Build a knowledge graph-based agent with Llama 3.1, NVIDIA NIM, and LangChain

Leverage Llama-3.1 native function-calling capabilities to retrieve structured data from a knowledge graph to power your RAG applications

Tomaz Bratanic · Follow

8 min read · 2 days ago

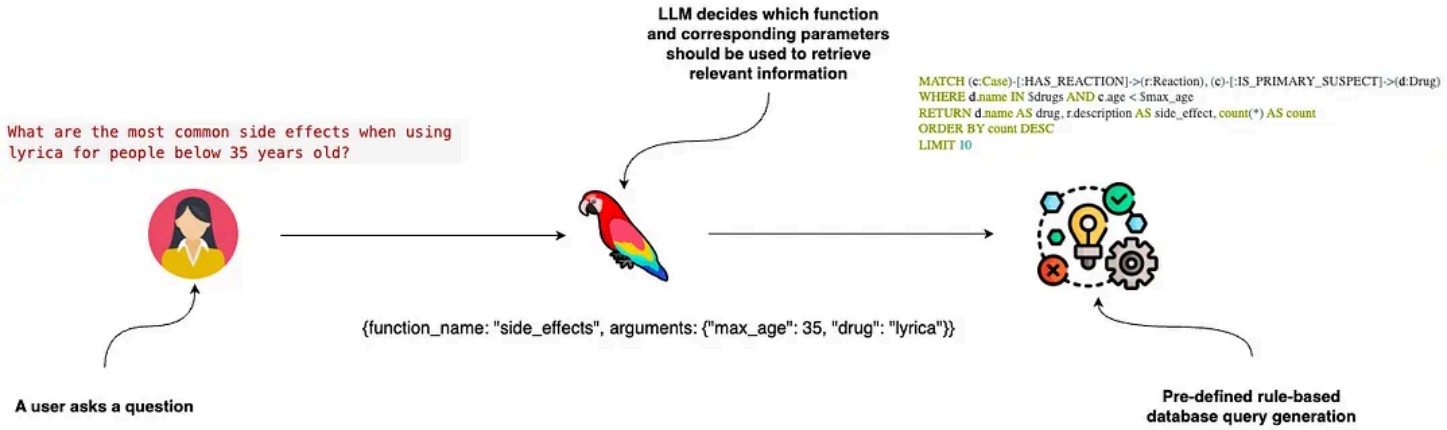
168 1

Created using ChatGPT

While most people focus on RAG over unstructured text, such as company documents or documentation, I am pretty bullish on retrieval systems over structured information, particularly knowledge graphs. There has been a lot of excitement about GraphRAG, specifically Microsoft's implementation. However, in their implementation, the input data is unstructured text in the form of documents, which is transformed into a knowledge graph using a Large Language Model (LLM).

In this blog post, we will show how to implement a retriever over a knowledge graph containing structured information from FDA Adverse Event Reporting System (FAERS), which offers the information about drug adverse events. If you have ever tinkered with knowledge graphs and retrieval, your first thought might be to use an LLM to generate database queries to retrieve relevant information from a knowledge graph to answer a given question. However, database query generation using LLMs is still evolving and may not yet offer the most consistent or robust solution. So, what are the viable alternatives at the moment?

In my opinion, the best solution at the moment is the so-called dynamic query generation. Rather than relying entirely on an LLM to generate the complete query, this method employs a logic layer that deterministically generates a database query from predefined input parameters.his solution can be implemented using an LLM with function-calling support. The advantage of using a function-calling feature lies in the ability to define to an LLM how it should prepare a structured input to a function. This approach ensures that the query generation process is controlled and consistent while allowing for user input flexibility.



Dynamic query generation flow. Image by author.

The image illustrates a process of a understanding a user's question to retrieve specific information. The flow involves three main steps:

1. A user asks a question about common side effects of the drug Lyrica for people under 35 years old.
2. The LLM decides which function to call and the parameters needed. In this example, it chose a function named "side_effects" with parameters including the drug "Lyrica" and a maximum age of 35.
3. The identified function and parameters are used to deterministically and dynamically generate a database query (Cypher) statement to retrieve relevant information.

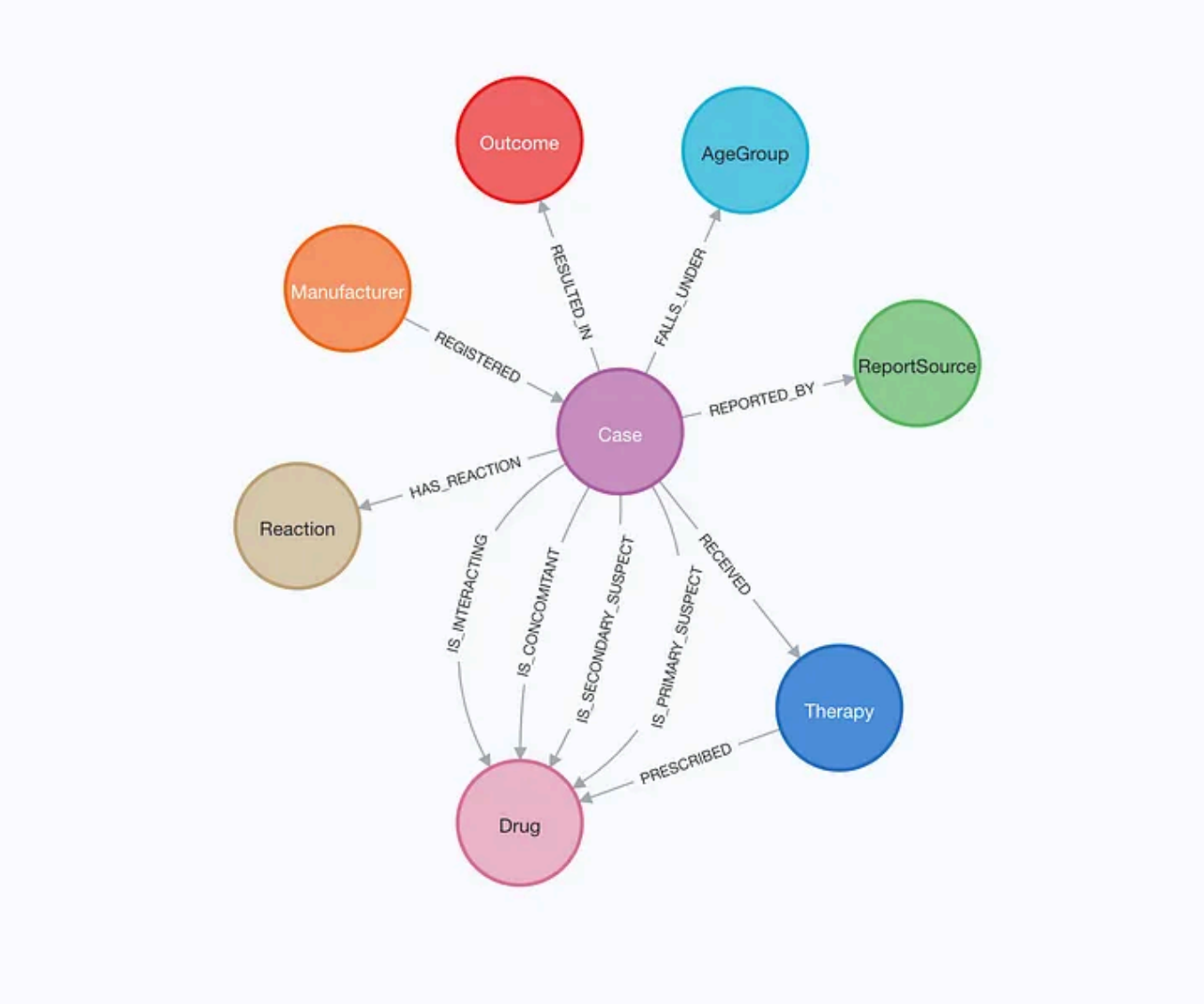
Function-calling support is vital for advanced LLM use cases, such as allowing LLMs to use multiple retrievers based on user intent or building multi-agent flows. I have written some articles using commercial LLMs with native function-calling support. However, in this blog post, we will use Llama-3.1, a superior open-source LLM with native function-calling support that was released just recently.

The code is available on GitHub.

Setting up the knowledge graph

We will use Neo4j, which is a native graph database to store the adverse event information. You can set up a free cloud Sandbox project that comes with pre-populated FAERS by following this link.

The instantiated database instance has a graph with the following schema.



Adverse events graph schema. Image by author

The schema centers on the Case node, which links various aspects of a drug safety report, including the drugs involved, reactions experienced, outcomes, and therapies prescribed. Each drug is characterized by whether it is primary, secondary, concomitant, or interacting. Cases are also associated with information about the manufacturer, the age group of the patient, and the source of the report. This schema allows for tracking and analyzing the relationships between drugs, their reactions, and outcomes in a structured manner.

We'll start by creating a connection to the database by instantiating a Neo4jGraph object.

```
os.environ["NEO4J_URI"] = "bolt://18.206.157.187:7687"
os.environ["NEO4J_USERNAME"] = "neo4j"
os.environ["NEO4J_PASSWORD"] = "elevation-reservist-thousands"

graph = Neo4jGraph(refresh_schema=False)
```

Setting up the LLM environment

There are many options to host open-source LLMs like the Llama-3.1. In this blog post, we will use the NVIDIA API catalog, which provides NVIDIA NIM inference microservices and supports function calling for Llama 3.1 models. When you create an account you get 1000 tokens, which is more than enough to complete this blog post. You'll need to create an API key and copy it to the notebook.

```
os.environ["NVIDIA_API_KEY"] = "nvapi-"
llm = ChatNVIDIA(model="meta/llama-3.1-70b-instruct")
```

We'll be using the **llama-3.1–70b** as the 8b version has some hiccups with optional parameters in function definitions.

The nice thing about NVIDIA NIM microservices is that you can easily host them locally if you have security or other concerns, so it's really easily swappable and you only need to add a url parameter to the LLM configuration.

```
# connect to an local NIM running at localhost:8000,
# specifying a specific model
llm = ChatNVIDIA(
  base_url="http://localhost:8000/v1",
  model="meta/llama-3.1-70b-instruct"
)
```

Tool definition

In this example, we will configure a single tool with four optional parameters. Based on those parameters, we will construct a corresponding Cypher statement that will be used to retrieve the relevant information from the knowledge graph.

Specifically, our tool will be able to identify the most frequent side effects based on input drug, age, and the drug manufacturer.

```
@tool
def get_side_effects(
    drug: Optional[str] = Field(
        description="disease mentioned in the question. Return None if no mentio
    ),
    min_age: Optional[int] = Field(
        description="Minimum age of the patient. Return None if no mentioned."
    ),
    max_age: Optional[int] = Field(
        description="Maximum age of the patient. Return None if no mentioned."
    ),
    manufacturer: Optional[str] = Field(
        description="manufacturer of the drug. Return None if no mentioned."
    ),
):
    """Useful for when you need to find common side effects."""
    params = {}
    filters = []
    side_effects_base_query = """
    MATCH (c:Case)-[:HAS_REACTION]->(r:Reaction), (c)-[:IS_PRIMARY_SUSPECT]->(d:
    """
```

```
    if drug and isinstance(drug, str):
        candidate_drugs = [el["candidate"] for el in get_candidates(drug, "drug"
        if not candidate_drugs:
            return "The mentioned drug was not found"
        filters.append("d.name IN $drugs")
        params["drugs"] = candidate_drugs

    if min_age and isinstance(min_age, int):
        filters.append("c.age > $min_age ")
        params["min_age"] = min_age
    if max_age and isinstance(max_age, int):
        filters.append("c.age < $max_age ")
        params["max_age"] = max_age
    if manufacturer and isinstance(manufacturer, str):
        candidate_manufacturers = [
            el["candidate"] for el in get_candidates(manufacturer, "manufacturer
        ]
        if not candidate_manufacturers:
            return "The mentioned manufacturer was not found"
        filters.append(
            "EXISTS {(c)<-[:REGISTERED]-(:Manufacturer {manufacturerName: $manuf
        )
        params["manufacturer"] = candidate_manufacturers[0]

    if filters:
        side_effects_base_query += " WHERE "
        side_effects_base_query += " AND ".join(filters)
    side_effects_base_query += """
    RETURN d.name AS drug, r.description AS side_effect, count(*) AS count
    ORDER BY count DESC
    LIMIT 10
    """
    print(f"Using parameters: {params}")
    data = graph.query(side_effects_base_query, params=params)
    return data
```

The `get_side_effects` function is designed to retrieve common side effects of drugs from a knowledge graph using specified search criteria. It accepts optional parameters for drug name, patient age range, and drug manufacturer to customize the search. Each parameter has a description that is passed to an LLM along with the function description, enabling the LLM to understand how to use them. The function then constructs a

dynamic Cypher query based on the provided inputs, executes this query against the knowledge graph, and returns the resulting side effects data.

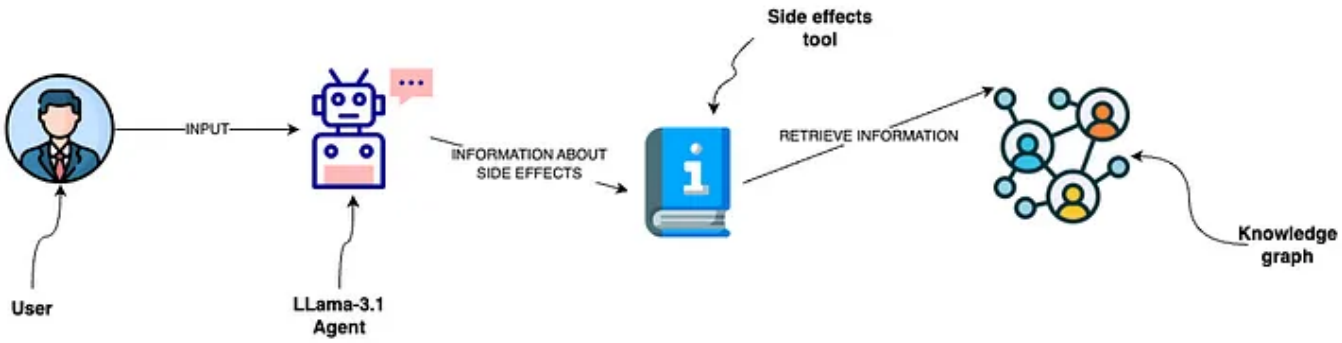
Let's test the function.

```
get_side_effects("lyrica")
# Using parameters: {'drugs': ['LYRICA', 'LYRICA CR']}
# [{'drug': 'LYRICA', 'side_effect': 'Pain', 'count': 32},
#  {'drug': 'LYRICA', 'side_effect': 'Fall', 'count': 21},
# {'drug': 'LYRICA', 'side_effect': 'Intentional product use issue', 'count': 20
# {'drug': 'LYRICA', 'side_effect': 'Insomnia', 'count': 19},
# ...
```

Our tool first mapped the "lyrica" drug mentioned in the question to "['LYRICA', 'LYRICA CR']" values in the knowledge graph and then executed corresponding Cypher statement to find the most frequent side effects.

Graph-based LLM Agent

The only thing left to do is configure an LLM agent that can use the defined tool to answer questions about the drug's side effects.



Agent data flow. Image by author

The image depicts a user interacting with a Llama-3.1 agent to inquire about drug side effects. The agent accesses a side effects tool that retrieves information from a knowledge graph to provide the user with the relevant data.

We'll start by defining the prompt template.

```
prompt = ChatPromptTemplate.from_messages(
    [
        (
            "system",
            "You are a helpful assistant that finds information about common sid
            "If tools require follow up questions, "
            "make sure to ask the user for clarification. Make sure to include a
            "available options that need to be clarified in the follow up questi
            "Do only the things the user specifically requested. ",
        ),
        MessagesPlaceholder(variable_name="chat_history"),
        ("user", "{input}"),
        MessagesPlaceholder(variable_name="agent_scratchpad"),
    ]
)
```

The prompt template includes the system message, optional chat history, and user input. The agent_scratchpad is reserved for the LLM, as it sometimes needs to perform multiple steps to answer the question, like executing and retrieving information from tools.

The LangChain library makes it straightforward to add tools to the LLM by using the `bind_tools` method.

```
tools = [get_side_effects]
llm_with_tools = llm.bind_tools(tools=tools)
agent = (
    {
        "input": lambda x: x["input"],
        "chat_history": lambda x: _format_chat_history(x["chat_history"])
        if x.get("chat_history")
        else [],
        "agent_scratchpad": lambda x: format_to_openai_function_messages(
            x["intermediate_steps"]
        ),
    }
    | prompt
    | llm_with_tools
    | OpenAIFunctionsAgentOutputParser()
)

agent_executor = AgentExecutor(agent=agent, tools=tools, verbose=True).with_type
    input_type=AgentInput, output_type=Output
)
```

The agent processes input through a series of transformations and handlers that format the chat history, apply the LLM with the bound tools, and parse the output. Finally, the agent is set up with an executor that manages the execution flow, specifies input and output types, and includes verbosity settings for detailed logging during execution.

Let's now test the agent.

```
agent_executor.invoke(
    {
        "input": "What are the most common side effects when using lyrica for pe
    }
)
```

Results

```
> Entering new AgentExecutor chain...
```



```
> Finished chain.
{'input': 'What are the most common side effects when using lyrica for people below 35 years
old?',
 'output': 'Based on the available data, the most common side effects of Lyrica for people
below 35 years old include sinusitis, hypoacusis, fall, brain injury, amnesia, off-label
use, impaired quality of life, somnolence, and drug ineffective for unapproved indication.'}
```

Agent execution. Image by author.

The LLM identified it needs to use the `get_side_effects` function with appropriate arguments. The function then dynamically generates a Cypher statement, fetches the relevant information, and returns it to the LLM to generate the final answer.

Summary

Function calling capabilities are a powerful addition to open-source models like Llama 3.1, enabling more structured and controlled interactions with external data sources and tools. Beyond just querying unstructured documents, graph-based agents offer exciting possibilities for interacting with knowledge graphs and structured data. The ease of hosting these models using platforms like NVIDIA NIM microservices makes them increasingly accessible.

As always, the code is available on GitHub.

Neo4j Graph Llm Nvidia Nemo

Written by Tomaz Bratanic

Follow

8.6K Followers

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

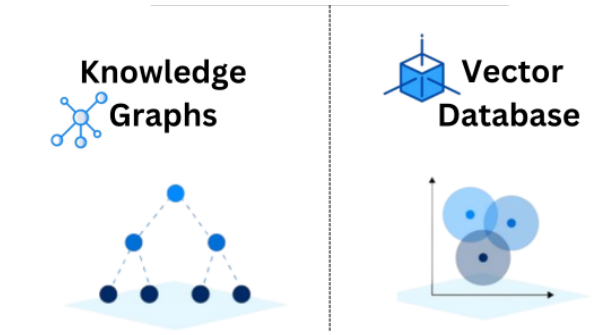
More from Tomaz Bratanic

Tomaz Bratanic in Neo4j Developer Blog

Implementing 'From Local to Global' GraphRAG with Neo4j and...

Combine text extraction, network analysis, and LLM prompting and summarization for...

Jul 9 455 8

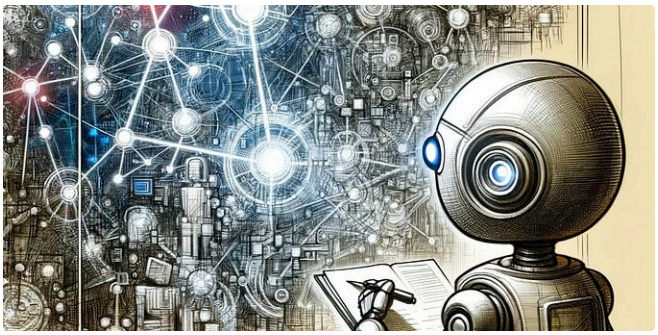


Chia Jeng Yang in Neo4j Developer Blog

Graph vs. Vector RAG—Benchmarking, Optimization...

Differences in graph & vector search, Benchmarks and understanding how depth ...

Jun 5 395 1

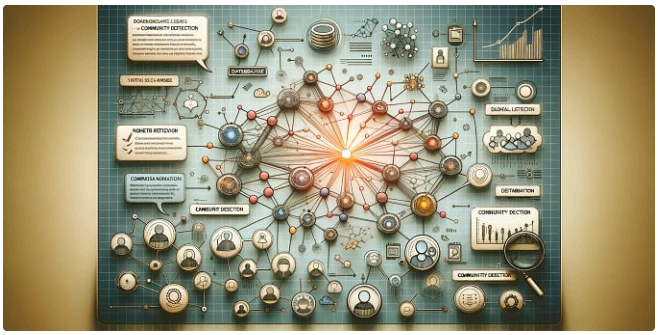


Tomaz Bratanic in Neo4j Developer Blog

Enhancing the Accuracy of RAG Applications With Knowledge...

A practical guide to constructing and retrieving information from knowledge...

Mar 30 627 7



Tomaz Bratanic in Towards Data Science

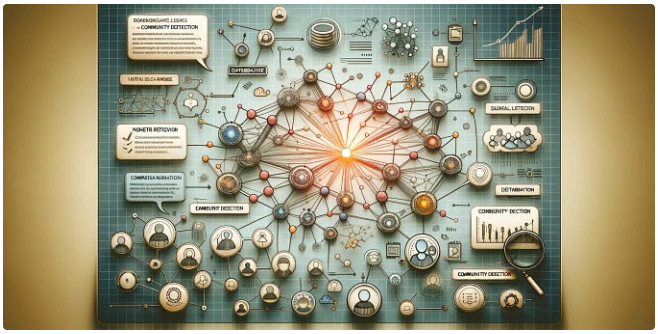
Integrating Microsoft GraphRAG into Neo4j

Store the MSFT GraphRAG output into Neo4j and implement local and global retrievers...

5d ago 448 5

See all from Tomaz Bratanic

Recommended from Medium

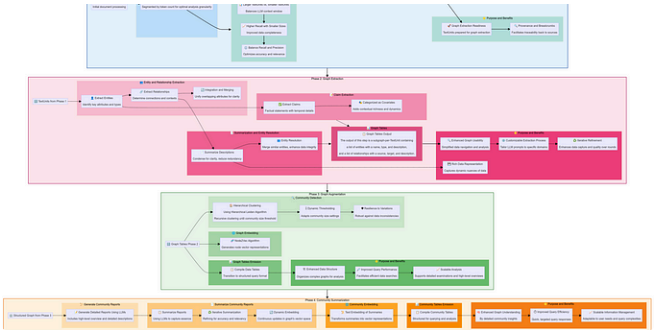


Tomaz Bratanic in Towards Data Science

Integrating Microsoft GraphRAG into Neo4j

Store the MSFT GraphRAG output into Neo4j and implement local and global retrievers...

5d ago 448 5



Dhruv Rathi

GraphRAG: Redefining Knowledge Extraction with Graphs

Deep dive into building a Seamless Knowledge Graph from Distributed Data...

Jul 23 180 1

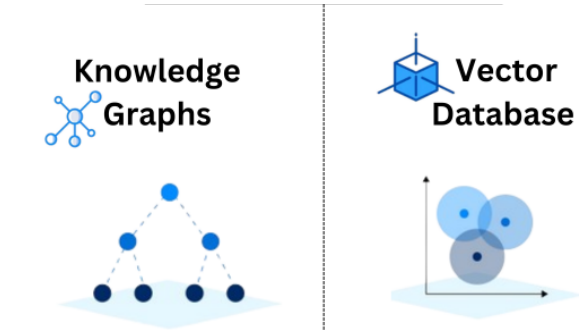
Lists

Natural Language Processing

1618 stories · 1186 saves

ChatGPT prompts

48 stories · 1861 saves

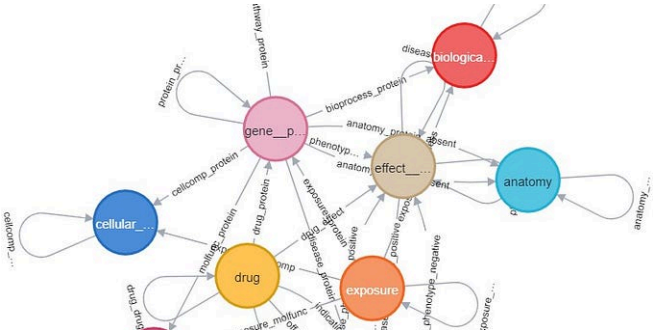


Chia Jeng Yang in Neo4j Developer Blog

Graph vs. Vector RAG—Benchmarking, Optimization...

Differences in graph & vector search, Benchmarks and understanding how depth ...

Jun 5 395 1

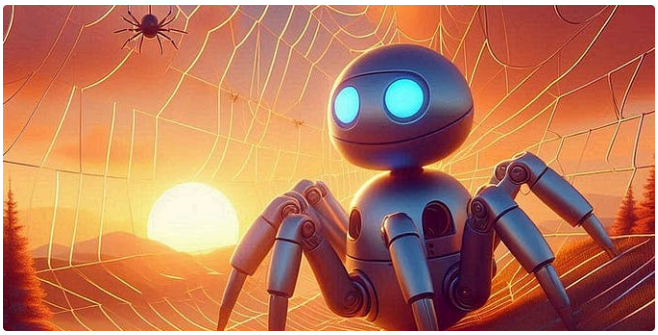


TRAN Ngoc Thach

Exploring PrimeKG—A Knowledge Graph for Medicine & Healthcare

This article will take a glance at the paper "Building a knowledge graph to enable...

Jul 23 172

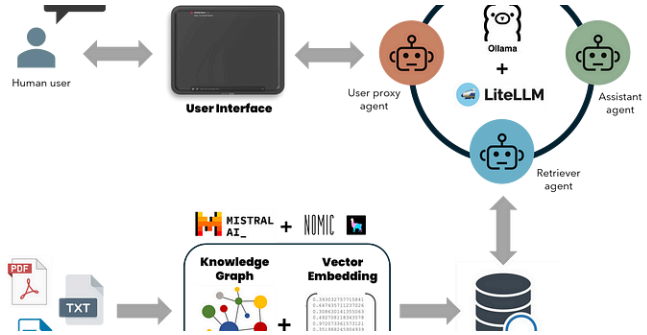


Valentina Alto in Microsoft Azure

Introducing GraphRAG with LangChain and Neo4j

Part 2: Implementing and evaluating a GraphRAG application

May 12 308 4



Karthik Rajan, Ph.D in AI Advances

Microsoft's GraphRAG + AutoGen + Ollama + Chainlit = Fully Local &...

This superbot app integrates GraphRAG with AutoGen agents, powered by local LLMs fro...

Jul 14 754 10

See more recommendations