RAG - https://python.langchain.com/docs/tutorials/rag

- Implement and explore the concepts that either new or for revision purpose

▼ Part 1

- rag implementation for q and a over text source
- using langsmith for tracing

Retrieval Techniques

- https://python.langchain.com/docs/concepts/retrieval/
- Prerequisites
  - **▼ https://python.langchain.com/docs/concepts/embedding_models/**
    - Embedding models transform human language into a format that machines can understand and compare with speed and accuracy. These models take text as input and produce a fixed-length array of numbers, a numerical fingerprint of the text's semantic meaning.
    - Leaderboard for a standard benchmark - Massive Text Embedding Benchmark (MTEB) here for objective comparisons.
    - BERT used transformer architecture to turn text inputs into vector embedding which provided gains in many of the NLP tasks
      - It was not optimized for sentecen generation tasks.
      - S-bERT was created.
    - BERT - https://www.nvidia.com/en-us/glossary/bert/
      - Bidirectional Encoder Representations from Transformers (BERT)
      - About left to right, and right to left processing by elMO which processing the entire sentence at one in both directions but does not combine it. whereas BERT unifies it, which leads to better contextual representation.
      - One of the key things in BERT, is language masking where it deletes or masks some of the words.
      - BERT's developers solved this problem by masking predicted words as well as other random words in the corpus. BERT also uses a simple training technique of trying to predict whether, given two sentences A and B, B is the antecedent of A or a random sentence.

▼ Directional Processing by elMO and BERT dimensionality of embeddings

1. Left-to-Right and Right-to-Left Contexts

In the context of language models like ELMo and BERT, the terms **left-to-right** and **right-to-left** refer to how the model processes the sequence of words in a sentence during training or inference.

- **Left-to-Right** means the model processes the sentence from the **first word** to the **last word**. It only considers words that come before a given word when generating its representation. For example, if the model is processing the word "bat" in the sentence "I hit the ball with a bat," it would process the words "I hit the ball with a" first, before considering "bat."

- **Right-to-Left** means the model processes the sentence from the **last word** to the **first word**. It only considers words that come after a given word when generating its representation. For example, in the same sentence, the model would first process the words "a bat" and then "with the ball hit I" when processing the word "bat."

Here's a simple illustration:

- **Sentence**: "The bat flew away."
- **Left-to-right processing** (For "bat"): It would consider: "The" → "bat" → "flew" → "away."
- **Right-to-left processing** (For "bat"): It would consider: "away" → "flew" → "bat" → "The."

ELMo used separate models for these two directions, while **BERT** uses **bidirectional** processing, meaning it considers both the left-to-right and right-to-left context **at the same time**. This gives a more holistic view of the word in the sentence.

2. What Drives the Size of Vector Representations?

The **size** of vector representations, also known as the **dimensionality** of the embedding, is determined by the model architecture and the **number of parameters** chosen for the model.

Several factors that influence the size of vector representations:

- **Model Architecture**:
  - In **Word2Vec** or **GloVe**, the size of the word vector is a hyperparameter you can set. Common choices are 50, 100, 200, or 300 dimensions.
  - In **ELMo** and **BERT**, the size is determined by the model architecture. For example, BERT-base uses vectors of size **768**, and BERT-large uses vectors of size **1024**.
- **Hidden Layers**:
  - **BERT**, being a transformer model, has multiple **layers** (e.g., 12 layers for BERT-base). Each layer processes information differently, and the output at each layer can be represented as a vector.
  - The size of these vectors typically corresponds to the **number of hidden units** in the transformer model. For example, BERT-base has **768 hidden units** per layer, and the final word embedding output (after processing) will also be a vector of size 768.
- **Model Capacity**:
  - Larger models (like **BERT-large**) tend to have larger vector sizes (e.g., 1024) to capture more complex patterns, but they also require more computation and memory.
  - Smaller models may use fewer dimensions (e.g., 256, 512) for efficiency but may not capture as much detail in the word representations.

In summary:

- **Left-to-Right and Right-to-Left** refer to how the model processes words sequentially from different directions. ELMo used these separately, while BERT combines both for a deeper understanding.
- **Vector Size**: The size of the vector representation (like 768 or 1024 in BERT) is determined by the architecture of the model (number of layers, hidden units, etc.). It impacts the model's ability to capture complex patterns but also affects computational requirements.

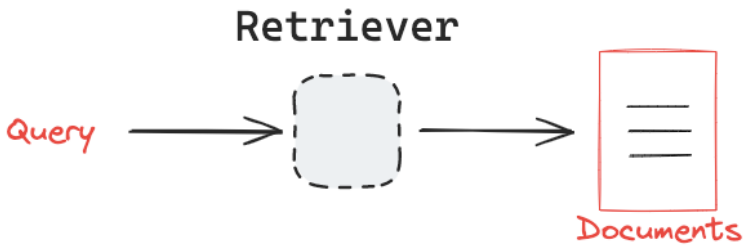
- more detail - https://research.google/blog/open-sourcing-bert-state-of-the-art-pre-training-for-natural-language-processing/
- paper for later - https://arxiv.org/abs/1810.04805

- Langchain offers a common interface, for these model to embed the documents or just text using `embed_documents` , `embed_query` methods

- Langchain integration supports for different providers - https://python.langchain.com/docs/integrations/text_embedding/

▼ Retrievers

- The interface takes in an input query which is a string and returns the list of documents



- This class is a `Runnable` type, which means the Runnable class methods are available and we can use `invoke` method

> 💡 https://python.langchain.com/docs/how_to/lcel_cheatsheet/

- It requires that the method `_get_relevant_documents` method is implemented.

https://www.pinecone.io/learn/series/langchain/langchain-expression-language/
the LECL usage broken down

▼ LECL LangChain Expression Language

- https://www.pinecone.io/learn/series/langchain/langchain-expression-language/
- Traditional syntax for calling a chain

```
chain = LLMChain(
    prompt=prompt,
    llm=model,
    output_parser=output_parser
)
```

- This uses pipe, which is | operator, it is the `__or__` method on the class.
  - It take the output from left operations and passes it as input to the neighboring right operations.
  - Sample example

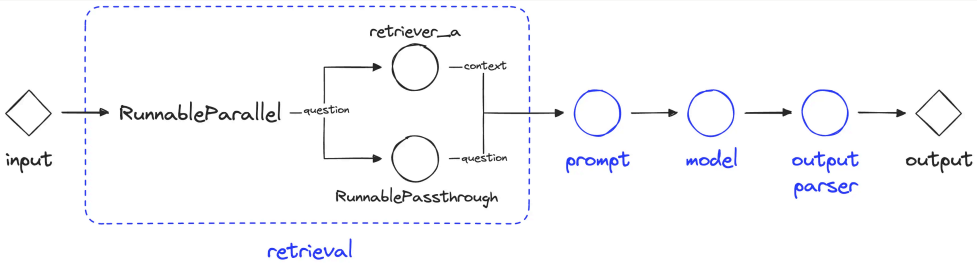
```
def add_five(x):
    return x + 5

def multiply_by_two(x):
    return x * 2

# wrap the functions with Runnable
add_five = Runnable(add_five)
multiply_by_two = Runnable(multiply_by_two)

# run them using the object approach
chain = add_five.__or__(multiply_by_two)
chain(3)  # should return 16
```

- A simple RAG pipeline is created.
  - with 2 retrievals, that have half of the information.
  - prompt the model using the first retrieval. (it does not have the data)
- New concepts



  - RunnableParallel
    - `RunnableParallel` object allows us to define multiple values and operations, and run them all in parallel.

    - RunnablePassThrough
        - The `RunnablePassthrough` object is used as a "passthrough" take takes any input to the current component (retrieval) and allows us to provide it in the component output via the "question" key.
- when the chain that is defined using the pipe operator is then invoked. using on vector_a as the context. The llm does not have an answer. which is expected.
    - Fortunately, `Runnable Parallel` allows us to provide multiple contexts, and this time we can pass both halves of the information.

```
prompt_str = """Answer the question below using the context:

Context:
{context_a}
{context_b}

Question: {question}

Answer: """
prompt = ChatPromptTemplate.from_template(prompt_str)

retrieval = RunnableParallel(
    {
        "context_a": retriever_a, "context_b": retriever_b,
        "question": RunnablePassthrough()
    }
)

chain = retrieval | prompt | model | output_parser
```

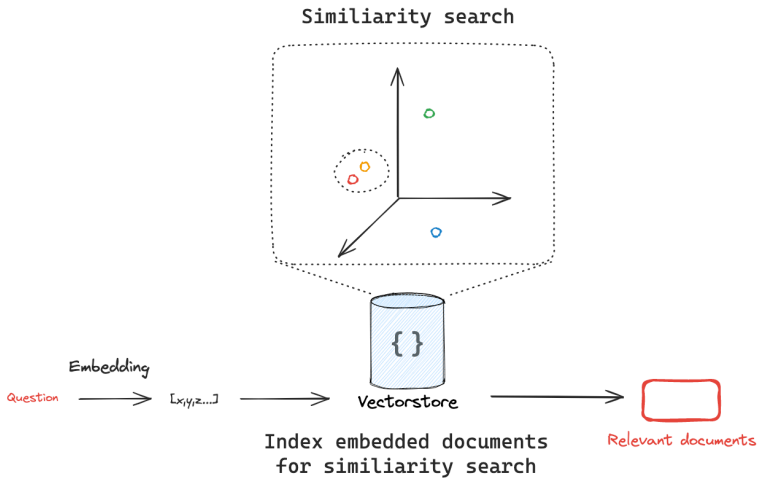
- The `RunnableLambda` is a LangChain abstraction that allows us to turn Python functions into pipe-compatible functions,
    - THE PIPE OPERATOR CAN BE USED WHEN A FUNCTION IS CONVERTED INTO `RUNNABLELAMBDA`

- Retrievers do not store documents
- Common types of retrieval systems
    - APIs
    - RDBMS
        - In these cases, query analysis techniques to construct a structured query from natural language is critical.
        - For example, you can build a retriever for a SQL database using text-to-SQL conversion. This allows a natural language query (string) retriever to be transformed into a SQL query behind the scenes.
    - Lexical Search
        - Many search engine use lexical search, words in the query to match with words in the documents

        - Example: TF-IDF, or BM-25
        - There' elastic search integration available for langchain
    - Vector Store
        - Powerful when dealing with unstructured data
        - You can convert a vector store in a retrieval using the `as_retriever()` method
- Advanced search techniques
    - Ensemble
        - It involves combines results of different retrievals that are good at finding different aspects of the query with the documents.
        - pass the retrievers in the list as input, specify the weights for each one.
        - Finally, reranking is done, using another algorithm
    - Source Document retention
        - The main idea is — after the indexing is done, being able to trace back to the original document .
        - 2 types
            - ParentDocument
                - retriever links document chunks from a text-splitter transformation for indexing while retaining linkage to the source document.
            - MultiVector
                - This may use LLM to store different transformation of the source document, perhaps captures things that provide various aspects of the source, using hypothetical questions or it can just be summary.

▼ Vector stores

- Embeddings
- These are used to index documents(unstructured) into vector representation (called embeddings) and then use for efficient retrieval for a given query.



- LangChain provides a standard interface for working with vector stores, allowing users to easily switch between different vectorstore implementations.
- The interface consists of basic methods for writing, deleting and searching for documents in the vector store.
- The key methods are:
  - `add_documents`: Add a list of texts to the vector store.
  - `delete`: Delete a list of documents from the vector store.
  - `similarity_search`: Search for similar documents to a given query.
- Provide the document ids when adding or deleting documents, as it makes the operations efficient
- Similarity Search
  - Now that documents are represented as some Vectors. Given a query in text format, is it converted into an embedding and a similarity check algorithm is run to score the matches and then rank the results.
  - Score or Metrics
    - Cosine
    - Dot product
    - Euclidean distance

```
## example code snippets

from langchain_core.vectorstores import InMemoryVectorStore
# Initialize with an embedding model
vector_store = InMemoryVectorStore(embedding=SomeEmbeddingModel())

from langchain_core.documents import Document

document_1 = Document(
    page_content="I had chocalate chip pancakes and scrambled eggs for breakfast this morning.",
    metadata={"source": "tweet"},
)

document_2 = Document(
    page_content="The weather forecast for tomorrow is cloudy and overcast, with a high of 62 degrees.",
    metadata={"source": "news"},
)

documents = [document_1, document_2]

vector_store.add_documents(documents=documents)

vector_store.add_documents(documents=documents, ids=["doc1", "doc2"])
vector_store.delete(ids=["doc1"])

query = "my query"
docs = vectorstore.similarity_search(query)
```

- MetaData filtering

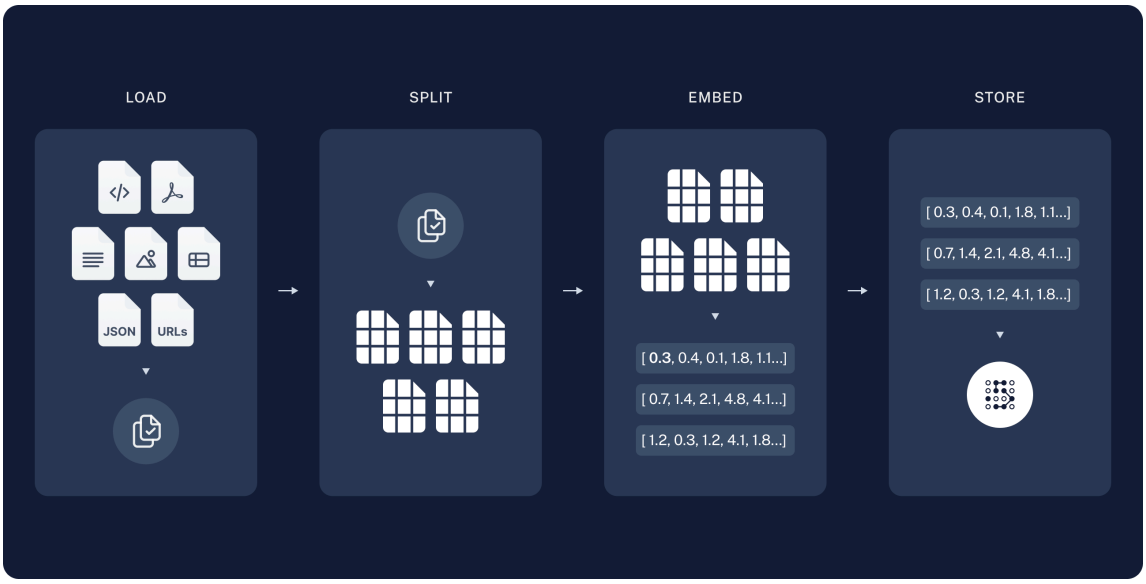
    - the interface offers filtering based on properties available in the documents. These are provided during creation of documents, as the metadata argument

▼ **Text splitters**

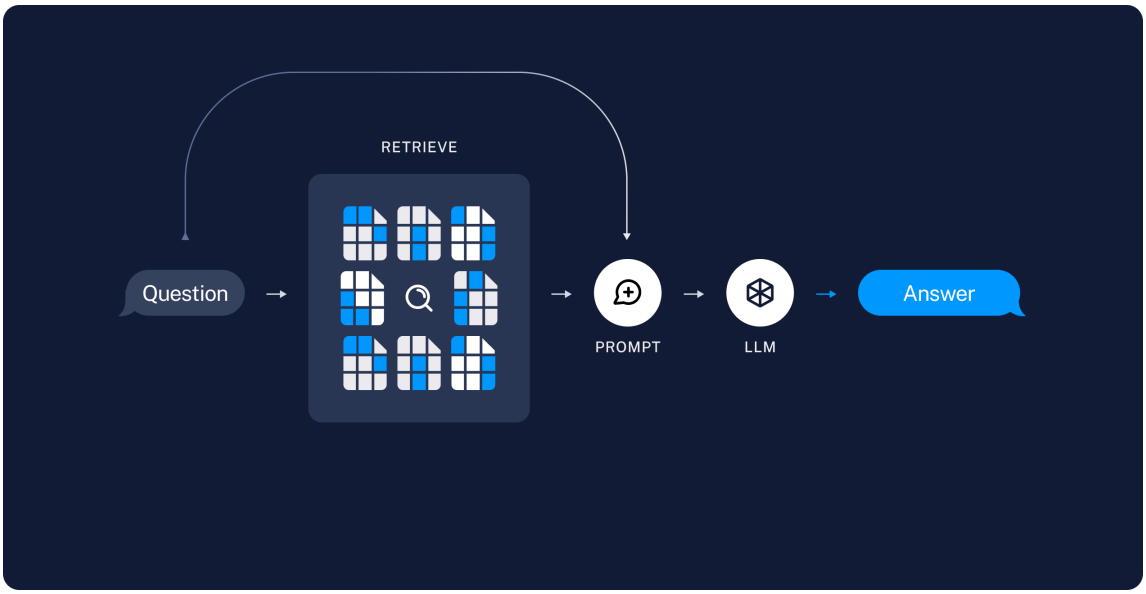
- Splitting is done for the following reasons
    - Limited windows size of the model
    - consistent length of documents
    - optimizing training performance
    - improves the quality of vectors/representation
- Types
    - Length
        - Number of characters, or tokens
    - Text
        - It tries to keep related information together, like a paragraph, sentence, words.
    - Document
        - If it;s a markdown, then split based on the different tags it uses.
        - JSON, HTML,
    - Semantic
        - While other approaches use document or text structure as proxies for semantic meaning, this method directly analyzes the text's semantics.

Semantic Search

- https://python.langchain.com/docs/tutorials/retrievers/
- This whole thing can be broken into the following steps
    1. loading
        a. use document loaders
    2. breaking into chunks
        a. some sort of splitters
        b. to ensure it fits into the model’s finite context windows size
    3. Storing into vector dbs



4. retrieval:
    a. given a user input, retrieves the relevant splits
5. generate
    a. user prompt, along with the document as the input to the models to generate an answer



- Once we've indexed our data, we will use LangGraph as our orchestration framework to implement the retrieval and generation steps.

Coding Part 1

- After getting the langsmith setup ready
    - install packages
    - Store the keys

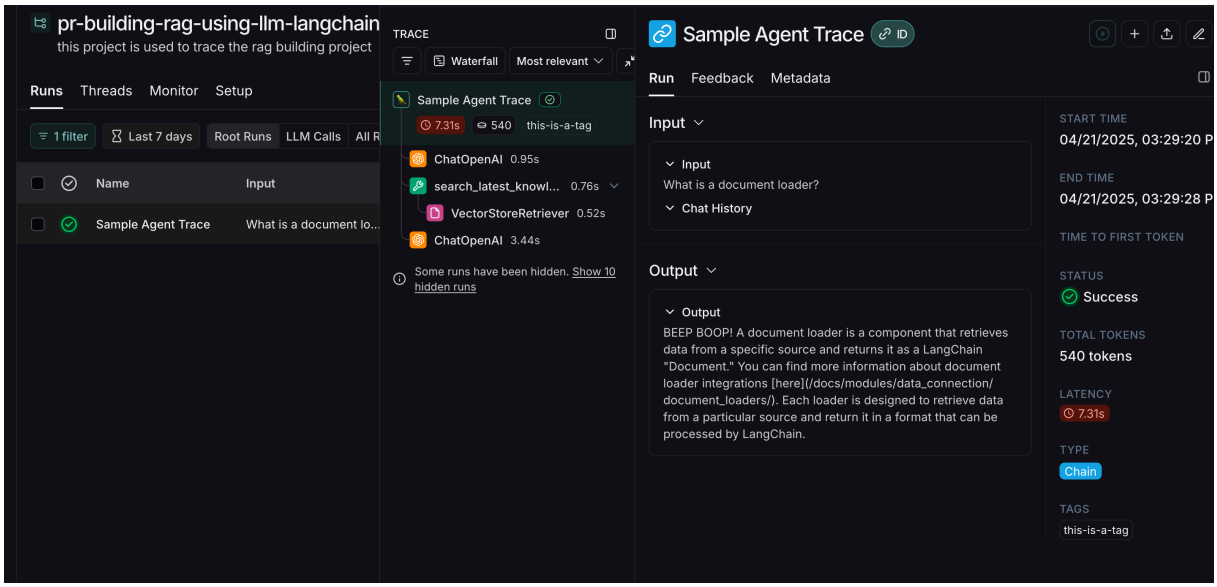
- Now, I require 3 components
  - Model
  - Embedding
  - Vectorstore
- The corresponding commands are

```
pip install -qU "langchain[openai]"
pip install -U langchain langchain-openai
pip install -qU langchain-core
```

GOAL

- build an app that answers questions about the website's content. The specific website we will use is the LLM Powered Autonomous Agents blog post by Lilian Weng, which allows us to ask questions about the contents of the post.

- Update the LLM init code to use the langsmith
- Trace output
- https://smith.langchain.com/



- Relation between Embedding and Model
  - In essence: The embedding model provides the "language" for the RAG system to understand and compare information, while the LLM (and potentially rerankers) handle
    the retrieval and generation aspects.

▼ Purpose of some code blocks

Indexing

- In this case, WebBaseLoader, is used to load a webpage, and we use soup to focus on relavent tags such as **“post-content”, “post-title”, or “post-header”**

> **In this case we'll use the WebBaseLoader, which uses urllib to load HTML from web URLs and BeautifulSoup to parse it to text. We can customize the HTML → text parsing by passing in parameters into the BeautifulSoup parser via bs_kwargs (see BeautifulSoup docs).**

- The splits are to be converted into embeddings, this is done by added them to the vector store
  - in this case, InMemoryDataStore is being used
  - along with OpenAI embedding

```
embeddings = OpenAIEmbeddings(model="text-embedding-3-large")
```

```
vector_store.add_documents(documents=all_splits)
```

- At this point we have a query-able vector store containing the chunked contents of our blog post. Given a user question, we should ideally be able to return the snippets of the blog post that answer the question.

- About the prompt using hub.pull
  - https://docs.smith.langchain.com/prompt_engineering/how_to_guides/langchain_hub
    - available prompts that can be pulled to chat with the LLM.
    - Depending on the usecase, different prompts are available.
      - Since, we are trying to build rag, using https://smith.langchain.com/hub/rlm/rag-prompt, to predefine things for us
      - it defines the question, context that are going to be passed in the prompt, and the expected output from the LLM
  - `StateGraph` is part of the langchain graph module. It helps using a State, which is essentially a dictionary, with the keys being the variable/placeholders that are being used by the LLM (originating from the prompt that's defined)
    - defines the order of operations
    - and add edges
    - compile, so that it becomes a Runnable object

Retrieval and Generation

- To use LangGraph, we need to define three things:
  1. The state of our application;
  2. The nodes of our application (i.e., application steps);
  3. The "control flow" of our application (e.g., the ordering of the steps).

  State:
  - of our application controls what data is input to the application, transferred between steps, and output by the application.
  - It is typically a `TypedDict`, but can also be a Pydantic BaseModel

Query Analysis

- The state can be extended to define the output structure using a pydantic model or typedDict

- If pydantic model is used, the return type will be a runnable object else a json string format.
- Add it as a step before the retrival
- - https://python.langchain.com/docs/how_to/structured_output/
- Query analysis employs models to transform or construct optimized search queries from raw user input. We can easily incorporate a query analysis step into our application. F
- https://python.langchain.com/docs/how_to/#query-analysis

▼ Part 2

- Recap: Part 1 was about RAG and a minimal implementation.
- Part 2 extends the implementation to accommodate conversation-style interactions and multi-step retrieval processes.
- So far, what has been built is
  - loading documents
  - splitting into chunks
  - indexing
    - use some embedding
    - convert the chunks into vectors
  - store the embedding on chunks into vector store
  - Create a state to capture the placeholders which uses dict to capture results during various steps of the process
    - Use stategraph to list the steps
      - retrieval : to fetch the relavant documents using some similarity search with the given query
      - generation: invokes the llm
    - Add retrieval as the first node in the graph
    - compile it
  - Side note
    - the state can be extended to define structured output, and this can be added to the state, and as the first step in the stategraph

---

💡 Reading material

- Return sources: Learn how to return source documents
- Streaming: Learn how to stream outputs and intermediate steps
- Add chat history: Learn how to add chat history to your app
- Retrieval conceptual guide: A high-level overview of specific retrieval techniques

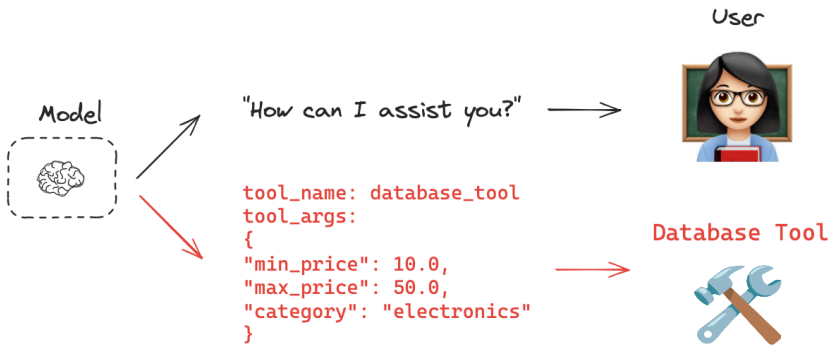
---

- There are 2 approaches
  - Chains
  - Agents

1. Chains
   - In the Part 1 of the RAG tutorial, we represented the user input, retrieved context, and generated answer as separate keys in the state. Conversational experiences can be naturally represented using a sequence of messages.
     In addition to messages from the user and assistant, retrieved documents and other artifacts can be incorporated into a message sequence via
     tool messages. This motivates us to represent the state of our RAG application using a sequence of messages. Specifically, we will have
     1. User input as a `HumanMessage`;
     2. Vector store query as an `AIMessage` with tool calls;
     3. Retrieved documents as a `ToolMessage`;
     4. Final response as a `AIMessage`.
   - This format is common and is availble within langchain
   - The graph is essentially modified so that,
     - the query are rewritten by the model to provide more context. it uses memory or chat history.
     - for example: a prompt may refer to “it”, the tooling allows the LLM to rewrite the query to specify that “it” means “tool decomposition” which enables for better responses

---

Tooling



- https://python.langchain.com/docs/concepts/tool_calling/
- Generally, an LLM responds to user prompts directly. (meaning it uses the documents used during training or the ones provided as part of the rag pipeline).
  - But we also need it interact with external systems such as APIs, and format the response in a way that LLM can consume it
- utilizing Tooling

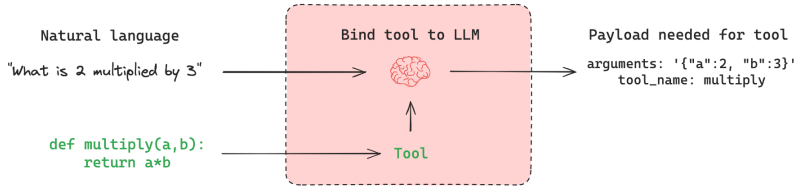
1. Tool creation
    a. is done using the `@tool` annotation
2. Model awareness or Tool binding
    a. Specify the expected schema by the tool
    b. Associate the tool with the model, so that the model is now aware that a tool is available
3. Tool Calling
    a. Model can choose to call and specify the arguments, usually from the context of the conversation
4. Tool Execution
    a. could be an API request via HTTP.
    b. invocation

```
# Tool creation
tools = [my_tool]
# Tool binding
model_with_tools = model.bind_tools(tools)
# Tool calling
response = model_with_tools.invoke(user_input)


from langchain_core.tools import tool

@tool
def multiply(a: int, b: int) -> int:
   """Multiply a and b."""
   return a * b
```

- Adding another tool to a model that supports tooling



    - defines the function with `@tool` and then bind it and use the new reference of the model to invoke with a prompt

- Responses after the invocation is complete.
    - The output result will be an `AIMessage`. But, if the tool was called, result will have a `tool_calls` attribute. This attribute includes everything needed to execute the tool, including the tool name and input arguments:
    - result would be an `AIMessage` containing the model's response in natural language (e.g., "Hello!").

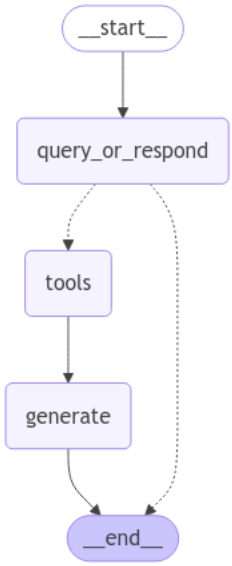
TOOL EXECUTION

- Tools implement the Runnable interface, which means that they can be invoked (e.g., `tool.invoke(args)`) directly.

- LangGraph offers pre-built components (e.g., `ToolNode`) that will often invoke the tool in behalf of the user.
- ToolNode
  - Creates a graph that works with a chat model that utilizes tool calling.

- Leveraging tool-calling to interact with a retrieval step has another benefit, which is that the query for the retrieval is generated by our model. This is especially important in a conversational setting, where user queries may require contextualization based on the chat history.
- 3 nodes are involved

> 1. A node that fields the user input, either generating a query for the retriever or responding directly;
> 2. A node for the retriever tool that executes the retrieval step;
> 3. A node that generates the final response using the retrieved context.



Tools

- https://python.langchain.com/docs/concepts/tools/

Tool interface

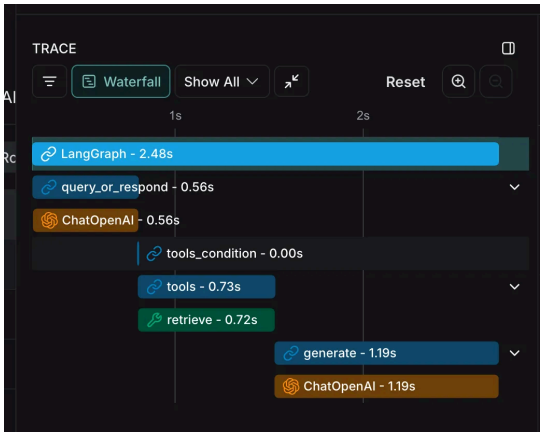
- The tool interface is defined in the BaseTool class which is a subclass of the Runnable Interface.
- The key attributes that correspond to the tool's **schema**:
  - **name**: The name of the tool.
  - **description**: A description of what the tool does.
  - **args**: Property that returns the JSON schema for the tool's arguments.

- The key methods to execute the function associated with the **tool**:
  - **invoke**: Invokes the tool with the given arguments.
  - **ainvoke**: Invokes the tool with the given arguments, asynchronously. Used for async programming with Langchain.

Tool Artifact

- This is essentially data or information that we do not want to send the model as a response directly, but it should be available to the downstream task to consume if they want.
- For example if a tool returns a custom object, a dataframe or an image, we may want to pass some metadata about this output to the model without passing the actual output to the model. At the same time, we may want to be able to access this full output elsewhere

Trace



STATEFUL MANAGEMENT OF CHAT HISTORY

- Reading material for deep dive - https://langchain-ai.github.io/langgraph/concepts/persistence/
- Langchain recommends using langgraph as it provides a way to add memory to our application
- LangGraph has a built-in persistence layer, implemented through checkpointers.
  - When you compile graph with a checkpointer, the checkpointer saves a `checkpoint` of the graph state at every super-step.
  - Those checkpoints are saved to a `thread`, which can be accessed after graph execution.

- https://langchain-ai.github.io/langgraph/concepts/memory/

2. Agents
   a. Unlike the chain, wherein once a tool is called it's response is either used or discarded once the invocation is complete

b. Agents can call the tool multiple times.

Agents

- By themselves, language models can't take actions - they just output text. Agents are systems that take a high-level task and use an LLM as a reasoning engine to decide what actions to take and execute those actions.
  - Prebuilt agents
    - https://langchain-ai.github.io/langgraph/reference/prebuilt/#langgraph.prebuilt.chat_agent_executor.create_react_agent
  - Agent architectures
    - https://langchain-ai.github.io/langgraph/concepts/agentic_concepts/
  - ReAct is a popular general purpose agent architecture that combines these expansions, integrating three core concepts.
    1. `Tool calling`: Allowing the LLM to select and use various tools as needed.
    2. `Memory`: Enabling the agent to retain and use information from previous steps.
    3. `Planning`: Empowering the LLM to create and follow multi-step plans to achieve goals.

- Good Article - https://blog.langchain.dev/how-to-think-about-agent-frameworks/

- Using Langchain
  - https://langchain-ai.github.io/langgraph/agents/overview/?ref=blog.langchain.dev#what-is-an-agent