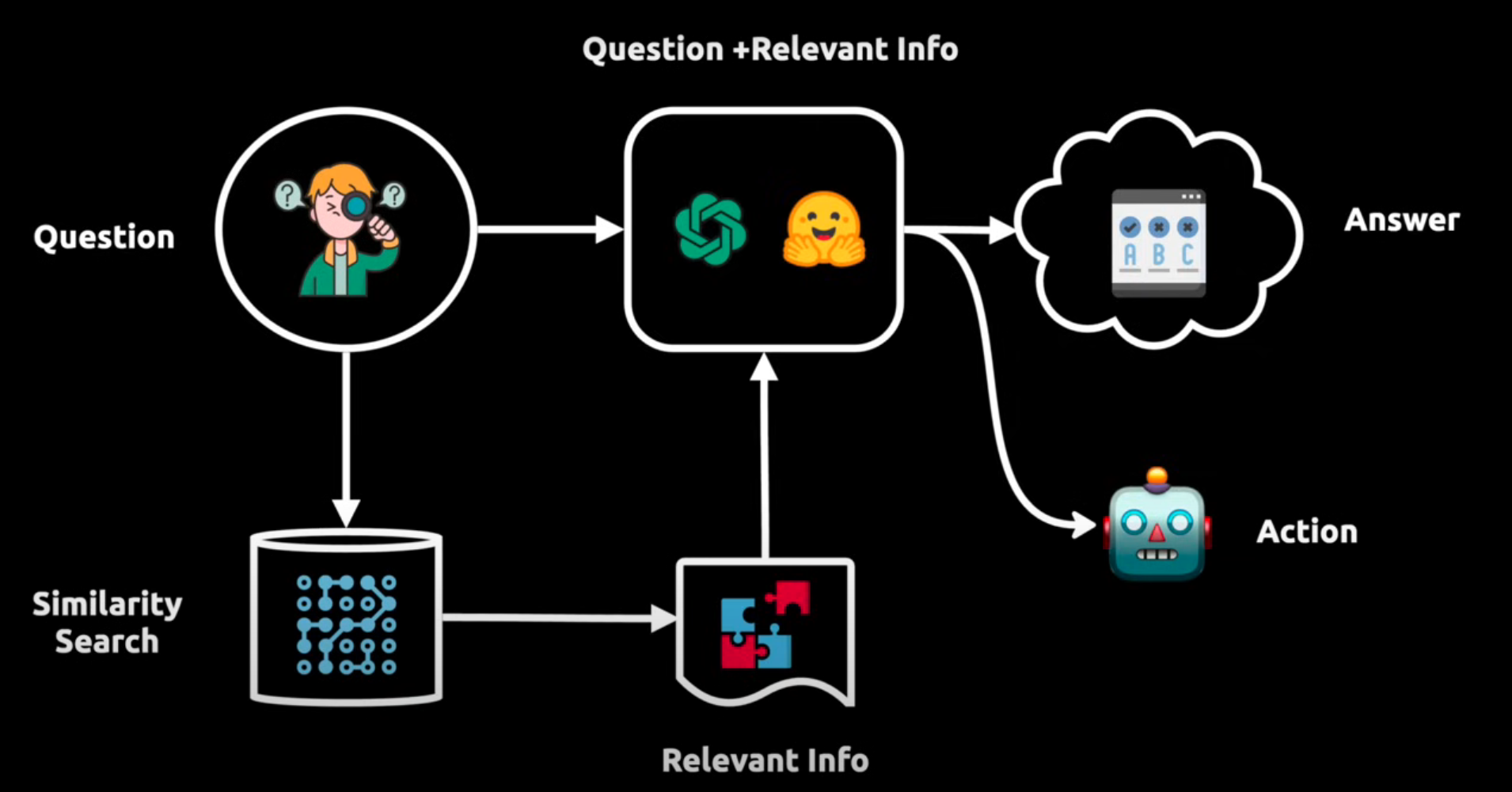
Learning and Applying LangChain and LangGraph with Gmail Integration

## Introduction to LangChain:

LangChain is an open-source framework that allows ai developers to combine LLMS like gpt-4 with external sources of computation and data.  
  
LangChain can be used as a python package or with javascript (Node.js)

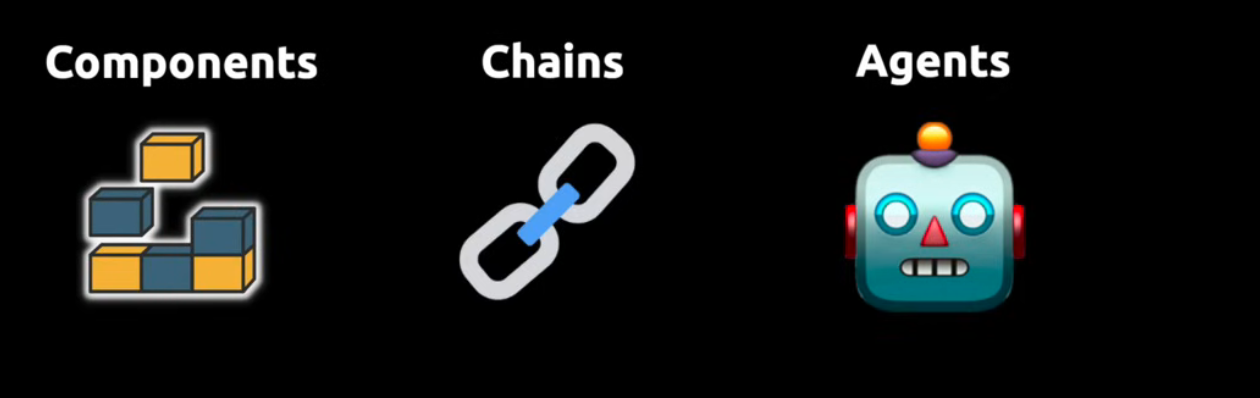
We’re essentially using LangChain to allow our LLMs of choice to have access to our own documents and data leveraging the LLMs’ general knowledge and our own sources of information which are stored in a vector database (as embeddings).



**This figure shows how when we ask the agent to the agent, we’re first passing the question into our vector database which does a similarity search and retrieves the relevant information regarding the question. This output is then passed along with the question itself into the LLM which then returns either an answer or takes an action depending on the LangChain pipeline we’ve built.**

Practical use cases include flight booking, money transfers, studying, analytics, and an agent that manages your gmail for you which we will be working on.  
  
Value proposition of LangChain:

Main value proposition can be divided into three main concepts as seen in the image:

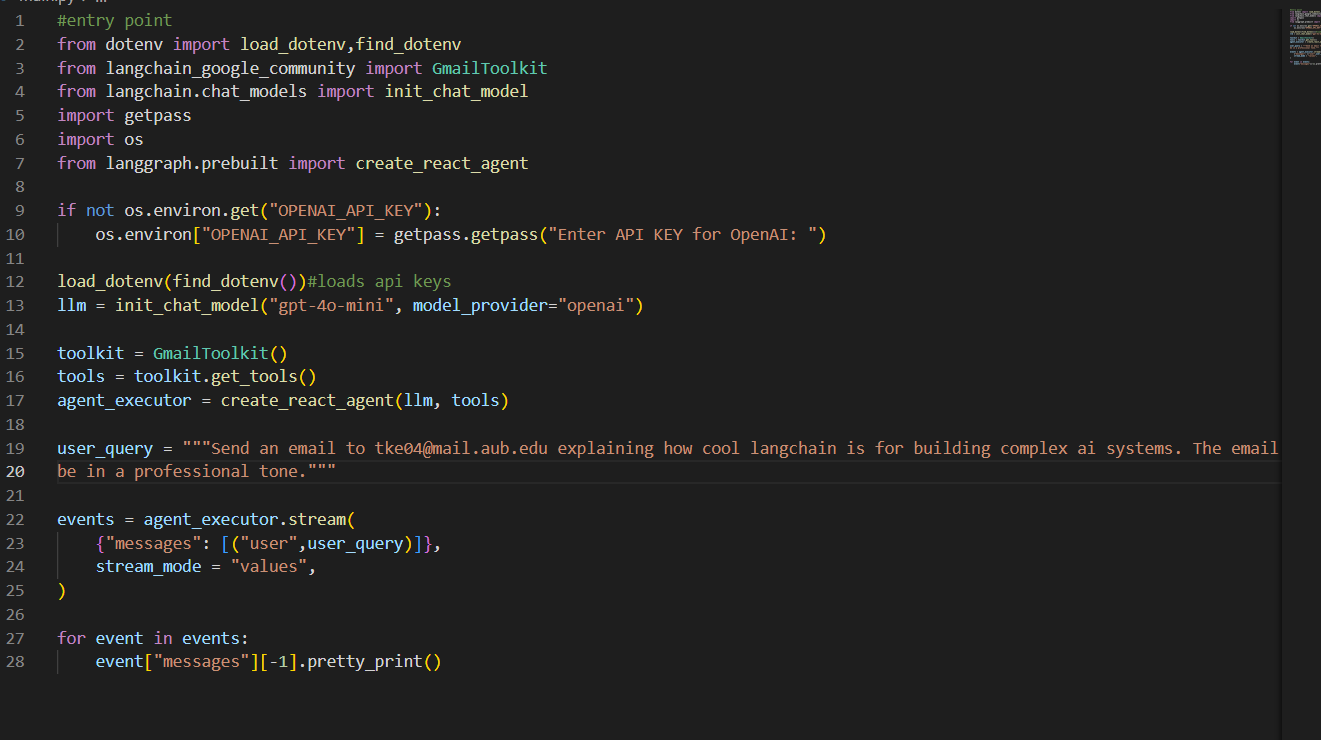


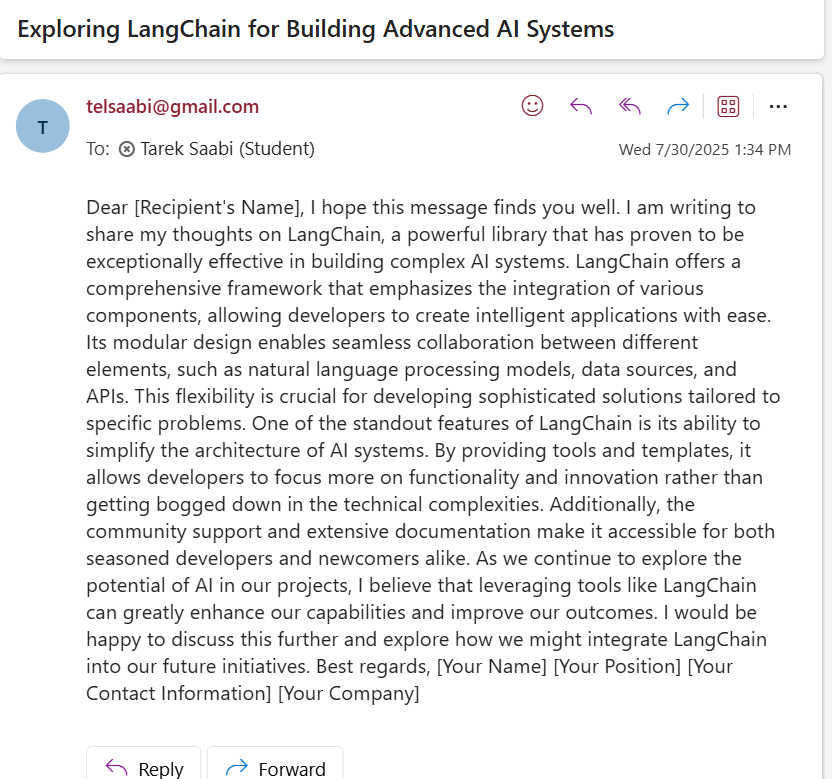
|  |  |  |
| --- | --- | --- |
| LLM Wrappers: allows us to connect to LLMs like gpt-4 or hugging face models. | Assemble components to solve a specific task, such as finding info in a book | Agents allow LLMs to interact with their environment. For example, making API requests with a specific action. |
| Prompt Templates: hardcode text as input to the LLMs (giving it specific instructions) |  |  |
| Indexes for relevant information retrieval |  |  |

## First Implementation: Following the Gmail Toolkit Document:

All the necessary packages we will be using in our python code can be found in the requirements.txt file.  
  
 After installation, we create an .env file containing our openai api key.

Following the tutorial on [Gmail Toolkit | 🦜️🔗 LangChain](https://python.langchain.com/docs/integrations/tools/gmail/), we are successful in sending an email using langchain as we can see in the given code:





## Learning LangGraph:

With the basics out of the way, we need to now implement a **graph-based** workflow that can retrieve and analyze information as well as send emails based on user query.

Different operations will have separate nodes.

**LangGraph Architecture:**

**Nodes**: Individual functions that perform specific tasks

**Edges**: Connections between nodes that define workflow

**State**: Shared data structure passed between nodes

**Graph**: The complete workflow definition

**User workflow:**

User query -> initial state -> graph execution -> final response

We need to define EmailState which will be used to share information between the nodes.

Our EmailState will be defined as a TypedDict which is basically an improved dictionary that lets us define the values clearly. EmailState will be in **state.py**

Other than saving our state, we need a file that directly interacts with the Gmail API tools. This file will be **utils.py. ->nodes call these methods to interact with Gmail.**

Now we need to define our graph workers. Each node in a graph is meant to do the following:

1. Receive the current state

2. Perform a specific task

3. Update state

4. Returns modified state

This will be done through the router node which will help determine what action to take based on user query which returns a word for which action to be taken next. ->**nodes.py**

Then, we need to build the graph itself and set up the routing decisions -> **graph.py**

Then finally we need a main application to connect everything together **-> main.py**

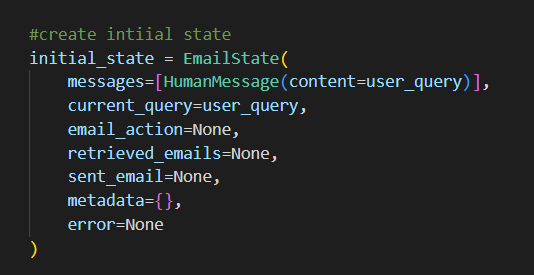
**For the code implementation, I’ve written comments along with the code to understand the implementation better.**

## How does the code work? High level walkthrough:

First, we look at main.py, which takes in user input -> this is the current\_query:

assistant = GmailAssistant()

response = assistant.process\_query("Find emails from telsaabi@gmail.com about xyz")

process\_query creates the initial state 

and runs the graph through graph.invoke(initial\_state) and we save the output of the final state into a variable “final\_state”

final\_state = self.graph.invoke(initial\_state)

from this we get the details of the different states

ok lets backtrack, we’ve just created our initial state. Now we need to execute the programs to complete the user request.  
the first thing that happens is that the graph will execute the routernode to determine which action to take (send or retrieve email)  
after it has decided, we go to the following node, execute search/retrieve depending on the action word chosen from the user query by the llm

after that, we will visit one more node which is the respond node which tells the llm to write a response summarizing the user actions taken.