Agentic Technical Documentation: Meeting the Snake in the Garden of Eden

The Devil Is in the Details

Michael Iantosca
Senior Director of Knowledge Platforms and Engineering
Avalara

Executive Synopsis and Main Thesis

If your company is aiming for a 10X improvement in software development and delivery through agentic AI, you will need to either hire significantly more writers or build and manage an industrial-strength, end-to-end agentic documentation workflow. Otherwise, you risk accumulating crushing documentation debt that will directly affect your customers - and your bottom line.

For enterprise-scale agentic documentation updates, an LLM alone is not enough. Reliable automation requires a governed, source-addressable semantic layer capable of identifying affected content, tracing dependencies, preserving version and publication context, and retrieving only the fragments that require revision. In our implementation, that foundation is a synchronized knowledge-graph digital twin, supported by graph-derived vector retrieval and tightly bounded LLM editing.

Whether documentation is written in Markdown, AsciiDoc, DITA, reStructuredText, or a combination of formats, the underlying requirement is the same: industrial-strength agentic workflows need an explicit, machine-readable model of the content and its relationships. We believe knowledge graphs provide the most robust foundation for that model.

Without such a foundation, AI agents must infer relationships across disconnected files, inconsistent metadata, incomplete context, and weakly defined dependencies. With a governed knowledge graph, the workflow can identify what changed, determine which information objects are affected, trace their relationships to products and publications, and retrieve the precise source fragments that require attention.

This article presents a production use case built on architectural models introduced in our earlier work. That work explains how a synchronized, source-addressable knowledge graph can be generated from structured documentation, enriched with broader semantic context, and combined with graph-derived vector retrieval. Here, we examine a more demanding question: Can that architecture support repeatable impact analysis and bounded revision when existing documentation must be updated in response to product changes?

Paradise: The Net-New Agentic Documentation Workflow

Everything was peachy in the Garden of Eden – until the serpent arrived and persuaded Eve to bite the forbidden fruit. Fast-forward a few millennia, and we have a new apple: agentic documentation. It promises a technological Garden of Eden, where we can use stateful and stateless agents and agentic workflows to create and update content. The apple is polished, tempting, and transformative. Some would call it epochal. When it works, it can be remarkably sweet. But every garden has a serpent, and when an agentic documentation process fails, the consequences are truly devilish. The devil, as always, is in the details.

In this modern scenario, I have the dubious honor to play Adam. I use every power at my disposal to warn about the snake in the weeds, to no avail. When Eden shatters, the only excuse is, "The devil made me do it."

My apologies for the allegory, but it turns out the allusion is particularly appropriate for this article.

The sad tale begins with a senior leadership directive to design and build an automated, multi-agent system for documentation creation and updates. So, we did exactly that. It's not the agentic system that is sad; it's the snake we encountered in the garden we needed to eradicate.

We spent several months designing and cultivating our garden. We didn't cut corners. We painstakingly analyzed our audiences, content, and processes in minute detail. We researched and selected the agentic framework that would orchestrate the workflow – Temporal, in our case, for the technically inclined. We defined the roles and skills of the dozens of agents required to support end-to-end workflows, along with the criteria, guardrails, and governance strategies needed to coax reliable agentic fruit from our garden.

It went swimmingly well – until…

We first deployed an agentic workflow. We called it the Net-New Agentic Workflow. The premise was straightforward: require upstream product experts to provide an adequate set of knowledge assets before the workflow could automatically launch, then have the automated agents use those assets to automatically generate high-quality initial drafts of end-user technical documentation articles and multi-topic user guides.

It wasn't easy, but after some trial and error – and a healthy amount of Python code – we hit pay dirt.

It was an industry-first achievement. In our initial implementation, some generated guides were assessed as more than 80 percent complete and accurate, reducing the time required to produce an initial draft from more than two weeks to approximately twenty minutes.

Product developers and designers were required to provide the necessary collateral – PRFAQs, narrated videos, user interface prototypes, Figma design assets, code, and other knowledge assets – through agentically validated Jira tickets. Our automated AI agents processed those assets, and the LLM generated the initial drafts using guardrails (agent skills and structured models).

The apple smelled very sweet. Naturally, we were tempted to pluck it from the tree and take another bite.

We continued to build out a complete, production-deployable end-to-end agentic workflow for net-new documentation. Let's call her Eve.

We added an agent to optimize and score the generated content. For that role, we used MarkupAI (formerly Acrolinx), because it was a proven, deterministic, and fully governed solution for enforcing style, grammar, terminology, and other content standards with a new API to use it programmatically in a workflow agent that the original Acrolinx system lacked.

We made Eve stateful, allowing us to pause it whenever side processes or human-in-the-loop interactions were required, then resume it afterward without losing context. Subject-matter experts could review, compare, edit, correct, complete, and optimize the generated content in a subsequent workflow step. Once they finished, a swarm of agents handled the remainder of the production workflow: automatically classifying

the content using the taxonomy, storing it, and publishing it first to staging and eventually to production delivery channels, complete with an audit trail.

That doesn't mean documentation professionals are no longer needed; they provide the second-level human-in-the-loop function starting from a much better position. They also own the design and governance of the workflow itself – sort of the Governors of Documentation (the GoD role in my tongue-in-cheek analogy).

More agents remained to be added for advanced tasks: deeper validation, accessibility, legal and trademark review, automated documentation testing – docs as tests – and more. But our Net-New Agentic Workflow had reached MVP deployment.

Life was good. The future seemed bright. Eve wasn't merely attractive; she was wickedly smart.

We gave Eve skills and memory. We even externalized the governance of those skills through MCP, so we didn't have to wake Eve up – or recompile her – every time we needed to modify or extend agent skills and templates to improve the model or to support new types of documents. We ensured that our agents captured metrics across the entire process with dashboards. We had operational governance. We had what appeared to be a functioning agentic AI paradise.

It couldn't get much better, we thought. We were featured in the company news and given a coveted spotlight to demonstrate the system to senior executives and the board. Bear in mind that, until then, documentation and technical documentation professionals rarely received higher billing than the maintenance staff. Now our hybrid documentation-development and platform-engineering team was receiving the kind of attention and accolades rarely afforded to documentation organizations.

Enter the Devil: The Net-Update Challenge

Fresh from the smashing success of our Net-New agentic documentation strategy and workflow, we were handed the next never-before-in-the-history-of-humankind challenge: create an end-to-end *Net-Update* Agentic Workflow capable of automating the myriads of constant changes required across existing documentation whenever products were updated. Net new automation is great, but any seasoned technical writer will tell you that making updates to existing documentation comprises much more of their day-to-day activities as their products are constantly changing and evolving.

At first, the idea didn't seem insurmountable. After all, we were still drunk on the nectar of our success with Eve. We didn't even mind that she had cost us a rib or two. We were virtually alone in this new world; few in the documentation industry were even close.

We extended the Net-New workflow so it could branch whenever the assigned task involved updating existing content rather than creating something new.

The principal challenges were formidable:

1. Determine what information had changed.
2. Identify the existing topics affected by those changes.
3. Find and retrieve the correct versions of those topics.
4. Update each affected topic accurately and in context.
5. Integrate the revised topics back into the larger documentation set – without breaking anything else.

Sounded simple enough, right?

Trouble in Paradise: When the LLM Became the Search Engine

The moment we began testing the Net-Update Agentic Workflow, we found ourselves face-to-face with the devil himself – the most unwelcome guest in our agentic paradise.

We had the requisite knowledge assets from the subject-matter experts to auto-launch the Net-Update workflow – which, by the way, requires something akin to herding cats when you don't yet have automated validation agents integrated with your agentic workflow.

We tasked the LLM to determine precisely which source topics were affected, locate and retrieve them from a corpus containing countless thousands of topics and user guides, and feed them back into the agentic workflow for updating.

That was the moment Eden began to crack.

It's one thing to use a chatbot to find content on a website and answer user questions; It's an entirely different challenge to automate finding, retrieving, and updating the source documentation.

LLMs are, by their very nature, probabilistic. Each time we ran the Net-Update workflow, the collection of topics identified as requiring updates was different. One run identified five topics. The next identified three. The run after that identified ten.

There was little consistency because probabilistic similarity does not produce deterministic results. LLMs rarely return precisely the same answer twice. When the task requires repeatable precision, only deterministic methods can reliably provide it.

The devil wasn't done with us. Not by a long shot.

Searching even a single large user guide using an LLM proved both painful and expensive. One run could consume upwards of eight hours of LLM processing time while devouring an inordinate number of budget-busting LLM tokens. We quickly exhausted our token allocation in just two days and ran up a rather nasty bill through light testing alone.

Ka-ching!

But hold your vectors – it gets worse.

Our Net-Update Agentic Workflow retrieved the affected topics and fed them to Luther-the-LLM. True to character, the LLM proceeded with its natural and inherent mischief. It updated the topics, all right, but it also altered portions of those topics that should never have been touched. Some changes were obvious. Others were subtle – and therefore far more dangerous.

Personally, I wasn't surprised one bit; I expected it to a degree. I just didn't know how severe and numerous the road blocks would be until implementation and testing. We simply hadn't had the proof, until now.

Many of us have used ChatGPT, Claude, Gemini – take your pick – and witnessed this phenomenon as the context grows larger. The devil's ugly horns begin to appear quickly when an entire topic spanning several pages is fed to an LLM. The problem is excessive volume. That's not context, it's noise.

I had encountered the same behavior outside technical documentation while using generative tools to revise long-form fiction and music. Ask the system to change one line, passage, or instrument, and it may obligingly alter several others that were supposed to remain untouched. In creative work, that is frustrating. In technical documentation, it is a deal breaker.

The same problem applies to updating existing documentation: you rarely receive precisely the same change twice, and you often receive the added insult of unwanted – and sometimes incorrect – modifications to material that was never affected in the first place. Subtle changes easily go unnoticed and can result in serious consequences.

Deterministic Angel: Narrowing the Work Before Generation

We were stuck in the burning fire of Hades. Without deterministic preprocessing, we would have to punt and revert to human authors, with AI serving as a marginally useful assistant rather than providing the high degree of reliable workflow automation we had set out to achieve.

The problem wasn't a lack of source context. We already had some of the most semantically rich source content possible, encoded in machine-optimized semantically rich DITA/XML, but that's only the first half of the solution.

Here is where a persistent myth enters the story: LLMs don't care as much as most assume whether content is structured or unstructured, or how richly it is encoded – unless it is also trained on the underlying content schema (hint: most aren't). The LLM will happily flatten beautifully structured markup into another stream of tokens throwing the rest of your content knowledge into the bit bucket and proceed probabilistically from there.

The real value of structure and semantic context isn't for the LLM; it is for the deterministic preprocessing that must happen before the content ever reaches the LLM. Deterministic preprocessing produces consistent, repeatable results while sharply reducing the opportunity for hallucination downstream.

That left us with a daunting challenge: How could we repeatedly search an massive source-content corpus in seconds or minutes rather than days, accurately and repeatedly identify every topic in a requiring an update, isolate only the affected subsections, and send only those precise fragments to the LLM for revision – without incinerating a small fortune in tokens?

For our architecture, one answer emerged as both practical and extensible: a knowledge graph representing the entire source content corpus.

Digital Twins to the Rescue

A couple of years earlier, we had invented and published the DOM Graph RAG open knowledge model. DOM stands for Document Object Model. When synchronized with the source content and extended into a graph representation; that model functions as what we call a *digital twin* – an ontological replica of the content store. It proved useful for building better chatbots and interrogating existing content. Now it was time for the main event: using it to make our agentic Net-Update solution work.

Until this point, having a knowledge graph might have been viewed as a luxury. But faced with the seemingly insurmountable challenges of Net-Update, one fact became crystal clear: For the level of scale, precision, and source fidelity we required, a knowledge graph and a graph-derived vector model were no longer optional.

With help from our partners at GraphWise, who co-developed the DOM Graph RAG model with my team at Avalara, we began redesigning the DOM graph model as a more robust digital twin.

For brevity, I won't explain the DOM Graph RAG digital-twin model in detail here. You can read a highly consumable introduction on Medium:

> https://medium.com/@nc_mike/document-object-model-graph-rag-af8ae452b0b6?sharedUserId=nc_mike

You can also read an overview of deterministic methods and digital twins here:

> https://medium.com/@nc_mike/deterministic-and-agentic-ai-architectures-for-technical-documentation-3fb2956a1334.

What the Devil is a Digital Twin?

When we set out to create a knowledge graph representing our massive source document corpus, we encountered an inconvenient truth: manually building and maintaining such a graph was a nonstarter.

Years earlier, I had watched one of my leading industry peers build a knowledge graph for the MSDN library. My immediate reaction was, "You're not going to mentally or physically survive manually building and maintaining that graph at scale for long." This is a universal and generally prohibitive issue with most knowledge graphs: most must be continuously validated against the authoritative source and maintained at scale. This is what we did using the DOM Graph RAG model, subsequently augmented using iiRDS which we'll get to shortly.

That realization was the genesis that sparked the development of the DOM Graph RAG digital knowledge graph model.

The central idea was stunningly simple: take the existing open-standard schema of our DITA document model and use it to generate an OWL ontology for the graph. This wasn't a novel idea; someone did an experiment way back in 2013 at the dawn of knowledge graphs that demonstrated such was possible. It's also how many graphs have been generated historically from relational databases over the years. OWL stands for Web Ontology Language. Converting the document schema into an OWL ontology was a straightforward, direct mapping.

The ontology was then loaded into a knowledge graph database, forming the semantic foundation and backbone for the knowledge graph. We use GraphDB from GraphWise as our RDF and OWL graph database.

The next step was to generate an RDF output representation of our source documentation files. RDF stands for Resource Description Framework.

Once the document schema had been loaded into GraphDB, all we had to do was transform and load the RDF output of the documents themselves. Voilà: an automatically constructed knowledge graph representing the entire source content corpus. The ontology serves as the blueprint against which the RDF export of our content is automatically mapped in the graph database which automatically and accurately generate the base graph of the documentation which can then be queried for many applications and purposes. Domain-specific taxonomy classification and ontology later extends the semantic intelligence and precision of the graph continually over time. By the way, there already exists a DITA-to-RDF transform add-on for the DITA open toolkit for you tool folks.

Once the transformation pipeline was established, maintaining the graph was nearly as easy as making instant coffee; it retained every available piece of document structure, metadata, content, reference, and relationship –all that intelligence a typical vector database or LLM throws away. That gave us maximum preservation and use of the original, lossless, non-chunked context. For shops that lack the expertise or resources to do this, I highly recommend working with the services organization of your knowledge graph database provider – we've provided the blueprint for you on a silver platter in the links referenced earlier.

Understand that the graph is not the documentation source. The authoritative source remains in the content management system – which, by the way, is also a prerequisite. Shops that have a real component content management system (cCMS) are already at heaven's gate because that means you already have persistent content-object identifiers that make such automated retrieval child's play, accurate, and reliable. The graph contains a synchronized representation of that source, together with links back to the original files.

In other words, the knowledge graph is a virtual mirror of your content corpus – that's why we call it a digital twin.

With this digital twin, we can query, interrogate, retrieve, infer over, and deterministically reason across our source material at extraordinary speed – without involving an LLM – and the budget-breaking token costs that come with it.

The graph enables explicit *symbolic reasoning* rather than relying solely on the probabilistic, pattern-based reasoning produced by large language models. I sometimes refer to the latter as "Splenda Reasoning": it can resemble genuine reasoning in form, yet it is primarily generated through learned statistical associations rather than the direct execution of formal logical rules.

By representing facts, entities, and relationships in a graph, the system can apply explicit rules, deductive logic, and symbolic inference. Instead of merely generating an answer that is statistically plausible in context, it can derive conclusions through transparent, inspectable reasoning steps grounded in the underlying data and relationships – otherwise known as *explainable AI*.

But we had another problem.

Our documentation ecosystem, like many, contains both highly structured content, primarily in DITA XML, and unstructured content. How could we represent and manage both within the same knowledge architecture?

Easy: two digital twins!

One represents highly structured content, such as XML. The other represents unstructured content using iiRDS, which can be Markdown or any other document format.

As it turns out, bringing the two together put our structured content on virtual steroids.

And that brings us to iiRDS.

What the Heck is iiRDS?

iiRDS stands for **i**ntelligent **i**nformation **R**equest and **D**elivery **S**tandard.

That is a mouthful, so let's put it in plain English.

Imagine walking into a massive warehouse filled with millions of unmarked boxes. The thing you need is probably in there, but finding it requires opening boxes one at a time.

That is what unstructured documentation often looks like to a computer. The words are available, but the computer may not know what each piece of information is about, who it is for, when it should be used, or how it relates to a particular product or task.

Now imagine that every box has a standardized label:

- This contains installation instructions.
- This applies to product model X.
- This is intended for a service technician.
- This procedure is used during maintenance.
- This information applies only to software version 4.2.
- This warning concerns a high-voltage component.

Suddenly, the warehouse becomes much easier to navigate.

That, in essence, is what iiRDS does for technical information.

iiRDS provides a common vocabulary for labeling pieces of technical content. These labels are metadata: information that describes other information. They tell a system what a piece of content represents and under what circumstances it is useful.

The labels do not replace the documentation. They make the documentation understandable, exchangeable, and retrievable by machines.

This distinction is important.

DITA XML gives us detailed structure inside a document. It can tell us that something is a task, a step, a warning, a concept, or a reference topic – not simply a generic paragraph or bulleted list item. But many organizations also have useful information in Markdown, Word files, PDFs, web pages, support articles, service notes, and other formats that do not share the same internal structure.

iiRDS gives us a standardized way to describe information across all those formats.

For those with DITA source, you can think of DITA as describing the anatomy of a document. iiRDS describes the document's identity, purpose, situation, and relationship to the outside world.

For example, DITA might tell us:

This content is a warning inside step four of a repair procedure.

iiRDS can add:

This warning applies to the cooling system of product model X, during corrective maintenance, and is intended for a certified service technician.

Those two kinds of knowledge are extremely powerful when combined.

A Twin Digital Twin?

Earlier, I said that we wanted two digital twins.

The first digital twin represents the internal structure of our highly structured content. It preserves the rich hierarchy, semantics, references, and relationships found in sources such as DITA XML.

The second digital twin uses iiRDS to represent what our information is about and when it should be delivered.

This second twin can include both structured and unstructured content. A PDF may not contain the detailed semantic structure of a DITA topic, but iiRDS metadata can still tell us that the PDF is a troubleshooting guide for a particular product, component, task, user role, and operating condition.

In simplified terms:

- The DITA DOM digital twin tells us what the content is made of.
- The iiRDS digital twin tells us what the content is about.
- The knowledge graph connects the two.

That gives the DITA DOM digital twin a much richer understanding of the documentation estate.

Instead of seeing a collection of files or isolated chunks of text, the graph can see a connected information environment containing products, components, tasks, document types, user roles, lifecycle stages, versions, warnings, dependencies, and relationships.

Note: To be fair, if DITA XML – or another XML vocabulary – is your only source format, a properly designed DITA digital twin can be sufficient. Most organizations, however, must also manage semistructured formats such as Markdown and fully unstructured content, so we prefer the dual-twin approach.

Why Does That Matter?

Suppose a field technician asks:

How do I replace the cooling fan in model X without disconnecting the control module?

A conventional search system may look for documents containing words such as “replace,” “cooling fan,” and “control module.”

A vector-search system may retrieve passages that appear semantically similar to the question.

Both approaches can be useful, but neither inherently understands the complete technical situation.

With iiRDS metadata and the DOM Graph RAG digital twins, the system can reason over explicit facts:

- The user is performing a maintenance task.
- The product is model X.
- The component is the cooling fan.
- The user is a field technician.
- The control module must remain connected.
- The instructions must apply to the correct product version.
- Any required safety warnings must accompany the procedure.

The graph can use those facts to narrow the available information before an LLM is ever involved.

It does not merely ask, “Which passage sounds most like this question?”

It can ask, “Which information objects are known to apply to this product, component, task, user, version, and operating condition?”

That is a very different kind of retrieval.

It is the difference between guessing which box in the warehouse probably contains the answer and reading standardized labels that tell us exactly where the answer belongs.

A Common Language for Information

iiRDS also helps solve a problem that extends beyond a single company.

Manufacturers, suppliers, customers, content management systems, service platforms, and content-delivery portals often describe similar information in different ways. One system might use “repair,” another “corrective maintenance,” and another “service operation.”

Without a shared vocabulary, exchanging and combining that information becomes difficult.

iiRDS provides a common language that independent systems can understand.

It is similar to putting standardized shipping labels on packages. The companies sending and receiving the packages do not need to use the same internal warehouse system. They only need to agree on what the labels mean.

In the same way, iiRDS does not force every organization to use the same authoring tool, content management system, or document format. It provides a standardized method for describing and packaging technical information so that other systems can interpret it.

Putting Our Structured Content on Steroids

Our DOM digital twin already provides remarkably detailed representation of structured content. Adding the iiRDS digital twin gives that structure a larger technical and operational context.

Now the graph could understand not only that a warning belonged to a particular step in a procedure, but also that the procedure applied to a specific component, product variant, lifecycle stage, user role, and type of activity:

- The structure tells us where the information lives.
- The iiRDS metadata tells us when and why it matters.

Connecting the two enables systems to retrieve information more precisely, apply deterministic rules, detect explicit and inferred relationships, and assemble the right content for a specific situation.

That is what I mean when I say iiRDS puts structured content on steroids.

It transforms the knowledge graph from a highly detailed map of documents into a context-aware model of technical information and its use in the real world.

That is exactly what our enterprise-scale Net-Update workflow required – and, at that scale, it was no longer optional

Use Case: Using the Digital Twin to Find Documentation Affected by Product Changes

One of the most valuable uses of the digital twin is helping documentation teams find which topics need to be updated when a product changes across a large corpus of existing source documents.

In many organizations, product changes are explained through several kinds of input. These may include video demos, Figma designs, UI prototypes, source code, PRFAQs, release notes, engineering notes, Jira tickets, and conversations with product managers or engineers. Today, an AI system may take those inputs, create a large prompt, and ask a language model to review a long user guide or a large set of topics.

That approach can work, but it has significant limits. The language model may have to process hundreds of pages before it can decide what matters. This can take hours for a single user guide, and much longer for a large content corpus consisting of hundreds or thousands of user guides and topics. It can also cost significantly more than necessary in LLM token costs. It can increase the risk that the model changes content that should not be changed.

The digital twin offers a better approach.

Instead of asking the language model to search through all the documentation, the system first turns the product-change inputs into a structured description of what changed. For example, it may identify the affected product, component, feature, screen, procedure, user role, product version, error condition, or lifecycle event.

The system can then use that structured change description to query the content corpus using a vector database model that is derived from the graph and optionally augmented with additional graph queries.

For example, if a product change affects a reset procedure for Controller X, the graph can help find:

- topics about Controller X
- topics connected to the affected component
- troubleshooting topics related to the changed behavior
- maintenance tasks that include the reset procedure
- service technician content for the affected product version
- specific steps, commands, warnings, notes, or UI labels inside those topics

This means the graph is not just finding whole documents. It can help find the exact parts of topics that may need attention.

The language model is then used only after the graph-driven model has narrowed the work. Instead of sending a 500-page user guide to the model, the system sends only the relevant topics or topic fragments. The graph-driven model is asked to update those selected pieces, while leaving the rest of the content alone.

This changes the role of the language model. It is no longer being used as the main search engine across all documentation. The graph-driven model handles the search and impact analysis. The language model handles the writing task after the correct content has already been selected.

This approach has many benefits.

1. **Impact analysis**: The model can follow known relationships between products, components, topics, maps, information units, and versions. The system does not have to wait for a language model to read every page before it finds the likely targets.
2. **Cost avoidance**: The language model receives a much smaller set of content. It does not need to process an entire guide when only a small number of topics or steps are affected.
3. **Reduce hallucination risk**: The model is given a bounded task. It is asked to modify specific content fragments that the model has selected. This reduces the chance that the model will rewrite unrelated parts of a topic or invent changes that are not supported by the product-change input.
4. **Traceability**: Each retrieved topic or fragment can be connected back to the DITA source, iiRDS delivery view, product or component vocabulary, publication context, and provenance record. Reviewers can see why the system selected a topic and what product-change signal caused it to be included.

Adding a Vector Database

A vector database, **derived from the digital twin knowledge graph**, can be added as an optional companion to the dual digital twin. It should not replace the graph. The graph remains the trusted source for products, components, topics, versions, relationships, provenance, and validation. The vector database addresses a different problem: finding content that is similar in meaning, even when exact graph relationships are missing or incomplete.

This can be useful because product-change inputs are often described in natural language. A video demonstration might say that “the device no longer needs to be power cycled after a fault.” A Figma design might show a changed recovery screen. A PRFAQ might describe a new “self-recovery experience.” These inputs may all refer to the same product change, but they may not use the exact words that appear in the documentation or the graph.

The vector database can help connect these different ways of expressing the same concept.

For example, the graph may find topics that are explicitly connected to:

- Controller X
- reset procedure
- fault recovery
- power module
- firmware version 4.2

The vector database may also find topics, sections, or steps that use related wording, such as:

- restart the unit
- recover after lockup
- disconnect power
- flashing amber indicator
- device becomes unresponsive
- restore operation
- clear the fault

Some of these topics may not have complete metadata. They may not be directly linked to the affected feature or component in the graph. Without vector search, they might be missed. With vector search, they can be suggested as possible additional candidates.

The best pattern is to use the vector database for fast source-topic identification and gap detection. The vector database suggests content that appears semantically similar to the product change. The graph then evaluates those suggestions. It can confirm whether each candidate belongs to the correct product, component, version, role, language, lifecycle status, publication, and delivery context.

RDF graph platforms can also apply deterministic inference and validation to identify related content through SPARQL queries, OWL reasoning, and SHACL constraints. Those capabilities are complementary to vector search, although they are more appropriate for an advanced curriculum.

In plain language, the vector database asks:

“Does anything in the documentation sound like this product change?”

The graph then asks:

“Is this content actually connected to the correct product, version, audience, source, and delivery context?”

This approach is especially useful in large or older documentation sets where metadata may be inconsistent. In a library of 10,000 topics, the graph may confidently identify 40 affected topics. The vector database may suggest 12 additional topics that use different wording or were not fully tagged. After graph-based filtering, perhaps 5 of those additional topics are confirmed as likely affected. Those 5 topics might otherwise have been missed.

The vector database can also operate below the topic level. Instead of embedding only whole topics, the system can create embeddings for smaller content units, such as sections, short descriptions, prerequisites, steps, results, warnings, notes, troubleshooting conditions, commands, UI labels, or other meaningful elements. This allows the system to identify not only the affected topic, but also the specific part of that topic that may need to change.

The embedding level should be selected carefully. Very small inline elements, such as bold text, individual cross-references, or isolated command fragments, may not contain enough semantic meaning and can introduce noise. Larger structural elements, such as short descriptions, prerequisites, procedures, steps, results, sections, and section divisions, usually provide stronger semantic signals. The architecture can therefore allow teams to specify which DITA elements should be embedded and retrieved.

A prototype generated from the knowledge graph demonstrated the practical value of this design. Using 126 DITA maps, a local vector database containing approximately 72,300 section-level embeddings was created from the graph in about 20 minutes. Searches returned deterministic results in under 20 seconds. A separate element-level vector store contained approximately 722,000 embeddings and returned results in about 30 seconds, although the smallest elements often produced noisier matches.

Because the vector database was generated from the graph and kept synchronized with the CMS and graph data, it could be used to identify relevant source content without first fetching and storing the complete structure of every map. This is significant because an individual map may contain more than 700 topics, and loading the full map into intermediate storage previously took approximately 10 minutes. The earlier process also required the language model to examine large volumes of topic content during topic identification, increasing both processing time and cost.

With the vector-assisted approach, the system can instead extract evidence from the product-change input and query the synchronized vector database. The same evidence produces the same retrieval result, supporting a more deterministic identification process. After the relevant topics or sections are identified, only that selected content needs to be fetched from the CMS and sent to the language model for revision.

This can further reduce hallucination and unintended-change risk. When a complete topic is sent to a language model, unrelated sections may be modified accidentally. When the system narrows the update to a specific section or major element, the model receives only the content that is likely to require revision. The remainder of the topic remains untouched.

The important rule is:

The vector database should suggest. The graph should verify. The language model should edit only the final selected content.

In plain language, the graph acts like an intelligent impact map for the documentation. It knows how products, features, components, topics, publications, versions, and delivery metadata are connected. The vector database acts like meaning-based radar. It helps find content that may be relevant even when exact graph connections are not yet complete.

This distinction remains important even when vector search is used as the primary discovery mechanism. Vector search can quickly locate semantically relevant content, but the graph is still required to provide authoritative context, relationships, provenance, lifecycle information, validation, and integration with other ontologies. The CMS, graph, and vector database should remain synchronized so that each supports a different part of the workflow without creating competing sources of truth.

At scale, this combined approach narrows the documentation set to the specific topics, sections, or elements most likely to require revision before the language model begins writing. The optional vector database improves the process by finding additional candidates that are semantically related but not perfectly modeled in the graph.

The result is a more controlled AI-assisted documentation workflow. The graph provides precision, structure, context, traceability, and validation. The vector database improves discovery, accelerates topic and section identification, and helps detect possible gaps. The language model provides drafting and revision support only after the correct content has been selected. Together, these capabilities allow documentation teams to respond to product changes faster, with lower processing cost, better coverage, and less risk of unintended content changes.

Could We Use Only a Vector Database Without the Graph?

That’s a fair question. Yes and no.

To update existing source content, the system must retain precise addressability back to the original topics and content structures. Virtually no traditional vector models preserve this level of source fidelity, including elements, links, topic types, relationships, and metadata. They are typically optimized for semantic retrieval and chatbot-style question answering rather than for maintaining a complete, addressable representation of the source.

It is possible, however, to build a vector model directly from the document source while retaining this information. That raises an important question: Why not use that model and skip the digital-twin knowledge graph altogether?

The first reason is that the graph supports a much broader range of applications. Graph queries are significantly more effective for tasks that require traversing explicit relationships, applying constraints, and performing deductive reasoning through rules or an inference engine.

The second reason is that we need a unified model capable of representing structured and unstructured content together. A knowledge graph provides a common semantic layer in which documents, metadata, business entities, classifications, and relationships can coexist and be queried consistently.

Most importantly, source content is often incomplete from the standpoint of semantic enrichment. Requiring subject-matter experts to manually classify large volumes of content for a specific domain is prohibitively labor-intensive. It is far more practical to perform domain-specific automated classification within the graph and then, after curation, write those classifications back to the source content for persistence, reuse, and portability.

Finally, knowledge graphs – particularly those based on open interchange standards such as RDF and OWL – can be linked to other enterprise graphs and ontological models. For example, the content graph can be connected to a company's product matrix, inventory systems, organizational models, or other sources of enterprise knowledge. These connections provide additional semantic context and intelligence that an isolated vector database cannot practically reproduce.

A vector representation of the source content remains a valuable component of the architecture, but using it without deriving it from the graph would sacrifice important capabilities in reasoning, integration, semantic enrichment, interoperability, and long-term strategic growth.

The Moral of the Parable

The governed graph establishes which facts, relationships, and constraints the workflow is authorized to treat as valid, and it determines which content is connected and potentially affected. The LLM is used only where probabilistic language generation is actually valuable.

Glossary

Key terms used in "Agentic Technical Documentation: Meeting the Snake in the Garden of Eden"

Agent
A software component assigned a specific task within a larger workflow. An agent may collect information, validate inputs, retrieve content, generate text, or pass work to another agent.

Agentic workflow
A coordinated process in which multiple software agents perform defined tasks, make limited decisions, and hand work from one step to the next.

Component content management system (cCMS)
A system that stores documentation as reusable content components rather than only as complete documents. A cCMS typically gives each topic or content object a persistent identifier, which makes precise retrieval and updating easier.

Deterministic
Designed to produce the same result when given the same input and rules. Deterministic processing is important when a workflow must identify the same affected content every time it runs.

Digital twin
A synchronized, machine-readable representation of a real source system. In this paper, the digital twin mirrors the documentation corpus, including its structure, metadata, references, and relationships, while the authoritative source remains in the content management system.

DITA
Darwin Information Typing Architecture, an XML-based standard for creating structured, reusable technical content. DITA identifies content types and elements such as concepts, tasks, references, steps, warnings, and notes.

Document Object Model (DOM)
A structured representation of a document and its elements. In this paper, the DOM provides the basis for modeling the internal structure of source documentation in a knowledge graph.

Embedding
A numerical representation of the meaning of a piece of content. Embeddings allow a system to compare passages and find content that is semantically similar even when it uses different words.

Graph-derived vector database
A vector database created from content and metadata represented in the knowledge graph. It supports semantic search while preserving links back to the graph and the original source content.

Hallucination
Output generated by a language model that is unsupported, incorrect, or unintentionally changes content that should remain unchanged.

Human in the loop
A workflow design in which people review, approve, correct, or complete automated work before it proceeds to publication or another critical stage.

iiRDS
The intelligent information Request and Delivery Standard. iiRDS provides a common vocabulary for describing what technical information is about, who it is for, when it applies, and how it should be delivered.

Inference
The process of deriving additional facts from known facts and defined rules. In a knowledge graph, inference can identify relationships that are not stated directly but follow logically from the available data.

Knowledge graph
A machine-readable network of entities, facts, and relationships. In this paper, the graph connects documentation topics and elements to products, components, versions, audiences, publications, and other technical context.

Language model or large language model (LLM)
An AI system trained to understand and generate language. In the workflow described in this paper, the language model is used primarily to draft or revise selected content after other systems have identified what should change.

Metadata
Information that describes other information. Examples include product, version, user role, language, lifecycle stage, content type, publication, and component.

Model Context Protocol (MCP)
A protocol for connecting AI systems to external tools, instructions, and data sources. In the workflow described here, it supports external management of agent skills and templates.

Ontology
A formal model that defines the kinds of things in a domain, their properties, and the relationships among them. An ontology gives a knowledge graph a consistent semantic structure.

OWL
Web Ontology Language, a standard used to define ontologies and support machine reasoning over classes, properties, and relationships.

Probabilistic
Based on likelihood rather than fixed rules. A probabilistic system may produce different results from the same input, which can be useful for language generation but risky for tasks requiring exact, repeatable identification.

Provenance
Information about where content or data came from, how it was created or changed, and which source evidence supports it.

RDF
Resource Description Framework, a standard for representing facts as relationships between identifiable entities. RDF is commonly used to build and exchange knowledge-graph data.

Semantic search
Search based on meaning rather than only exact words or character matches. Semantic search can find related content that uses different terminology.

SHACL
Shapes Constraint Language, a standard for validating RDF graph data against defined structural and business rules.

Source-addressable
Able to trace a retrieved or selected result back to the exact source topic, section, element, or file from which it came.

SPARQL
A query language used to search and analyze RDF knowledge graphs.

Stateful agent
An agent that retains workflow context across pauses, handoffs, or multiple steps. A stateful process can resume without losing information from earlier stages.

Stateless agent
An agent that performs a task using only the information provided for that specific request and does not retain workflow context afterward.

Structured content
Content organized using explicit, machine-readable elements and rules. DITA XML is an example because it distinguishes tasks, steps, warnings, notes, references, and other information types.

Symbolic reasoning
Reasoning based on explicit facts, relationships, and rules. Unlike similarity-based retrieval, symbolic reasoning can apply logical constraints to determine whether content is connected to the correct product, version, audience, or context.

Taxonomy
A controlled classification system used to organize content by categories such as product, component, task, audience, or subject.

Temporal
The workflow-orchestration platform used in the implementation described in this paper. It coordinates long-running processes, pauses, retries, and stateful handoffs between workflow steps.