[Week 7] Building Your Own LLM Application

ETMI5: Explain to Me in 5

In the previous parts of the course we covered techniques such as prompting, RAG, and fine-tuning, this section will adopt a practical, hands-on approach to showcase how LLMs can be employed in application development. We'll start with basic examples and progressively incorporate more advanced functionalities like chaining, memory management, and tool integration. Additionally, we'll explore implementations of RAG and fine-tuning. Finally, by integrating these concepts, we'll learn how to construct LLM agents effectively.

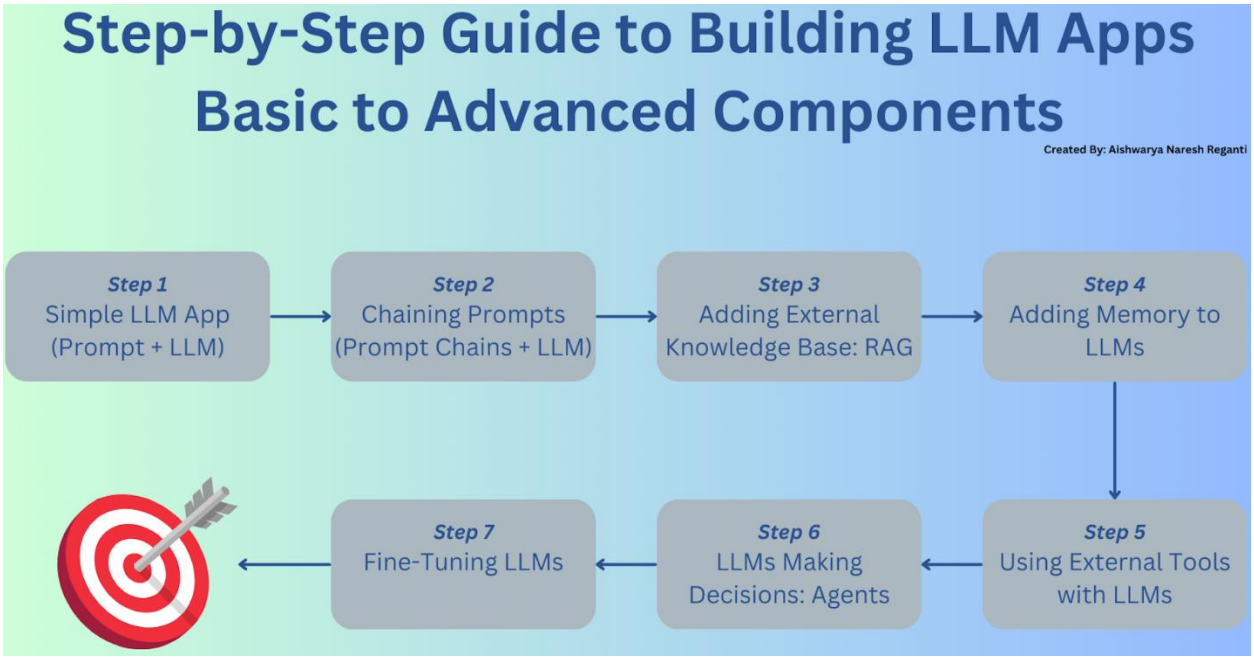
Introduction

As LLMs have become increasingly prevalent, there are now multiple ways to utilize them. We'll start with basic examples and gradually introduce more advanced features, allowing you to build upon your understanding step by step.

This guide is designed to cover the basics, aiming to familiarize you with the foundational elements through simple applications. These examples serve as starting points and are not intended for production environments. For insights into deploying applications at scale, including discussions on LLM tools, evaluation, and more, refer to our content from previous weeks. As we progress through each section, we'll gradually move from basic to more advanced components.

In every section, we'll not only describe the component but also provide resources where you can find code samples to help you develop your own implementations. There are several frameworks available for developing your application, with some of the most well-known being LangChain, LlamaIndex, Hugging Face, and Amazon Bedrock, among others. Our goal is to supply resources from a broad array of these frameworks, enabling you to select the one that best fits the needs of your specific application.

As you explore each section, select a few resources to help build the app with the component and proceed further.



llm_app_steps.png

1. Simple LLM App (Prompt + LLM)

Prompt: A prompt, in this context, is essentially a carefully constructed request or instruction that guides the model in generating a response. It's the initial input given to the LLM that outlines the task you want it to perform or the question you need answered. In the second week's content, we delved extensively into prompt engineering, please head back to older content to learn more.

The foundational aspect of LLM application development is the interaction between a user-defined prompt and the LLM itself. This process involves crafting a prompt that clearly communicates the user's request or question, which is then processed by the LLM to generate a response. For example:

```
# Define the prompt template with placeholders
prompt_template = "Provide expert advice on the following topic: {topic}."
# Fill in the template with the actual topic
prompt = prompt_template.replace("{topic}", topic)
# API call to an LLM
llm_response = call_llm_api(topic)
```

Observe that the prompt functions as a template rather than a fixed string, improving its reusability and flexibility for modifications at run-time. The complexity of the prompt can vary; it can be crafted with simplicity or detailed intricacy depending on the requirement.

Resources/Code

1. [**Documentation/Code**] LangChain cookbook for simple LLM Application (link)
2. [**Video**] Hugging Face + LangChain in 5 mins by AI Jason (link)
3. [**Documentation/Code**] Using LLMs with LlamaIndex (link)
4. [**Blog**] Getting Started with LangChain by Leonie Monigatti (link)
5. [**Notebook**] Running an LLM on your own laptop by LearnDataWithMark (link)

---

2. Chaining Prompts (Prompt Chains + LLM)

Although utilizing prompt templates and invoking LLMs is effective, sometimes, you might need to ask the LLM several questions, one after the other, using the answers you got before to ask the next question. Imagine this: first, you ask the LLM to figure out what topic your question is about. Then, using that information, you ask it to give you an expert answer on that topic. This step-by-step process, where one answer leads to the next question, is called "chaining." Prompt Chains are essentially this sequence of chains used for executing a series of LLM actions.

LangChain has emerged as a widely-used library for creating LLM applications, enabling the chaining of multiple questions and answers with the LLM to produce a singular final response. This approach is particularly beneficial for larger projects requiring multiple steps to achieve the desired outcome. The example discussed illustrates a basic method of chaining. LangChain's documentation offers guidance on more complex chaining techniques.

```
prompt1 ="what topic is the following question about-{question}?"
prompt2 = "Provide expert advice on the following topic: {topic}."
```

Resources/Code

1. **[Article]** **Prompt Chaining Article on Prompt Engineering Guide(link)
2. [**Video**] LLM Chains using GPT 3.5 and other LLMs — LangChain #3 James Briggs (link)
3. [**Video**] LangChain Basics Tutorial #2 Tools and Chains by Sam Witteveen (link)
4. [**Code**] LangChain tools and Chains Colab notebook by Sam Witteveen (link)

---

3. Adding External Knowledge Base: Retrieval-Augmented Generation (RAG)

Next, we'll explore a different type of application. If you've followed our previous discussions, you're aware that although LLMs excel at providing information, their knowledge is limited to what was available up until their last training session. To generate meaningful outputs beyond this point, they require access to an external knowledge base. This is the role that Retrieval-Augmented Generation (RAG) plays.

Retrieval-Augmented Generation, or RAG, is like giving your LLM a personal library to check before answering. Before the LLM comes up with something new, it looks through a

bunch of information (like articles, books, or the web) to find stuff related to your question. Then, it combines what it finds with its own knowledge to give you a better answer. This is super handy when you need your app to pull in the latest information or deep dive into specific topics.

To implement RAG (Retrieval-Augmented Generation) beyond the LLM and prompts, you'll need the following technical elements:

A knowledge base, specifically a vector database

A comprehensive collection of documents, articles, or data entries that the system can draw upon to find information. This database isn't just a simple collection of texts; it's often transformed into a vector database. Here, each item in the knowledge base is converted into a high-dimensional vector representing the semantic meaning of the text. This transformation is done using models similar to the LLM but focused on encoding texts into vectors.

The purpose of having a vectorized knowledge base is to enable efficient similarity searches. When the system is trying to find information relevant to a user's query, it converts the query into a vector using the same encoding process. Then, it searches the vector database for vectors (i.e., pieces of information) that are closest to the query vector, often using measures like cosine similarity. This process quickly identifies the most relevant pieces of information within a vast database, something that would be impractical with traditional text search methods.

Retrieval Component

The retrieval component is the engine that performs the actual search of the knowledge base to find information relevant to the user's query. It's responsible for several key tasks:

1. **Query Encoding:** It converts the user's query into a vector using the same model or method used to vectorize the knowledge base. This ensures that the query and the database entries are in the same vector space, making similarity comparison possible.
2. **Similarity Search:** Once the query is vectorized, the retrieval component searches the vector database for the closest vectors. This search can be based on various algorithms designed to efficiently handle high-dimensional data, ensuring that the process is both fast and accurate.
3. **Information Retrieval:** After identifying the closest vectors, the retrieval component fetches the corresponding entries from the knowledge base. These entries are the pieces of information deemed most relevant to the user's query.
4. **Aggregation (Optional):** In some implementations, the retrieval component may also aggregate or summarize the information from multiple sources to provide a consolidated response. This step is more common in advanced RAG systems that aim to synthesize information rather than citing sources directly.

In the RAG framework, the retrieval component's output (i.e., the retrieved information) is then fed into the LLM along with the original query. This enables the LLM to generate

responses that are not only contextually relevant but also enriched with the specificity and accuracy of the retrieved information. The result is a hybrid model that leverages the best of both worlds: the generative flexibility of LLMs and the factual precision of dedicated knowledge bases.

By combining a vectorized knowledge base with an efficient retrieval mechanism, RAG systems can provide answers that are both highly relevant and deeply informed by a wide array of sources. This approach is particularly useful in applications requiring up-to-date information, domain-specific knowledge, or detailed explanations that go beyond the pre-existing knowledge of an LLM.

Frameworks like LangChain already have good abstractions in place to build RAG frameworks

A simple example from LangChain is shown here

Resources/Code

1. [**Article**] All You Need to Know to Build Your First LLM App by Dominik Polzer (link)
2. [**Video**] RAG from Scratch series by LangChain (link)
3. [**Video**] A deep dive into Retrieval-Augmented Generation with LlamaIndex (link)
4. [**Notebook**] RAG using LangChain with Amazon Bedrock Titan text, and embedding, using OpenSearch vector engine notebook (link)
5. [**Video**] LangChain - Advanced RAG Techniques for better Retrieval Performance by Coding Crashcourses (link)
6. [**Video**] Chatbots with RAG: LangChain Full Walkthrough by James Briggs (link)

---

4. Adding Memory to LLMs

We've explored chaining and incorporating knowledge. Now, consider the scenario where we need to remember past interactions in lengthy conversations with the LLM, where previous dialogues play a role.

This is where the concept of Memory comes into play as a vital component. Memory mechanisms, such as those available on platforms like LangChain, enable the storage of conversation history. For example, LangChain's ConversationBufferMemory feature allows for the preservation of messages, which can then be retrieved and used as context in subsequent interactions. You can discover more about these memory abstractions and their applications on LangChain's documentation.

Resources/Code

1. [**Article**] Conversational Memory for LLMs with LangChain by Pinecone(link)
2. [**Blog**] How to add memory to a chat LLM model by Nikolay Penkov (link)
3. [**Documentation**] Memory in LlamaIndex documentation (link)
4. [**Video**] LangChain: Giving Memory to LLMs by Prompt Engineering (link)

5. [**Video**] Building a LangChain Custom Medical Agent with Memory by (link)

---

5. Using External Tools with LLMs

Consider a scenario within an LLM application, such as a travel planner, where the availability of destinations or attractions depends on seasonal openings. Imagine we have access to an API that provides this specific information. In this case, the application must query the API to determine if a location is open. If the location is closed, the LLM should adjust its recommendations accordingly, suggesting alternative options. This illustrates a crucial instance where integrating external tools can significantly enhance the functionality of LLMs, enabling them to provide more accurate and contextually relevant responses. Such integrations are not limited to travel planning; there are numerous other situations where external data sources, APIs, and tools can enrich LLM applications. Examples include weather forecasts for event planning, stock market data for financial advice, or real-time news for content generation, each adding a layer of dynamism and specificity to the LLM's capabilities.

In frameworks like LangChain, integrating these external tools is streamlined through its chaining framework, which allows for the seamless incorporation of new elements such as APIs, data sources, and other tools.

Resources/Code

1. [**Documentation/Code**] List of LLM tools by LangChain (link)
2. [**Documentation/Code**]Tools in LlamaIndex (link)
3. [**Video**] Building Custom Tools and Agents with LangChain by Sam Witteveen (link)

---

6. LLMs Making Decisions: Agents

In the preceding sections, we explored complex LLM components like tools and memory. Now, let's say we want our LLM to effectively utilize these elements to make decisions on our behalf.

LLM agents do exactly this, they are systems designed to perform complex tasks by combining LLMs with other modules such as planning, memory, and tool usage. These agents leverage the capabilities of LLMs to understand and generate human-like language, enabling them to interact with users and process information effectively.

For instance, consider a scenario where we want our LLM agent to assist in financial planning. The task is to analyze an individual's spending habits over the past year and provide recommendations for budget optimization.

To accomplish this task, the agent first utilizes its memory module to access stored data regarding the individual's expenditures, income sources, and financial goals. It then employs a planning mechanism to break down the task into several steps:

1. **Data Analysis**: The agent uses external tools to process the financial data, categorizing expenses, identifying trends, and calculating key metrics such as total spending, savings rate, and expenditure distribution.
2. **Budget Evaluation**: Based on the analyzed data, the LLM agent evaluates the current budget's effectiveness in meeting the individual's financial objectives. It considers factors such as discretionary spending, essential expenses, and potential areas for cost reduction.
3. **Recommendation Generation**: Leveraging its understanding of financial principles and optimization strategies, the agent formulates personalized recommendations to improve the individual's financial health. These recommendations may include reallocating funds towards savings, reducing non-essential expenses, or exploring investment opportunities.
4. **Communication**: Finally, the LLM agent communicates the recommendations to the user in a clear and understandable manner, using natural language generation capabilities to explain the rationale behind each suggestion and potential benefits.

Throughout this process, the LLM agent seamlessly integrates its decision-making abilities with external tools, memory storage, and planning mechanisms to deliver actionable insights tailored to the user's financial situation.

Here's how LLM agents combine various components to make decisions:

1. **Language Model (LLM)**: The LLM serves as the central controller or "brain" of the agent. It interprets user queries, generates responses, and orchestrates the overall flow of operations required to complete tasks.
2. **Key Modules**:
   - **Planning**: This module helps the agent break down complex tasks into manageable subparts. It formulates a plan of action to achieve the desired goal efficiently.
   - **Memory**: The memory module allows the agent to store and retrieve information relevant to the task at hand. It helps maintain the state of operations, track progress, and make informed decisions based on past observations.
   - **Tool Usage**: The agent may utilize external tools or APIs to gather data, perform computations, or generate outputs. Integration with these tools enhances the agent's capabilities to address a wide range of tasks.

Existing frameworks offer built-in modules and abstractions for constructing agents. Please refer to the resources provided below for implementing your own agent.

Resources/Code

1. [**Documentation/Code**] Agents in LangChain (link)
2. [**Documentation/Code**] Agents in LlamaIndex (link)
3. [**Video**] LangChain Agents - Joining Tools and Chains with Decisions by Sam Witteveen (link)
4. [**Article**] Building Your First LLM Agent Application by Nvidia (link)

5. [**Video**] OpenAI Functions + LangChain : Building a Multi Tool Agent by Sam Witteveen (link)

---

7. Fine-Tuning

In earlier sections, we explored using pre-trained LLMs with additional components. However, there are scenarios where the LLM must be updated with relevant information before usage, particularly when LLMs lack specific knowledge on a subject. In such instances, it's necessary to first fine-tune the LLM before applying the strategies outlined in sections 1-5 to build an application around it.

Various platforms offer fine-tuning capabilities, but it's important to note that fine-tuning demands more resources than simply eliciting responses from an LLM, as it involves training the model to understand and generate information on the desired topics.

Resources/Code

1. [**Article**] How to Fine-Tune LLMs in 2024 with Hugging Face by philschmid (link)
2. [**Video**] Fine-tuning Large Language Models (LLMs) | w/ Example Code by Shaw Talebi (link)
3. [**Video**] Fine-tuning LLMs with PEFT and LoRA by Sam Witteveen (link)
4. [**Video**] LLM Fine Tuning Crash Course: 1 Hour End-to-End Guide by AI Anytime (link)
5. [**Article**] How to Fine-Tune an LLM series by Weights and Biases (link)

Read/Watch These Resources (Optional)

1. List of LLM notebooks by aishwaryanr (link)
2. LangChain How to and Guides by Sam Witteveen (link)
3. LangChain Crash Course For Beginners | LangChain Tutorial by codebasics (link)
4. Build with LangChain Series (link)
5. LLM hands on course by Maxime Labonne (link)