PROFORMA GLOBAL RESEARCH

Semantic Layers in Enterprise Agent Systems

Matt Rollings, Founder and Principal, Proforma Global

Whitepaper · 2026-05-18

Executive Summary

Few terms in enterprise data are used as loosely as semantic layer, and for financial systems the common understanding of it is not just loose but misunderstood in ways that decide whether an agent works. In business intelligence the term has settled into meaning a governed catalog of metrics and dimensions over the warehouse, a single place where revenue and margin are each defined once. Useful as that catalog is, it is far too narrow for the systems an agent has to reason over in finance, and the distance between the catalog a team builds and the grounding an agent actually needs is where most deployments quietly go wrong. This paper uses the term in its broader and more demanding sense.

A semantic layer is anything that bridges the gap between what a model can natively interpret and what the domain requires it to understand. For a general LLM working in natural language, that bridge is linguistic. For a model trained on a specific kind of data, whether XML, code, or a structured schema, it takes whatever form connects the model's native vocabulary to what the domain means.

A semantic layer is necessary only where the model's native reading diverges from what the domain or the task requires in a way that changes the answer. Concepts that already line up with the model's training need no override. Concepts whose meaning diverges from that native reading have to be represented explicitly.

That representation is rarely a single object. Most domain concepts break into smaller pieces joined by relationships, and the reasoning unit that consumes them decides which pieces and which relationships have to be present together. A semantic layer therefore has to be built around the reasoning units it serves, with mechanisms that hold each constellation of related concepts together instead of letting it fragment.

This paper sets out those principles, catalogs the kinds of semantic layer that show up across enterprise systems, and describes the structural properties (hierarchy, inheritability, cohesion) that govern how they are organized and composed.

1. What a Semantic Layer Is

A semantic layer is anything that bridges the gap between what a model natively understands and what the domain, or the task, requires it to understand.

The definition is deliberately broad, because what the layer looks like depends on the model. A general LLM trained on natural language reads English and the other languages it was trained on, so its semantic layers are linguistic, translating domain concepts into language the model already interprets correctly. A model fine-tuned on XML schemas works in XML, so its layers are schema mappings, type hierarchies, and structural encodings. A model trained on code works in syntax trees and tokens, so its layers look like type systems, API documentation, and call-graph annotations. The intent is the same every time. Only the gap differs, and with it the layer.

This paper takes the LLM-on-natural-language case, because that is the dominant setting for enterprise agents today. The principles carry to other model classes. The specific layers do not. As organizations start to use internally fine-tuned models for particular tasks, these premises will move with them.

2. When a Semantic Layer Is Necessary

An LLM's training gives it a workable default reading of most general business concepts. Build semantic layers only where the domain's reading diverges from that default in a way that changes the answer.

Consider the concept "invoice." A user asks an agent, "Show me invoices that are overdue." The model already knows what an invoice is and what overdue means in business, and can reason about both at a conversational level. No semantic layer is required.

Now put the same concept in a different sentence. "Show me invoices that are overdue in NetSuite, account 4000–Revenue, customer segment Tier 2, where the dunning workflow has reached stage 3 and the credit hold flag has not been cleared." Invoice is no longer a generic document. It is a row in an ERP table with a unique identifier, system-specific fields, rules for how dunning state moves, and a relationship to customer segmentation that NetSuite encodes a particular way. The model's native reading gives an answer that is generically reasonable and operationally wrong, and the agent has no way to tell which sense the user meant.

The example points to a test for when a semantic layer is required.

> **The native test.** If the model would reach a substantively correct answer with no domain context beyond the user's question, no semantic layer is needed for that concept. If its native reading would be confidently wrong or incomplete, a semantic layer is required.

The native test is necessary but not sufficient. Depending on complexity, the task, and the process, an agent can reason wrongly in silence and still return a correct-looking output while getting the underlying task wrong. Success has to be judged against the reasoning and the output together, not the output alone. Judging an agent by whether its output looks right is one of the largest reasons agent programs stall.

A few refinements matter on top of the test.

The gap has to actually change the answer. Sometimes the model's reading differs from the domain's but reaches the same conclusion. A layer there adds cost and no capability. The layers worth building are the ones that close a gap that would otherwise produce a wrong answer.

The gap is per concept and per question, not per domain. Within a single ERP-grounded agent, some concepts are fine at the native level (month-end, approval, vendor) while others need a layer (invoice, GL account, consolidation entity, dunning state). Within one concept, different questions need different depth. "Total of Q3 invoices" needs less depth than "explain why invoice INV-2024–9982 was held three days longer than its segment average." Build where the gaps are, at the depth each question needs.

Native interpretation has limits even where it works. An agent that leans on its native interpretation is right when the user speaks generically and wrong when the user speaks in the system's terms, with no way to tell the two apart. For that reason, even a concept the model could handle on its own often gets a thin disambiguating layer anyway.

Building a semantic layer comes down to finding where the gaps actually are, working out how they affect the answers, and closing them at the lowest cost.

3. The Atomic Composition of Semantics

A semantic concept is rarely a single object. Most domain concepts break into smaller pieces joined by relationships, and the reasoning unit that consumes them decides which pieces and relationships have to be present together. The unit of design is the constellation a reasoning unit needs, not the concept on its own. Define a concept in isolation and the reasoning unit cannot use the definition.

Take the invoice again. A reasoning unit like "validate this invoice during monthly close" does not treat invoice as a primitive. It works on the invoice as part of a much larger process, and in isolation an invoice can be meaningless. In the worst case it is not an invoice at all, but a piece of fraud the agent has to catch. A semantic layer supplies context and definition, and it also confers validity on each object as part of the whole it sits inside. Every piece in a reasoning unit's constellation has to line up, or the process breaks.

Validating an invoice during monthly close means having every piece of its constellation present, with the relationships between them intact:

- The invoice has an amount
- The invoice references a purchase order
- The purchase order is tied to an active contract
- The contract has a remaining unspent balance
- The contract is valid for the current fiscal year
- The PO has an authorized spend ceiling
- The invoice has a payables account assignment
- The vendor on the invoice has approved status

A different reasoning unit pulls a different constellation around the same invoice. "Forecast cash outflow" needs invoice plus payment terms plus aging bucket plus bank routing plus scheduled payment date. The concept overlaps. The constellation does not.

The same concepts show up in many reasoning units inside different constellations, so the layer has to be built around the reasoning units it serves rather than the concepts it catalogs. Building it that way takes three commitments.

1. **Atomic definition of every concept in the constellation.** Invoice, PO, contract, balance, vendor, fiscal period each needs its own definition, with the relationships it carries modeled explicitly.
2. **Relationships as first-class objects.** "Invoice references PO" cannot be left as an implicit join. It has to be an object with its own definition that says what the reference means, what makes it valid, and what makes it broken. Leave relationships implicit and the agent infers them from the raw data and gets them wrong.
3. **Composability into the constellations reasoning units actually need.** The layer has to be queryable in those groupings. A reasoning unit that needs invoice plus PO plus contract plus balance should be able to ask for the whole constellation as one composed grounding, not as four separate queries it then has to stitch together.

---

4. The Layer Classes

The kinds of semantic layer that show up consistently in enterprise systems each add a different kind of meaning. The catalog below covers the world-model layers, the ones that describe the domain the agent reasons about. The reasoning-context layers (intent, constraints, examples, output schema, state, domain knowledge) frame the act of reasoning itself and sit on top of these. They are the subject of a later paper.

Vocabulary. Maps user-facing terms to entities in the underlying data model, the synonyms, aliases, abbreviations, and code-to-name mappings. Matching terms is harder than it looks, because how much tolerance to allow depends on the kind of term, and the layer has to know when to refuse rather than guess. Get this wrong and the agent resolves a term to a nearby meaning the user never intended, with full confidence.

Structural. Maps entities to the dimensional context they sit in. It is harder to design than it sounds, because dimension membership in practice is conditional rather than absolute. When it is wrong, the agent applies a dimension to an entity it does not belong to and returns a number that is technically computed and operationally meaningless.

Attribution. Maps each value to its source, its derivation, and who owns it. The depth that matters is the kind of derivation, not just the fact of one. An attribution layer that records only where a value came from cannot reason about how it was produced. An agent then quotes a derived value as if it were original.

Temporal. Resolves time semantics, including period, scenario (Actual, Plan, Forecast), version, and fiscal calendar. Time earns its own layer because mixing up two time slices is both serious and invisible. An agent that returns a Plan value when the user wanted Actual gives a number that is right in every respect but the one that matters. Versioning and scenario lifecycle bring their own design questions, which decide whether the temporal layer stays usable as the business piles up plan iterations.

Topology. Captures the dependency graph between entities. It is what lets the agent answer structural questions, like what feeds this entity, what depends on it, and what would move if a driver changed. A bare nodes-and-edges representation is not enough, because real relationships carry more structure than that. Without it, the agent reasons over values with no idea what produced them.

Calculation. Captures the logic that produces values. It is distinct from topology. Topology is the dependency graph, Revenue depends on ASP x Volume. Calculation is the formula itself, with all of its conditional handling. An agent reasoning about derived values needs both. Topology alone says what depends on what, not how the value is produced.

---

5. Hierarchy, Inheritability, and Cohesion

Individual layers are not enough. Whether the layer works depends on how each one is organized inside, and on how the clusters of concepts hold together across layers.

Hierarchy within a layer. A flat list is unmaintainable at scale. A hierarchy lets a rule set at a parent apply to everything beneath it without restating. Define "Revenue" at the parent of your vocabulary layer and "Product Revenue" and "Service Revenue" inherit its base properties. Set source-of-truth at a parent account class and the descendants inherit attribution by default.

Real enterprise data models are not strict trees. One member commonly rolls up into several parents at once, across functional, geographic, and regulatory rollups. The layer has to handle these n-ary parent-child relationships and the diamond hierarchies they create, including the traversals that go with them, lowest common ancestor for grouping and ancestor-walking for inheritance, where a leaf branches on the way to the root.

Inheritability across layers. When the agent resolves a term, the resolution cascades. Vocabulary produces an entity. The entity carries structural projections. Structural carries default temporal and attribution context. Temporal and attribution carry their own scopes. At

each level the layer either gives a definite answer or inherits from a declared default, so the agent never has to guess what was left unspecified.

When inheritance breaks, the layer either flags the gap or falls back to a default the model has to interpret. The fallback is the silent breakage the whole discipline exists to prevent.

Cohesion across constellations. Keeping the concepts in one reasoning-unit constellation coherent with each other is the harder problem. The invoice, PO, contract, and balance constellation cannot be assembled by querying four independent definitions and joining them inside the model's context. Querying them separately produces fragmented context the model has to stitch, and the stitching is where it goes wrong.

Cohesion takes several mechanisms working together.

1. **Bounded contexts.** A semantic layer applies within a defined scope. Inside "accounts payable monthly close," invoice has one set of relationships. Inside "treasury cash management," it has another. Within a bounded context, cohesion is reachable. Across all of business, it is not.
2. **Shared identity across concepts.** The concepts in a constellation have to share an identity space inside their bounded context. Invoice ID, PO ID, contract ID, and vendor ID are the threads that stitch the constellation together. Identity coherence is a foundational requirement, not a detail.
3. **Closure checks.** For any reasoning unit, the layer should check that every concept in the declared constellation is defined and every relationship resolves, before reasoning starts. Gaps get flagged rather than quietly filled with a default the model has to interpret.
4. **Hierarchical containment where it exists naturally.** Some constellations contain each other naturally. A contract contains POs, a PO contains invoices, a close period contains many invoices. Where the containment is real, modeling it makes cohesion easier. Where it is forced onto concepts that do not actually nest, it makes the structure brittle.

Cohesion breaks by silent fragmentation. The layer hands back concept definitions one at a time, the agent stitches them together as it reasons, the stitching is wrong, and the answer still looks plausible. Every mechanism above exists to stop that.

The complexity of the constellations a working semantic layer produces is also why the orchestration of an enterprise agent workflow has to be deterministic. The model cannot reliably work out which constellation a reasoning unit needs, query the right parts of the layer, and assemble the result correctly as the workflow grows. The workflow declares the reasoning

unit. The orchestration knows which queries produce the constellation it needs. Dynamic context assembly composes the result. The full case against LLM-driven orchestration is in "Where Agent Orchestration Breaks."

---

6. What Organizations Need to Get Right

Building a semantic layer is a design exercise, not a documentation exercise. A handful of decisions determine whether the result supports an agent system or merely catalogs the business, and in our experience the same few show up at the root of every program that fails to land.

The gap audit. Build layers for concepts the model already handles natively and you pay for capability you did not need. Miss the concepts where the gap actually bites and you ship agents that are confidently wrong. The audit that finds which concepts need an explicit override, measured against the agent's real workload, is the first design step and the one most often skipped.

The reasoning unit catalog. A layer designed without reference to the reasoning units it will serve catalogs the business in full and serves no actual workload. The reasoning units the agent performs (validate invoice, forecast cash, reconcile account, plan headcount, explain variance) drive the constellations the layer has to produce. The catalog of reasoning units comes before the catalog of concepts. We have not seen this hold up in production any other way.

The bounded contexts. Invoice defined once across every context produces a definition that satisfies no one. Naming the contexts is the first cohesion decision, and deferring it is the most common mistake we see before production.

Layer composition per context. How the layers combine depends on the kinds of ambiguity each context carries. A consumer-products business with a simple chart of accounts usually needs fewer world-model layers than a regulated financial-services firm. Compose the layers uniformly across contexts and you overspend in some and leave gaps in others, and both only surface once real workloads hit the layer.

Relationship modeling. Implicit relationships fragment under load. Retrofitting explicit modeling means rewriting both the concept definitions and the agents that consume them, roughly the scope of the original build. The decision to model relationships as first-class objects has to be made when the layer is designed.

Layer governance. A layer without a named owner drifts. The layer that worked at deployment stops working as the business changes, and the drift stays silent until the agent's answers start to slip. Governance is what keeps each layer aligned with the business over the life of the agent system.

Composition engineering. A layer that hands the agent fragmented queries to stitch at reasoning time has the cohesion problem built in. How composition works is a foundational decision that constrains every layer built after it.

No two organizations resolve these the same way. The answers depend on the complexity of the domain, the existing data architecture, and the workloads the agents will serve. Working through them before construction costs far less than discovering them in production.

---

7. Boundary

This paper has set out the foundational definitions and design principles of semantic layers in enterprise agent systems. It has not said which layers a given organization should build, how to store or transmit them, how to model relationships, or how to engineer composition. Those decisions depend on the domain.

The rest of the series goes deeper. It treats the individual world-model layer classes that warrant their own treatment, the reasoning-context layers that frame the act of reasoning (intent, constraints, examples, output schema, state, domain knowledge), the patterns for resolving ambiguity when a layer cannot give a definite answer, the versioning and evolution that keep the layer aligned with the business, and the multi-workload patterns for sharing one layer across deployments without losing isolation.

A semantic layer bridges the gap between what the model natively understands and what the domain requires it to understand. Build the layers where the gaps are, build them around the reasoning units they serve, and engineer cohesion across the constellations those units consume.

DOWNLOAD PDF

Work with Proforma Global

The thinking in this paper is public; the methods that turn it into a working system are not. If it fits a problem your team is working on, that is what we bring to an engagement. Start a conversation: **info@proforma.global**.