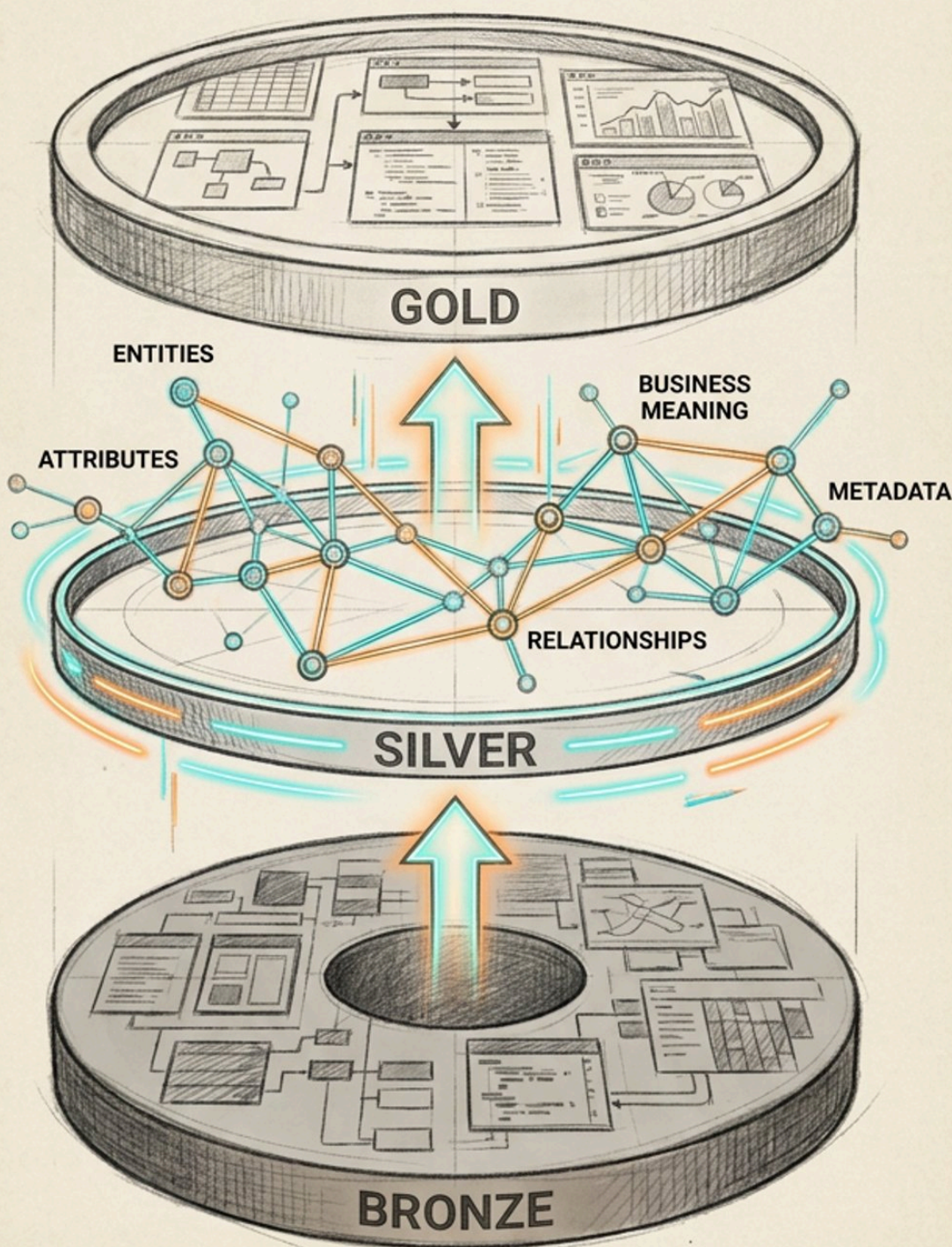


AIMS

Rethinking the Lakehouse Silver Layer

A Meaning-First Approach to Medallion Architecture in the AI Era



AI-Ready, Meaning-First Silver

Model the meaning. Preserve the source. Integrate by exception.

DELTA VAULT

AIMS

Rethinking the Lakehouse Silver Layer

A meaning-first approach to medallion architecture in the AI era: a thin silver layer, an ontology, and the AI agents that read it.

Model the meaning. Preserve the source. Integrate by exception.

2026

Contents

INTRODUCTION	Read this if you have ever waited for an integration layer
--------------	--

CHAPTER 1	Meaning used to live in the tables
-----------	------------------------------------

CHAPTER 2	The economics of meaning flipped
-----------	----------------------------------

CHAPTER 3	Every method stops at the concept
-----------	-----------------------------------

CHAPTER 4	The business model is the ontology
-----------	------------------------------------

CHAPTER 5	Keep the shape of the source
-----------	------------------------------

CHAPTER 6	A key you can read
-----------	--------------------

CHAPTER 7	Attributes are where the payoff lives
-----------	---------------------------------------

CHAPTER 8	Integration by exception
-----------	--------------------------

CHAPTER 9	AI proposes, a person approves, templates generate
-----------	--

CHAPTER 10	The agent reads the contract
------------	------------------------------

CHAPTER 11	Seven steps to a meaning layer
------------	--------------------------------

APPENDIX	Silver layer glossary and reference tables
----------	--

Read this if you have ever waited for an integration layer

Third normal form and Data Vault wrote business meaning into the shape of the tables, because for decades only humans had to make sense of that meaning. Both are long-standing ways to design an enterprise data warehouse. In both, a customer gets its own table and every relationship becomes a join.

Now the question is whether AI needs meaning in the same form, and it doesn't.

Meaning can move out of the table shapes into a description of the things your business runs on. That description says how each one is identified, how they relate and what describes them. Written down, that's your business model. Written so a machine can read it, it's an ontology, held in catalogs and metadata.

A lakehouse moves data from raw bronze through cleaned silver to report-ready gold. With the ontology written over the tables, the [silver layer](#) no longer needs a third normal form or Data Vault shape to carry meaning. It can stay thin and keep its sources' shape. An AI agent reading the ontology works out which table, column and join a question needs.

We call this approach AIMS

We call this approach **AIMS: AI-ready, Meaning-first Silver**. Rather than restructuring source data to fit a predefined modeling methodology, AIMS starts with business meaning. That meaning is captured in governed metadata that people, AI agents and code generators can understand. The silver layer keeps the structure of its sources wherever practical. More integration comes in only when a specific business or technical requirement justifies it.

Each part of the name carries part of the argument.

- **AI-ready.** Business meaning is defined explicitly, so an AI agent can understand the data and use it.
- **Meaning-first.** The business things your data describes, how they relate and the columns everything depends on are defined and approved before a table publishes.
- **Silver.** The silver layer stays light and keeps the shape of its sources, with integration added only where it's needed.

AIMS: Model the meaning. Preserve the source. Integrate by exception.

A row added, nothing retracted

Our first book, [The Data Lakehouse for Everyone](#), stayed neutral on the silver layer's shape. [Its chapter on choosing an integration shape](#) gave the shapes one table of factors instead of a verdict. This book keeps every factor and adds the row AI put into that table: what each shape leaves for a machine to read.

Where DeltaVault, our metadata catalog, already does part of AIMS, a closing "In DeltaVault today" section shows it on screen. Any part DeltaVault doesn't do yet is the design of AIMS. DeltaVault and our AI agents never access your data, only the metadata provided.

Words used one way

Customer number 123 in your order system and customer 123 in your sales system can be two different customers. A key only means something inside the system that issued it. That system is the key's **record source**.

The business model plus the mappings that tie silver columns to it is the **meaning layer**: how each row is identified and related, what each column and code means, which rules and history apply. The catalog's summary of each table is that table's data contract.

What's left of data modeling? All of the deciding stays. None of the rebuilding does. You still decide what the business things are, what identifies each one, how they relate, what each column means and how much history to keep. But you stop restructuring every source table to carry them, and that's **just enough modeling**.

Integration only when it's needed

Traditional data warehouse approaches often require you to integrate data from different systems into a common structure before anyone can use it. That can involve considerable modeling, development and maintenance.

But what if you didn't have to do all that work up front?

AIMS keeps data largely in its original source structure. It uses the ontology to describe what the data means and how it relates to other data. Extra integration structures come in only when there's a clear reason for them.

For example, if two systems identify the same customer with different customer numbers, you can match those identifiers to one **shared key**. If someone needs one combined record for that customer, a **golden record**, you can add a view or table that brings the sources together. And if you need every historical change for auditing, you already have it: persistent staging in bronze keeps every version each source delivered, and silver reads from it.

The principle is simple: integrate when the requirement exists, not because a modeling methodology demands it. We call this *integration by exception*.

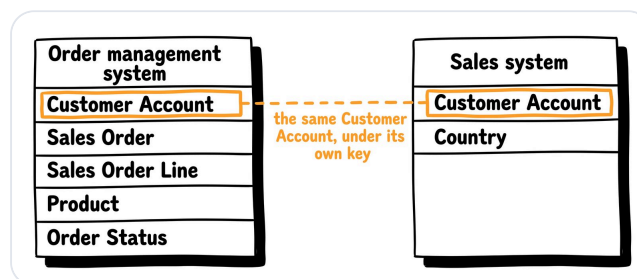
Tables are matched by meaning, records by a person

The ontology tells an AI agent that two systems' customer tables hold the same kind of thing. So a question can cross both, and nobody builds a table to combine them first. But are two of those records the same customer? A person approves the match rules and types the rest, so nothing decides alone that two records are one customer. Either way, reports always get their own data marts.

One table and one example

Each argument closes on one table: the structural answer (what third normal form or Data Vault built, at what cost), what the agent reads, and what's yours to decide.

In the worked example, an order management system knows Customer Account, Sales Order and its lines, Product and Order Status. A sales system knows the same Customer Account under its own key, with its Country. It's a composite built for this book, not a case study.



Where should you start?

Start with [Meaning used to live in the tables](#) if you learned your craft on Data Vault or third normal form. If you lead a data platform, start with [The economics of meaning flipped](#). If you build the silver layer, read the first three chapters, then [The business model is the ontology](#), through [Integration by exception](#). Read [The agent reads the contract](#) if somebody asks what "AI-ready" means. And end on [Seven steps to a meaning layer](#), steps you can start on Monday.

Meaning used to live in the tables

In this chapter

- **A Data Vault diagram is a paragraph.** Read aloud, its tables are sentences about your business.
- **The readers set the limits.** Tables were all a machine could read, so meaning went into tables.
- **The bill is the price of the encoding.** What it bought, and the one change that brings it down.

The methods put business meaning in the tables because nothing else could read it. Now something can.

That's a claim about methods you may have built a career on, so it has to be earned. Third normal form and Data Vault aren't the villains of this story. They're the most disciplined answers anyone gave to a real problem, and the problem hasn't gone away.

What changed is who, or what, can read the answer.

The hub is a sentence

A [Data Vault](#) model is built from three kinds of table: hubs, links and satellites. Read a Data Vault diagram aloud instead of looking at it. A hub says, "This is a core business concept, and this is the key that identifies one instance of it." A link says, "These two concepts relate, and this is the role each plays in the relationship." And a satellite says, "These attributes describe that concept, as this source system saw them, as of this load."

The diagram isn't a picture of tables. It's a paragraph about your business, written in the only language a database could execute.

A canonical third normal form warehouse is a longer paragraph with stricter grammar. Every concept has one definition and one owner. Any source that knows about a customer gets bent into the one shape the model allows. Inmon's definition of the warehouse, in *Building the Data Warehouse*, carries the grammar in four adjectives: subject-oriented, integrated, time-variant, non-volatile. Three of the four are statements about meaning. Only one is about storage.

Read this way, both methods are ontologies. They name the things your business runs on and say how each is identified, how they relate and which attributes describe them. And they write it all in table structure, because when they were designed, nothing else could hold meaning and still do work.

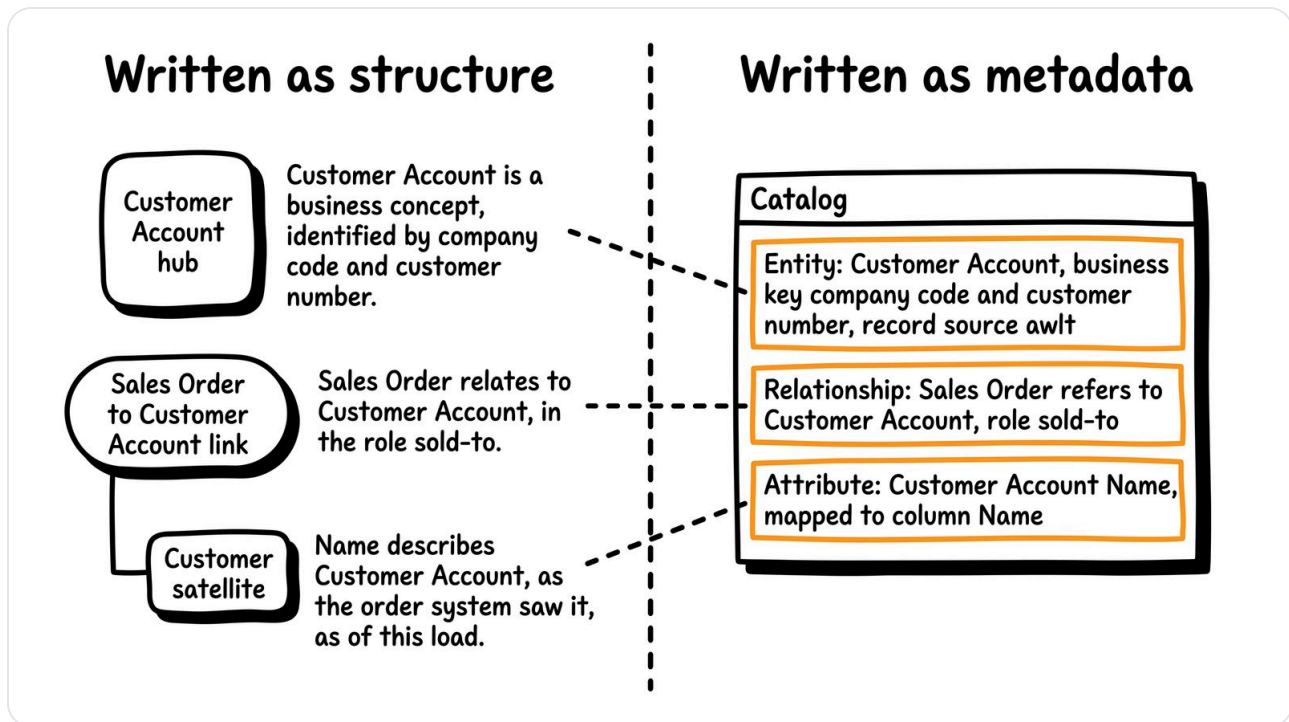


Figure 1.1: the hub is a sentence. Each table kind beside the sentence it encodes, and on the right the same sentences written once as catalog metadata.

The methods were right, for the readers they had

Who read a data model in 1995, or 2005, or honestly 2015? A load pattern read it. It had to know which column was the key, which table owned the history and which row was current. A query optimizer read it too. So did a reporting tool, through a layer of business names and measures that somebody built by hand on top, a semantic model.

Every one of those readers could consume a table. Not one could consume a glossary entry, a written definition or a mapping spreadsheet. So if you wanted a machine to use your meaning, you had one option: make it structural.

That's what the methods did, deliberately and well. Linstedt described the Data Vault in 2002 as "a detail oriented, historical tracking and uniquely linked set of normalized tables." Each design choice makes a fact about the business survive inside a table where a load could find it.

Here's the tell that the readers, not the authors, set the limits. Both canons give metadata a chapter, and both treat it as documentation that sits beside the system. Neither says "and the system executes this," because no system could. The meaning that mattered at run time had already been poured into the tables. The metadata was the commentary.

A Data Vault author says the same from inside the method. In a vault, Patrick Cuba writes, "hub tables are the nodes reserved for business entities". Links are its edges, he says, and satellites are their properties. Cuba adds that "what a traditional Data Vault lacks are explicit, machine-understandable semantics". The meaning is there, written as structure. Nothing but a table could carry it to a machine.

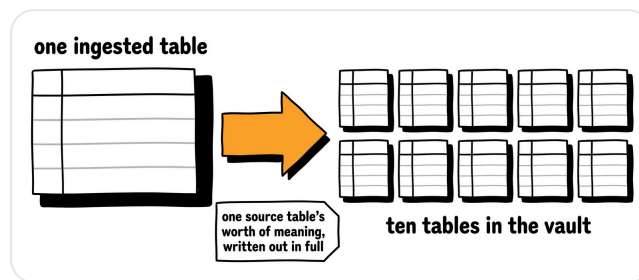
Metadata-driven automation arrived later and seems to contradict this. Teams generated their vaults from mapping sheets for a decade. But the sheet told a generator which hubs and satellites to build,

and every consumer read those tables. The automation shortened the build. It never moved the meaning, because the readers downstream still couldn't read anything else.

The bill is the price of the encoding

Once you see the methods as meaning written in structure, their cost stops looking like overhead. It looks like an invoice, itemized by unit of meaning.

A concept costs a table, and each fact about it costs another. Andrew Foad created a lighter approach that keeps each source table whole, the Hook method. He reports from his own team: "Our single ingested table eventually blew out to ten tables for Data Vault." Ten tables isn't waste. It's one source table's worth of your meaning, written out in full.



Reading it back pays the encoding again. In Foad's worked comparison, a two-table join over data kept in its source's shape is a four-table join in the vault. Real questions are wider than that. So a vault adds two more kinds of table. One lines up the version of each record in effect on a date, a point-in-time table. The other pre-joins hubs and links along a common query path, a bridge table. A 2026 industry analysis says the hub-link-satellite structure "requires assembling data through multiple joins before it resembles anything analytically useful."

A canonical definition costs the analysis before anything ships. Nothing can be loaded until the one shape is agreed. So the slowest conversation in your company sets the delivery date of your fastest pipeline.

A change to the meaning is a migration of the structure. What happens when your business decides that a customer account and a billing account are different things after all? A vault adds hubs and reroutes links and satellites. A canonical model reopens the agreement.

None of these line items is a mistake. Each is the honest cost of making a sentence about the business executable when only tables could be executed.

What the bill bought, and what we keep

It would be cheap, and wrong, to stop here and call the methods expensive. The bill bought auditability of every version a source ever sent. It bought parallel loading, because hubs, links and satellites don't depend on each other at load time. It let you add a source without altering what was already there. And applications that can't afford to learn each source's quirks got a stable, source-independent interface.

As we put it in "[How raw is your Data Vault?](#)", "A source-shaped vault does one thing genuinely well: it captures every source faithfully and keeps the full history." That one thing no longer needs a vault.

Persistent staging, the layer below silver that keeps every version a source delivered, already does it in bronze, insert-only and in the source's shape. It loads one table per source table, in parallel, and a new source adds without altering anything. A stable interface is a golden record, one combined view or table built when someone asks for it.

AIMS (AI-ready, Meaning-first Silver) also keeps the discipline the methods taught: identity first, relationships explicit with their roles, history chosen on purpose per table, nothing dropped silently. Nothing in this book calls a method obsolete.

What changes is the default, and where the meaning lives.

What changed to make data modeling for AI possible?

For the bill to come down, one thing had to become true. A machine had to read a sentence about the business without that sentence being a table.

It can now.

A model can read "Customer Account is identified by company code and customer number in the order system," and "Sales Order refers to Customer Account as sold-to and again as bill-to," and "Status code C means Cancelled in this system," and "One row is one Sales Order version." Having read them, it can propose which column carries which meaning, generate the load and answer the question.

[Databricks, introducing the data ontology](#), names why that matters now. With AI agents, it says, "there may be no knowledgeable human sitting between the data and the decision. The agent has to discover the meaning itself."

Once a sentence can be read as a sentence, it no longer has to be written as a table. The meaning can move into a governed store of metadata that describes your data, a catalog, where it's cheap to add and cheap to change. A person, an AI agent and the generator that turns metadata into tables and loads (a template engine) all read it there.

So the tables can stay the shape of the source.

The structural answer	A vault builds five or six tables for one Customer Account in two systems, four of them before the first report, where the catalog holds one entity, each key with the system that issued it, and a match rule tying both keys to one.
What the agent reads	The three sentences the vault encoded become three lines of metadata: the entity with its business key and record source, the relationship with its role, and the attribute tied to its column.
Yours to decide	Nothing yet: the first decision, the one both methods put first, is what the business things are and what identifies each one.

The economics of meaning flipped

In this chapter

- **Two readers, one question.** What a model does with stated meaning that it can't do with a schema.
- **The numbers and the platforms.** What an ontology adds on enterprise questions, and the semantic layer every platform now ships.
- **Where the AI stops.** Why the line a person holds sits exactly where models read badly.

Can a model really do more with stated business meaning than with a schema? Benchmarks that test turning a plain question into a database query, text-to-SQL benchmarks, have measured it. Every major platform now builds on the answer.

Two readers, one question

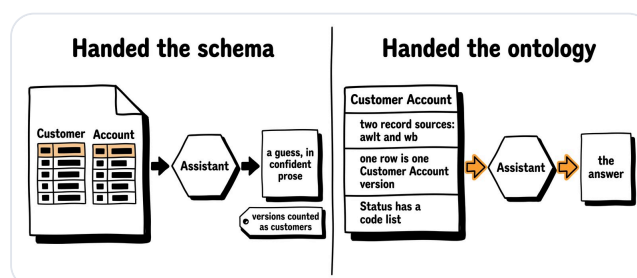
Ask an AI assistant how many customer accounts you lost last quarter. Hand it only the schema of two systems: an order management system with a Customer table and a sales system with an Account table. It has to guess which table is the customer, or whether both are. It'll count rows in a table that holds one row per customer version and report versions as customers.

Every guess arrives in confident prose.

Now hand the same assistant a description of what your data means, written for a machine. That's the [ontology](#), and from it the assistant learns that a customer account is a thing the business tracks, an entity named Customer Account. The order system identifies one by customer number, and that number is only unique within one company code of the order system. So the account's business key is the pair, company code and customer number, from record source awlt. The sales system identifies a Customer Account by its own account identifier, from its own record source.

It also learns that the customer table holds a row for every version of an account, so one row is one version of a Customer Account. Only the current version should be counted. And the Status column is tied to a defined business attribute whose list of codes says which codes mean closed.

The question and the model haven't changed. The answer has, because the second reader was handed meaning as text. The first had to rebuild it from column names.



How much does an ontology lift AI accuracy?

In 2023, Sequeda, Allemang and Jacob built a benchmark from an enterprise schema. They asked a large language model business questions against it two ways. Posed directly against the SQL schema, the model “achieves an accuracy of 16%”. Keep the questions and the model the same, and “this accuracy increases to 54% when questions are posed over a Knowledge Graph representation of the enterprise SQL database”. The knowledge graph was an ontology over the same tables, so nothing about the data moved.

A follow-up a year later added the ontology’s rules to the prompt and let the model decline. Accuracy rose to 72 percent, “including an additional 8% of ‘I don’t know’ unknown results”.

In 2026 a paired benchmark tested three of the most capable models of the day. It found that a written guide to what the tables and their measures mean, a semantic layer document, “improves accuracy by +17 to +23 percentage points across all three models”. With the document, the three models were statistically indistinguishable.

The model isn’t the variable. The context is.

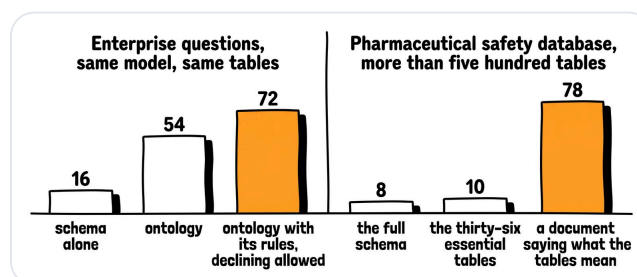
Which kind of meaning does which job? In 2024 Painter and colleagues gave a model a pharmaceutical company’s safety database of more than five hundred tables. Cutting its schema document to the thirty-six essential tables lifted the share of queries an expert passed from 8 percent to 10. A document describing what the tables meant lifted it to 78.

Descriptions find the right table and column. Structure gets the number right. In the 2023 benchmark, the questions that needed more than four tables scored zero against the schema. Against the ontology, whose relationships were explicit, they scored over a third.

AIMS (AI-ready, Meaning-first Silver) answers both. Every business attribute a column is tied to carries a description. Each relationship states the role it plays and how many rows on each side can match, its cardinality. So an agent finds the column, and joins without doubling a row.

Benchmarks are tidy and your estate isn’t. But the lift didn’t come from a bigger model or a cleverer prompt. It came from an ontology written over tables that already existed. That’s exactly what this book tells you to build.

Our post [“Meaning before machinery”](#) says the model now writes working code in minutes, and that “the bottleneck has moved to the one thing the machine cannot know on its own: what your business means.”



The vendors settled the direction

You don't have to take it from us. Every major platform now says in its own documentation that answer quality depends on curated semantic context. Each has built its own semantic layer to hold it.

[Databricks describes its knowledge store](#), which grounds its Genie assistant, as "a collection of curated semantic definitions that enhances Genie's understanding of your data and improves response accuracy." [Snowflake says its semantic views](#) "address the mismatch between how business users describe data and how it's stored in database schemas."

Microsoft's Copilot documentation for its reporting tool is the most candid: "Without preparation, Copilot can misinterpret the data and return generic or inaccurate results." The ontology item in Fabric, still in preview, promises to "ground agent experiences in governed definitions, relationships, metrics, rules, and source mappings instead of relying only on raw schemas or prompts."

Notice what none of those objects asks for: a restructured silver. Each sits over the tables that already exist and supplies the meaning as context. That's the AI era's answer to where modeling goes.

Gartner's May 2026 release put it flatly: "Traditional schema-based data models alone no longer suffice for agentic AI because they lack business context and data meaning."

In September 2025 Snowflake, Salesforce, dbt Labs and more than a dozen others launched the Open Semantic Interchange initiative, now Apache Ossie, in the Apache Incubator. It aims to define "a common, vendor-neutral semantic model specification that standardizes how semantic metadata is defined and shared". You only need an interchange format when every vendor's object is an island.

Written first, the meaning generates what the vendors annotate

Each of those objects annotates tables that already exist. It's in one platform's format, read by that platform's assistant, and written after the data landed. None generates the tables or drafts the marts. None exports a contract saying what one row is, in a form that a code generator working from metadata on another platform, a template engine, can read.

They're the right idea at the wrong altitude: a semantic layer over a physical layer still built the old way.

AIMS inverts that order. The meaning layer, the business model plus the mappings that tie silver columns to it, is written first, in the catalog, on no platform in particular. The silver layer is generated from it. The marts are drafted from it. And the same metadata is what the agent reads. So "use my platform's semantic layer and stop" isn't the end of the argument.

Where the AI stops

The economics flipped because a model reads meaning, and the limit of what it may decide follows from what it reads badly. It's reliable at drafting, and unreliable at what a business key really identifies, at the rules your business never wrote down, and at whether two identifiers in two systems are the same customer.

So AIMS draws a line there. The AI proposes, with evidence or a stated assumption on every suggestion. A person approves every decision that changes meaning or values, however confident the model is. That line is the approval boundary.

The AI does most of the hours and a person carries most of the risk. That split isn't a brake on the AI. It's the reason the result can be trusted.

The flip, stated

Before, meaning had to be structural to be usable by a machine. So it was expensive to add (a table per concept, another per fact about it) and expensive to change (a migration per changed definition). Third normal form and Data Vault were right to pay that price when it was the only price on offer.

After, meaning is metadata a machine reads. It's cheap to add (a sentence in the catalog, mapped to a column) and cheap to change (edit the sentence, regenerate what depends on it, compare the contract). It no longer has to be carried by the shape of the tables.

So the tables can keep the shape of the source, with integration built only where a specific business requirement calls for it.

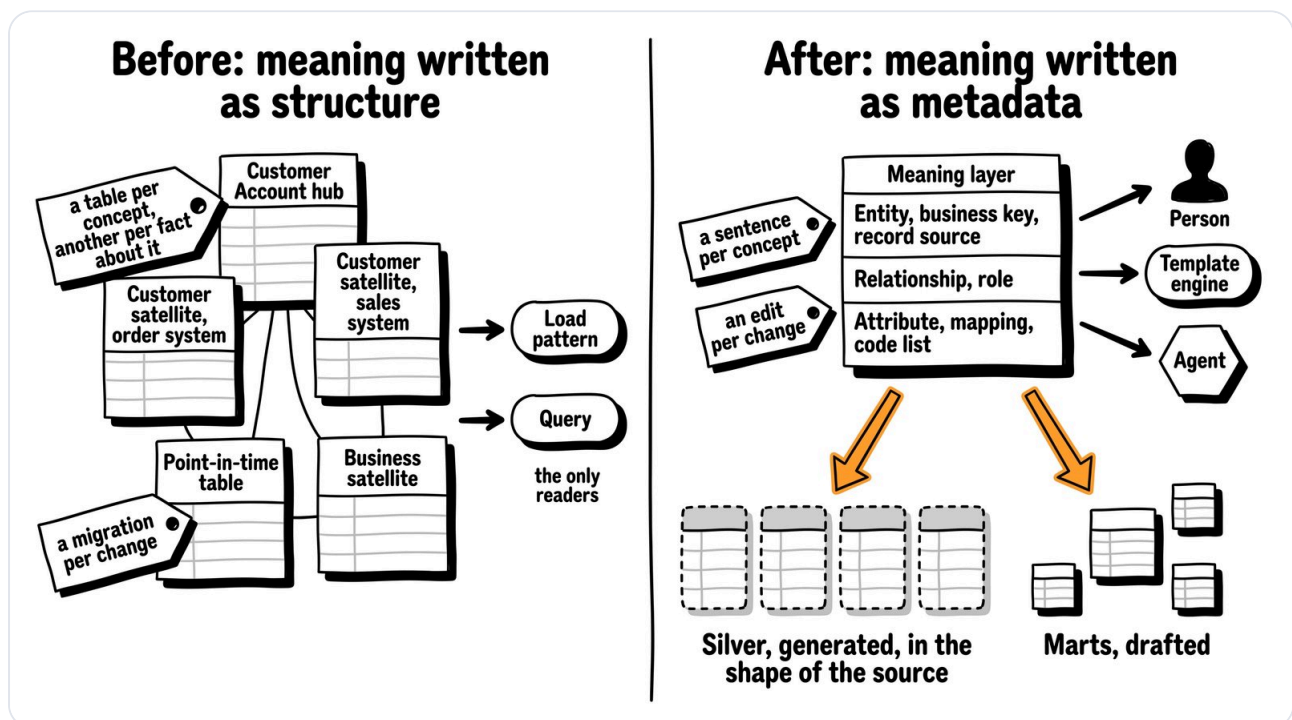


Figure 2.1: where the meaning lives. Before, inside the table shapes, read only by a load pattern and a query. After, in a catalog read by a person, a template engine and an agent, with silver generated from it and the marts drafted from it.

The structural answer	A vault answers the churn question with four or five tables for the order system alone, loaded before the first query, where the meaning layer answers it with six lines of metadata.
What the agent reads	The entity, its grain sentence (one row is one Customer Account version from record source awlt), its business key and record source, the sold-to relationship from Sales Order, the code list, and the status attribute that uses it.
Yours to decide	How much of the AI's drafting a person reviews, knowing that every decision changing meaning or values stays with a person, whatever the model's confidence.

Further reading in this book

- [AI proposes, a person approves, templates generate](#): the full approval boundary.
- [The agent reads the contract](#): what each platform's agent reads, traced through one question.

Every method stops at the concept

In this chapter

- **Six patterns, one gap.** What each silver pattern does to a source table, and the question none answers.
- **Where the AI payoff dies.** Why an agent needs each column's meaning.
- **A level, not a gate.** The five meaning levels, and which columns must be mapped.

If a model reads meaning stated as metadata, why doesn't any existing method already state it? The methods aren't careless. Every one of them states meaning down to the concept, the business thing a table is about. Then it stops, one step short of where an AI agent needs it most.

The platforms ask less of silver than the arguments do

[Databricks' documentation](#) says the silver layer "should always include at least one validated, non-aggregated representation of each record." [Its glossary](#) adds: "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."

[Microsoft's guidance for Fabric](#) lists "Match customer records across systems" without saying how. Snowflake's documentation gives silver one clause, "a cleaned or conformed dynamic table". Databricks names two methods and requires neither. Microsoft and Snowflake name none.

So the shape is yours to choose.

The same glossary says how much work silver should take. Its data is "matched, merged, conformed and cleansed ('just-enough')".

[Our first book](#) read the glossary's verbs as four unanswered questions: matched on which keys, merged how, conformed to whose definitions, recorded where. AIMS (AI-ready, Meaning-first Silver) answers them without a method's shape. It matches on a key you can read and merges only where a specific requirement calls for it. It conforms in meaning by agreeing what each column means rather than reshaping the data, and records that agreement in the catalog. That's just enough modeling. Microsoft describes its ontology item the same way, as a means to "model business domains without requiring you to reshape the source data solely to fit the ontology."

Six patterns, judged on what they do to a source table

A canonical third normal form model redesigns the source. Inmon's integrated warehouse gives every concept one definition and one owner, then bends every source into that shape. It buys integration and a stable interface, and costs the analysis before anything ships.

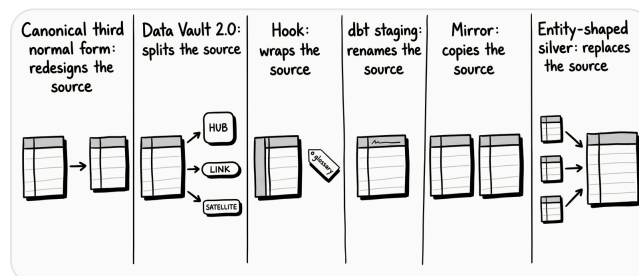
Data Vault 2.0 splits the source. Linstedt's hubs carry business keys, links carry relationships and satellites carry context. That buys parallel loading, the audit of every source version and new sources that add without altering. It costs several tables per source table, point-in-time tables to read them back, and automation as a precondition.

Hook wraps the source. Andrew Foad's method keeps each source table whole, one to one. It adds business key columns. Each is tagged with a label saying which system and context the key belongs to (Hook calls it a key set). And Hook lets nothing exist without a glossary definition. That's the lowest upfront modeling of any published method, at a cost named below.

dbt's staging convention renames the source. It puts one light layer over each source table, often a view, renamed, cast and lightly cleaned, and dbt calls this first layer a staging model. It's cheap, testable and widely known. But it brings no business key discipline. It keeps history only through dbt's record of what changed between runs, its snapshots, and it defers integration.

A mirror copies the source. It's typed, renamed, deduplicated, validated and historized, and nothing more. It's the cheapest silver there is. But there's no business meaning in it, so every gold model resolves identity. Our post ["Show Me Your Silver Layer"](#) called the result replication: "If silver holds a faithful copy of each source, then everything downstream is replication with extra steps."

An entity-shaped silver replaces the source. It builds one table per business entity, keyed on the business key and fed by every source. You get one governed record per entity, and you have to decide how to integrate before anything ships. Every feeding source needs a mapping first, and a rule that says which source wins when two disagree, a survivorship rule.



Two unpublished research documents prepared for the design of AIMS shaped it: a cited survey of the published methods and an advisory synthesis for practitioners. Both were inputs to the specification behind AIMS, which restates them. From different directions, both reached the same default, source-aligned tables.

Hook got closest, and we credit it

Of the six, Hook is the pattern we owe the most, and the one a careful reader will suspect we renamed. It's the only published method that formalizes an ontology over source-shaped data.

In Foad's words, a key set is "a qualifier that is tagged onto a business key, making it unique," and the rule is absolute: "Each and every business key must, must, MUST be qualified with a keyset." No restructuring and a qualified key on every row, gated by the glossary, is the instinct AIMS starts from.

Its limit is the one both research documents name. Integration in Hook means joinability, not resolved entities, where one combined record stands for each customer. Two key sets that identify the same customer stay two keys, and the columns describing that customer are never consolidated. So AIMS took the identity half and made the business key first-class metadata, written with its record source code in front. Then a customer's `aw1t || AU | 00001042` is generated and validated. And two record sources never join until an approved match says they describe the same thing.

"So this is Hook with new names." The identity half shares Hook's instinct, and we'd rather credit it than disguise it. Around it, AIMS adds what Hook leaves to the team: column meanings defined once, a code list behind every code, how much history each table keeps (its history policy), a person on every change of meaning or values, and a data contract per table saying what one row is.

What does every silver layer modeling method leave empty?

A silver table has to answer four questions. Which source record is this row? Which business thing is it about? How does it relate to the others? What does each column mean? The patterns that keep the source's shape answer the first. All but dbt's convention and the mirror answer the second. The canonical model, the vault and Hook answer the third. The fourth column is empty in every row but one. Only the canonical model fills it, and it pays by rewriting every source before anything ships.

	Which source record is this row?	Which business thing is it about?	How does it relate to the others?	What does each column mean?
Canonical third normal form				
Data Vault 2.0				
Hook				
dbt staging convention				
Mirror				
Entity-shaped silver				

filled by rewriting every source before anything ships

empty in every row but one

Figure 3.1: the empty column. Six patterns against the four questions, the fourth column empty in every row but the canonical model's.

Take one attribute from the worked example. The order system's Customer table has a Name column, and the sales system's Account table has an AccountName column. Both mean Customer Account Name. No pattern short of the canonical model has a place to say so. AIMS gives it one: a business attribute, defined once, that both columns are tied to. The tie is called an attribute mapping. Without it, the gap costs you in six places.

- **Gold re-derives the customer's name per model.** Every report decides for itself which column holds it.
- **Cross-source joins work and cross-source comparison does not.** Identity lets you join the two systems, but units, currencies and codes stay unaligned.
- **Sensitivity is classified per column.** Classification and masking belong to the attribute, so mapping once should mask everywhere.
- **Business-level lineage has no edge to follow.** "What breaks if Customer Account Name changes?" needs the column-to-attribute edge.
- **Descriptions and rules are rewritten per column.** Written once on the attribute, they should travel to every mapped column.
- **Marts start from column names.** Dimensions and facts could be drafted from attribute roles (identity, descriptive, measure) if the roles existed.

This is where the AI payoff dies.

Ask an assistant how many customer accounts you lost last quarter. It answers because the status column is mapped to a business attribute with a code list. Take the mapping away and it's guessing from column names again.

A level, not a gate

Neither research document left the fourth column empty by accident. Agreeing what every column means across sources is where the effort of a canonical model lives. Require every column mapped before anything ships and you've rebuilt the upfront cost both set out to avoid, one column at a time. So what fills the empty column without paying that cost again?

We chose a level, not a gate.

Some columns carry weight, because a join, the reading of a code or a governance rule depends on what they mean. They identify a record, point to another record, hold a code or hold sensitive data. AIMS calls these four classes of column critical (identity, relationship, code and sensitive). For them, mapping is mandatory. A column that carries identity but has no mapping can't be joined safely. A code with no code list is a value nobody can interpret, and an unclassified sensitive column is a governance failure.

Everywhere else, mapping is measured as coverage, one visible number per table and per entity, and low coverage warns rather than blocks.

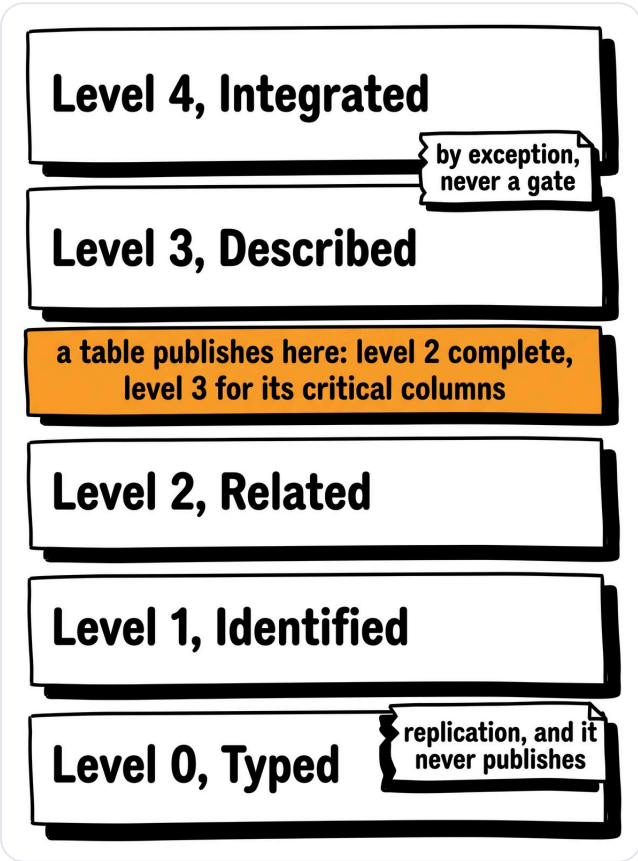
And mapping is cheaper now. That's because an AI proposes every column's match, with its evidence, for a person to review.

AIMS names five meaning levels. Each says how much a table tells its reader, person or machine.

- **Level 0, Typed.** Typed, deduplicated, tested and historized, but silent about what its rows mean. This is replication, and it never publishes.
- **Level 1, Identified.** Tied to its entity, with a sentence saying what one row is (its grain sentence), a readable business key and a history policy. This is Hook's level, the floor.
- **Level 2, Related.** Every reference to another business thing is tied to it, with its role, how many rows can match and what happens when the referenced row is missing.

- **Level 3, Described.** Each column is mapped to a business attribute or marked as having no business meaning, with its kind of value, classification and code list. Critical columns must reach it.
- **Level 4, Integrated.** A shared key across the entity’s record sources, then a golden record (one combined record per business thing), built by exception and never as a gate.

Publishing means gold and agents may read a table. A table publishes at level 2 complete plus level 3 for its critical columns, once it passes its acceptance checklist of identity, values, history, rules and governance. So AIMS conforms in meaning, on tables that keep their source’s shape.

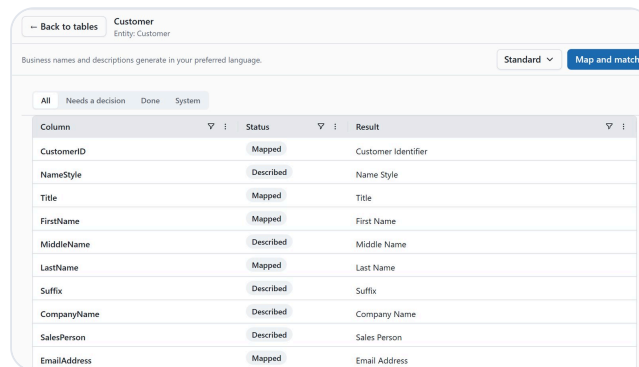


The structural answer	A canonical model fills the empty column by rewriting Name and AccountName into one Customer Account Name before anything ships, and a vault half-fills it later with a hand-designed satellite of conformed values, a business satellite.
What the agent reads	Nothing new yet, but the missing edge from column to business attribute is the one an agent needs most, because it turns two columns into one comparable fact.
Yours to decide	Whether a project, the unit of work a set of tables is organized and generated in, widens the four critical classes, for example to make every monetary amount critical, since the four are only the floor.

In DeltaVault today

DeltaVault, our metadata catalog, holds business attributes. Its Map & Match workbench suggests an attribute match for every column of a table. Each suggestion carries a confidence score and a written rationale, and an accepted match keeps both.

The screen below shows the Customer table from Microsoft's AdventureWorks sample database, tied to the Customer entity. A Mapped column is tied to a business attribute, such as CustomerID to Customer Identifier. A Described column has a business name and a description, but no attribute yet. It's short of meaning level 3.



The screenshot shows the DeltaVault Map & Match workbench for the 'Customer' table. The interface includes a header with a 'Back to tables' link, the table name 'Customer', and the entity 'Entity: Customer'. Below this is a language selection dropdown set to 'Standard' and a 'Map and match' button. A tabbed interface shows 'All' as the active tab, with other tabs for 'Needs a decision', 'Done', and 'System'. The main table lists columns and their corresponding business attributes with status labels.

Column	Status	Result
CustomerID	Mapped	Customer Identifier
NameStyle	Described	Name Style
Title	Mapped	Title
FirstName	Mapped	First Name
MiddleName	Described	Middle Name
LastName	Mapped	Last Name
Suffix	Described	Suffix
CompanyName	Described	Company Name
SalesPerson	Described	Sales Person
EmailAddress	Mapped	Email Address

The business model is the ontology

In this chapter

- **You have built this before.** The ontology an AI agent reads is the business model your team already draws.
- **Five ways to tell things apart.** Entity classes, and the history each one implies.
- **Roles and one sentence per table.** Relationships that name their role, and a sentence that says what one row is.

Before anything is stored, somebody has to say what the business is made of. Every silver layer method that carries meaning starts there, and so does AIMS (AI-ready, Meaning-first Silver), with one difference. The business model stops being a diagram drawn before the tables. It becomes the ontology that the tables, the templates and the AI agents all read.

You have built an ontology before, and you called it a business model

An ontology sounds like something a research lab owns. It's the named things your business runs on, how each is identified, how they relate and what each attribute means, written where a machine can read it. Your team already builds most of it in a modeling workshop and calls it the business model. That business model, plus the mappings that tie silver columns to it, is the meaning layer.

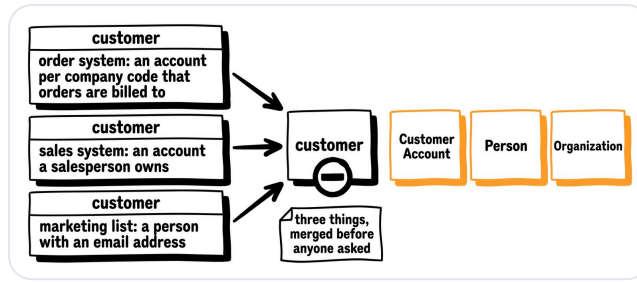
Lawrence Corr's modelstorming puts people first, and John Giles insists a vault starts from a model of the business. Remco Broekmans's Ensemble Logical Modeling walks from business stories to the things they name. Our first book argued for that order in ["Speak before you store"](#): "The semantics that matter most were never columns, and no engine can extract what was never persisted." AIMS adds the precision that lets a template generate from the model and an agent read it with nobody beside it.

Customer Account is not Person is not Organization

The failure a business model guards against is the undefined noun. Ask three systems what a customer is. The order system means an account per company code that orders are billed to. The sales system means an account a salesperson owns. A marketing list means a person with an email address.

File all three under "customer" and the silver layer has merged three things before anyone asked.

So each business entity, a business thing your data is about, gets a definition. A Customer Account isn't a Person, because one person can hold accounts in two companies. It isn't an Organization, because one organization can run several accounts. Where the business needs to connect them, AIMS records a specialization (a Customer Account is a kind of Party) or a relationship, never one identity.



The entity class says how you tell one thing from another

Once an entity is defined, how do you tell one instance from another? The answer comes in five shapes, which AIMS calls entity classes, using entity-relationship modeling's words. Each carries a default history policy, how much history a table keeps (every event, the current record or every version). The AI proposes the policy with a reason, for a person to confirm. An independent entity's past usually matters, so its default is versioned.

Entity class	Identified by	History the AI proposes	In the worked example
Independent	Its own identity attributes, unique in their record source	versioned	Customer Account, Sales Order, Product
Dependent	The parent's identity plus a local identifier	follows the parent	Sales Order Line (its order plus a line number)
Event	Its own identifier, or its participants plus the event time	append	none
Reference	A code in a code list	current	Order Status, Country
Association	The identities of two or more participants, each with a role	versioned, or append when the association is itself an event	none

The class is derived, never picked, from what the model says. An entity typed as an event is an event, and one typed as a relationship is an association. One that's part of another is dependent, one that owns a code list is reference, and everything else is independent. Our first book's [happening test](#) settles hard cases: a happening gets its own entity only when the business names and identifies it separately. A payment with its own reference is one such happening.

Can one entity have two business keys?

The order system identifies a Customer Account by company code and customer number. The sales system uses an account identifier, with no company code.

One key column can't hold both.

So AIMS gives an entity one business key per record source. Customer Account has two, one per system: awlt and wb. Two record sources never join until an approved match says they describe the

same thing.

Relationships carry roles, or reports guess

A Sales Order refers to Customer Account twice, as the sold-to customer and as the bill-to customer. An unnamed line leaves every report guessing. So in AIMS a business relationship carries a role, a cardinality (how many rows on each side can match), an optionality and a policy for a reference that doesn't resolve. Each role gets its own key column on the referencing table.

What happens when the referenced row is missing? A relationship can allow it and count the orphans (references with no match). It can warn and flag the row, or quarantine it, setting the row aside where it's counted and can be fixed.

Warn is the default, because a late row is normal in a lakehouse, and the orphan count shows when late has turned into missing. The worked example overrides it once. An order line whose order is missing is quarantined, because a line without an order has no grain.

[Readers of our first book](#) met this in the Willibald demo dataset, a garden webshop whose customer table and club partner table point at each other. No constraint could say which pointer was membership and which the club's contact person. [A nine-minute meeting](#) said it, and it mattered, because a contact person's orders are attributed to the club.

With roles, membership and contact person become two relationships, and the next reader sees the role instead of a second foreign key.

The business model, as the agent reads it

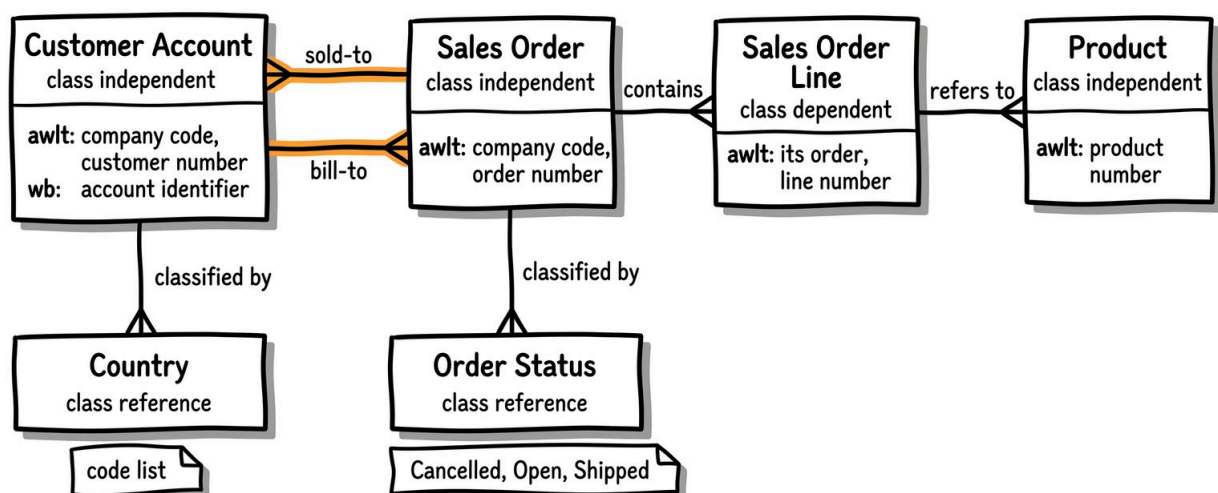


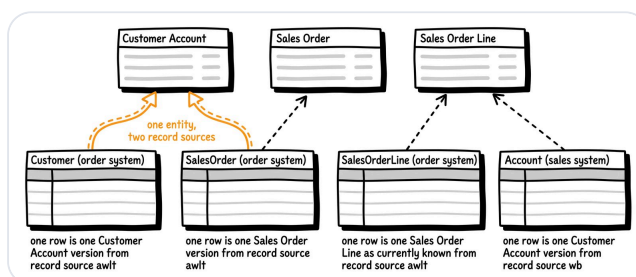
Figure 4.1: the business model as the agent sees it. Six entities with their classes, business keys and record sources, code lists and roles.

One sentence says what a row is

Every table that carries an entity gets one sentence before its column list, its **grain sentence**. It's generated from the entity class and the history policy. A person confirms the sentence, and it opens the table's data contract, the summary of what the catalog holds about it. If a source table mixes grains, say order headers and order lines, the sentence names the column that tells them apart. So an agent filters on it before it counts anything. For the worked example's four source tables:

- **The order system's Customer table:** one row is one Customer Account version from record source awlt.
- **The order system's SalesOrder table:** one row is one Sales Order version from record source awlt.
- **The order system's SalesOrderLine table:** one row is one Sales Order Line as currently known from record source awlt.
- **The sales system's Account table:** one row is one Customer Account version from record source wb.

Two tables carry one entity from two record sources, so nobody will mistake their union for a customer list. And the order line is current while its order is versioned. That's a deliberate override of the dependent class's default that a reviewer can challenge in one line.



The workshop hands over to the meaning layer

The workshop's output is a model a person can read. For its identities and relationships, the meaning layer adds what a machine needs to act without asking. That's the class, a business key per record source, roles with their policies and a grain sentence per table. None of it needs a second workshop.

The structural answer	A vault builds the worked example's six entities as twelve tables (three hubs, three links, four satellites, two reference tables), and the bill-to role alone costs one.
What the agent reads	Each entity's class, definition and business keys with their record sources, each relationship's role and cardinality, and every table's grain sentence.
Yours to decide	Creating or redefining an entity, its business keys, and every relationship's role and cardinality.

In DeltaVault today

DeltaVault, our metadata catalog, holds entities, attributes and relationships as metadata. Each entity has a type, such as Person, Place, Thing or Event. The catalog's relationship diagram asks you to name the role when you draw a line.

Its business modeling workshop turns a plain description of how the business works into proposed things and the connections between them. Only what you keep reaches your business model. The AI never chooses keys. Below, concepts already in the model are marked so, and one judged out of scope carries its rationale.

INDIVIDUAL PERSON

Customer

✓

×

Already in the model

OTHER

Sales Order

✓

×

Already in the model

<- places Customer

✓

×

Invoice

✓

×

Out of scope

> Rationale

<- bills Sales Order

Keep the shape of the source

In this chapter

- **One source-aligned table per source table.** The source's grain and columns, reshaped only to fix an identified problem.
- **Identity before deduplication.** Three questions every row answers, and three things a repeated row can be.
- **History, time and failure on purpose.** Four history policies, the audit trail bronze already keeps, three clocks, and no row dropped silently.

What should a silver table look like before anybody integrates anything? AIMS (AI-ready, Meaning-first Silver) keeps the shape your source already has. Then it decides what that shape leaves open.

Two bars for silver, and the meaning level moves the higher one

Our post ["Show Me Your Silver Layer"](#) set two bars. It called a faithful copy of each source replication. And it said that "if silver holds one governed Customer fed by every system that knows one, the toolset is doing warehouse work." Read literally, that demands one governed record per entity, decided before anything ships.

AIMS keeps the first bar and moves the second. What made the faithful copy replication? It was never the shape. It was that nothing in it said what the rows meant. On the five meaning levels AIMS uses to say how much a table tells its reader, that's level 0, Typed. AIMS never publishes it.

Tie the table to its entity with a business key and its record source, the system the key is unique in. Give every reference its role and every critical column its business attribute. Critical columns are the ones that identify, relate, hold a code or hold sensitive data. The table is then conformed in meaning, with its shape intact.

The governed Customer becomes a golden record at level 4, built only when a specific business requirement needs one record across systems. That's a deliberate revision of that post. Until someone asks, published tables that keep the source's shape are complete silver.

One table per source table, reshaped only for an identified problem

The default fits in one line: one source-aligned table per source table, at the same grain, with the same columns, typed and renamed. The worked example has four, three from the order system and one from the sales system.

Wouldn't a flat, pre-joined table be easier for a model? In [a 2025 study by Kohita](#), it did worse on aggregate questions, because it invites double counting. So the source's grain is the safer shape too.

Typing means the business kind, not the storage kind. A customer number is an identifier, so it keeps its leading zeros and nobody sums it. An order amount is a monetary amount, so it carries a scale and a currency column.

The default materialization is a view over persistent staging, the layer below silver that keeps every version a source delivered. So the view stores no second copy. A switch makes it a table when it's read hard enough to earn one.

Restructuring needs an identified problem: an overloaded table holding unrelated record kinds, a repeating group that hides the grain (twelve monthly amount columns on one row), or one column holding three facts. Nothing else is split, merged, pivoted or flattened, because every restructuring is a decision somebody must keep defending.

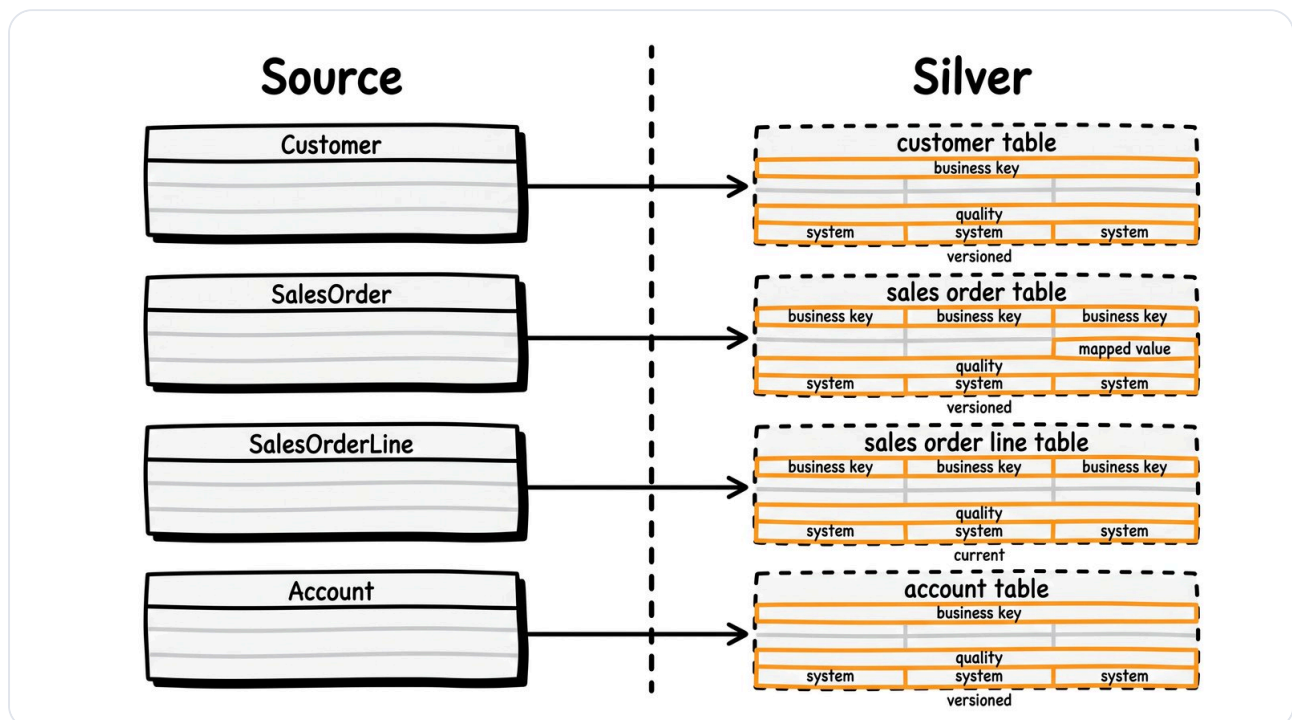


Figure 5.1: one table per source table. The four source tables beside their source-aligned tables, with each one's history policy beneath it.

Why does identity come before deduplication?

Every source-aligned table answers three questions before anything is removed.

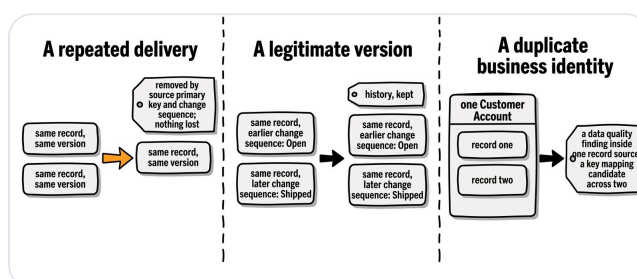
- **Which source record is this?** The source primary key, or a hash of the whole row.
- **Which business thing does it describe?** The business key from its record source.
- **Which version or event is this?** The change sequence: the source's change column, or else the load timestamp.

For the SalesOrder table, that's its primary key, the company code and order number from record source awlt, and ChangedAt, when an order last changed. SalesOrderLine has no change column, so its versions follow the load timestamp.

Those answers make “duplicate” a precise word. A repeated row is one of three things, and only the first is a deduplication problem.

- **A repeated delivery** is the same record at the same version, sent twice. Removing it by source primary key and change sequence loses nothing.
- **A legitimate version** is the same record at a new change sequence, like an order moving from open to shipped. That’s history.
- **A duplicate business identity** is two records describing one business thing. Inside one record source, it’s a data quality finding. Across two, it’s a candidate for a match.

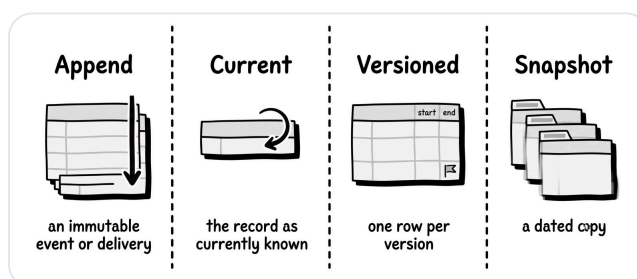
A blanket “keep the latest row per key” treats all three as the first. It discards the order’s history and silently picks a winner between two customers who never should have shared a number.



Choose the smallest history that answers the business question

History is a decision teams often make by accident. The template had a merge, so the table is current. AIMS makes the choice explicit, per table, from four history policies.

- **Append** keeps an immutable event or delivery.
- **Current** keeps the record as currently known.
- **Versioned** is the [Type 2 pattern](#), one row per version.
- **Snapshot** keeps a dated copy, for a source that hands over pictures rather than changes.



In AIMS, the AI proposes each table’s policy from its entity class, ingest mode and change rate, and a person confirms it. In the worked example, Customer, SalesOrder and Account are versioned, each version with derived start and end dates, because their past matters. SalesOrderLine is current, because quantity corrections overwrite and persistent staging keeps the history.

Business history is never the storage engine’s time travel, which records when storage changed, not when the business did, and lasts only as long as a retention setting allows.

Bronze keeps the audit trail before silver starts

Bronze keeps what each source delivered. Its persistent staging keeps every version, insert-only and in the source's shape. That's your audit trail, and it exists before silver starts. A source-aligned table reads from it: typed, cleansed and deduplicated, with its columns mapped in the catalog. Give it a versioned or append history policy and silver is a cleansed persistent staging, history included. With current instead, it's a typed, cleansed, deduplicated staging. Neither needs a vault to be auditable.

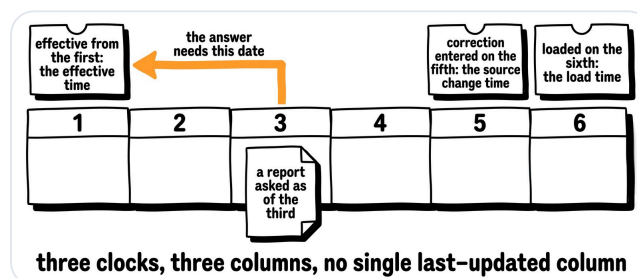
Three clocks, kept in three columns

Every change has three times: when the source changed the row, when the change took effect in the business, and when the lakehouse learned. Merge them and an as-of report goes wrong. Take a correction entered on the fifth, effective from the first and loaded on the sixth. Which date does a report as of the third need? It needs the effective date. No single "last updated" column holds it.

So AIMS names a column for each.

- **The source change time** is the source's own stamp, like SalesOrder's ChangedAt.
- **The effective time** comes from a source column or from a business rule a person approved.
- **The load time** is the load timestamp every load adds. Standing in for a missing change column, it still says only when the lakehouse learned.

In the worked example, no table or rule supplies an effective time, so none is invented.



Nothing is silently dropped

Every quality rule in AIMS carries one of three failure actions.

- **Warn** keeps the row, flags it in a quality column and counts it.
- **Quarantine** diverts the row, with the rules it failed, to a quarantine table and counts it.
- **Block** fails the load or the publish.

AIMS retires "drop", and an existing drop becomes a quarantine. A silently dropped row is a lie in the row count. The totals look clean. As far as gold can tell, the order that failed its cast never happened. A quarantined row can still be counted, read and fixed.

The defaults follow the damage a bad row can do. Identity rules and cast failures quarantine, because a duplicate key corrupts every join and an unreadable amount corrupts every total. Referential rules warn, because a late reference is normal.

A delete policy says what a source's delete does. The sales order table's policy closes the version, so a deleted order keeps its past.

Under the schema evolution policy, a new source column arrives unmapped, tied to no business attribute yet. That's a finding, not a failure. The policy keeps a removed column, marked deprecated, until a person retires it.

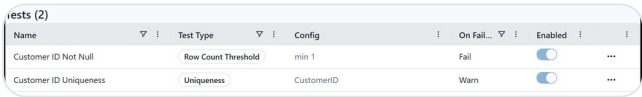
The structural answer [A vault](#) takes the four tables apart into hubs, links and insert-only satellites, so quantity corrections become versions whether anyone needs them or not, where persistent staging had already kept every version.

What the agent reads Each table's source primary key, business key and change sequence, its history and delete policies, its three time columns and every rule's failure action.

Yours to decide Each table's history policy and delete policy, and any rule whose failure action is block.

In DeltaVault today

Data tests are the nearest thing to failure actions that DeltaVault, our metadata catalog, ships today. Each test's On Failure setting is Fail or Warn.



Name	Test Type	Config	On Fail...	Enabled	
Customer ID Not Null	Row Count Threshold	min 1	Fail	☒	...
Customer ID Uniqueness	Uniqueness	CustomerID	Warn	☒	...

A key you can read

In this chapter

- **A key needs its record source.** The system a key is unique in, written into the key with its rules.
- **A key you can read.** One plain-text column per role, record source code first, and why not a hash.
- **The key register.** Every identity an entity has, and why a changed key is a one-way door.

A business key looks like the plainest column in a table. But it packs in the decisions every integration effort ends up arguing about: which system a key came from, what it's unique within and how its parts are cleaned. AIMS (AI-ready, Meaning-first Silver) writes each of those decisions down.

Why does a business key mean little without its record source?

As our post [“Where, Oh Where, Has Integration Gone?”](#) put it, “Integration becomes real only when the agreement is written into a key that every load has to honor.”

A key alone doesn't say where it's unique. In the worked example, the order system's customer number is unique only inside a company code. So the same number under a second company code is a different account. The sales system's account identifier is unique system-wide, with no company code. Join the two systems on their customer columns and you're betting they identify customers the same way. They don't.

A key without its record source is a false join waiting to happen.

A business key carries its record source, with its rules

A business key is the set of values the business uses to identify one thing, in a fixed order, such as a company code and an account number. It's unique only inside the system that issued it, its record source. Written out in a column, the business key starts with the record source code, such as awlt for the order system. Data Vault calls that prefix a business key collision code, and some tools call the whole value an integration key.

Each key part is a column or a fixed value. The key belongs to one entity and states the rules its parts follow.

- **Normalization** cleans a value before it identifies anything: whitespace trimmed and collapsed, upper-cased, and padded to the part's own width. So 1042 and 00001042 are one account.
- **The separator** is a setting every key shares. It joins the parts of a composite key, one between every part. Without it, the parts 12 and 345 and the parts 123 and 45 would make the same key.
- **The null rule** says a missing part makes the identity unknown, never an empty string. An empty string is a value, and two customers with a missing number would quietly become one.

An entity has one business key per record source. So the order system's customer key is one business key wherever it appears, on customers and on every order's sold-to and bill-to. A business key per source table would multiply keys one system already shares. Two record sources never join until an approved match says they describe the same thing.

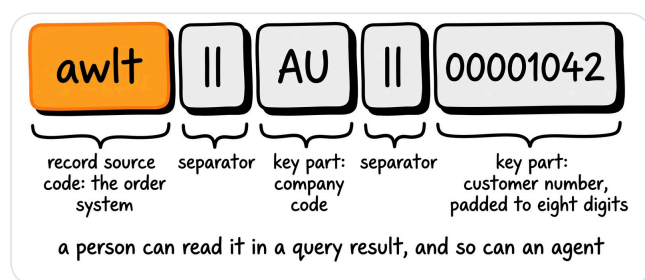
Isn't that the mistake our post ["How raw is your Data Vault?"](#) warned against, a vault in which "each hub keys on whatever its source happened to key on"? No, because a person decides the business key rather than a constraint implying it, and the record source code on every value names its system.

The agreement across systems gets its own shared key

(`customer_account` | `tax_number` | 48123456789 in the worked example) only when a business question needs one record across both. When that key arrives, nothing re-keys, because Sales Order keeps pointing at `awlt` | `AU` | 00001042 .

The business key value spells out where it came from

The business key column carries the key physically, one generated column per role on every source-aligned table. It holds the record source code, then each key part in order, all joined by one separator. For the worked example's customer, that's `awlt` | `AU` | 00001042 .



We made the column physical and the value plain text on purpose. A person can read it in a query result, and so can an agent with nobody beside it. Why not derive identity at query time? Every join and orphan check would recompute the key.

Why not a hash? A hash buys a fixed width, so joins on it stay narrow. But a hash of

`awlt` | `AU` | 00001042 is a run of hexadecimal digits. It names no system, no company code and no customer, and nothing reads it back into them. The readable value answers on sight. Long keys make wide columns, a cost we accepted.

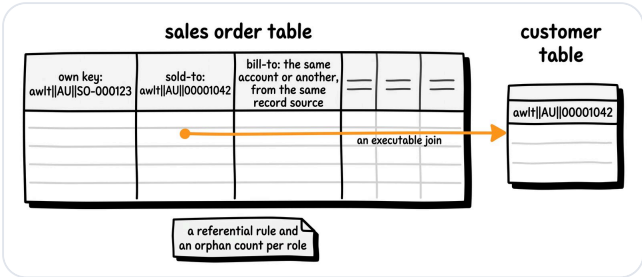
Every role carries its own key

Every source-aligned table carries one business key column for its own identity and one for every entity it refers to, in each role. The sales order table carries three:

- **Its own key**, `awlt` | `AU` | `SO-000123` .
- **The sold-to customer**, `awlt` | `AU` | 00001042 .
- **The bill-to customer**, the same account or another, from the same record source.

A role with its own column gets its own referential rule and orphan count, so a missing bill-to account never hides inside a healthy sold-to figure. It also makes the join executable. The sold-to column already carries the company code beside the customer number.

A dependent entity's key starts from its parent. A Sales Order Line is its order's identity plus a line number.



The key register holds every identity an entity has

Every sold-to key needs one place to find its customer, whichever system the customer came from. That place is the key register, every distinct business key of one entity across its record sources. Customer Account's register unions the order system's customers and the sales system's accounts. So aw1t| |AU| |00001042 and wb| |A-937 are two rows. They stay two rows after a match says they're one account, and both rows carry the same shared key. It's built from the start, a view by default, for three jobs.

- **It's the referential rule's target**, where a sold-to key must find its identity.
- **It counts the orphans**, every reference with no matching identity, per role.
- **A dimension reads it as its natural key** in the mart.

A changed business key is a one-way door

When a business key changes (a new part, a new width, a different column), every value built from it changes. Every match a person typed for one of those values changes too. So in AIMS, nobody accepts a changed key without a stated migration of its values and those matches. That's one reason the business key sits above the approval boundary, where a person approves every decision that changes meaning or values.

The structural answer	A Data Vault carries the key and its source as a hash key and a record source column, and a hash says nothing until it's joined back to its hub.
What the agent reads	Each business key's record source code, parts and padding, the business key column for every role, each entity's key register and each role's orphan count.
Yours to decide	Every business key, its parts and their padding, and the migration when a key changes.

Attributes are where the payoff lives

In this chapter

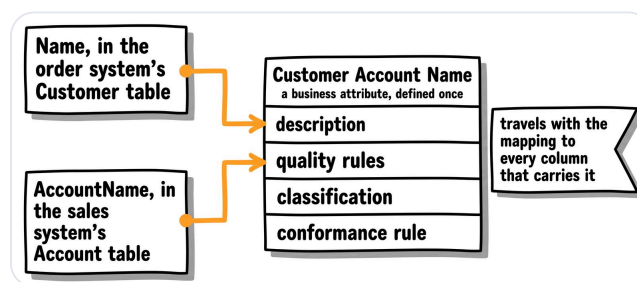
- **Defined once, mapped everywhere.** A business attribute, and the facets that make a column comparable across sources.
- **Keep the source code, add the mapped value.** Code lists, code mappings, and a code nobody mapped.
- **The contract is a projection.** What a published table says about itself, and why it isn't a separate table.

Which column is the customer's name? What currency is each amount in? Which status code means Cancelled in this system? In most estates the answer lives in every gold model that needed it, written again by whoever built the report. Two reports that answered differently both look right.

Identity lets an agent join two systems. Only attributes let it compare them. That's where AIMS (AI-ready, Meaning-first Silver) pays for itself.

The column everyone re-derives is the one nobody mapped

Our first book put the remedy in one line in ["The mapping is the deliverable"](#): "this table is that entity, this column carries that attribute." AIMS makes the second half metadata that every source shares. A business attribute is one meaning, defined once on its entity. An attribute mapping ties each column that carries it to that meaning. The attribute's description, quality rules, classification and conformance travel with it instead of being rewritten per column. The order system's Name and the sales system's AccountName both map to Customer Account Name, so a consumer of either reads one definition.



Facets make a column comparable across sources

Each facet of a business attribute answers a question a consumer would otherwise answer by guessing.

- **Semantic type** says what kind of value it is, such as an identifier, a monetary amount or a date. An identifier is never summed. A monetary amount needs a scale and a currency.
- **Attribute role** says what the column does in the row (identity, descriptive, measure, relationship, status, temporal or classification). The role gives dimensions and facts their first draft.

- **Unit and currency** say where the unit lives: fixed on the attribute, in a column beside the value, or set per record source.
- **Code list** names the reference entity a code belongs to. Like any entity, it relates to others, so Country rolls up to Region along a relationship with a role, never a guessed mapping.
- **Classification** applies your sensitivity, personal data and regulatory schemes once, on the attribute.

Every mapped column inherits its attribute's classification. So you map once and mask everywhere. The attribute's conformance rule applies to every mapped column unless a mapping states its own.

On the worked example's sales order table, the company code and order number are identifiers that make the key. The table's two customer numbers are relationships, and its currency and status are codes on their code lists.

Every column is mapped, source-only or unmapped

An attribute mapping carries a state. The state makes the gap countable.

- **Mapped** columns point at a business attribute, keeping the mapping's origin, confidence and rationale.
- **Source-only** columns deliberately carry no business meaning, such as the sales system's OwnerUser, an internal user identifier, which stays in the table classified internal.
- **Unmapped** columns are debt, and AIMS counts them.

A column a person confirms has no business meaning becomes source-only, so "we looked and it means nothing" no longer reads the same as "nobody looked". A source that departs from the attribute's rule gets a conformance expression on its own mapping, such as a code mapping or a unit conversion. [Our first book](#) named the cost of leaving meaning implicit: "sniffing and inference are contract debt."

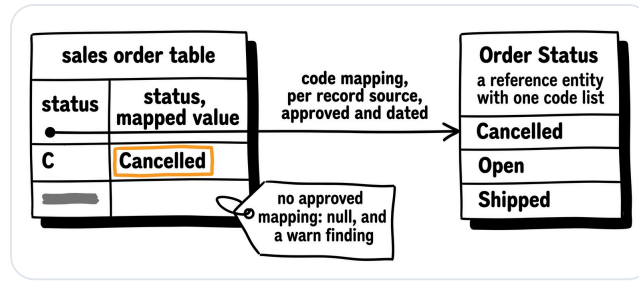
An unmapped column is that debt with a number on it.

Keep the source code, add the mapped value

Codes are where comparison breaks first. The order system sends order status as C , O and S . A second system that sent CXL for Cancelled would need its own translation.

So AIMS names reference entities one by one (Order Status, Country, Currency). Each owns a code list (reference data, in most tools) of business values with definitions, such as Cancelled, Open and Shipped. Per record source, a code mapping (elsewhere, a value mapping) says which source code means which value. A person approves it. It's dated, so a changed mapping keeps its history.

The source-aligned table keeps the source code untouched. Where the attribute is marked for conformance in silver, the table also gains a generated mapped value column. So the sales order table keeps C in status and adds Cancelled beside it. The mart templates apply every other code mapping downstream from the same attribute, so silver doesn't grow a column per attribute. A code with no approved mapping maps to nothing, visibly, as a null and a warn finding, never to a guess.



Why not keep one catch-all list of every code in the estate? Because one Cancelled would have to serve every status list at once, and it couldn't define any of them.

Ask an agent for the value of Cancelled orders by month. It reads amount as a monetary amount whose currency sits in the next column, so it groups by currency instead of adding two currencies together. And it filters on the mapped value Cancelled, not a letter it would have to guess.

The critical columns are mapped, and the rest is measured

The five meaning levels run from 0 Typed to 4 Integrated. A table publishes when every reference to another business thing is tied to it (level 2) and its critical columns are mapped (level 3). Publishing also needs its acceptance checklist to pass: identity, values, history, rules and governance. Then gold and agents may read the table.

Critical columns are the ones that identify, relate, hold a code or hold sensitive data. Your project, the unit of work a set of tables is organized and generated in, may add classes. On the sales order table, six of the nine source columns are critical. The order date, the amount and the change time count toward coverage instead.

Coverage is a measure, not a bar. Below your project's threshold, 80 percent by default, it raises a warn finding and never blocks. The layer ships. The debt stays visible as one number per table and per entity, falling as columns get mapped in use.

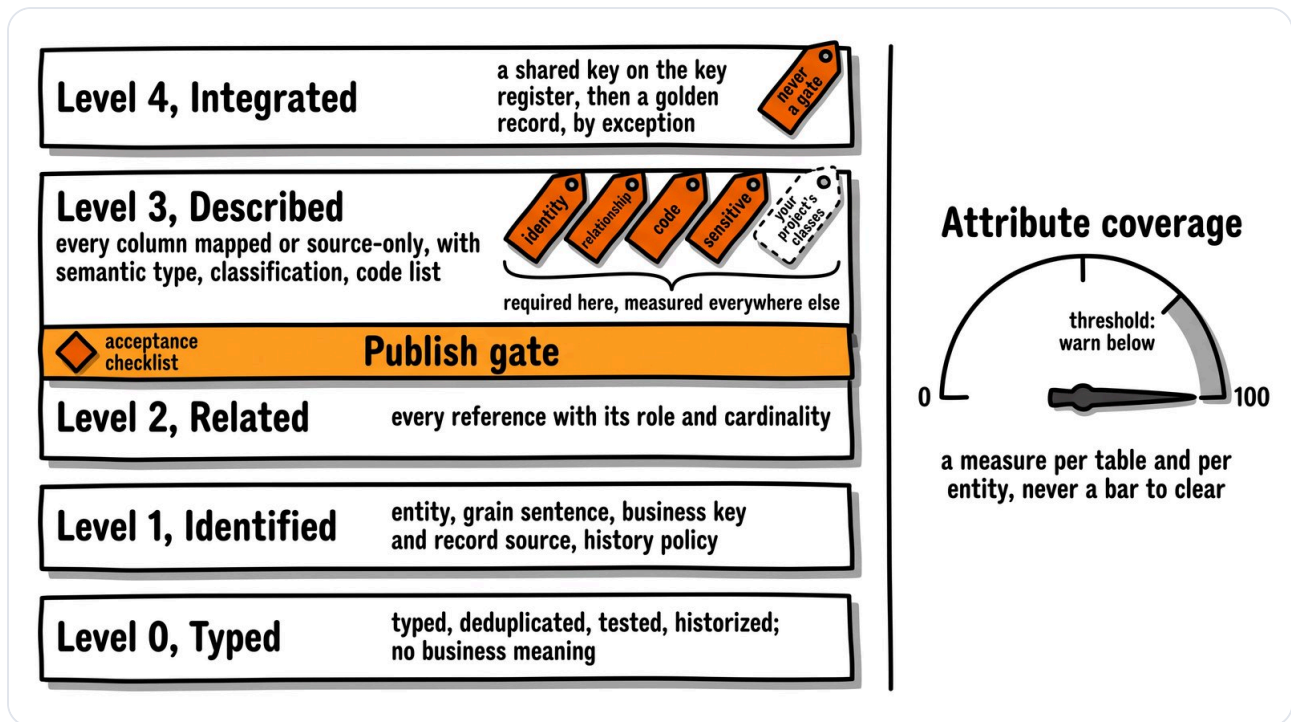
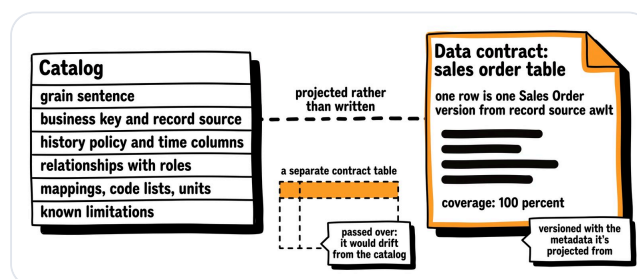


Figure 7.1: the meaning level. Five levels, the publish gate at level 2 with the critical columns of level 3 attached, and coverage as a dial beside it.

Why is the data contract a projection, not a new table?

In AIMS, a published table's data contract already exists, because it's projected rather than written. It's what the catalog holds about the table: owner and steward, entity, synonyms and grain sentence, business key and record source, history and delete policies, time columns, relationships with roles and cardinalities, every column's mapping and semantic type, quality rules with their failure actions, lineage, classification, the freshness target and the known limitations (the unmapped columns and unmapped codes). It's versioned with the metadata it's projected from.

The sales order table's contract opens on its grain sentence: one row is one Sales Order version from record source awlt. It closes on its coverage, 100 percent.



We passed over a separate contract table. It would drift from the catalog the first time someone edited one and not the other. We passed over prose too. One team wrote what its agent needs as markdown files. The modeling happened, but it moved into a paragraph. A projection keeps it a row, which a template can render and a diff can catch.

At LinkedIn, structured table and column attributes lifted the share of an agent's answers rated correct from 11 percent to 24, and example queries lifted it to 49. But free-text notes about the business

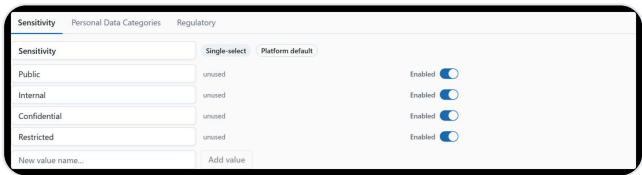
lowered it to 42. Meaning tied to tables and columns helped, there as in the text-to-SQL benchmarks. Exporting the contract in the [Open Data Contract Standard](#)'s shape is later work.

The structural answer	A canonical model or a vault's business satellite translates codes inside its load logic, so the mapping is readable by whoever reads that load.
What the agent reads	Each column's mapping, semantic type and other facets, each code's dated mapping, coverage, and each table's contract.
Yours to decide	Any conformance that changes a value, every classification sign-off, and adding a critical class to your project or removing one it added.

In DeltaVault today

DeltaVault, our metadata catalog, already holds business attributes. A mapped column shows its attribute's classification as derived. The catalog's metadata is serialized to readable YAML that's [reviewed, committed and versioned in Git](#).

Below are its classification settings, with the Sensitivity scheme open.



Integration by exception

In this chapter

- **Complete without integration.** No shared key until a business requirement asks for one record.
- **Matching and the golden record.** Approved match rules, a shared key and a survivorship rule per attribute.
- **Requirements, not preferences.** What each business need calls for, and why audit needs nothing new.

When does a silver layer need integration structures at all? AIMS (AI-ready, Meaning-first Silver) answers “when a question needs one”, not “before anyone may read it”. And it builds them beside the tables, never instead of them.

A finished silver layer needs no integration structures

In the worked example, four source-aligned tables, one per source table, hold the order system’s customers, orders and order lines and the sales system’s accounts. Every row carries its readable business keys. Every reference has a role. The critical columns, the ones that identify, relate, hold a code or hold sensitive data, are mapped. Gold can read every table that passes the publish gate. Nothing in them is waiting for integration.

Silver layer integration is the last step you take when a question needs it, not the entry fee you pay before anyone may read the data. It’s level 4, Integrated, on the five meaning levels that say how much a table tells its reader. Placing it there revises our earlier silver posts. They asked for one governed Customer fed by every system that knows one.

Matching says two keys are one customer account

The finance team needs one customer list across both systems. But Customer Account has two business keys, one per record source, and they never join until an approved match says they describe the same thing.

A match states that two or more business keys denote one real-world instance of an entity. A person approves the match rules and types a manual match for an instance no rule can decide. The key register already lists every distinct business key of one entity across its record sources. Each run gives every row on it a shared key and what matched it. A manual match wins over every rule while its effective dates hold. A key nothing matched keeps itself as its shared key.

In the worked example, the rule is an exact match on a tax number. It finds that customer 00001042 under company code AU in the order system and account A-937 in the sales system are one customer account. Both rows get the shared key `customer_account || tax_number || 48123456789`, and each says the tax number rule matched it.

A match is valid only between keys of the same entity. An organization and a billing account are connected by a relationship. A match between them would claim two things are one.

We left probabilistic entity resolution out on purpose. A match comes from an exact-match rule on a shared natural key or from a manual match a person types. The AI may suggest a rule, never a match, and no rule counts until a person approves it. Why draw the line there? A guessed identity is wrong in every join downstream of it. And nobody downstream can see which joins it touched. So AIMS would rather leave two keys apart than merge them on a likelihood, and it leaves a master data management engine out of scope.

The golden record sits beside what it summarizes

Then Customer Account gets a golden record beside its key register, one combined record per shared key. It holds the record sources that contributed to each, and a survivorship rule for every attribute it carries. A survivorship rule says which source's value wins, by precedence of record source, by the most recent change, or by the first value that isn't null in a stated order.

- **Name** survives from the sales system, with precedence wb, then awlt.
- **Status** survives from the order system, the only source that has it.
- **Country** survives from the sales system.

Survivorship works per business attribute. The rule can rank wb above awlt for Name only because both systems' name columns are mapped to the same attribute, Customer Account Name. Without that mapping, they're just two unrelated columns that happen to look alike. Our post ["The Lakehouse Ended One Satellite Per Source"](#) made the same point about the vault: "Conform the business attribute first, or you never get a common satellite at all."

What makes this integration by exception rather than integration by rebuild? Sales Order keeps pointing at `awlt | |AU| |00001042`. Gold joins through the key register when it needs the shared key, and through the business key when it doesn't. The golden record sits beside the four source-aligned tables. Nothing about them changed.

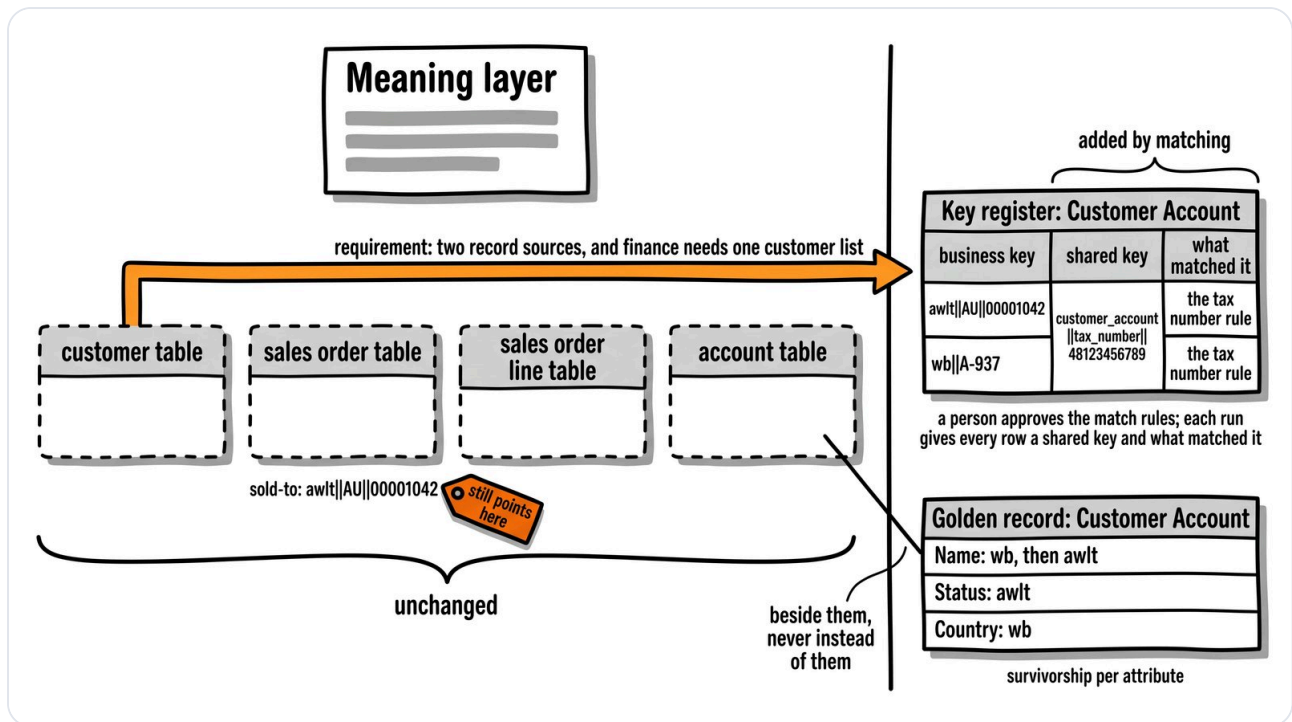


Figure 8.1: integration by exception. The four source-aligned tables unchanged, the key register carrying its shared key, and the golden record beside it.

Which business requirement calls for more?

Customer Account met one business requirement. AIMS names the others, and what each calls for. So the build for each business area follows a requirement you can point to, not a preference nobody wrote down. [Our first book](#) left the choice of shape to you. Now that choice can follow the requirement.

Business requirement	Build
One record source per entity, consumers doing analytics	Source-aligned tables only, levels 1 to 3
Two or more record sources per entity and a consumer who needs one record, or an entity many gold products reference	A shared key on the key register, then a golden record, level 4
Every source version kept for audit	Nothing new: bronze already keeps every version, before silver starts
Consumers that need a stable interface independent of any source	A golden record, a view or table beside the key register
Consumption	The mart, always

Why does audit need nothing new? An auditor asks what each source sent and when. Persistent staging, the layer below silver that keeps every version a source delivered, already answers that. It sits in bronze, insert-only and in the source's shape. A source-aligned table with a versioned or append history policy, the history chosen for it on purpose, shows the same history in silver, cleansed.

What about the things a vault was built for besides audit? Many volatile sources and parallel loading need nothing new either, because persistent staging loads one table per source table, silver keeps one table per source table, and each new source adds its own. A team organized around Data Vault is a preference, not a requirement. The mart closes that table, because consumption always needs one.

The mart reads the meaning layer

The mart reads the meaning layer, the business model plus the mappings that tie silver columns to it. The key register gives a dimension its natural key. Roles become role-playing dimension references, and the shared key, where matching exists, becomes a [conformed dimension](#)'s key map.

The structural answer	A vault holds equivalence in a same-as link and the conformed view in a business hub and satellite, each with its own load, and survivorship sits in that load logic.
What the agent reads	The register's shared keys with what matched each, every rule, which tables are summaries and which are sources, and the matching coverage, the share of keys something matched.
Yours to decide	Every match rule, manual match and survivorship rule, and when a business need calls for a golden record.

AI proposes, a person approves, templates generate

In this chapter

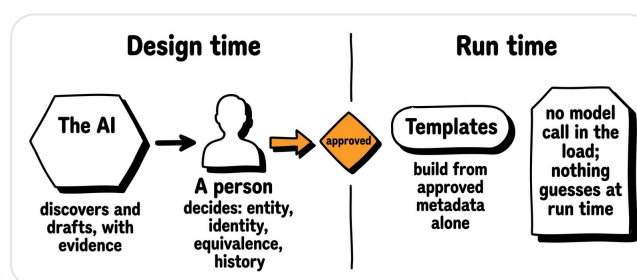
- **Design time and run time.** Why the AI drafts, a person decides and templates build.
- **Approvals compound.** Every approval becomes the next suggestion's demonstration.
- **The approval boundary.** Nine kinds of decision a person approves, whatever the AI recommendation workflow.

Answering a question about your data is run time. So is the load that feeds it. Deciding what that data means is design time, and it matters more if you build it. AIMS (AI-ready, Meaning-first Silver) splits the work three ways. The AI drafts, a person decides, and templates build from approved metadata alone.

Why does the split fall where it does?

A model is reliable at drafting: descriptions, types, candidate mappings and classifications. It's unreliable at what a business key really identifies, at the rules your business never wrote down, and at whether two identifiers in two systems are the same customer. Those decisions change meaning or values. That's where being confidently wrong costs you most.

So the AI does most of the hours: the discovery, the profiling, the drafting, the first version of every mapping. A person does most of the risk: identity, equivalence, history, anything that changes a value. That's controlled acceleration, and it's how AIMS does AI governance. The repetitive modeling work gets cheaper while you keep control of meaning, standards and approval.



A suggestion reads what an analyst would read

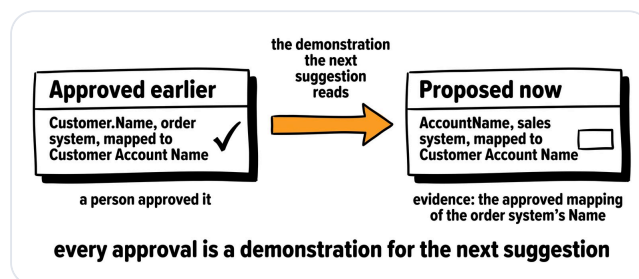
Give a model the inputs you'd give an analyst. That means a source table's profile (statistics on each column's values) and its declared keys. The inputs also include the meanings a person has already approved for sibling tables, the other tables from the same source.

How well can a model map a database to an ontology? In September 2026, [Thombre, Patwardhan and Sarawagi](#) measured large language models on a standard benchmark for the task. In one pass, a model scored 0.10 on the benchmark's measure, a harmonic mean of precision and recall where 1.0 is a complete and correct mapping.

Their staged pipeline first rules out impossible candidates with fixed rules (symbolic constraints), before the model ranks the rest. Then it shows the model worked examples, called demonstrations, grounded in mapping patterns. It reached 0.56, more than half of the mapping, with a person checking the rest.

Here AIMS makes an inference the paper doesn't. The paper's demonstrations are generated automatically. In AIMS, they're your prior approvals. So in the worked example, the suggestion for the sales system's AccountName reads the approved mapping of the order system's Name to Customer Account Name. That's our reading, not the authors' claim. It's also why the ontology compounds. Every approval is a fact for the agent and a demonstration for the model that proposes.

A finished hub taught the next hub nothing.



Every suggestion shows its evidence

A suggestion is a claim with its reasons attached. A suggestion with neither evidence nor a stated assumption isn't shown. That's because a guess that can't say why should never reach your attention.

In AIMS, the model never faces the whole search space. As in the study, fixed rules prune the candidates a column's type or profile rules out. So the model chooses among plausible answers. A fixed self-check then catches a confident answer the profile disproves, before anything reaches the queue.

The approval boundary, decision by decision

AIMS draws a line where models read badly. Above it, a person approves each of these nine kinds of decision, however confident the model. That line is the approval boundary.

- **Creating or redefining a business entity**, a business thing your data is about.
- **A business key**, what identifies one thing in its record source.
- **An equivalence**, the decision that records in two systems are the same thing, stated in match rules and manual matches between business keys.
- **A relationship's cardinality or role**, how many rows on each side can match and the part each side plays.
- **A history policy or a delete policy**, how much history a table keeps and what a source's delete does to it.

- **Any conformance that changes a value**, bringing a source's values in line, such as a code mapping, a unit conversion or a cleanup beyond trimming spaces and fixing case.
- **Signing off a classification**, such as how sensitive the data is.
- **A survivorship rule**, which source's value wins when two disagree.
- **Adding a critical class to a project, or removing an added one.** A project is the unit of work a set of tables is organized and generated in.

The four built-in critical classes are the columns that identify, relate, hold a code or hold sensitive data. Below the boundary, a suggestion may be accepted in bulk and reviewed later. That covers descriptions, business names inside your naming convention, high-confidence mappings on columns outside the critical classes, and rules that only warn. A wrong answer there changes no value. Fixing it later is cheap.

How much of that lower queue must a person click through? That's the AI recommendation workflow. A person can review each suggestion, accept suggestions in bulk while still confirming them, or let suggestions at or above a set confidence go through automatically.

The AI recommendation workflow is a throughput setting. The boundary is a trust setting. A throughput setting must never be able to spend trust.

So no option, automatic acceptance included, can accept a decision on the list above.

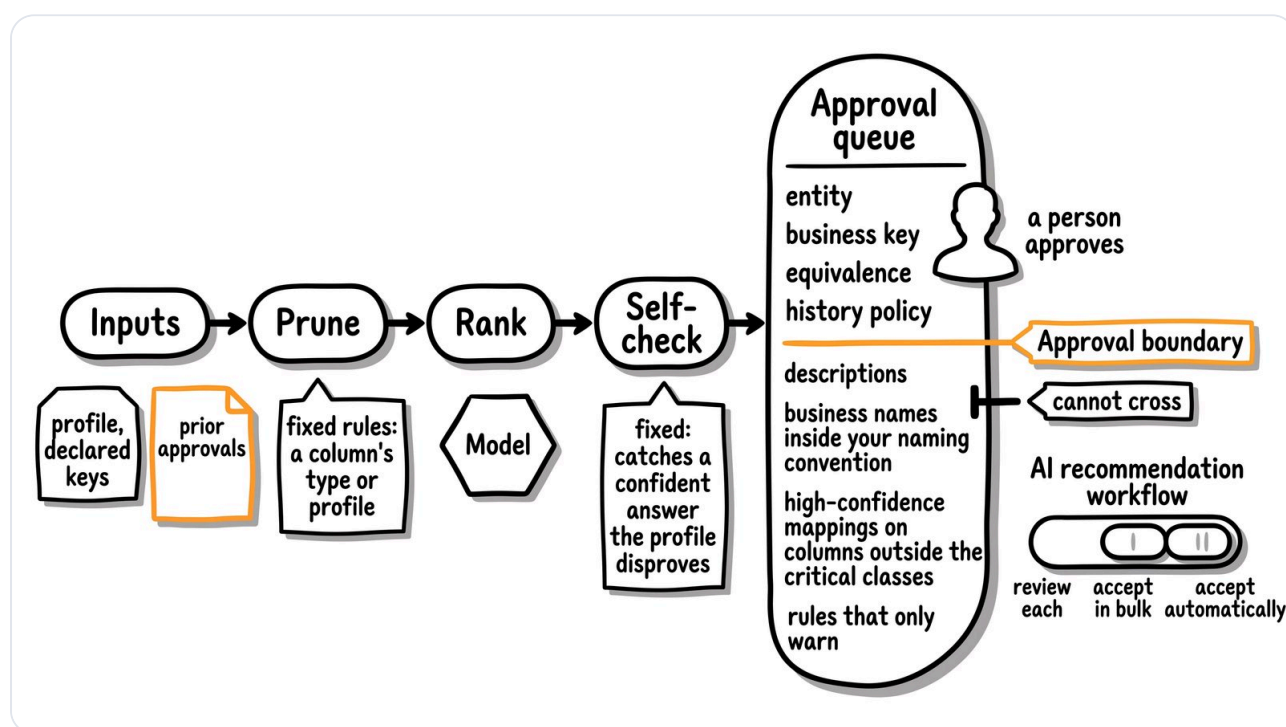
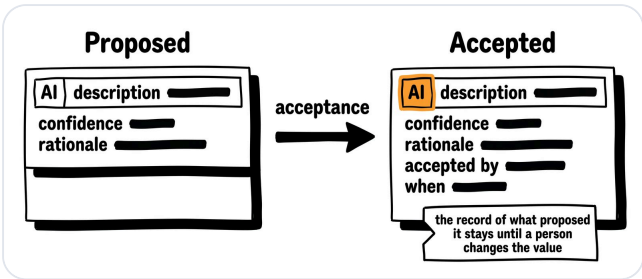


Figure 9.1: suggestions with evidence, and a boundary the AI recommendation workflow cannot cross.

An accepted guess is still a guess

Accepting a suggestion means “good enough for now”. It never means whoever accepted two hundred mappings in bulk vouched for each one. Our post [“Keep AI-generated metadata marked after you accept it”](#) put the principle in a heading: “Accepting is a decision, not authorship.”

So in AIMS, a value the AI proposed keeps the record of what proposed it, with its [confidence score](#) and rationale, until a person changes the value. That record is its provenance. It survives acceptance. The audit trail adds who accepted it and when. Both are kept because the next reader of a mapping is often another AI run.



Templates generate, and nothing guesses at run time

Once a person approves, the model leaves the room. In AIMS, approved metadata renders through templates into the tables, loads and tests. No model call happens in the load, and no template reads anything a person hasn't approved.

That's what makes the boundary worth drawing. If a model ran inside the load, every approval would be advice to a process that could reinterpret it at three in the morning.

[A 2026 study](#) measured the difference. Given exact metric and dimension definitions as text but no enforced policy, an agent still leaked data across tenants on 35 percent of a hundred reporting questions. That's because a rule it can only read is advice. With the same definitions governed and the join paths and access policy enforced, it leaked on none.

The loop closes with numbers a machine computes the same way on every run, from quarantine rates to counts of references that point at no record. Where's the number that says all of this is faster? We don't have one. And we won't borrow the time-to-value figures in circulation, which are vendors' marketing.

The structural answer	A vault team wrote its mapping sheet by hand, so the judgment that a column was a business key lived in a modeler's head, with no evidence beside it.
What the agent reads	The evidence on each suggestion and the provenance on each accepted value, so it can say which mappings a person decided and which are still guesses.
Yours to decide	How much of the AI's work below the boundary a person reviews one by one, knowing no setting reaches above it.

In DeltaVault today

DeltaVault, our metadata catalog, already keeps provenance this way, in the catalog itself, whichever platform you generate for. When you accept a value its AI proposed, the value keeps an AI badge, with the model's confidence and rationale where it gave one. The badge stays until a person edits that field.

The AI recommendation workflow is there too, as a setting. Review each suggestion has a person approve AI matches one at a time. Allow bulk review is the default, and a person still confirms every match. Auto-accept high-confidence matches applies matches and drafted attributes at or above the confidence floor, with no person reviewing them first, and the product marks it as not recommended. An automatic accept is recorded under the same audit trail as a manual one.

The silver templates AIMS describes aren't shipped in DeltaVault yet.

AI recommendation workflow

- ☐ **Review each suggestion**
Approve AI matches one at a time in Guided Review. Bulk accept is turned off.
- ☒ **Allow bulk review (default)**
Review AI matches in a grid and accept many at once. A person still confirms every match.
- ☐ **Auto-accept high-confidence matches (Not recommended)**
Apply matches and drafted attributes at or above the confidence floor automatically, with no person reviewing them first.

The agent reads the contract

In this chapter

- **One question, six decisions.** A business question traced through the metadata that answers it, and the clause it declines.
- **The contract is the briefing.** What an agent reads in a data contract, and why the known limitations matter most.
- **Levels and checkable answers.** What an agent can do at each meaning level, and an answer that cites where it came from.

How many customer accounts in Australia had an order move to Cancelled last quarter, by sales owner? A question like that takes several steps. An AI agent that plans and runs those steps itself, rather than returning a chat answer, makes a decision about meaning at every one. That's agentic reporting.

The agent is yours or your platform's. AIMS (AI-ready, Meaning-first Silver) designs what it reads.

One question touches six decisions a person already made

In AIMS, each clause of the question reads a decision a person already approved.

- **Customer accounts** are Customer Account, a business entity identified by two business keys, one per record source.
- **An order moving to Cancelled** is a new version of a Sales Order, one row per version. So the agent counts accounts, never rows, and reads `C` as Cancelled through the code mapping.
- **Last quarter** falls on the source change time, not the load time or the order date. Nothing supplies an effective time, so the answer says its dates are when the order system recorded the change.
- **Which account** is the sold-to role, the account that placed the order, carried in its own key column, not the bill-to.
- **In Australia** is the trap. The `AU` inside the order system's key is a company code, the key's first part, typed as an identifier. Nothing in the contract says it's a country.
- **By sales owner** has no business attribute. The only candidate, the sales system's OwnerUser, is marked source-only, carrying no business meaning, and classified internal.

Country lives only on the sales system's account. So the agent crosses systems through matching, which ties two systems' keys to one shared key. Customer `00001042` under company code `AU` and account `A-937` share the key `customer_account || tax_number || 48123456789`.

The matching coverage, the share of the entity's keys something matched, tells the agent how far that path reaches. The answer reports the accounts it can't place as unknown.

The agent answers the rest of the question. For the last clause, it says no business attribute for sales owner is mapped and that the source column OwnerUser is internal.

That refusal is the most useful line in the answer. A guess would have grouped the accounts by an internal user identifier and called it sales ownership.

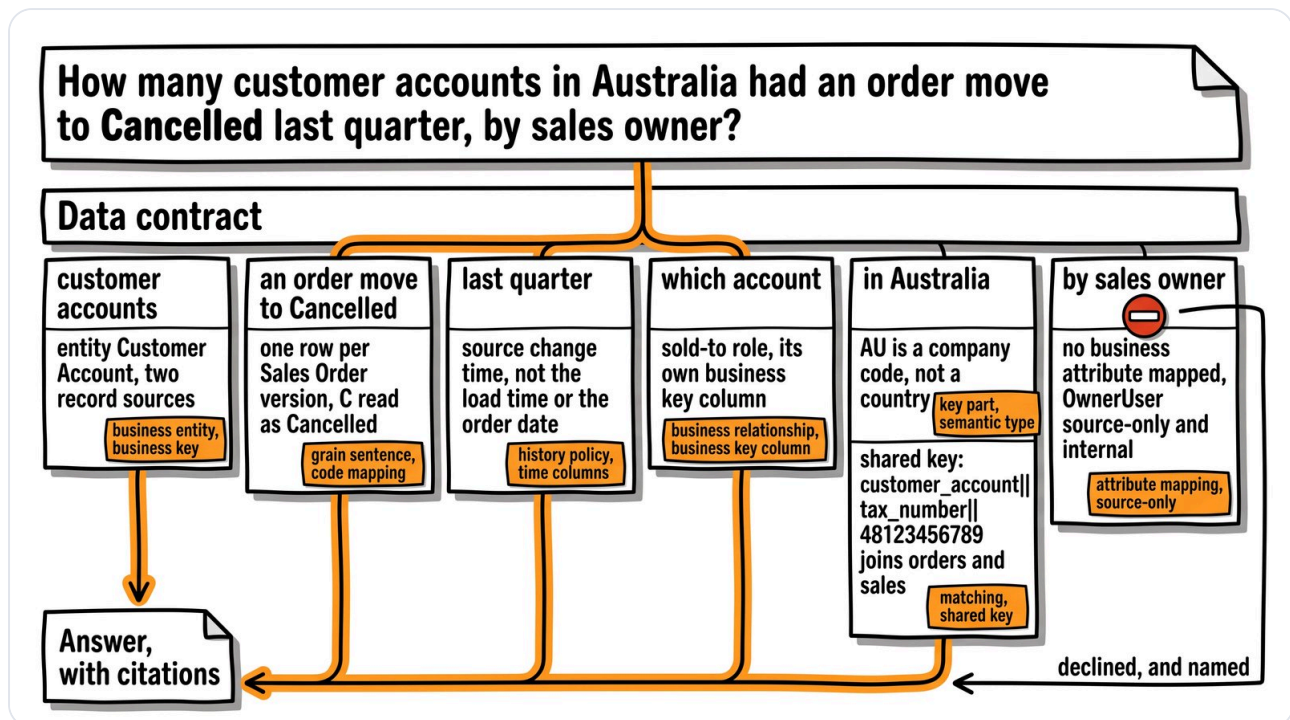


Figure 10.1: what the agent reads, from the question to the contract fields it touched and the clause it declines.

What does an AI agent read in a data contract?

Most of those steps read one document, the data contract, in AIMS a projection of what the catalog holds about a table. Read from the agent's side, it's a briefing.

The owner says whom to ask, and the synonyms say which of the asker's words mean this entity and these attributes. The grain sentence says what to count. The business key and its record source say what's unique and where. The history policy and the time columns settle which version and which date. The roles and cardinalities show how to join without doubling a row.

Each column's mapping, with its semantic type, code list, unit and currency, says what a value means and what may be added to what. The rules and their failure actions say what was flagged, diverted or blocked. Lineage says where a column came from, and the freshness target says how old the data may be. The known limitations say what's missing.

[Our first book](#) put the rule in a heading: "The AI cannot see what was never captured." The contract is that capture, projected before anyone asks, so an agent never has to [reconstruct meaning from column names](#).

Third normal form and Data Vault settled which object, column and join path a question needs by restructuring the sources. In AIMS the agent reads them off the contract, from tables that kept their shape.

The known limitations keep an agent honest

That last field matters most. The known limitations list every unmapped column and every unmapped code, so an agent can say “I cannot answer that part” [instead of guessing](#). A source-only column, like OwnerUser, lets an agent decline too.

What if the order system starts sending a status code nobody has mapped? It maps to nothing, visibly, as a null mapped value and a warn finding. The contract lists the gap. So the agent can say its count leaves out orders in a status nobody has defined, rather than leaving them out silently.

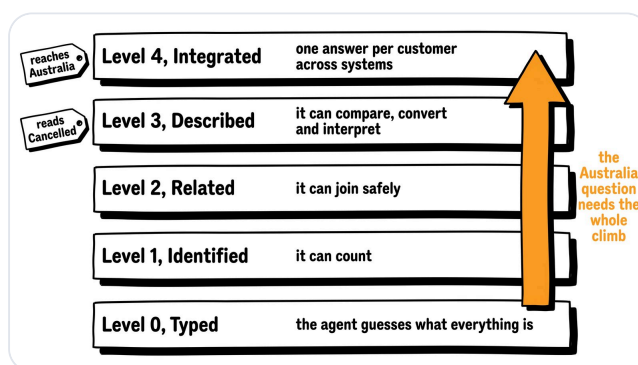
A refusal that names what’s missing tells a person what to map next. A confident guess never does.

The meaning level says how much an agent can answer

The five meaning levels, which say how much a table tells its reader, also measure how much of a question an agent can answer without guessing. At level 0, Typed, it guesses what everything is. At level 1, Identified, it can count, because it knows what one row is. At level 2, Related, it can join safely, because every reference has a role and a cardinality. At level 3, Described, it can compare, convert and interpret, because the columns carry their meaning. At level 4, Integrated, it can give one answer per customer across systems.

The worked example’s question needs the whole climb: level 3 to read Cancelled, level 4 to reach Australia.

So what number can a data leader ask for? In AIMS, it’s attribute coverage. Per table and per entity, it says how much of the estate an agent can read rather than guess. It rises with every column mapped or marked source-only.



Point the agent at the mart, and keep metrics there

In AIMS, consumption always gets a mart. Where an agent runs its own query, point it at the mart, or at the few published tables a question needs, never at every table in silver.

The platforms’ own guidance keeps an agent’s tables to a handful. In [a 2025 study by Kohita](#), each join a question needed roughly halved a model’s accuracy with no worked examples to copy. But in that study, a flat table did worse on aggregate questions.

In AIMS, the contract's roles and cardinalities are the join paths the agent reads, so it never invents one. The mart is drafted from the same metadata and designed by people, so a dimension's Cancelled is the contract's Cancelled.

One kind of meaning the contract never carries is a metric. Business attributes in silver are operational facts. Net Revenue, Active Customer and churn are definitions, so they belong to the consumption layer, in the mart or the platform's semantic layer, built from mapped attributes. The churn rate built on a count of closed accounts is a definition, written once where the agent reads it. A definition nobody has written is a clause the agent declines like any other.

Every platform builds its own semantic layer for AI agents

Each platform's agent reads a curated semantic layer over the tables, and each keeps it inside its own walls.

On Databricks, Genie reads its knowledge store and generates from a table's comments and the primary and foreign keys declared on it. In Fabric, a data agent's descriptions of a lakehouse table are entered in the agent itself, in preview, or live in the reporting tool's semantic model. In that model, Copilot reads the first 200 characters of each.

Snowflake's [semantic views page](#) says "Currently, Cortex Agents reads the information captured in the semantic view definition and generates the SQL against the physical tables directly." So a table's own comment reaches it only through that definition.

Meaning written once still has to be rendered into each of those objects, by a template that knows their limits. In AIMS the meaning layer is written first, on no platform, instead of annotating tables after the data landed. That's how an agent on Databricks, Snowflake or Fabric reads the same contract.

The templates differ, but the meaning doesn't.

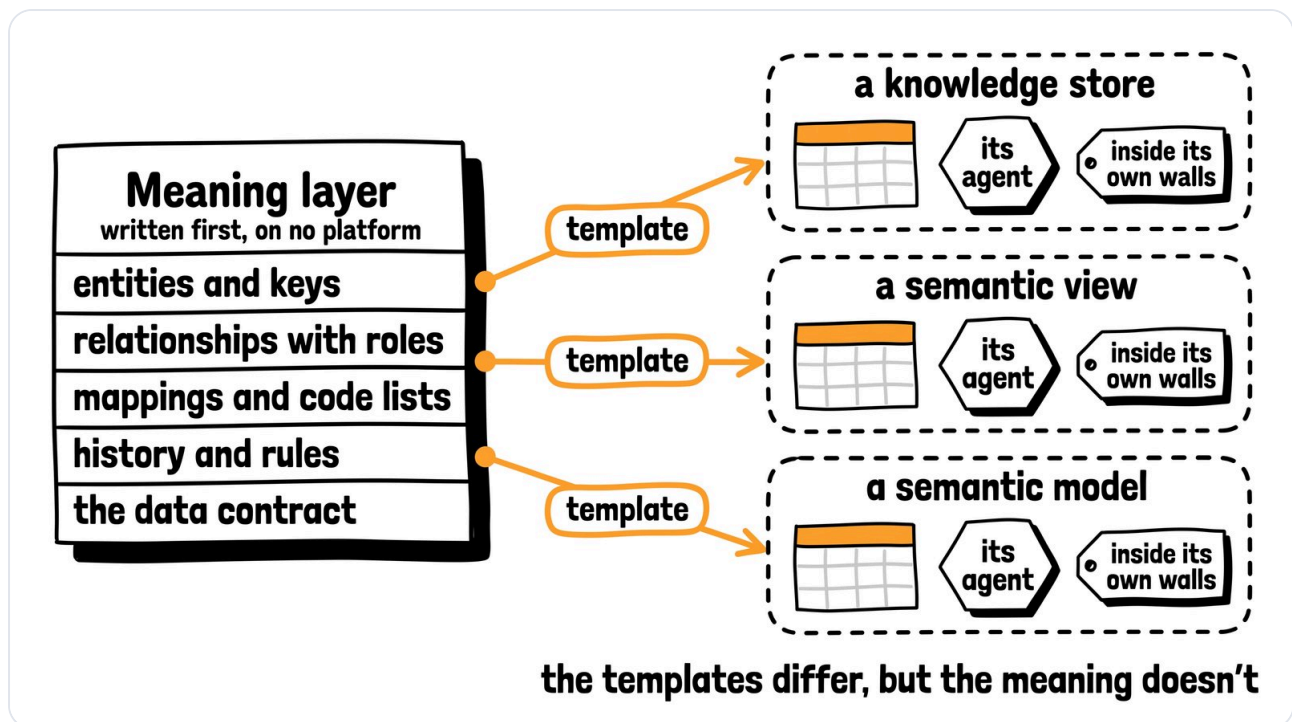


Figure 10.2: a platform-neutral ontology generates what the others annotate.

An answer you can check cites where it came from

What can the answer to the Australia question cite? It can point to the source-aligned sales order table, the source table under it and the approved code mapping that made C mean Cancelled. It can cite the shared keys and the rule behind them, and say which mappings a person decided and which remain the AI's guesses. And it can name the rows the identity rule quarantined and left out of the count. In AIMS, every row on that trail carries its provenance, the record of what proposed it, and an audit trail of who approved it.

That's what turns an agent's answer from plausible into checkable.

The structural answer	A vault team answered the Australia question with a point-in-time table, a bridge, a same-as link, a business satellite and a semantic model hand-built in the reporting tool.
What the agent reads	The data contract in order, from the owner, entity and grain sentence to the known limitations, with the shared keys and the matching coverage beside it.
Yours to decide	Which questions an agent may answer without a person reading the answer first, which only while the known limitations are empty, and each entity's freshness target.

Seven steps to a meaning layer

In this chapter

- **Seven steps in order.** One business area taken up the meaning levels, each step ending in a check.
- **Done when, and how it fails.** The check that closes each step, and the mistake that sinks it.
- **One area before the next.** Why approvals compound, and how to know the layer works.

Where do you start on Monday? You take one business area's silver layer up the five meaning levels, which say how much a table tells its reader. AIMS (AI-ready, Meaning-first Silver) breaks that climb into seven steps. At each, the AI proposes and a person approves, along the line [AI proposes, a person approves, templates generate](#) draws.

Scope it the way [our first book's closing chapter](#) did, one business area end to end before the second starts. Read the steps as dependencies, not calendar slots.

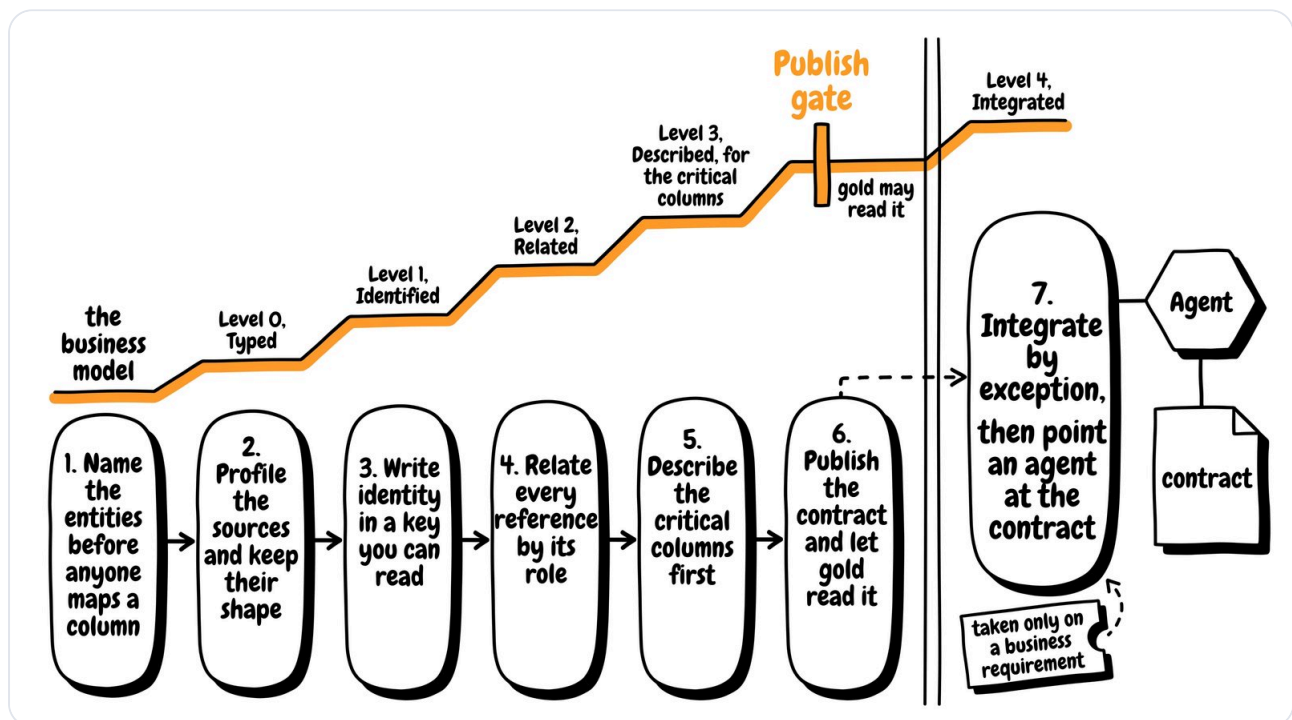


Figure 11.1: seven steps up the meaning level, with step 7 taken only on a requirement.

Step 1: Name the entities before anyone maps a column

Run a [business modeling workshop](#) for that business area. Give each entity its type, its definition and a draft of its [grain sentence](#), the line that says what one row will mean. See [The business model is the ontology](#).

You are done when every entity the sources carry has a type, a definition and a draft grain sentence your business expert has agreed. **The way this fails is** the undefined noun, three systems saying “customer” about an account, a person and a billing party.

Step 2: Profile the sources and keep their shape

While the workshop runs, engineers profile the business area’s source tables, collecting the column statistics the suggestions read. Each source table gets one source-aligned table at the same grain. Its default materialization is a view over persistent staging, the layer that keeps every version delivered. See [Keep the shape of the source](#).

You are done when every source table has its source-aligned table at level 0, Typed, and its profile is accepted. **The way this fails is** “we will model later”. Level 0 is replication. The day somebody reports from it, the meaning gets invented inside the report.

Step 3: Write identity in a key you can read

For each entity and record source, the AI proposes a business key, its parts in order. A person approves each with its padding, history policy and delete policy. One customer’s key reads
aw1t | |AU| |00001042 : record source code, company code, customer number. A person confirms each grain sentence, now naming its record source. That’s level 1, Identified. See [A key you can read](#).

You are done when every business key passes its uniqueness test and every table has a chosen history and delete policy. **The way this fails is** a hash nobody can read, from a record source nobody wrote down. So the first clash between two systems’ customer numbers surfaces in a report.

Step 4: Relate every reference by its role

Every reference gets a role, a cardinality and a policy for a reference that doesn’t resolve, such as warn. Each role gets its own key column. Each entity gets its key register, every business key it has. That’s level 2, Related. See [The business model is the ontology](#).

You are done when every relationship has a role and its orphan count is reported. **The way this fails is** two columns both called customer, joined on whichever one the analyst found first.

Step 5: Describe the critical columns first

Mapping starts with the critical columns, the ones that identify, relate, hold a code or sensitive data. Each maps to a business attribute with its semantic type and unit or currency. A code column gets its code list and an approved code mapping, such as the order system’s, where C means Cancelled. Level 3, Described, asks every column to be mapped or marked source-only. But the gate asks it only of the critical columns. The others’ coverage is measured, never gated. See [Attributes are where the payoff lives](#).

You are done when every critical column is mapped or marked source-only and the attribute coverage number is known. **The way this fails is** mapping everything before shipping anything, the canonical

model's cost in a new coat.

Step 6: Publish the contract and let gold read it

The data contract is a projection of what the catalog holds after steps 1 to 5. So publishing is a review, from the grain sentence to the known limitations. A table that passes the publish gate (level 2 complete, the critical columns at level 3, and the acceptance checklist) counts toward the layer. Gold, the mart drafted from the same metadata, may read that table. A source that changes shape shows up as a contract diff. See [The agent reads the contract](#).

You are done when the contract is versioned and the first consumer reads the table. **The way this fails is** a contract nobody diffs, so a dropped source column reaches gold as nulls before anyone reads what changed.

Step 7: Integrate by exception, then point an agent at the contract

Integration comes last, and only where a business requirement calls for it.

When finance asks for one customer list, an exact-match rule on tax number, approved by a person, gives the order system's 00001042 and the sales system's A-937 the shared key `customer_account || tax_number || 48123456789` on Customer Account's key register. A golden record beside it carries a survivorship rule per attribute, saying which source's value wins. That's level 4, Integrated, where it's earned. See [Integration by exception](#).

Then ask an agent how many customer accounts in Australia had an order move to Cancelled last quarter, by sales owner. Give it eight instruction lines, rendered from the contract, telling it to read the grain sentence first and decline each clause with no mapping.

You are done when the answer cites the contract and declines the clause it can't answer, the sales owner. **The way this fails is** entity resolution by guesswork. A merge on a likely match is wrong in every join downstream, and nobody downstream can see which.

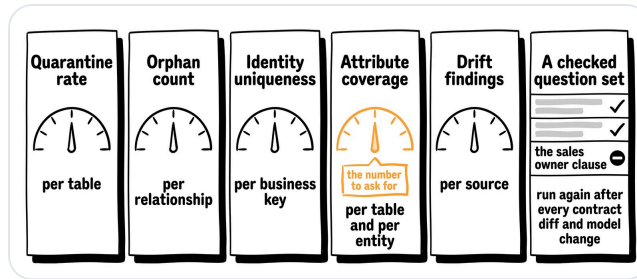
Why finish one business area before the next?

Every approval compounds. It's a fact for the next agent question and a demonstration for the next suggestion.

Two half-finished business areas give gold nothing it may read yet.

How will you know the layer works after go-live? It reports its own health without a model call, in quarantine rates, orphan counts, identity uniqueness, attribute coverage and drift.

One more measure belongs to the agent: a set of business questions with answers you've checked, including ones it should decline, run again after every contract diff and model change. Run your questions as each table climbs. The level where they start answering correctly is the level that business area needs.



You don't need a new modeling method. You need a meaning layer, the meaning your methods always carried, written where machines can read it, with the people who always decided it still deciding.

Start this week with one business area and one workshop.

The structural answer	A vault team built hubs, links, satellites, the business vault and the marts in order, so a business key decided wrongly was already hashed into everything loaded after it.
What the agent reads	Nothing new: the seven steps produce the data contract in nearly the order the agent reads it.
Yours to decide	Which business area goes first, and which checked questions an agent must answer, or decline, after every contract diff.

Silver layer glossary and reference tables

What does each AIMS term mean?

Rather than restructuring source data to fit a modeling methodology, **AIMS (AI-ready, Meaning-first Silver)** starts with business meaning. Written so a machine can read it, your business model is an **ontology**. The business model plus the mappings that tie silver columns to it is the **meaning layer**.

A table can be tested yet silent about what its rows mean. Five **meaning levels** say how much it tells its reader: 0 Typed, 1 Identified, 2 Related, 3 Described and 4 Integrated.

Joins, codes and governance rules depend on columns that identify, relate, hold a code or hold sensitive data. So these **critical columns** must be mapped. The unit of work a set of tables is organized and generated in, a **project**, may add classes. Gold and agents may read a table once it **publishes**: level 2 complete, level 3 for its critical columns, its checklist passed. Elsewhere, **attribute coverage** measures mapping, and low coverage only warns.

Three systems can mean three things by customer. Each business thing your data is about, a **business entity**, gets a definition. How do you tell one instance from another? Five shapes answer, the **entity classes**: independent, dependent, event, reference and association.

An unnamed line leaves every report guessing. So a **business relationship** carries a role, a cardinality, an optionality and a policy for an unresolved reference. Count versions as customers and the answer's wrong, so a table carrying an entity says what one row is in its **grain sentence**.

Every restructuring is a decision somebody must keep defending, so the silver layer's default is one table per source table, at the same grain, typed and renamed, a **source-aligned table**. Its default materialization is a view over the layer below silver that keeps every version a source delivered, **persistent staging**, in bronze. That's your audit trail, and it exists before silver starts.

Each row says which source record it is, its **source primary key**, and which version or event, its **change sequence**. History is chosen on purpose, as a **history policy**: append, current, versioned ([Type 2](#)) or snapshot. A silently dropped row is a lie in the row count, so each quality rule's **failure action** warns, quarantines (sets the row aside, counted) or blocks.

Customer 123 in two systems can be two customers. A **business key** is the set of values the business uses to identify one thing, in a fixed order, such as a company code and an account number. It's unique only inside the system that issued it, its **record source**. Written out in a **business key column**, it starts with the **record source code**: `aw1t||AU||00001042`. Every reference needs one place to find its identity, the entity's **key register** of distinct business keys.

Which column holds the customer's name? One meaning, defined once on its entity, is a **business attribute**. An **attribute mapping** ties a column to it. Every column is mapped, source-only on purpose, or unmapped. Codes differ by system, so a **code list** holds the business values (reference data), and an approved **code mapping** (a value mapping) says, per record source, which source code means which.

An agent must know what one row is before it counts. A published table's **data contract** projects what the catalog holds, down to its **known limitations**: unmapped columns and unmapped codes.

Integrating every source up front is costly. So AIMS integrates only when a **business requirement** calls for it. That's **integration by exception**. Two record sources never join until **matching** gives their keys one **shared key**. A person approves the **match rules** and types a **manual match** where no rule can decide.

One combined record per business thing is a **golden record**. When sources disagree, a **survivorship rule** says which value wins.

Models are unreliable at what a key identifies. So above the **approval boundary**, a person approves every decision that changes meaning or values, however confident the model. Below the boundary, how much a person reviews is the **AI recommendation workflow**. Accepting is a decision, not authorship, so a value the AI proposed keeps the record of what proposed it, its **provenance**, until a person changes the value.

Other methods' names

The AIMS name	Hook	Data Vault	dbt
business entity	business concept	core business concept (a hub)	entity
business key	business key	business key	natural key
record source code	key set	business key collision code	source
business key column	hook column	business key column, hash key	(none)
source primary key	(none)	(none)	unique key
change sequence	(none)	load date	updated_at in snapshots
business attribute	(none)	satellite attribute	column
attribute mapping	(none)	(none)	(none)
code list, code mapping	(none)	reference table	seed
business relationship	compound hook	link	relationship test
source-aligned table	frame (formerly bag)	a raw satellite, loosely	staging model
key register	an implied hub	hub	(none)
shared key, matching	(none)	same-as link	(none)
golden record	(none)	business hub and business satellite	intermediate model
history policy	(none)	insert-only, Type 1, Type 2	incremental, snapshot
failure action	(none)	(none)	warn, error

Relationship kinds

Kind	Carried as
Reference	many-to-one, with a role
Containment	the parent inside the child's key
Hierarchy	a pointer to its own kind
Association	its own entity, a role per side
Classification	a code, beside any mapped value
Equivalence	shared keys, never a source column
Specialization	a kind of another entity

Kinds name links, not entity classes. Each relationship states its cardinality (1:1, 1:N, M:N or N:1) and optionality. A row with an orphaned reference is allowed and counted, warned and flagged (the default), or quarantined.

Columns and source changes

Order	Columns
1	business key columns: own identity, then each referenced role
2	source columns in order, typed and renamed, source-only ones kept
3	mapped values, beside the column each translates
4	quality flags for the warn rules a row broke
5	system columns: load timestamp, record source, hash difference, history dates and flags

Source change	Result
Deleted row	ignored, tombstoned, version closed, or hard deleted (append tables reloadable from persistent staging only)
Added column	unmapped, a finding, not a failure
Type change	quarantined at the cast
Removed column	kept, nullable and deprecated, until a person retires it

Business keys

Field	Holds
Key parts	in order, each a column or a fixed value, with its padding
Normalization	trim, upper-case and collapse whitespace
Separator	from Settings, one between parts

Entity	Record source code	Parts
Customer Account	awlt	company code, customer number padded to eight digits
Sales Order	awlt	company code, order number
Sales Order Line	awlt	company code and order number from its order, line number
Product	awlt	product number
Customer Account	wb	account identifier, upper-cased

Types and mappings

Semantic types	Type contract
identifier, code, name, free text, flag, geography, reference to another entity	an identifier keeps leading zeros, never summed
quantity, monetary amount, percentage, ratio, count	a monetary amount needs scale and currency
date, instant, duration	a date isn't an instant

Sales order column	Business attribute	Type and role
company_code	Company Code	identifier, part of the business key
order_number	Order Number	identifier
customer_number	Sold-to Customer	relationship
bill_to_customer_number	Bill-to Customer	relationship
order_date	Order Date	date, temporal
amount	Order Amount	monetary amount, measure, currency from the currency column
currency	Order Currency	code, code list Currency
status	Order Status Code	code, code list Order Status

The ninth, `changed_at`, is the source's `ChangedAt`: the source change time and change sequence.

Publishing and matching

Area	Acceptance checklist
Identity	grain confirmed, business key approved, uniqueness tested
Values	type contracts and units set
History	policy chosen, replay repeatable, time columns apart
Rules	orphans counted, failure actions set, row counts reconciled
Governance	lineage, ownership and classification on record

Matching	Holds
Match rules	exact matches after normalization, in order, each approved by a person
Manual matches	a person's: business key, shared key, effective dates
Shared keys	on the key register, computed each run

Templates and run metrics

Kind	AIMS templates write
Structure	create statement, current view, key register view
Loads	the history policy's load, the clean and quarantine pair
Expressions	business key columns, mapped values from code mappings
Tests	a data test per block rule

Grain	Every run measures
Table	quarantine rate, warn counts per rule, row count against source, freshness against the entity's target
Business key	identity uniqueness
Relationship	orphan count
Entity	matching coverage and attribute coverage
Source	drift: columns added, removed or retyped since the last contract

Agent instructions

Instruction

Read the grain sentence first, name the counted thing, and on a version grain count keys, not rows.

Answer “when” from the time the question means (source change, effective or load), using the version valid then.

Join only along the contract’s relationships, by role, and name any join that changes the grain.

Add only what the semantic type allows, in one unit and currency, and recompute ratios from their parts.

Filter and group on mapped values, never source codes.

Take every metric and segment from the consumption layer’s definitions, inventing none.

Decline or qualify each clause with no mapping, naming the gap.

Cite the contract version, the tables read and the load behind them.
