# C API-KEY

gsk\_82VobCBNDDfoHV1mNqFoWGdyb3FYI3jbGSkzU5V7KHJ485atAxCy

**Resources**

RESOURCES:

[Build a Retrieval Augmented Generation (RAG) App: Part 1 | 🦜️🔗 LangChain](https://python.langchain.com/docs/tutorials/rag/)

[Starter Tutorial (Using OpenAI) - LlamaIndex](https://docs.llamaindex.ai/en/stable/getting_started/starter_example/)

1. Use a new Colab instance or [follow.](https://colab.research.google.com/drive/1nlcEyGpwP3EyjD2XJ3GQVdA-u1qZ0dOT?usp=sharing)

[Colab](https://colab.research.google.com/)

2. Get a new API KEY from Groq

* **Sign Up/Log In:** If you don't have an account, create one on the Groq Cloud website. If you already have an account, log in.
* **Navigate to API Keys:** Once logged in, go to the "API Keys" section, usually found in the left-side navigation panel.
* **Create a New Key:** Click the "Create API Key" button.
* **Name Your Key:** In the pop-up window, enter a descriptive name for the key (e.g., "AI Content Labs") in the "Display name for the key" field.
* **Submit:** Click "Submit" to generate the API key.
* **Copy the Key:** Carefully copy the generated API key, as you will not be able to view it again once you leave the page.
* **Store Safely:** Treat your API key as a confidential piece of information, similar to a password, and store it securely.

3. Load the env variables first:

* Best practice to load these variables using a .env file. For demo purpose we can load it in the COLAB.
* Please use the API key you got

LANGSMITH\_TRACING="true"

LANGSMITH\_ENDPOINT="https://api.smith.langchain.com"

LANGSMITH\_API\_KEY="lsv2\_pt\_bde7f95f197c4252984dc8b05a0d9726\_832920537a"

LANGSMITH\_PROJECT="pr-stupendous-anywhere-42"

GROQ\_API\_KEY="<YOUR\_API\_KEY>"

# 1. PDF READER

4. We need to install following packages.

!pip install -qU pypdf

!pip install -qU langchain\_community

!pip install -qU tiktoken

!pip install langgraph

!pip install -qU langchain-core

5. Let’s then use the PyPDFLoader to load the file and RecursiveCharacterSplitter to split the data into chunks

from langchain\_community.document\_loaders import PyPDFLoader

from langchain\_text\_splitters import RecursiveCharacterTextSplitter

# Load the PDF

loader = PyPDFLoader('./EducationinSriLanka.pdf')

pages = []

# Each page is a Document object

async for page in loader.alazy\_load():

pages.append(page)

# Split the text into chunks

text\_splitter = RecursiveCharacterTextSplitter(chunk\_size=1000, chunk\_overlap=200)

# Pass the list of Document objects

all\_splits = text\_splitter.split\_documents(pages)

# Print first 5 chunks

print(all\_splits[:5])

6. Then let’s use the langchain-huggingface package to implement embeddings

!pip install -qU langchain-huggingface

7. Let’s then create Embeddings

# Create Embeddings

from langchain\_huggingface import HuggingFaceEmbeddings

embeddings = HuggingFaceEmbeddings(model\_name="sentence-transformers/all-mpnet-base-v2")

8. Let’s create an In-memory VectorStore

# Create an In-Memory Vector Store

from langchain\_core.vectorstores import InMemoryVectorStore

vector\_store = InMemoryVectorStore(embeddings)

9. Add the document chunks to the vector store

document\_ids = vector\_store.add\_documents(documents=all\_splits)

10. Let’s install the Groq package

!pip install -qU "langchain[groq]"

11. Let’s implement the Groq with llama-8b-8192 model

import getpass

import os

if not os.environ.get("GROQ\_API\_KEY"):

os.environ["GROQ\_API\_KEY"] = GROQ\_API\_KEY

from langchain.chat\_models import init\_chat\_model

llm = init\_chat\_model("llama3-8b-8192", model\_provider="groq")

Note: can load other models other than "llama3-8b-8192" like deep-seek 😉

12. If you want you can use OPENAI

# import getpass

# import os

# if not os.environ.get("OPENAI\_API\_KEY"):

# os.environ["OPENAI\_API\_KEY"] = OPENAI\_API\_KEY

# from langchain.chat\_models import init\_chat\_model

# llm = init\_chat\_model("gpt-3.5-turbo", model\_provider="openai")

13. Then we can invoke the Chat to ask the questions from our PDF

from langchain import hub

from langchain\_core.documents import Document

from langgraph.graph import START, StateGraph

from typing\_extensions import List, TypedDict

prompt = hub.pull("rlm/rag-prompt")

# Define state for application

class State(TypedDict):

question: str

context: List[Document]

answer: str

# Define application steps

def retrieve(state: State):

retrieved\_docs = vector\_store.similarity\_search(state["question"])

return {"context": retrieved\_docs}

def generate(state: State):

docs\_content = "\n\n".join(doc.page\_content for doc in state["context"])

messages = prompt.invoke({"question": state["question"], "context": docs\_content})

response = llm.invoke(messages)

return {"answer": response.content}

# Compile application and test

graph\_builder = StateGraph(State).add\_sequence([retrieve, generate])

graph\_builder.add\_edge(START, "retrieve")

graph = graph\_builder.compile()

14. Let’s ask questions

response = graph.invoke({"question": "How many students and schools in Central Province?"})

print(response["answer"])

EXERCISE:

Try other models as well 😉

# 2. WEB SCRAPING

This is the url that you scrape: <https://lilianweng.github.io/posts/2023-06-23-agent/>

You can try other sites as well 😉

15. Similarly we can use WEB Scraping with beautiful soup to set up a Q&A for a web page. See the below code:

import bs4

from langchain import hub

from langchain\_community.document\_loaders import WebBaseLoader

from langchain\_core.documents import Document

from langchain\_text\_splitters import RecursiveCharacterTextSplitter

from langgraph.graph import START, StateGraph

from typing\_extensions import List, TypedDict

# Load and chunk contents of the blog

loader = WebBaseLoader(

web\_paths=("https://lilianweng.github.io/posts/2023-06-23-agent/",),

bs\_kwargs=dict(

parse\_only=bs4.SoupStrainer(

class\_=("post-content", "post-title", "post-header")

)

),

)

docs = loader.load()

text\_splitter = RecursiveCharacterTextSplitter(chunk\_size=1000, chunk\_overlap=200)

all\_splits = text\_splitter.split\_documents(docs)

# Index chunks

\_ = vector\_store.add\_documents(documents=all\_splits)

# Define prompt for question-answering

prompt = hub.pull("rlm/rag-prompt")

# Define state for application

class State(TypedDict):

question: str

context: List[Document]

answer: str

# Define application steps

def retrieve(state: State):

retrieved\_docs = vector\_store.similarity\_search(state["question"])

return {"context": retrieved\_docs}

def generate(state: State):

docs\_content = "\n\n".join(doc.page\_content for doc in state["context"])

messages = prompt.invoke({"question": state["question"], "context": docs\_content})

response = llm.invoke(messages)

return {"answer": response.content}

# Compile application and test

graph\_builder = StateGraph(State).add\_sequence([retrieve, generate])

graph\_builder.add\_edge(START, "retrieve")

graph = graph\_builder.compile()

response = graph.invoke({"question": "What is Task Decomposition?"})

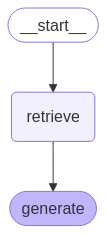
print(response["answer"])

# 3. LANG GRAPH

We'll use LangGraph to tie together the retrieval and generation steps into a single application. This will bring a number of benefits:

We can define our application logic once and automatically support multiple invocation modes, including streaming, async, and batched calls. We get streamlined deployments via LangGraph Platform. LangSmith will automatically trace the steps of our application together. We can easily add key features to our application, including persistence and human-in-the-loop approval, with minimal code changes.

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).

from langgraph.graph import START, StateGraph

graph\_builder = StateGraph(State).add\_sequence([retrieve, generate])

graph\_builder.add\_edge(START, "retrieve")

graph = graph\_builder.compile()

from IPython.display import Image, display

display(Image(graph.get\_graph().draw\_mermaid\_png()))

result = graph.invoke({"question": "What is Task Decomposition?"})

print(f'Context: {result["context"]}\n\n')

print(f'Answer: {result["answer"]}')

# 4. LANG-GRAPH

Install the packages

!pip install langsmith

!pip install langchain-community

!pip install -qU langchain-huggingface

!pip install langgraph

Get the langsmith API

import getpass

import os

LANGSMITH\_API\_KEY="lsv2\_pt\_bde7f95f197c4252984dc8b05a0d9726\_832920537a"

os.environ["LANGSMITH\_TRACING"] = "true"

os.environ["LANGSMITH\_API\_KEY"] = LANGSMITH\_API\_KEY

In the same code add traceble decorator

import bs4

from langchain import hub

from langchain\_community.document\_loaders import WebBaseLoader

from langchain\_core.documents import Document

from langchain\_text\_splitters import RecursiveCharacterTextSplitter

from langgraph.graph import START, StateGraph

from typing\_extensions import List, TypedDict

from langsmith import traceable

# Load and chunk contents of the blog

loader = WebBaseLoader(

web\_paths=("https://lilianweng.github.io/posts/2023-06-23-agent/",),

bs\_kwargs=dict(

parse\_only=bs4.SoupStrainer(

class\_=("post-content", "post-title", "post-header")

)

),

)

docs = loader.load()

text\_splitter = RecursiveCharacterTextSplitter(chunk\_size=1000, chunk\_overlap=200)

all\_splits = text\_splitter.split\_documents(docs)

# Index chunks

\_ = vector\_store.add\_documents(documents=all\_splits)

# Define prompt for question-answering

prompt = hub.pull("rlm/rag-prompt")

# Define state for application

class State(TypedDict):

question: str

context: List[Document]

answer: str

# Define application steps

**@traceable**

def retrieve(state: State):

retrieved\_docs = vector\_store.similarity\_search(state["question"])

return {"context": retrieved\_docs}

**@traceable**

def generate(state: State):

docs\_content = "\n\n".join(doc.page\_content for doc in state["context"])

messages = prompt.invoke({"question": state["question"], "context": docs\_content})

response = llm.invoke(messages)

return {"answer": response.content}

# Compile application and test

graph\_builder = StateGraph(State).add\_sequence([retrieve, generate])

graph\_builder.add\_edge(START, "retrieve")

graph = graph\_builder.compile()

**@traceable**

def run\_pipeline(question: str):

state = {"question": question, "context": [], "answer": ""}

final\_state = graph.invoke(state)

return final\_state["answer"]

# Test the execution with LangSmith tracing

answer = run\_pipeline("How do autonomous agents work?")

print(answer)

# 5. LLAMA-INDEX- **A SIMPLE AGENT**

!pip install llama-index-llms-ollama llama-index-embeddings-huggingface

!pip install llama-index groq

!pip install llama-index-llms-groq

!pip install llama-index

import asyncio

from llama\_index.core.agent.workflow import FunctionAgent

from llama\_index.llms.groq import Groq

llm = Groq(model="llama3-70b-8192", api\_key="gsk\_82VobCBNDDfoHV1mNqFoWGdyb3FYI3jbGSkzU5V7KHJ485atAxCy")

# Define a simple calculator tool

def multiply(a: float, b: float) -> float:

"""Useful for multiplying two numbers."""

return a \* b

# Create an agent workflow with our calculator tool

agent = FunctionAgent(

name="Agent",

description="Useful for multiplying two numbers",

tools=[multiply],

llm=llm,

system\_prompt="You are a helpful assistant that can multiply two numbers.",

)

response = await agent.run(user\_msg="What is 20+(2\*4)?")

print(response)

# 6. AGENT with multiple tools and Chat Memory

import nest\_asyncio

import asyncio

from llama\_index.core.agent.workflow import FunctionAgent

from llama\_index.llms.groq import Groq

from llama\_index.core.memory import ChatMemoryBuffer

import re # To extract numbers from responses

# Fix event loop issue in Google Colab

nest\_asyncio.apply()

# Initialize LLM

llm = Groq(model="llama3-70b-8192", api\_key=GROQ\_API\_KEY)

# Define mathematical operations with result tracking

last\_result = None # Stores the last numeric result

def multiply(a: float, b: float) -> float:

"""Multiply two numbers and store the result."""

global last\_result

last\_result = a \* b

return last\_result

def add(a: float, b: float) -> float:

"""Add two numbers and store the result."""

global last\_result

last\_result = a + b

return last\_result

def subtract(a: float, b: float) -> float:

"""Subtract two numbers and store the result."""

global last\_result

last\_result = a - b

return last\_result

def divide(a: float, b: float) -> float:

"""Divide two numbers, handling division by zero, and store the result."""

global last\_result

if b == 0:

return "Cannot divide by zero"

last\_result = a / b

return last\_result

# Initialize memory with a larger token limit

memory = ChatMemoryBuffer.from\_defaults(token\_limit=2048)

# Create an enhanced agent workflow with memory

agent = FunctionAgent(

name="MathAgent",

description="A math assistant that performs calculations and remembers past results.",

tools=[multiply, add, subtract, divide], # Adding more tools

llm=llm,

memory=memory, # Enabling memory

system\_prompt=(

"You are an intelligent math assistant. You remember past calculations and use them in follow-up questions. "

"If the user refers to 'that' or 'previous answer', use the last computed numerical result."

),

)

# Function to extract the last numerical response from memory

def extract\_last\_number(response):

numbers = re.findall(r"[-+]?\d\*\.\d+|\d+", str(response)) # Extract numbers

return float(numbers[-1]) if numbers else None # Return last detected number

# Run async commands directly in a cell

async def test\_agent():

global last\_result

response1 = await agent.run(user\_msg="What is 10 + 5?")

last\_result = extract\_last\_number(response1) # Store last number explicitly

print("Response 1:", response1)

response2 = await agent.run(user\_msg=f"Multiply {last\_result} by 3") # Explicit reference

last\_result = extract\_last\_number(response2)

print("Response 2:", response2)

response3 = await agent.run(user\_msg=f"Now subtract 4 from {last\_result}") # Clear phrasing

last\_result = extract\_last\_number(response3)

print("Response 3:", response3)

# Run the async function

await test\_agent()