ARTIFICIAL INTELLIGENCE • KNOWLEDGE GRAPHS • NEUROSYMBOLIC RETRIEVAL

Why AI Systems Fail: They Need More Than Vector Search

Why preserving structure and meaning can make AI retrieval more reliable, explainable, and useful

By Michael Iantosca
Senior Director of Knowledge Platforms and Engineering
Avalara

The simple idea

Vectors are very good at finding information that looks related. Knowledge graphs are good at preserving what things are, how they relate, and which rules apply. AI systems become stronger when they use both – but not just any vector!

The shortcut most AI systems take

A large number of AI search and question-answering systems are built the same way. Documents are broken into small pieces, those pieces are converted into numerical representations called embeddings, and the embeddings are stored in a vector index. When someone asks a question, the system searches for chunks that are mathematically similar to the question and passes them to a large language model.

This approach is useful. It is fast, flexible, and often super easy to build. That's the problem. It also has a critical weakness that is easy to overlook and dismiss: breaking information into chunks significantly weakens and removes much of the essential structure that gave the original content its meaning and context.

What chunking can throw away

Think of a technical document as a carefully organized filing cabinet. Its folders, labels, cross-references, and rules tell you not only where information is stored, but also how the pieces relate to one another. Now imagine cutting the manual into small strips and tossing them into a large box. The words are still there, and the strips can still be searched, but much of the structure and context that gave those words meaning has been lost. A typical naïve vector index works much the same way: it reaches into that box and retrieves the strips that appear most similar to the question, without truly understanding the relationships, rules, product versions, or other context that determine whether the information is actually the right answer.

Frankly, I'd trust a fortune cookie about the same.

Yet years after the weaknesses of naive vector indexes became obvious, the average developer still reaches for the same architecture. We built one ourselves in under a week nearly four years ago with a little Python and Pinecone. It was easy to build then, and it is easy to build now. That ease helped turn a quick prototype into an industry default, with developers worldwide repeating the same pattern largely because everyone else was doing it.

Worse, once the limitations become impossible to ignore, teams often spend years layering on increasingly elaborate compensating mechanisms – rerankers, weighting schemes, metadata filters, hard-coded rules, exception logic, and other patches – all to work around a weakness in the underlying architecture. Those mechanisms may improve individual cases, but they do not eliminate the root cause. They add complexity, cost, and maintenance burden while still producing something inherently less precise and less reliable than an architecture that represents the relationships correctly in the first place.

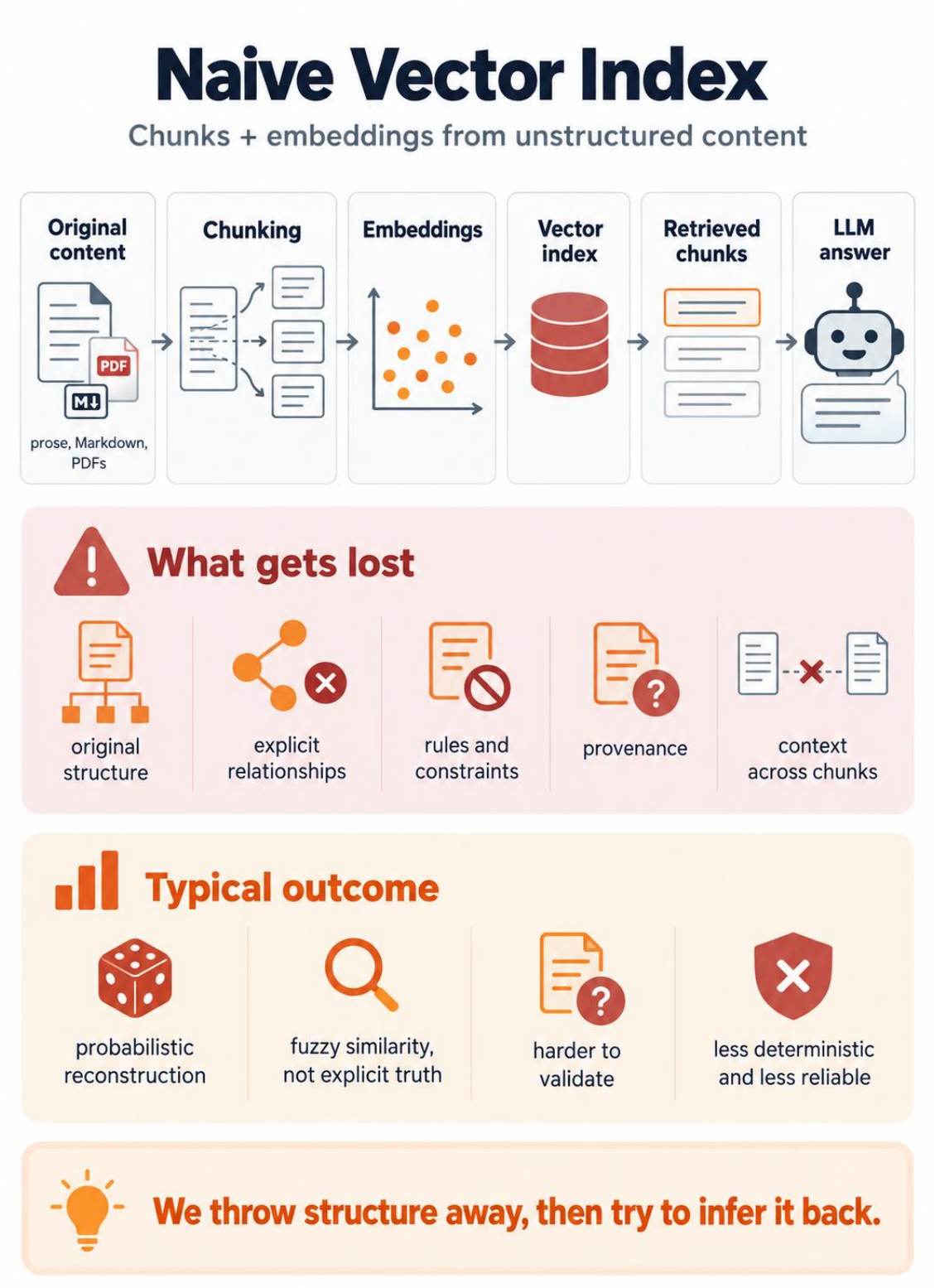
The troubling part is that we now understand exactly where the approach fails, yet most continue refining the workaround instead of fixing the foundation. If repeatedly applying the same flawed architecture, then building ever more elaborate machinery to compensate for it, is not a form of insanity, it is at least a remarkable failure to learn.

Some might call it insanity.

The same thing can happen when content is chunked for vector search. A chunk may no longer carry all of the surrounding context that explains where it came from, what other information it depends on, which rule governs it, or whether two different terms refer to the same thing.

Good engineering can preserve some of this information as metadata, and not every vector system is naïve. The important point is that these relationships are not inherent in the embedding itself. If they matter, they must be deliberately preserved or recreated elsewhere.

Figure 1. Conventional vector retrieval can weaken or lose explicit structure, relationships, rules, provenance, and cross-chunk context.



"We often throw structure away, then spend enormous effort trying to infer it back."

What a knowledge graph changes

A knowledge graph stores important relationships explicitly instead of leaving them for the AI to guess. Rather than keeping only a paragraph about a person, product, component, regulation, or procedure, the graph can record what that thing is and how it connects to other things.

For example, a graph can represent that Henry VIII is a king, that he ruled England, and that he succeeded Henry VII. Those relationships are stored directly. The AI does not have to reconstruct all of them from scattered sentences every time a question is asked.

Standards such as RDF provide a common way to represent these relationships. OWL can add formally defined meaning and support logical inference. SHACL can be used to validate whether graph data follows required shapes and constraints. These technologies do not make the information automatically correct, but they make its structure, rules, and assumptions far more explicit and inspectable.

Use both, not either-or

The strongest design is often not "vectors versus graphs." It is vectors plus graphs. The two technologies solve different problems. But not jus any vector – *a vector index derived from the knowledge graph.*

Vectors are excellent for broad semantic recall. They can find passages that use different wording but discuss the same general idea. A graph is better at explicit identity, relationships, rule-based reasoning, provenance, and controlled traversal from one concept to another.

In a neurosymbolic system, those capabilities work together. Embeddings help locate information that may be relevant. Graph queries and symbolic rules help determine how the relevant entities are connected and what follows from the encoded knowledge.

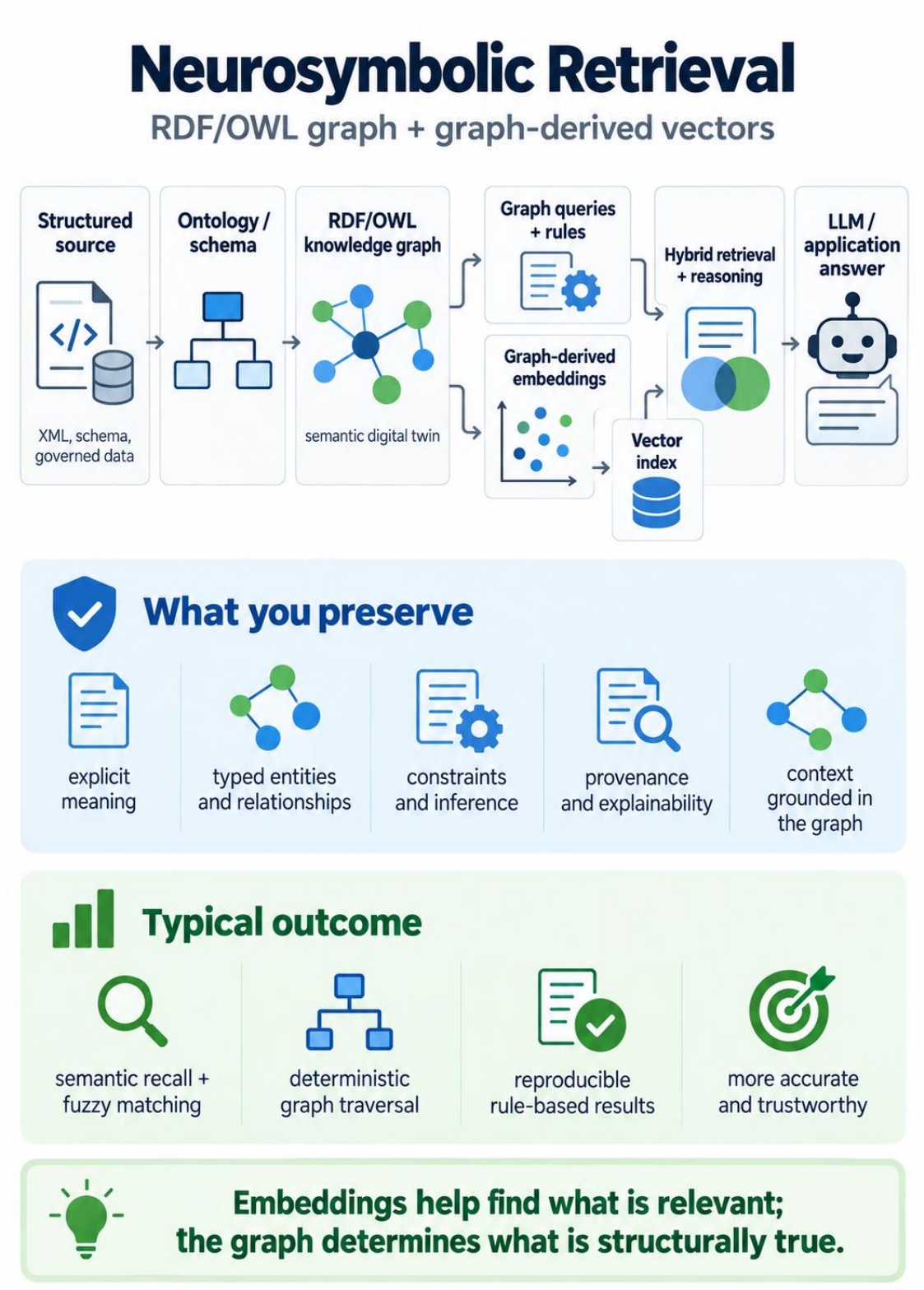
Why structured source content matters

This becomes especially important when the source information was already structured or semi-structured in the first place. XML combined with an XML Schema can define named domain elements, typed attributes, nesting, identifiers and references, controlled values, namespaces, and structural rules. A well-designed XML vocabulary can therefore encode far more than document formatting: it can provide a rich, machine-readable semantic representation of the content itself. A procedure step, warning, parameter, part number, software setting, version, applicability statement, or cross-reference can be represented as a distinct and addressable object before AI ever touches the content.

Markdown and AsciiDoc can also preserve useful structure through headings, blocks, anchors, attributes, front matter, document attributes, and other conventions, and they can be enriched with iiRDS or organization-specific metadata. They can coexist with XML in the same ontology and knowledge graph. The difference is one of precision and automation: XML/XSD usually exposes more formal, fine-grained structure natively, while Markdown and AsciiDoc typically depend more on conventions, extensions, or supplemental metadata to achieve the same degree of addressability.

A better architecture does not require a single authoring or source content format. Each governed source can be mapped into a common RDF/OWL model. XML can drive very granular entity and relationship creation, retrieval, and update directly from the schema and markup, while Markdown and AsciiDoc can contribute at the granularity their structure and metadata support. The graph becomes the common semantic layer without forcing every authoring team to use the same syntax.

Figure 2. A neurosymbolic retrieval model combines vector-based semantic recall with explicit graph-based meaning, rules, and reasoning.



The graph does not have to be built by hand

One common objection to a knowledge-graph approach is that building and maintaining the graph sounds like a large manual effort. It can be if the graph is treated as a separate database that people must update node by node. A digital-twin approach changes that model. Instead of maintaining the graph independently, the graph is generated from the governed source – or governed collection of sources – and kept synchronized with it.

Use the source as the blueprint

An organization does not need to author everything in one format. A semantic digital twin can be generated from a governed source estate that includes XML/XSD, Markdown, AsciiDoc, iiRDS metadata, or combinations of them. Format-aware parsers and mapping rules can translate those sources into the same identifiers, classes, properties, and relationships. The knowledge graph becomes the shared machine-readable model of the information rather than a second body of content that must be recreated and maintained by hand.

iiRDS provides another strong starting point because its metadata model is expressed in RDF and is independent of the authoring syntax used for the content. It can identify things such as the product or component that information applies to, the information type, and lifecycle-related context. When XML, Markdown, or AsciiDoc content is enriched with iiRDS metadata, a large part of the applicability and delivery semantics needed by the graph is already available in a standardized, graph-friendly form.

Why XML can automate much more of the graph

XML/XSD offers an especially powerful starting point because the markup itself can carry formal, machine-addressable structure. Named elements and attributes, type definitions, cardinality, identifiers, references, keys, namespaces, controlled values, and explicit nesting can be mapped deterministically to graph classes, nodes, properties, and edges. The pipeline does not have to infer from prose that a string is a warning, part number, parameter, prerequisite, or software setting when the source already identifies it as one.

That structure also enables a much more granular and addressable graph. Instead of creating one node for an entire page or topic, the pipeline can create separate entities for a procedure, each step, each warning, required tool, part, setting, value, unit, prerequisite, product/version applicability rule, and cross-reference – then connect them with explicit relationships. In a well-designed XML vocabulary, much of this graph construction can be schema-driven and automatic. LLM-based extraction becomes an optional enrichment or fallback mechanism rather than the primary way the system discovers what the content means.

Markdown, AsciiDoc, XML, and iiRDS can coexist

Markdown and AsciiDoc are not excluded from this architecture. Parsers can map headings, lists, code blocks, tables, anchors, front matter, document attributes, custom directives, and other structured constructs into the same graph model. iiRDS can add standardized product, component, information-type, applicability, and lifecycle metadata regardless of the source syntax. An XML topic and an AsciiDoc or Markdown topic can therefore refer to the same Product, Component, ProcedureType, or other shared graph entity instead of creating separate semantic islands.

The formats are not equally expressive by default. Markdown is intentionally lightweight, and AsciiDoc provides richer structural constructs and attributes, but XML with a governing schema normally provides the highest degree of formal typing, constraint, granularity, and direct addressability. A mixed-format pipeline can accept all of them while allowing XML content to produce a deeper, more precise graph wherever the source vocabulary supports it.

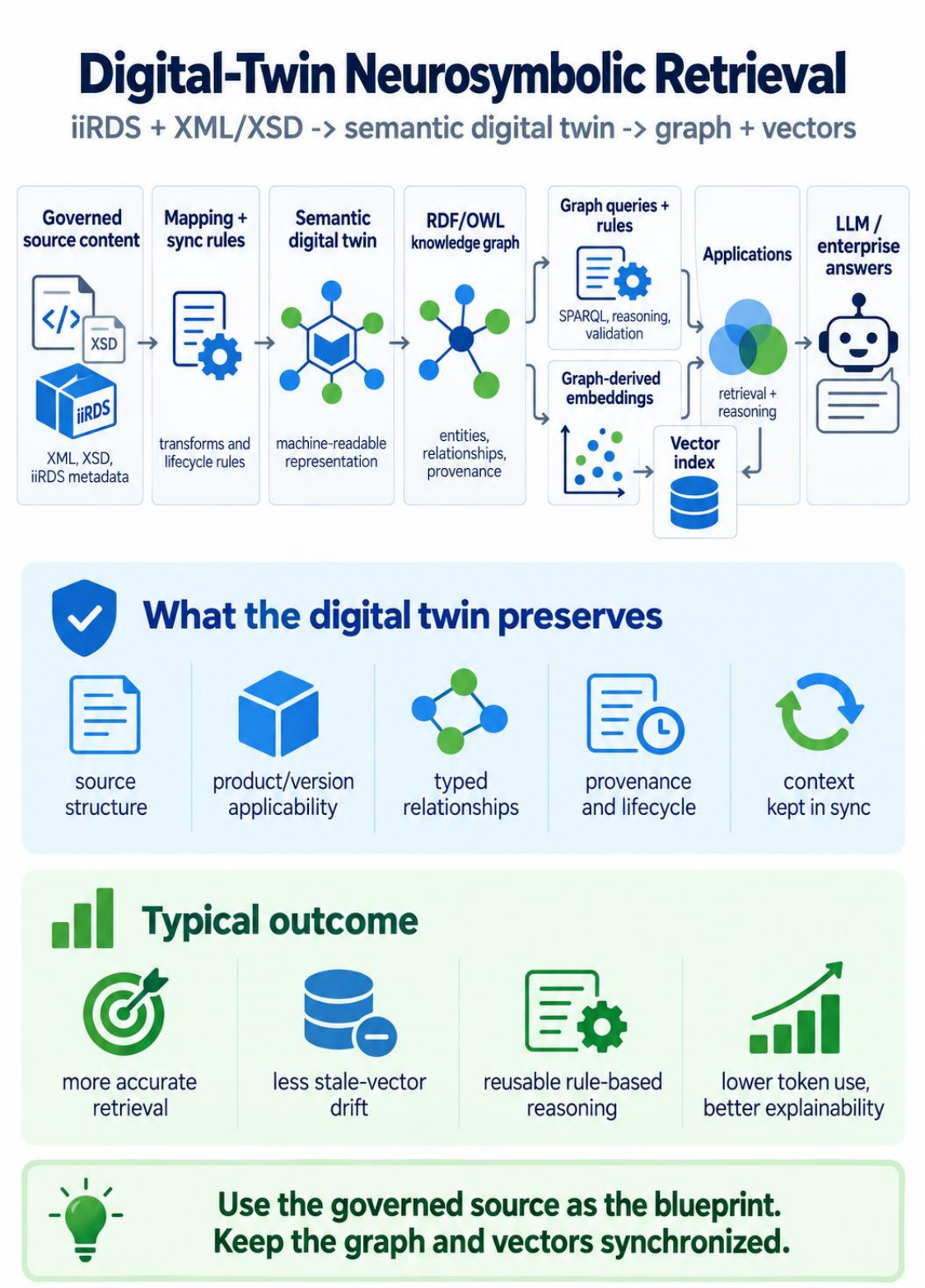
Keep the digital twin synchronized

The most important benefit is maintenance. When an authoritative source changes, a format-aware pipeline can identify the affected source objects, update the corresponding graph entities and relationships, mark older information as superseded or retired according to lifecycle rules, validate the updated graph, and regenerate only the vector representations that are affected. Stable graph identifiers can connect the same real-world entity across XML, Markdown, AsciiDoc, and iiRDS metadata even when the content is authored or published in different forms. The vector index becomes a downstream search layer derived from governed knowledge instead of an independent memory that gradually fills with competing versions.

This does not eliminate the need for human governance. People still need to define the ontology, mappings, identifiers, lifecycle policies, and validation rules, including the mapping rules for each source format. But once those rules are established, software can apply them repeatedly and at scale. With XML/XSD, a particularly large share of the repetitive graph construction can be generated directly from the governed schema and markup. The experts maintain the model and the rules; the pipeline performs much of the construction, synchronization, validation, and selective re-embedding.

That distinction matters. A knowledge graph does not have to create a new maintenance burden. Properly designed, it can become the continuously updated semantic representation of information the organization is already governing – accepting multiple authoring formats while taking maximum advantage of the richer, more granular semantics available from XML/XSD.

Figure 3. The XML/XSD- and iiRDS-led digital-twin path provides highly automatic, fine-grained graph construction and synchronization. Markdown and AsciiDoc sources can coexist in the same semantic graph and feed the same graph-derived vector layer.



The graph, and your AI solutions, can keep getting smarter

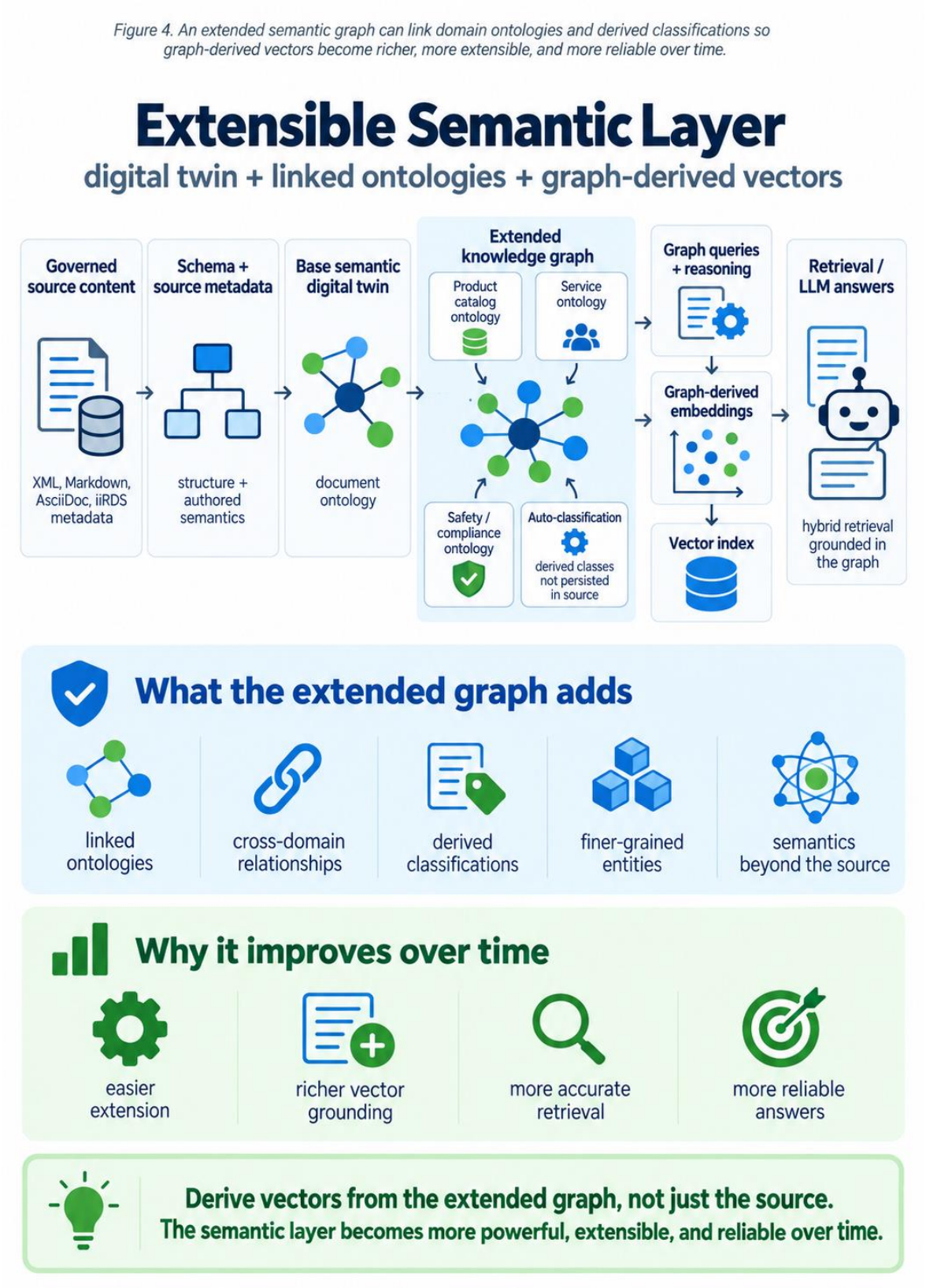
The source-derived digital twin is the stable foundation, not the ceiling. The schema provides the structural ontology – the classes, objects, containment, identifiers, and relationships implied by the document model – while semantic metadata in the source provides meaning such as product, component, information type, applicability, and lifecycle. Once those source-derived facts are represented in a graph, they can be linked to additional domain-specific ontologies that describe knowledge the document model was never intended to own.

This gives the graph a long useful half-life and a path to constant improvement. A product, service, regulatory, organizational, geographic, or other domain ontology can be added or refined without redesigning every source document. The ontologies do not have to be collapsed into one giant model; they can remain modular and be linked through shared identifiers and explicit relationships. At the graph-database layer, rules, reasoning, and content auto-classification can also create useful derived classifications that do not need to be persisted back into XML, Markdown, or AsciiDoc. Those graph-level assertions can be regenerated as classifiers and business rules improve, while the governed source remains the authority for what authors explicitly maintain.

A simple example. The document ontology might define Topic, Procedure, Step, Warning, and an appliesTo relationship. A separate product-catalog ontology might define ProductFamily, Model, SKU, Component, and compatibleWith. At the graph layer, a procedure whose source metadata says it applies to Model X can be linked to the catalog entity for Model X. The graph can then traverse from that model to its product family, installed components, supported SKUs, or successor models. A graph-level classifier could additionally classify the procedure as BatteryService or HighVoltageContent based on its content and relationships, even if authors never maintain those classifications in the source.

That separation is powerful: the source remains clean, governed, and authorable; the graph becomes the extensible integration and intelligence layer. Linked ontologies can extend the schema-derived document ontology over time, and new domain knowledge can improve retrieval, reasoning, classification, and downstream applications without requiring every semantic enhancement to be pushed back into every content source.

Figure 4. An extended semantic graph can link domain ontologies and derived classifications so graph-derived vectors become richer, more extensible, and more reliable over time.



Why vector-only systems can become harder to trust over time

Many real-world failures happen because similarity is being asked to stand in for identity, version, status, and rules – things a conventional embedding does not represent explicitly. The vector search may be

working exactly as designed and still retrieve the wrong information for the decision the user is trying to make.

Similar is not the same as correct

Suppose support content exists for Product A version 4 and Product A version 5. Much of the wording may be nearly identical. A question about version 5 can therefore retrieve a version 4 passage because it is semantically very close. If version information is missing, weakly attached, or not used as a strict filter, the LLM may give a fluent answer based on obsolete instructions. The retrieval can be mathematically reasonable and operationally wrong.

Stale vectors can create a mixed memory

A vector database does not have to become stale. Modern vector systems can update, overwrite, and delete records. The risk appears in naïve ingestion pipelines that simply add newly published chunks without reliably finding and retiring the chunks they replace. Over time, the index can contain several generations of nearly identical guidance. Current, obsolete, and duplicate chunks may all score well for the same question, making retrieval less dependable as the collection grows.

The key point is that the vector itself does not inherently know that one passage supersedes another. That lifecycle meaning has to be carried in metadata and enforced by the surrounding pipeline. In a knowledge graph, relationships such as currentVersion, supersedes, appliesTo, effectiveDate, and retired can be modeled directly and queried explicitly.

Hard-coded answers become maintenance debt

A common response to repeated mistakes is to patch the system: add another prompt instruction, another exception, another routing rule, or a hard-coded answer for a known question. This may solve one visible problem, but each patch becomes another rule that must be remembered, tested, and kept synchronized with the source content. As products, versions, and policies multiply, the workaround becomes a growing web of special cases. The return on each new patch gets smaller while maintenance cost and the chance of contradiction rise.

Weighting changes the odds, not the facts

Another common fix is to boost newer chunks, preferred sources, certain metadata, or particular keywords so they rank higher. Weighting, reranking, and metadata filters are useful tools, and strict filters can prevent many mistakes. But weighting alone still changes the odds of what will be retrieved. It does not express that version 5 supersedes version 4, that a procedure applies only to one product family, or that one rule overrides another. A graph can store those facts as explicit relationships and let a query use them directly instead of merely hoping the preferred chunk receives the higher score.

A graph can move deductive reasoning out of the prompt

A semantic graph can also support formal deductive reasoning. With RDF/OWL and a suitable reasoner or rule engine, the system can derive conclusions from explicitly encoded relationships before the LLM is asked to compose an answer. For example, if the graph states that version 5 supersedes version 4, that a feature applies only to version 5, and that a customer has version 5, the system can determine which facts apply without asking the LLM to read several competing passages and reconstruct the relationships from prose.

The LLM can then receive a much smaller, already-grounded result and explain it in natural language. Graph queries and reasoning still require computing resources, but the logical work does not have to consume a large LLM context window or generation tokens. This can reduce both cost and variability, especially when the same relationships and rules are used repeatedly across many questions.

This is an important distinction: vectors rank candidates. A governed graph can state what is asserted, traverse exact relationships, validate constraints, and – under the encoded ontology and rules – infer what follows. Those conclusions are reproducible from the same graph state and rule set. They are not

guaranteed to be true about the real world if the graph itself is wrong, but they do not depend on an LLM probabilistically reconstructing the logic each time.

Example	A user asks: "Does Feature Z apply to Product A version 5?"
Vector-only risk	Version 4 and version 5 passages use almost the same language. If both remain in the index, either can rank highly and the LLM may blend or select the obsolete guidance.
Graph + vector	Vectors can still find relevant material, while the graph explicitly identifies the product, version, supersession status, and applicability. Retrieval and reasoning can therefore use the exact relationship instead of similarity alone.

The difference in practical terms

Question the system must answer	Naïve vector approach	Graph + vector approach
What seems related to this question?	Strong	Strong
Which product or version does this apply to?	Can be confused by highly similar content unless metadata and filters are rigorously managed	Version, applicability, and supersession can be explicit relationships
What happens when content changes?	Old chunks must be deliberately updated or retired; append-only pipelines can leave conflicting generations	Current, superseded, and retired states can be modeled and queried
How are these entities explicitly related?	Often inferred from retrieved text	Stored and queryable
Which rules or constraints apply?	Usually handled outside the vector index	Can be represented, validated, and reasoned over explicitly
How are known failures corrected?	Prompt patches, weights, filters, and special cases can accumulate	Correct the governed fact, relationship, or rule and reuse it across queries
Where did this fact come from?	Depends on attached metadata	Can be modeled as provenance in the graph
Where does the reasoning happen?	Often inside the LLM over retrieved text, consuming context and tokens	Graph query or reasoner can resolve relationships first; the LLM can explain the result
Can the same rule-based query be reproduced?	Not the main strength	Yes, from the same graph state and rule set

Where more deterministic behavior helps

Large language models and vector similarity are probabilistic. That is not a defect – it is part of what makes them flexible. But some business questions require a different kind of behavior. An organization may need to determine whether a rule has been violated, whether one component depends on another, whether a product satisfies a requirement, or exactly why a conclusion was reached.

For questions like these, explicit relationships and rules matter. A graph query over the same graph state can return the same result repeatedly. A reasoner or rule engine can produce conclusions that follow from the encoded ontology and rules. The LLM can still explain those results in natural language, but it does not have to invent the underlying logic or spend tokens repeatedly reconstructing relationships that the knowledge model already knows.

The practical takeaway

Vector databases remain valuable tools. The problem comes when they are treated as if similarity alone were a complete knowledge model. For simple search, that may be enough. For systems that depend on identity, relationships, provenance, constraints, or explainability, it often is not.

A more capable architecture preserves important meaning before retrieval begins. The governed source estate can include XML, Markdown, AsciiDoc, and iiRDS metadata. XML/XSD provides the richest path when fine-grained, machine-addressable semantics and highly automated graph construction are required; Markdown and AsciiDoc can still participate in the same model through explicit structure, metadata, and mapping rules. The knowledge graph normalizes those sources into shared entities and relationships, then can be extended over time with linked domain ontologies and graph-level derived classifications without forcing every enrichment back into the source. Vector retrieval provides flexible semantic search. Large language models provide natural-language understanding and generation.

Together, these layers can create AI systems that do more than find text that looks relevant. They can retrieve information in the context of an explicit model of what the information means and how it fits together – while allowing organizations to keep the authoring formats they need and exploit much deeper semantics wherever XML/XSD is available.

Vectors help find the information.
The graph preserves the meaning.

Selected technical references

- Twin Digital Twins
- W3C – OWL 2 Web Ontology Language: Document Overview
- W3C – SPARQL 1.1 Query Language
- W3C – Shapes Constraint Language (SHACL)
- Pinecone – Update records
- Weaviate – Update objects
- iiRDS Consortium – iiRDS Standard, metadata vocabulary, and RDF schemas
- W3C – XML Schema Definition Language (XSD)
- CommonMark – CommonMark Specification
- AsciiDoc – AsciiDoc Language Documentation

More from the author

Michael Iantosca publishes additional writing on structured content, knowledge graphs, and AI at ThinkingDocumentation.com.