**AI MakerSpace GitHub Week 2**

[A white and blue logo

AI-generated content may be incorrect.](https://private-user-images.githubusercontent.com/37101144/261444999-d1343317-fa2f-41e1-8af1-1dbb18399719.png?jwt=eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9..BY7y9VVe7BKPDdfztAQEnIdGUe7QtT0xdO2-rD1tBgY)

**Session 3: End-to-End RAG**

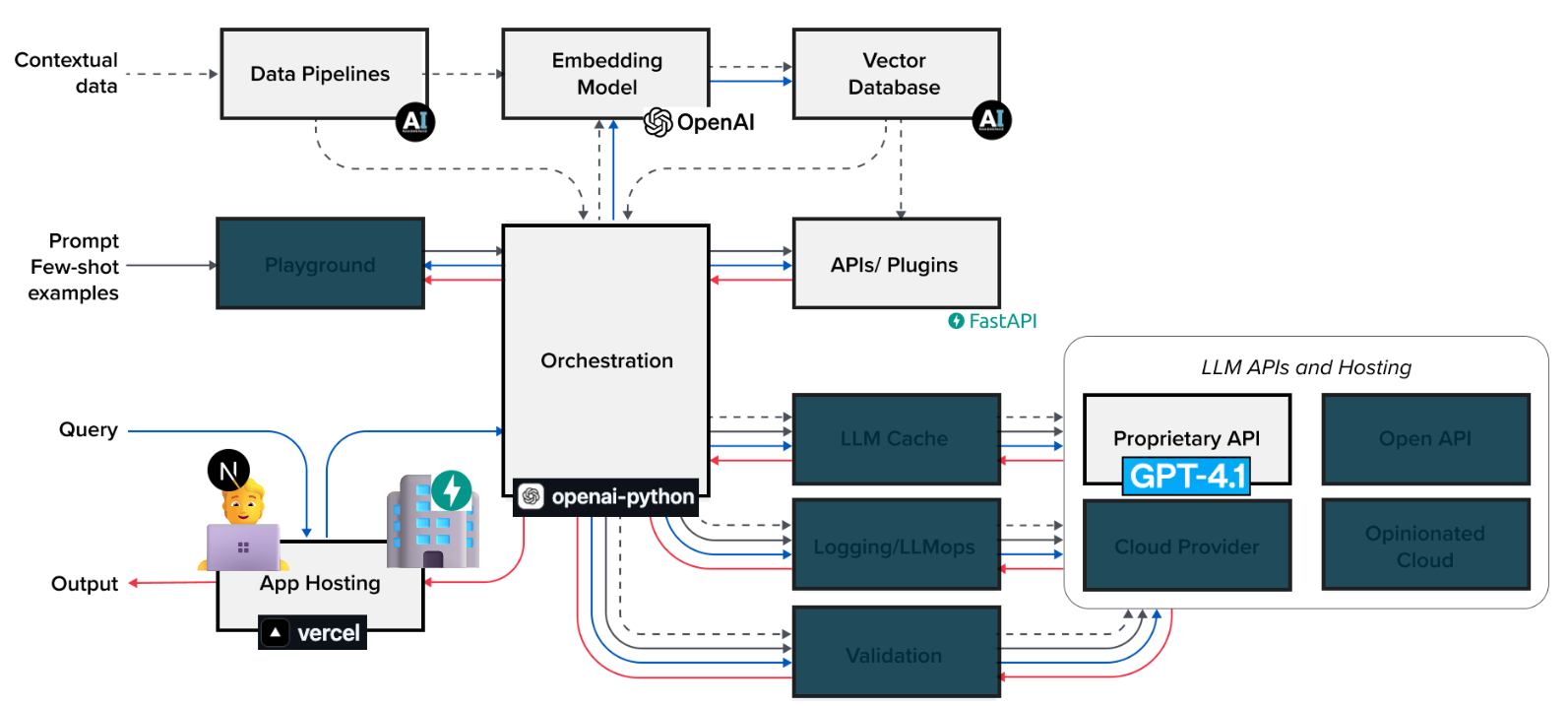
<https://github.com/AI-Maker-Space/AIE8/tree/main/03_End-to-End_RAG>

[**Quicklinks**](https://github.com/AI-Maker-Space/AIE7/tree/main/00_AIM_Quicklinks)

| **📰 Session Sheet** | **⏺️ Recording** | **🖼️ Slides** | **👨‍💻 Repo** | **📝 Homework** | **📁 Feedback** |
| --- | --- | --- | --- | --- | --- |
| [Session 3: E2E & Use Cases](https://www.notion.so/Session-3-End-to-End-AI-Applications-OSS-Models-I-and-2025-Industry-Use-Cases-26acd547af3d80b4b646e2fd6f1fd31c) | [Recording!](https://us02web.zoom.us/rec/share/7UJErmXFPnBQmIxWoeHVtCVcjtF1c_XmzAybJLGgei5Xrju_Q2jgPzgjYI8YT06o.pRQgg0m-t4-HHAmV) (\*6zDd0%S) | [Session 3 Slides](https://www.canva.com/design/DAGzJw-3i34/1UdGr5HlXlPjFtabOAWdkw/edit?utm_content=DAGzJw-3i34&utm_campaign=designshare&utm_medium=link2&utm_source=sharebutton) | You are here! | [Session 3 Assignment: E2E](https://forms.gle/ZVvwkbg4jEpHKpCY9) | [AIE8 Feedback 9/16](https://forms.gle/9SnYW7vgNLeGpkh47) |

Building off last week, we're going to take our Pythonic RAG application to the next level!

This will be our updated system diagram for our application after the changes you make below are completed!

[](https://camo.githubusercontent.com/90084575a0300e8ee5767d20b08f7e1fab8e6077679d31a8cbd6c840519bd1f1/68747470733a2f2f692e696d6775722e636f6d2f46734e534739542e706e67)

**🏗️ Activity #1:**

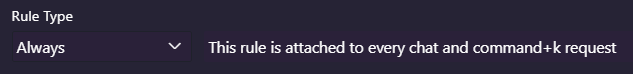
This week, we'll be working out of your challenge repository!

1. You can begin by copying and pasting the new aimakerspace library into your challenge repository at the ROOT LEVEL (top level directory).

NOTE: You can use the following command to do the same, just be sure to replace the paths with the correct ones based on your local environment.

cp /PATH/TO/THIS/FOLDER/aimakerspace /PATH/TO/YOUR/CHALLENGE

1. Create a new rule in your .cursor folder with the following text, make this a global rule that applies at all times:

[](https://camo.githubusercontent.com/efd2292174bbbc214be9494905f36c51bc0c4991eca34f69e48dfc8f7b7d2a9a/68747470733a2f2f692e696d6775722e636f6d2f755765796f48432e706e67)

You always prefer to use branch development. Before writing any code - you create a feature branch to hold those changes.

After you are done - provide instructions in a "MERGE.md" file that explains how to merge the changes back to main with both a GitHub PR route and a GitHub CLI route.

1. Use Cursor (or your own mind, of course) to include the RAG functionality we discussed last week to allow users to upload PDFs and "chat with them" (interact with a RAG pipeline).

NOTE: You can do this with the following prompt as a way to get started:

Modify the application (frontend and backend) to:

- Allow the user to upload a PDF

- Your PDF should be used to build your RAG's context

- The LLM should only answer questions using information from the provided context

- Index the PDF using the `aimakerspace` library

- Chat with the PDF using a simple RAG system built with the `aimakerspace` library

NOTE: You need to do this from your challenge repo.

1. Deploy the application with Vercel.

**🏗️ Activity #2:**

Determine a specific use-case for RAG, and adapt your challenge application to that new use-case.

Think about how people from that domain or expertise area may interact with your application - and ensure it's suited to them.

This may involve:

* Modifying the UI
* Adding additional file-types to be ingested by RAG

Once done, deploy your customized application to Vercel!

**🚧 Advanced Build:**

🚧 Advanced Build 🚧 (OPTIONAL - *open this section for the requirements*)

Leverage Together API's endpoints to power the LLM in your deployed application - this will require a small change to the aimakerspace/openai\_utils/chatmodel.py file.

NOTE: Also describe the process of modifying the endpoints used - and what made it difficult or not to make this change.

**Submitting Your Homework**

**Main Assignment**

Follow these steps to prepare and submit your homework:

1. Verify that Activity #1 was completed by validating that:
   * Cursor/Claude followed the global rules you added to the .cursor file:
     + It created a new branch prior to generating code
     + It created a MERGE.md file with appropriate instructions for merging this new branch

NOTE: If you "used your own mind" rather than vibe coding with Cursor/Claude then it is your responsibility to do these two things, or ask Claude directly to do them for you.

* + The deployed application is able to upload a PDF, index it, and then chat about the contents of the PDF.

1. Verify that Activity #2 was completed by validating that:
   * Cursor/Claude followed the global rules you added to the .cursor file (same as in Step 1 above)
   * The deployed application is still able to upload, process, and chat about a PDF (the functionality of Activity #1)
   * The deployed application meets your new functionality requirements for the RAG-specific use-case that you implemented
2. Create a ***5 minute or less*** Loom video about the assignment and your modified challenge application
3. Post on social media:
   * LinkedIn or X ⬅️ 2 extra credit points! *OR*
   * [Discord's #build-ship-share-🏗️-🚢-🚀 channel](https://discord.com/channels/1135695983720792216/1415130012394192896) ⬅️ 1 extra credit point!
4. Complete the Homework Form!

**Advanced Build (Alternative to the Main Assignment)**

Follow these steps to prepare and submit your homework:

1. Deploy your updated application to Vercel
2. Create a ***5 minute or less*** Loom video about modifying the endpoints, including a discussion of what made this change difficult and/or easy to implement.
3. Post on social media:
   * LinkedIn or X ⬅️ 2 extra credit points! *OR*
   * [Discord's #build-ship-share-🏗️-🚢-🚀 channel](https://discord.com/channels/1135695983720792216/1415130012394192896) ⬅️ 1 extra credit point!
4. Complete the Homework Form!

<https://github.com/AI-Maker-Space/AIE8/blob/main/03_End-to-End_RAG/aimakerspace/text_utils.py>

from pathlib import Path

from typing import Iterable, List

import PyPDF2

class TextFileLoader:

"""Load plain-text documents from a single file or an entire directory."""

def \_\_init\_\_(self, path: str, encoding: str = "utf-8"):

self.path = Path(path)

self.encoding = encoding

self.documents: List[str] = []

def load(self) -> None:

"""Populate ``self.documents`` from the configured path."""

self.documents = list(self.\_iter\_documents())

def load\_file(self) -> None:

"""Load a single file specified by ``self.path``."""

self.documents = [self.\_read\_text\_file(self.path)]

def load\_directory(self) -> None:

"""Load all text files contained within ``self.path``."""

self.documents = list(self.\_iter\_directory(self.path))

def load\_documents(self) -> List[str]:

"""Convenience wrapper returning the loaded documents."""

self.load()

return self.documents

def \_iter\_documents(self) -> Iterable[str]:

if self.path.is\_dir():

yield from self.\_iter\_directory(self.path)

elif self.path.is\_file() and self.path.suffix.lower() == ".txt":

yield self.\_read\_text\_file(self.path)

else:

raise ValueError(

"Provided path must be a directory or a .txt file: " f"{self.path}"

)

def \_iter\_directory(self, directory: Path) -> Iterable[str]:

for entry in sorted(directory.rglob("\*.txt")):

if entry.is\_file():

yield self.\_read\_text\_file(entry)

def \_read\_text\_file(self, file\_path: Path) -> str:

with file\_path.open("r", encoding=self.encoding) as file\_handle:

return file\_handle.read()

class CharacterTextSplitter:

"""Naively split long strings into overlapping character chunks."""

def \_\_init\_\_(

self,

chunk\_size: int = 1000,

chunk\_overlap: int = 200,

):

if chunk\_size <= chunk\_overlap:

raise ValueError("Chunk size must be greater than chunk overlap")

self.chunk\_size = chunk\_size

self.chunk\_overlap = chunk\_overlap

def split(self, text: str) -> List[str]:

"""Split ``text`` into chunks preserving the configured overlap."""

step = self.chunk\_size - self.chunk\_overlap

return [text[i : i + self.chunk\_size] for i in range(0, len(text), step)]

def split\_texts(self, texts: List[str]) -> List[str]:

"""Split multiple texts and flatten the resulting chunks."""

chunks: List[str] = []

for text in texts:

chunks.extend(self.split(text))

return chunks

class PDFLoader:

"""Extract text from PDF files stored at a path."""

def \_\_init\_\_(self, path: str):

self.path = Path(path)

self.documents: List[str] = []

def load(self) -> None:

"""Populate ``self.documents`` from the configured path."""

self.documents = list(self.\_iter\_documents())

def load\_file(self) -> None:

"""Load a single PDF specified by ``self.path``."""

self.documents = [self.\_read\_pdf(self.path)]

def load\_directory(self) -> None:

"""Load all PDF files contained within ``self.path``."""

self.documents = list(self.\_iter\_directory(self.path))

def load\_documents(self) -> List[str]:

"""Convenience wrapper returning the loaded documents."""

self.load()

return self.documents

def \_iter\_documents(self) -> Iterable[str]:

if self.path.is\_dir():

yield from self.\_iter\_directory(self.path)

elif self.path.is\_file() and self.path.suffix.lower() == ".pdf":

yield self.\_read\_pdf(self.path)

else:

raise ValueError(

"Provided path must be a directory or a .pdf file: " f"{self.path}"

)

def \_iter\_directory(self, directory: Path) -> Iterable[str]:

for entry in sorted(directory.rglob("\*.pdf")):

if entry.is\_file():

yield self.\_read\_pdf(entry)

def \_read\_pdf(self, file\_path: Path) -> str:

with file\_path.open("rb") as file\_handle:

pdf\_reader = PyPDF2.PdfReader(file\_handle)

extracted\_pages = [page.extract\_text() or "" for page in pdf\_reader.pages]

return "\n".join(extracted\_pages)

if \_\_name\_\_ == "\_\_main\_\_":

loader = TextFileLoader("data/KingLear.txt")

loader.load()

splitter = CharacterTextSplitter()

chunks = splitter.split\_texts(loader.documents)

print(len(chunks))

print(chunks[0])

print("--------")

print(chunks[1])

print("--------")

print(chunks[-2])

print("--------")

print(chunks[-1])

<https://github.com/AI-Maker-Space/AIE8/blob/main/03_End-to-End_RAG/aimakerspace/vectordatabase.py>   
  
import asyncio

from typing import Callable, Dict, Iterable, List, Optional, Tuple, Union

import numpy as np

from aimakerspace.openai\_utils.embedding import EmbeddingModel

def cosine\_similarity(vector\_a: np.ndarray, vector\_b: np.ndarray) -> float:

"""Return the cosine similarity between two vectors."""

norm\_a = np.linalg.norm(vector\_a)

norm\_b = np.linalg.norm(vector\_b)

if norm\_a == 0 or norm\_b == 0:

return 0.0

dot\_product = np.dot(vector\_a, vector\_b)

return float(dot\_product / (norm\_a \* norm\_b))

class VectorDatabase:

"""Minimal in-memory vector store backed by numpy arrays."""

def \_\_init\_\_(self, embedding\_model: Optional[EmbeddingModel] = None):

self.vectors: Dict[str, np.ndarray] = {}

self.embedding\_model = embedding\_model or EmbeddingModel()

def insert(self, key: str, vector: Iterable[float]) -> None:

"""Store ``vector`` so that it can be retrieved with ``key`` later on."""

self.vectors[key] = np.asarray(vector, dtype=float)

def search(

self,

query\_vector: Iterable[float],

k: int,

distance\_measure: Callable[[np.ndarray, np.ndarray], float] = cosine\_similarity,

) -> List[Tuple[str, float]]:

"""Return the ``k`` vectors most similar to ``query\_vector``."""

if k <= 0:

raise ValueError("k must be a positive integer")

query = np.asarray(query\_vector, dtype=float)

scores = [

(key, distance\_measure(query, vector))

for key, vector in self.vectors.items()

]

scores.sort(key=lambda item: item[1], reverse=True)

return scores[:k]

def search\_by\_text(

self,

query\_text: str,

k: int,

distance\_measure: Callable[[np.ndarray, np.ndarray], float] = cosine\_similarity,

return\_as\_text: bool = False,

) -> Union[List[Tuple[str, float]], List[str]]:

"""Vector search using an embedding generated from ``query\_text``."""

query\_vector = self.embedding\_model.get\_embedding(query\_text)

results = self.search(query\_vector, k, distance\_measure)

if return\_as\_text:

return [result[0] for result in results]

return results

def retrieve\_from\_key(self, key: str) -> Optional[np.ndarray]:

"""Return the stored vector for ``key`` if present."""

return self.vectors.get(key)

async def abuild\_from\_list(self, list\_of\_text: List[str]) -> "VectorDatabase":

"""Populate the vector store asynchronously from raw text snippets."""

embeddings = await self.embedding\_model.async\_get\_embeddings(list\_of\_text)

for text, embedding in zip(list\_of\_text, embeddings):

self.insert(text, embedding)

return self

if \_\_name\_\_ == "\_\_main\_\_":

list\_of\_text = [

"I like to eat broccoli and bananas.",

"I ate a banana and spinach smoothie for breakfast.",

"Chinchillas and kittens are cute.",

"My sister adopted a kitten yesterday.",

"Look at this cute hamster munching on a piece of broccoli.",

]

vector\_db = VectorDatabase()

vector\_db = asyncio.run(vector\_db.abuild\_from\_list(list\_of\_text))

k = 2

searched\_vector = vector\_db.search\_by\_text("I think fruit is awesome!", k=k)

print(f"Closest {k} vector(s):", searched\_vector)

retrieved\_vector = vector\_db.retrieve\_from\_key(

"I like to eat broccoli and bananas."

)

print("Retrieved vector:", retrieved\_vector)

relevant\_texts = vector\_db.search\_by\_text(

"I think fruit is awesome!", k=k, return\_as\_text=True

)

print(f"Closest {k} text(s):", relevant\_texts)  
  
  
<https://github.com/AI-Maker-Space/AIE8/blob/main/03_End-to-End_RAG/aimakerspace/openai_utils/prompts.py>   
  
import re

from typing import Any, Dict, List

class BasePrompt:

"""Simple string template helper used to format prompt text."""

def \_\_init\_\_(self, prompt: str):

self.prompt = prompt

self.\_pattern = re.compile(r"\{([^}]+)\}")

def format\_prompt(self, \*\*kwargs: Any) -> str:

"""Return the prompt with ``kwargs`` substituted for placeholders."""

matches = self.\_pattern.findall(self.prompt)

replacements = {match: kwargs.get(match, "") for match in matches}

return self.prompt.format(\*\*replacements)

def get\_input\_variables(self) -> List[str]:

"""Return the placeholder names used by this prompt."""

return self.\_pattern.findall(self.prompt)

class RolePrompt(BasePrompt):

"""Prompt template that also captures an accompanying chat role."""

def \_\_init\_\_(self, prompt: str, role: str):

super().\_\_init\_\_(prompt)

self.role = role

def create\_message(self, apply\_format: bool = True, \*\*kwargs: Any) -> Dict[str, str]:

"""Build an OpenAI chat message dictionary for this prompt."""

content = self.format\_prompt(\*\*kwargs) if apply\_format else self.prompt

return {"role": self.role, "content": content}

class SystemRolePrompt(RolePrompt):

def \_\_init\_\_(self, prompt: str):

super().\_\_init\_\_(prompt, "system")

class UserRolePrompt(RolePrompt):

def \_\_init\_\_(self, prompt: str):

super().\_\_init\_\_(prompt, "user")

class AssistantRolePrompt(RolePrompt):

def \_\_init\_\_(self, prompt: str):

super().\_\_init\_\_(prompt, "assistant")

if \_\_name\_\_ == "\_\_main\_\_":

prompt = BasePrompt("Hello {name}, you are {age} years old")

print(prompt.format\_prompt(name="John", age=30))

prompt = SystemRolePrompt("Hello {name}, you are {age} years old")

print(prompt.create\_message(name="John", age=30))

print(prompt.get\_input\_variables())

<https://github.com/AI-Maker-Space/AIE8/blob/main/03_End-to-End_RAG/aimakerspace/openai_utils/embedding.py>  
  
import asyncio

import os

from typing import Iterable, List

from dotenv import load\_dotenv

from openai import AsyncOpenAI, OpenAI

class EmbeddingModel:

"""Helper for generating embeddings via the OpenAI API."""

def \_\_init\_\_(self, embeddings\_model\_name: str = "text-embedding-3-small"):

load\_dotenv()

self.openai\_api\_key = os.getenv("OPENAI\_API\_KEY")

if self.openai\_api\_key is None:

raise ValueError(

"OPENAI\_API\_KEY environment variable is not set. "

"Please configure it with your OpenAI API key."

)

self.embeddings\_model\_name = embeddings\_model\_name

self.async\_client = AsyncOpenAI()

self.client = OpenAI()

async def async\_get\_embeddings(self, list\_of\_text: Iterable[str]) -> List[List[float]]:

"""Return embeddings for ``list\_of\_text`` using the async client."""

embedding\_response = await self.async\_client.embeddings.create(

input=list(list\_of\_text), model=self.embeddings\_model\_name

)

return [item.embedding for item in embedding\_response.data]

async def async\_get\_embedding(self, text: str) -> List[float]:

"""Return an embedding for a single text using the async client."""

embedding = await self.async\_client.embeddings.create(

input=text, model=self.embeddings\_model\_name

)

return embedding.data[0].embedding

def get\_embeddings(self, list\_of\_text: Iterable[str]) -> List[List[float]]:

"""Return embeddings for ``list\_of\_text`` using the sync client."""

embedding\_response = self.client.embeddings.create(

input=list(list\_of\_text), model=self.embeddings\_model\_name

)

return [item.embedding for item in embedding\_response.data]

def get\_embedding(self, text: str) -> List[float]:

"""Return an embedding for a single text using the sync client."""

embedding = self.client.embeddings.create(

input=text, model=self.embeddings\_model\_name

)

return embedding.data[0].embedding

if \_\_name\_\_ == "\_\_main\_\_":

embedding\_model = EmbeddingModel()

print(asyncio.run(embedding\_model.async\_get\_embedding("Hello, world!")))

print(

asyncio.run(

embedding\_model.async\_get\_embeddings(["Hello, world!", "Goodbye, world!"])

)

)  
  
<https://github.com/AI-Maker-Space/AIE8/blob/main/03_End-to-End_RAG/aimakerspace/openai_utils/chatmodel.py>  
  
import os

from typing import Any, AsyncIterator, Iterable, List, MutableMapping

from dotenv import load\_dotenv

from openai import AsyncOpenAI, OpenAI

load\_dotenv()

ChatMessage = MutableMapping[str, Any]

class ChatOpenAI:

"""Thin wrapper around the OpenAI chat completion APIs."""

def \_\_init\_\_(self, model\_name: str = "gpt-4o-mini"):

self.model\_name = model\_name

self.openai\_api\_key = os.getenv("OPENAI\_API\_KEY")

if self.openai\_api\_key is None:

raise ValueError("OPENAI\_API\_KEY is not set")

self.\_client = OpenAI()

self.\_async\_client = AsyncOpenAI()

def run(

self,

messages: Iterable[ChatMessage],

text\_only: bool = True,

\*\*kwargs: Any,

) -> Any:

"""Execute a chat completion request.

``messages`` must be an iterable of ``{"role": ..., "content": ...}``

dictionaries. When ``text\_only`` is ``True`` (the default) only the

completion text is returned; otherwise the full response object is

provided.

"""

message\_list = self.\_coerce\_messages(messages)

response = self.\_client.chat.completions.create(

model=self.model\_name, messages=message\_list, \*\*kwargs

)

if text\_only:

return response.choices[0].message.content

return response

async def astream(

self, messages: Iterable[ChatMessage], \*\*kwargs: Any

) -> AsyncIterator[str]:

"""Yield streaming completion chunks as they arrive from the API."""

message\_list = self.\_coerce\_messages(messages)

stream = await self.\_async\_client.chat.completions.create(

model=self.model\_name, messages=message\_list, stream=True, \*\*kwargs

)

async for chunk in stream:

content = chunk.choices[0].delta.content

if content is not None:

yield content

def \_coerce\_messages(self, messages: Iterable[ChatMessage]) -> List[ChatMessage]:

if isinstance(messages, list):

return messages

return list(messages)

[A white and blue logo

AI-generated content may be incorrect.](https://private-user-images.githubusercontent.com/37101144/261444999-d1343317-fa2f-41e1-8af1-1dbb18399719.png?jwt=eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9..RtRT0C609I0Zh4V3itRsAAMLZjNuNdn8CB3FiRxG6cs)

Session 04

**Production RAG with LangGraph and LangChain**<https://github.com/AI-Maker-Space/AIE8/blob/main/04_Production_RAG/README.md>

| **📰 Session Sheet** | **⏺️ Recording** | **🖼️ Slides** | **👨‍💻 Repo** | **📝 Homework** | **📁 Feedback** |
| --- | --- | --- | --- | --- | --- |
| [Session 4: RAG with LangGraph, OSS Local Models, & Eval w/ LangSmith](https://www.notion.so/Session-4-Production-Grade-RAG-with-LangChain-and-LangSmith-26acd547af3d80838d5beba464d7e701) | [Recording!](https://us02web.zoom.us/rec/share/jEs9TS_re1f9X3y2T61Dgv_bEp6EmVzVkiYDOC-cEU8WA2tR5jMI1bwsn4L_Al1n.msDqlCRCROFBaRCH) (78y?PRTg) | [Session 4 Slides](https://www.canva.com/design/DAGzMO1y0FQ/oJaw4HMIFecP3oX9jSO4fw/edit?utm_content=DAGzMO1y0FQ&utm_campaign=designshare&utm_medium=link2&utm_source=sharebutton) | You are here! | [Session 4 Assignment: Prod. RAG](https://forms.gle/i2SdxgWX4ahFwNrCA) | [AIE8 Feedback 9/18](https://forms.gle/ymYqK5MBLAG11jDB9) |

**Build 🏗️**

If running locally:

1. uv sync
2. **NEW**: Set up Ollama by running the Ollama\_Setup\_and\_Testing.ipynb notebook first to verify your local LLM setup
3. Open the main assignment notebook
4. Select the venv created by uv sync as your kernel

**Setting up Ollama (Local LLM)**

Before starting the main assignment, run the Ollama\_Setup\_and\_Testing.ipynb notebook to:

* Verify Ollama is installed and running
* Test embeddings with LangChain connectors
* Test model inference with LangChain connectors
* Ensure all models are properly downloaded

Run the preparation notebook and complete the contained tasks:

* 🤝 Breakout Room #1:
  1. Install and run Ollama
  2. Ensure all the required models are pulled
  3. Test them!

Next, run the Assignment notebook and complete the contained tasks:

* 🤝 Breakout Room #2:
  1. LangChain and LCEL Concepts
  2. Understanding States and Nodes
  3. Introduction to QDrant Vector Databases
  4. Building a Basic Graph

**Ship 🚢**

* The completed notebook.
* 5min. Loom Video

**Share 🚀**

* Walk through your notebook and explain what you've completed in the Loom video
* Make a social media post about your final application and tag @AIMakerspace
* Share 3 lessons learned
* Share 3 lessons not learned

**Submitting Your Homework**

Follow these steps to prepare and submit your homework:

1. Create a branch of your AIE7 repo to track your changes. Example command: git checkout -b s04-assignment
2. Responding to the activities and questions inline in the Assignment\_Introduction\_to\_LCEL\_and\_LangGraph\_LangChain\_Powered\_RAG.ipynb

*NOTE on the Assignment\_Introduction\_to\_LCEL\_and\_LangGraph\_LangChain\_Powered\_RAG notebook: You will also need to enter your response to Question #1 in the code cell directly below it which contains this line of code:*  embedding\_dim = # YOUR ANSWER

Introduction to LCEL and LangGraph: LangChain Powered RAG

<https://github.com/AI-Maker-Space/AIE8/blob/main/04_Production_RAG/Assignment_Introduction_to_LCEL_and_LangGraph_LangChain_Powered_RAG.ipynb>

In the following notebook we're going to focus on learning how to navigate and build useful applications using LangChain, specifically LCEL, and how to integrate different APIs together into a coherent RAG application!

We'll be building a RAG system to answer questions about how people use AI, using the "How People Use AI" dataset.

In the notebook, you'll complete the following Tasks:

* 🤝 Breakout Room #2:
  1. LangChain and LCEL Concepts
  2. Understanding States and Nodes
  3. Introduction to QDrant Vector Databases
  4. Building a Basic Graph

Let's get started!

Installation Requirements

Also, make sure Ollama is installed and running with the required models pulled (see instructions below).

Optional: LangSmith Setup for Tracing and Monitoring

LangSmith provides powerful tracing, monitoring, and debugging capabilities for LangChain applications. While not required for this notebook, setting it up will give you valuable insights into your RAG system's performance.

Getting LangSmith Credentials

1. **Sign up for LangSmith**: Visit [smith.langchain.com](https://smith.langchain.com/) and create a free account
2. **Get your API Key**:
   * Go to Settings → API Keys
   * Create a new API key and copy it
3. **Set your environment variables** (choose one method below):

**Option A: Set environment variables in your terminal before starting Jupyter:**

export LANGCHAIN\_TRACING\_V2**=**true

export LANGCHAIN\_API\_KEY**=**"your-api-key-here"

export LANGCHAIN\_PROJECT**=**"RAG-Assignment"

**Option B: Set them in the notebook (run the cell below):**

*# Optional: Set up LangSmith tracing*

*# Uncomment and fill in your credentials if you want to use LangSmith*

**import** os

**import** getpass

*# Uncomment the lines below to enable LangSmith tracing*

*# os.environ["LANGCHAIN\_TRACING\_V2"] = "true"*

*# os.environ["LANGCHAIN\_API\_KEY"] = getpass.getpass("Enter your LangSmith API key: ")*

*# os.environ["LANGCHAIN\_PROJECT"] = "RAG-Assignment"*

*# Verify setup (uncomment to check)*

*# print("LangSmith tracing enabled:", os.getenv("LANGCHAIN\_TRACING\_V2", "false"))*

*# print("Project name:", os.getenv("LANGCHAIN\_PROJECT", "Not set"))*

LangSmith tracing enabled: true

Project name: RAG-Assignment

What LangSmith Provides

Once set up, LangSmith will automatically trace your LangChain operations and provide:

* **Execution traces**: See exactly how your RAG pipeline processes each query
* **Performance metrics**: Monitor latency, token usage, and costs
* **Debugging tools**: Inspect intermediate outputs at each step
* **Error tracking**: Identify and debug issues in your chains
* **Dataset management**: Collect and organize your queries and responses

You can view all traces and analytics in your LangSmith dashboard at [smith.langchain.com](https://smith.langchain.com/).

**Note**: LangSmith is completely optional for this assignment. The notebook will work perfectly fine without it, but it's a valuable tool for production applications.

🤝 Breakout Room #2

Set Up Ollama

We'll be using Ollama to run local LLM models. Make sure you have Ollama installed and running:

1. Install Ollama from [https://ollama.ai](https://ollama.ai/) (curl https://ollama.ai/install.sh | sh)
2. Make sure the output of ollama -v reads 0.11.10 or greater.
3. Pull the models we'll use:
4. ollama pull gpt-oss:20b *# For the chat model*
5. ollama pull embeddinggemma:latest *# For embeddings*
6. Ensure Ollama is running (it should start automatically after installation)

A Note On Runnables

Understanding LangChain Runnables and LCEL

In LangChain, a Runnable is like a LEGO brick in your AI application - it's a standardized component that can be easily connected with other components. The real power of Runnables comes from their ability to be combined in flexible ways using LCEL (LangChain Expression Language).

Key Features of Runnables

1. Universal Interface

Every Runnable in LangChain follows the same pattern:

* Takes an input
* Performs some operation
* Returns an output

This consistency means you can treat different components (like models, retrievers, or parsers) in the same way.

2. Built-in Parallelization

Runnables come with methods for handling multiple inputs efficiently:

*# Process inputs in parallel, maintain order*

results **=** chain**.**batch([input1, input2, input3])

*# Process inputs as they complete*

**for** result **in** chain**.**batch\_as\_completed([input1, input2, input3]):

print(result)

3. Streaming Support

Perfect for responsive applications:

*# Stream outputs as they're generated*

**for** chunk **in** chain**.**stream({"query": "Tell me a story"}):

print(chunk, end**=**"", flush**=True**)

4. Easy Composition

The | operator makes building pipelines intuitive:

*# Create a basic RAG chain*

rag\_chain **=** retriever **|** prompt **|** model **|** output\_parser

Common Types of Runnables

* **Language Models**: Like our ChatOllama instance (running locally with Ollama)
* **Prompt Templates**: Format inputs consistently
* **Retrievers**: Get relevant context from a vector store
* **Output Parsers**: Structure model outputs
* **LangGraph Nodes**: Individual components in our graph

Think of Runnables as the building blocks of your LLM application. Just like how you can combine LEGO bricks in countless ways, you can mix and match Runnables to create increasingly sophisticated applications!

LangGraph Based RAG

Now that we have a reasonable grasp of LCEL and the idea of Runnables - let's see how we can use LangGraph to build the same system!

Primer: What is LangGraph?

LangGraph is a tool that leverages LangChain Expression Language to build coordinated multi-actor and stateful applications that includes cyclic behaviour.

Why Cycles?

In essence, we can think of a cycle in our graph as a more robust and customizable loop. It allows us to keep our application agent-forward while still giving the powerful functionality of traditional loops.

Due to the inclusion of cycles over loops, we can also compose rather complex flows through our graph in a much more readable and natural fashion. Effectively allowing us to recreate application flowcharts in code in an almost 1-to-1 fashion.

Why LangGraph?

Beyond the agent-forward approach - we can easily compose and combine traditional "DAG" (directed acyclic graph) chains with powerful cyclic behaviour due to the tight integration with LCEL. This means it's a natural extension to LangChain's core offerings!

NOTE: We're going to focus on building a simple DAG for today's assignment as an introduction to LangGraph

Putting the State in Stateful

Earlier we used this phrasing:

coordinated multi-actor and stateful applications

So what does that "stateful" mean?

To put it simply - we want to have some kind of object which we can pass around our application that holds information about what the current situation (state) is. Since our system will be constructed of many parts moving in a coordinated fashion - we want to be able to ensure we have some commonly understood idea of that state.

LangGraph leverages a StatefulGraph which uses an AgentState object to pass information between the various nodes of the graph.

There are more options than what we'll see below - but this AgentState object is one that is stored in a TypedDict with the key messages and the value is a Sequence of BaseMessages that will be appended to whenever the state changes.

However, in our example here, we're focusing on a simpler State object:

**class** State(TypedDict):

question: str

context: list[Document]

response: str

Let's think about a simple example to help understand exactly what this means (we'll simplify a great deal to try and clearly communicate what state is doing):

1. **We initialize our state object**:
2. {
3. "question": "",
4. "context": [],
5. "response": ""
6. }
7. **Our user submits a query to our application.**  
   We store the user's question in state["question"]. Now we have:
8. {
9. "question": "How tall is the Eiffel Tower?",
10. "context": [],
11. "response": ""
12. }
13. **We pass our state object to an Agent node** which is able to read the current state. It will use the value of state["question"] as input and might retrieve some context documents related to the question. It then generates a response which it stores in state["response"]. For example:
14. {
15. "question": "How tall is the Eiffel Tower?",
16. "context": [Document(page\_content**=**"...some data...")],
17. "response": "The Eiffel Tower is about 324 meters tall..."
18. }

That's it! The important part is that we have a consistent object (State) that's passed around, holding the crucial information as we go from one node to the next. This ensures our application has a single source of truth about what has happened so far and what is happening now.

**from** langgraph.graph **import** START, StateGraph

**from** typing\_extensions **import** TypedDict

**from** langchain\_core.documents **import** Document

**class** State(TypedDict):

question: str

context: list[Document]

response: str

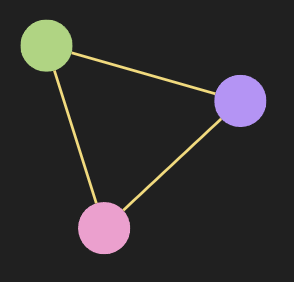
Now that we have state, and we have tools, and we have an LLM - we can finally start making our graph!

Let's take a second to refresh ourselves about what a graph is in this context.

Graphs, also called networks in some circles, are a collection of connected objects.

The objects in question are typically called nodes, or vertices, and the connections are called edges.

Let's look at a simple graph.



Here, we're using the coloured circles to represent the nodes and the yellow lines to represent the edges. In this case, we're looking at a fully connected graph - where each node is connected by an edge to each other node.

If we were to think about nodes in the context of LangGraph - we would think of a function, or an LCEL Runnable.

If we were to think about edges in the context of LangGraph - we might think of them as "paths to take" or "where to pass our state object next".

Building Nodes

We're going to need two nodes:

A node for retrieval, and a node for generation.

Let's start with our retrieve node!

Notice how we do not need to update the state object in the node, but can instead return a modification directly to our state.

Building a Retriever with LangChain

In order to build our retrieve node, we'll first need to build a retriever!

This will involve the following steps:

1. Ingesting Data
2. Chunking the Data
3. Vectorizing the Data and Storing it in a Vector Database
4. Converting it to a Retriever

Retreiver Step 1: Ingesting Data

In today's lesson, we're going to be building a RAG system to answer questions about how people use AI - and we will pull information into our index (vectorized chunks stored in our vector store) through LangChain's [PyMuPDFLoader](https://python.langchain.com/api_reference/community/document_loaders/langchain_community.document_loaders.pdf.PyMuPDFLoader.html)!

NOTE: We'll be using an async loader during our document ingesting - but our Jupyter Kernel is already running in an asyc loop! This means we'll want the ability to *nest* async loops.

**import** nest\_asyncio

nest\_asyncio**.**apply()

Now, we're good to load our documents through the [PyMuPDFLoader](https://python.langchain.com/api_reference/community/document_loaders/langchain_community.document_loaders.pdf.PyMuPDFLoader.html)!

**from** langchain\_community.document\_loaders **import** DirectoryLoader

**from** langchain\_community.document\_loaders **import** PyMuPDFLoader

directory\_loader **=** DirectoryLoader("data", glob**=**"\*\*/\*.pdf", loader\_cls**=**PyMuPDFLoader)

ai\_usage\_knowledge\_resources **=** directory\_loader**.**load()

ai\_usage\_knowledge\_resources[0]**.**page\_content[:1000]

'NBER WORKING PAPER SERIES\nHOW PEOPLE USE CHATGPT\nAaron Chatterji\nThomas Cunningham\nDavid J. Deming\nZoe Hitzig\nChristopher Ong\nCarl Yan Shan\nKevin Wadman\nWorking Paper 34255\nhttp://www.nber.org/papers/w34255\nNATIONAL BUREAU OF ECONOMIC RESEARCH\n1050 Massachusetts Avenue\nCambridge, MA 02138\nSeptember 2025\nWe acknowledge help and comments from Joshua Achiam, Hemanth Asirvatham, Ryan \nBeiermeister, Rachel Brown, Cassandra Duchan Solis, Jason Kwon, Elliott Mokski, Kevin Rao, \nHarrison Satcher, Gawesha Weeratunga, Hannah Wong, and Analytics & Insights team. We \nespecially thank Tyna Eloundou and Pamela Mishkin who in several ways laid the foundation for \nthis work. This study was approved by Harvard IRB (IRB25-0983). A repository containing all \ncode run to produce the analyses in this paper is available on request. The views expressed herein \nare those of the authors and do not necessarily reflect the views of the National Bureau of \nEconomic Research.\nAt least one co-author has disclosed a'

TextSplitting aka Chunking

We'll use the RecursiveCharacterTextSplitter to create our toy example.

It will split based on the following rules:

* Each chunk has a maximum size of 1000 tokens
* It will try and split first on the \n\n character, then on the \n, then on the <SPACE> character, and finally it will split on individual tokens.

Let's implement it and see the results!

**import** tiktoken

**from** langchain.text\_splitter **import** RecursiveCharacterTextSplitter

**def** tiktoken\_len(text):

*# Using cl100k\_base encoding which is a good general-purpose tokenizer*

*# This works well for estimating token counts even with Ollama models*

tokens **=** tiktoken**.**get\_encoding("cl100k\_base")**.**encode(

text,

)

**return** len(tokens)

text\_splitter **=** RecursiveCharacterTextSplitter(

chunk\_size **=** 750,

chunk\_overlap **=** 0,

length\_function **=** tiktoken\_len,

)

ai\_usage\_knowledge\_chunks **=** text\_splitter**.**split\_documents(ai\_usage\_knowledge\_resources)

🏗️ Activity #1:

While there's nothing specifically wrong with the chunking method used above - it is a naive approach that is not sensitive to specific data formats.

Brainstorm some ideas that would split large single documents into smaller documents.

1. YOUR IDEA HERE
2. YOUR IDEA HERE
3. YOUR IDEA HERE

Embeddings and Dense Vector Search

Now that we have our individual chunks, we need a system to correctly select the relevant pieces of information to answer our query.

This sounds like a perfect job for embeddings!

We'll be using Ollama's embeddinggemma model as our embedding model today! This is a powerful open-source embedding model that runs locally.

Let's load it up through LangChain.

**from** langchain\_ollama **import** OllamaEmbeddings

*# Using embeddinggemma which is a powerful open-source embedding model*

embedding\_model **=** OllamaEmbeddings(model**=**"embeddinggemma:latest")

❓ Question #1:

What is the embedding dimension, given that we're using embeddinggemma?

You will need to fill the next cell out correctly with your embedding dimension for the rest of the notebook to run.

embedding\_dim **=** *# YOUR ANSWER HERE*

Using A Vector Database - Intoduction to Qdrant

Up to this point, we've been using a dictionary to hold our embeddings - typically, we'll want to use a more robust strategy.

In this bootcamp - we'll be focusing on leveraging [Qdrant's vector database](https://qdrant.tech/qdrant-vector-database/).

Let's take a look at how we set-up Qdrant!

NOTE: We'll be spending a lot of time learning about Qdrant throughout the remainder of our time together - but for an initial primer, please check out [this resource](https://qdrant.tech/articles/what-is-a-vector-database/)

We are going to be using an "in-memory" Qdrant client, which means that our vectors will be held in our system's memory (RAM) - this is useful for prototyping and developement at smaller scales - but would need to be modified when moving to production. Luckily for us, this modification is trivial!

NOTE: While LangChain uses the terminology "VectorStore" (also known as a Vector Library), Qdrant is a "Vector Database" - more info. on that [here.](https://weaviate.io/blog/vector-library-vs-vector-database)

**from** langchain\_qdrant **import** QdrantVectorStore

**from** qdrant\_client **import** QdrantClient

**from** qdrant\_client.http.models **import** Distance, VectorParams

client **=** QdrantClient(":memory:")

Next, we need to create a collection - a collection is a specific...collection of vectors within the Qdrant client.

These are useful as they allow us to create multiple different "warehouses" in a single client, which can be leveraged for personalization and more!

Also notice that we define what our vector shapes are (embedding dim) as well as our desired distance metric.

client**.**create\_collection(

collection\_name**=**"ai\_usage\_knowledge\_index",

vectors\_config**=**VectorParams(size**=**embedding\_dim, distance**=**Distance**.**COSINE),

)

True

Now we can assemble our vector database! Notice that we provide our client, our created collection, and our embedding model!

vector\_store **=** QdrantVectorStore(

client**=**client,

collection\_name**=**"ai\_usage\_knowledge\_index",

embedding**=**embedding\_model,

)

Now that we have our vector database set-up, we can add our documents into it!

\_ **=** vector\_store**.**add\_documents(documents**=**ai\_usage\_knowledge\_chunks)

Creating a Retriever

Now that we have an idea of how we're getting our most relevant information - let's see how we could create a pipeline that would automatically extract the closest chunk to our query and use it as context for our prompt!

This will involve a popular LangChain interace known as as\_retriever!

NOTE: We can still specify how many documents we wish to retrieve per vector.

retriever **=** vector\_store**.**as\_retriever(search\_kwargs**=**{"k": 5})

retriever**.**invoke("How do people use AI in their daily work?")

[Document(metadata={'producer': 'macOS Version 15.4.1 (Build 24E263) Quartz PDFContext, AppendMode 1.1', 'creator': 'LaTeX with hyperref', 'creationdate': '2025-09-12T20:05:32+00:00', 'source': 'data/howpeopleuseai.pdf', 'file\_path': 'data/howpeopleuseai.pdf', 'total\_pages': 64, 'format': 'PDF 1.6', 'title': 'How People Use ChatGPT', 'author': '', 'subject': '', 'keywords': '', 'moddate': '2025-09-15T10:32:36-04:00', 'trapped': '', 'modDate': "D:20250915103236-04'00'", 'creationDate': 'D:20250912200532Z', 'page': 34, '\_id': 'e060d9fe56994b61841d2d220a033503', '\_collection\_name': 'ai\_usage\_knowledge\_index'}, page\_content='Panel A. Work Related\nPanel B1. Asking.\nPanel B2. Doing.\nFigure 23: (continued on next page)\n33'),

Document(metadata={'producer': 'macOS Version 15.4.1 (Build 24E263) Quartz PDFContext, AppendMode 1.1', 'creator': 'LaTeX with hyperref', 'creationdate': '2025-09-12T20:05:32+00:00', 'source': 'data/howpeopleuseai.pdf', 'file\_path': 'data/howpeopleuseai.pdf', 'total\_pages': 64, 'format': 'PDF 1.6', 'title': 'How People Use ChatGPT', 'author': '', 'subject': '', 'keywords': '', 'moddate': '2025-09-15T10:32:36-04:00', 'trapped': '', 'modDate': "D:20250915103236-04'00'", 'creationDate': 'D:20250912200532Z', 'page': 30, '\_id': '7c24917ac2bf47f19cf05dca5822b8c1', '\_collection\_name': 'ai\_usage\_knowledge\_index'}, page\_content='Panel A. Work Related\nPanel B1. Asking.\nPanel B2. Doing.\nPanel B3. Expressing.\nFigure 22: (continued on next page)\n29'),

Document(metadata={'producer': 'macOS Version 15.4.1 (Build 24E263) Quartz PDFContext, AppendMode 1.1', 'creator': 'LaTeX with hyperref', 'creationdate': '2025-09-12T20:05:32+00:00', 'source': 'data/howpeopleuseai.pdf', 'file\_path': 'data/howpeopleuseai.pdf', 'total\_pages': 64, 'format': 'PDF 1.6', 'title': 'How People Use ChatGPT', 'author': '', 'subject': '', 'keywords': '', 'moddate': '2025-09-15T10:32:36-04:00', 'trapped': '', 'modDate': "D:20250915103236-04'00'", 'creationDate': 'D:20250912200532Z', 'page': 53, '\_id': '94c3d1d00d584ad199f807424e761a38', '\_collection\_name': 'ai\_usage\_knowledge\_index'}, page\_content="E.g. 'User is rewriting email to neighbors about\nplumbing to be more friendly,'\nor 'User is complaining about grandmother'\nor 'User is asking for help fixing python databricks error.'\n31The IWA classifications were carried out by two annotators, while all other classifications had three.\n52"),

Document(metadata={'producer': 'macOS Version 15.4.1 (Build 24E263) Quartz PDFContext, AppendMode 1.1', 'creator': 'LaTeX with hyperref', 'creationdate': '2025-09-12T20:05:32+00:00', 'source': 'data/howpeopleuseai.pdf', 'file\_path': 'data/howpeopleuseai.pdf', 'total\_pages': 64, 'format': 'PDF 1.6', 'title': 'How People Use ChatGPT', 'author': '', 'subject': '', 'keywords': '', 'moddate': '2025-09-15T10:32:36-04:00', 'trapped': '', 'modDate': "D:20250915103236-04'00'", 'creationDate': 'D:20250912200532Z', 'page': 37, '\_id': '67282288707b4a8da8eec2c56ad98bdf', '\_collection\_name': 'ai\_usage\_knowledge\_index'}, page\_content='in knowledge-intensive jobs where productivity is increasing in the quality of decision-making.\n36'),

Document(metadata={'producer': 'macOS Version 15.4.1 (Build 24E263) Quartz PDFContext, AppendMode 1.1', 'creator': 'LaTeX with hyperref', 'creationdate': '2025-09-12T20:05:32+00:00', 'source': 'data/howpeopleuseai.pdf', 'file\_path': 'data/howpeopleuseai.pdf', 'total\_pages': 64, 'format': 'PDF 1.6', 'title': 'How People Use ChatGPT', 'author': '', 'subject': '', 'keywords': '', 'moddate': '2025-09-15T10:32:36-04:00', 'trapped': '', 'modDate': "D:20250915103236-04'00'", 'creationDate': 'D:20250912200532Z', 'page': 32, '\_id': '8a5b523262f4425f983f1140b2830f01', '\_collection\_name': 'ai\_usage\_knowledge\_index'}, page\_content='27As discussed in Section: Data and Privacy, our dataset only includes users on ChatGPT Consumer plans. Corporate\nusers may also use ChatGPT Business (formerly known as Teams) or ChatGPT Enterprise.\n28Very few work-related messages are classified as Expressing.\n29Appendix D contains a full report of GWA counts broken down by occupation, for both work-related ChatGPT\n31')]

Creating the Node

We're finally ready to create our node!

**def** retrieve(state: State) **->** State:

retrieved\_docs **=** retriever**.**invoke(state["question"])

**return** {"context" : retrieved\_docs}

Generate Node

Next, let's create our generate node - which will leverage LangChain and something called an "LCEL Chain" which you can read more about [here](https://python.langchain.com/docs/concepts/lcel/)!

We'll want to create a chain that does the following:

1. Formats our inputs into a chat template suitable for RAG
2. Takes that chat template and sends it to an LLM
3. Parses that output into str format

Let's get chaining!

Chain Components: RAG Chat Template

We'll create a chat template that takes in some query and formats it as a RAG prompt using LangChain's prompt template!

**from** langchain\_core.prompts **import** ChatPromptTemplate

HUMAN\_TEMPLATE **=** """

#CONTEXT:

{context}

QUERY:

{query}

Use the provide context to answer the provided user query. Only use the provided context to answer the query. If you do not know the answer, or it's not contained in the provided context response with "I don't know"

"""

chat\_prompt **=** ChatPromptTemplate**.**from\_messages([

("human", HUMAN\_TEMPLATE)

])

chat\_prompt**.**invoke({"context" : "OUR CONTEXT HERE", "query" : "OUR QUERY HERE"})**.**messages[0]**.**content

'\n#CONTEXT:\nOUR CONTEXT HERE\n\nQUERY:\nOUR QUERY HERE\n\nUse the provide context to answer the provided user query. Only use the provided context to answer the query. If you do not know the answer, or it\'s not contained in the provided context response with "I don\'t know"\n'

Chain Components: Generator

We'll next set-up the generator - which will be Ollama's gpt-oss:20b model running locally!

**from** langchain\_ollama **import** ChatOllama

*# Using gpt-oss:20b which is a powerful and efficient local model*

ollama\_chat\_model **=** ChatOllama(model**=**"gpt-oss:20b", temperature**=**0.6)

Let's now call our model with a formatted prompt.

Notice that we have some nested calls here - we'll see that this is made easier by LCEL.

ollama\_chat\_model**.**invoke(chat\_prompt**.**invoke({"context" : "Paris is the capital of France", "query" : "What is the capital of France?"}))

AIMessage(content='Paris', additional\_kwargs={}, response\_metadata={'model': 'gpt-oss:20b', 'created\_at': '2025-09-18T16:59:06.037892Z', 'done': True, 'done\_reason': 'stop', 'total\_duration': 13615905167, 'load\_duration': 12595585125, 'prompt\_eval\_count': 132, 'prompt\_eval\_duration': 383249084, 'eval\_count': 57, 'eval\_duration': 635981541, 'model\_name': 'gpt-oss:20b'}, id='run--dee7da1a-ed5e-4227-8081-ed98c5e9a0cf-0', usage\_metadata={'input\_tokens': 132, 'output\_tokens': 57, 'total\_tokens': 189})

Chain Components: str Parser

Finally, let's set-up our StrOutputParser() which will transform our model's output into a simple str to be provided to the user.

NOTE: You can see us leveraging LCEL in the example below to avoid needing to do nested calls.

**from** langchain\_core.output\_parsers **import** StrOutputParser

generator\_chain **=** chat\_prompt **|** ollama\_chat\_model **|** StrOutputParser()

generator\_chain**.**invoke({"context" : "Paris is the capital of France", "query" : "What is the capital of France?"})

'Paris'

generate Node:

Now we can create our generate Node!

**def** generate(state: State) **->** State:

generator\_chain **=** chat\_prompt **|** ollama\_chat\_model **|** StrOutputParser()

response **=** generator\_chain**.**invoke({"query" : state["question"], "context" : state["context"]})

**return** {"response" : response}

Now we can start defining our graph!

Think of the graph's state as a blank canvas that we can add nodes and edges to.

Every graph starts with two special nodes - START and END - the act as the entry and exit point to the other nodes in the graphs.

All valid graphs must start at the START node and end at the END node.

*# Start with the blank canvas*

graph\_builder **=** StateGraph(State)

Now we can add a sequence to our "canvas" (graph) - this can be done by providing a list of nodes, the will automatically have edges that connect the i-th element to the i+1-th element in the list. The final element will be added to the END node unless otherwise specified.

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

Next, let's connect our START node to our retrieve node by adding an edge.

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

<langgraph.graph.state.StateGraph at 0x122c5a270>

Finally we can compile our graph! This will do basic verification to ensure that the Runnables have the correct inputs/outputs and can be matched.

graph **=** graph\_builder**.**compile()

Finally, we can visualize our graph!

graph

A diagram of a diagram

AI-generated content may be incorrect.

Let's take it for a spin!

We invoke our graph like we do any other Runnable in LCEL!

NOTE: That's right, even a compiled graph is a Runnable!

**from** IPython.display **import** Markdown, display

response **=** graph**.**invoke({"question" : "What are the most common ways people use AI in their work?"})

display(Markdown(response["response"]))

Based on the study’s categorization, the most common ways people use AI in their work are grouped into four broad activities:

1. **Work‑related tasks** – using AI to help with day‑to‑day professional duties (e.g., drafting emails, preparing documents, or other job‑specific tasks).
2. **Asking** – seeking information or advice from the AI (e.g., “User is asking for help fixing a Python Databricks error”).
3. **Doing** – having the AI carry out a task directly (e.g., generating code, summarizing data).
4. **Expressing** – using AI to brainstorm, rewrite, or refine content (e.g., “User is rewriting an email to neighbors about plumbing to be more friendly”).

These four categories capture the most frequent ways people employ AI to support their work.

response **=** graph**.**invoke({"question" : "Do people use AI for their personal lives?"})

display(Markdown(response["response"]))

Yes. The study’s examples show people using ChatGPT for everyday, personal tasks—e.g., “User is rewriting an email to neighbors about plumbing to be more friendly,” “User is complaining about grandmother,” and other non‑work‑related requests. These illustrate that AI is used in personal life contexts.

response **=** graph**.**invoke({"question" : "What concerns or challenges do people have when using AI?"})

display(Markdown(response["response"]))

Based on the material provided, the main concern that emerges is **social‑desirability bias** when people report their use of AI.

* The cited study by Ling and Imas (“Underreporting of AI use: The role of social desirability bias”) highlights that many users may **underreport or conceal their AI usage** because they feel it could be viewed negatively or because they want to appear more competent.
* This bias creates a challenge for researchers and practitioners who rely on self‑reported data to understand how AI is actually being used, especially in **knowledge‑intensive jobs** where AI can influence productivity and decision‑making.

In short, a key challenge people face when using AI is the tendency to underreport their usage due to social‑desirability concerns, which complicates accurate assessment of AI’s impact.

response **=** graph**.**invoke({"question" : "Who is Batman?"})

display(Markdown(response["response"]))

I don't know.

❓ Question #2:

LangGraph's graph-based approach lets us visualize and manage complex flows naturally. How could we extend our current implementation to handle edge cases? For example:

* What if the retriever finds no relevant context?
* What if the response needs fact-checking? Consider how you would modify the graph to handle these scenarios.

✅ Answers

2.1 #Your answer here

2.2 #Your answer here

Ollama Setup and Testing

<https://github.com/AI-Maker-Space/AIE8/blob/main/04_Production_RAG/Ollama_Setup_and_Testing.ipynb>  
  
This notebook will help you set up and test Ollama with LangChain connectors before starting the main RAG assignment.

Prerequisites

1. **Install Ollama** from [https://ollama.ai](https://ollama.ai/)
   * On Linux/Mac: curl https://ollama.ai/install.sh | sh
   * On Windows: Download and run the installer
2. **Verify Installation**
   * Run ollama -v in your terminal
   * Should show version 0.11.10 or greater
3. **Pull Required Models**
4. *# For chat/inference*
5. ollama pull gpt-oss:20b
6. *# For embeddings*
7. ollama pull embeddinggemma:latest

Step 1: Test Ollama Connection

First, let's verify that Ollama is running and accessible:

**import** requests

**import** json

*# Test if Ollama is running*

**try**:

response **=** requests**.**get('http://localhost:11434/api/tags')

**if** response**.**status\_code **==** 200:

models **=** json**.**loads(response**.**text)

print("✅ Ollama is running!")

print("\nAvailable models:")

**for** model **in** models**.**get('models', []):

print(f" - {model['name']}")

**else**:

print("❌ Ollama is not responding properly")

**except** requests**.**exceptions**.**ConnectionError:

print("❌ Cannot connect to Ollama. Make sure it's running!")

print("Start Ollama by running 'ollama serve' in a terminal")

✅ Ollama is running!

Available models:

- gpt-oss:20b

- embeddinggemma:latest

Step 2: Test Embeddings with Ollama

Now let's test creating embeddings using the LangChain Ollama connector:

**from** langchain\_ollama **import** OllamaEmbeddings

*# Initialize the embedding model*

embedding\_model **=** OllamaEmbeddings(

model**=**"embeddinggemma:latest",

base\_url**=**"http://localhost:11434" *# Default Ollama URL*

)

print("✅ Embedding model initialized")

✅ Embedding model initialized

*# Test embedding a single query*

test\_query **=** "What is the meaning of life?"

print(f"Embedding query: '{test\_query}'")

embedding **=** embedding\_model**.**embed\_query(test\_query)

print(f"\n✅ Successfully created embedding!")

print(f"Embedding dimension: {len(embedding)}")

print(f"First 10 values: {embedding[:10]}")

Embedding query: 'What is the meaning of life?'

✅ Successfully created embedding!

Embedding dimension: 768

First 10 values: [-0.14624307, 0.029132523, 0.037615955, -0.02487349, -0.02655731, 0.016060056, -0.027486855, 0.027256342, 0.01139732, -2.40085e-05]

*# Test embedding multiple documents*

test\_documents **=** [

"The quick brown fox jumps over the lazy dog.",

"Machine learning is a subset of artificial intelligence.",

"Python is a popular programming language for data science."

]

print("Embedding multiple documents...")

embeddings **=** embedding\_model**.**embed\_documents(test\_documents)

print(f"\n✅ Successfully created {len(embeddings)} embeddings!")

**for** i, doc **in** enumerate(test\_documents):

print(f"\nDocument {i**+**1}: '{doc[:50]}...'")

print(f" Embedding dimension: {len(embeddings[i])}")

print(f" First 5 values: {embeddings[i][:5]}")

Embedding multiple documents...

✅ Successfully created 3 embeddings!

Document 1: 'The quick brown fox jumps over the lazy dog....'

Embedding dimension: 768

First 5 values: [-0.14781645, 0.002890088, 0.052144155, -0.029089162, -0.036695607]

Document 2: 'Machine learning is a subset of artificial intelli...'

Embedding dimension: 768

First 5 values: [-0.1240763, -0.0027511106, -0.00032809668, 0.010076388, 0.001677464]

Document 3: 'Python is a popular programming language for data ...'

Embedding dimension: 768

First 5 values: [-0.16269195, -0.010295422, 0.025464684, 0.000692403, -0.01887165]

Step 3: Test Model Inference with Ollama

Now let's test using Ollama for text generation/inference using the LangChain connector:

**from** langchain\_ollama **import** ChatOllama

**from** langchain\_core.messages **import** HumanMessage, SystemMessage

*# Initialize the chat model*

chat\_model **=** ChatOllama(

model**=**"gpt-oss:20b",

temperature**=**0.6,

base\_url**=**"http://localhost:11434",

verbose**=True**

)

print("✅ Chat model initialized")

✅ Chat model initialized

Let's add a procedure to measure the inference performance of the model

**def** detailed\_performance\_metrics(response\_metadata):

"""

Calculate comprehensive performance metrics from Ollama response metadata

"""

*# Extract all timing data (in nanoseconds)*

total\_duration **=** response\_metadata**.**get('total\_duration', 0)

load\_duration **=** response\_metadata**.**get('load\_duration', 0)

prompt\_eval\_duration **=** response\_metadata**.**get('prompt\_eval\_duration', 0)

eval\_duration **=** response\_metadata**.**get('eval\_duration', 0)

*# Extract token counts*

prompt\_eval\_count **=** response\_metadata**.**get('prompt\_eval\_count', 0)

eval\_count **=** response\_metadata**.**get('eval\_count', 0)

*# Convert to seconds*

total\_seconds **=** total\_duration **/** 1\_000\_000\_000

load\_seconds **=** load\_duration **/** 1\_000\_000\_000

prompt\_eval\_seconds **=** prompt\_eval\_duration **/** 1\_000\_000\_000

eval\_seconds **=** eval\_duration **/** 1\_000\_000\_000

*# tokens per second*

tokens\_per\_second **=** eval\_count **/** eval\_seconds

*# Calculate metrics*

metrics **=** {

'generation\_tokens\_per\_second': eval\_count **/** eval\_seconds **if** eval\_seconds **>** 0 **else** 0,

'prompt\_tokens\_per\_second': prompt\_eval\_count **/** prompt\_eval\_seconds **if** prompt\_eval\_seconds **>** 0 **else** 0,

'total\_tokens': prompt\_eval\_count **+** eval\_count,

'total\_time\_seconds': total\_seconds,

'load\_time\_seconds': load\_seconds,

'generation\_time\_seconds': eval\_seconds,

'prompt\_processing\_time\_seconds': prompt\_eval\_seconds,

}

print(f"Total tokens: {metrics['total\_tokens']}")

print(f"Total time seconds: {metrics['total\_time\_seconds']}")

print(f"Load time seconds: {metrics['load\_time\_seconds']}")

print(f"Generation time seconds: {metrics['generation\_time\_seconds']}")

print(f"Prompt processing time seconds: {metrics['prompt\_processing\_time\_seconds']}")

print(f"Generation tokens per second: {metrics['generation\_tokens\_per\_second']}")

print(f"Prompt tokens per second: {metrics['prompt\_tokens\_per\_second']}")

**return** metrics

*# Test simple inference*

prompt **=** "Explain quantum computing in one sentence."

print(f"Prompt: {prompt}")

print("\nGenerating response...")

response **=** chat\_model**.**invoke(prompt)

print(f"\n✅ Response generated!")

print(f"\nModel output: {response**.**content}")

Prompt: Explain quantum computing in one sentence.

Generating response...

✅ Response generated!

Model output: Quantum computing harnesses qubits that can occupy superpositions and become entangled, allowing quantum gates to perform massively parallel operations that can solve certain problems exponentially faster than classical computers.

\_ **=** detailed\_performance\_metrics(response**.**response\_metadata)

Total tokens: 191

Total time seconds: 2.793579209

Load time seconds: 1.101918542

Generation time seconds: 1.336429292

Prompt processing time seconds: 0.354481542

Generation tokens per second: 87.54671923189184

Prompt tokens per second: 208.75558028350036

*# Test with system message and human message*

messages **=** [

SystemMessage(content**=**"You are a helpful AI assistant that explains complex topics simply."),

HumanMessage(content**=**"What is machine learning?")

]

print("Sending messages to model...")

response **=** chat\_model**.**invoke(messages)

print(f"\n✅ Response generated!")

print(f"\nModel output: {response**.**content}")

Sending messages to model...

✅ Response generated!

Model output: \*\*Machine learning (ML)\*\* is a way to let computers learn patterns and make decisions from data instead of being told exactly how to do everything.

---

### The basic idea

| Step | What happens | Analogy |

|------|--------------|---------|

| 1. \*\*Collect data\*\* | Gather examples (images, text, numbers, etc.) | A student collects notes from many books |

| 2. \*\*Teach the model\*\* | The computer looks at the data and finds regularities | The student reads the notes and notices common themes |

| 3. \*\*Make predictions\*\* | When given new data, the model uses the patterns it learned to guess the answer | The student uses what they learned to solve a new problem |

---

### Types of learning

| Type | How it works | Everyday example |

|------|--------------|------------------|

| \*\*Supervised\*\* | The data comes with the correct answer (label). The model learns to map input → output. | Spam filter: emails labeled “spam” or “not spam”. |

| \*\*Unsupervised\*\* | No labels. The model finds hidden structure (clusters, associations). | Grouping customers by buying habits. |

| \*\*Reinforcement\*\* | The model gets rewards or penalties for actions and learns a strategy. | A game‑playing AI learning to win by trial and error. |

---

### Why it matters

- \*\*Automation\*\*: Let computers do tasks that need pattern recognition (image recognition, speech recognition, recommendation engines).

- \*\*Speed\*\*: Once trained, models can process huge amounts of data quickly.

- \*\*Adaptability\*\*: Models can improve over time as they see more data.

---

### A quick example

1. \*\*Collect\*\*: 1,000 pictures of cats and dogs, each labeled “cat” or “dog”.

2. \*\*Train\*\*: The computer looks at pixel patterns and learns which features (eyelash length, ear shape, etc.) correlate with each label.

3. \*\*Predict\*\*: When you show it a new picture, it outputs “cat” or “dog” based on what it learned.

---

### Bottom line

Machine learning is \*\*training a computer to recognize patterns in data and use those patterns to make decisions or predictions\*\*—much like how we learn from experience.

\_ **=** detailed\_performance\_metrics(response**.**response\_metadata)

Total tokens: 622

Total time seconds: 6.603062583

Load time seconds: 0.105166042

Generation time seconds: 6.159919833

Prompt processing time seconds: 0.313681375

Generation tokens per second: 86.04008077518758

Prompt tokens per second: 293.29124178953884

Step 4: Test Streaming Response

Ollama supports streaming responses, which is useful for real-time applications:

*# Test streaming*

prompt **=** "Write a haiku about artificial intelligence."

print(f"Prompt: {prompt}")

print("\nStreaming response:")

print("-" **\*** 40)

**for** chunk **in** chat\_model**.**stream(prompt):

print(chunk**.**content, end**=**"", flush**=True**)

print("\n" **+** "-" **\*** 40)

print("\n✅ Streaming completed!")

Prompt: Write a haiku about artificial intelligence.

Streaming response:

----------------------------------------

Silent code hums deep

Minds born of silicon dreams

Stars in data glow

----------------------------------------

✅ Streaming completed!

Summary

If all the tests above passed, you're ready to use Ollama with LangChain! Here's what we tested:

✅ **Embeddings**:

* Created embeddings for single queries
* Created embeddings for multiple documents
* Verified embedding dimensions

✅ **Model Inference**:

* Simple text generation
* Chat with system and human messages
* Streaming responses
* Integration with LangChain chains

Troubleshooting

If you encounter issues:

1. **Model Not Found**: Pull the required models (ollama pull <model-name>)
2. **Slow Performance**: Ollama models run on CPU by default. For better performance:
   * Use smaller models for testing
   * Consider GPU acceleration if available
3. **Memory Issues**: Large models require significant RAM. Try smaller variants if needed.

Next Steps

Now you're ready to proceed with the main RAG assignment using Ollama!

https://github.com/AI-Maker-Space/AIE8/blob/main/04\_Production\_RAG/pyproject.toml

*[project]*

*name = "04-producton-rag"*

*version = "0.1.0"*

*description = "Add your description here"*

*readme = "README.md"*

*requires-python = "==3.12.\*"*

*dependencies = [*

*"jupyter>=1.1.1",*

*"langchain-community>=0.3.26",*

*"langchain-core>=0.3.67",*

*"langchain-ollama>=0.3.8",*

*"langchain-openai>=0.3.27",*

*"langchain-qdrant>=0.2.0",*

*"langgraph>=0.5.0",*

*"langsmith>=0.4.4",*

*"pandas>=2.3.0",*

*"pymupdf>=1.26.1",*

*]*