the first step that needs both.

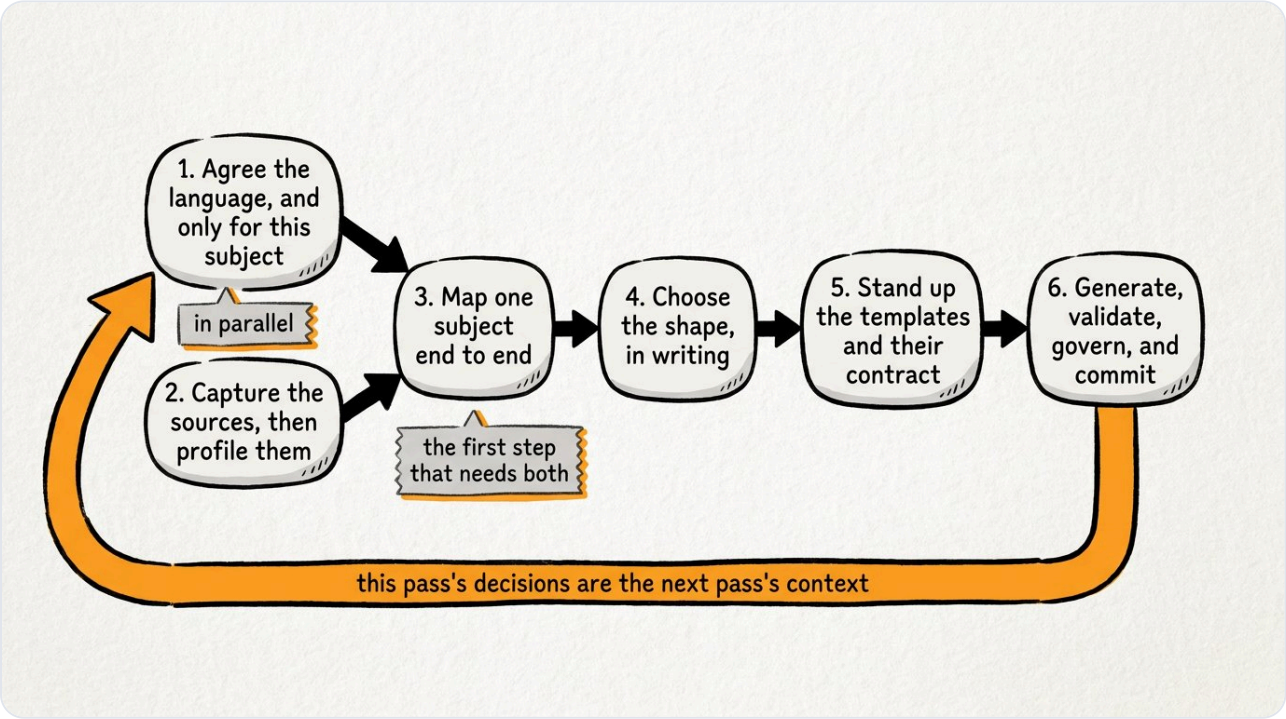

Figure 10.1: read the sequence as dependencies, not calendar slots. This pass's decisions are the next pass's context.

Step 1: Agree the language, and only for this subject

Start by writing down the entities and their definitions in business terms, the synonyms the business already uses, the relationships as sentences, and a business key for each entity, or an explicit note that nobody can name one. Book the meeting before you open the tool, because the meeting is the model. Then add chapter 3's naming rules as ordered steps somebody can execute, not as a style preference nobody enforces.

Keep the room small: a modeler running the method, a steward who owns the standard, and the business expert who owns the meaning.

Chapter 3's demo meeting shows why that hour is not optional. Two lines connect Customer and Club Partner on the source diagram, and they are not one relationship drawn twice. One says which club a customer registered with; the other says which customer is that club's contact person. The glossary now carries them as separate entries.

You are done when each relationship reads aloud as a sentence the business would say, and the business expert agrees it is true. **The way this fails is** the enterprise-wide glossary program: a year of workshops, a taxonomy nobody consumes, and no subject area finished.

Step 2: Capture the sources, then profile them

Two artifacts come out of this step, and they are not the same thing. The register holds one entry per source along with its capability profile: what it can be asked, how it exposes changes, what it refuses. The inventory holds the tables and columns, imported on stable identifiers so a rename does not orphan everything downstream, and reviewed as a diff that somebody approves.

Once both exist, profile the data, because the schema is a claim and the data is the witness. Capture nullability, distinct counts, real formats, and candidate keys, keeping the observations apart from the rulings: unique in today's extract makes a column a candidate key, not a key.

The engineer runs this work, but book the source system owner's hours now rather than on the day their absence blocks a mapping.

You are done when you can answer "what can this source not tell us" for every source in the register. **The way this fails is** trusting the schema as the truth about the business. The demo source declares both club relationships as foreign keys, and that constraint metadata cannot say which one is membership and which one is the contact person. Only step 1 can.

Step 3: Map one subject end to end

This is chapter 2's bronze-to-silver handshake, finally on somebody's calendar. The work is to bind the physical estate to the agreed language, source table to entity and source column to attribute, with every decision carrying its decider, its date, and its reason.

Treat disagreement as an outcome in its own right. A proposal you reject with a recorded reason is worth nearly as much as one you accept, because it stops the same wrong suggestion coming back next

quarter. For the same reason, publish the mapping as the deliverable rather than as working notes: your build, your testing, and your next auditor all run on it.

All four disciplines sit in this room. The engineer knows what the columns hold, the modeler holds the target language, the steward owns the standard, and the business expert settles the meaning.

You are done when a governed attribute traces back to every column that populates it, across every system that feeds it. **The way this fails is** a mapping spreadsheet with a deadline and no owner, finished once and stale by the following sprint.

Step 4: Choose the shape, in writing

Weigh chapter 6's factors against your own situation: how often your sources change shape, what your auditors actually ask for, what your team can already run at three in the morning, and what each shape costs to query on the platform you own. Once you have an answer, write the decision down along with those factors, the date, and the names of the deciders.

Underneath that decision sit the rulings that make it work: which business keys the estate counts by, which system wins when two disagree, how much history the business owes, and where each split boundary falls and why. The architect proposes those rulings, but the stewards and the business rule on the keys.

You are done when the rationale exists where the next engineer will find it, in the catalog rather than in a slide deck in somebody's downloads folder. **The way this fails is** choosing by religion, or by the last conference somebody attended, and then defending that choice for three years because nobody ever wrote down what it was for.

Step 5: Stand up the templates and their contract

This is where the machine starts earning its keep. You need a pattern for each kind of object, and you need your conventions declared as metadata rather than buried in code: naming rules, system columns and the role each one plays, hash definitions, and data-type mappings. On top of that, you need bindings that say which pattern serves which object, along with the declared context a render depends on.

Prove it against real metadata before you trust it. On the demo dataset, a single source table of 5 columns generates a landing table of 7, a staging table of 8, and a persistent staging table of 11. Nobody typed the extra columns or chose their names. The load timestamp, the record source, the hash difference, the effective-date pair, and the current-row flag all come from the declared conventions, which is why they arrive identically named on every table those conventions touch.

Name the template author while you are at it. In most teams nobody holds that role, so the patterns end up being whatever the last contractor left behind.

You are done when you change one naming pattern and every generated name moves with it. **The way this fails is** hand-editing generated output, which quietly turns a generated object back into a maintained one and tells nobody downstream.

Step 6: Generate, validate, govern, and commit

Build the layer as a reviewable artifact set with a pass or fail on every file, not as a script that writes straight into a schema. Point chapter 9's validation rules at it, tuned so every finding is actionable. Then put the whole model on branches, with diffs a reviewer can read, a commit gated by the rules, and a history that survives a delete.

Only one person needs the reviewer role, and that person is named. Everyone else keeps their day job, which is the point: the machine does the repetitive work, and the team keeps meaning, standards, and approval.

Watch what survives generation. In the demo dataset, the foreign key naming the club's contact person is still there in persistent staging, re-pointed at the persistent staging copy of the customer table. What the business explained in step 1 is still a relationship four layers down, because somebody recorded it rather than leaving it to be re-derived.

You are done when you can answer "who accepted this, and when" for any object in the layer. **The way this fails is** a catalog nobody can diff.

The people part. The chapter is the staffing plan, so treat this box as the calendar. Four meetings carry the six steps. The language meeting is a weekly hour or two for the modeler, the steward, and the business expert, running until the subject area is agreed. The source interview puts the engineer and the source system owner together, one table at a time, booked the week the source enters the register. The mapping review gathers all four disciplines in a standing slot, until the subject is bound. The shape decision is one meeting the architect prepares and the stewards rule in. Steps 5 and 6 add no meeting at all: they need a named template author and a named reviewer, whose work shows up in pull requests anyone can read.

Metadata to capture. Every box in this book collapses into one scorecard, and you can score a subject area against it in an afternoon. Start with entities, definitions, synonyms, relationships, business keys, and executable naming rules. Add the source register with capability notes, the inventory on stable identifiers, the approved import diffs, the observed profiles, and the candidate keys held apart from rulings. Record mappings with deciders, dates, and reasons, the rejections included, and the shape decision with its rulings underneath. Declare conventions as metadata: naming patterns, system columns, hashes, type mappings, bindings. Carry provenance at column grain, written at derivation, and classification and ownership on the attributes. Close with governance requirements and the actions they gate, the rule catalog, and the audit trail. Anything there you cannot point at is the next thing to capture.

Why is the second subject area faster than the first?

The six steps are a loop, not a waterfall. Step 6 does not end the work, it feeds step 1, because the decisions you recorded on this pass become the context the next pass reads. A mapping you accepted turns into the standard the next proposal is measured against, and a naming rule you settled generates the next hundred names without anyone reopening the discussion. Your decisions do not exit the process once you have made them: they become it.

That is why the second subject area moves faster than the first and the tenth is routine, and why none of it needs a platform migration or a new budget line to start.

Regardless of stack. Read back through the six steps and count the platform decisions: there is one, in step 4, and even there only the economics of the factor table moved. Every platform will tell you to land your data and let its AI do the rest, but the rest assumes a language somebody agreed, sources somebody profiled, a mapping somebody signed, a shape somebody chose, conventions somebody declared, and a history somebody can read. No vendor supplies those, and none forbid you from building them: AI proposes, a person decides, and every step runs on governed context you control.