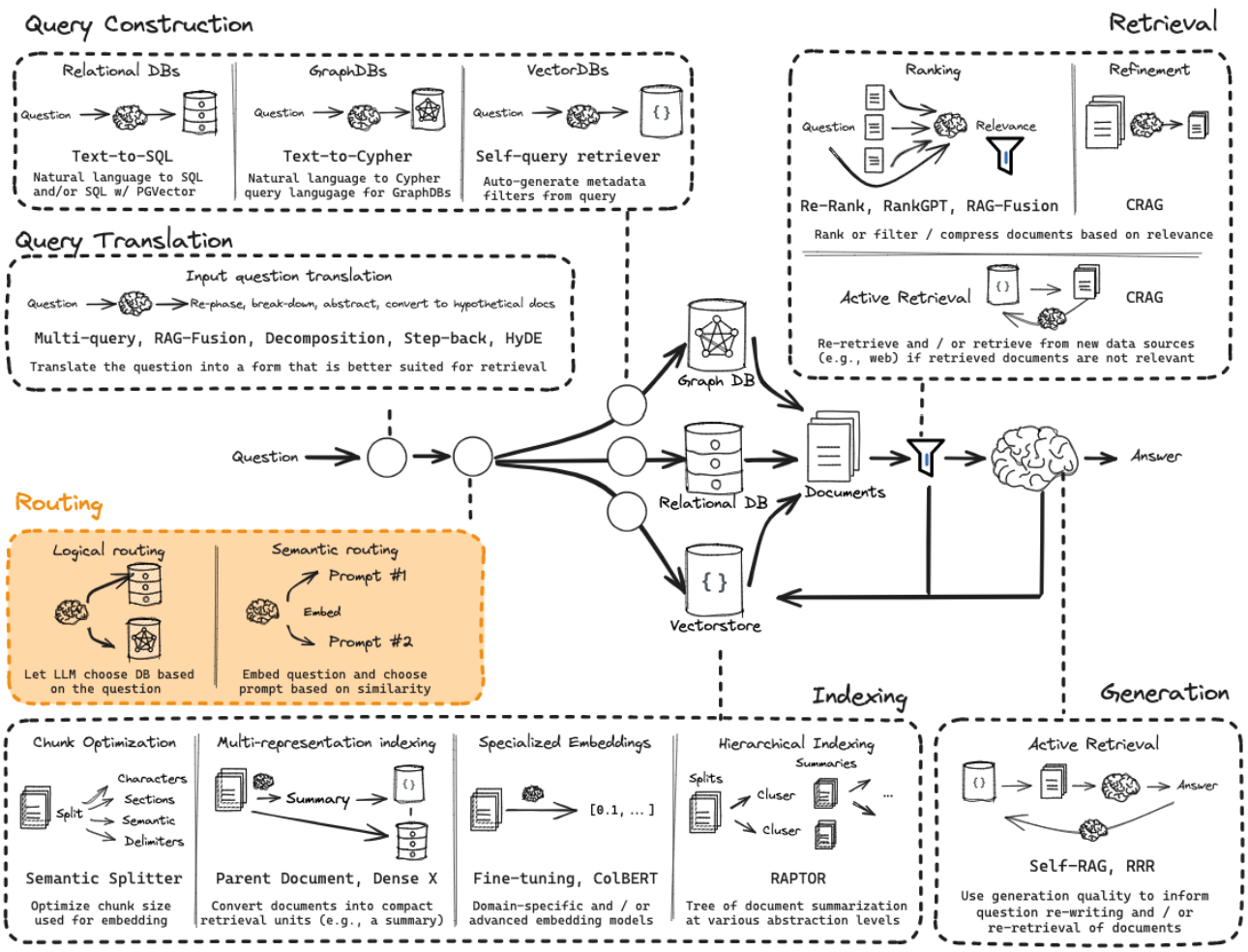
Rag for Scratch——langchain

Routing

Routing是RAG pipeline的第二个阶段，目的是依据query的内容路由到正确的数据源。实际上就是一个分类任务。



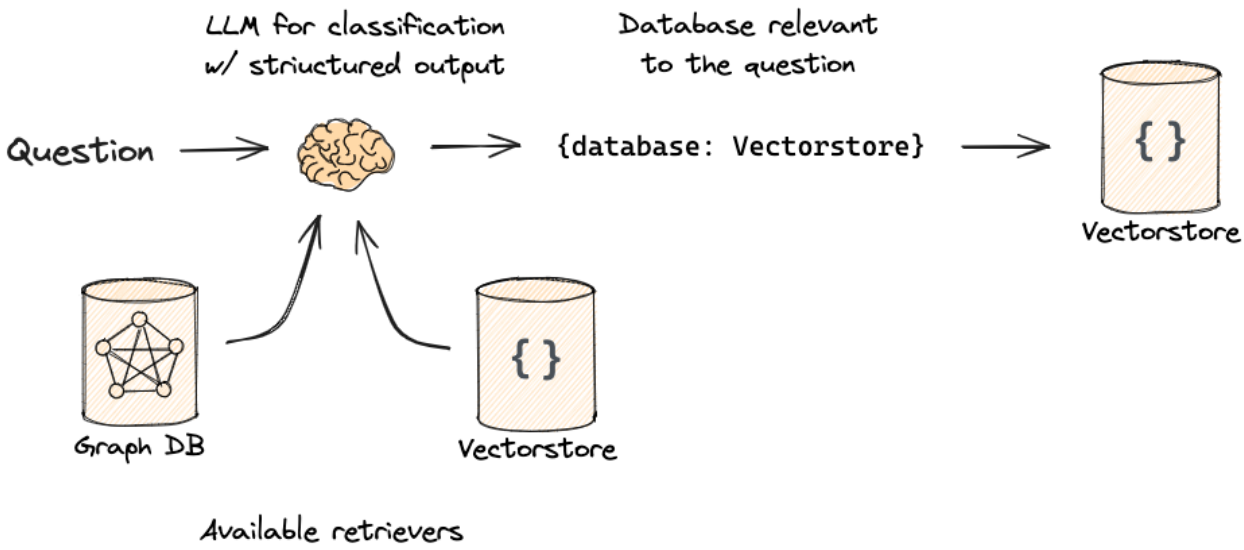
举例：

假设我们有多个索引：

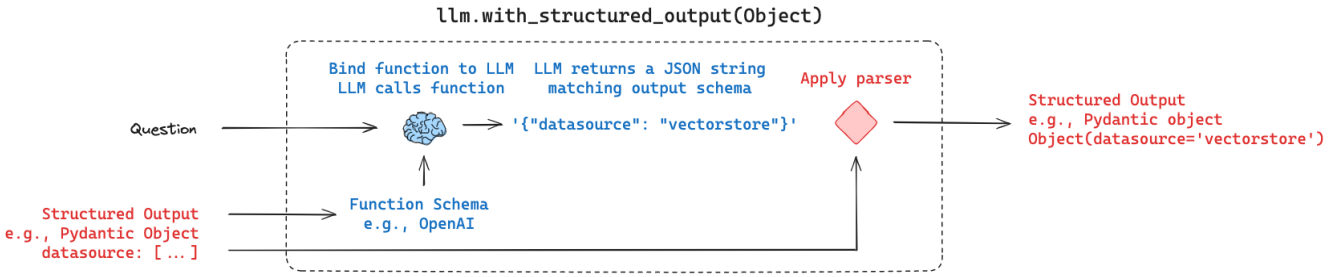
1. 一个索引包含了所有与LangChain相关的Python文档。
2. 另一个索引包含了所有与LangChain相关的JavaScript文档。

- 用户query：“如何在LangChain中使用Python进行文本分析？”
    - 选择包含所有LangChain Python文档的索引进行查询。
- 用户query：“LangChain的JS库是否支持WebSocket？”
    - 选择包含LangChain JavaScript文档的索引进行查询。

logical routing



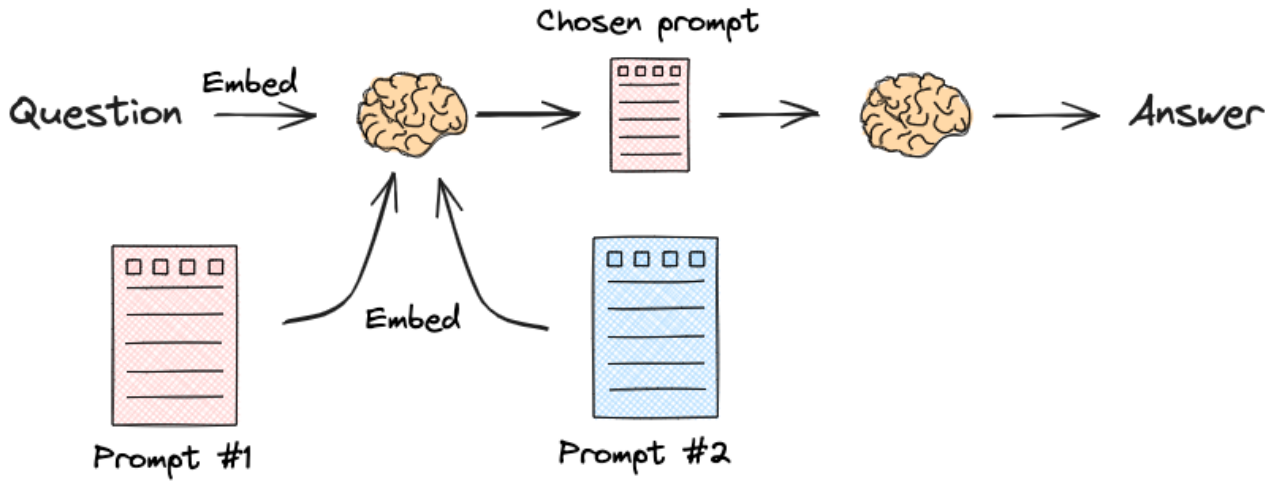
使用function calling得到结构化的输出




```
from typing import List


class RouteQuery(BaseModel):
    """Route a user query to the most relevant datasource."""

    datasources: List[Literal["python_docs", "js_docs", "golang_docs"]] = Field(
        ...,
        description="Given a user question choose which datasources would be most relevant
for answering their question",
    )


llm = ChatOpenAI(model="gpt-3.5-turbo-0125", temperature=0)
structured_llm = llm.with_structured_output(RouteQuery)
router = prompt | structured_llm
router.invoke(
    {
        "question": "is there feature parity between the Python and JS implementations of
OpenAI chat models"
    }
)
```

semantic rounting

通过语义相似度进行路由， 一种特别有用的技术是使用embeddings将query路由到最相关的prompt。

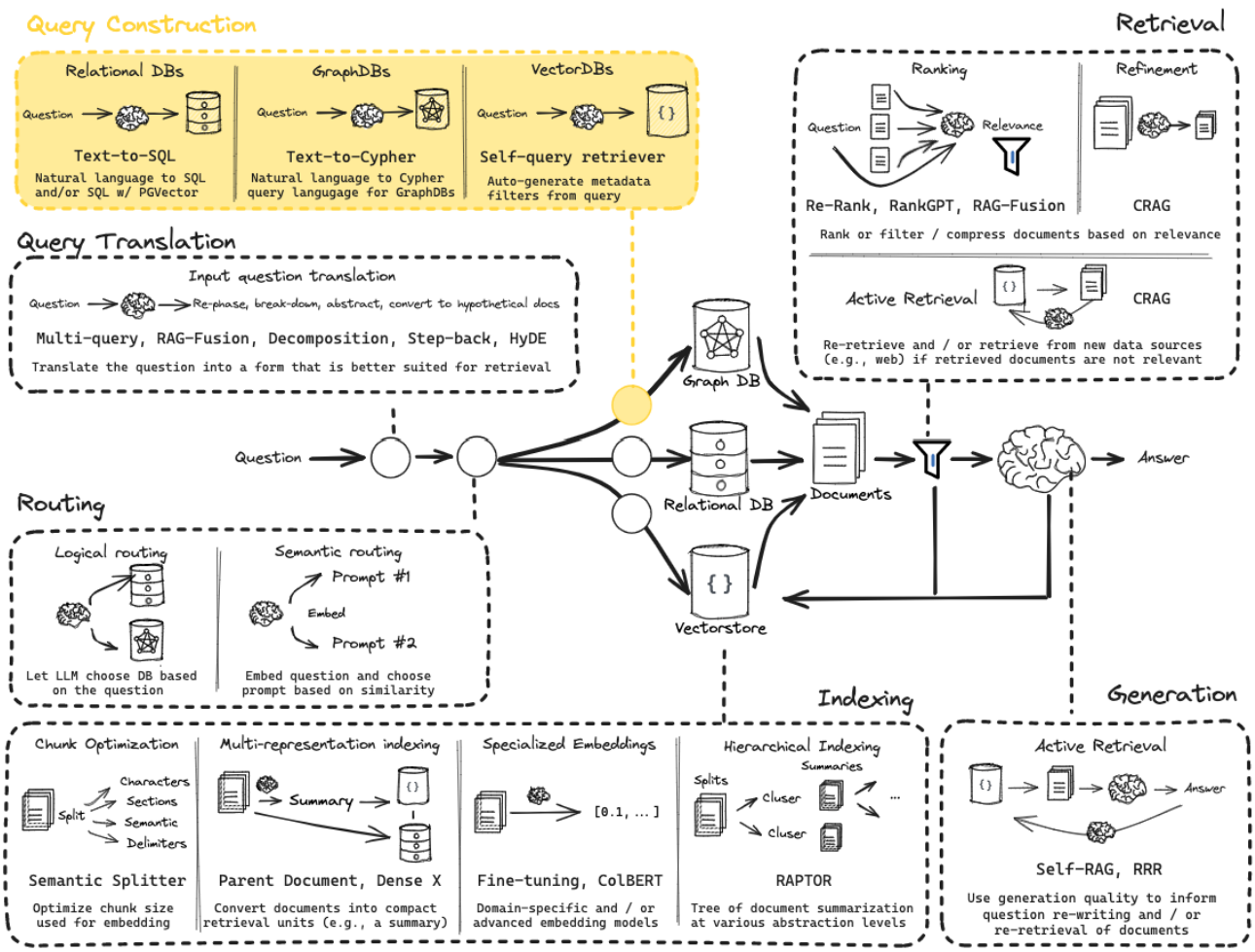


关键代码：

```
# Route question to prompt
def prompt_router(input):
    # Embed question
    query_embedding = embeddings.embed_query(input["query"])
    # Compute similarity
    similarity = cosine_similarity([query_embedding], prompt_embeddings)[0]
    most_similar = prompt_templates[similarity.argmax()]
    # Chosen prompt
    print("Using MATH" if most_similar == math_template else "Using PHYSICS")
    return PromptTemplate.from_template(most_similar)
```

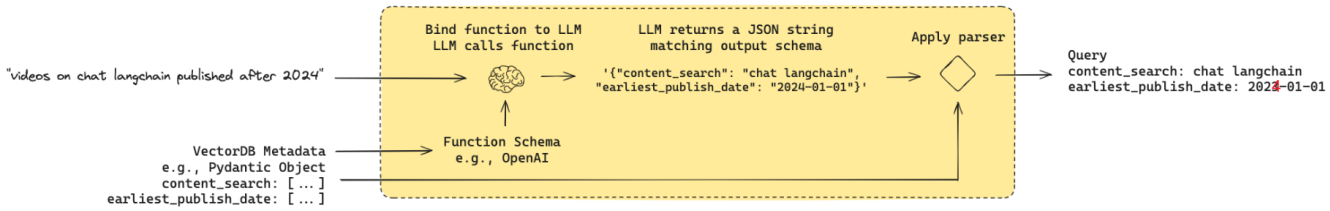
Query Construction

Query Construction是RAG pipeline的第三个阶段，就是将自然语言query转换为结构化语言，以便于进行有效的搜索和过滤。



Query structuring for metadata filters

将自然语言形式的quey输入转化为合适的搜索和过滤参数。这一过程被称为query structuring。



举例：

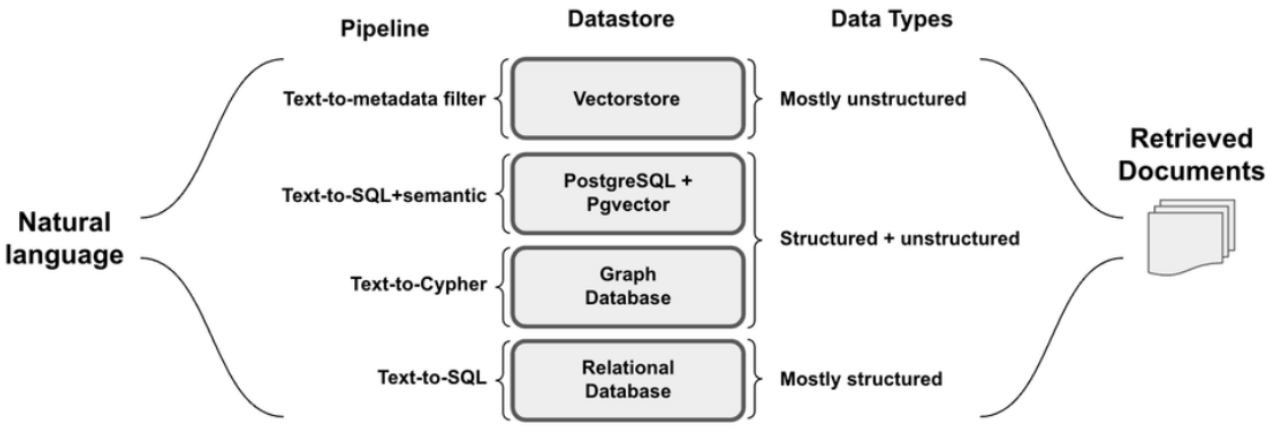
用户query：how to use multi-modal models in an agent, only videos under 5 minutes

输出：

content_search: multi-modal models agent title_search: multi-modal models agent max_length_sec: 300

Query Construction的几种形式

Query Construction (langchain.dev)



Text-to-metadata-filter

vectorstore具备 metadata filtering。

self-query retriever把自然语言query转化为结构化query的步骤：

1. **数据源定义**：对相关元数据（Metadata）进行明确指定。

   例如，检索音乐时，元数据可能包括artist, length, genre等。

2. **用户查询解释**：将用户query分解为两个部分：一个是query（用于语义检索），另一个是过滤元数据的 filter。

   例如，对于query："songs by Taylor Swift or Katy Perry about teenage romance under 3 minutes long in the dance pop genre"，会将其分解为一个query和一个filter。

3. **逻辑条件提取**：过滤条件是通过向量存储中定义的比较器和操作符（如eq、lt）来构建的。

4. **结构化请求的形成**：最后，self-query retriever会组装成一个结构化请求，将query与filter区分开来。

```
# Generate a prompt and parse output
prompt = get_query_constructor_prompt(document_content_description, metadata_field_info)
output_parser = StructuredQueryOutputParser.from_components()
query_constructor = prompt | llm | output_parser

# Invoke the query constructor with a sample query
query_constructor.invoke({
 "query": "Songs by Taylor Swift or Katy Perry about teenage romance under 3 minutes long
in the dance pop genre"
})
```

输出：

```
{
 "query": "teenager love",
 "filter": "and(or(eq(\"artist\", \"Taylor Swift\"), eq(\"artist\", \"Katy Perry\")),
lt(\"length\", 180), eq(\"genre\", \"pop\"))"
}
```

Text-to-SQL

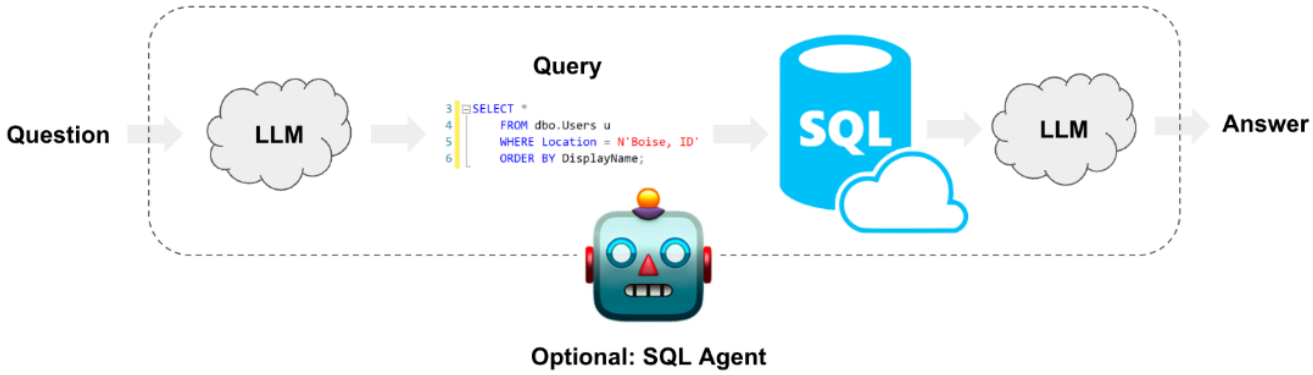
将自然语言转换为SQL请求。

挑战：

1. **幻觉现象**：LLMs有时会'幻觉'出不存在的表或字段。
2. **用户错误**：需要对用户的拼写错误或其他不规范输入具有鲁棒性。

应对的方法：

1. **数据库描述**：一个常见的text-to-SQL提示方法是给LLM提供每个表的CREATE TABLE描述（包括列名、类型等）以及三行示例数据。
2. **少样本示例**：这可以通过在提示中附加标准的静态示例来实现，指导模型根据问题生成查询。
3. **错误处理**：数据分析师在遇到错误时不会放弃，他们会继续迭代修正。可以使用类似SQL Agent的工具来从错误中恢复。
4. **发现拼写错误的专有名词**：在查询专有名词（如人名）时，用户可能会拼写错误（例如，把Frank Sinatra拼成Franc Sinatra）。可以允许模型在一个vectorstore中搜索正确拼写的专有名词，以便在SQL数据库中使用。



Text-To-SQL+semantic

数据存储中既有结构化又有非结构化数据的情况。例如，PostgreSQL的开源扩展pgvector就结合了SQL的表达能力和语义搜索的细致理解。

pgvector使得在嵌入向量列上执行相似度搜索变得可能（例如，余弦相似度、L2距离、内积），并使用<-> 运算符：

```
SELECT * FROM tracks ORDER BY "name_embedding" <-> {sadness_embedding}
```

通过上述查询，我们可以使用 LIMIT 3 获取前3个最悲伤的曲目，也可以进行更复杂的操作，例如选择最悲伤的曲目，以及第90和50百分位的曲目。

这种方法解锁了两个新功能：

1. **进行语义搜索**：可以进行使用vectorstore难以实现的语义搜索。
2. **增强 text-to-SQL 的功能**：借助语义操作符，可以实现文本到语义搜索（例如，查找标题传达特定情感的歌曲）和SQL查询（例如，按类型过滤）。

例子：

查找名字中含有"lovely"并且最悲伤的前三个音乐：

```
SELECT *
FROM tracks
WHERE album_name LIKE '%lovely%'
ORDER BY "name_embedding" <-> {sadness_embedding}
LIMIT 3;
```

Text-to-Cypher

利用图数据库和LLM来处理复杂的数据查询需求。

1. **图数据库与Cypher查询语言**：
   - 图数据库通常使用特定的查询语言，叫做Cypher，它提供了一种可视化的方式来匹配模式和关系。
   - Cypher使用类ASCII艺术的语法描述数据节点和关系。例如：

     ```
     (:Person {name:"Tomaz"})-[:LIVES_IN]->(:Country {name:"Slovenia"})
     ```

     这一模式描述了一个名为Tomaz的Person节点有一个LIVES_IN关系指向名为Slovenia的Country节点。
2. **自然语言到Cypher的转换**：
   - 可以使用自然语言将查询转换为Cypher查询。例如，使用图数据库查询链（GraphCypherQAChain）和LLM。

例子：

```
from langchain.chains import GraphCypherQAChain

graph.refresh_schema()

cypher_chain = GraphCypherQAChain.from_llm(
    cypher_llm = ChatOpenAI(temperature=0, model_name='gpt-4'),
    qa_llm = ChatOpenAI(temperature=0), graph=graph, verbose=True,
)
```

```
cypher_chain.run(
    "How many open tickets there are?"
)
```

通过上述代码，我们可以使用自然语言来提问，例如询问有多少未解决的票据，系统会自动生成相应的Cypher查询。

知识图谱（我也不懂）