

Enhancing Interaction between Language Models and Graph Databases via a Semantic Layer

Provide an LLM agent with a suite of robust tools it can use to interact with a graph database



Tomaz Bratanic · [Follow](#)

Published in Towards Data Science · 11 min read · Jan 18, 2024



606

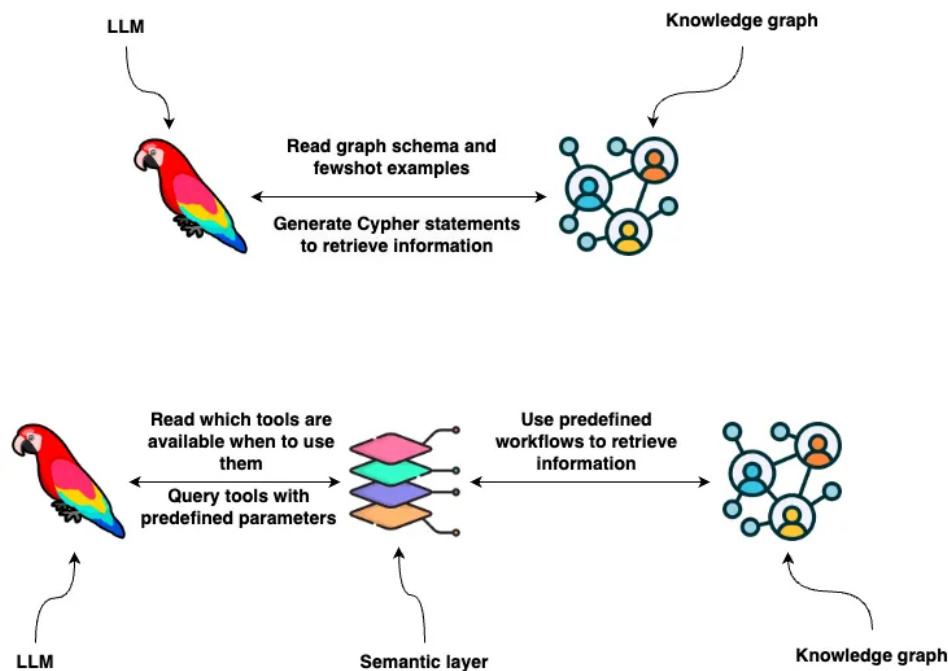


6



Knowledge graphs provide a great representation of data with flexible data schema that can store structured and unstructured information. You can use Cypher statements to retrieve information from a graph database like Neo4j. One option is to use LLMs to generate Cypher statements. While that option provides excellent flexibility, the truth is that base LLMs are still brittle at consistently generating precise Cypher statements. Therefore, we need to look for an alternative to guarantee consistency and robustness. What if, instead of developing Cypher statements, the LLM extracts parameters from user input and uses predefined functions or Cypher templates based on the user intent? In short, you could provide the LLM with a set of predefined tools and instructions on when and how to use them based on the user input, which is also known as the semantic layer.

Top highlight



Semantic layer is an intermediate step that provides additional accuracy and robust way of LLMs interacting with a Knowledge graph. Image by the author. Inspired by [this image](#).

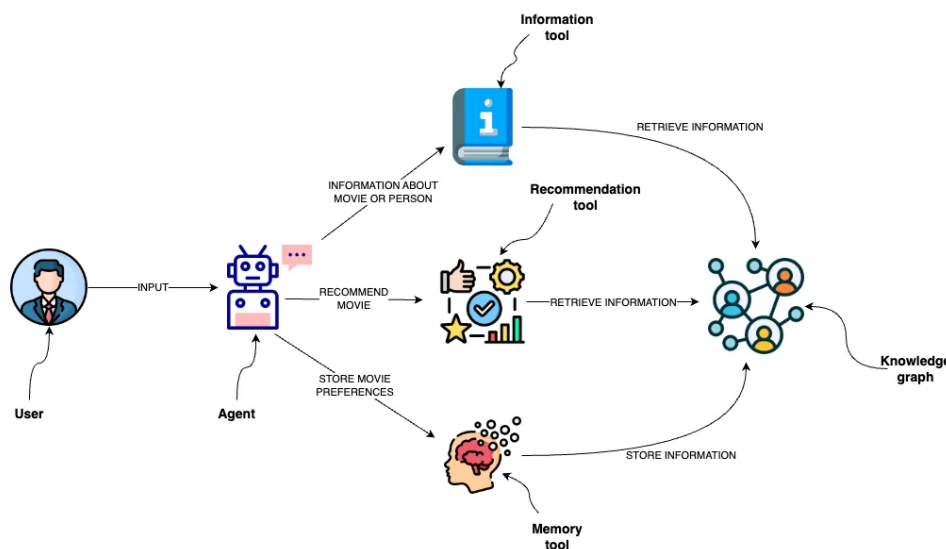
A semantic layer consists of various tools exposed to an LLM that it can use to interact with a knowledge graph. They can be of various complexity. You can think of each tool in a semantic layer as a function. For example, take a look at the following function.

```
def get_information(entity: str, type: str) -> str:
    candidates = get_candidates(entity, type)
    if not candidates:
        return "No information was found about the movie or person in the database"
    elif len(candidates) > 1:
        newline = "\n"
        return (
            "Need additional information, which of these "
            f"did you mean: {newline + newline.join(str(d) for d in candidates)}"
        )
    data = graph.query(
        description_query, params={"candidate": candidates[0]["candidate"]}
    )
    return data[0]["context"]
```

The tools can have multiple input parameters, like in the above example, which allows you to implement complex tools. Additionally, the workflow can consist of more than a database query, allowing you to handle any edge cases or exceptions as you see fit. The advantage is that you turn prompt engineering problems, which might work most of the time, into code engineering problems, which work every time exactly as scripted.

Movie agent

In this blog post, we will demonstrate how to implement a semantic layer that allows an LLM agent to interact with a knowledge graph that contains information about actors, movies, and their ratings.



Movie agent architecture. Image by the author.

Taken from the documentation (also written by me):

The agent utilizes several tools to interact with the Neo4j graph database effectively.

** **Information tool:** Retrieves data about movies or individuals, ensuring the agent has access to the latest and most relevant information.*

** **Recommendation Tool:** Provides movie recommendations based upon user preferences and input.*

** **Memory Tool:** Stores information about user preferences in the knowledge graph, allowing for a personalized experience over multiple interactions.*

An agent can use information or recommendation tools to retrieve information from the database or use the memory tool to store user preferences in the database.

Predefined functions and tools empower the agent to orchestrate intricate user experiences, guiding individuals towards specific goals or delivering tailored information that aligns with their current position within the user journey.

This predefined approach enhances the robustness of the system by reducing the artistic freedom of an LLM, ensuring that responses are more structured and aligned with predetermined user flows, thereby improving

the overall user experience.

The semantic layer backend of a movie agent is implemented and available as a [LangChain template](#). I have used this template to build a simple streamlit chat application.

Movie agent



What's a good comedy to watch this weekend?



✓ Finished!

Invoking: `Recommender` with `{'genre': 'Comedy'}`

Here are some comedy movies you might enjoy:

1. The Secret Life of Pets
2. Ghostbusters
3. Keanu

Let me know if you need more information about these movies.

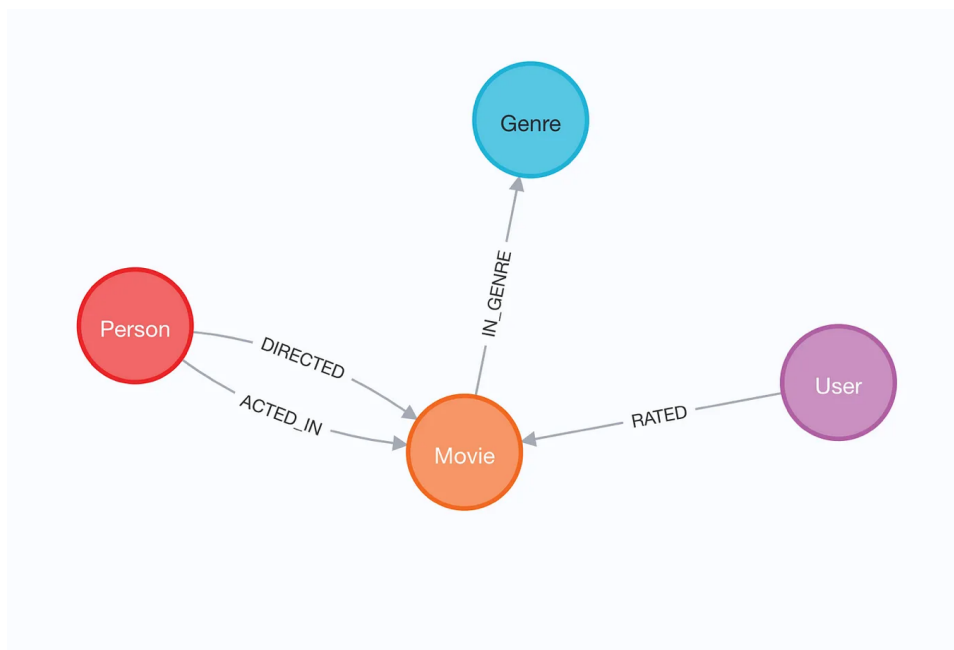
Streamlit chat interface. Image by the author.

Code is available on [GitHub](#). You can start the project by defining environment variables and executing the following command:

```
docker-compose up
```

Graph model

The graph is based on the [MovieLens](#) dataset. It contains information about actors, movies, and 100k user ratings of movies.



Graph schema. Image by the author.

The visualization depicts a knowledge graph of individuals who have either acted in or directed a movie, which is further categorized by genre. Each movie node holds information about its release date, title, and IMDb rating. The graph also contains user ratings, which we can use to provide recommendations.

You can populate the graph by executing the `ingest.py` script, which is located in the root directory of the folder.

Defining tools

Now, we will define the tools an agent can use to interact with the knowledge graph. We will start with the **information tool**. Information tool is designed to fetch relevant information about actors, directors, and movies. The Python code looks the following:

```

def get_information(entity: str, type: str) -> str:
    # Use full text index to find relevant movies or people
    candidates = get_candidates(entity, type)
    if not candidates:
        return "No information was found about the movie or person in the database"
    elif len(candidates) > 1:
        newline = "\n"
        return (
            "Need additional information, which of these "
            f"did you mean: {newline + newline.join(str(d) for d in candidates)}"
        )
    data = graph.query(
        description_query, params={"candidate": candidates[0]["candidate"]}
    )
    return data[0]["context"]
  
```

The function starts by finding relevant people or movies mentioned using a full-text index. The full-text index in Neo4j uses Lucene under the hood. It enables a seamless implementation of text distance-based lookups, which allow the user to misspell some words and still get results. If no relevant entities are found, we can directly return a response. On the other hand, if multiple candidates are identified, we can guide the agent to ask the user a follow-up question and be more specific about the movie or person they are interested in. Imagine that a user asks, “Who is John?”.

```
print(get_information("John", "person"))
# Need additional information, which of these did you mean:
# {'candidate': 'John Lodge', 'label': 'Person'}
# {'candidate': 'John Warren', 'label': 'Person'}
# {'candidate': 'John Gray', 'label': 'Person'}
```

In this case, the tool informs the agent that it needs additional information. With simple prompt engineering, we can steer the agent to ask the user a follow-up question. Suppose the user is specific enough, which allows the tool to identify a particular movie or a person. In that case, we use a parametrized Cypher statement to retrieve relevant information.

```
print(get_information("Keanu Reeves", "person"))
# type: Actor
# title: Keanu Reeves
# year:
# ACTED_IN: Matrix Reloaded, The, Side by Side, Matrix Revolutions, The, Sweet N
# DIRECTED: Man of Tai Chi
```

With this information, the agent can answer most of the questions that concern Keanu Reeves.

Now, let's guide the agent on utilizing this tool effectively. Fortunately, with LangChain, the process is straightforward and efficient. First, we define the input parameters of the function using a Pydantic object.

```
class InformationInput(BaseModel):
    entity: str = Field(description="movie or a person mentioned in the question")
    entity_type: str = Field(
        description="type of the entity. Available options are 'movie' or 'person'"
    )
```

Here, we describe that both `entity` and `entity_type` parameters are strings. The `entity` parameter input is defined as the movie or a person mentioned in the question. On the other hand, with the `entity_type`, we also provide available options. When dealing with low cardinalities, meaning when there is a small number of distinct values, we can provide available options directly to an LLM so that it can use valid inputs. As we saw before, we use a full-text index to disambiguate movies or people as there are too many values to provide directly in the prompt.

Now let's put it all together in an `Information` tool definition.

```
class InformationTool(BaseTool):
    name = "Information"
    description = (
        "useful for when you need to answer questions about various actors or mo"
    )
    args_schema: Type[BaseModel] = InformationInput

    def _run(
        self,
        entity: str,
        entity_type: str,
        run_manager: Optional[CallbackManagerForToolRun] = None,
    ) -> str:
        """Use the tool."""
        return get_information(entity, entity_type)
```

Accurate and concise tool definitions are an important part of a semantic layer, so that an agent can correctly pick relevant tools when needed.

The recommendation tool is slightly more complex.

```
def recommend_movie(movie: Optional[str] = None, genre: Optional[str] = None) -> str:
    """
    Recommends movies based on user's history and preference
    for a specific movie and/or genre.
    Returns:
        str: A string containing a list of recommended movies, or an error message
    """
    user_id = get_user_id()
    params = {"user_id": user_id, "genre": genre}
    if not movie and not genre:
        # Try to recommend a movie based on the information in the db
        response = graph.query(recommendation_query_db_history, params)
        try:
            return ", ".join([el["movie"] for el in response])
        except Exception:
            return "Can you tell us about some of the movies you liked?"
    if not movie and genre:
        # Recommend top voted movies in the genre the user haven't seen before
        response = graph.query(recommendation_query_genre, params)
        try:
            return ", ".join([el["movie"] for el in response])
        except Exception:
```

```

        return "Something went wrong"

    candidates = get_candidates(movie, "movie")
    if not candidates:
        return "The movie you mentioned wasn't found in the database"
    params["movieTitles"] = [el["candidate"] for el in candidates]
    query = recommendation_query_movie(bool(genre))
    response = graph.query(query, params)
    try:
        return ", ".join([el["movie"] for el in response])
    except Exception:
        return "Something went wrong"

```

The first thing to notice is that both input parameters are optional. Therefore, we need to introduce workflows that handle all the possible combinations of input parameters and the lack of them. To produce personalized recommendations, we first get a `user_id`, which is then passed into downstream Cypher recommendation statements.

Similarly as before, we need to present the input of the function to the agent.

```

class RecommenderInput(BaseModel):
    movie: Optional[str] = Field(description="movie used for recommendation")
    genre: Optional[str] = Field(
        description=(
            "genre used for recommendation. Available options are:" f"{all_genres}"
        )
    )

```

Since only 20 available genres exist, we provide their values as part of the prompt. For movie disambiguation, we again use a full-text index within the function. As before, we finish with the tool definition to inform the LLM when to use it.

```

class RecommenderTool(BaseTool):
    name = "Recommender"
    description = "useful for when you need to recommend a movie"
    args_schema: Type[BaseModel] = RecommenderInput

    def _run(
        self,
        movie: Optional[str] = None,
        genre: Optional[str] = None,
        run_manager: Optional[CallbackManagerForToolRun] = None,
    ) -> str:
        """Use the tool."""
        return recommend_movie(movie, genre)

```

So far, we have defined two tools to retrieve data from the database.

However, the information flow doesn't have to be one-way. For example, when a user informs the agent they have already watched a movie and maybe liked it, we can store that information in the database and use it in further recommendations. Here is where the memory tool comes in handy.

```
def store_movie_rating(movie: str, rating: int):
    user_id = get_user_id()
    candidates = get_candidates(movie, "movie")
    if not candidates:
        return "This movie is not in our database"
    response = graph.query(
        store_rating_query,
        params={"user_id": user_id, "candidates": candidates, "rating": rating},
    )
    try:
        return response[0]["response"]
    except Exception as e:
        print(e)
        return "Something went wrong"

class MemoryInput(BaseModel):
    movie: str = Field(description="movie the user liked")
    rating: int = Field(
        description=(
            "Rating from 1 to 5, where one represents heavy dislike "
            "and 5 represent the user loved the movie"
        )
    )
```

The memory tool has two mandatory input parameters that define the movie and its rating. It's a straightforward tool. One thing I should mention is that I noticed in my testing that it probably makes sense to provide examples of when to give a specific rating, as the LLM isn't the best at it out of the box.

Agent

Let's put it now all together using LangChain expression language (LCEL) to define an agent.

```
llm = ChatOpenAI(temperature=0, model="gpt-4", streaming=True)
tools = [InformationTool(), RecommenderTool(), MemoryTool()]

llm_with_tools = llm.bind(functions=[format_tool_to_openai_function(t) for t in
tools])

prompt = ChatPromptTemplate.from_messages(
    [
        (
            "system",
            "You are a helpful assistant that finds information about movies "
```



"Do only the things the user specifically requested. ",

To make Medium work, we log user data. By using Medium, you agree to our Privacy Policy, including cookie policy.

```

        messages.append(dict(var_table_name= "agent_scratchpad" ),
    ]
)

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
)

```

LangChain expression language makes it very convenient to define an agent and expose all its functionalities. We won't go into LCEL syntax as that is beyond the scope of this blog post.

The movie agent backend is exposed as an API endpoint using [LangServe](#).

Streamlit chat application

Now we just have to implement a streamlit application that connects to the LangServe API endpoint and we are good to go. We'll just look at the async function that is used to retrieve an agent response.

```

async def get_agent_response(
    input: str, stream_handler: StreamHandler, chat_history: Optional[List[Tuple]]
):
    url = "http://api:8080/movie-agent/"
    st.session_state["generated"].append("")
    remote_runnable = RemoteRunnable(url)
    async for chunk in remote_runnable.astream_log(
        {"input": input, "chat_history": chat_history}
    ):
        log_entry = chunk.ops[0]
        value = log_entry.get("value")
        if isinstance(value, dict) and isinstance(value.get("steps"), list):
            for step in value.get("steps"):
                stream_handler.new_status(step["action"].log.strip("\n"))
        elif isinstance(value, str):
            st.session_state["generated"][-1] += value
            stream_handler.new_token(value)

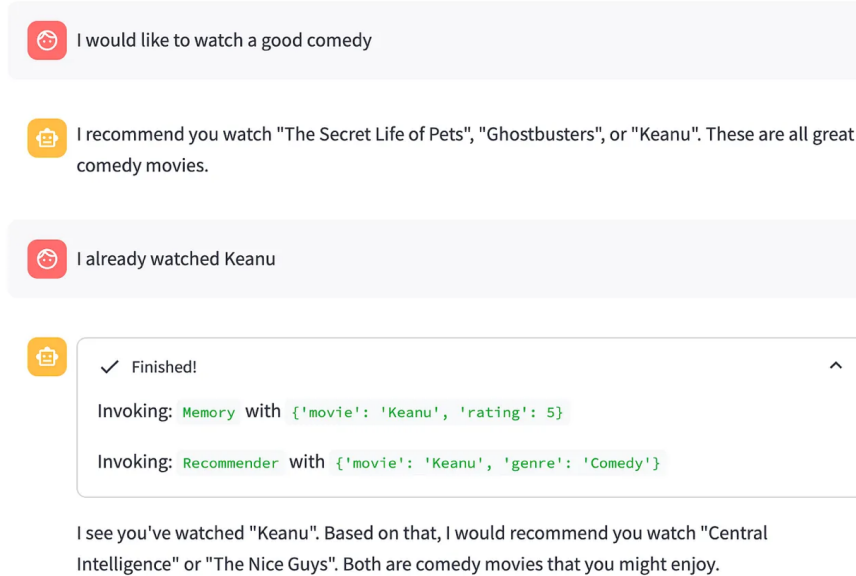
```

The function `get_agent_response` is designed to interact with a movie-agent API. It sends a request with the user's input and chat history to the API and then processes the response asynchronously. The function handles different

types of responses, updating the stream handler with new statuses and appending the generated text to the session state, which allows us to stream results to the user.

Let's now test it out

Movie agent



Movie agent in action. Image by the author.

The resulting movie agent offers a surprisingly good and guided interaction with the user.

Conclusion

In conclusion, the integration of a semantic layer in language model interactions with graph databases, as exemplified by our Movie Agent, represents a significant leap forward in enhancing user experience and data interaction efficiency. By shifting the focus from generating arbitrary Cypher statements to utilizing a structured, predefined suite of tools and functions, the semantic layer brings a new level of precision and consistency to language model engagements. This approach not only streamlines the process of extracting relevant information from knowledge graphs but also ensures a more goal-oriented, user-centric experience.

The semantic layer acts as a bridge, translating user intent into specific, actionable queries that the language model can execute with accuracy and reliability. As a result, users benefit from a system that not only understands their queries more effectively but also guides them towards their desired

outcomes with greater ease and less ambiguity. Furthermore, by constraining the language model's responses within the parameters of these predefined tools, we mitigate the risks of incorrect or irrelevant outputs, thereby enhancing the trustworthiness and reliability of the system.

The code is available on [GitHub](#).

Dataset

F. Maxwell Harper and Joseph A. Konstan. 2015. The MovieLens Datasets: History and Context. *ACM Transactions on Interactive Intelligent Systems (TiiS)* 5, 4: 19:1–19:19. <https://doi.org/10.1145/2827872>

Neo4j

Graph

Llm

Langchain

ChatGPT



Written by Tomaz Bratanic


7.5K Followers · Writer for Towards Data Science

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

Follow



More from Tomaz Bratanic and Towards Data Science

 Tomaz Bratanic

Constructing knowledge graphs from text using OpenAI functions

Seamlessly implement information extraction pipeline with LangChain and Neo4j

11 min read · Oct 20, 2023

 1.3K

 10



 Dave Melillo in Towards Data Science

Building a Data Platform in 2024


How to build a modern, scalable data platform to power your analytics and data

9 min read · Feb 6, 2024

 1.7K

 26



 Cristian Leo in Towards Data Science

The Math behind Adam Optimizer


Why is Adam the most popular optimizer in Deep Learning? Let's understand it by diving

16 min read · Jan 30, 2024

 2.5K

 20



 Tomaz Bratanic in Neo4j Developer Blog

JSON-based Agents With Ollama & LangChain

Learn to implement a Mixtral agent that interacts with a graph database Neo4j

8 min read · Feb 28, 2024

 540

 2



See all from Tomaz Bratanic

See all from Towards Data Science

Recommended from Medium



Tomaz Bratanic

Constructing knowledge graphs from text using OpenAI functions

Seamlessly implement information extraction pipeline with LangChain and Neo4j

11 min read · Oct 20, 2023



1.3K



10



NebulaGraph Database

Graph RAG: Unleashing the Power of Knowledge Graphs with LLM

In the era of information overload, sifting through vast amounts of data to provide

6 min read · Sep 8, 2023



438



4



Lists

What is ChatGPT?

9 stories · 314 saves

The New Chatbots: ChatGPT, Bard, and Beyond

12 stories · 327 saves

ChatGPT prompts

45 stories · 1222 saves

ChatGPT

21 stories · 502 saves

 Peter Lawrence, answering users' data ... in GoPe...

LLM Ontology-prompting for Knowledge Graph Extraction


Prompting an LLM with an ontology to drive Knowledge Graph extraction from

5 min read · Jun 27, 2023

 667

 9



 Chia Jeng Yang in Enterprise RAG

Injecting Knowledge Graphs in different RAG stages


Injecting KGs in RAG in pre-processing, post-processing, chunk extraction, document and

10 min read · Jan 5, 2024

 526

 2



 Fanghua (Joshua) Yu in Neo4j Developer Blog

Building A Graph+LLM Powered RAG Application from PDF


A Step-by-step Walkthrough with GenAI-Stack and OpenAI

6 min read · Jan 15, 2024

 508

 2



 Florian June in Towards AI

Advanced RAG 02: Unveiling PDF Parsing

Including key points, diagrams, and code

🌟 · 13 min read · Feb 2, 2024

 1.3K

 11



See more recommendations