Building a Multi-Format RAG App

Retrieval Augmented Generation on {Code, FAQs, SQL, Docs}

4 min read · 2 hours ago

Yogesh Haribhau Kulkarni (PhD)

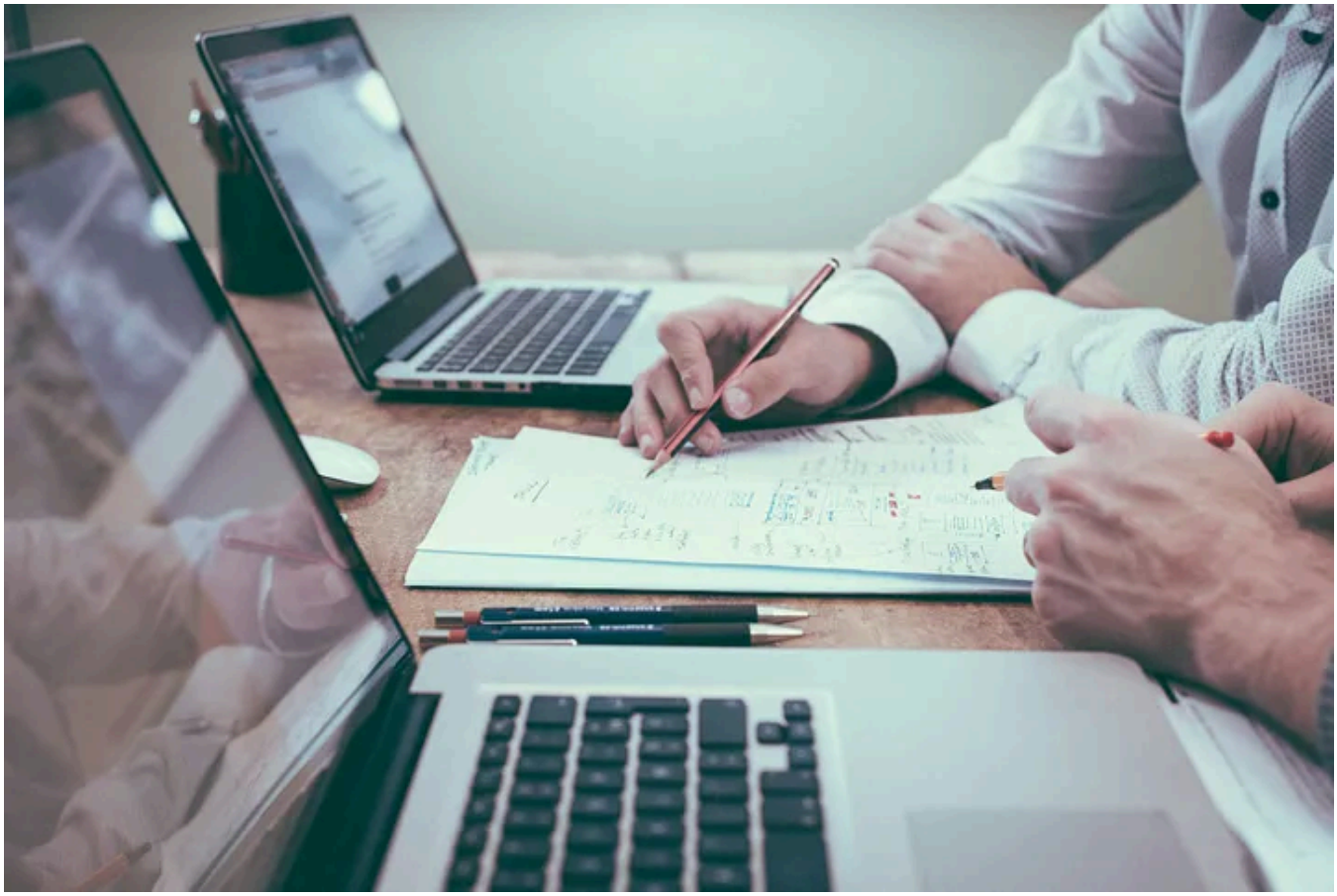
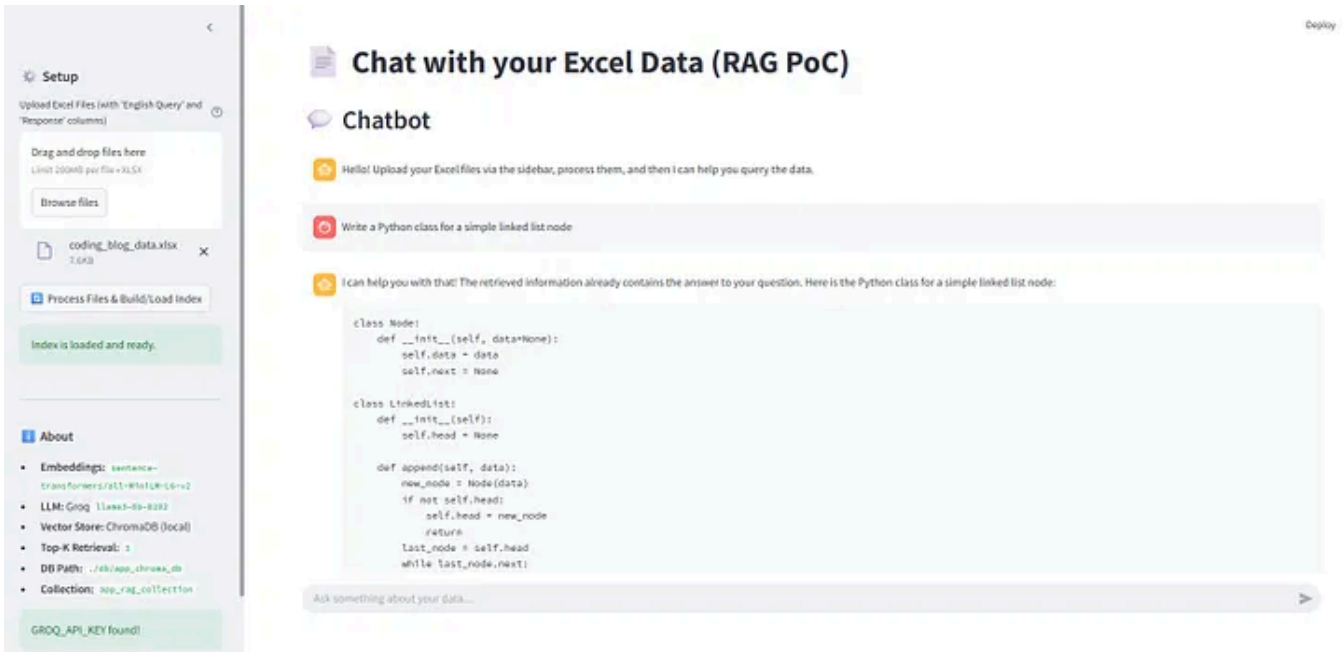


Photo by Scott Graham on Unsplash

Retrieval-Augmented Generation (RAG) has emerged as a powerful technique that combines the best of information retrieval and language generation. Here, we will go through steps for building a versatile chatbot application that can handle multiple use cases, from FAQ responses to SQL generation, all powered by data stored in simple Excel files.



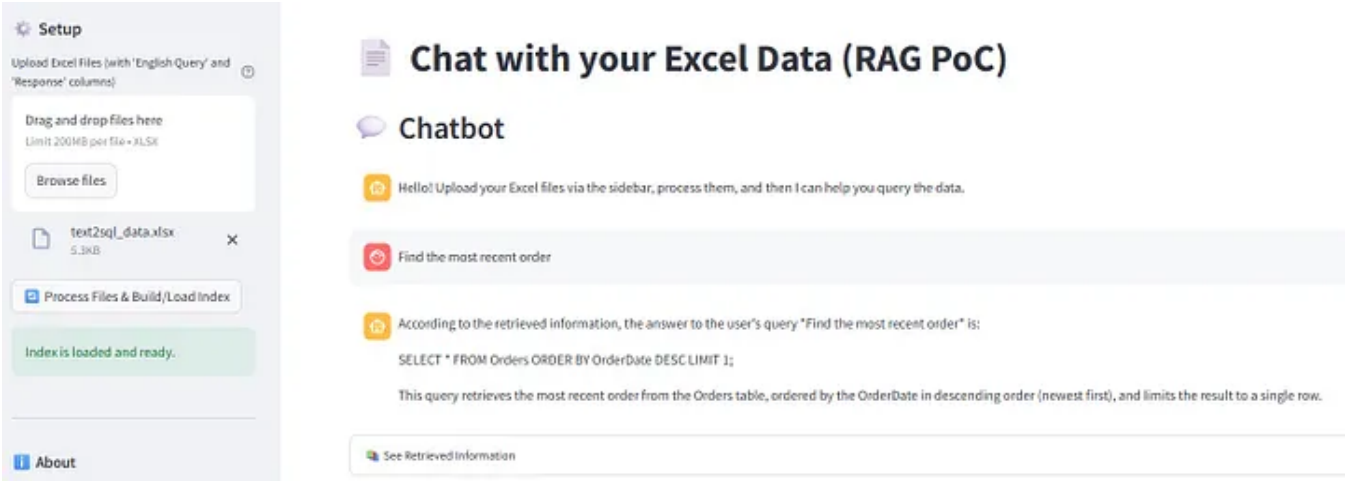
(Screenshot of the running app: text2code)

The Problem

Traditional chatbots often struggle with domain-specific knowledge and maintaining accuracy across different use cases. This approach addresses this by creating a unified RAG system that can:

- **Answer FAQs** with detailed paragraph responses
- **Generate SQL queries** from natural language
- **Provide QA testing steps** for software testing scenarios
- **Share code snippets** for programming queries

The beauty of this approach lies in its simplicity: all training data is stored in Excel files with just two columns: “English Query” and “Response”, making it accessible to non-technical users who can easily update the knowledge base.



(Screenshot of the running app: text2sql)

Understanding RAG Architecture

RAG works by combining retrieval mechanisms with generative AI models. Instead of relying solely on a language model's pre-trained knowledge, RAG retrieves relevant information from a knowledge base and uses it to generate more accurate, contextual responses.

Implementation below follows this workflow:

1. **Indexing Phase:** Convert Excel data into vector embeddings and store them in ChromaDB
2. **Retrieval Phase:** Find similar queries from the indexed data based on user input
3. **Generation Phase:** Use retrieved context to generate informed responses via Groq's LLM

Technical Stack Overview

Our application leverages several cutting-edge technologies:

- **LlamaIndex:** Framework for building RAG applications
- **ChromaDB:** Vector database for storing and retrieving embeddings
- **Groq:** Fast inference API for language models
- **Streamlit:** Web interface for the chatbot
- **HuggingFace Embeddings:** For converting text to vector representations

Core Implementation: The RAGSystem Class

The heart of the application is the `RAGSystem` class in `rag.py`. Here's how the key components work:

Initialization and Configuration

```
class RAGSystem:
    def __init__(self, embed_model_name="sentence-transformers/all-MiniLM-L6-v2
                 llm_model_name="llama3-8b-8192", chunk_size=512, top_k=3,
                 db_path="./db/chroma_db_excel_poc_class",
                 collection_name="my_excel_rag_collection_class"):

        self.groq_api_key = os.getenv("GROQ_API_KEY")
        if not self.groq_api_key:
            raise ValueError("GROQ_API_KEY not found in environment variables."

        self._configure_llamaindex()
        self.index = None
```

The constructor sets up essential parameters including the embedding model (all-MiniLM-L6-v2 for efficient semantic similarity), the LLM model (Llama3–8B via Groq), and database configurations.

Data Loading and Indexing

The most critical part is converting Excel data into a searchable vector index:

```
def _load_data_from_excel(self, file_path):
    df = pd.read_excel(file_path)
    if "English Query" not in df.columns or "Response" not in df.columns:
        print(f"Warning: Excel file {file_path} must contain 'English Query' ar
        return []

    documents = []
    for _, row in df.iterrows():
        query = str(row["English Query"])
        response = str(row["Response"])
        doc = Document(
            text=query,  # This gets embedded for similarity search
            metadata={"response": response, "filename": os.path.basename(file_p
        )
```

```
            documents.append(doc)
    return documents
```

This function reads Excel files and creates LlamaIndex `Document` objects where the query text becomes the searchable content, while the response is stored as metadata for later retrieval.

The Query Process

When a user asks a question, the system performs semantic search and generates contextual responses:

```
def query_index(self, user_query: str):
    retriever = self.index.as_retriever(similarity_top_k=self.top_k)
    retrieved_nodes = retriever.retrieve(user_query)

    context_parts = []
    for node_with_score in retrieved_nodes:
        original_query = node_with_score.node.get_text()
        response_from_excel = node_with_score.node.metadata.get("response")
        context_parts.append(f"Retrieved Question: {original_query}\nRetrieved

    context_str = "\n---\n".join(context_parts)

    prompt_template = f"""
    Based on the following retrieved information, answer the user's query.
    If the query is similar to a 'Retrieved Question', prioritize its 'Retrieve

    Retrieved Information: {context_str}
    User's Query: {user_query}
    Your Answer:
    """

    response_llm = Settings.llm.complete(prompt_template)
    return str(response_llm), retrieved_info_for_display
```

The system achieves this versatility by storing the appropriate response type in the Excel's "Response" column, allowing the same retrieval mechanism to serve different purposes.

Streamlit Frontend Integration

The `app.py` file creates a web interface using Streamlit's caching mechanism to maintain the RAG system state:

```
@st.cache_resource
def get_rag_system():
    try:
        rag_system_instance = RAGSystem(
            db_path="./db/app_chroma_db",
            collection_name="app_rag_collection"
        )
        if rag_system_instance.index is None:
            rag_system_instance._load_existing_index()
        return rag_system_instance
    except ValueError as e:
        st.error(f"Failed to initialize RAG System: {e}")
        return None
```

The `@st.cache_resource` decorator ensures the RAG system is initialized only once, preserving the loaded index across user interactions.

Key Advantages and Best Practices

Advantages:

1. **Accessibility**: Non-technical users can update the knowledge base via Excel
2. **Flexibility**: Supports multiple response types from a single system
3. **Scalability**: ChromaDB efficiently handles growing datasets
4. **Cost-Effective**: Uses open-source models via Groq's fast inference

Best Practices:

- Maintain consistent column naming in Excel files
- Use descriptive queries in the first column for better retrieval accuracy
- Regularly test with diverse query types to ensure robust performance
- Monitor retrieval scores to identify gaps in the knowledge base

Conclusion

This RAG-powered chatbot demonstrates how modern AI frameworks can create sophisticated applications with relatively simple data sources. By combining

LlamaIndex's powerful abstractions, ChromaDB's vector storage capabilities, and Groq's fast inference, we've built a system that's both technically robust and user-friendly.

The modular architecture makes it easy to extend, whether adding new response types, integrating different data sources, or scaling to handle larger datasets. As organizations continue to seek ways to leverage their existing data for AI applications, approaches like this provide a practical pathway from traditional data storage to intelligent, conversational interfaces.

The complete source code and sample Excel files are available for experimentation (not for ready usage or production) at Github, making it easy to adapt this solution for your specific use cases. Whether you're building customer support systems, internal knowledge bases, or specialized query interfaces, this RAG framework provides a solid foundation for intelligent data interaction.

Retrieval Augmented Gen | Artificial Intelligence | Chatbots | Llamaindex | Groq