If DITA Is Already Structured, Why Add a Knowledge Graph?

Understanding the distinct roles of DITA, XPath/XQuery, vector retrieval, and graph reasoning in agentic documentation updates

Michael Iantosca

Senior Director of Content Platforms and Knowledge Engineering

Discussion paper and technical appendix

Executive Summary

A knowledgeable reader asked a deceptively simple question: If the authoritative source is already DITA XML, why not use XPath and XQuery to discover dependencies, reuse, applicability, publication context, and structural relationships directly, rather than first materializing an RDF knowledge graph?

The honest answer is that, for a purely DITA-native corpus, a graph may not be required to solve the immediate retrieval problem. In one test, a vector representation derived directly from the DITA instances and another derived from the dual digital-twin graph solved the same state and retrieval problems equally well. Both approaches could identify the correct topics and affected elements quickly, reliably, and with source-level precision.

The graph's additional value lies primarily around the retrieval step, not necessarily inside it. The graph can enrich classifications, connect DITA to external ontologies and non-DITA systems, apply reusable business rules, infer wider impact through relationships, validate results, preserve explainable evidence paths, detect missing content, and provide a canonical semantic layer for many applications.

Bottom line
A DITA-native architecture using XPath, XQuery, and a source-addressable vector index may be sufficient for high-speed, deterministic net-update retrieval. A graph becomes more compelling when the problem expands beyond direct DITA traversal into enterprise integration, semantic enrichment, inference, validation, and reuse across many applications.

The Question

A thoughtful reader recently asked an important question about the architecture described in my article on agentic documentation updates:

Reader question
If the authoritative source is already DITA, why not implement the graph traversal layer directly with XQuery rather than first materializing an RDF graph? Dependencies, reuse, applicability, publication context, and structural relationships can all be discovered directly from the XML. Is the RDF graph primarily for integration with external ontologies and non-DITA data, or does it provide advantages even for a purely DITA-native corpus?

It is a fair question. It also exposes an important architectural distinction that is easy to obscure when graph and vector technologies are discussed together.

What Our Test Demonstrated

In one test, we created two vector representations of the same DITA documentation. The first was derived directly from the DITA source using XML-native technologies, including XPath and XQuery. The second was derived from our dual digital-twin knowledge graph, which represented both the internal structure of the DITA content and its broader technical context.

Both vector representations solved the state and retrieval problems described in the original paper equally well. Both could rapidly and reliably identify the correct topics, locate affected elements within those topics, and return only the source fragments that needed revision.

That result matters because it shows that the essential requirement for this particular application is not necessarily RDF or a knowledge graph. The essential requirement is a governed, source-addressable representation that preserves the structure, identifiers, metadata, relationships, and publication context needed to find the correct content reliably.

A carefully designed DITA-native solution can provide all of those capabilities.

What the DITA-Native Approach Can Already Do

DITA is not an ordinary collection of text files. It already contains machine-readable structure and relationships. A DITA system may know that a particular piece of content is:

- a task, concept, reference topic, warning, step, prerequisite, or result;
- reused through a conref, conkeyref, or key reference;
- included in one or more maps and publication structures;
- filtered by product, platform, audience, market, or version;
- connected to other topics through links, keys, dependencies, and reuse relationships.

XPath and XQuery can traverse and evaluate those relationships directly. A DITA-native processing layer can therefore answer questions such as:

- Which topics use this reusable procedure?
- Which publications contain this topic?
- Which steps are applicable to product version 4.2?
- Which warnings occur in procedures that use a changed component?
- Which maps include an affected troubleshooting topic?
- Which exact DITA elements should be sent to the language model?

A vector index can also be created directly from selected DITA elements. Each vector record can retain the topic ID, element ID, element type, source location, map context, applicability values, and other metadata. In that design, vector search finds content that is similar in meaning, while XPath and XQuery confirm where that content lives and how it participates in the DITA structure.

For the net-update workflow, that architecture can already provide high-speed retrieval, repeatable identification, source-level traceability, topic- and element-level precision, reduced token use, and smaller, safer language-model editing tasks.

A Better Way to Frame the Comparison

The real comparison is not knowledge graph versus vector database. A direct DITA architecture is far more capable than a typical standalone vector database.

The more accurate comparison
DITA + XPath/XQuery + a source-addressable DITA-derived vector index

versus

DITA + a synchronized knowledge graph + graph reasoning and validation + a graph-derived vector index

Both architectures can be effective. The graph does not automatically make vector similarity more accurate, and it does not magically discover a topic that a well-designed DITA-derived vector representation could

never find. Instead, the graph supplies a shared operational layer of facts, relationships, classifications, constraints, and evidence that can be used by vector search and by other systems.

A Plain-Language Analogy

Imagine that the DITA corpus is a well-organized library. The books are properly shelved. Chapters, sections, warnings, procedures, and references are clearly labeled. The catalog knows which books belong to which collections and which chapters are reused elsewhere.

XPath and XQuery are highly skilled librarians. They can follow the catalog, inspect the shelves, trace references, and retrieve exactly the right passage.

The vector index is a meaning-based search assistant. It can recognize that "restart the controller," "power-cycle the unit," and "recover after lockup" may refer to related actions even when they use different words.

The knowledge graph is a model of the larger world around the library. It can connect the books to products, components, software releases, customer types, regulatory obligations, support cases, engineering systems, and business rules.

The graph is not necessarily a better librarian. Its advantage is that it connects the library's contents to a broader map of organizational knowledge and can apply common rules across that map.

What Additional Value Can the Graph Provide?

1. Add classifications that are missing from the DITA

Source content is often incomplete from the standpoint of business or domain classification. A DITA topic may be correctly structured as a troubleshooting task but may not be classified with every relevant product component, failure mode, user role, lifecycle stage, service condition, or regulatory category.

For example, a topic may explain how to clear a controller fault. The DITA source identifies it as a troubleshooting task, but it does not explicitly identify the affected subsystem, related fault family, required technician certification, product variants that share the controller, or lifecycle stage in which the action is permitted.

A classification process can add those relationships in the graph. After review, selected classifications can be written back into the DITA source so they become portable and persistent. The vector index can then use the new classifications as filters or ranking signals.

2. Connect DITA to other ontologies

DITA describes documents exceptionally well, but enterprise workflows often need knowledge that does not originate in the documentation schema. A graph can connect the DITA representation to iiRDS, product ontologies, service taxonomies, safety models, software-release models, and company-specific vocabularies.

The vector representation can still perform the initial meaning-based search. The graph then uses these additional relationships to determine whether a result belongs in the final candidate set.

3. Connect documentation to non-DITA systems

A product change rarely begins as a DITA change. It may originate in Jira, a product lifecycle management system, a code repository, Figma, a component inventory, a release-management system, a customer-support platform, or a regulatory database.

A graph can connect those objects to the documentation model. For example, when a component changes, the graph can identify the component in the product system, find all product variants that use it, locate the procedures associated with those variants, identify every DITA topic that contains or reuses those procedures, determine which active publications contain those topics, exclude obsolete versions, and return the affected sections to the vector-assisted update workflow.

XPath and XQuery can traverse relationships inside the DITA estate. The graph becomes more useful when the path begins outside DITA and crosses several different systems.

4. Apply explicit business rules

Vector retrieval is based on similarity. It asks which content sounds as though it may be related to a change. A graph can apply rules that ask which content is permitted, required, missing, or logically affected according to known facts and constraints.

Suppose vector search identifies 20 sections that appear relevant to a firmware change. The graph can apply rules such as:

- include only topics applicable to firmware 4.2 and later;
- exclude archived publications;
- include all warnings inherited by the affected procedure;
- include every active topic that reuses the changed step;
- include troubleshooting topics for products that contain the affected controller;
- exclude instructions intended only for internal engineering audiences;
- flag any publication in which a required safety warning is missing.

The vector index proposes candidates. The graph evaluates those candidates against explicit rules.

How the Vector Representation and Graph Work Together

Step 1: Interpret the product change

An agent processes a Jira ticket, engineering note, demonstration, design asset, release description, or code change and extracts evidence such as product, component, feature, prior behavior, new behavior, affected version, user role, and procedure.

Step 2: Use vectors to find semantically related content

Suppose the product change states: “The controller now recovers automatically after a communication fault and no longer requires a manual power cycle.”

The vector representation may find passages containing phrases such as restart the controller, disconnect and restore power, recover from a communication failure, clear the fault, restore controller operation, or reboot the unit after lockup. This is where vectors are especially strong: they connect different ways of expressing the same concept.

Step 3: Use the graph to verify context

Not every similar passage is truly affected. The graph can check whether the passage concerns the correct controller, product, firmware version, user role, publication, lifecycle status, and authoritative source. It can also determine whether the passage is reused elsewhere and whether required warnings or prerequisites accompany it.

The graph is not replacing vector retrieval. It is testing vector results against known facts.

Step 4: Expand the impact set through relationships

Once a likely affected element has been confirmed, the graph can find content that may not be semantically similar but is nevertheless affected. Examples include a prerequisite that says the controller must be powered off, a warning attached to the old recovery procedure, a troubleshooting decision table that directs the user to power-cycle the device, a service checklist that requires recording the restart, or another topic that reuses the same DITA step.

Some of these objects may not rank highly in vector search because their wording differs from the product-change description. The graph finds them because they are connected.

Step 5: Send only the final affected elements to the language model

After vector discovery and graph verification, only the approved source fragments are retrieved from the content management system. The language model receives a narrowly defined task, such as: “Revise this step to describe automatic recovery. Do not change the warning, prerequisite, or surrounding steps unless separately identified.”

This reduces the chance that the language model will alter unrelated content.

Useful Graph Actions Beyond Retrieval

Automatically create an impact report

The system can produce a report showing the changed product behavior, every selected topic and element, why each item was selected, the relationship path connecting it to the change, the maps and publications in which it appears, the affected versions, the vector similarity evidence, and the rules that confirmed or rejected it.

Detect missing documentation

If the product graph shows that six product variants use a changed component but documentation exists for only five, the graph can identify the gap. Vector search is good at finding existing content. A graph can also reveal that expected content or relationships do not exist.

Detect conflicting applicability

A vector search may find a procedure that appears correct, while the graph discovers that the topic is classified for firmware 4.1 but appears in a guide for firmware 5.0. The system can flag the inconsistency before editing begins.

Propagate a change through reused content

A changed DITA step may be reused in multiple topics. The graph can identify the source element, every reuse location, every map containing those topics, every active publication created from those maps, and every product and version served by those publications.

Enforce safety completeness

If a changed procedure involves high voltage, a graph rule can require an approved warning and a qualified-technician audience classification. If the required warning relationship is missing, the workflow can block automatic publication and route the content for review.

Write improved classifications back to DITA

The graph may determine that a group of topics is consistently associated with a particular component, task type, or lifecycle stage. After validation, approved classifications can be written back into DITA, improving future source-derived vectors and downstream portability.

Explainability: Showing Why Something Was Selected

One practical advantage of a graph is that relationships can be presented as an evidence path:

Example evidence path
Firmware 4.2 changes automatic fault recovery
→ affects Controller X
→ Controller X is used in Product Models A, B, and C
→ Recovery Procedure P applies to those models
→ Step S instructs the user to power-cycle the controller
→ Step S is reused in Topics T1, T2, and T3
→ Those topics appear in active Publications G1 and G2

A DITA/XQuery system can also generate this evidence, but it must be deliberately programmed to capture and present the complete traversal. A graph naturally stores relationships in a form that can be returned as an inspectable path.

Validation with SHACL – and Its XML Equivalent

A graph can use SHACL to validate structural and business rules, such as whether every active procedure has a product classification, every safety-critical task has a warning, every vector record points to an authoritative source element, and every automatically classified relationship includes provenance.

It is important not to imply that XML lacks validation capabilities. Similar checks can be implemented using XML Schema, Schematron, XQuery, or application code. The graph's advantage is not exclusive validation. Its advantage is that the same validation framework can operate across DITA content, iiRDS metadata, enterprise entities, inferred relationships, and data originating outside the XML environment.

A Single Canonical Semantic Layer for Many Uses

The documentation source should remain authoritative in the content management system. The graph should not replace it. Instead, the graph can serve as a canonical semantic representation used by multiple applications, including agentic updates, customer-support retrieval, dynamic delivery, product portals, impact analysis, compliance checks, content-health dashboards, documentation testing, missing-content detection, recommendation systems, training-content assembly, and product or service assistants.

Without a shared layer, each application may independently rebuild its own interpretation of products, topics, versions, audiences, and relationships. The graph allows those interpretations to be modeled and governed once, then reused.

Is the Graph Required for a Purely DITA-Native Corpus?

Not necessarily. If an organization has a fully structured DITA corpus, persistent source identifiers, a capable XML database, strong XPath and XQuery expertise, complete map and applicability metadata, a source-

addressable vector index, and no pressing requirement to connect external systems or ontologies, then a DITA-native architecture may solve the net-update problem extremely well without first materializing an RDF graph.

Our test demonstrated that a vector representation derived directly from DITA and one derived from the dual digital-twin graph could solve the tested state problems equally well. In that situation, the graph should not be justified merely by claiming that it retrieves the correct topics faster or more accurately. That claim would need to be demonstrated through measurement and may not be true.

The stronger justification is that the graph provides capabilities around retrieval: semantic enrichment, attachment of additional ontologies, reasoning across systems, reusable business-rule execution, relationship-based expansion of vector results, validation across multiple data types, inspectable evidence paths, missing-content detection, and enterprise-wide reuse of the same semantic model.

Conclusion

A knowledge graph is not the only way to build a reliable agentic documentation-update workflow. For a DITA-native corpus, XPath, XQuery, and a carefully designed source-derived vector representation can provide fast, repeatable, element-level retrieval while preserving source fidelity and publication context.

The graph becomes valuable when the problem expands beyond direct DITA retrieval. It can classify what the source has not classified, connect content to outside systems and ontologies, apply shared business rules, infer additional impact through relationships, validate candidates, expose the reasoning behind a decision, and support many applications from a single governed semantic layer.

The most useful division of responsibility
The vector representation finds what might be relevant.

The graph determines what is connected, permitted, required, missing, or affected.

XPath and XQuery retrieve and manipulate the authoritative DITA source.

The language model edits only the final, bounded content selected by those deterministic systems.

For the narrow task of finding the right DITA topics and affected elements, the graph may be optional. For a broader enterprise knowledge and automation strategy, it can provide value far beyond retrieval.

Appendix A – Example Use Cases and Graph Queries

This appendix summarizes representative combined vector-and-graph use cases and provides illustrative SPARQL and SHACL examples. The namespace and property names are examples only. They should be adapted to the organization's DITA ontology, iiRDS model, product ontology, provenance model, and graph implementation.

A.1 Example namespace assumptions

```
PREFIX dita:  <https://example.com/ontology/dita#>
PREFIX prod:  <https://example.com/ontology/product#>
PREFIX pub:   <https://example.com/ontology/publication#>
PREFIX ex:    <https://example.com/resource/>
PREFIX prov:  <http://www.w3.org/ns/prov#>
PREFIX sh:    <http://www.w3.org/ns/shacl#>
PREFIX rdf:   <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX xsd:   <http://www.w3.org/2001/XMLSchema#>
```

A.2 Use-case summary

Use case	Vector contribution	Graph contribution	Resulting action
Impact analysis	Find semantically similar topics, sections, and steps.	Verify product, version, publication, reuse, and lifecycle context; expand through relationships.	Produce the final bounded update set.
Reuse propagation	Find the changed source fragment or similar instructions.	Trace conref/keyref reuse and all publication contexts.	Update once or create controlled variants.
Missing-content detection	Find existing related content.	Compare expected product/component coverage against actual documentation links.	Create a documentation gap or authoring task.
Safety validation	Locate procedures discussing hazardous actions.	Require warning, audience, and safety classifications.	Block publication or route for review.
Applicability conflict detection	Identify likely relevant instructions.	Compare product and version constraints across topic, map, and publication.	Flag inconsistent applicability.
External change propagation	Match Jira or release language to documentation wording.	Connect changed product entities to DITA topics and publications.	Generate an enterprise impact report.

Classification enrichment	Suggest semantic categories from content meaning.	Store, govern, validate, and connect classifications to formal ontologies.	Write approved metadata back to DITA.
Explainable selection	Provide similarity scores and matching text.	Return an inspectable relationship path and rule evidence.	Show reviewers why each item was selected.

A.3 Retrieve active DITA elements associated with a changed component

Use after vector retrieval to confirm that candidate elements are connected to the affected component and occur in active publications.

```
SELECT DISTINCT ?topic ?element ?elementType ?publication
WHERE {
  VALUES ?changedComponent { ex:ControllerX }

  ?topic a dita:Topic ;
         dita:containsElement ?element ;
         prod:appliesToComponent ?changedComponent .

  ?element a ?elementType .

  ?publication a pub:Publication ;
               pub:includesTopic ?topic ;
               pub:lifecycleStatus pub:Active .
}
ORDER BY ?publication ?topic ?element
```

A.4 Expand a selected element through DITA reuse relationships

Find all active contexts in which a changed source element is reused, including transitive reuse chains.

```
SELECT DISTINCT ?sourceElement ?usingElement ?usingTopic ?publication
WHERE {
  VALUES ?sourceElement { ex:Step_Reset_Controller }

  ?usingElement dita:reusesElement+ ?sourceElement .
  ?usingTopic dita:containsElement ?usingElement .
  ?publication pub:includesTopic ?usingTopic ;
               pub:lifecycleStatus pub:Active .
}
ORDER BY ?publication ?usingTopic
```

A.5 Verify product and version applicability for vector candidates

Filter semantically similar candidates to those that are valid for the affected product and version.

```
SELECT ?candidate ?product ?minVersion ?maxVersion ?publication
WHERE {
  VALUES ?candidate {
    ex:Section_1042
    ex:Step_1887
    ex:Warning_2110
  }

  ?candidate prod:appliesToProduct ?product .
  OPTIONAL { ?candidate prod:minVersion ?minVersion }
  OPTIONAL { ?candidate prod:maxVersion ?maxVersion }

  ?topic dita:containsElement ?candidate .
  ?publication pub:includesTopic ?topic ;
               pub:lifecycleStatus pub:Active .

  FILTER(?product = ex:ProductModelX)
  FILTER(!BOUND(?minVersion) || ?minVersion <= "4.2"^^xsd:decimal)
  FILTER(!BOUND(?maxVersion) || ?maxVersion >= "4.2"^^xsd:decimal)
}
```

A.6 Find warnings and prerequisites connected to an affected procedure

Expand beyond the vector-matched procedure to structurally connected elements that may also need review.

```
SELECT DISTINCT ?procedure ?relatedElement ?relationship
WHERE {
  VALUES ?procedure { ex:Procedure_Reset_Controller }

  {
    ?procedure dita:hasWarning ?relatedElement .
    BIND("warning" AS ?relationship)
  }
  UNION
  {
    ?procedure dita:hasPrerequisite ?relatedElement .
    BIND("prerequisite" AS ?relationship)
  }
  UNION
  {
    ?procedure dita:hasResult ?relatedElement .
    BIND("result" AS ?relationship)
  }
}
```

A.7 Detect products that use a changed component but lack documentation

Identify documentation gaps that vector search cannot find because the expected content does not yet exist.

```
SELECT ?product
WHERE {
  ?product a prod:Product ;
           prod:usesComponent ex:ControllerX ;
           prod:lifecycleStatus prod:Active .

  FILTER NOT EXISTS {
    ?topic a dita:Topic ;
           prod:appliesToProduct ?product ;
           prod:documentsComponent ex:ControllerX .
    ?publication pub:includesTopic ?topic ;
                 pub:lifecycleStatus pub:Active .
  }
}
```

A.8 Build an explainable evidence path for an impact report

Return the relationship chain that explains why a source element was selected for review.

```
SELECT ?change ?component ?procedure ?element ?topic ?publication
WHERE {
  VALUES ?change { ex:ChangeRequest_4821 }

  ?change prod:affectsComponent ?component .
  ?procedure prod:usesComponent ?component .
  ?procedure dita:containsElement ?element .
  ?topic dita:containsElement ?element .
  ?publication pub:includesTopic ?topic ;
               pub:lifecycleStatus pub:Active .
}
```

A.9 Retrieve classifications suitable for writing back to DITA

Select approved, high-confidence graph classifications for controlled persistence back into DITA metadata.

```
SELECT ?topic ?classification ?scheme ?confidence ?source
WHERE {
  ?topic a dita:Topic ;
         prod:proposedClassification ?assertion .

  ?assertion prod:classification ?classification ;
             prod:classificationScheme ?scheme ;
             prod:confidence ?confidence ;
             prov:wasDerivedFrom ?source ;
             prod:reviewStatus prod:Approved .

  FILTER(?confidence >= 0.90)
}
```

Appendix B – Illustrative SHACL Validation Shapes

The following shapes illustrate how a graph can validate documentation and workflow rules. They are examples rather than production-ready shapes. Property paths and severities should be adapted to local ontology design and governance requirements.

B.1 Every active procedure must have product applicability

```
ex:ActiveProcedureProductShape
    a sh:NodeShape ;
    sh:targetClass dita:Procedure ;
    sh:sparql [
        a sh:SPARQLConstraint ;
        sh:message "An active procedure must be classified for at least one product." ;
        sh:select """
            SELECT $this
            WHERE {
              $this pub:lifecycleStatus pub:Active .
              FILTER NOT EXISTS { $this prod:appliesToProduct ?product }
            }
        """ ;
    ] .
```

B.2 Safety-critical tasks must include an approved warning

```
ex:SafetyCriticalTaskWarningShape
    a sh:NodeShape ;
    sh:targetClass dita:Task ;
    sh:sparql [
        a sh:SPARQLConstraint ;
        sh:message "A safety-critical task must include an approved warning." ;
        sh:select """
            SELECT $this
            WHERE {
              $this prod:safetyClassification prod:SafetyCritical .
              FILTER NOT EXISTS {
                $this dita:hasWarning ?warning .
                ?warning prod:approvalStatus prod:Approved .
              }
            }
        """ ;
    ] .
```

B.3 Every embedded element must point to an authoritative source element

```
ex:EmbeddingSourceShape
    a sh:NodeShape ;
    sh:targetClass prod:EmbeddingRecord ;
    sh:property [
        sh:path prod:representsSourceElement ;
        sh:minCount 1 ;
        sh:maxCount 1 ;
        sh:class dita:Element ;
        sh:message "Each embedding record must identify exactly one authoritative DITA source element."
;
    ] ;
    sh:property [
        sh:path prov:wasDerivedFrom ;
        sh:minCount 1 ;
        sh:message "Each embedding record must include source provenance." ;
    ] .
```

B.4 Automatically generated classifications require provenance and review status

```
ex:GeneratedClassificationShape
    a sh:NodeShape ;
    sh:targetClass prod:ClassificationAssertion ;
    sh:property [
        sh:path prov:wasDerivedFrom ;
        sh:minCount 1 ;
        sh:message "A generated classification must identify its source evidence." ;
    ] ;
    sh:property [
        sh:path prod:reviewStatus ;
        sh:minCount 1 ;
        sh:in (prod:Proposed prod:Approved prod:Rejected) ;
        sh:message "A generated classification must have a governed review status." ;
    ] ;
    sh:property [
        sh:path prod:confidence ;
        sh:minCount 1 ;
        sh:datatype xsd:decimal ;
        sh:minInclusive 0.0 ;
        sh:maxInclusive 1.0 ;
    ] .
```

B.5 Active publications must not include obsolete topics

```
ex:ActivePublicationObsoleteTopicShape
    a sh:NodeShape ;
    sh:targetClass pub:Publication ;
    sh:sparql [
        a sh:SPARQLConstraint ;
        sh:message "An active publication must not include an obsolete topic." ;
        sh:select """
            SELECT $this ?topic
            WHERE {
              $this pub:lifecycleStatus pub:Active ;
                    pub:includesTopic ?topic .
              ?topic pub:lifecycleStatus pub:Obsolete .
            }
        """ ;
    ] .
```

B.6 Product-version applicability must not conflict with publication scope

```
ex:PublicationVersionCompatibilityShape
    a sh:NodeShape ;
    sh:targetClass pub:Publication ;
    sh:sparql [
        a sh:SPARQLConstraint ;
        sh:message "A topic version range conflicts with the publication version." ;
        sh:select """
            SELECT $this ?topic
            WHERE {
              $this pub:productVersion ?pubVersion ;
                    pub:includesTopic ?topic .
              OPTIONAL { ?topic prod:minVersion ?minVersion }
              OPTIONAL { ?topic prod:maxVersion ?maxVersion }
              FILTER((BOUND(?minVersion) && ?pubVersion < ?minVersion) ||
                     (BOUND(?maxVersion) && ?pubVersion > ?maxVersion))
            }
        """ ;
    ] .
```

Appendix C – Suggested Operational Pattern

1. Extract structured change evidence from product-change inputs.
2. Use the vector index to retrieve semantically related DITA topics and elements.
3. Use SPARQL to verify product, version, publication, audience, provenance, and lifecycle context.
4. Use graph traversal to expand the impact set through reuse, prerequisites, warnings, dependencies, and external product relationships.
5. Run SHACL validation before allowing content into the automated editing stage.
6. Retrieve the authoritative source fragments from the cCMS using persistent DITA identifiers.
7. Send only the bounded fragments to the language model for revision.
8. Validate the revised DITA with XML-native tools and rerun graph validation where applicable.
9. Produce an impact and provenance report for review and audit.
10. Write approved enrichment metadata back to DITA when persistence and portability are desired.

Implementation note

The examples assume that the graph is synchronized with the authoritative DITA source. The cCMS remains the system of record. The graph provides a governed semantic and reasoning layer, while XPath, XQuery, XML Schema, and Schematron remain valuable for source-level processing and validation.

Appendix D – When the Source Is Markdown, AsciiDoc, or reStructuredText

This appendix explains how the architectural conclusion changes when the authoritative source is not DITA XML but instead consists of Markdown, AsciiDoc, reStructuredText, or a mixture of formats. The central conclusion remains qualified: a knowledge graph is not the only technically possible solution, but it becomes substantially more valuable as the source formats provide less consistent semantic structure and relationship intelligence.

D.1 The qualified answer

The graph would not be strictly required merely because the source is not DITA. Markdown, AsciiDoc, and reStructuredText can all be parsed into document trees, and each can support identifiers, metadata, links, includes, and conditional processing to varying degrees. A system could therefore combine format-specific parsers, persistent identifiers, dependency indexes, metadata stores, a source-addressable vector index, and deterministic application logic without using RDF.

However, DITA already supplies a rich and comparatively standardized information model. It explicitly distinguishes tasks, concepts, reference information, steps, prerequisites, results, warnings, reuse mechanisms, applicability, maps, and publication structures. With Markdown, AsciiDoc, or reStructuredText, many of those semantics must be created through local conventions, extensions, front matter, directives, attributes, or external processing logic.

As a result, the graph is much more likely to become the practical semantic and relationship layer for a non-DITA or mixed-format corpus.

D.2 What the source formats provide on their own

These formats are not devoid of structure, but their built-in semantics differ considerably:

Format	Useful native capabilities	Typical limitation for deterministic impact analysis
Markdown	Headings, paragraphs, lists, links, code blocks, and optional front matter	Usually weak or locally defined semantics for procedures, warnings, reuse, applicability, and publication context
AsciiDoc	Rich blocks, attributes, includes, cross-references, conditionals, and extensions	More expressive than basic Markdown, but semantic meaning often depends on organizational conventions and the processing toolchain
reStructuredText	Sections, directives, roles, references, substitutions, and includes	Can be semantically rich, but meaning frequently depends on custom directives, Sphinx domains, and local schemas
DITA	Typed topics and elements, reuse, keys, maps, profiling, applicability, and publication structures	Provides a comparatively standardized semantic and relationship model directly in the source

D.3 The graph as a normalization layer

In a mixed-format environment, the graph can normalize different source constructs into a shared semantic model. For example:

- a DITA step, an AsciiDoc numbered instruction, a reStructuredText procedure directive, and a Markdown list item can all be represented as a common ProcedureStep concept;
- a DITA conref, an AsciiDoc include, a reStructuredText include, and a Markdown transclusion can all be represented as a common reusesContent relationship;
- front-matter values, AsciiDoc attributes, Sphinx metadata, and DITA profiling attributes can all be mapped to common product, version, audience, and lifecycle properties.

The rest of the workflow can then query one model rather than requiring separate traversal and business-rule logic for every source format.

D.4 Direct vector creation is still possible

Embeddings can be created directly from Markdown, AsciiDoc, or reStructuredText. Each vector record can preserve file paths, anchors, headings, line ranges, section hierarchy, front-matter values, attributes, directives, include relationships, and repository or publication context. This can support fast semantic retrieval without first creating a graph.

The harder problem occurs after retrieval. A vector search may find several passages that sound relevant, but the workflow must still determine which passage is authoritative, which product and version it applies to, whether it is obsolete, where it is reused, which publications contain it, and whether related warnings or prerequisites must also be reviewed. The graph can provide this broader context consistently across formats.

D.5 Mixed-format example

Assume a product-change description states: “Controller X now recovers automatically after a communication fault.” A source-derived vector index searches across all formats and returns:

1. an AsciiDoc troubleshooting procedure that says “power-cycle the controller”;

2. a Markdown support article that says “restart the device”;

3. a reStructuredText service guide that says “disconnect and reconnect power”;

4. a retired Markdown article for an older controller that uses similar wording;

5. a DITA warning linked to the old manual recovery procedure.

The vector representation has done its job: it found passages that are semantically related to the change. The graph can then determine that the first three results concern Controller X, the fourth concerns Controller W and is obsolete, the AsciiDoc procedure is included in four publications, the reStructuredText guide applies only to firmware versions earlier than 4.2, and the DITA warning accompanies the old manual recovery action.

The vector model finds what sounds relevant. The graph establishes what is actually affected and what action should follow.

D.6 Could the same result be achieved without RDF?

Yes. An organization could construct a relational dependency database, a search index with extensive metadata, a custom document registry, format-specific parsing services, a rules engine, and application code that joins those systems. That architecture could reproduce many graph capabilities.

At sufficient scale, however, the organization would be implementing graph-like behavior indirectly. The practical decision is whether it is easier to maintain relationships through custom tables, indexes, parsers, and code, or to represent those relationships explicitly in a governed graph model.

D.7 When the graph becomes especially compelling

A graph becomes increasingly valuable when one or more of the following conditions apply:

- multiple source formats must be normalized into one information model;
- relationships cross repositories, authoring systems, or enterprise platforms;
- source metadata is incomplete, inconsistent, or locally defined;
- content must be connected to products, components, software releases, audiences, regulations, or service processes;
- the workflow must infer additional impact beyond semantic similarity;
- validation rules must operate across both content and external enterprise data;
- the same semantic model will support update automation, support retrieval, dynamic delivery, compliance, analytics, and other applications.

D.8 Architecture guidance by corpus type

Corpus type	**Architectural implication**
Pure, well-governed DITA corpus	Graph may be optional for the immediate net-update retrieval problem because DITA, XPath/XQuery, and a source-addressable vector index can already provide much of the needed structure and traversal.
Single, carefully governed AsciiDoc or reStructuredText corpus	Graph is useful but may remain optional when identifiers, metadata, includes, conditions, and publication conventions are consistently enforced.
Basic Markdown corpus	Graph becomes more valuable because important semantics and dependencies are usually external to the format or encoded only through local conventions.
Mixed-format enterprise corpus	Graph is often the most robust practical choice because it provides a common semantic, relationship, inference, and validation layer across otherwise incompatible source models.

D.9 Final conclusion

Non-DITA content does not automatically require a knowledge graph. It does require an explicit, governed semantic and relationship layer when the workflow must support deterministic, source-addressable impact analysis at enterprise scale.

A graph is usually the most robust and extensible implementation of that layer when formats are mixed, semantics are inconsistent, and relationships extend beyond the documentation repositories. The less semantic structure and relationship intelligence the source formats provide consistently, the more valuable – and eventually operationally necessary – the graph becomes.