Document Object Model Graph RAG

A semantic, content-first, and knowledge-management architecture for neuro-symbolic RAG

September 15, 2024

Michael Iantosca
Senior Director of Knowledge Platforms and Engineering
Avalara Inc.

Helmut Nagy
Chief Product Officer
Semantic Web Company GmbH

William Sandri
Data & Knowledge Engineer
Semantic Web Company GmbH

Synopsis

The DOM Graph RAG project represents a new, novel approach and a significant advancement in artificial intelligence (AI), particularly in Retrieval-Augmented Generation (RAG). The project not only addresses the limitations of current vector-based RAG models but also introduces an approach that significantly improves the accuracy, reliability, and contextual understanding of AI-generated content. This reliability and trustworthiness, especially in high-stakes environments, will instill confidence in the capabilities of this new model.

Traditional vector-based RAG models rely heavily on stochastic processes, making them prone to inaccuracies, hallucinations, and context loss. These limitations are exacerbated in industries where precision, reliability, and trustworthiness are critical. Vector-based models, which typically break content into arbitrary chunks for retrieval, often fail to maintain the integrity of the original content structure, leading to inefficient and unreliable results.

Graph-based RAG models, on the other hand, offer a more structured approach by leveraging knowledge graphs and structured intelligence. These graphs allow for richer contextual understanding, multi-hop reasoning, and explicit entity recognition, overcoming many of the issues vector-only systems face. However, constructing and maintaining knowledge graphs at scale has traditionally been a resource-intensive task, a significant challenge that this model solves.

The DOM Graph RAG model employs a Document Object Model (DOM) to structure content and uses knowledge graphs to map relationships between topics, elements, and metadata. This graph-driven approach ensures content is not arbitrarily chunked but retains its original structure and context, which is crucial for accurate content retrieval and response generation. The project uses the Darwin Information Typing Architecture (DITA) schema to automate the construction of these graphs, further simplifying the process of managing complex content.

In addition to addressing issues of content integrity and accuracy, the DOM Graph RAG model also incorporates neuro-symbolic reasoning, which combines the pattern recognition capabilities of neural networks with the logical reasoning and fact-checking abilities of symbolic models. This hybrid approach enables AI systems to generate responses based on retrieved content and reason over the data to ensure logical consistency and factual accuracy.

A significant advantage of the DOM Graph RAG model is its ability to manage dynamic and evolving content, which is challenging for vector-based systems. The DOM-based knowledge graph allows continuous updates to quickly changing content, maintaining its relevance and accuracy over time.

The sample chatbot demonstrates the practical applications of the DOM Graph RAG model. This chatbot uses the enriched DOM graph to enhance user queries, providing more accurate, explainable, and context-aware answers. The DOM Graph RAG model reduces dependency on large language models (LLMs), lowering computational costs while improving performance.

Overall, the DOM Graph RAG offers a robust, scalable, and cost-effective solution for industries requiring high levels of accuracy and trustworthiness. Its hybrid approach, combining graph and semantic retrieval with neuro-symbolic reasoning, significantly improves over traditional RAG models.

Document Object Model Graph RAG

A semantic content architecture for neuro-symbolic RAG

Michael Iantosca
Senior Director of Knowledge Platforms and Engineering
Avalara Inc.

Helmut Nagy
Chief Product Officer
Semantic Web Company GmbH

William Sandri
Data & Knowledge Engineer
Semantic Web Company GmbH

The advent of OpenAI's ChatGPT and other large language models (LLMs) marked a pivotal moment, catapulting generative AI into the public consciousness. This sudden rise sparked both excitement and fear, leading to a frenzy of investment that, while initially overhyped, has shifted towards more practical applications of generative artificial intelligence technology.

Despite this, corporate leaders were alarmed. Many hastily commissioned generative artificial intelligence projects, pressuring engineering teams to produce rapid results out of fear of disruption. Such pressure led to a flurry of "solutions looking for problems" as companies scrambled to adopt the technology without fully understanding its implications.

Generative artificial intelligence offers vast potential, but the quickest application for most teams was developing generative question-and-answer chatbots. Engineering teams embarked on creating domain-specific chatbot applications using their company's content. However, they quickly encountered the limitations of LLMs, particularly the issues of hallucinations and inaccuracies. These issues led to the emergence of Retrieval-augmented generative AI.

Retrieval-augmented generation

Retrieval-augmented generation (RAG) is a concept in natural language processing (NLP). A RAG model first retrieves relevant content or data from an external source and then uses that information to generate a response or complete a task. The retrieval step helps improve the quality and accuracy of the generated output by grounding responses in specific, relevant information.

A skilled Python programmer can build a basic RAG model prototype using an LLM API in a few hours or days. A full production model can take several months to develop when factoring in infrastructure, guardrails, and tuning.

The standard process involves breaking content into small pieces, storing them in a vector database, and processing a user prompt to search the database for relevant chunks. These chunks are then fed to the LLM to generate output. Many teams implemented this approach shortly after the OpenAI APIs became available, well before the term "RAG" was coined.

While RAG models improved over standalone LLMs, most quickly realized that vector databases feeding a vector-based LLM had significant limitations. By their nature, LLMs are stochastic systems based on randomness or probability, and this inherent characteristic undermines their accuracy, precision, reliability, and explainability. We explained these limitations at conferences long ago, demonstrating how vector math and cosine similarity work and their vulnerabilities.

Companies experimenting with generative artificial intelligence soon recognized the need for governance and control. Usually, the responsibility for developing AI solutions fell to engineering teams, who approached the task almost entirely as a coding problem. However, this perspective overlooked the critical roles of content design and knowledge management. The most advanced organizations now understand that successful AI development requires a balanced combination of engineering, content, and knowledge management.

In the rush to implement RAG, engineering teams often scraped content from internal sources without considering the content's quality, structure, or mission-critical management requirements. Technical documentation became a prime target. Content teams, who understood the intricacies of the content supply chain, were sidelined and often ignored or dismissed as engineering teams took the lead. This behavior leads to subpar results.

As teams evaluated these RAG solutions, it became evident that better models and content were needed to achieve greater accuracy, reliability, and explainability, which are required in industries with significant legal liabilities or highly regulated sectors. While some engineering teams began to understand the value of taxonomies and labeling content, it was clear that these alone were insufficient for the degree of precision needed in critical applications.

The *precision paradox* emerges as AI systems become more accurate. The error tolerance decreases as the system's precision increases, making even minor mistakes more noticeable and impactful. This is particularly crucial as organizations move towards autonomous or semi-autonomous AI agents. The current vector-based RAG models struggle to meet the required levels of accuracy and reliability, necessitating a shift towards more advanced AI models.

Vector RAG versus graph-driven RAG

Vector-based retrieval-augmented generation (RAG) models are AI models that rely on similarity search and predictive algorithms. These models have strengths in these areas but also critical weaknesses, such as inaccuracies and hallucinations, which stem from their inherent randomness and probabilistic nature.

Engineering teams globally are grappling with these issues, attempting to mitigate problems fundamentally tied to vector-centric retrieval limitations. Increasing the parameters of large language models (LLMs) or building ever more complex infrastructures around vector RAG models cannot alter the mathematical reality inherent in these systems.

Critical Challenges with Vector-Only RAG Models:

- **Lack of neuro-symbolic reasoning and fact-checking**: LLMs and vector databases are limited in their ability to infer, reason, or fact-check independently. While they may appear capable of these tasks, the reality is different. Efforts to integrate these abilities into vector-

based models have been inadequate. To achieve robust reasoning and independent validation, an external source of curated knowledge, such as a knowledge graph that provides validated facts and an inference engine, is required to query against.

- **Challenges in building and maintaining knowledge graphs**: While knowledge graphs offer a powerful alternative to vector databases for retrieval, constructing and maintaining them at scale for content, which may include thousands to millions of nodes, is daunting. Manual graph construction has been impractical for most organizations, especially those managing large volumes of content. There's a pressing need for innovative, reliable methods to automate the creation and updating of curated and validated knowledge graphs, especially for complex and specialized knowledge domains. The DOM Graph RAG model solves this vexing problem elegantly.
- **Change management and content currency**: Content is dynamic and constantly evolving, particularly in business and technical domains. Current vector-based models often fail to consider or effectively manage frequent and voluminous content updates, retirement, versioning, entitlement, reuse, audience, personalization, reuse, and more. The inability to manage frequent changes creates significant risks to the business as vector stores struggle to manage expired or outdated content. In contrast, graph-based models can embed metadata and state information with the graph, enabling sophisticated content management and automation of change management processes.
- **Loss of context**: Vector-based models often require content to be divided into chunks, typically in arbitrary sizes (for example, 4K tokens). This approach can fragment the original content, making it difficult to reassemble content correctly and meaningfully in its original context during retrieval. Organizations with well-structured content that content developers have meticulously chunked over many years into component/topic content models find it nonsensical and unnecessary to chunk any further to attempt (poorly) to reassemble the content along with its context, losing vital context and relationships.
- **Lack of explicit entity recognition**: Vector-based RAG models primarily use dense embeddings to represent queries and documents in a high-dimensional vector space. This approach works well for semantic similarity but lacks the explicit ability to recognize or handle named entities with the same granularity as graph-based approaches. The model may identify similar topics or concepts based on their proximity in the vector space, but it won't explicitly "know" or "link" entities like a graph does.
- **Limited Multi-hop Retrieval**: Vector-based RAG typically retrieves documents or knowledge in a single hop—finding the nearest neighbors to a query in embedding space. It does not inherently support multi-hop reasoning. While some vector-based systems might attempt to perform multi-step reasoning, this process is often less structured and less reliable than graph-based methods.
- **Limited recommendations**: Vector-based retrieval models cannot capture complex relationships between entities. While vector embeddings represent entities, they miss the rich contextual connections. Graph-based RAG models excel at generating recommendations by integrating knowledge graphs with neural networks. In this approach,

entities and their relationships are represented as nodes and edges in a graph, which are then converted into vector representations (graph embeddings). The neural network can retrieve relevant information based on the query, and the knowledge graph provides rich contextual relationships, leading to more accurate and context-aware recommendations. Integrating neural retrieval and symbolic graph reasoning enhances the model's relevance, personalization, and logical consistency.

- **Lack of localization management**: How would a vector-only RAG model manage multiple national language variants (NLVs)? Would such a model translate the content and put the NLVs in separate vector DBs for retrieval? That's one messy option. Use an on-demand MT service? That sounds like a nonstarter precision and performance-wise, or should we use the LLM to translate on the fly? Possibly, if machine translation accuracy is acceptable (there's a reason why a massive language service provider (LSP) industry exists). With DOM Graph RAG, we can translate the content into as many NLVs as we want, using any method we want, and carry the language identifier in the metadata of each document object graph node. Then, the query can pick which NLVs to retrieve. If NLV retrieval is complex enough using a vector-based RAG model, imagine dealing with *versioned* NLV content.
- **Overfitting**: Vector-based models are prone to overfitting. This occurs when teams excessively tune models to compensate for the shortcomings of vector-only retrieval. Overfitting leads to excellent performance on tested queries but poor generalization for new, unanticipated questions or tasks.
- **Efficiency and performance**: Graph-based RAG models can offer greater efficiency than vector-based models, particularly in structured retrieval, contextual reasoning, scalability, and interpretability. While vector databases perform better in multidimensional spaces, graph databases excel in handling relationship-based queries. Hybrid models that combine these strengths can optimize performance and scalability.
- **Cost-effectiveness**: Graph-based RAG models can significantly reduce costs compared to vector-based models. This is due to the more efficient retrieval process, which reduces dependencies on the LLM for handling voluminous transactions required with fine-tuning and embeddings. The graph-based approach reduces computation time and resource consumption by providing more concise and relevant inputs to the LLM, leading to overall cost savings.

While vector-based RAG models have their place, they face significant challenges that graph-driven RAG models address through better reasoning, contextual understanding, efficiency, and cost-effectiveness. Graph-based approaches offer a more robust framework for complex, dynamic content environments.

Neuro-symbolic RAG: Enhancing AI with integrated reasoning and knowledge

The next advancement in retrieval-augmented generation (RAG) is neuro-symbolic RAG, which integrates neural networks with symbolic reasoning, inferencing, and fact-checking. This approach uses the strengths of both neural and symbolic models, allowing AI systems to learn from data and

reason over it using predefined rules and logic. Knowledge graph-based RAG is gaining popularity because it effectively addresses the limitations of vector-based retrieval models.

However, developing and maintaining high-quality knowledge graphs is often human resource-intensive and prohibitive. Although LLMs can assist in constructing these graphs, relying solely on LLMs is risky. A well-designed and managed knowledge model, grounded in a robust ontology, remains under the organization's control and governance and becomes a company's most valuable intellectual property and competitive advantage.

With its GraphRAG and similar models, companies like Microsoft have explored automating knowledge graph generation from text using LLMs. While these automatically generated graphs can be helpful, especially for monolithic unstructured content sources, they often need more accuracy to avoid pitfalls like the precision paradox. A more effective approach combines traditional and hybrid models, emphasizing deliberate and strategic development.

The cornerstone of a reliable knowledge graph is an intentionally constructed and human-validated ontology. An ontology defines the concepts and relationships within a specific domain. Once developed, the ontology can be loaded into a graph database, where physical or logical content objects are mapped against it to create a comprehensive knowledge graph. This graph can then be queried to retrieve relevant content objects and feed them into the LLM, ensuring that outputs are accurate and trustworthy. Models like Microsoft GraphRAG, which attempt to automatically generate graphs from text or models that automatically generate ontologies from text, often result in weak and error-prone substitutes for carefully constructed and validated schema and domain-specific knowledge models.

AI models must evolve beyond traditional RAG approaches to achieve the high precision and reliability required for critical applications. The Document Object Model Graph RAG (DOM Graph RAG) represents a novel, sophisticated, more straightforward, and elegant method for integrating componentized content with AI. By harnessing the combined strengths of engineering, content design, and knowledge engineering, DOM Graph RAG delivers more accurate and trustworthy generative AI solutions, setting a new standard in the field.

Knowledge graphs and DITA: Structured content synergy

Knowledge graphs are a form of structured content, meaning they organize and categorize information in a clearly defined way, simplifying processing, retrieval, and analysis. In a knowledge graph, data is represented in a structured format where entities (nodes) and their relationships (edges) are clearly defined, often following a specific blueprint in the form of an ontology or schema. This structured approach allows for efficient querying, reasoning, and data integration within and between domains, making knowledge graphs a powerful tool for organizing and understanding complex information.

DITA (Darwin Information Typing Architecture) is an XML architecture and an open OASIS industry standard for structured documents. DITA's well-defined schema makes it easy to convert into the Ontology Web Language (OWL) format. OWL represents complex knowledge about entities, groups of entities, and their interrelations, making it a crucial component of semantic solutions.

OWL ontologies use the Resource Description Framework (RDF), representing information as triples (subject-predicate-object). RDF provides the basic structure for representing data in a graph. At the same time, OWL extends RDF's capabilities by allowing for the definition of more complex relationships and constraints, such as classes, properties, hierarchies, and rules. This makes OWL ideal for creating ontologies that formally represent knowledge within a domain. RDF graphs are exceptionally well suited over other technologies, such as property-based graphs representing hierarchical content and linked data.

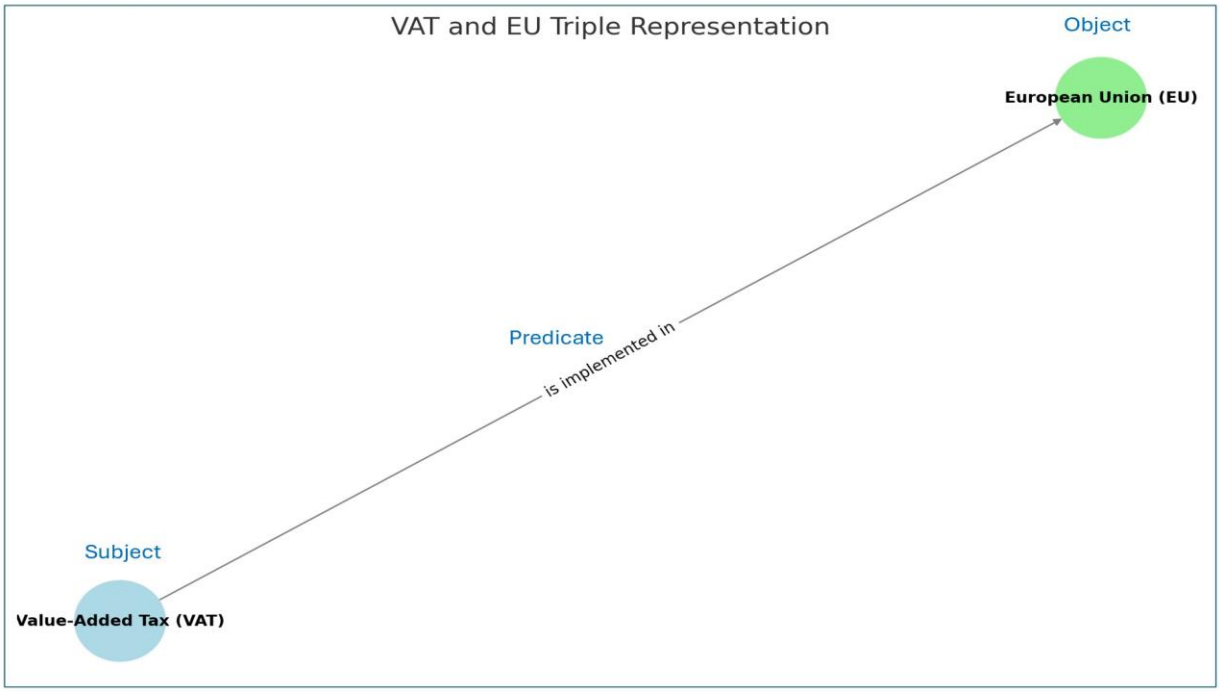

Figure 1. A visual representation of an RDF triple

In Figure 1, a triple represents a relationship between three entities: a subject, predicate, and object. For example, relating VAT tax to Europe:

Triple:

- **Subject**: Value-Added Tax (VAT)
- **Predicate**: is implemented in
- **Object**: European Union (EU)

This expresses the fact that VAT is a tax system that is widely implemented across the European Union. Here's a breakdown:

- **Subject**: "Value-Added Tax (VAT)" — the tax being discussed.
- **Predicate**: "is implemented in" — the relationship between VAT and the entity.
- **Object**: "European Union (EU)" — the geographic or political entity where VAT is applied.

This structure could easily be adapted for other relationships or countries with VAT tax systems.

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
         xmlns:owl="http://www.w3.org/2002/07/owl#">

  <!-- Declaration of Classes -->
  <owl:Class rdf:about="#Tax"/>
  <owl:Class rdf:about="#Region"/>

  <!-- Declaration of Individuals -->
  <owl:NamedIndividual rdf:about="#ValueAddedTax">
    <rdf:type rdf:resource="#Tax"/>
  </owl:NamedIndividual>

  <owl:NamedIndividual rdf:about="#EuropeanUnion">
    <rdf:type rdf:resource="#Region"/>
  </owl:NamedIndividual>

  <!-- Declaration of Object Property -->
  <owl:ObjectProperty rdf:about="#isImplementedIn">
    <rdfs:domain rdf:resource="#Tax"/>
    <rdfs:range rdf:resource="#Region"/>
  </owl:ObjectProperty>

  <!-- Assertion of Property -->
  <rdf:Description rdf:about="#ValueAddedTax">
    <isImplementedIn rdf:resource="#EuropeanUnion"/>
  </rdf:Description>

</rdf:RDF>
```

Figure 2. Relationships between entities using classes, individuals, and properties in OWL

In Figure 2, The OWL representation formalizes the relationship between VAT and the European Union in a machine-readable format.

- **Classes**: Tax and Region are declared as classes. VAT is a type of tax, and the European Union is a type of region.
- **Individuals**: ValueAddedTax belongs to the Tax class, and EuropeanUnion is an individual of the Region class.
- **Object Property**: isImplementedIn is the predicate that connects a tax (domain) to a region (range).
- **Assertion**: It declares that the individual ValueAddedTax is implemented in the individual European Union.

Like other XML languages, DITA is based on a Document Object Model (DOM). The DOM is a programming interface representing a document's structure as a tree of objects, with each node representing a part of the document. This hierarchical tree structure allows scripting and query languages to dynamically interact with and manipulate the document's content, structure, and styling. DITA brings several principles of object orientation to content, including containment and inheritance.

In a DOM, the topic collection file, a DITAMap, is the tree's root, branching into various subcomponents, including sub-maps, topics, elements, and sub-elements. With a DOM, everything in a document instance is a container or sub-container, and the containers are self-describing with meaningful tags. Every XML element can carry metadata about its container. A graph represents elements as nodes, including map nodes, topic nodes, text nodes (containing the text within these elements), and attribute nodes (representing the attributes of elements). This structure is fundamental to how DITA documents are organized and processed. See https://groups.oasis-open.org/communities/tc-community-home2?CommunityKey=c2c11e3a-c9cd-43d9-840b-018dc7cd5db9 for more information about the open DITA standard.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE map PUBLIC "-//OASIS//DTD DITA Map//EN" "map.dtd">
<map xmlns:ditaarch="http://dita.oasis-open.org/architecture/2005/">

    <!-- Title of the map -->
    <title>Tax System Documentation</title>

    <!-- Topic references -->
    <topicref href="vat-overview.dita" navtitle="Overview of VAT"/>
    <topicref href="vat-eu-implementation.dita" navtitle="VAT Implementation in Europe"/>
    <topicref href="sales-tax-us.dita" navtitle="Sales Tax in the United States"/>

    <!-- Nested Topic Reference for a sub-section -->
    <topicref href="vat-us-comparison.dita" navtitle="Comparison between VAT and Sales Tax
        <topicref href="vat-advantages.dita" navtitle="Advantages of VAT"/>
        <topicref href="sales-tax-advantages.dita" navtitle="Advantages of Sales Tax"/>
    </topicref>

</map>
```

Figure 3. Example of a top-level DITAMap that references topics or sub-maps

Knowledge graphs and XML are structured models with a powerful and natural affinity. DITA's inherent structure, implied and explicit relationships, content type classification, and extensibility make it well-suited to be represented, explored, and mined as a knowledge graph. It also opens up a world of advanced content management and retrieval possibilities.

Using schemas as the basis to automatically map content into a knowledge graph is not new; it has been common practice in representing relational databases (RDBs) that use schemas. AI engineers, who have long recognized the efficiency of working with structured content over unstructured content, have found this approach far simpler and more effective.

Representing DITA as a knowledge graph is also not a novel idea. In 2013, Colin Maudry initiated an open-source project called *The DITA RDF Project*, which used the DITA schema to develop an RDF knowledge graph of the DITA model. The project aimed to provide detailed metrics to authors and streamline processes such as content translation and document status consistency checks. Maudry's work, presented at the 10th Summer School on Ontology Engineering and the Semantic Web and presented at the 2013 DITA Europe conference, remains available online and on SourceForge. His project was rediscovered years later during explorations of using DITA's structure as a basis for a graph-based RAG model.

In summary, knowledge graphs and DITA offer structured approaches to content organization, making them highly compatible. DITA's robust structure lends itself well to knowledge graph representation, enabling advanced content management and retrieval capabilities that use both technologies' strengths.

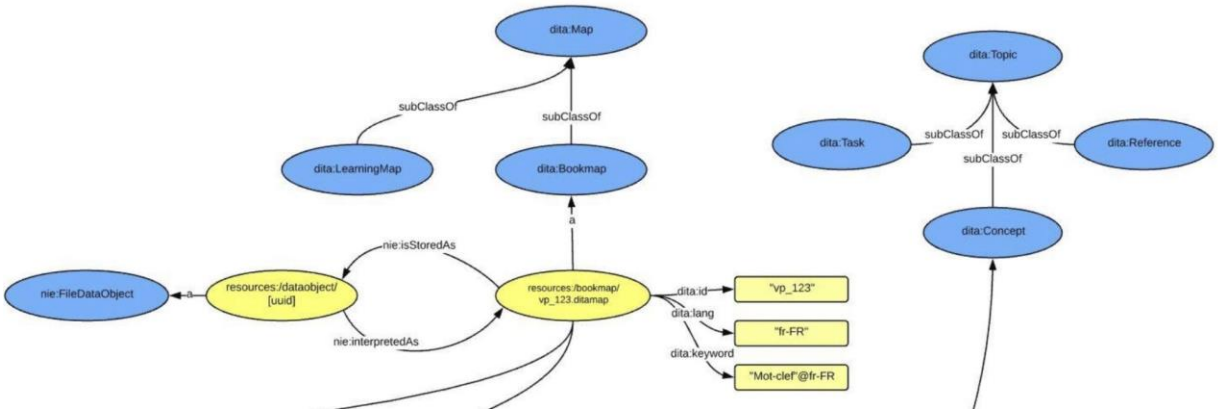

Figure 4. A visual representation of an RDF model of DITA

Chunked versus componentized content in RAG models

In most vector-centric RAG models, content is preprocessed and divided into smaller units, or "chunks," before being stored in a vector database. When a user enters a prompt, the system searches the vector database to retrieve the most relevant chunks combined and passed to the LLM to generate a response. This chunking serves two primary purposes: to ensure efficient storage and retrieval of relevant information.

First, chunking allows the embedding model to effectively capture the essence of the text within its short context windows. By breaking the content into smaller chunks, the vector database can more

accurately represent and retrieve the semantic meaning of the text. Second, chunking addresses the limited input context length of transformer-based Large Language Models (LLMs). For instance, OpenAI's GPT-3.5 model has a token limit of 4K tokens. Therefore, it is crucial to provide only the most relevant parts of the content and minimize unnecessary information.

However, far too many AI/ML engineering teams make a deeply flawed assumption from the outset that all documents are unstructured blobs. Then, they build convoluted RAG models based on that fundamental and erroneous assumption. Over the past two decades, thousands of organizations have transitioned from book/chapter-oriented content models to component-based topic models. They have adopted reusable, topic-oriented information architectures, often using XML formats like DITA to facilitate this shift. Other formats used to componentize content include Markdown, AsciiDoc, ReStructuredText, and custom SGML/XML schema. The DOM Graph RAG model also supports these topic/component-based document formats.

In many content environments today, document source content is already properly segmented into components with the correct context intact. When using DITA Maps as topic collectors, the Document Object Model and other relational constructs declare the relationships between topics. Therefore, it is counterproductive and entirely counterintuitive to break down these well-structured topics any further into random chunks, discarding the carefully designed context and relationships. Doing so only leads to using an inefficient and error-prone process in a futile attempt to correctly reassemble and restore original context and re-establish contextual relationships using a vector-based retrieval model—a process inherently flawed in all vector-based RAG models, including models based on text-generated knowledge graphs.

A more effective approach is to use the existing topic-component DOM model, allowing it to manage the relationships and context. This approach eliminates the deficiencies and weaknesses associated with content retrieval from vector databases, providing a more straightforward, more accurate, and elegant solution and architecture for content management and retrieval in RAG systems.

Constructing the enriched DOM knowledge graph

The image below shows how to build a high-level DOM-based knowledge graph. A prerequisite is having the DOM ontology in place. The content graph is created in the first step, and domain taxonomies and ontologies can enrich it in the second step.

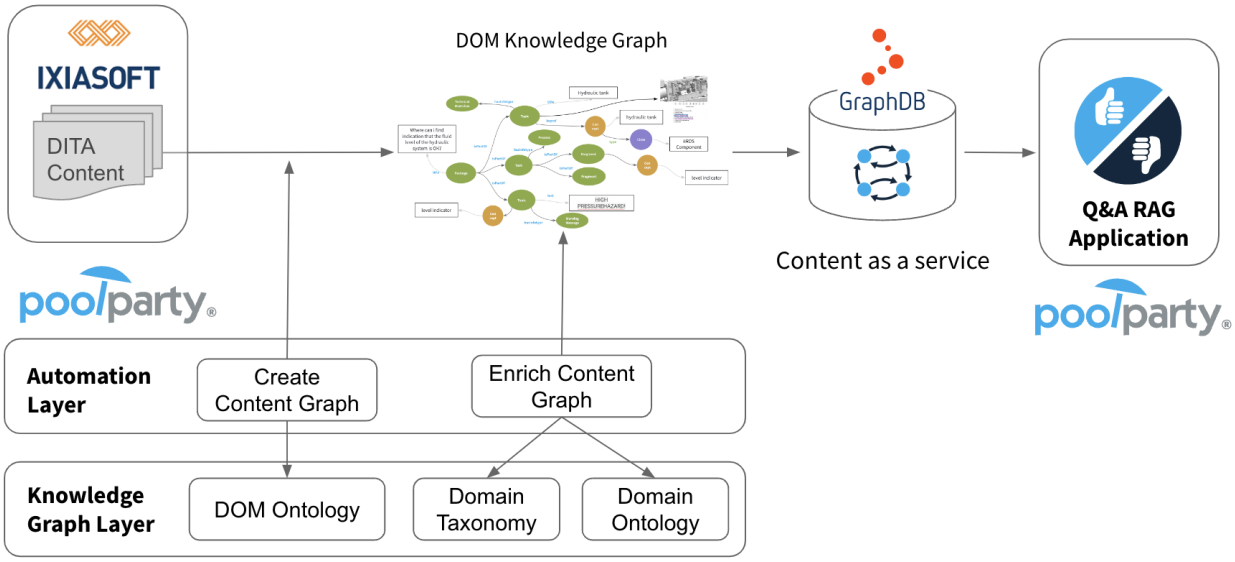

Figure 5. Creating the DOM Graph

Prerequisite: Converting the DITA schema into an ontology

The first step and key to automatically generate a graph using DITA is to convert the DITA schema into an ontology and then load that ontology into a graph database. For this project, we used Ontotext's GraphDB as the graph database. The ontology serves as a schema the model uses to map the DITA (or other non-DITA) source topics and automatically create a knowledge graph. This process also allows updates to the knowledge graph when content is changed.

Our implementation used the open-source DITA ontology created by Colin Maudry. For this project, we used Semantic Web Company's PoolParty Semantic Suite to import and complement the DITA ontology to automate the transformation process.

The DITA ontology represents the content structure for the graph (for example, *Task is a child of DITAMap, Element x is a child of Element y,* and so on). OWL classes, properties, and constraints represent the semantics of DITA concepts and relationships. The ontology can be output as RDF/XML, Turtle, OWL/XML, or JSON/LD and imported into a graph database. The ontology captures the relationships of top-level topic collectors (DITAMaps) and their subtrees of child maps, topics, and elements.

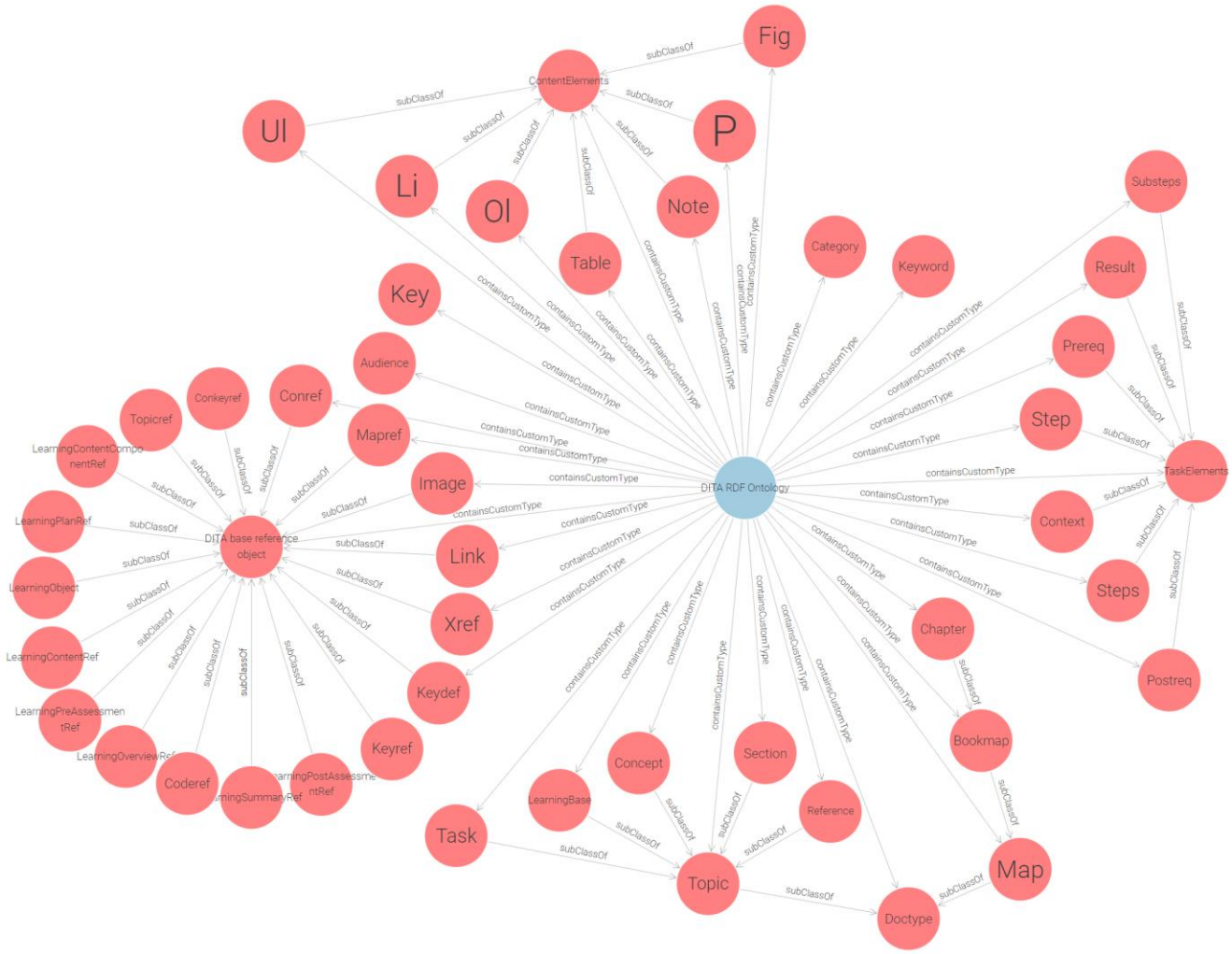

Figure 6. A visual representation of a document object model (DOM) graph

DITA provides a default set of topic types: Topic, Concept, Task, Reference, Troubleshooting, Q&A, and Learning and Training. The DITA topic architecture is also extensible, allowing organizations to create new topic types that are inherited from these basic topic types (a process called *specialization*). DITA specializations can also be added as custom ontology, extending the default DITA ontology. The ontology reflects these topic types and cascades down the DOM tree, representing content elements within topics as graph nodes. The structure looks something like this:

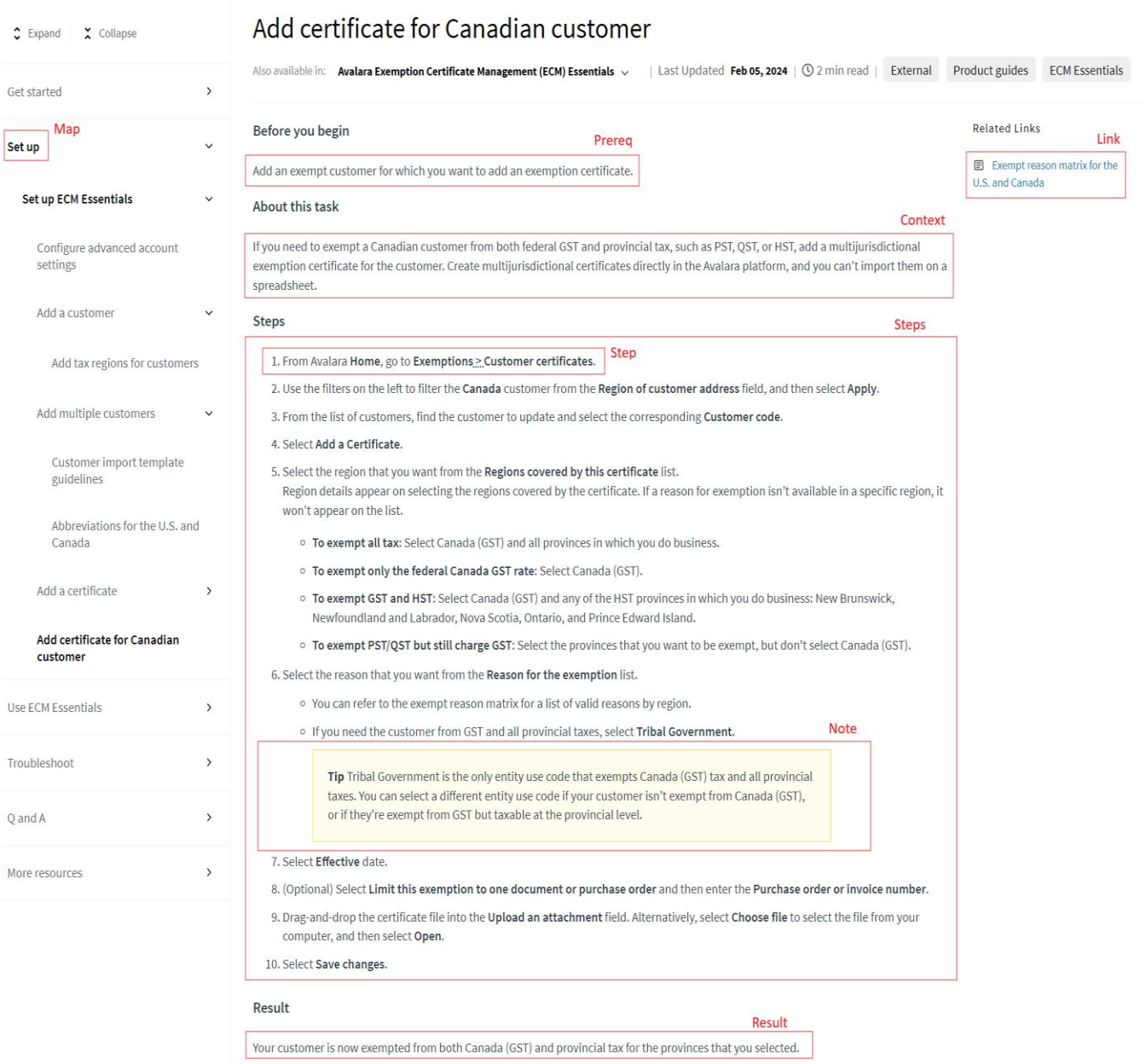

Figure 7. Logical components of a DITA topic

Note: *DITA is not the only structure content schema for creating a DOM graph-mapping ontology. Other structured DOM content models can be used, such as DocBook, S1000D, TEI (Text Encoding Initiative), and custom XML dialects. You can also create your own content model and content ontology. We recommend implementors build on top of widely used standards such as DITA. Other standards, such as iiRDS (International Standard for Intelligent Information Request and Delivery), can be used as a superset of DITA to extend the interchange and interoperability between organizations. In a sense, iiRDS can be an optional top-level ontology (TLO) for structured content interchange.*

Step 1: Automating the content graph construction

The model then uses the ontology to automatically create and maintain a content or DOM graph from the DITA document instances or other content formats. The DITAMap, topics, elements, and child maps are represented as nodes in the DOM graph with a persistent URI linking to the actual content files. As for the ontology, Ontotext's GraphDB is used as a graph database to store the content graph for this project.

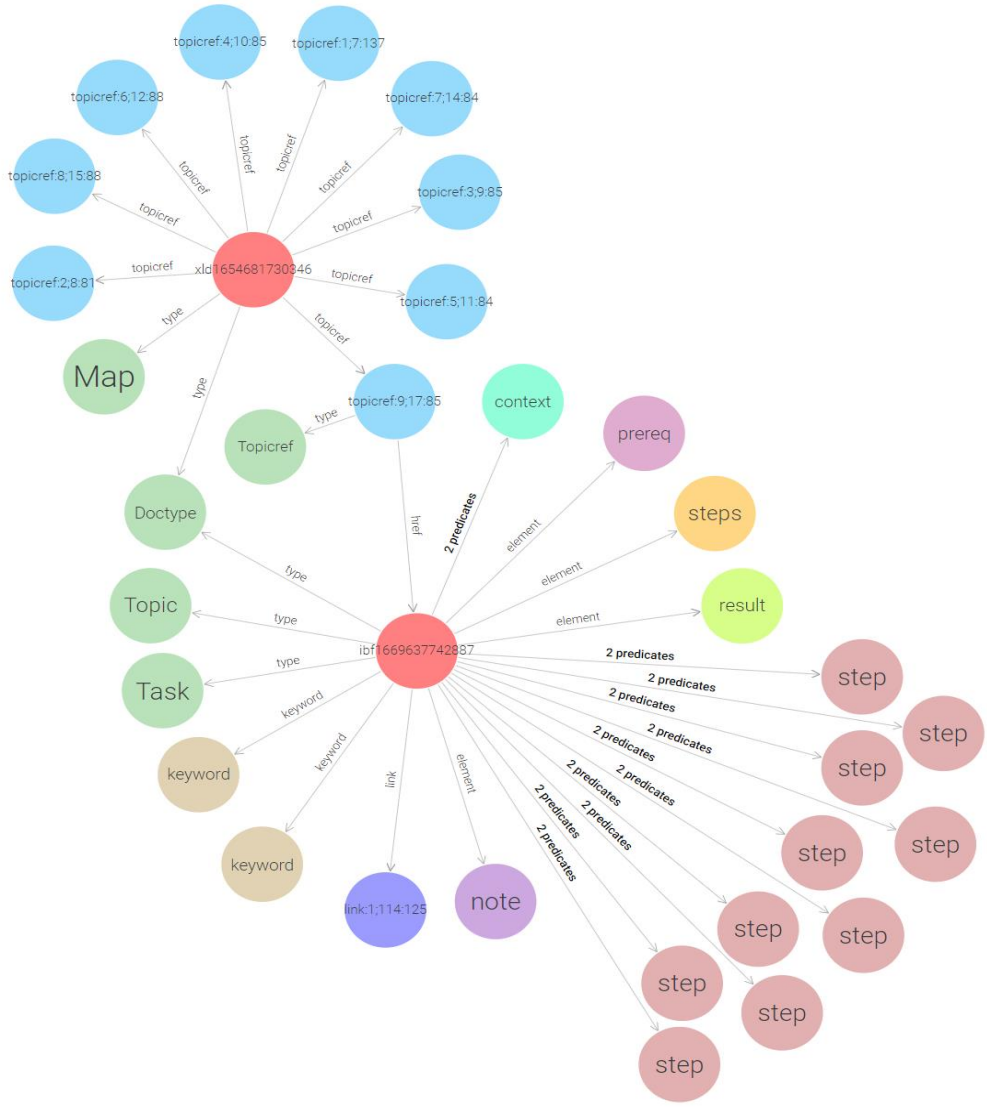

Figure 8. Document component graph nodes

We used the DITA Open Toolkit (available on SourceForge) to automate the transformation of the content instances represented by top-level DITAMaps. The transform converts source topic files to RDF for ingestion into the graph database. When the RDF is imported into the graph database, the graph database automatically builds the graph based on the DITA ontology. The depth of the

resulting DOM tree represented as nodes in the resulting graph can be controlled through the XSLT transformation process. For example, you might optionally choose to include topics encoded in formats like Markdown or AsciiDoc as DITAMap TopicRefs, treating them as whole topics for retrieval while providing more granular subtopic element retrieval for highly structured DITA and other XML-encoded documents.

Likewise, using the same method, you can automate the reingestion of new or updated documents. You can also automate RDF transformation and content updating from a content management system or repository to any degree of frequency.

Metadata associated with maps and topics, whether embedded within them or sourced externally, provides state metadata to automate sophisticated AI content operations and change management that traditional RAG models universally lack. For instance, we can designate content by node as updated, expired, retired, internal use only, unreleased, confidential, versioned, entitled, localized, and more. Proper content management ensures organizations can deliver the right content, to the right person, at the right time, and in the right experience.

If the topics are in DITA format, you can include such metadata directly in the topic or element containers. For non-DITA topics, such as Markdown, ASCIIDoc, and other topical formats, you can apply metadata to the TopicRef elements in the DITAMap. The TopicRef element is itself a container. Most vector-based RAG models lack this crucial content management capability, making them vulnerable to inaccuracies over time as content is rarely static; additions, changes, deletions, and modifications to the content corpora are often constant and voluminous.

Step 2: Enriching the DOM graph

While the DITA structure and intelligent content are beneficial, incorporating Named Entity Recognition (NER) and Named Entity Extraction (NEE) along with classifying maps and topics (and even elements) with domain taxonomies and ontologies further enhances the graph's utility for accuracy, inferencing, and reasoning.

Authors can manually apply taxonomy to the content, assist with recommenders, or completely automate it with autoclassifiers. While our enriched DITA DOM-Graph can be used for retrieval without an inference engine, extending the graph with a domain-specific knowledge model ontology adds powerful inferencing, reasoning, and fact-checking capabilities that LLMs and vector-based RAG models lack.

Step 3: Leveraging the graph database

With the knowledge graph in place, we can now mine the graph for retrieval, interrogate the content corpus, and use SPARQL, a graph query language, to query the knowledge graph. The choice of the graph database is critical for scalability, performance, and inferencing capabilities.

In addition, the graph model is perfect for data integration when you want to go beyond the DITA content and integrate other content sources like e-learning data, product information, and so on. Based on the graph model, it is easy to create a semantic search index or a vector database on top of the knowledge graph and use the most appropriate model for your application's purpose.

This approach allows us to harness the full potential of the DITAMap structure, combining the strengths of XML, taxonomies and ontologies, and graph databases to deliver a more robust and

precise content retrieval and reasoning system than what traditional vector-based RAG retrieval models can offer.

Building a DITA GraphRAG application - A working chatbot

To illustrate the power and viability of the DOM Graph RAG model, we built a Graph RAG application on top of the enriched DOM graph to assess whether this approach can significantly improve the results compared to a traditional RAG approach. The image below outlines the high-level architecture and data flow of this application.

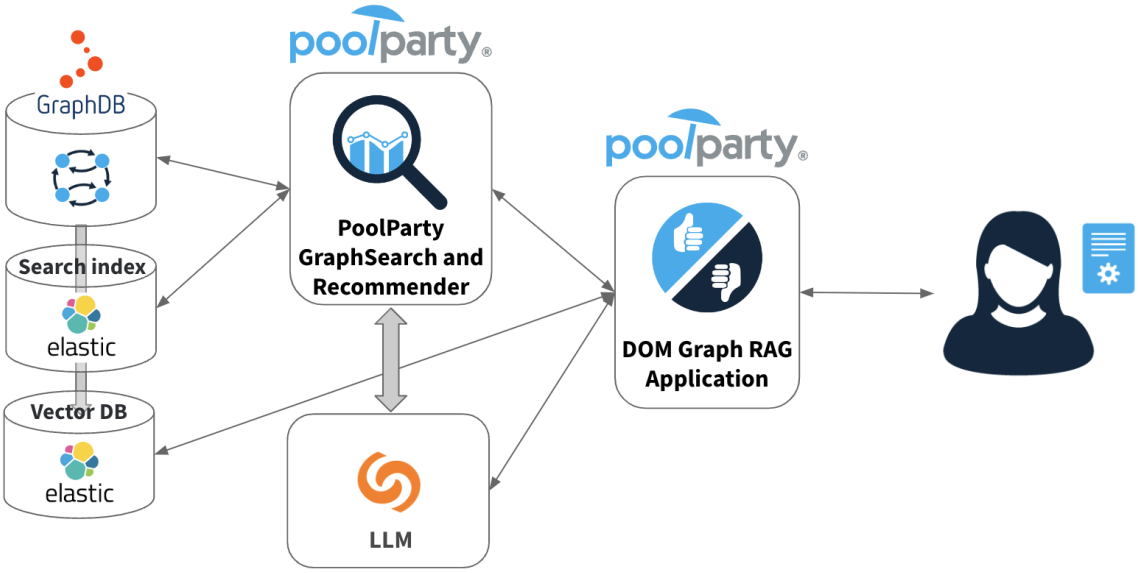

Figure 9. DOM graph reference architecture

Why we included a vector DB

Including a vector database in this graph-driven retrieval model might initially appear confusing and need clarification. We've emphasized that we don't want to break our contextually accurate topics and self-describing elements into smaller fragments arbitrarily only to attempt to piece them back together using unreliable, probabilistic vector retrieval—an approach we've criticized for its lack of precision and context preservation - and we're not.

Since vector databases excel at similarity search, we applied their power and created a hybrid model. The system can manage well-defined and more open-ended or unclear questions by combining a vector database (for flexible, meaning-based semantic search) and a graph database (for precise, structured search). This combination makes the system more accurate and can provide better answers, regardless of how users phrase prompts.

The vector database is populated by converting text chunks extracted from the structured RDF content into vector embeddings. These embeddings allow for semantic similarity searches, linking to the original RDF triples stored in the Ontotext GraphDB. The similarity search relies on only the vector database; the LLM is not involved. The similarity engine receives the user's request and

performs a search and a re-rank of similar content using a local embedding model. From that result, the model collects the DITA document references and uses them to query the graph for the actual content to be sent to the LLM. This hybrid approach ensures that structured (RDF) and unstructured (semantic similarity) content retrieval can work together, making the system more robust and flexible.

The system can still discover and perform multi-hop retrieval without the vector database. It relies on the graph's structure and conceptual knowledge model relationships, which work well for clear and precise questions but can struggle with more vague or complex ones. It might miss out on related content where the structure and knowledge model concept relationships do not connect and could slow down when managing large amounts of information. Semantic retrieval is critical when including a mix of structured and unstructured content formats in the same model, such as Markdown or AsciiDoc, alongside DITA.

Let’s break this down in more detail.

Vector database as a semantic helper

In standard vector-based RAG, models break content into random chunks, vectorize them, and store them in a vector database. Standard vector-based RAG models employ similarity search to directly retrieve relevant chunks from the vector database fed to the LLM.

The vector database plays a different and complementary role in a DOM Graph RAG model. It isn’t used to feed chunks of content directly to the LLM. Instead, the vector DB is a semantic helper that helps *identify* contextually correct and relevant topics or elements. The DOM graph model retrieves content objects from the graph database using a SPARQL query.

The vector database in this architecture contains embeddings (vector representations) of the logical components of the documents as represented by the document structure. The model derives embeddings from the graph, not the source XML files or other document encodings. The model does not randomly chunk the content for the vector database as would be done when using a vector database in a conventional RAG model. Instead, the vector database content retains the original context of the local components of the original topics, as illustrated in Figure 4.

The similarity search in the vector database helps identify related or similar topics and elements based on the semantic meaning of the user query, even if those topics or elements are not directly connected through the graph structure or the taxonomy and ontological relationships.

After the vector database identifies similar topics or elements, it tells the system which specific nodes (topics or sub-elements) in the graph database to retrieve. The content used to generate answers or respond to queries comes from the graph database using SPARQL queries.

Graph database for structure-based retrieval:

As mentioned, the graph database remains the primary source for retrieving the content. The content is stored in RDF format, with structured relationships, hierarchies, and taxonomies. This structured content (like DITA topics and elements) is retrieved through SPARQL queries.

The graph database manages structured queries where the relationships (hierarchies, taxonomies, direct links between topics) are clearly defined. It is ideal for retrieving content explicitly linked in the structure, like parent-child or ontological concept relationships.

The role of the vector database in this hybrid model:

The vector database's role is to help identify similar or related topics that might not be linked in the graph. It provides semantic guidance that suggests content that could be relevant based on the meaning of the query, even if graph relationships don't connect those topics.

Once the vector database helps identify these relevant nodes, the graph database retrieves the actual content (topics, sub-elements, and so on) to generate the answer.

Native graph query without a vector database:

The native graph queries also use named entity recognition and retrieval to extract nouns and noun strings to search for related topics. In a pure graph-based-only model, we can build queries based on these extracted entities or labels and rely on inference and reasoning to find related content.

However, this approach might miss some semantic connections the vector database can capture. The vector database helps overcome limitations where the graph doesn't explicitly connect specific topics but where semantic similarity still exists. It adds a layer of flexibility that purely graph-based queries might lack, especially for handling ambiguous or more loosely structured queries.

To sum it up:

- **Graph database**: This database retrieves content from the structured RDF graph (such as DITA topics and elements) using structured SPARQL queries.
- **Vector database**: Acts as a semantic helper, identifying related content that may not be linked in the graph but is semantically similar. It helps guide the graph DB on which additional content should be retrieved.
- **Final content**: The graph database, not the vector database, contains all the content used to generate answers.

While the vector database aids the graph database by suggesting semantically related content, the model always retrieves the final payload of content sent to the LLM. The hybrid approach ensures flexibility and precision by combining the strengths of both systems.

How the DOM Graph RAG application works

We'll explain step-by-step what happens in this application when the user asks a question and include images of the results that the user receives based on sample questions. This working prototype illustrates integrating generative AI capabilities into a search experience. The possibilities are endless. Also, note that the success of graph-driven generative AI applications depends heavily on a good user experience and user interface design.

Step 1: Prompt refinement and assist:

As the user inputs a prompt, the knowledge graph assists the user in writing the prompt by making dynamic suggestions to formulate and improve the question as they type. The knowledge graph

enriches (grounds) the prompt, significantly reducing the burden on the user with prompt engineering.

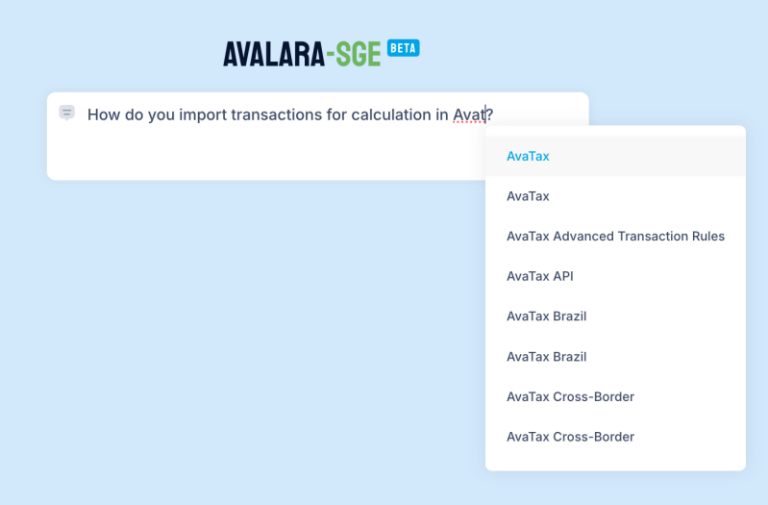

Figure 10. A sample DOM Graph RAG application

Step 2: Determining user intent:
The model sends the prompt to the LLM for intent recognition. That way, the model can tailor the content that best fits to answer the question to the user's intent, such as determining whether the user is seeking only conceptual information, obtaining reference information such as comparing functionality, obtaining a specific set of steps to perform a task, or how to fix a problem with troubleshooting instructions. You might recall that the topic model includes self-describing typed topics that categorize their purpose.

Step 3: Leveraging the vector database for semantic similarity search:
In addition to the LLM processing the user prompt for intent recognition (Step 2), the prompt is also sent to the vector database to identify semantically similar chunks of information. This step plays a crucial role in handling unstructured or less structured content and addressing queries where semantic similarity is important but direct links between concepts in the knowledge graph may not exist. As described earlier, the vector database is a semantic helper for the graph, not for direct content retrieval. Notice that there is no direct connection in Figure 9 between the vector database and the LLM.

Step 4: Query the graph to retrieve the content to send to the LLM:
Based on the user intent and the DITA elements suggested by the similarity search, the model selects the relevant DITA elements from the enriched DOM Graph via a dynamically generated SPARQL query to the graph database. The structure and traversal of this query adapt to the user's intent, targeting different and specific elements, such as whole topics, particular elements (for XML topics), or a combination of both. The query may also retrieve additional content associated

with the initial elements. The user request is then sent with the DITA elements to the LLM to generate a preliminary answer. The system augments the response by highlighting concepts from the knowledge model found in the answer. That way, we improve the user experience by providing contextual information.

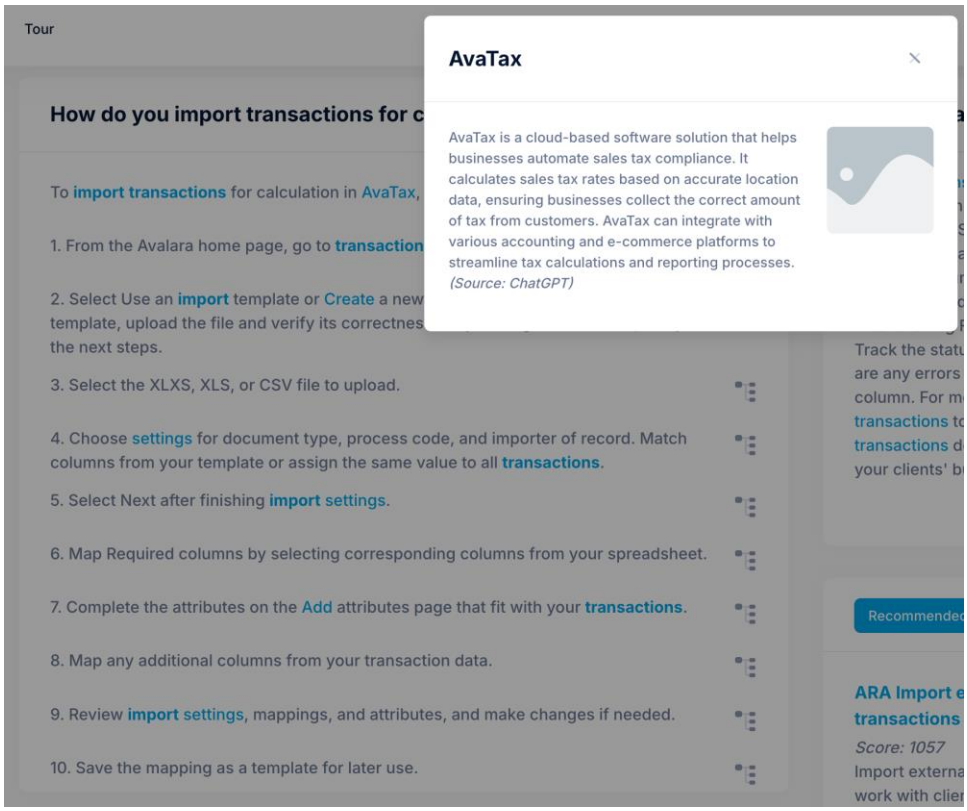

Figure 11. Summarized response

Step 5: The user request and the reply from the LLM are further expanded by the knowledge model and sent to Poolparty's Recommender engine to perform a lexical-weighted search and identify the additional documents most relevant to the user prompt. This process is highly configurable. Teams can add a domain ontology to the recommendation algorithm to identify content based on the user intent or context.

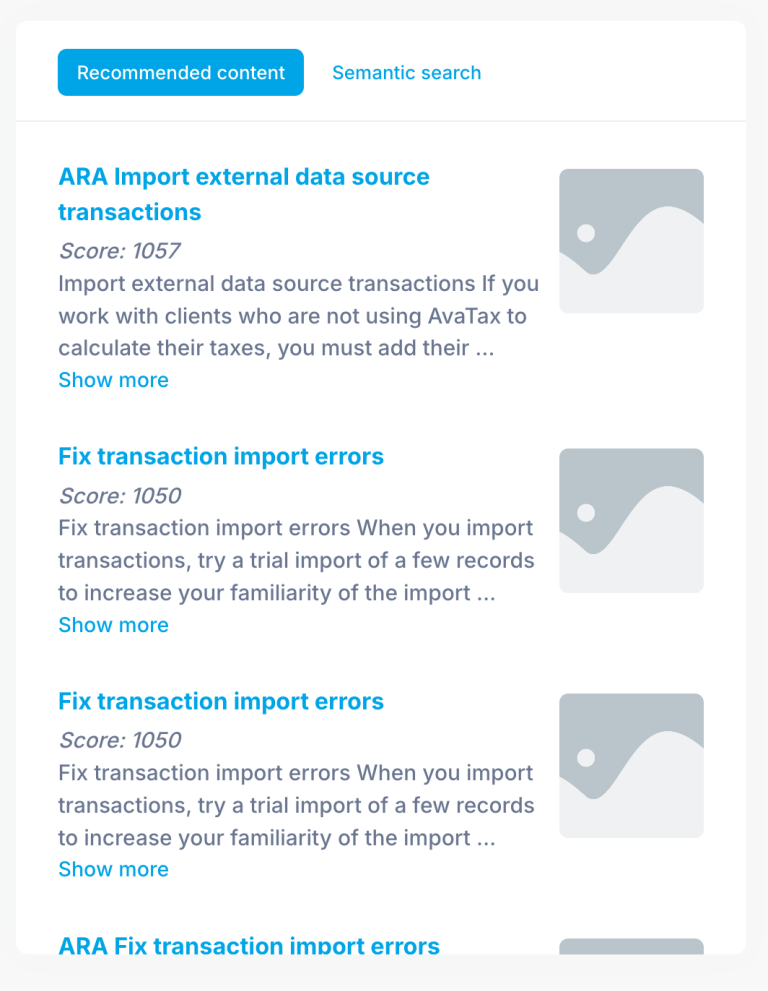

Figure 12. Recommended content

Step 6: Finally, the user prompt, the preliminary answer from the LLM, and the recommended documents are sent to the LLM to generate an answer to the user's question summarizing all the provided information.

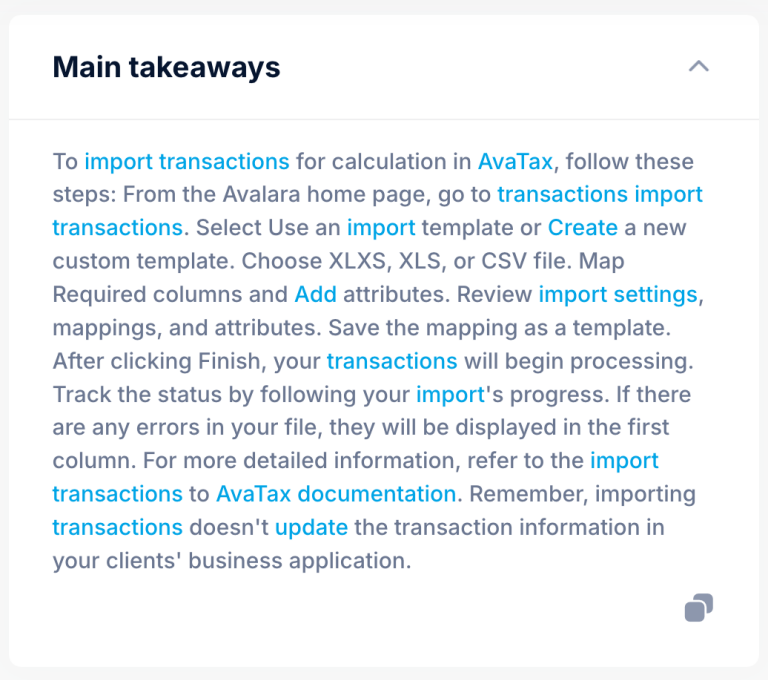

Figure 13. Main takeaway - generated summary

The application also generates follow-up questions that the user can ask to go into more detail as a next step.

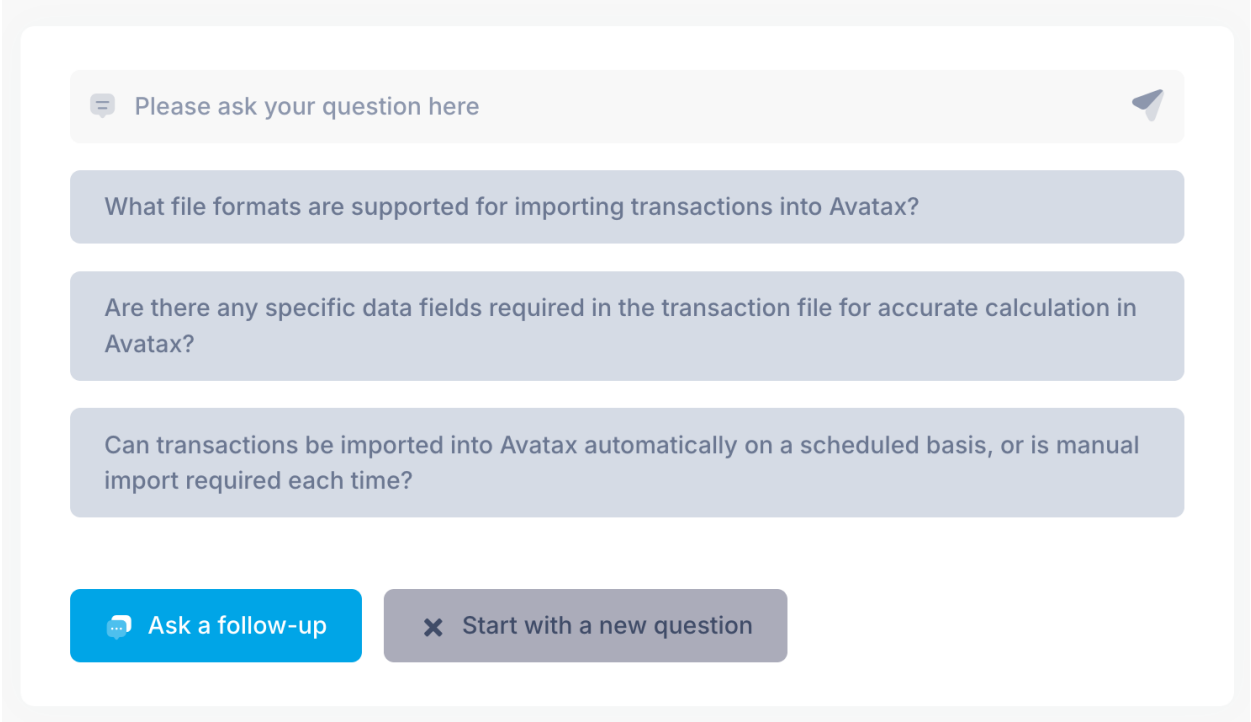

Figure 14. Suggested follow-up questions

The knowledge model guides the user from beginning to end throughout the search process. Since it is not a black box like the LLM algorithm, it allows tailoring of the request, gives context to the prompts sent to the LLM, and augments the answer that you get back to make the answer more explainable. In that sense, it allows you to tailor the answers you get to the specific domain of the users without training the model. It increases the precision of the answers, makes them explainable, and assists the user in better prompt engineering.

That also means that the quality of the knowledge model (taxonomy and ontology) plays an important role. Again, LLMs can help suggest new concepts and labels for the taxonomy and suggest classes and relations relevant to the ontology. However, this was not part of building this DOM GraphRAG application on top of an enriched DOM graph.

Links to referenced web pages

You might wonder how the model provides links to published and related content with answers. Generating those links was simple. We used IDs in the source topics to match the identical IDs carried through to the published web pages. Therefore, we can easily refer to or match corresponding XML files, knowledge graph resources, web pages, and elements within web pages.

Dependencies

Building a DOM Graph RAG solution requires time and investment, as with any custom RAG solution. Requirements to consider:

Topic-oriented content: The model can accommodate a variety of topic/component-oriented content formats, assuming you either have content in a component/topic model or are open to converting your content to a topic model. It can support topics written in many formats. Structured formats like DITA, other XML, or JSON-encoded topics can enhance content object retrieval. Suppose you need to manage monolithic sources of unstructured content or data. The DOM graph model can be combined with a separate vector database or graph solution in a hybrid configuration.

Domain knowledge model: The DOM Graph Model forms the structural basis for automating the construction and updating of a knowledge graph. Incorporating concepts related to your business, product, or services significantly enhances reasoning and inference to uncover indirectly connected content.

Taxonomy and ontology management: Applying taxonomy labels to your content significantly improves named entity recognition and retrieval, semantic and neuro-symbolic inferencing, and reasoning. Developing and maintaining taxonomy takes time, effort, and governance. Taxonomy labels can be applied to content manually, semi-automatically (using a recommender), or automatically using an autoclassifier provided by many taxonomy management tools. Creating taxonomies and ontologies requires knowledge of your business and design skills.

Designing a process or system that manages changes to the applied taxonomies and ontologies is essential. We recommend using semantic platforms such as PoolParty Semantic Suite to develop, maintain, manage, and govern taxonomies and ontologies and apply them to your content. Some

content management systems integrate or have out-of-the-box connectors with commercial taxonomy/ontology management platforms, or you can establish a custom integration via API.

Transforms: The DITA Open Toolkit on SourceForge contains a transform engine. Creating or modifying transforms requires expertise in XSLT or similar skills. In addition, a text-mining tool is needed to enrich the DOM graph with taxonomies and ontologies. In our case, the respective component of PoolParty Semantic Suite was used for this task.

Application building: This model requires you to develop a user-facing application, such as a chatbot, and connect it with one or more delivery channels, such as your help website, technical support site, and in-product help. The application we built in our case was built on Poolparty's GraphSearch and Recommender components, which provide APIs to build such applications.

Automating content updates: Most production systems will not have static content. By that, you need to plan for continuous content updates, and your taxonomies and ontologies will change over time. This can be achieved by integrating automation tools to automate and schedule content source transforms between the involved content management system or source repository of the used semantic platform and its component, the graph database, and other indexes (e.g., vector database) that must be created and continuously updated. PoolParty's automation component was used for our project UnifiedViews.

State metadata: The paper discusses using state metadata to fulfill advanced content management requirements, including entitlement, multilingual support, version control, content retirement, and more. State metadata can be embedded within content nodes in a graph, in DITAMaps, and optionally within individual topics. State metadata can be included for non-DITA files using the OtherProps or custom attributes on DITAMap TopicRef elements. Ideally, maintaining a separate electronic system of record (SoR), such as a publishing plan or agenda, can track the status and information about all content and interface with the graph to populate and maintain state metadata automatically.

Summary and Conclusions

The development of large language models has revolutionized AI by bringing generative models to the forefront. While initial applications led to a frenzy of investments, the limitations of LLMs—especially in accuracy and reliability—became apparent, spurring the adoption of retrieval-augmented generation (RAG) models. However, vector-based RAG models still face critical challenges, such as hallucinations, loss of context, and lack of precision due to their reliance on probabilistic methods.

Moreover, critical content management capabilities are virtually absent from most vector-based RAG models. One can argue that the lack of vital content management capabilities in current models, such as those outlined in this paper, is tantamount to professional content management malpractice, exposing a company's reputation, if not liability.

The Document Object Model GraphRAG (DOM Graph RAG) model was developed to overcome these limitations by integrating structured content and neuro-symbolic reasoning. Unlike vector-

based models, which struggle with preserving context and managing dynamic content, DOM Graph RAG utilizes a graph database to maintain the original structure, relationships, and metadata. This ensures more accurate, reliable, and explainable outputs. It also automates graph construction from existing structured content, supporting frequent updates without loss of context and significantly improving change management.

We evaluated the DOM Graph RAG prototype model against a mature traditional vector-based RAG model using the same content corpora. We achieved reliable neuro-symbolic and semantic retrieval even with a minimal domain knowledge model. The DOM Graph RAG model effectively answered test questions and provided accurate recommendations. The results showed significant improvement when domain knowledge concepts were integrated, with continued gains as the concept model became more mature and robust. As the content was further enhanced with taxonomy, the model's performance improved even further.

Key benefits of the DOM Graph RAG model include enhanced accuracy through knowledge graphs, the ability to manage complex relationships and multi-hop retrieval, and superior efficiency in managing large-scale content. Reducing dependence on LLMs for retrieval also lowers computational costs. The model further supports advanced personalization and recommendations, improving the user experience by providing precise, context-aware outputs.

The DOM Graph RAG model is scalable, can integrate with hybrid approaches (combining vector and graph databases), and supports critical capabilities like version control, reuse, entitlement, and localization. Using a graph-based approach, DOM Graph RAG provides a more robust, cost-effective solution for AI-driven content management, offering substantial advantages in high-stakes environments requiring accuracy, trustworthiness, and efficiency.

The DOM Graph RAG project was developed as an industry model using open standards without intending to sell products or services. While the tools and providers used in its development are recommended, organizations can choose their technology partners and tools. The key benefit of adopting DOM Graph RAG is complete control over your solution and knowledge model, avoiding dependency on external providers and the risks of obsolescence or vendor lock-in in this rapidly evolving space.

DOM Graph RAG - a sustainable model that adds content and knowledge in the loop.

Glossary

AI (Artificial Intelligence)
AI refers to machines or software that can perform tasks typically requiring human intelligence. These include problem-solving, understanding natural language, and decision-making. For example, AI powers virtual assistants like Siri or Alexa.

Chunking
In RAG models, content is often broken into smaller pieces, or "chunks," to make it easier to process. For example, a long article might be split into segments paragraphs, storing each segment separately to make retrieval faster.

Componentized Content
Componentized content breaks down large documents into smaller, reusable components or topics. Unlike "chunking," componentization ensures that content is structured and organized to retain context and relationships between sections, which can be efficiently retrieved and reused.

Content Ontology
A content ontology formally represents the structure and relationships between various content components. It defines how content types relate, enabling better management, retrieval, and reuse of structured content. Content ontologies are vital for creating efficient and scalable content management systems in industries requiring precision and organization.

Content State Management
Content state management involves tracking and maintaining content's various statuses, such as whether it is published, expired, in draft mode, or versioned. In structured content systems, this metadata ensures that only the appropriate content is retrieved and displayed, reducing errors and ensuring compliance with content governance policies.

Cosine Similarity
Cosine similarity is a mathematical measure used to compare the similarity between two pieces of text, represented as vectors, by calculating the angle between them. If two vectors point in the same direction, they are considered similar. For example, two documents discussing "Paris" and "France" might have a high cosine similarity because they contain related concepts.

DITA (Darwin Information Typing Architecture)
DITA is a standard for creating structured documents. It's widely used for writing technical content, and its structured nature makes it easy to manage, reuse, and transform content. For instance, technical manuals are often written in DITA format.

DITA Specialization
DITA Specialization refers to extending the Darwin Information Typing Architecture (DITA) to create custom topic types tailored to specific industry or organizational needs. These specializations inherit properties from basic DITA types and allow for greater flexibility in structured content authoring.

Document Object Model (DOM)
DOM is a programming structure that represents a document (like an HTML or XML file) as a tree of objects. Each part of the document (for example, a paragraph or heading) is a node in this tree, which programs can manipulate. For example, JavaScript uses the DOM to dynamically change a webpage’s content.

Fact-Checking in AI
Fact-checking involves validating the information generated by an AI system against a reliable external source, like a knowledge graph, to ensure accuracy.

Generative AI
Generative artificial intelligence refers to models that can create new content, like text, images, or music. An example is DALL·E, which generates images based on textual descriptions.

GraphDB
GraphDB, from Ontotext, is a graph database management system designed to store and manage large amounts of data as a knowledge graph. It allows users to query and infer relationships between data points using SPARQL, making it ideal for complex data retrieval applications. For example, a company might use GraphDB to store product information, customer data, and their interrelations, which can then be queried to provide tailored recommendations or insights.

These tools are essential in building AI and knowledge-based systems. They allow for the structured and dynamic management of large datasets and enhance the capabilities of content retrieval, reasoning, and decision-making.

Graph Embeddings
Graph embeddings represent the nodes and edges of a graph as numerical vectors, making it easier for AI models to work with graph data. For example, in a social network graph, each person could be a node, and their relationships would be edges.

Hybrid Model (Graph and Vector)
A hybrid model uses graph databases (structured retrieval based on relationships) and vector databases (semantic similarity search). This makes the system more flexible, handling both well-defined and ambiguous queries.

Inference Engine
An inference engine is a component of a knowledge system that applies logical rules to the data within a knowledge graph to derive new information or conclusions. In neuro-symbolic AI, inference engines help validate the information, provide fact-checking capabilities, and enable more accurate and logical responses from AI models.

Knowledge Graph
A knowledge graph is a database that stores information in a structured format where each piece of data is linked to others, forming a network. It helps machines understand relationships between data. For example, a knowledge graph might link "Paris" to "France" as the capital city.

LLM (Large Language Models)
LLMs are an AI model that can understand and generate human language based on vast amounts of text data. Examples include OpenAI's ChatGPT, which can generate coherent text based on prompts.

Multi-Hop Retrieval
Multi-hop retrieval is an AI method in which multiple pieces of information are connected to answer a complex question. For example, to answer "Which country is the president of Apple from?" the AI needs first to find "Tim Cook" (the president of Apple) and then search for "Tim Cook's nationality."

Neuro-Symbolic Reasoning
Neuro-symbolic reasoning refers to AI models that combine neural networks (which learn patterns) with symbolic reasoning

(which uses logical rules). Symbolic reasoning helps AI systems reason about data, check facts, and infer new information rather than just relying on patterns.

NER (Named Entity Recognition)
NER is a process where AI identifies and classifies proper nouns (like names of people, organizations, dates, or places) in a text. For example, in the sentence "Steve Jobs founded Apple," NER would identify "Apple" as a company and "Steve Jobs" as a person.

Ontology
An ontology is a formal representation of knowledge in a domain. It defines the categories, properties, and relationships between concepts. For example, in healthcare ontology, concepts like "Doctor," "Patient," and "Hospital" would be defined, along with their relationships.

PoolParty Semantic Suite
PoolParty Semantic Suite, from Semantic Web Company, is a semantic technology platform for building and managing knowledge graphs, ontologies, taxonomies, and linked data. It helps organize and link data meaningfully, enabling better content retrieval and reasoning. For example, PoolParty can be used by businesses to categorize and structure large datasets, improving searchability and decision-making by connecting related concepts.

Precision Paradox
This is a phenomenon where improving a system's precision (accuracy) makes any errors more noticeable and impactful. For example, if a system becomes particularly good at answering questions, even a tiny mistake becomes more glaring.

RAG (retrieval-augmented generation)
RAG is an AI technique in which a model retrieves relevant information from external sources. For instance, when you ask a RAG model a question, it might search a database for relevant documents and then use that information to create an accurate response.

RDF (Resource Description Framework)
RDF is a framework for representing information about resources on the web. It represents data as triples: subject-predicate-object, which describe relationships. For example, "Paris is the capital of France" can be an RDF triple.

SPARQL
SPARQL is a query language that retrieves and manipulates data stored in knowledge graphs. It works like SQL for databases but is designed for graphs. For example, it could be used to find all the cities in France using a knowledge graph.

Vector Database
A vector database stores data in numerical form (vectors) based on their meaning. It allows for similarity searches. In AI, it helps find content that is "similar" to a query. For example, if you search for "cat," it might return information on "felines" as well.

XML (Extensible Markup Language)
XML is a markup language that defines rules for encoding documents in a human-readable and machine-readable format. Websites often use XML to store data that needs to be shared between different systems.