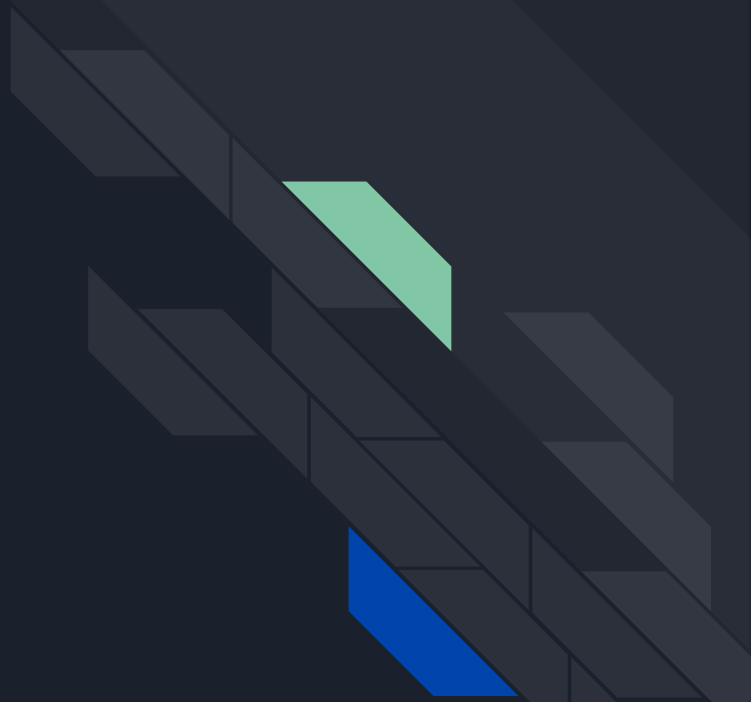
Building A Natural Language Query Agent

Leveraging Simple Concepts to Handle Complexity

By
Joshua Jacobs
Travis Richardson

TOC



Overview

Interrogating a RAG-Encoded EMR Database with LLMs

This project showcases a Natural Language Query interface for Electronic Medical Record (EMR) data using Large Language Models. It enables subject matter experts to query complex healthcare data in plain English–without writing SQL.

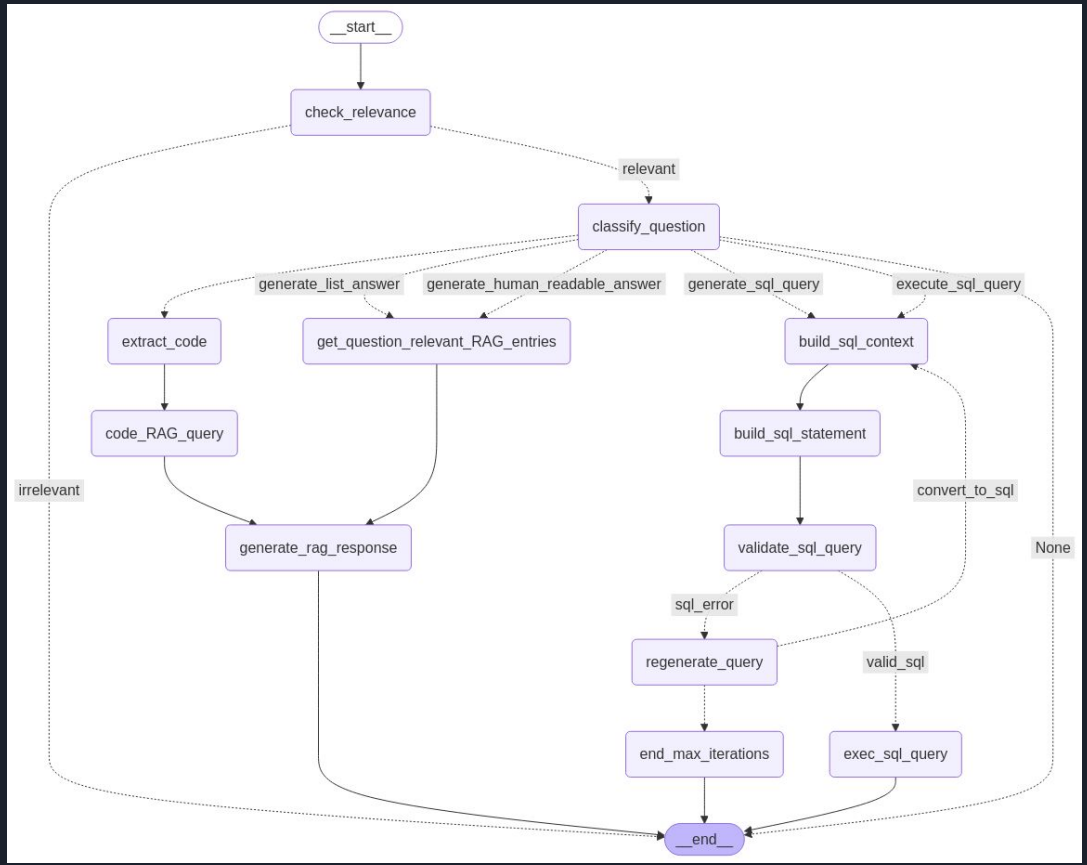
This frame works address key challenges in healthcare data analysis:

- **Semantic Variability** in categorical data (e.g. synonyms, ontologies)
- Unifying **Heterogeneous Sources** with differing structures.
- **Metadata Curation** for improving sharing, standardization, and reuse.

Demonstration

Switch to Jupyter Notebook

Demonstration



Prepping the Database

- The Database was created from the Synthea Patient Generator.
  - Integrations with Flywheel are not a part of this presentation
  - Tables and Schemas are compliant with the GA4GH Data Connect Standard
  - Queries are enabled by a DuckDB OLAP Engine
- Table and Column descriptions are inferred by LLM calls
  - Or manually entered.
- Primary Key/Foreign Key relationships are established
- Tables are profiled with ydata-profiling for
  - Cardinality
  - Type
  - Distributions
  - Categorical Data (Code, Description, Code/Description Pairs)
- Tables, Columns, and Categoricals are RAG-Encoded

Prepping the Database

RAG-Encoding Database Properties

Retrieval-Augmented Generation (RAG)

- Semantic “Meaning” -> high-dimensional vector
- Each Table, Column, and Categorical gets encoded into a RAG Table with the fields:
  - **id**: A unique identifier that is a hash of the metadata
  - **metadata**: A JSON object built from the “class” of the object (e.g. RAGTableItem)
  - **contents**: The string representation of metadata used for RAG-Encoding
  - **embedding**: The high-dimensional (e.g. 1136) semantic embedding vector
- Example: metadata for the ***allergies*** table:

  {'column': None, 'data_type': None, 'description': 'Table derived from Tabular Data File: allergies.csv. This table captures detailed information about allergies recorded for patients, including the type, category, reactions, and severity of each allergy.', 'is_categorical': False, 'table': 'allergies', 'type': 'table', 'value': 'allergies'}

The Tool Stack

01 LangGraph

02 Instructor

03 OpenAI Models

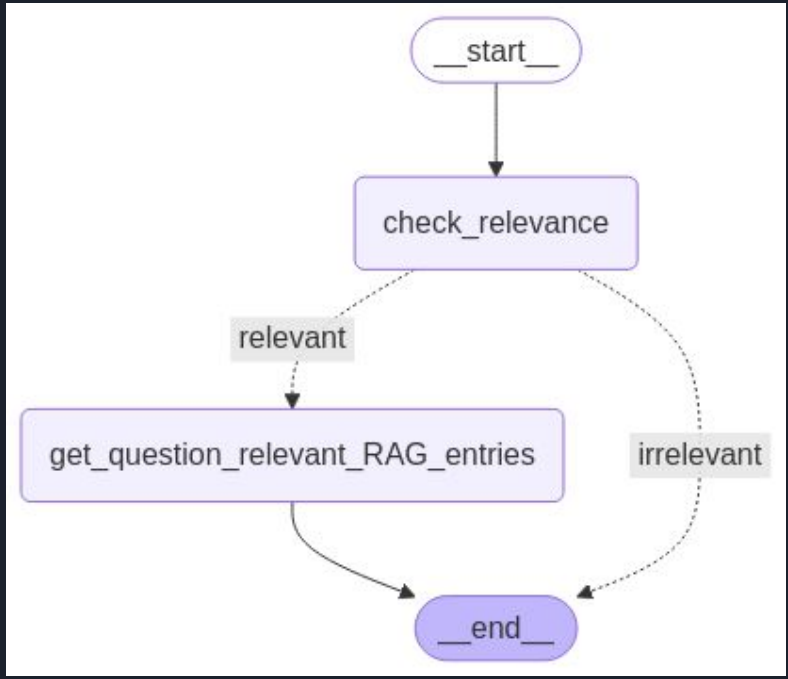
Tool Stack

LangGraph

LangGraph divides LLM calls to distinct nodes that update the AgentState. This allows for sub-task-specific context, increased accuracy, and configurable workflow.

The components of LangGraph are:

- **Nodes**: Specific Tasks with results written to an “agent state”
- **Edges**: Directed Links between nodes
- **Edge Routers**: Conditional Links between nodes
- **Workflow:** The complete graph of tasks



Tool Stack

LangGraph - Example Workflow

```
rag_workflow = StateGraph(AgentState)

# Add Nodes to rag_workflow
rag_workflow.add_node("check_relevance", check_relevance)
rag_workflow.add_node("get_question_relevant_RAG_entries", get_question_relevant_RAG_entries)

# Add Edge Router
rag_workflow.add_conditional_edges(
    "check_relevance",
    check_relevance_router,
    {"irrelevant": END, "relevant": "get_question_relevant_RAG_entries"},
)
# Start and End
rag_workflow.set_entry_point("check_relevance")
rag_workflow.add_edge("get_question_relevant_RAG_entries", END)
```

Tool Stack

LangGraph - Example Edge Router

```
def check_relevance_router(state: AgentState) -> str:
    """Determine whether the question is relevant or not.

    Args:
        state (AgentState): The state of the agent.

    Returns:
        str: Whether the question is relevant or not.
    """
    relevant = state.get("relevant", None)
    if not relevant:
        return "irrelevant"
    return "relevant"
```

Tool Stack

LangGraph - Example Node

```
def check_relevance(state: AgentState) -> AgentState:
    question = state["question"]
    system_prompt = "You are an expert in biomedical database design and analysis…"
    user_prompt = f"Question: '{question}'\nCheck the relevance of the question to the database."
    messages = [{"role":"system", "content":system_prompt},{"role": "user", "content":user_prompt}]
    instructor_client = instructor.from_openai(OpenAI())
    response = instructor_client.chat.completions.create(
        model="gpt-3.5-turbo",
        temperature=0.2,
        response_model=QuestionRelevanceModel,
        messages=messages,
    )
    state["relevant"] = response.is_relevant
    state["response"] = response.justification
    return state
```

Tool Stack

Instructor

“Instructor” is a library that enhances the interaction between Python and LLMs by enabling structured output extraction using Pydantic models.

It simplifies the process of getting predictable and validated responses from LLMs, making it easier to build applications that rely on structured data.

Tool Stack

Instructor - Anatomy of an Instructor Call

```
response = instructor_client.chat.completions.create(
    model="gpt-3.5-turbo", # <- The model to use for the completion
    temperature=0.2, # <- The "randomness" of the completion, lower values are more deterministic
    response_model=QuestionRelevanceModel, # <- The response model to use for the completion
    max_retries=2, # <- The maximum number of retries for the completion
    max_tokens=100, # <- The maximum number of tokens for the completion
    messages=messages, # <- The system and user prompts to use for the completion
)
```

Tool Stack

Instructor - What's in a "Model"?

Pydantic models provide both a structure for the LLM model response and additional context for the LLM to leverage in formulating that response. Furthermore, Pydantic models offer an additional layer of validation to ensure LLM model responses are within expected constraints.

```
class QuestionRelevanceModel(BaseModel):
    """Model to store the relevance of a question and a justification."""
    is_relevant: bool = Field(..., description="Whether the question is relevant to the context.")
    justification: str = Field(..., description="A brief explanation on the relevance.")
```

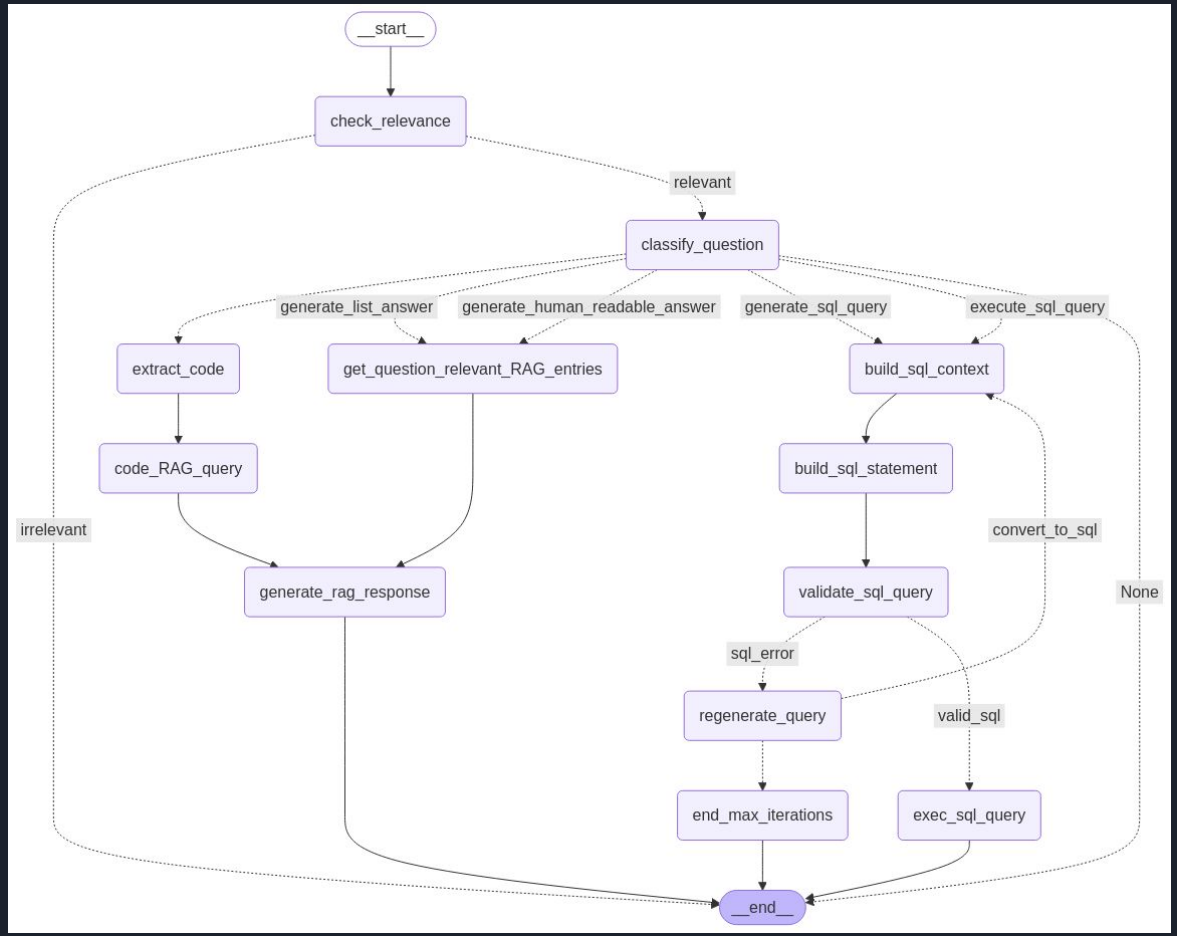
Tool Stack

OpenAI Models used through the API

- `text-embedding-3-small`
  - Used to create 1536-dimensional embedding vectors for RAG DB
- `gpt-3.5-turbo`
  - Simpler model used for components and smaller context
- `gpt-4o`
  - More complex model used for larger context and final output
- Instructor can use many other Models:
  - Ollama
  - Deep Seek
  - …

Tool Stack

Bringing it all together



Future Directions

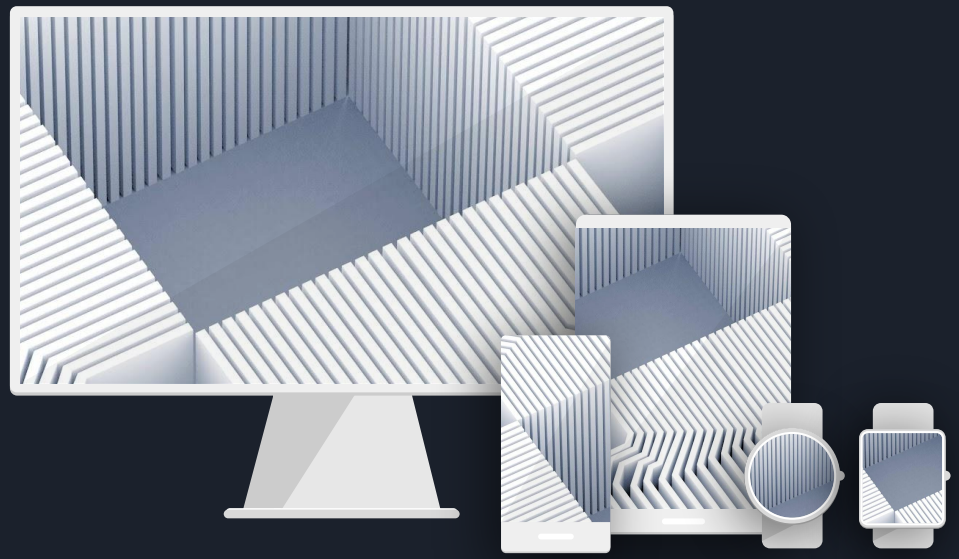
Potential Improvements

- Replace Instructor with PydanticAI calls
- Replace LangGraph with PydanticAI Graphs
- Integrate functionality into a Model Context Protocol (MCP) server/client.
- Extend metadata extraction and curation to multiple file types

Thank you!

Questions?

Feel Free to Email me at
joshua.jacobs.phd@gmail.com



References

- DuckDB - A fast and portable OLAP database engine
- Synthea Patient Generator - Generate representative patient data
- Ydata-profiling - Profile the tables of a database
- LangGraph - Divide an LLM workflow into simple calls
- Instructor - Structure responses from an LLM
- Pydantic - Create self-validating LLM response models
- OpenAI Models - Choose a model that fits constraints