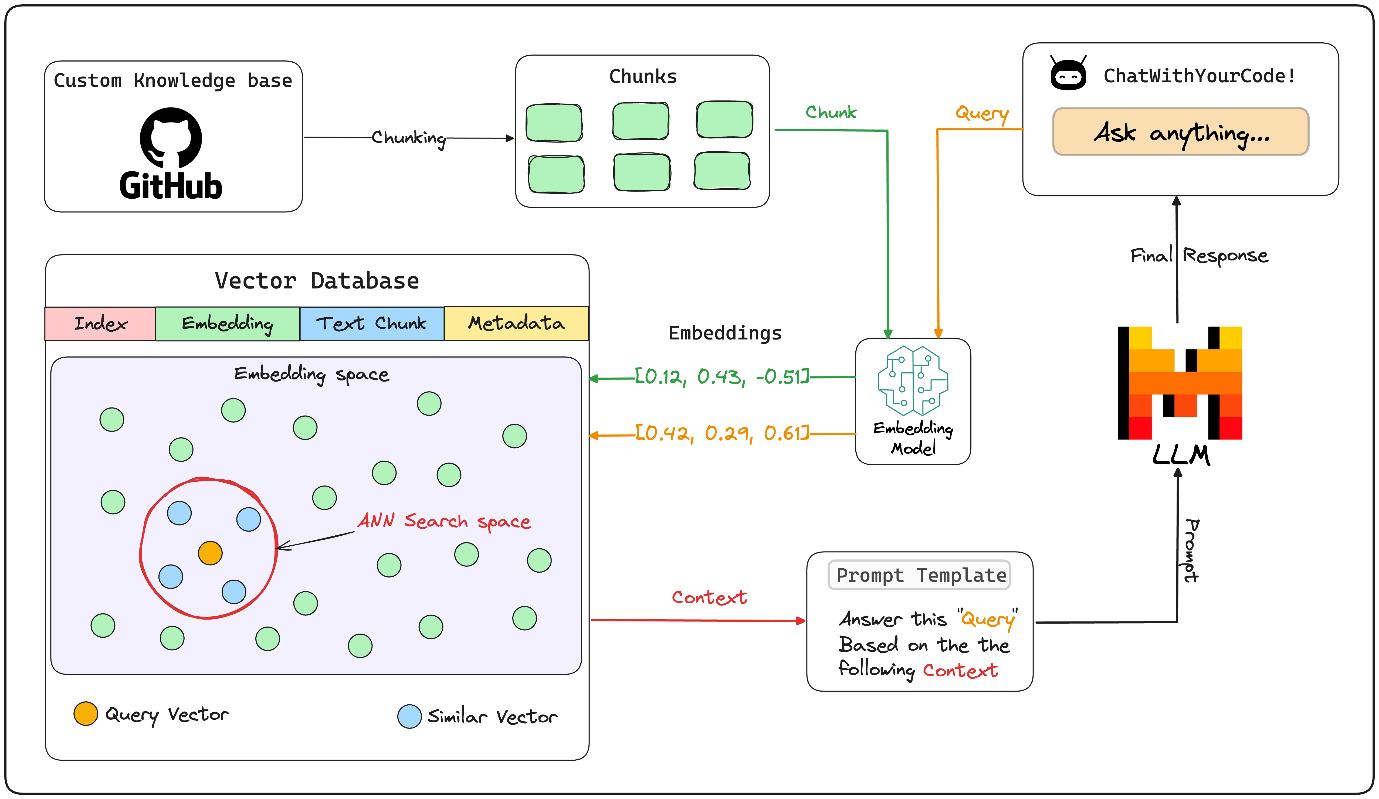
**Chat with your code using RAG!**

[**https://lightning.ai/lightning-ai/studios/chat-with-your-code-using-rag?\_\_s=u4pvflfacap82vd4gibe&utm\_source=drip&utm\_medium=email&utm\_campaign=LlamaIndex+news+2024-03-12**](https://lightning.ai/lightning-ai/studios/chat-with-your-code-using-rag?__s=u4pvflfacap82vd4gibe&utm_source=drip&utm_medium=email&utm_campaign=LlamaIndex+news+2024-03-12)



A chat with your code RAG application!

**1. Custom knowledge base:**

Custom Knowledge Base: A collection of relevant and up-to-date information that serves as a foundation for RAG.  It can be a database, a set of documents, or a combination of both. In this case it's a GitHub repository that will be used as a source of truth to provide answers to user queries.

**2. Chunking:**

Chunking is the process of breaking down a large input text into smaller pieces. This ensures that the text fits the input size of the embedding model and improves retrieval efficiency.

Following code can be used to load GitHub repository data as document object in llamaindex:

|  |
| --- |
| from llama\_index.readers.github import GithubRepositoryReader, GithubClient  def initialize\_github\_client(github\_token):  return GithubClient(github\_token)  github\_client = initialize\_github\_client(github\_token)  loader = GithubRepositoryReader(  github\_client,  owner=owner,  repo=repo,  filter\_file\_extensions=(  [".py", ".ipynb", ".js", ".ts", ".md"],  GithubRepositoryReader.FilterType.INCLUDE,  ),  verbose=False,  concurrent\_requests=5,  )    docs = loader.load\_data(branch="main") |

**3. Embeddings & Embedding Model:**

A technique for representing text data as numerical vectors, which can be input into machine learning models. The embedding model is responsible for converting text into these vectors.

|  |
| --- |
| from langchain\_community.embeddings import HuggingFaceBgeEmbeddings  from llama\_index.embeddings.langchain import LangchainEmbedding  def load\_embedding\_model(  model\_name: str = "BAAI/bge-large-en-v1.5", device: str = "cuda"  ) -> HuggingFaceBgeEmbeddings:  model\_kwargs = {"device": device}  encode\_kwargs = {  "normalize\_embeddings": True  } # set True to compute cosine similarity  embedding\_model = HuggingFaceBgeEmbeddings(  model\_name=model\_name,  model\_kwargs=model\_kwargs,  encode\_kwargs=encode\_kwargs,  )    return embedding\_model  # setting up the embedding model  lc\_embedding\_model = load\_embedding\_model()  embed\_model = LangchainEmbedding(lc\_embedding\_model) |

**4. Vector Databases:**

A collection of pre-computed vector representations of text data for fast retrieval and similarity search, with capabilities like CRUD operations, metadata filtering, and horizontal scaling. By default, LlamaIndex uses a simple in-memory vector store that’s great for quick experimentation.

|  |
| --- |
| from llama\_index.core import Settings  from llama\_index.core import VectorStoreIndex  # ====== Create vector store and upload indexed data ======  Settings.embed\_model = embed\_model # we specify the embedding model to be used  index = VectorStoreIndex.from\_documents(docs) |

**5. User Chat Interface:**

A user-friendly interface that allows users to interact with the RAG system, providing input query and receiving output. We have built a streamlit app to do the same. The code for it can be found in app.py

**6. Query Engine:**

The query engine takes query string to use it to fetch relevant context and then sends them both as a prompt to the LLM to generate a final natural language response. The LLM used here is Mistral-7B which is served locally, thanks to [Ollama](http://ollama.com)! The final response is displayed in the user interface.

|  |
| --- |
| from llama\_index.llms.ollama import Ollama  from llama\_index.core import Settings  # setting up the llm  llm = Ollama(model="mistral", request\_timeout=60.0)  # ====== Setup a query engine on the index previously created ======  Settings.llm = llm # specifying the llm to be used  query\_engine = index.as\_query\_engine(streaming=True, similarity\_top\_k=4) |

**7. Prompt Template:**

A custom prompt template is use to refine the response from LLM & include the context as well:

|  |
| --- |
| qa\_prompt\_tmpl\_str = (  "Context information is below.\n"  "---------------------\n"  "{context\_str}\n"  "---------------------\n"  "Given the context information above I want you to think step by step to answer the query in a crisp manner, incase case you don't know the answer say 'I don't know!'.\n"  "Query: {query\_str}\n"  "Answer: "  )  qa\_prompt\_tmpl = PromptTemplate(qa\_prompt\_tmpl\_str)  query\_engine.update\_prompts({"response\_synthesizer:text\_qa\_template": qa\_prompt\_tmpl})  response = query\_engine.query('What is this repository about?')  print(response) |

**Conclusion**

In this studio, we developed a Retrieval Augmented Generation (RAG) application that allows you to "Chat with your code." Throughout this process, we learned about [LlamaIndex](https://www.llamaindex.ai/), the go to library for building RAG application & [Ollama](https://ollama.com/) for locally serving LLMs.

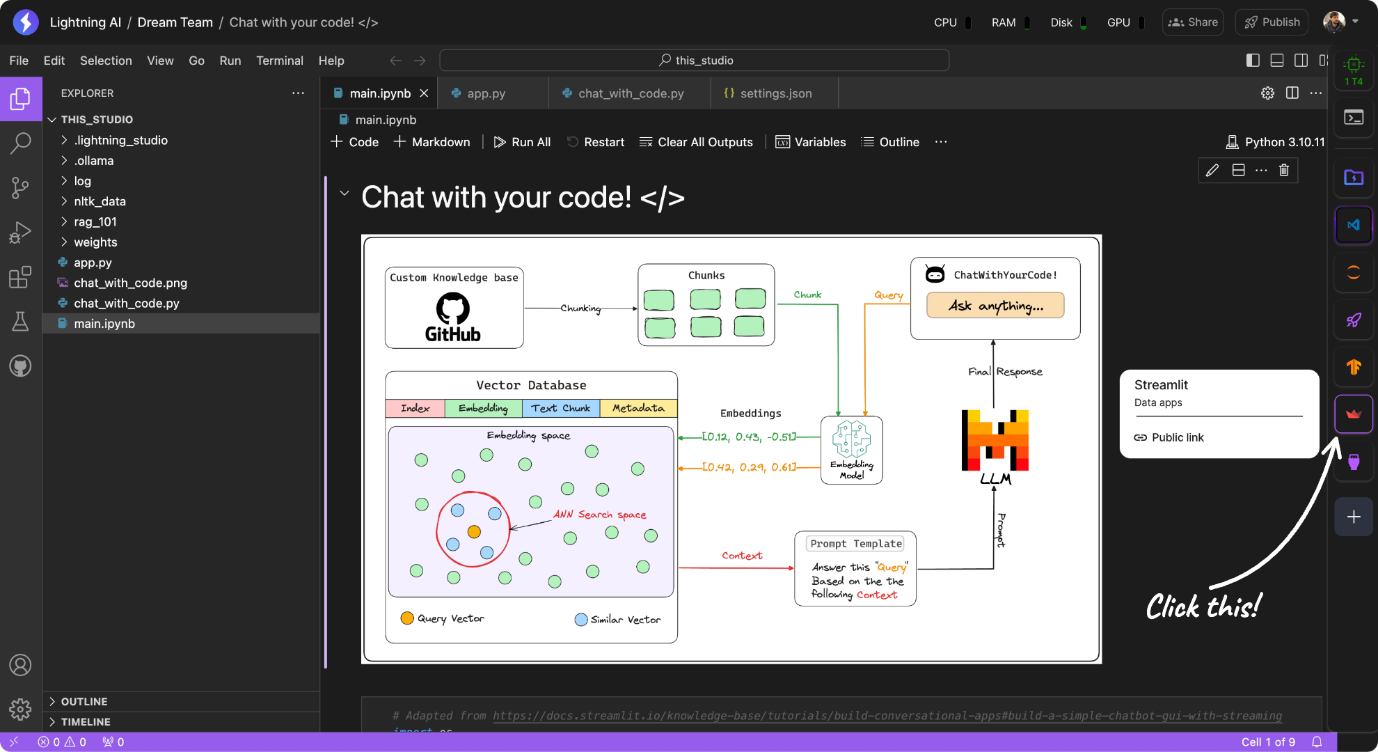
We also explored the concept of prompt engineering to refine and steer the responses of our LLM. These techniques can similarly be applied to anchor your LLM to various knowledge bases, such as documents, PDFs, videos, and more.

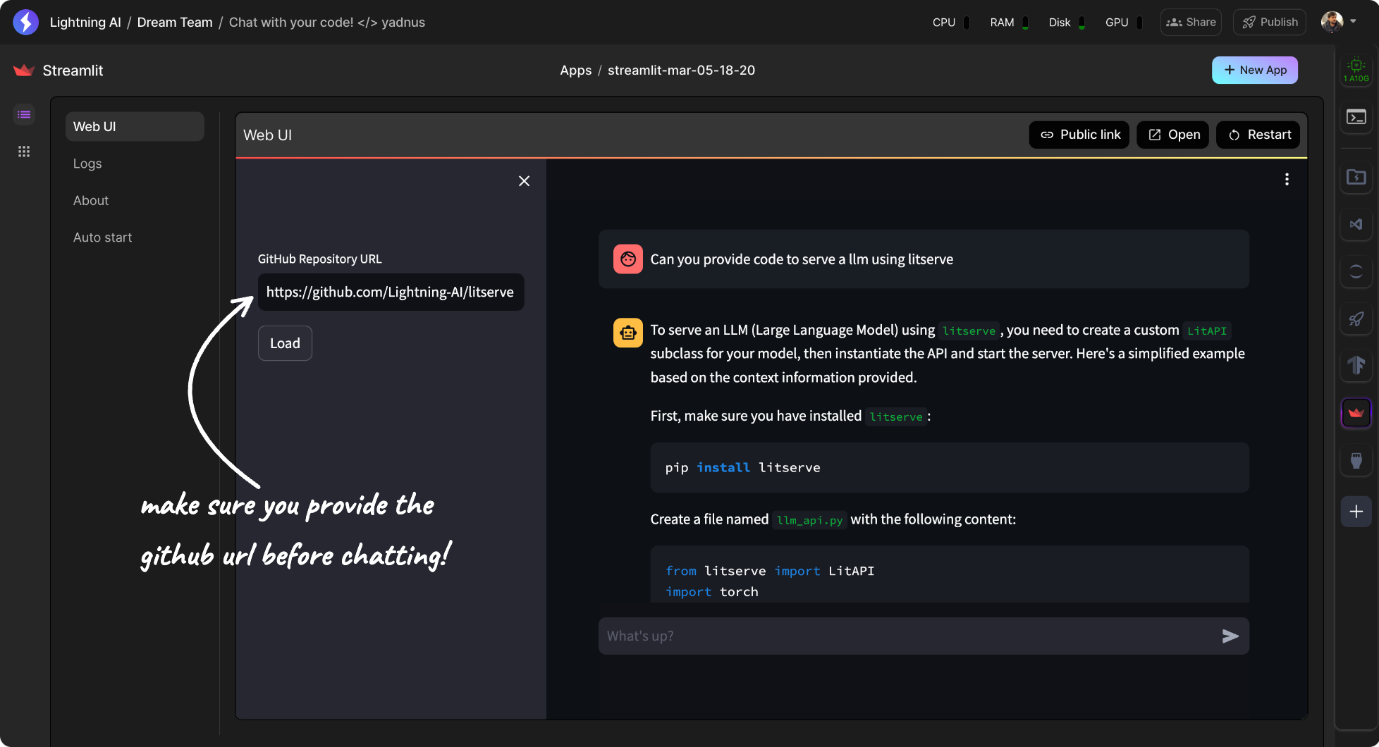
LlamaIndex has a variety of data loaders you can learn more about the same [here](https://llamahub.ai/).

Transform your interaction with GitHub repositories through a natural language interface. In this studio we are building a **"Chat with your code"** RAG application that simplifies code queries, making coding more intuitive and productive.

https://youtu.be/3V-rpBofej8                                                              Chat with your code demo!

### Try it yourself

*launching the streamlit plugin* Select an Image

Select an Image