

## Interacting with LLMs and Retrieval Augmented Generation (RAG)

### Introduction

Welcome to Module 3 of Week 4 of our AI course. Today we're going to learn about interacting with Large Language Models (LLMs) and Retrieval Augmented Generation (RAG) - essential skills in the modern AI landscape. By the end of this module, you'll understand how to communicate with LLMs through APIs, build applications using frameworks like LangChain, and enhance LLM capabilities with external knowledge through RAG.

Let's start with an interesting question:

#### Interactive Question #1:

**What is the main benefit of directly interacting with LLMs through APIs?**

- A) Free access to all models
- B) No coding knowledge required
- C) Integration of AI capabilities into custom applications
- D) Unlimited processing power

### 1. Understanding LLM Interactions

Let's first explore the different ways we can interact with Large Language Models and understand their capabilities and limitations.

#### 1.1 Types of LLM Interactions

There are several ways to interact with Large Language Models:

1. **Web Interfaces:** Services like Gemini, Claude, or Bard that offer chat-based interactions
2. **API Calls:** Programmatic access to LLMs through code
3. **Local Deployment:** Running open-source models on your own hardware
4. **Integrated Tools:** Using LLMs embedded in software like Microsoft Copilot

Each approach has different trade-offs in terms of ease of use, customization, privacy, and cost.

## 1.2 Understanding LLM APIs


LLM APIs (Application Programming Interfaces) allow your programs to communicate with language models hosted by AI providers. This means you can add AI capabilities to your applications without building the models yourself.

Key benefits include:

- Access to powerful pre-trained models
- No need for specialized hardware
- Simple integration with existing applications
- Pay-as-you-go pricing models

## 1.3 Working with Gemini API

Google's Gemini is a powerful LLM with a user-friendly API. Here's how to get started:



```
# First, install the required package
!pip install google-generativeai

# Import the library
import google.generativeai as genai

# Configure with your API key
genai.configure(api_key="YOUR_API_KEY")

# Initialize the model
model = genai.GenerativeModel('gemini-pro')

# Generate a response
response = model.generate_content("Explain quantum computing in simple terms")

# Print the response
print(response.text)
```

### Interactive Question #2:

In the code above, what does 'gemini-pro' represent?

- A) The AI provider's name
- B) The specific model being used
- C) The programming language
- D) The API version

## 1.4 Understanding Model Parameters

When working with LLMs, you can adjust various parameters to control the output:

- **Temperature:** Controls randomness (0.0 = deterministic, 1.0 = more creative)
- **Max tokens:** Limits the length of the generated text
- **Top-p (nucleus sampling):** Controls diversity of responses
- **Top-k:** Limits vocabulary choices to the k most likely tokens

Example with parameters:

```
response = model.generate_content(  
    "Write a short poem about AI",  
    generation_config={  
        "temperature": 0.7,  
        "max_output_tokens": 100,  
        "top_p": 0.9,  
        "top_k": 40  
    }  
)
```



## 2. Introduction to LangChain

LangChain is a framework designed to build applications powered by language models. It simplifies creating complex LLM applications by providing reusable components.

### 2.1 Key Concepts in LangChain

LangChain organizes functionality around these core components:

- **Models:** Wrappers around LLMs (like Gemini, Claude, etc.)
- **Prompts:** Templates for generating inputs to models
- **Chains:** Sequences of operations (e.g., prompt → LLM → output parser)
- **Memory:** Storage for conversation history
- **Tools:** Functions that LLMs can use (calculators, search engines, etc.)
- **Agents:** LLMs that decide which tools to use to accomplish a task

### 2.2 Basic LangChain Example

```
# Install required packages
!pip install langchain-google-genai langchain

# Import libraries
from langchain_google_genai import ChatGoogleGenerativeAI
from langchain.chains import LLMChain
from langchain.prompts import PromptTemplate

# Initialize the LLM
llm = ChatGoogleGenerativeAI(model="gemini-pro", google_api_key="YOUR_API_KEY")

# Create a prompt template
prompt = PromptTemplate(
    input_variables=["topic"],
    template="Explain {topic} in simple terms a child would understand."
)

# Create a chain
chain = LLMChain(llm=llm, prompt=prompt)

# Run the chain
result = chain.run(topic="black holes")
print(result)
```

### Interactive Question #3: In LangChain, what is the purpose of a 'Chain'?

- A) To secure API connections
- B) To link multiple operations together in sequence
- C) To restrict model outputs
- D) To track token usage

### 3. Introduction to LangGraph

LangGraph extends LangChain to create more sophisticated applications with stateful, cyclic flows.

#### 3.1 What is LangGraph?

LangGraph is a library for building stateful, multi-actor applications using LLMs. It allows you to:

- Create complex workflows with multiple steps
- Implement feedback loops and recursion
- Maintain state across multiple interactions
- Build systems where multiple LLM agents collaborate

#### 3.2 LangGraph Basics

LangGraph uses a graph-based approach where:

- Nodes represent operations (like calling an LLM)
- Edges represent transitions between operations
- The graph maintains state as execution proceeds

Here's a simple example of a decision-making flow:

```
from langgraph.graph import StateGraph
from langchain_core.messages import HumanMessage, AIMessage

# Define graph states
class GraphState:
    messages: list
    decision: str = None

# Create a state graph
graph = StateGraph(GraphState)

# Define nodes
def get_decision(state):
    messages = state["messages"]
    # Call LLM to make a decision based on messages
    decision = llm.invoke([
        HumanMessage(content="Based on the conversation, should I proceed with A or B?"),
        *messages
    ])
    return {"decision": decision.content}

# Add nodes to graph
graph.add_node("make_decision", get_decision)

# Define edges
graph.add_edge("make_decision", "end")

# Compile the graph
compiled_graph = graph.compile()
```

## 4. LLM Limitations and the Need for RAG

Despite their impressive capabilities, LLMs have several important limitations:

1. **Knowledge Cutoff:** LLMs only know information up to their training cutoff date
2. **Hallucinations:** They can generate plausible-sounding but incorrect information
3. **No Real-time Data:** They can't access current information without assistance
4. **No Private Knowledge:** They don't know about your specific documents or data
5. **No Source Citations:** Typically can't provide references for their answers

## 5. What is Retrieval Augmented Generation (RAG)?

Now we can explore how RAG addresses these limitations!

### 5.1 The Core Concept

RAG combines the generation capabilities of LLMs with retrieval of relevant information from external sources.

The process works in three main steps:

1. User query is received
2. Relevant documents are retrieved from a knowledge base
3. The documents and query are sent to the LLM to generate an informed response

This approach helps overcome key limitations of LLMs:

- Knowledge cutoff dates (by providing up-to-date information)
- Hallucinations (by grounding responses in retrieved facts)
- Domain-specific knowledge (by accessing specialized information)

### 5.2 How RAG Differs from Fine-tuning

RAG	Fine-tuning
Incorporates external knowledge at query time	Incorporates knowledge during model training
Easy to update knowledge	Requires re-training to update knowledge
Lower computational requirements	Computationally intensive
Works with any LLM	Changes the underlying model
Better for factual Q&A	Better for changing model behavior

## 6. Components of a RAG System

Let's break down the key components needed to build a RAG system:

### 6.1 Document Processing

Before retrieving information, you need to prepare your documents:

1. **Document Loading:** Import text from PDFs, websites, databases, etc.
2. **Text Chunking:** Split documents into smaller pieces (paragraphs, sentences)
3. **Text Cleaning:** Remove irrelevant content, fix formatting issues

```
# Example of document loading and chunking
from langchain_community.document_loaders import PyPDFLoader
from langchain.text_splitter import RecursiveCharacterTextSplitter

# Load a PDF
loader = PyPDFLoader("your_document.pdf")
documents = loader.load()

# Split into chunks
text_splitter = RecursiveCharacterTextSplitter(
    chunk_size=1000,
    chunk_overlap=100
)
chunks = text_splitter.split_documents(documents)
```

### 6.2 Vector Embeddings

To find relevant information, RAG systems convert text into vector embeddings (numerical representations that capture meaning).

Popular embedding models include:

- **Sentence Transformers:** Open-source models like BERT
- **FastEmbed:** Lightweight and fast embedding generation
- **Google Embeddings:** Available through Vertex AI

```
# Creating embeddings
from langchain_community.embeddings import HuggingFaceEmbeddings

# Initialize embedding model
embeddings = HuggingFaceEmbeddings(model_name="all-MiniLM-L6-v2")

# Create embeddings for our chunks
embedded_chunks = embeddings.embed_documents([chunk.page_content for chunk in chunks])
```

### 6.3 Vector Databases

Vector databases store embeddings and allow efficient similarity searches:

- **ChromaDB**: Simple, open-source vector database
- **FAISS**: Facebook AI Similarity Search library
- **Pinecone**: Managed vector database service
- **Qdrant**: Self-hosted or cloud vector search engine

```
# Using ChromaDB
from langchain_community.vectorstores import Chroma

# Create a vector store
vector_db = Chroma.from_documents(
    documents=chunks,
    embedding=embeddings,
    persist_directory="./chroma_db"
)

# Search for similar documents
retrieved_docs = vector_db.similarity_search("What is quantum computing?", k=3)
```

#### Interactive Question #4:

What is the primary purpose of using vector embeddings in a RAG system?

- A) To compress documents to save storage space
- B) To convert text into numerical representations for similarity search
- C) To translate documents into multiple languages
- D) To speed up the LLM's processing time

### 6.4 Retrieval Strategies

Different strategies can be used to retrieve relevant information:

- **Similarity Search**: Find documents most similar to the query
- **Hybrid Search**: Combine keyword and semantic search
- **Maximum Marginal Relevance (MMR)**: Balance relevance with diversity
- **Re-ranking**: Use a second model to re-order initial search results



## 6.5 Prompt Construction

How you combine retrieved information with the user's query affects results:

*# Basic RAG prompt construction*

```
def create_rag_prompt(query, retrieved_docs):  
    context = "\n\n".join([doc.page_content for doc in retrieved_docs])  
    prompt = f"""  
    Answer the question based on the following context:  
  
    Context:  
    {context}  
  
    Question: {query}  
  
    Answer:  
    """"  
    return prompt
```



## 7. Building a Simple RAG System with LangChain

Let's put everything together to build a basic RAG system:

```
from langchain_community.document_loaders import PyPDFLoader
from langchain.text_splitter import RecursiveCharacterTextSplitter
from langchain_community.embeddings import HuggingFaceEmbeddings
from langchain_community.vectorstores import Chroma
from langchain_google_genai import ChatGoogleGenerativeAI
from langchain.chains import create_retrieval_chain
from langchain.chains.combine_documents import create_stuff_documents_chain
from langchain_core.prompts import ChatPromptTemplate

# 1. Load and process documents
loader = PyPDFLoader("your_document.pdf")
documents = loader.load()

text_splitter = RecursiveCharacterTextSplitter(chunk_size=1000, chunk_overlap=100)
chunks = text_splitter.split_documents(documents)

# 2. Create embeddings and vector store
embeddings = HuggingFaceEmbeddings(model_name="all-MiniLM-L6-v2")
vector_db = Chroma.from_documents(documents=chunks, embedding=embeddings)
retriever = vector_db.as_retriever(search_kwargs={"k": 3})

# 3. Setup the LLM
llm = ChatGoogleGenerativeAI(model="gemini-pro", google_api_key="YOUR_API_KEY")

# 4. Create the RAG prompt
prompt = ChatPromptTemplate.from_template("""
Answer the following question based only on the provided context:

Context:
{context}

Question: {input}

Answer:
""")

# 5. Create the RAG chain
document_chain = create_stuff_documents_chain(llm, prompt)
retrieval_chain = create_retrieval_chain(retriever, document_chain)

# 6. Query the system
response = retrieval_chain.invoke({"input": "What is quantum computing?"})
print(response["answer"])
```

## Summary

This module explored the fundamentals of interacting with LLMs and implementing Retrieval Augmented Generation (RAG). We started by examining different ways to interact with LLMs, from web interfaces to APIs, and learned how to use the Gemini API to build AI-powered applications.

We then explored frameworks like LangChain and LangGraph that simplify building complex LLM applications with multiple components and workflows. These frameworks provide abstractions for common operations and help manage the interaction between different parts of an LLM application.

Finally, we dove deep into RAG, which enhances LLMs by retrieving relevant information from external sources before generating responses. The core components include document processing, vector embeddings, vector databases, retrieval strategies, and prompt construction.

RAG solves key LLM limitations by providing access to up-to-date information, reducing hallucinations, and enabling domain-specific knowledge without retraining models.

## Glossary

- **Chunking:** Splitting documents into smaller, manageable pieces for processing
- **Embeddings:** Numerical vector representations of text that capture semantic meaning
- **Hallucination:** When an LLM generates factually incorrect information
- **Knowledge Cutoff:** The date after which an LLM has no training data
- **LangChain:** A framework for building applications with LLMs
- **LangGraph:** A library for building stateful multi-actor LLM applications
- **Parameter:** Adjustable value that controls aspects of model behavior (like temperature)
- **Prompt Engineering:** The practice of designing effective prompts for LLMs
- **RAG:** Retrieval Augmented Generation, combining LLMs with external knowledge retrieval
- **Retriever:** Component that finds relevant documents from a knowledge base
- **Similarity Search:** Finding documents with embeddings closest to a query
- **Token:** The basic unit of text that LLMs process (roughly 4 characters in English)
- **Vector Database:** Specialized database for storing and searching vector embeddings

## Multiple Choice Answers

1. C) Integration of AI capabilities into custom applications
2. B) The specific model being used
3. B) To link multiple operations together in sequence
4. B) To convert text into numerical representations for similarity search