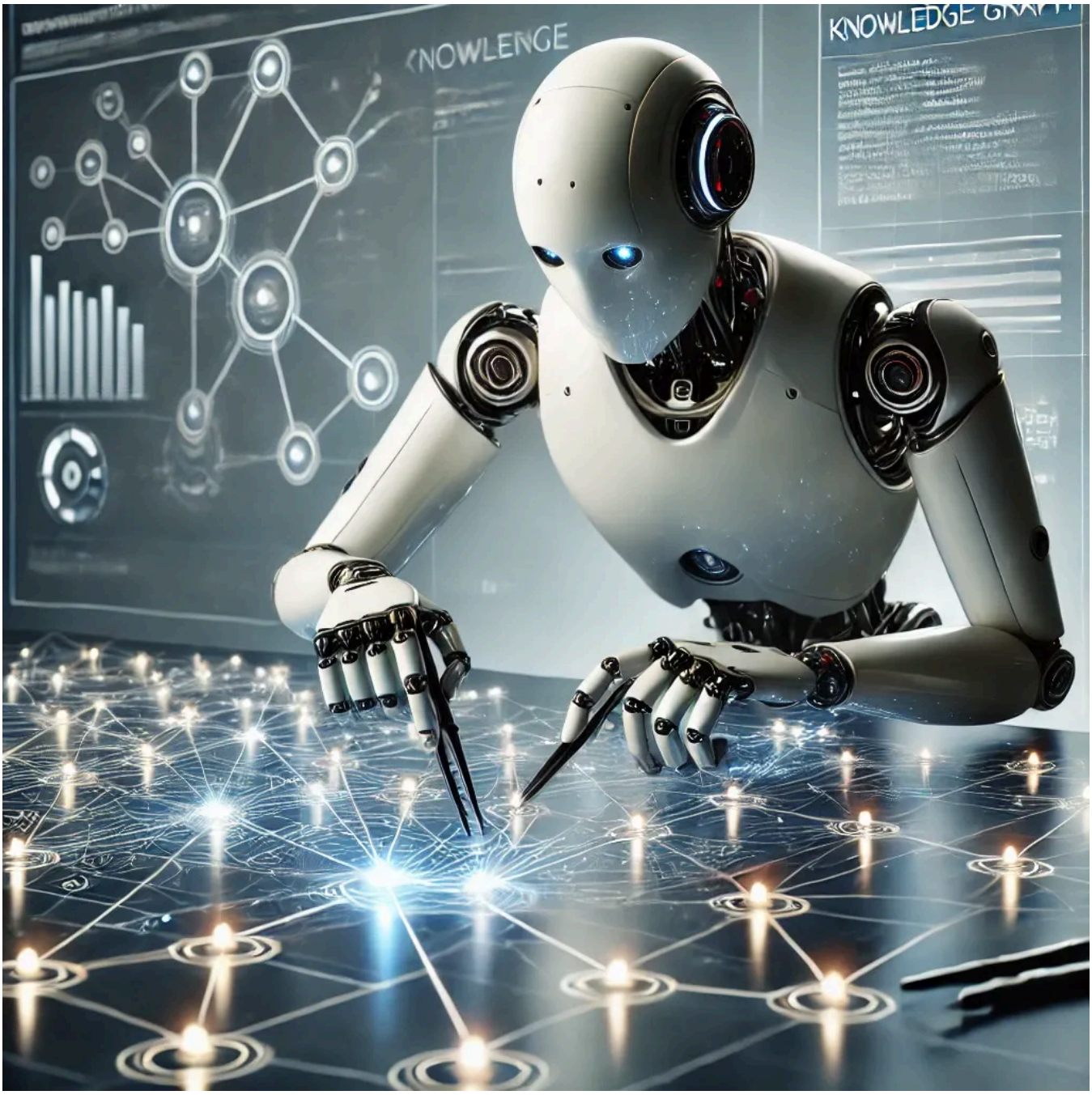
Building Knowledge Graphs with LLM Graph Transformer

A deep dive into LangChain's implementation of graph construction with LLMs

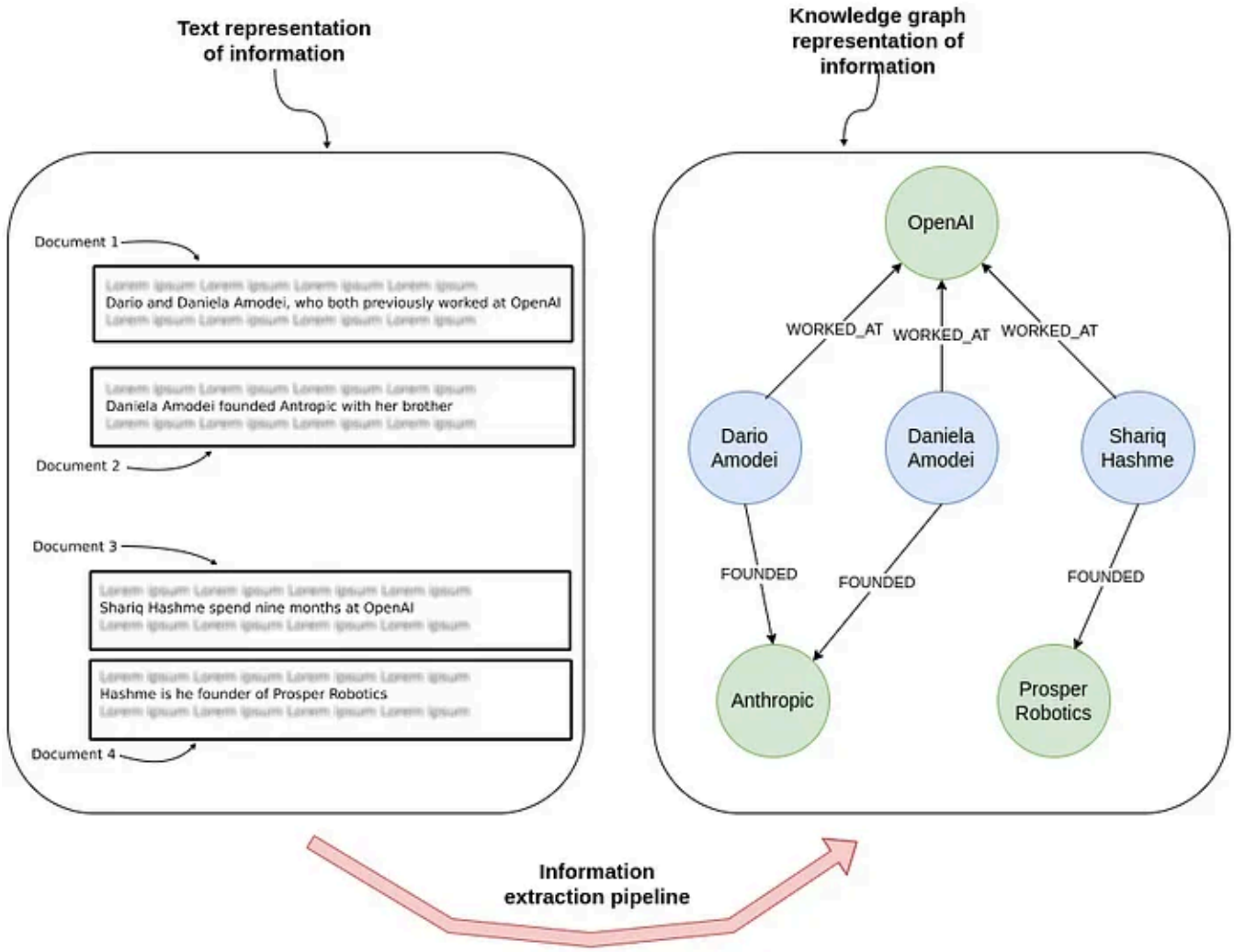
17 min read · Nov 5, 2024

Tomaz Bratanic



Building knowledge graph. Image by ChatGPT.

Creating graphs from text is incredibly exciting, but definitely challenging. Essentially, it's about converting unstructured text into structured data. While this approach has been around for some time, it gained significant traction with the advent of Large Language Models (LLMs), bringing it more into the mainstream.



Extracting entities and relationships from text to construct a knowledge graph. Image by author.

In the image above, you can see how information extraction transforms raw text into a knowledge graph. On the left, multiple documents show unstructured sentences about individuals and their relationships with companies. On the right, this same information is represented as a graph of entities and their connections, showing who worked at or founded various organizations.

But why would you want to extract structured information from text and represent it as a graph? One key reason is to power retrieval-augmented generation (RAG) applications. While using text embedding models over unstructured text is an useful approach, it can fall short when it comes to answering complex, multi-hop questions that require understanding connections across multiple entities or question where structured operations like filtering, sorting, and aggregation is required. By extracting structured information from text and constructing knowledge graphs, you not only organize data more effectively but also create a powerful framework for understanding complex relationships between entities. This structured approach makes it much easier to retrieve and leverage specific

information, expanding the types of questions you can answer while providing greater accuracy.

Around a year ago, I began experimenting with building graphs using LLMs, and due to the growing interest, we decided to integrate this capability into LangChain as the LLM Graph Transformer. Over the past year, we've gained valuable insights and introduced new features, which we'll be showcasing in this blog post.

The code is available on GitHub.

Setting up Neo4j environment

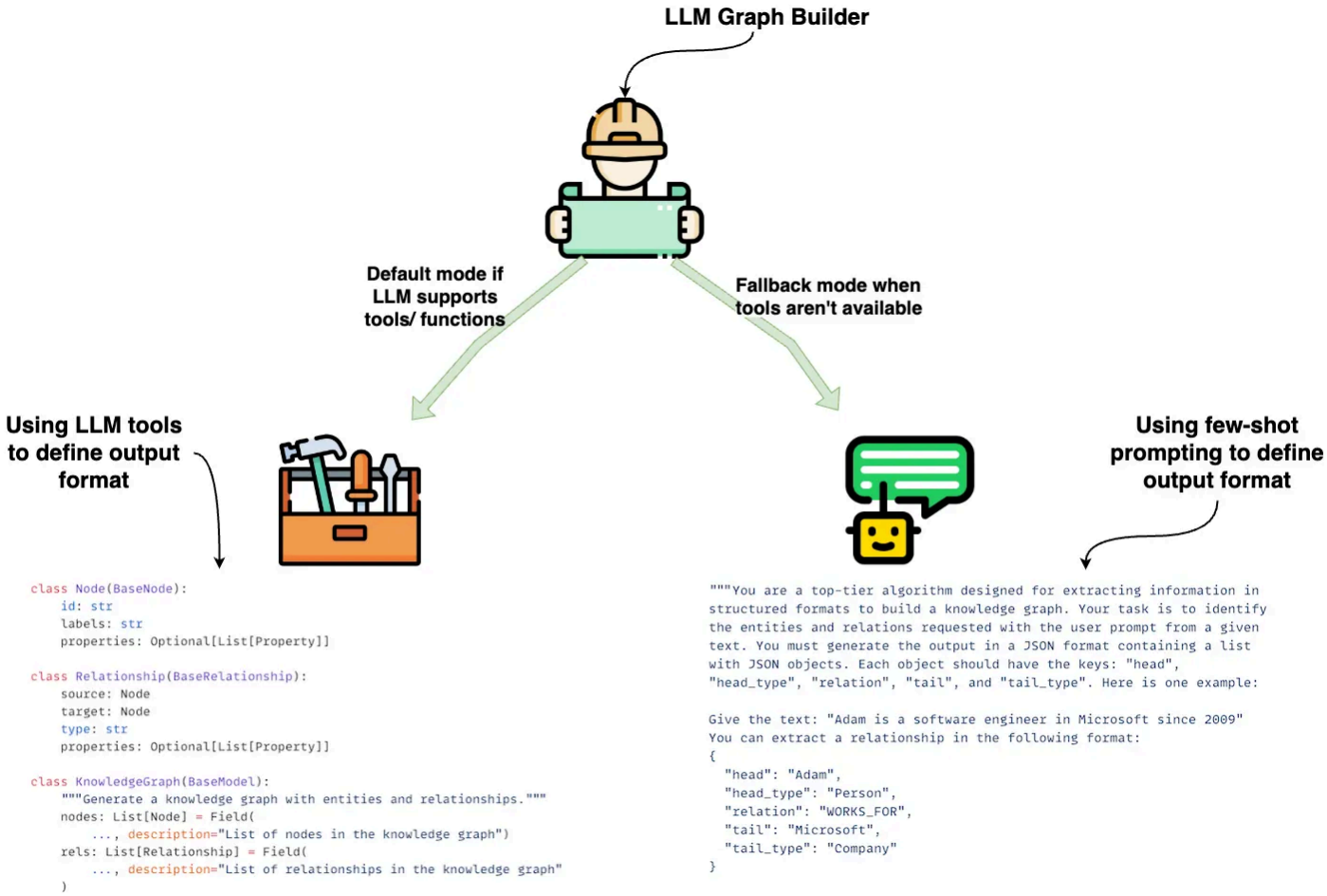
We will use Neo4j as the underlying graph store, which comes with out-of-the box graph visualizations. The easiest way to get started is to use a free instance of Neo4j Aura, which offers cloud instances of the Neo4j database. Alternatively, you can set up a local instance of the Neo4j database by downloading the Neo4j Desktop application and creating a local database instance.

```
from langchain_community.graphs import Neo4jGraph

graph = Neo4jGraph(
    url="bolt://54.87.130.140:7687",
    username="neo4j",
    password="cables-anchors-directories",
    refresh_schema=False
)
```

LLM Graph Transformer

The LLM Graph Transformer was designed to provide a flexible framework for building graphs using any LLM. With so many different providers and models available, this task is far from simple. Fortunately, LangChain steps in to handle much of the standardization process. As for the LLM Graph Transformer itself, it's like two cats stacked in a trench coat —with the ability to operate in two completely independent modes.



LLM Graph Transformer consists of two separate modes of extracting graphs from text. Image by user.

The LLM Graph Transformer operates in two distinct modes, each designed to generate graphs from documents using an LLM in different scenarios.

1. **Tool-Based Mode (Default):** When the LLM supports structured output or function calling, this mode leverages the LLM's built-in `with_structured_output` to use tools. The tool specification defines the output format, ensuring that entities and relationships are extracted in a structured, predefined manner. This is depicted on the left side of the image, where code for the Node and Relationship classes is shown.

2. **Prompt-Based Mode (Fallback):** In situations where the LLM doesn't support tools or function calls, the LLM Graph Transformer falls back to a purely prompt-driven approach. This mode uses few-shot prompting to define the output format, guiding the LLM to extract entities and relationships in a text-based manner. The results are then parsed through a custom function, which converts the LLM's output into a JSON format. This JSON is used to populate

nodes and relationships, just as in the tool-based mode, but here the LLM is guided entirely by prompting rather than structured tools. This is shown on the right side of the image, where an example prompt and resulting JSON output are provided.

These two modes ensure that the LLM Graph Transformer is adaptable to different LLMs, allowing it to build graphs either directly using tools or by parsing output from a text-based prompt.

Note that you can use prompt-based extraction even with models that support tools/functions by setting the attribute `ignore_tools_usage=True` *.*

Tool-based extraction

We initially chose a tool-based approach for extraction since it minimized the need for extensive prompt engineering and custom parsing functions. In LangChain, the `with_structured_output` method allows you to extract information using tools or functions, with output defined either through a JSON structure or a Pydantic object. Personally, I find Pydantic objects clearer, so we opted for that.

We start by defining a `Node` class.

```
class Node(BaseNode):
    id: str = Field(..., description="Name or human-readable unique identifier"
    label: str = Field(..., description=f"Available options are {enum_values}")
    properties: Optional[List[Property]]
```

Each node has an `id`, a `label`, and optional `properties`. For brevity, I haven't included full descriptions here. Describing ids as human-readable unique identifier is important since some LLMs tend to understand ID properties in more traditional way like random strings or incremental integers. Instead we want the name of entities to be used as id property. We also limit the available label types by simply listing them in the `label` description. Additionally, LLMs like OpenAI's, support an `enum` parameter, which we also use.

Next, we take a look at the `Relationship` class

```
class Relationship(BaseRelationship):
    source_node_id: str
    source_node_label: str = Field(..., description=f"Available options are {er
    target_node_id: str
    target_node_label: str = Field(..., description=f"Available options are {er
    type: str = Field(..., description=f"Available options are {enum_values}")
    properties: Optional[List[Property]]
```

This is the second iteration of the `Relationship` class. Initially, we used a nested `Node` object for the source and target nodes, but we quickly found that nested objects reduced the accuracy and quality of the extraction process. So, we decided to flatten the source and target nodes into separate fields—for example, `source_node_id` and `source_node_label`, along with `target_node_id` and `target_node_label`. Additionally, we define the allowed values in the descriptions for node labels and relationship types to ensure the LLMs adhere to the specified graph schema.

The tool-based extraction approach enables us to define properties for both nodes and relationships. Below is the class we used to define them.

```
class Property(BaseModel):
    """A single property consisting of key and value"""
    key: str = Field(..., description=f"Available options are {enum_values}")
    value: str
```

Each `Property` is defined as a key-value pair. While this approach is flexible, it has its limitations. For instance, we can't provide a unique description for each property, nor can we specify certain properties as mandatory while others optional, so all properties are defined as optional. Additionally, properties aren't defined individually for each node or relationship type but are instead shared across all of them.

We've also implemented a detailed system prompt to help guide the extraction. In my experience, though, the function and argument descriptions tend to have a greater impact than the system message.

Unfortunately, at the moment, there is no simple way to customize function or argument descriptions in LLM Graph Transformer.

Prompt-based extraction

Since only a few commercial LLMs and LLaMA 3 support native tools, we implemented a fallback for models without tool support. You can also set `ignore_tool_usage=True` to switch to a prompt-based approach even when using a model that supports tools.

Most of the prompt engineering and examples for the prompt-based approach were contributed by Geraldus Wilsen.

With the prompt-based approach, we have to define the output structure directly in the prompt. You can find the whole prompt here. In this blog post, we'll just do a high-level overview. We start by defining the system prompt.

```
You are a top-tier algorithm designed for extracting information in structured

- **"head"**: The text of the extracted entity, which must match one of the typ
- **"head_type"**: The type of the extracted head entity, selected from the spe
- **"relation"**: The type of relation between the "head" and the "tail," chose
- **"tail"**: The text of the entity representing the tail of the relation.
- **"tail_type"**: The type of the tail entity, also selected from the provided

Extract as many entities and relationships as possible.

**Entity Consistency**: Ensure consistency in entity representation. If an enti

**Important Notes**:
- Do not add any extra explanations or text.
```

In the prompt-based approach, a key difference is that we ask the LLM to extract only relationships, not individual nodes. This means we won't have any *isolated nodes*, unlike with the tool-based approach. Additionally, because models lacking native tool support typically perform worse, we do not allow extraction any properties — whether for nodes or relationships, to keep the extraction output simpler.

Next, we add a couple of few-shot examples to the model.

```
examples = [
    {
        "text": (
            "Adam is a software engineer in Microsoft since 2009, "
            "and last year he got an award as the Best Talent"
        ),
        "head": "Adam",
        "head_type": "Person",
        "relation": "WORKS_FOR",
        "tail": "Microsoft",
        "tail_type": "Company",
    },
    {
        "text": (
            "Adam is a software engineer in Microsoft since 2009, "
            "and last year he got an award as the Best Talent"
        ),
        "head": "Adam",
        "head_type": "Person",
        "relation": "HAS_AWARD",
        "tail": "Best Talent",
        "tail_type": "Award",
    },
...
]
```

In this approach, there's currently no support for adding custom few-shot examples or extra instructions. The only way to customize is by modifying the entire prompt through the `prompt` attribute. Expanding customization options is something we're actively considering.

Next, we'll take a look at defining the graph schema.

Defining the graph schema

When using the LLM Graph Transformer for information extraction, defining a graph schema is essential for guiding the model to build meaningful and structured knowledge representations. A well-defined graph schema specifies the types of nodes and relationships to be extracted, along with any attributes associated with each. This schema serves as a blueprint, ensuring that the LLM consistently extracts relevant information in a way that aligns with the desired knowledge graph structure.

In this blog post, we'll use the opening paragraph of Marie Curie's Wikipedia page for testing with an added sentence at the end about Robin Williams.

```
from langchain_core.documents import Document

text = """
Marie Curie, 7 November 1867 – 4 July 1934, was a Polish and naturalised-French
She was the first woman to win a Nobel Prize, the first person to win a Nobel P
Her husband, Pierre Curie, was a co-winner of her first Nobel Prize, making the
She was, in 1906, the first woman to become a professor at the University of Pa
Also, Robin Williams.
"""
documents = [Document(page_content=text)]
```

We'll also be using GPT-4o in all examples.

```
from langchain_openai import ChatOpenAI
import getpass
import os

os.environ["OPENAI_API_KEY"] = getpass.getpass("OpenAI api key")

llm = ChatOpenAI(model='gpt-4o')
```

To start, let's examine how the extraction process works without defining any graph schema.

```
from langchain_experimental.graph_transformers import LLMGraphTransformer

no_schema = LLMGraphTransformer(llm=llm)
```

Now we can process the documents using the `aconvert_to_graph_documents` function, which is asynchronous. Using async with LLM extraction is recommended, as it allows for parallel processing of multiple documents. This approach can significantly reduce wait times and improve throughput, especially when dealing with multiple documents.

```
data = await no_schema.aconvert_to_graph_documents(documents)
```

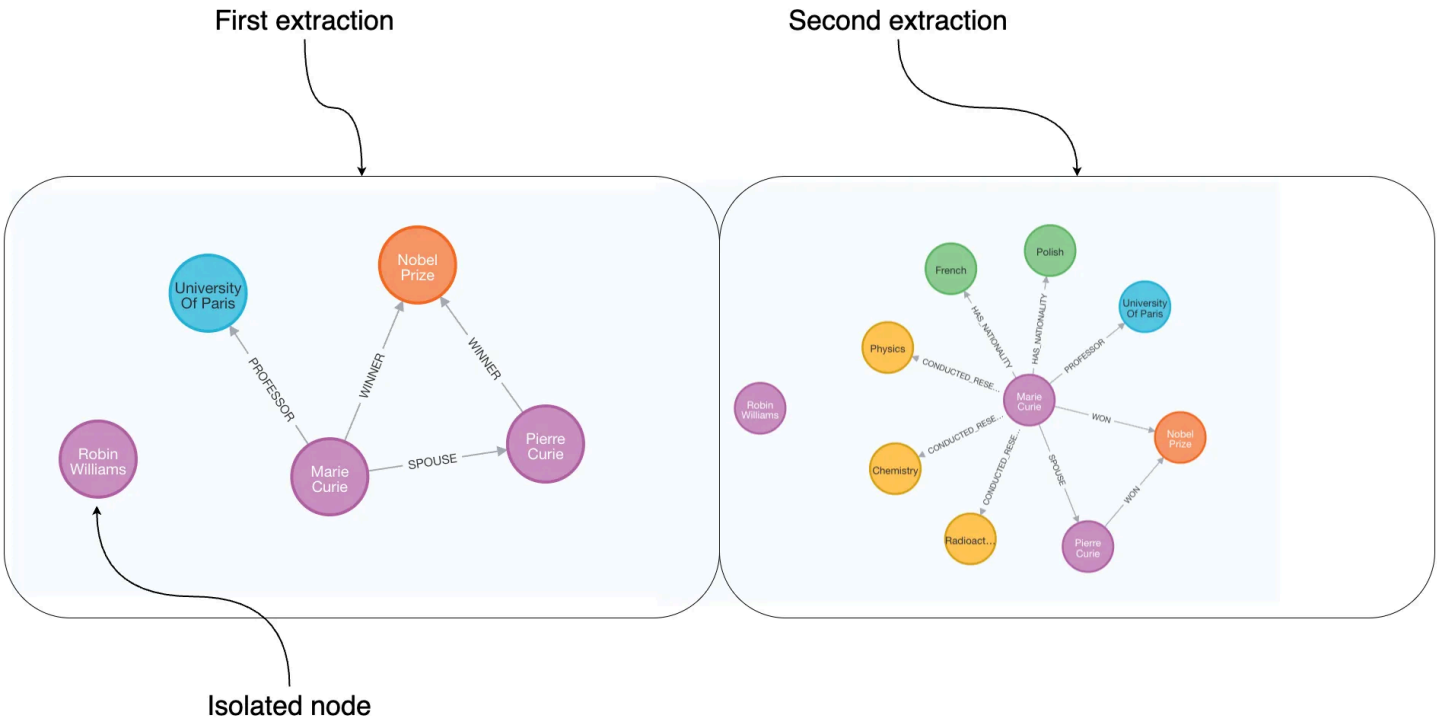
The response from the LLM Graph Transformer will be a graph document, which has the following structure:

```
[
    GraphDocument(
        nodes=[
            Node(id="Marie Curie", type="Person", properties={}),
            Node(id="Pierre Curie", type="Person", properties={}),
            Node(id="Nobel Prize", type="Award", properties={}),
            Node(id="University Of Paris", type="Organization", properties={}),
            Node(id="Robin Williams", type="Person", properties={}),
        ],
        relationships=[
            Relationship(
                source=Node(id="Marie Curie", type="Person", properties={}),
                target=Node(id="Nobel Prize", type="Award", properties={}),
                type="WON",
                properties={},
            ),
            Relationship(
                source=Node(id="Marie Curie", type="Person", properties={}),
                target=Node(id="Nobel Prize", type="Award", properties={}),
                type="WON",
                properties={},
            ),
            Relationship(
                source=Node(id="Marie Curie", type="Person", properties={}),
                target=Node(
                    id="University Of Paris", type="Organization", properties={
                ),
                type="PROFESSOR",
                properties={},
            ),
            Relationship(
                source=Node(id="Pierre Curie", type="Person", properties={}),
                target=Node(id="Nobel Prize", type="Award", properties={}),
                type="WON",
                properties={},
            ),
        ],
        source=Document(
            metadata={"id": "de3c93515e135ac0e47ca82a4f9b82d8"},
            page_content="\nMarie Curie, 7 November 1867 - 4 July 1934, was a F
        ),
```

```
        )
    ]
```

The graph document describes extracted `nodes` and `relationships`. Additionally, the source document of the extraction is added under the `source` key.

We can use the Neo4j Browser to visualize the outputs, providing a clearer and more intuitive understanding of the data.



Visualization of two extraction passes over the same dataset without a defined graph schema. Image by author.

The image above shows two extraction passes over the same paragraph about Marie Curie. In this case, we used GPT-4 with tool-based extraction, which also allows for isolated nodes, as illustrated in the image. Because no graph schema was defined, the LLM determines at runtime what information to extract, which can lead to variations in the output, even from the same paragraph. As a result, some extractions are more detailed than others and may vary in structure, even for the same information. For instance, on the left, Marie is represented as the `WINNER` of the Nobel Prize, while on the right, she `WON` the Nobel Prize.

Now, let's try the same extraction using the prompt-based approach. For models that support tools, you can enable prompt-based extraction by setting the `ignore_tool_usage` parameter.

```
no_schema_prompt = LLMGraphTransformer(llm=llm, ignore_tool_usage=True)
data = await no_schema.aconvert_to_graph_documents(documents)
```

Again, we can visualize two separate executions in Neo4j Browser.

Visualization of two extraction passes over the same dataset without a defined graph schema using the prompt-based approach. Image by author.

With the prompt-based approach, we won't see any isolated nodes. However, as with previous extractions, the schema can vary between runs, resulting in different outputs on the same input.

Next, let's walk through how defining a graph schema can help produce more consistent outputs.

Defining allowed nodes

Constraining the extracted graph structure can be highly beneficial, as it guides the model to focus on specific, relevant entities and relationships. By defining a clear schema, you improve consistency across extractions, making the outputs more predictable and aligned with the information you actually need. This reduces variability between runs and ensures that the extracted data follows a standardized

structure, capturing expected information. With a well-defined schema, the model is less likely to overlook key details or introduce unexpected elements, resulting in cleaner, more usable graphs.

We'll start by defining the expected types of nodes to extract using the `allowed_nodes` parameter.

```
allowed_nodes = ["Person", "Organization", "Location", "Award", "ResearchField"
nodes_defined = LLMGraphTransformer(llm=llm, allowed_nodes=allowed_nodes)
data = await allowed_nodes.aconvert_to_graph_documents(documents)
```

Here, we defined that the LLM should extract five types of nodes like *Person*, *Organization*, *Location*, and more. We visualize two separate executions in Neo4j Browser for comparison.

Visualization of two extraction passes with predefined node types. Image by author.

By specifying the expected node types, we achieve more consistent node extraction. However, some variation may still occur. For example, in the first run, "radioactivity" was extracted as a research field, while in the second, it was not.

Since we haven't defined relationships, their types can also vary across runs. Additionally, some extractions may capture more information than others. For

instance, the `MARRIED_TO` relationship between Marie and Pierre isn't present in both extractions.

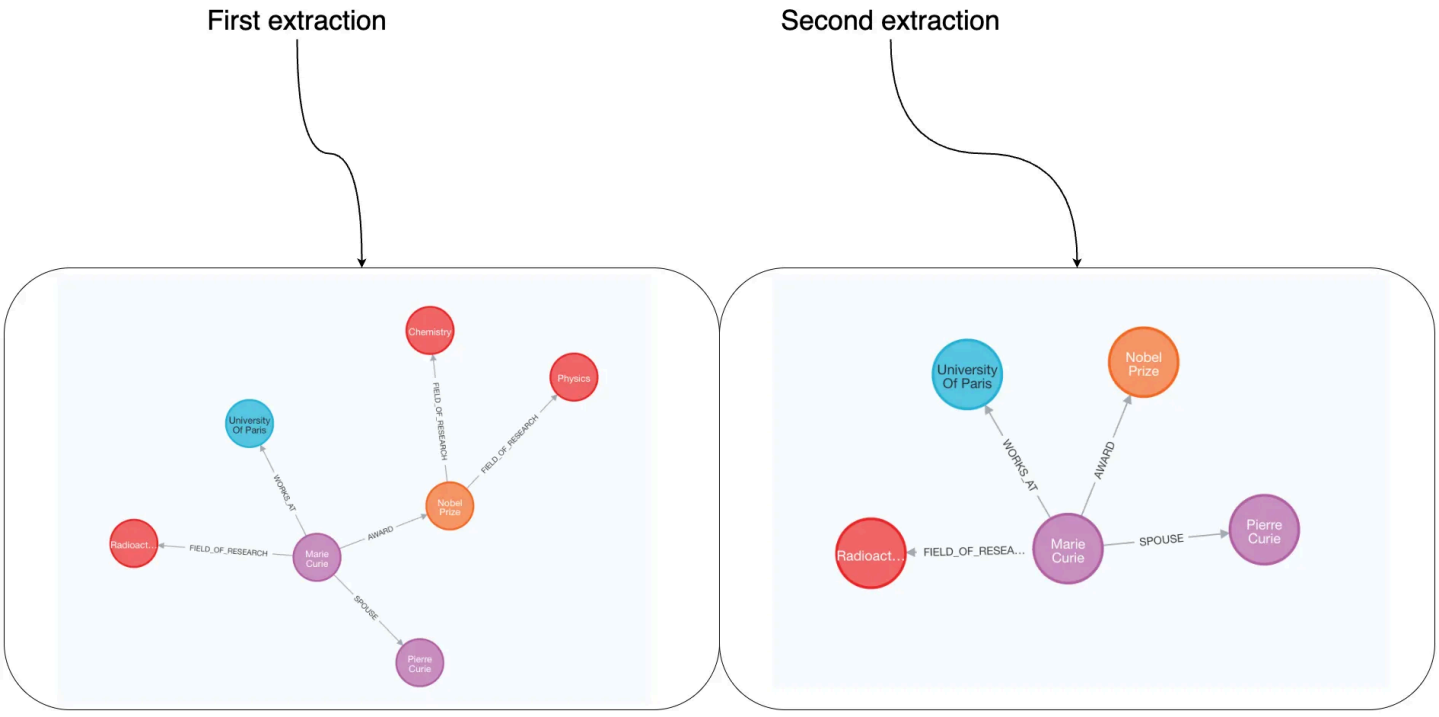
Now, let's explore how defining relationship types can further improve consistency.

Defining allowed relationships

As we've observed, defining only node types still allows for variation in relationship extraction. To address this, let's explore how to define relationships as well. The first approach is to specify allowed relationships using a list of available types.

```
allowed_nodes = ["Person", "Organization", "Location", "Award", "ResearchField"
allowed_relationships = ["SPOUSE", "AWARD", "FIELD_OF_RESEARCH", "WORKS_AT", "I
rels_defined = LLMGraphTransformer(
  llm=llm,
  allowed_nodes=allowed_nodes,
  allowed_relationships=allowed_relationships
)
data = await rels_defined.aconvert_to_graph_documents(documents)
```

Let's again examine two separate extractions.



Visualization of two extraction passes with predefined node and relationship types. Image by author.

With both nodes and relationships defined, our outputs become significantly more consistent. For example, Marie is always shown as winning an award, being the

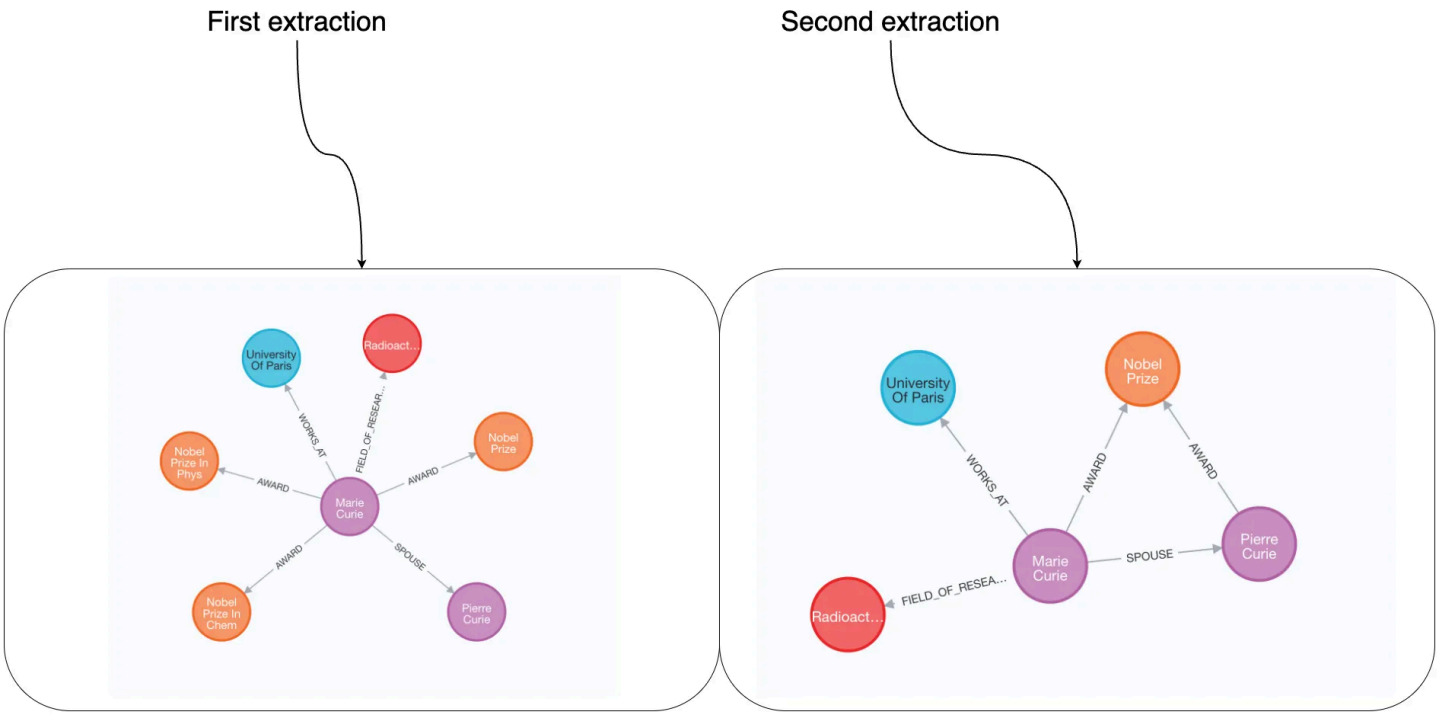
spouse of Pierre, and working at the University of Paris. However, since relationships are specified as a general list without restrictions on which nodes they can connect, some variation still occurs. For instance, the `FIELD_OF_RESEARCH` relationship might appear between a `Person` and a `ResearchField`, but sometimes it links an `Award` to a `ResearchField`. Additionally, since relationship directions aren't defined, there may be differences in directional consistency.

To address the issues of not being able to specify which nodes a relationship can connect and enforcing relationship direction, we recently introduced a new option for defining relationships, as shown below.

```
allowed_nodes = ["Person", "Organization", "Location", "Award", "ResearchField"
allowed_relationships = [
    ("Person", "SPOUSE", "Person"),
    ("Person", "AWARD", "Award"),
    ("Person", "WORKS_AT", "Organization"),
    ("Organization", "IN_LOCATION", "Location"),
    ("Person", "FIELD_OF_RESEARCH", "ResearchField")
]
rels_defined = LLMGraphTransformer(
  llm=llm,
  allowed_nodes=allowed_nodes,
  allowed_relationships=allowed_relationships
)
data = await rels_defined.aconvert_to_graph_documents(documents)
```

Rather than defining relationships as a simple list of strings, we now use a three-element tuple format, where the elements represents the source node, relationship type, and target node, respectively.

Let's visualize the results again.



Visualization of two extraction passes with predefined node and advanced relationship types. Image by author.

Using the three-tuple approach provides a much more consistent schema for the extracted graph across multiple executions. However, given the nature of LLMs, there may still be some variation in the level of detail extracted. For instance, on the right side, Pierre is shown as winning the Nobel Prize, while on the left, this information is missing.

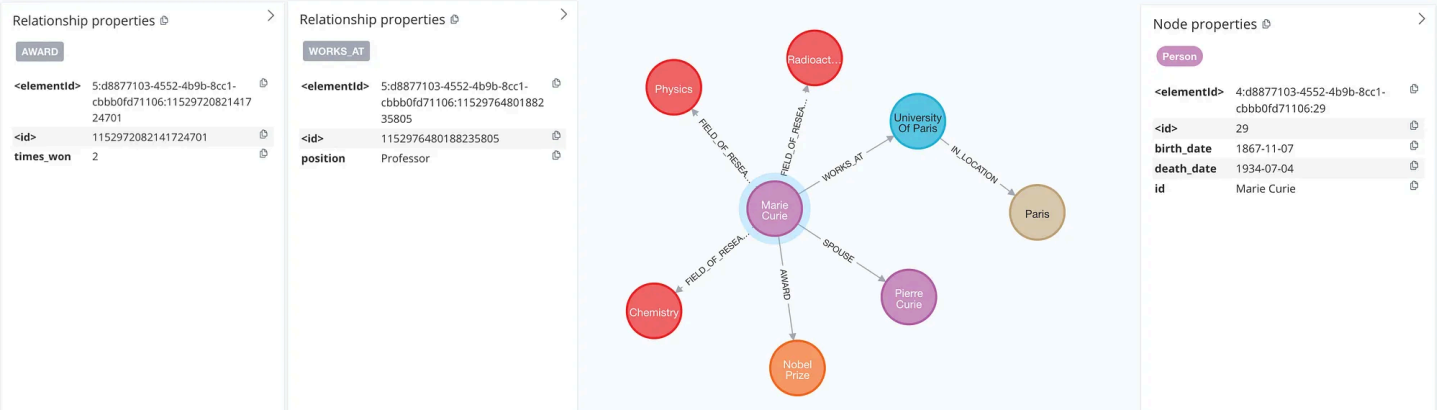
Defining properties

The final enhancement we can make to the graph schema is to define properties for nodes and relationships. Here, we have two options. The first is setting either `node_properties` or `relationship_properties` to `true` allows the LLM to autonomously decide which properties to extract.

```
allowed_nodes = ["Person", "Organization", "Location", "Award", "ResearchField"
allowed_relationships = [
    ("Person", "SPOUSE", "Person"),
    ("Person", "AWARD", "Award"),
    ("Person", "WORKS_AT", "Organization"),
    ("Organization", "IN_LOCATION", "Location"),
    ("Person", "FIELD_OF_RESEARCH", "ResearchField")
]
node_properties=True
relationship_properties=True
props_defined = LLMGraphTransformer(
```

```
    llm=llm,
    allowed_nodes=allowed_nodes,
    allowed_relationships=allowed_relationships,
    node_properties=node_properties,
    relationship_properties=relationship_properties
)
data = await props_defined.aconvert_to_graph_documents(documents)
graph.add_graph_documents(data)
```

Let's examine the results.



Extracted node and relationship properties. Image by author.

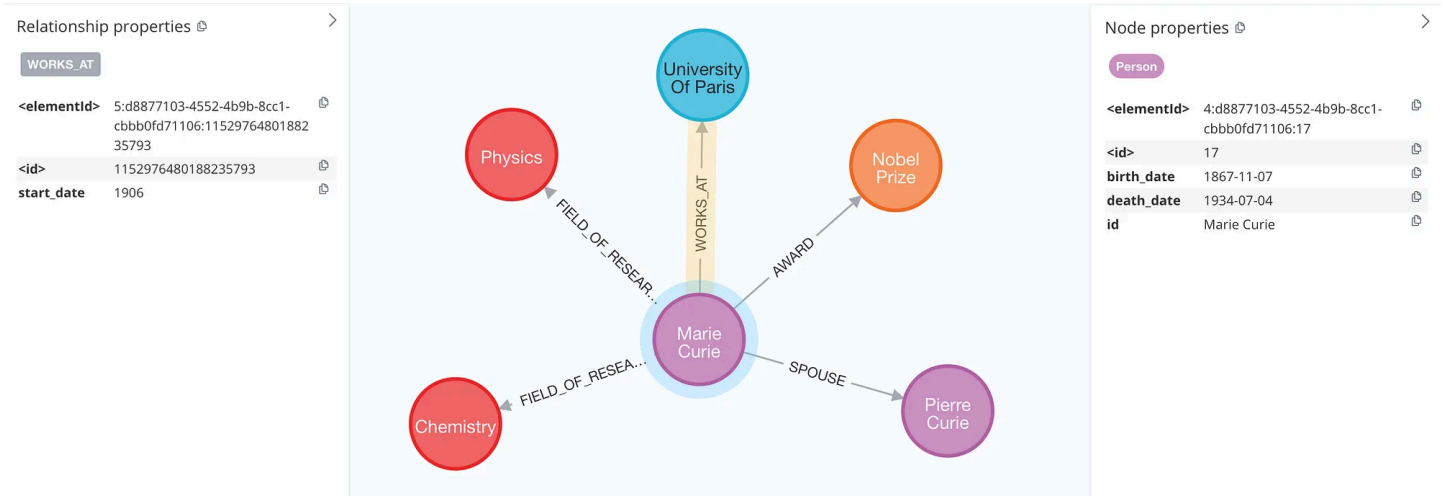
We've enabled the LLM to add any node or relationship properties it considers relevant. For instance, it chose to include Marie Curie's birth and death dates, her role as a professor at the University of Paris, and the fact that she won the Nobel Prize twice. These additional properties significantly enrich the extracted information.

The second option we have is to define the node and relationship properties we want to extract.

```
allowed_nodes = ["Person", "Organization", "Location", "Award", "ResearchField"
allowed_relationships = [
    ("Person", "SPOUSE", "Person"),
    ("Person", "AWARD", "Award"),
    ("Person", "WORKS_AT", "Organization"),
    ("Organization", "IN_LOCATION", "Location"),
    ("Person", "FIELD_OF_RESEARCH", "ResearchField")
]
node_properties=["birth_date", "death_date"]
relationship_properties=["start_date"]
props_defined = LLMGraphTransformer(
```

```
    llm=llm,
    allowed_nodes=allowed_nodes,
    allowed_relationships=allowed_relationships,
    node_properties=node_properties,
    relationship_properties=relationship_properties
)
data = await props_defined.aconvert_to_graph_documents(documents)
graph.add_graph_documents(data)
```

The properties are simply defined as two lists. Let's see what the LLM extracted.



Extracted predefined node and relationship properties. Image by author.

The birth and death dates remain consistent with the previous extraction. However, this time, the LLM also extracted the start date of Marie's professorship at the University of Paris.

Properties indeed add valuable depth to the extracted information, though there are currently some limitations in this implementation:

- Properties can only be extracted using the tool-based approach.
- All properties are extracted as strings.
- Properties can only be defined globally, not per node label or relationship type.
- There is no option to customize property descriptions to guide the LLM for more precise extraction.

Strict mode

If you thought we had perfected a way to make the LLM follow the defined schema flawlessly, I have to set the record straight. While we invested considerable effort into prompt engineering, it's challenging to get LLM, especially the less performant one, to adhere to instructions with complete accuracy. To tackle this, we introduced a post-processing step, called `strict_mode`, that removes any information not conforming to the defined graph schema, ensuring cleaner and more consistent results.

By default, `strict_mode` is set to `True`, but you can disable it with the following code:

```
LLMGraphTransformer(
  llm=llm,
  allowed_nodes=allowed_nodes,
  allowed_relationships=allowed_relationships,
  strict_mode=False
)
```

With strict mode turned off, you may get node or relationship types outside the defined graph schema, as LLMs can sometimes take creative liberties with output structure.

Importing graph documents into graph database

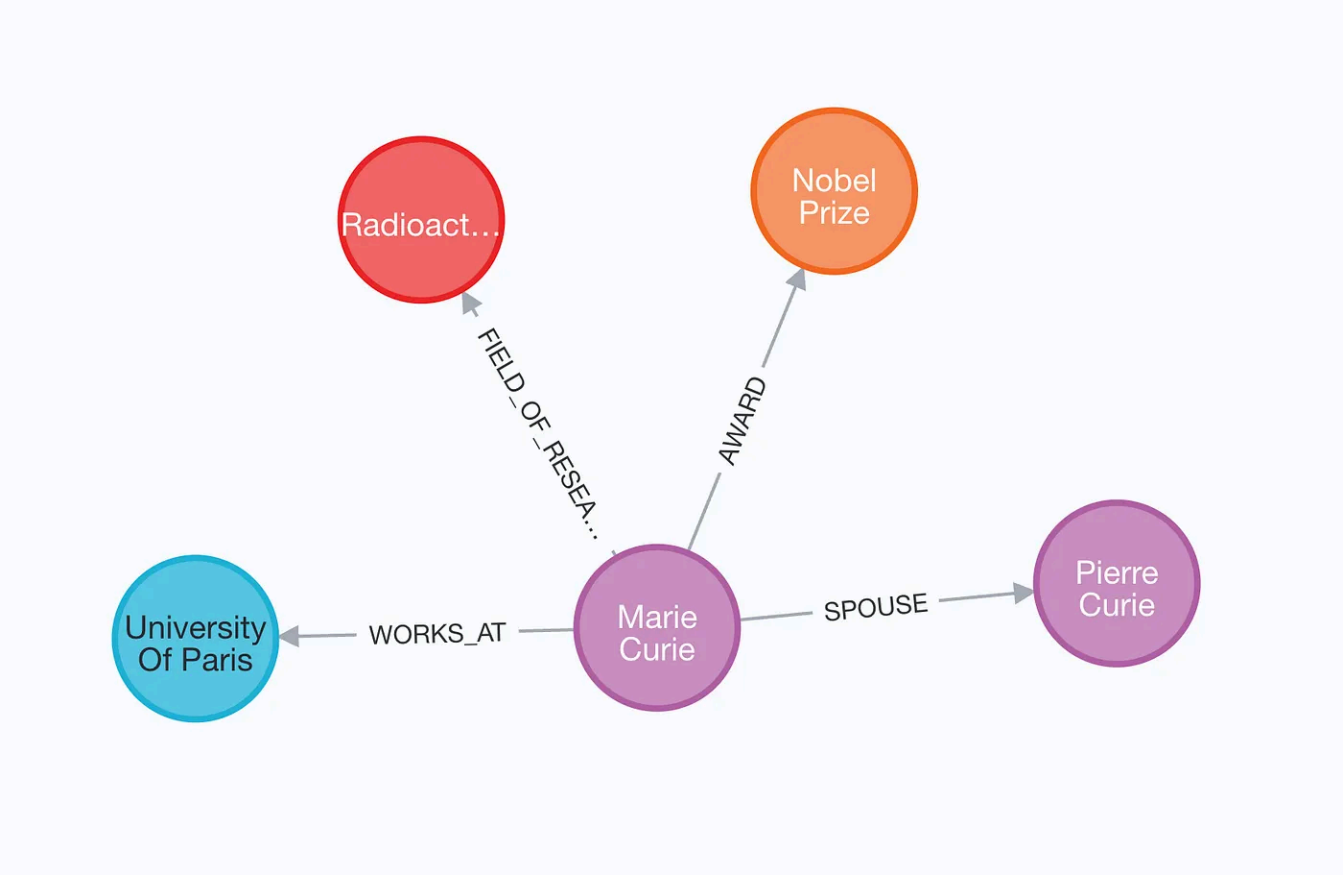
The extracted graph documents from the LLM Graph Transformer can be imported into graph databases like Neo4j for further analysis and applications using the `add_graph_documents` method. We'll explore different options for importing this data to suit different use cases.

Default import

You can import nodes and relationships into Neo4j using the following code.

```
graph.add_graph_documents(graph_documents)
```

This method straightforwardly imports all nodes and relationships from the provided graph documents. We've used this approach throughout the blog post to review the results of different LLM and schema configurations.



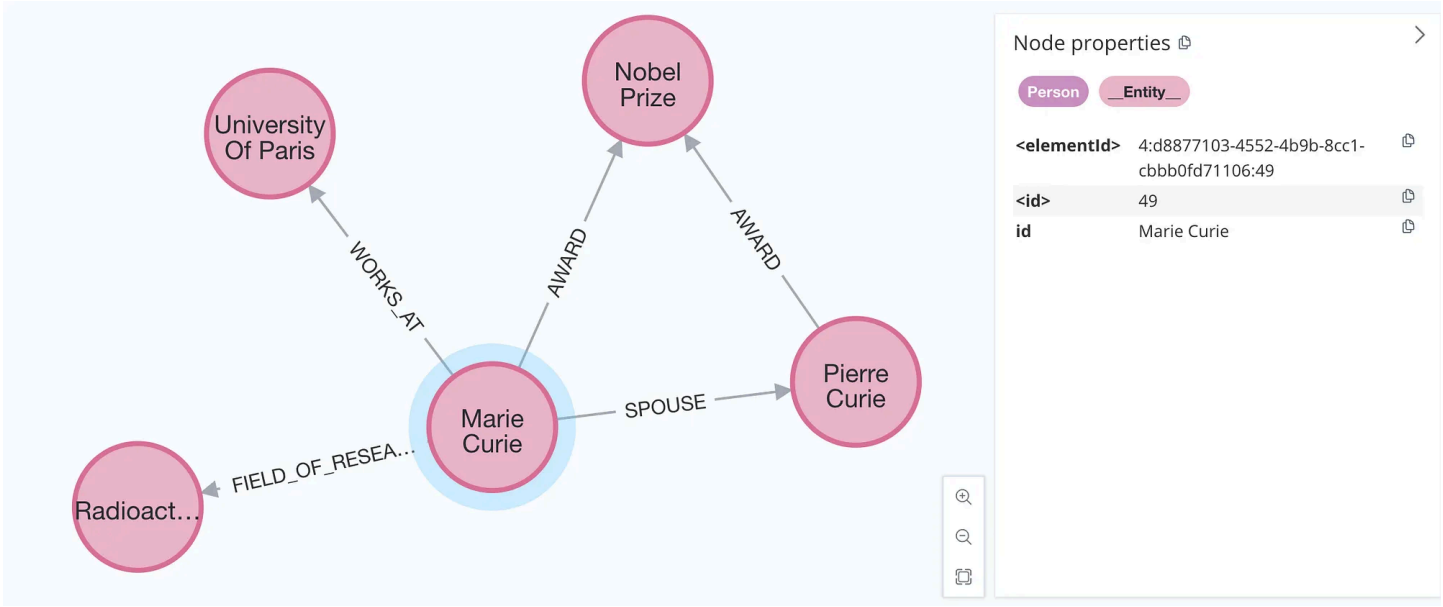
Default import setting. Image by author.

Base entity label

Most graph databases support indexes to optimize data import and retrieval. In Neo4j, indexes can only be set for specific node labels. Since we might not know all the node labels in advance, we can handle this by adding a secondary base label to each node using the `baseEntityLabel` parameter. This way, we can still leverage indexing for efficient importing and retrieval without needing an index for every possible node label in the graph.

```
graph.add_graph_documents(graph_documents, baseEntityLabel=True)
```

As mentioned, using the `baseEntityLabel` parameter will result in each node having an additional `__Entity__` label.



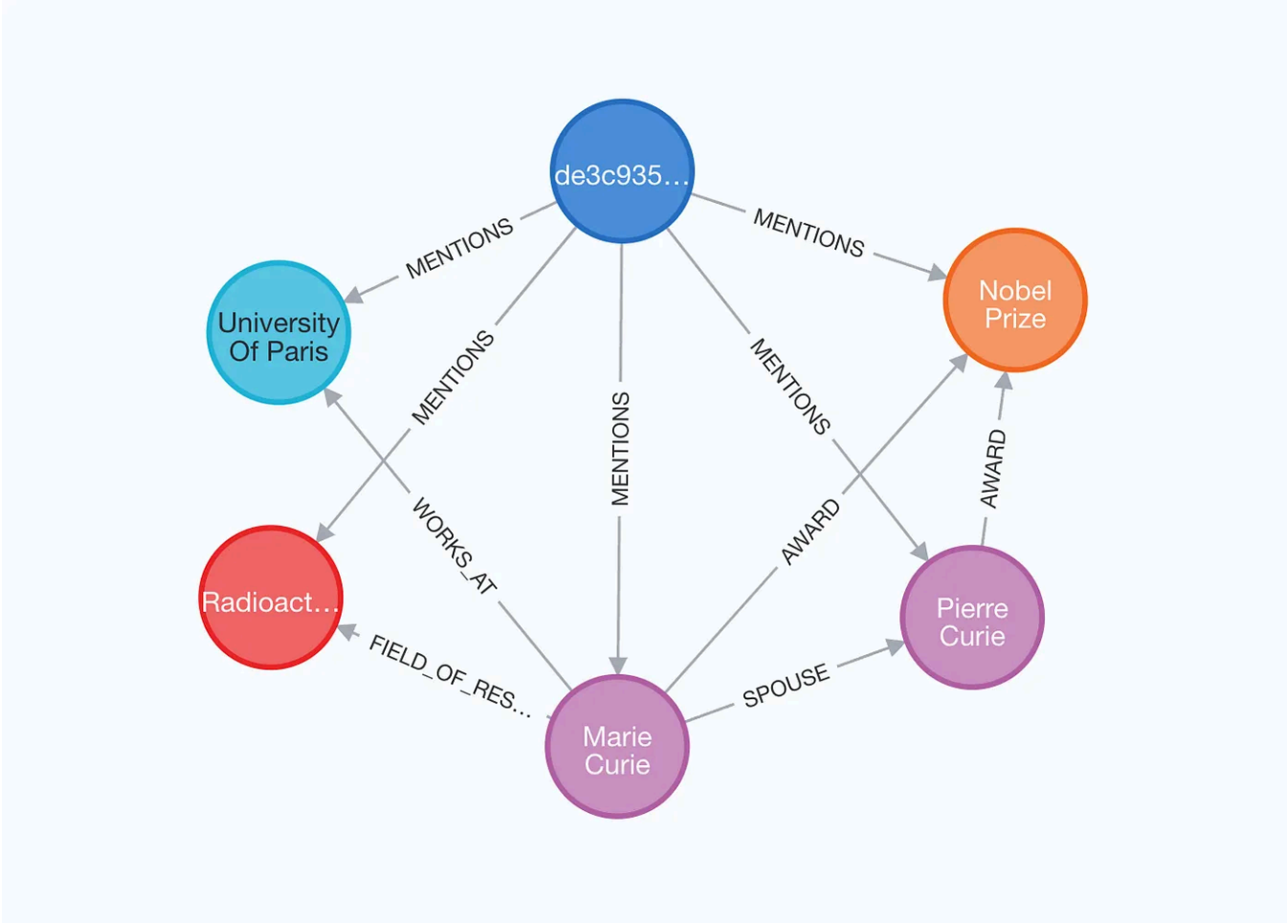
Each node gets a secondary label using the baseEntityLabel parameter. Image by author.

Include source documents

The final option is to also import the source documents for the extracted nodes and relationships. This approach lets us track which documents each entity appeared in. You can import the source documents using the `include_source` parameter.

```
graph.add_graph_documents(graph_documents, include_source=True)
```

Upon inspecting the imported graph, we should see a result similar to this.



Imported source document. Image by author.

In this visualization, the source document is highlighted in blue, with all entities extracted from it connected by `MENTIONS` relationships. This mode allows you to build retrievers that utilize both structured and unstructured search approaches.

Summary

In this post, we explored LangChain's LLM Graph Transformer and its dual modes for building knowledge graphs from text. The tool-based mode, our primary approach, leverages structured output and function calling, which reduces prompt engineering and allows for property extraction. Meanwhile, the prompt-based mode is useful when tools aren't available, relying on few-shot examples to guide the LLM. However, prompt-based extraction does not support property extraction and also yields no isolated nodes.

We observed that defining a clear graph schema, including allowed node and relationship types, improves extraction consistency and performance. A constrained schema helps ensure that the output adheres to our desired structure, making it more predictable, reliable, and applicable. Whether using tools or prompts, the LLM Graph Transformer enables more organized, structured

representations of unstructured data, enabling better RAG applications and multi-hop query handling.

The code is available on GitHub. You can also try out the LLM Graph Transformer in a no-code environment using Neo4j's hosted **LLM Graph Builder** application.

Neo4j graph builder

No-code

llm-graph-builder.neo4jlabs.com

Neo4j | Llm | Langchain | Knowledge Graph | Hands On Tutorials

Data Science

Following

Published in TDS Archive

829K followers · Last published Feb 3, 2025

An archive of data science, data analytics, data engineering, machine learning, and artificial intelligence writing from the former Towards Data Science Medium publication.

Following

Written by Tomaz Bratanic

10K followers · 27 following

Data explorer. Turn everything into a graph. Author of Graph algorithms for Data Science at Manning publication. http://mng.bz/GGVN

Responses (16)

Andrzej Wodecki

What are your thoughts?

Rainer Kempkes
Nov 5, 2024

Hi Tomaz, thank you for sharing your insights with building knowledge graphs with Neo4j out of unstructured text!

My problem is that I want to create a KB out of a German legal domain corpus and I can't define all possible nodes or relationships... more

24 2 replies Reply

Fernando Abolafio
Nov 6, 2024

How does RAG perform better in a graph knowledge base vs in a Vector Store?
Can you recommend some sources to check what's the pros/cons of one or the other?

I'd also expect the graph based approach to be way more expensive to build than... more

7 1 reply Reply

Munaf he/him
Nov 9, 2024

Off topic but, how would you update your system to remember the unrelated data about "Robin Williams"? Yes, it's not relevant to this knowledge graph, but creativity is taking adding irrelevant into the picture, and forcing new perspectives.

15 Reply

See all responses

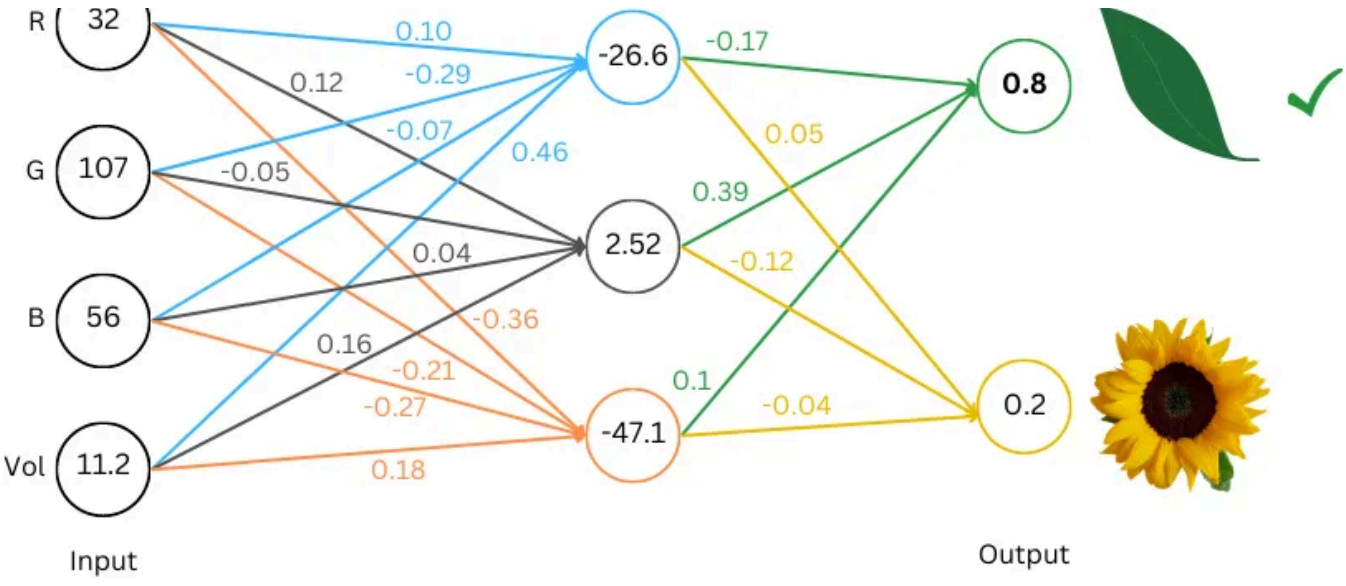
More from Tomaz Bratanic and TDS Archive

In Neo4j Developer Blog by Tomaz Bratanic

Enhancing the Accuracy of RAG Applications With Knowledge Graphs

A practical guide to constructing and retrieving information from knowledge graphs in RAG applications with Neo4j and LangChain

Mar 30, 2024 797 9

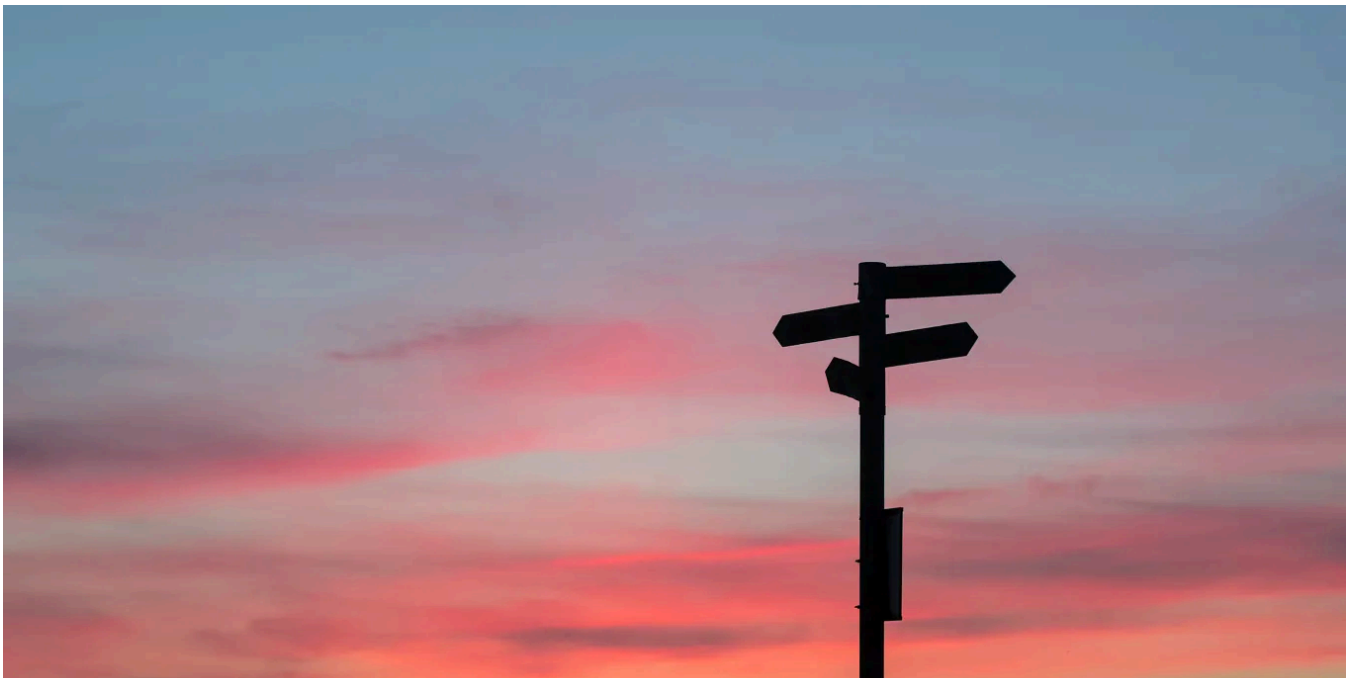


In TDS Archive by Rohit Patel

Understanding LLMs from Scratch Using Middle School Math

In this article, we talk about how LLMs work, from scratch—assuming only that you know how to add and multiply two numbers. The article...

Oct 19, 2024 7.9K 95

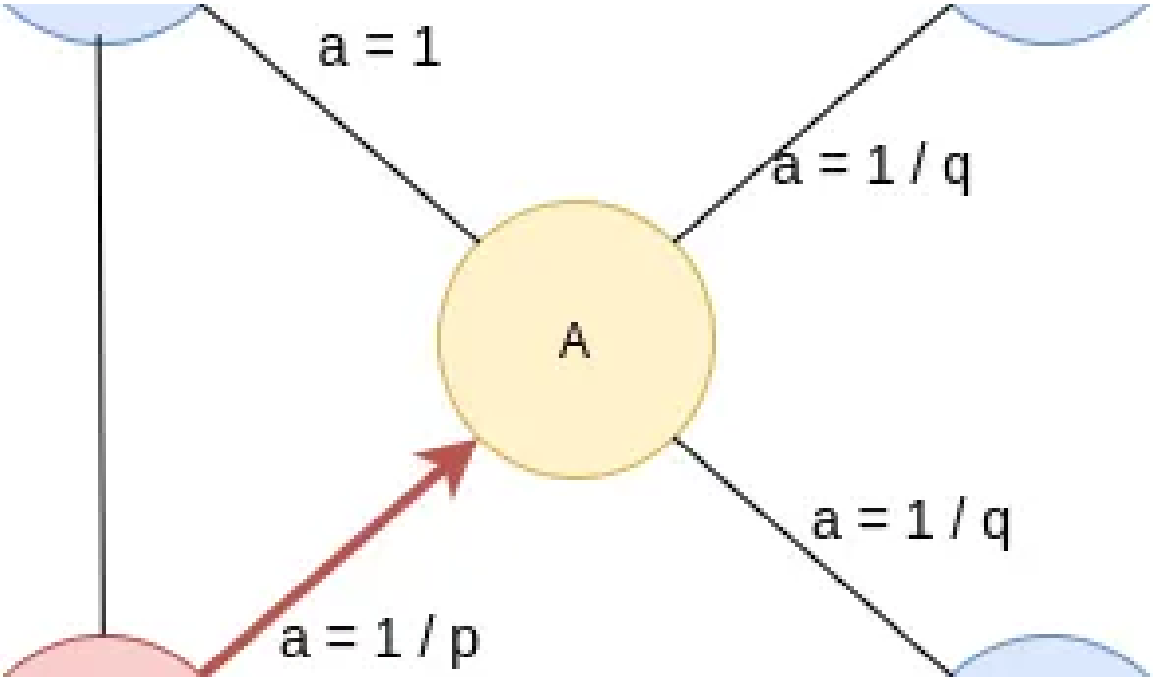


In TDS Archive by Francesco Casalegno

Recommender Systems—A Complete Guide to Machine Learning Models

Leveraging data to help users discovering new contents

Nov 25, 2022 537 5



In TDS Archive by Tomaz Bratanic

Complete guide to understanding Node2Vec algorithm

An in-depth guide to understanding node2vec algorithm and its hyper-parameters

Aug 16, 2021 428 3

See all from Tomaz Bratanic

See all from TDS Archive

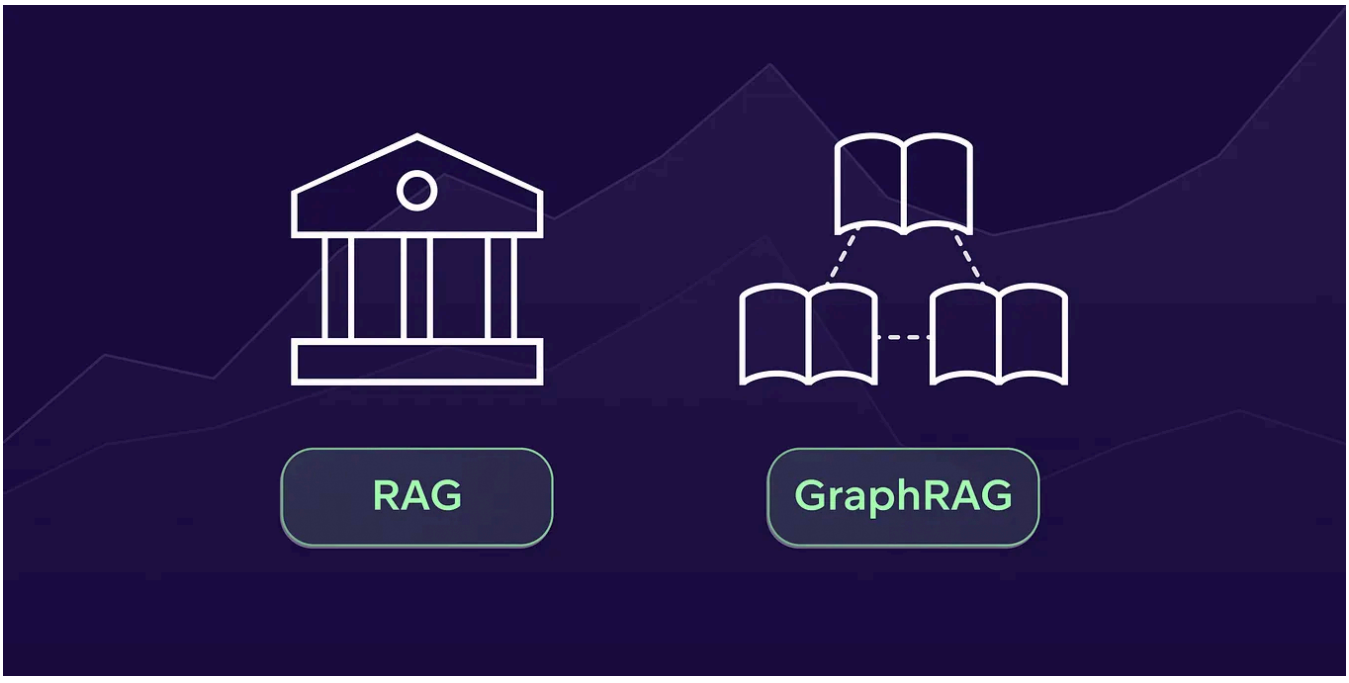
Recommended from Medium

In Level Up Coding by Fareed Khan

Converting Unstructured Data into a Knowledge Graph Using an End-to-End Pipeline

Step by Step guide

Apr 17 2.5K 47

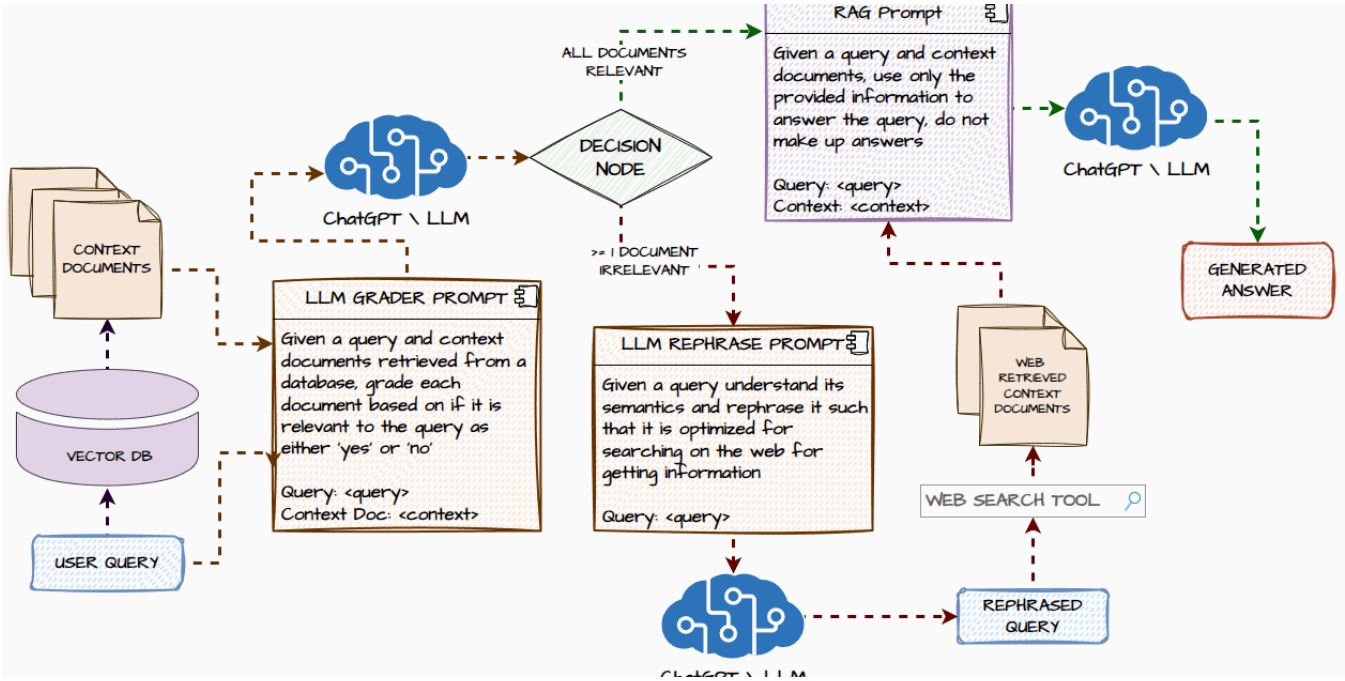


In GoodData Developers by Marcelo G. Almiron

From RAG to GraphRAG: Knowledge Graphs, Ontologies and Smarter AI

Modern AI chatbots often rely on Retrieval-Augmented Generation (RAG), a technique where the chatbot pulls in external data to ground its...

Sep 2 32 3



In Data Science Collective by Samvardhan Singh

Understanding LangGraph: Creating Agentic AI Systems for Enterprise Applications

As enterprises move toward autonomous operations, they require systems that do more than just answer questions

Apr 9 409 8

Bishal Bose

Building a Simple RAG System from Scratch: A Comprehensive Guide

Photo by Steve Johnson on Unsplash

Apr 25 677

In GoPenAI by Subrata Samanta

Hybrid Graph RAG: Harnessing Graph and Vector for Financial Analysis

Every year, companies like Apple publish annual reports packed with valuable insights—financial data, strategic initiatives...

May 1 1

In Towards AI by Mariana Avelino

How GraphRAG Works Step-by-Step

Perhaps you've come across the paper From Local to Global: A GraphRAG Approach to Query-Focused Summarization, which is Microsoft...

May 6 1.1K 10

See more recommendations