PROFORMA GLOBAL RESEARCH

Semantic Layers in Enterprise Agent Systems

Matt Rollings, Founder and Principal, Proforma Global

Whitepaper · 2026-05-18

Executive Summary

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 generalized LLM consuming natural language, semantic layers are linguistic. For a model trained on a specific dataset (XML, code, structured schema), semantic layers take whatever form bridges the model's native vocabulary to domain meaning.

A semantic layer is necessary only where the model's native interpretation diverges from what the domain or task requires in ways that affect correctness. Generic concepts that align with the model's training corpus need no override; domain-specific concepts that diverge from native interpretation require explicit semantic representation.

The semantic representation itself is rarely a single object. Most domain concepts decompose into atomic components linked by intrinsic relationships, and the reasoning unit that consumes them determines which components and relationships must be present together. The semantic layer therefore has to be designed reasoning-unit-shaped, with cohesion mechanisms that keep constellations of related concepts coherent rather than fragmented.

This paper establishes these foundational principles, catalogs the categories of semantic layer that typically appear in enterprise systems, and describes the structural properties (hierarchy, inheritability, cohesion) that govern how semantic layers 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 task) requires it to understand.

The definition is intentionally broad because the form depends on the model. A generalized LLM trained on natural language has a native interpretation framework that operates on English and the other languages it was trained on; semantic layers for an LLM are linguistic, translating domain concepts into language patterns the model interprets natively. A model fine-tuned on XML schemas operates on XML structures natively; semantic layers for that model take the form of schema mappings, type hierarchies, and structural meaning encodings. A model trained on

code operates on syntax trees and tokens natively; semantic layers look like type systems, API documentation, and call-graph annotations. The intent is the same in every case. The form varies because the gap varies.

This paper addresses the LLM-on-natural-language case because it is the dominant context for enterprise agents today. The principles generalize to other model classes; the specific layer forms do not. As organizations begin to leverage internally fine-tuned models for specific tasks, these premises will shift in parallel.

---

2. When a Semantic Layer Is Necessary

The LLM's training corpus provides a native semantic baseline for most general business concepts. Build semantic layers only where the domain's interpretation diverges from that baseline in ways that affect correctness.

Consider the concept "invoice." A user asks an agent, "Show me invoices that are overdue." The LLM already knows what an invoice is from its training corpus. It knows what overdue means in business context. It can reason about the concept at a conversational level. No semantic layer is required.

Now consider the same concept in a different context: "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 hasn't been cleared." Now invoice is no longer a generic document. It is a row in an ERP table with a unique identifier, with system-specific fields, with rules about how dunning state transitions, with a relationship to customer segmentation that NetSuite encodes a particular way. The LLM's native interpretation produces an answer that is generically reasonable and operationally wrong, and the agent has no way to detect which mode the user is operating in.

This gives us the test for when a semantic layer is required.

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

The native test is necessary but not sufficient. Depending on complexity, task, and business process, agents will silently reason incorrectly and produce correct-looking outputs while essentially failing the underlying task. Success and validation have to be measured against the reasoning logic and the outputs together, not the outputs alone. Treating outputs as the sole validation signal is one of the largest causes of agentic deployment failures in production.

A few refinements matter on top of the test itself:

The gap has to actually affect the answer. Sometimes the LLM's interpretation differs from the domain's, but the difference doesn't change the conclusion the agent reaches. A semantic layer for those cases adds cost without adding capability. The layers that matter are the ones that close gaps which would otherwise produce wrong answers.

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

Native interpretation has limits even where it works. An agent that relies on native interpretation produces correct answers when the user is speaking generically and wrong answers when the user is speaking in the system's terms, with no way to detect which mode the user is in. This is why even concepts that could be handled natively often get a thin disambiguating layer anyway.

The discipline of building a semantic stack is identifying where the gaps actually exist, how they affect the answers the agent produces, and how to close them at the lowest possible cost.

---

3. The Atomic Composition of Semantics

A semantic concept is rarely a single object. Most domain concepts decompose into atomic components linked by intrinsic relationships, and the reasoning unit that consumes them determines which components and relationships must be present together. The unit of semantic-layer design is the constellation the reasoning unit requires. Defining a concept in isolation produces a definition the reasoning unit cannot use.

Take the same invoice concept from above. A reasoning unit like "validate this invoice during monthly close" does not operate on invoice as a primitive. It operates on the invoice as part of a much larger process; in isolation, an invoice may be meaningless. In the worst case, an invoice may not actually be an invoice. It may be an element of fraud the agent has to detect. Semantic layers provide context and definition. They also provide validity to objects as part of the greater whole each one sits inside. Each atomic component in a reasoning unit's constellation has to be semantically aligned, or critical processes break down.

Validating an invoice during monthly close means having every piece of its constellation present, with its relationships 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 + payment terms + aging bucket + bank routing + scheduled payment date. The concept overlaps; the constellation does not.

This changes how the semantic layer is designed. The same concepts appear in many reasoning units with different surrounding constellations, and the substrate has to be designed around the reasoning units it will serve rather than the concepts it catalogs. That requires three things:

1. **Atomic definition of every concept in the constellation.** Invoice, PO, contract, balance, vendor, fiscal period: each needs its own semantic definition with the relationships it carries explicitly modeled.
2. **Relationships as first-class semantic objects.** "Invoice references PO" cannot be an implicit join. It has to be a semantic object with its own definition: what does the reference mean, what makes it valid, what makes it broken. When relationships are implicit, the agent infers them from data shape and gets them wrong.

3. **Composability into reasoning-unit-shaped constellations.** The semantic layer has to be queryable in the shapes that reasoning units actually need. A reasoning unit that needs invoice + PO + contract + balance should be able to request the constellation as a single composed grounding, not as four separate semantic queries the agent has to stitch together.

---

4. The Layer Classes

The categories of semantic layer that appear consistently in enterprise systems describe what kind of meaning each layer adds. The catalog below covers the world-model side of the semantic stack: the layers that describe the domain the agent reasons against. The reasoning-context side (intent, constraints, examples, output schema, state, domain knowledge) describes what frames the reasoning act itself and composes on top of the world-model foundation; it is the subject of a later paper.

Vocabulary. Maps user-facing terms to entities in the underlying data model: synonyms, aliases, abbreviations, code-to-name. The matching discipline is harder than it looks because tolerance has to vary by the kind of term, and the layer has to know when to refuse rather than guess. The failure mode is the agent that confidently resolves to an adjacent meaning the user did not intend.

Structural. Maps entities to the dimensional context they sit in. The layer is harder to design than its description suggests because dimension membership in practice is conditional rather than absolute. The failure mode is the agent that applies a dimension to an entity where the dimension does not actually apply, producing a result that is technically computed and operationally meaningless.

Attribution. Maps each value to its source, its derivation, and its authoritative owner. The depth that matters is the kind of derivation, not the fact of it; an attribution layer that records only where a value came from cannot support reasoning about how it was produced. The failure mode is the agent that quotes a derived value as if it were original.

Temporal. Resolves time semantics: period, scenario (Actual, Plan, Forecast), version, fiscal calendar. Time deserves its own layer because the failure mode of conflating time slices is catastrophic and undetectable: an agent that returns a Plan value when the user wanted an

Actual produces a number that looks right in every respect except the one that matters. Versioning and scenario lifecycle have their own design considerations that determine whether the temporal layer remains usable as the business accumulates plan iterations.

Topology. Captures the dependency graph between entities. The layer is what lets the agent answer structural questions: what feeds this entity, what depends on it, what would change if a driver moved. A minimal nodes-and-edges representation is insufficient because real relationships carry more structure than that, and the failure mode is the agent that reasons over values without understanding what produced them.

Calculation. Captures the computational logic that produces values. Distinct from topology: topology is the dependency graph (Revenue depends on ASP × Volume); calculation is the formula itself with its full conditional handling. An agent reasoning about derived values needs both; topology alone tells the agent what depends on what but not how the value is produced.

---

5. Hierarchy, Inheritability, and Cohesion

Layers individually are not enough. The way each layer is organized internally, and the way constellations of concepts are held together coherently across layers, is what determines whether the substrate actually works.

Hierarchy within a layer. A flat list is unmaintainable at scale. A hierarchically organized layer lets rules defined at parent levels apply to descendants without re-specification: define "Revenue" at the parent level in your vocabulary layer, and "Product Revenue" and "Service Revenue" inherit base properties; define source-of-truth at a parent account class, and descendants inherit attribution by default.

Real enterprise data models do not produce strict trees. A single member commonly participates in multiple parent rollups: functional, geographic, regulatory. The semantic layer has to support these n-ary parent-child relationships and the diamond hierarchies they produce, including traversal operations (lowest common ancestor for grouping, ancestor walking for inheritance) that handle paths where leaves branch 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. Each layer either produces a definite answer or inherits from a declared default at its own level. The agent never has to guess what was left unspecified.

When inheritability is broken, the substrate either reports the gap explicitly or falls back to a default the model has to interpret. The latter is the silent-failure pattern the discipline exists to prevent.

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

Cohesion requires several mechanisms working together:

1. **Bounded contexts.** A semantic layer applies within a defined scope. Within "accounts payable monthly close," invoice has one set of relationships; within "treasury cash management," it has another. Within a bounded context, cohesion is achievable. Across all of business, it is not.
2. **Shared identity across concepts.** The concepts in a constellation must share an identity space within their bounded context. Invoice ID, PO ID, contract ID, vendor ID: these 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 substrate should verify that every concept in the declared constellation is defined and every relationship is resolvable, before the reasoning starts. Gaps get flagged rather than silently filled with defaults the model has to interpret.
4. **Hierarchical containment where it exists naturally.** Some constellations have natural containment: 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 produces brittle structure.

The cohesion failure mode is silent fragmentation: the substrate returns concept definitions individually, the agent stitches them at reasoning time, the stitching is wrong, and the answer looks plausible. Every cohesion mechanism above exists to prevent that failure.

The complexity of the constellations a working semantic stack produces is also why the orchestration of an enterprise agent workflow has to be deterministic. The LLM cannot reliably discover which constellation a given reasoning unit needs, query the right substrate components, and assemble the result in the right shape as workflow complexity grows. The

workflow declares the reasoning unit; the orchestration knows which substrate queries produce the required constellation; dynamic context assembly composes the result. The full case against LLM-driven orchestration is made in "Where Agent Orchestration Breaks."

---

6. What Organizations Need to Get Right

The construction of a semantic stack is a design exercise, not a documentation exercise. Several decisions govern whether the resulting stack supports an agent system or merely catalogs the business; in our experience, the same handful of decisions show up at the source of every program that fails to land.

The gap audit. Organizations that build semantic layers for concepts the model already handles natively spend on capability they did not need. Organizations that miss the concepts where the gap is operationally consequential deploy agents that produce confidently wrong answers. The audit that surfaces which concepts require explicit override against the agent's actual workload is the first design step, and it is the one most often skipped.

The reasoning unit catalog. Substrates designed without reference to the reasoning units they will serve catalog the business comprehensively and serve no actual workload. The reasoning units the agent performs (validate invoice, forecast cash, reconcile account, plan headcount, explain variance) drive the constellations the substrate has to produce. The catalog of reasoning units precedes the catalog of concepts; we have not seen this discipline carry through to production without it.

The bounded contexts. Invoice defined once across every business context produces a definition that satisfies no one. Naming the contexts is the first cohesion decision; deferring it is the most common failure we see in pre-production substrates.

Layer composition per context. Layer composition depends on the kinds of ambiguity each context exhibits. A consumer product business with a simple chart of accounts typically needs fewer world-model layers than a regulated financial services firm. Uniform layer composition across contexts produces overspend in some and gaps in others, and both failure modes only surface once workloads are running against the substrate.

Relationship modeling. Implicit relationships fragment under load. Retrofitting explicit modeling requires rewriting both concept definitions and the consuming agents, and the work has roughly the scope of the original build. The decision to model relationships as first-class semantic objects has to be made at substrate design time.

Layer governance. A layer without a named owner drifts. The substrate that worked at deployment ceases to work as the business evolves, and the drift is silent until the agent's answers begin to slip. Governance is what keeps each layer aligned with the business over the lifespan of the agent system.

Composition engineering. A substrate that produces fragmented queries the agent stitches at reasoning time has the cohesion failure mode baked in. The composition mechanism is a foundational architectural decision that constrains every layer that comes after it.

No two organizations resolve these the same way. The answers depend on the domain's complexity, the existing data architecture, and the workloads the agents will serve. The cost of working through them before construction begins is significantly lower than the cost of discovering them in production.

---

7. Boundary

This paper has established the foundational definitions and design principles of semantic layers in enterprise agent systems. It has not specified which layers any particular organization should build, how layers should be stored or transmitted, how relationships should be modeled, or how composition should be engineered. Those decisions depend on the domain.

The remaining papers in this series go deeper on individual world-model layer classes where the design considerations warrant dedicated treatment, the reasoning-context layers that frame the reasoning act (intent, constraints, examples, output schema, state, domain knowledge), the resolution patterns for handling ambiguity when a layer cannot produce a definite answer, the versioning and evolution patterns that keep the stack aligned with the business, and the multi-workload patterns that govern how a stack is shared across agent deployments while preserving 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 exist, shape them around the reasoning units they serve, and engineer cohesion across the constellations the units consume.