**Slide 1: Introduction to LLM-based Web Applications**

Welcome everyone to Part 5 of our course. In this part, we will explore the essential components required to build web applications powered by Large Language Models (LLMs).

**Slide 2: What We Will Learn**

Our goal is to understand the various components necessary for creating LLM-based web applications. There are two main sessions in this part.

In the first session, we will discuss backend components, including LLM orchestration frameworks, the differences between open-source and proprietary LLMs, vector embeddings and vector databases, and prompt engineering.

In the second session, we will explore Python-based frameworks for the front-end side of the application.

**Slide 3: Backend for LLM-based Applications**

In the first section, we will discuss the backend components for LLM-based applications.

We will cover orchestrating frameworks such as LangChain and LlamaIndex.

Next, we will compare open-source and proprietary LLMs and how to leverage them in practice.

Finally, I will present other essential components, including vector embeddings, vector databases, and prompt engineering.

**Slide 4: LLM Orchestration Frameworks**

Now let’s look at frameworks that play an orchestration role in building LLM-based application.

We will focus on LangChain and LlamaIndex as orchestration frameworks. LangChain supports a wide range of LLM applications, whereas LlamaIndex specializes in document-related applications. Each has unique strengths that we will explore further.

**Slide 5: LangChain**

Let’s talk about LangChain first.

LangChain is a versatile framework for developing LLM-powered applications. It allows for context-aware interactions, reasoning, and integration with various tools and resources, making it suitable for diverse applications.

**Slide 6: Why LangChain?**

LangChain serves as a comprehensive solution for orchestrating various components essential to building sophisticated language applications.

One of the primary advantages of LangChain is its ability to empower applications by making them context aware. This is achieved by connecting the language model to diverse context sources, including prompt instructions, few-shot examples, and other grounding content, which helps the model generate more accurate and relevant responses.

Additionally, LangChain enhances the reasoning capabilities of language models, enabling them to interpret and act based on the provided context. This includes determining how to answer questions appropriately and deciding on the necessary actions to take in various scenarios.

Moreover, LangChain facilitates seamless interaction with a wide range of external tools and resources.

It supports integration with diversified large language models (LLMs) such as Hugging Face, Llama, OpenAI, and Mistral AI.

The framework also connects to various data sources and search APIs, making it a versatile and powerful tool for developing robust language-based applications.

**Slide 7: LangChain Components**

Now let’s talk about the key components of the LangChain framework, including its foundational and modular elements.

The first component is LangChain-Core, which encompasses the essential abstractions and the LangChain Expression Language (LCEL). This core provides fundamental capabilities such as parallelization, fallbacks, tracing, batching, streaming, asynchronous processing, and composition, all critical for building robust and efficient language model applications.

The second component is LangChain-Community/Modules, which focus on third-party integrations. These integrations extend LangChain's functionality through components like Model I/O, retrieval mechanisms, and agent tooling, allowing for seamless interaction with external tools and resources.

The central concept of the framework is simply LangChain, which brings together chains, agents, and retrieval strategies to form the cognitive architecture of an application.

This cohesive structure ensures that LangChain can support complex and scalable language-based applications by leveraging both its core capabilities and community-driven modules.

**Slide 8: LangChain-Community/Modules**

Let’s discuss more about "LangChain-Community/Modules", the key components that extend the functionality of LangChain through various third-party integrations and modular tools. These components are essential for enhancing the capabilities and flexibility of language model-based applications.

**Model I/O** focuses on the interaction between the application and the language models. It includes elements such as the model itself, prompts for guiding the model’s responses, example selectors for providing context or few-shot learning examples, and output parsers to process the model’s responses effectively. This section is crucial for configuring and optimizing how the application communicates with and leverages language models.

**Retrieval** covers the tools and techniques necessary for obtaining and managing information. It includes retrievers for fetching relevant data, document loaders for ingesting various types of content, vector stores for efficient storage and retrieval of vector embeddings, text splitters for processing large documents into manageable chunks, and embedding models for transforming data into vector representations. This section is vital for applications that require robust information retrieval capabilities, ensuring that they can access and utilize the necessary data efficiently.

**Agent Tooling** encompasses the tools and toolkits required for building and managing intelligent agents within the application. These agents can perform a variety of tasks, from simple automation to complex decision-making processes. The tools and toolkits provided in this section facilitate the development and deployment of these agents, making it easier to integrate them into the overall application architecture.

By combining these three components, LangChain-Community/Modules provides a comprehensive suite of tools that enable developers to build powerful, flexible, and efficient language model-based applications.

**Slide 9: How to Use LangChain**

LangChain can be easily installed via pip. For leveraging LangChain in practice, I recommend checking the documentation of LangChain.

The documentation provides detailed usage instructions, such as the LangChain Expression Language, LangChain components like Prompt templates, chat models, retrievers, agents, etc.

You can access the notebook playground files for each feature.

**Slide 9: LangChain’s simple use-case**

Let’s see a simple use case of LangChain,

The code snippet begins by importing necessary modules, including load\_dotenv from dotenv, and various components from langchain, such as ChatOpenAI, ChatPromptTemplate, and StrOutputParser.

The environment variables are loaded using load\_dotenv().

A chat model is initialized with the ChatOpenAI class, specifying the model name as "gpt-3.5-turbo" and setting the temperature to 0 for deterministic outputs.

A prompt template is defined to structure the input for the language model, including a placeholder {request} for dynamic content. This template is used to create a PromptTemplate object with input variables and template arguments.

An LLM chain is created by combining the prompt template, the chat model, and the output parser.

Finally, the invoke method is used to get a response from the chain by passing the question to the variable “request”.

This example illustrates the process of setting up and using LangChain to handle and process prompts with language models efficiently.

**Slide 10: LlamaIndex**

Let’s talk about LlamaIndex

LlamaIndex is another framework designed for LLM applications, particularly those involving documents. It provides specific tools and strategies for document-based use cases, making it distinct from LangChain's broader application scope.

**Slide 11: Why LlamaIndex?**

LlamaIndex focuses on Retrieval-Augmented Generation (RAG) use cases, offering specialized tools for handling document-centric applications.

**Slide 11: How to guides LlamaIndex?**

How-to Guides LlamaIndex: As always, I recommend checking the official LlamaIndex documentation, which is frequently updated.

Let me give you a quick overview of the documentation and resources available for LlamaIndex users.

On the "Home" page, users can find high-level concepts, installation and setup instructions, starter examples, video series, FAQs, and starter tools.

The "Learn" section covers topics such as using LLMs, loading and ingestion, indexing and embedding, storing, querying, tracing and debugging, evaluating, and integrating all components.

The "Examples" section includes practical applications and tutorials on agents, callbacks, chat engines, cookbooks, customization, data connectors, and more.

These resources will help you easily navigate and utilize the LlamaIndex framework for context-augmented LLM applications.

**Slide 11: LlamaIndex’s simple use-case**

Let’s see a simple use case for LlamaIndex, demonstrating how to set up a query engine using various tools and libraries.

The code snippet begins by importing necessary modules such as load\_dotenv from dotenv and VectorStoreIndex, SimpleDirectoryReader from llama\_index.

It starts by loading environment variables from a .env file using load\_dotenv()

The data source path is defined, pointing to a directory named ./datasource.

The create\_query\_engine function is then defined, which loads documents from the specified directory using SimpleDirectoryReader and creates an index from these documents with VectorStoreIndex.

The index is persisted for future use, and a query engine is created from the index.

Then the function returns query\_engine

Next, we execute this function with the DataSource\_DIR as argument to obtain the query\_engine

An example QA scenario is provided, where a question about the geography of Paris is posed, and by calling the method query of query\_engine on the question query\_engine.query(question), the response is returned.

This setup highlights the simple process of using LlamaIndex for querying and indexing documents efficiently.

**Slide 12: Open-source vs Proprietary LLMs**

Now let’s talk about Open-source and proprietary LLMs

Let’s see what the difference between open-source and proprietary large language models (LLMs) are.

Open-source LLMs are shared publicly on platforms like Hugging Face Hub and GitHub, either with source code included or as open-weights only. These models come with appropriate licenses for research and commercial usage. Examples include Llama 2 and 3 from Meta AI, Mistral, and OLMo.

In contrast, proprietary LLMs do not share weights or source code publicly and require payment for use. Examples of proprietary LLMs include OpenAI's GPT-4 and GPT-4o, and Anthropic's Claude 3.5.

**Slide 13: Usage Examples**

Next, let’s examine some code snippets for utilizing both open-source and proprietary LLMs using frameworks like Hugging Face and LangChain.

For the open-source LLM, we use the HuggingFace library to import an AutoTokenizer and pipeline, and then we use the LangChain library to import huggingface\_pipeline. The model used is "meta-llama/Llama-2-7b-chat-hf." The tokenizer is loaded from the pre-trained model.

After that, we create text\_pipeline using the pipeline class by specifying the ‘text-generation’ task, model name, and tokenizer. Next, the HuggingFacePipeline is created with the text generation pipeline and model keyword arguments, such as temperature.

For the proprietary LLM, such as that of OpenAI, we first need to load the OpenAI API key stored in environment variables. Then we import the ChatOpenAI class from LangChain. We initialize the ChatOpenAI model using the model name like "gpt-4o" and parameters like temperature.

These examples illustrate the practical steps to implement and interact with both open-source and proprietary LLMs in the coding environment.

**Slide 14: Vector Embedding**

Next, let’s talk about Vector Embedding

Vector embedding is a crucial process in NLP applications, transforming text into numerical vectors that capture semantic meaning.

This section will answer key questions related to understanding and utilizing vector embeddings in natural language processing:

1. **Why vector embedding?**
2. **What is the vector embedding process?**
3. **How does a pre-trained embedding model capture the semantic meaning of words?**
4. **How to choose an embedding model?**

**Slide 15: Why Vector Embedding?**

Let’s first discuss why we need vector embeddings.

As you may know, raw text data cannot be directly processed by LLMs. Instead, text data must be converted into numerical vectors through a process called embedding.

These vectors represent the text data in a numerical form that LLMs can understand and work with.

By transforming text into numerical vectors, embeddings allow LLMs to effectively analyze and generate human-like text based on the provided inputs, empowering various NLP tasks like text classification, text generation, translation, question-answering, and summarization.

**Slide 16: What is The Vector Embedding Process**

Now let’s see how vector embeddings work.

The vector embedding process involves creating a numerical representation of words, sentences, or even entire documents, capturing the semantic and syntactic meanings between words. This process enables algorithms to comprehend the contextual meaning and reason with the text data.

The inputs to this process can be individual words, sentences, or entire documents.

The outputs are high-dimensional vectors, which are sequences of continuous values representing the text in a numerical form.

For example, here how the word "cool" is transformed into a high-dimensional vector through an embedding model.

This process is fundamental for enabling NLP models to understand and work with text data effectively.

**Slide 17: Pre-trained Embedding Models**

Now, let’s discuss another fundamental question “How does pre-trained embedding models, like Word2Vec, GloVe, or BERT, capture semantic meaning of words”

A pre-trained embedding model is trained on a large corpus of text. During training, the model assigns vectors to words or sequences of words. These vectors are adjusted so that they reflect the semantic similarity and context of the words. This means that similar words, such as synonyms or words used in similar contexts, will have vectors that are close to each other in the vector space.

The model adjusts its weights to capture various aspects of meaning, such as synonyms, context, analogy, gender, sentiment, and more. For example, the phrases "canine companions say," "feline friends say," and "bovine buddies say" would be converted into vectors that reflect their semantic relationships and contexts. Their vectors should be close to each other and far from the vector representing the phrase "a quarterback throws a football.

The goal is to minimize the distance between vectors of semantically similar words and maximize the distance between those that are different. This process allows the model to capture the nuanced meanings of words and use this understanding for various natural language processing tasks.

**Slide 18: Choosing an Embedding Model**

Now, let’s discuss the question “How to choose an embedding model”

Selecting an appropriate embedding model depends on various factors, including model type, downstream tasks, and language requirements.

For model types, which include open-sourced embedding models hosted on platforms like Hugging Face, and proprietary embedding models accessible via APIs, such as OpenAI's Embedding models.

The benefit of Open-sourced models is that they can also be fine-tuned on domain-specific data.

Another criterion is the downstream tasks that the model will support, such as classification, retrieval, summarization, and more.

Finally, selecting embedding models by language requirements, such as English, French, and others.

The provided table from OpenAI's embedding model guides shows examples of embedding models and their use cases, emphasizing the importance of aligning the model choice with specific application needs.

**Slide 18: Choosing an Embedding Model - MTEB**

What is the tool that helps in selecting the most suitable embedding model based on various criteria? The answer is the Massive Text Embedding Benchmark (MTEB) leaderboard in Hugging Face Space.

The MTEB leaderboard allows users to filter models by type, including open-source, proprietary, sentence transformers, cross-encoders, and bi-encoders.

Additionally, users can filter by model size, ranging from less than 100 million to over 1 billion parameters.

The leaderboard provides performance metrics across multiple tasks, such as bitext mining, classification, clustering, pair classification, reranking, retrieval, and summarization, among others.

We can also filter the embedding by language, although it’s quite limit now with just 4 language tabs for English, Chinese, French, Polish. I believe that we will have more non-English tab soon.

It displays detailed statistics for each model, including memory usage, embedding dimensions, maximum tokens, and average performance scores across different datasets and tasks.

This comprehensive tool helps users make informed decisions by comparing models based on their specific requirements and performance benchmarks.

The MTEB leaderboard is an invaluable resource for selecting the best embedding model for various natural language processing tasks.

**Slide 19: Vector Database**

Now let’s talk about Vector Databases. In this section, we will discuss what a vector database is and then examine the landscape of vector database providers.

**Slide 19: What is a vector Database**

Let’s see what a vector database is.

It is a database that is designed to store high-dimensional vectors and perform operations such as similarity or semantic search.

These vectors are mathematical representations of features or attributes derived from raw data, such as text, and are obtained through an embedding process.

As demonstrated here documents are converted into vectors using an embedding model and then stored in a vector database.

This allows for efficient retrieval and analysis based on the semantic content of the data.

**Slide 20: Vector Database Provider Landscape**

Now let’s have a look at various vector database providers, categorized by their features and availability.

1. **The first group is Open Source Dedicated Vector Databases**: These databases are specifically designed for storing and querying high-dimensional vectors and are available under open-source licenses like Apache 2.0 or MIT. Examples include Chroma, Vespa, LanceDB, Marqo, Qdrant, and Milvus.
2. **The second group is Commercial or Source Available Dedicated Vector Databases**: These are dedicated vector databases that are available either commercially or with source availability. Examples include Pinecone and Weaviate.
3. **The third group is Open Source Databases that Support Vector Search**: These are general-purpose databases that support vector search functionalities and are available as open-source software. Examples include OpenSearch, ClickHouse, PostgreSQL, and Cassandra.
4. **And the last group is Commercial Databases that Support Vector Search**: These are general-purpose databases with vector search capabilities, available commercially. Examples include Elasticsearch, Redis, Rockset, and SingleStore.

This comprehensive overview helps in understanding the different options available for vector databases, whether you are looking for open-source solutions or commercial offerings.

**Slide 20: Vector Embedding & Vector store example**

Let’s examine two small code snippets for creating a vector database using FAISS, and Chroma, along with the support tools like LangChain, OpenAIEmbeddings

The process begins by importing necessary modules, including load\_dotenv from dotenv, and components from langchain.vectorstores and langchain.embeddings.

Firstly, the environment variables, such as the OPENAI\_API\_KEY, are loaded using load\_dotenv().

For creating a Chroma vector store:

* Import Chroma from langchain.vectorstores.chroma.
* Use the Chroma.from\_documents() method to create a vector store from a collection of text documents. The OpenAIEmbeddings class is used to embed these text chunks.

For creating a FAISS vector store, the process is like the Chroma case.

* Import FAISS from langchain.vectorstores.faiss.
* Use the FAISS.from\_documents() method to create a FAISS vector store from the same text documents, again utilizing OpenAIEmbeddings for embedding.

Both Chroma and FAISS are powerful tools for managing vector data, enabling efficient storage and retrieval of embeddings.

This setup facilitates the creation of vector databases for various applications involving text embedding and similarity search.

**Slide 21: Prompt Engineering**

Let’s talk about Prompt Engineering. In this section, we will discuss one of the techniques that helps LLMs generate desired responses.

We will cover four main points: What are prompts and prompt engineering for LLMs? Prompt components, basic prompt engineering, and advanced prompt engineering.

**Slide 22: Prompt vs Prompt Engineering**

Let’s talk about prompts and prompt Engineering

 **Prompts**: These are sets of instructions designed to guide LLMs in generating desired responses. By providing specific inputs, prompts help shape the output produced by the model.

 **Prompt Engineering**: This involves various techniques and strategies to optimize prompts, ensuring that the LLMs produce the most relevant and accurate responses.

**Slide 23: Prompt Components**

Let’s break down the key elements that make up an effective prompt for guiding Large Language Models (LLMs):

* **Context**: The first element is context. This provides the background information necessary for the model to understand the user's request. The context can be as simple as a single sentence or as complex as a detailed paragraph.
* **Instruction**: The second element is Instruction. This specifies how the model should respond to the user's input, such as providing a step-by-step response or summarizing the information at the end.
* **Optional Parameters / Constraints**: Thirdly, optional parameters or constraints. These include additional guidelines that can refine the response, such as the desired format (e.g., JSON, dictionary), tone, or length of the response.
* **User Query / Questions**: Another element is user query or questions. These are the specific queries or questions that the user wants the model to address in the response.

By incorporating these components into prompts, users can effectively guide LLMs to generate accurate and relevant responses tailored to their needs.

**Slide 24: Prompt Example**

Now let’s examine a practical illustration of how to construct a prompt for guiding an LLM, such as ChatGPT GPT-3.5, to generate a specific output. In this example,

* **Context**: "You are a marketing manager for a new product launch." This sets the stage and provides the model with the necessary background information to understand the scenario.
* **Instruction**: "Write a short paragraph to promote the new product on a social media platform." This clearly directs the model on what type of response is expected.
* **Constraint**: "Your response should be no more than 200 words and should focus on unique features of the product." This provides specific guidelines to ensure the response is concise and focused on key aspects.
* **Query**: "The new product is organic and natural face care." This specifies the main topic or subject matter that the response should address.

Using these components, the prompt guides the LLM to generate a promotional paragraph that effectively highlights the unique features of an organic and natural face care product, suitable for posting on social media. This structured approach ensures that the generated content meets the desired criteria and effectively communicates the intended message.

**Slide 25: Basic Prompt Engineering**

Let’s discuss the basic prompt engineering principles. Firstly, I would like to emphasize the importance of providing clear and specific instructions when creating prompts for LLMs. To do this, we can leverage following basic techniques:

* **Use Delimiters**: First, use delimiters. Clearly mark the beginning and end of instructions using delimiters such as triple quotes """ """ or triple backticks to avoid any ambiguity.
* **Specific Structured Output**: Secondly, Specify the format in which the response should be structured, such as JSON, HTML, etc., to ensure the output meets the desired requirements.
* **Provide Contextual Instructions**: Thirdly, include relevant context, assumptions, or conditions that help the model understand the background and generate appropriate responses.
* **Give Examples**: Moreover, use one-shot or few-shot prompting to provide examples that illustrate the expected response, helping the model to learn the pattern and generate similar outputs.
* **Role**: Furthermore, assign a specific role to the model, which can guide its responses according to the scenario. For instance, specifying "You are a customer support agent" can influence the model's tone and content.

By following these basic prompt engineering principles, we can create more effective and precise prompts, leading to better performance and more accurate responses from LLMs.

**Slide 26: Example: Use Delimiters**

Let’s see an example of how to use delimiters in a prompt to clearly define the boundaries of instructions and content.

The example prompt template is designed to request a short and concise summary of a given text;

As you can see, the triple quotes """ are used to enclose the entire prompt template, marking the beginning and end of the instructions. Within the template, triple backticks are used to clearly indicate the start and end of the text that needs to be summarized.

**Slide 27: Examples: Giving Specific Instructions**

Now let’s contrast the effects of general versus specific and precise prompts in guiding LLM responses.

* **General Prompt**: "Write a paragraph about the benefits of exercise."
  + This prompt is broad and results in a general response: "Regular exercise enhances overall well-being by improving cardiovascular health, boosting mood, and maintaining a healthy weight."
* **Specific and Precise Prompt**: "Compose a concise comparison between the advantages of aerobic exercise and strength training."
  + This prompt is more detailed and leads to a more focused response: "Aerobic exercise promotes cardiovascular health through increased oxygen consumption, while strength training builds muscle mass, aiding metabolism for efficient weight management."

Here we demonstrate how providing clear and specific instructions can significantly refine the output, making it more relevant and targeted to our needs. By being explicit in what you ask, you can guide LLMs to generate more precise and useful content.

**Slide 28: Example: Structured Output**

Now let’s compare the results of a prompt with general response structure with one with structured response in guiding LLM responses.

* **General-response Prompt**: "Given a customer review: 'I love this product'. Determine the sentiment and give a 20-word email response."
  + This prompt results in a straightforward response: "Sentiment: Positive Email Response: 'Thank you for your kind words! We're thrilled to hear that you love our product. Your satisfaction means the world to us.'"
* **Structured-response Prompt**: "Given a customer review: 'I love this product'. Determine the sentiment and give a 20-word email response. Return your response in JSON object with two keys: sentiment and response email."
  + This prompt results in a more structured and formatted response:

Here we demonstrate how specifying the desired output format, such as JSON, can lead to more organized and easily interpretable results. Structured prompts help ensure the response not only meets the content requirements but also adheres to a specific format, making it more useful for further processing or integration into other systems.

**Slide 29: Example: Few-shot Prompting**

Well, let’s talk about few-shot prompting. Few-shot prompting involves giving the model a small number of example inputs and their corresponding outputs, which helps it understand the pattern or task it needs to perform.

In this example, the model uses these examples to learn how to predict the sentiment of the user's statements.

By observing the patterns in the provided examples, the model can make informed predictions for new, similar inputs. In this case, the few-shot examples help the model determine that the sentiment for the statement "That player is phenomenal" should be "Positive."

Few-shot prompting is a powerful technique to improve the performance of LLMs in various tasks by providing context and examples that clarify what is expected.

**Slide 30: Example: Role Prompting**

Let’s discuss Role prompting technique. Assigning specific roles to an LLM can help generate responses tailored to different scenarios. Here are several examples

By specifying these roles, the LLM can adopt the appropriate tone, style, and content relevant to each scenario, producing more accurate and contextually appropriate responses. This technique helps in generating diverse and specialized outputs based on the role assigned to the model.

**Slide 31: Advanced Prompt Engineering**

Now let’s look at more advanced techniques for optimizing prompts to enhance the performance of Large Language Models (LLMs).

* **Chain of Thought (CoT)**:
  + **Purpose**: Provides intermediate reasoning steps to guide the model's responses.
  + **Example**: Using prompts like "Let's think step by step" to break down the reasoning process.
  + **Method**: A series of manual demonstrations, each composed of a question and a reasoning chain that leads to an answer.
* **Retrieval Augmented Generation (RAG)**:
  + **Purpose**: Incorporates up-to-date external knowledge into the model's input.
  + **Benefits**: Helps in reducing hallucinations by grounding the model's responses in accurate information.
* **Other Methods**:
  + **Self-consistency**: Ensures that the model's outputs are consistent across different prompts or iterations.
  + **Generated Knowledge**: Uses the model's previous outputs as context or knowledge for generating new responses.
  + **Least-to-most Prompting**: Starts with minimal prompts and gradually adds more context or detail as needed.
  + **Tree of Thoughts**: Explores different branches of reasoning or thought processes to arrive at an answer.
  + **Graph of Thoughts**: Utilizes a network of interconnected ideas or concepts to guide the model's responses.

These advanced techniques aim to improve the quality and relevance of the outputs generated by LLMs by leveraging structured reasoning, external knowledge, and iterative refinement.

.

**Slide 32: Example: RAG Prompting**

Now let’s see an example of how to structure a prompt for Retrieval Augmented Generation (RAG). This technique integrates external knowledge into the model's input to enhance its responses.

 **Purpose**: This template sets up the model to function as an assistant for question-answering tasks, using specific context retrieved from external sources.

 **Instructions**:

* The model is instructed to use the provided context to formulate its answer.
* It is also directed to acknowledge when it doesn't know the answer, ensuring honesty and reducing hallucinations.

Variables:

* {question}: Placeholder for the question being asked.
* {context}: Placeholder for the relevant context retrieved from external sources.

By following this template, the model can provide more accurate and contextually relevant answers, leveraging up-to-date external knowledge. This method helps improve the reliability and quality of the responses generated by the model.

**Slide 33: Frontend Frameworks for LLM-based Applications**

Well, welcome to the next session: Frontend Frameworks for LLM-based Web Applications.

**Slide 34: Frontend Frameworks**

Let’s discuss several frameworks that can be used for developing web applications. I categorize these frameworks based on the programming language used:

* **For web applications with JavaScript**:
  + **React**: A popular JavaScript library for building user interfaces, particularly single-page applications.
  + **Vue.js**: A progressive JavaScript framework used for building user interfaces and single-page applications.
* **For web applications with Python**:
  + **Streamlit**: An open-source Python library that makes it easy to create and share custom web apps for machine learning and data science.
  + **Gradio**: A Python library that allows you to quickly create user-friendly web interfaces for your machine learning models.

These frameworks offer different approaches and tools to build robust, interactive, and user-friendly web applications, catering to both JavaScript and Python developers.

In the frame of this course, I will just introduce Streamlit and Gradio.

**Slide 35: Streamlit**

Let’s first take a look at Streamlit.

As always, I recommend you checking the Streamlit documentation page from [Streamlit.io](https://streamlit.io/)

It provides an overview of the documentation and resources available for using Streamlit, a popular open-source Python framework for creating dynamic data apps.

The Streamlit documentation is structured to guide users through every stage of app development.

The Get Started section offers installation guides, fundamental concepts, and initial steps to help new users begin using Streamlit effectively.

The Develop section provides detailed explanations of core concepts, a comprehensive API reference, step-by-step tutorials, and quick reference materials for efficient app building.

In the Deploy section, users can find information on deploying apps, utilizing the free Streamlit Community Cloud, integrating with Snowflake for enterprise solutions, and exploring other deployment platforms.

The Knowledge Base addresses frequently asked questions, dependency management, and troubleshooting deployment issues.

In summary, Streamlit documentation is organized to support users in every stage of their app development journey, from setup and installation to development and deployment.

**Slide 36: Streamlit Components**

Another interesting feature of Streamlit is the concept of Streamlit components, which are third-party modules that enhance the capabilities of Streamlit applications.

These components are created by the community to extend what’s possible with Streamlit, offering additional functionality and customizations. For each component, we will have a dedicated page that includes information about the creators, installation instructions, and either a demo or source code to inspire you to integrate it into your application.

Categories listed on the left side include options such as LLMs, Widgets, Charts, Authentication, Connections, Images & video, Audio, Text, Maps, and Dataframes, suggesting a wide variety of extensions available to enhance Streamlit applications.

Overall, the Streamlit Components gallery showcases the flexibility and extensibility of Streamlit, empowering users to add diverse features to their apps through community-contributed modules.

**Slide 37: Streamlit Gallery**

Now let’s take a look at the Streamlit Gallery page. This is a collection of example apps built with Streamlit, categorized to help users explore and find inspiration. The gallery includes for example:

* **Favorites**: Highlighting popular and widely used apps, such as:
  + **Streamlit cheat sheet** by daniellewisdl with 203K views.
  + **Streamlit extras** by arnaudmiribel with 110K views.
  + **Roadmap** by streamlit with 103K views.
* **Other Categories**: Allowing users to browse by different themes such as Trending, LLMs (Language Learning Models), Snowflake powered, Data visualization, Geography & society, Sports & fun, Science & technology, NLP & language, Finance & business, and Other.

To start leveraging Streamlit quickly, let’s take a closer look at Streamlit cheat sheet app, created by Streamlit user and published on the Streamlit Gallery page.

**Slide 38: Streamlit Cheat Sheet**

This is a comprehensive cheat sheet for Streamlit

The cheat sheet is divided into several key sections to help you quickly get started and optimize your use of Streamlit.

On the left, you’ll find installation instructions and basic import commands.

The middle section covers how to display text, data, and media, as well as how to use interactive widgets like buttons, sliders, and file uploaders.

On the right, we have sections on connecting to data sources and optimizing performance, including caching strategies, and many more if you scroll-down.

This cheat sheet is a handy reference for both beginners and advanced users, providing essential code snippets and tips to streamline your development process.

**Slide 38: Example: Show Logos and Headers in Streamlit**

In this example, we will learn how to display logos and headers in a Streamlit application.

The code snippet demonstrates setting up a Streamlit app with a custom page configuration, including a title and a logo icon.

The st.set\_page\_config function configures the page title and icon, while st.columns is used to create the columns with specified widths and gaps.

It uses a three-column layout to showcase different elements: the product logo, the header text, and the company logo.

Each column (c1, c2, and c3) contains either an image or markdown text, ensuring that the logos are displayed with appropriate widths and the header text is formatted using HTML for enhanced styling.

This setup provides a visually appealing way to integrate branding elements into a Streamlit application.

**Slide 38: Gradio**

Now let’s talk about Gradio.

Gradio is another Python framework for creating web applications, focusing on simplicity and accessibility.

Gradio, a platform designed for building and sharing machine learning applications with a user-friendly web interface.

In the homepage of Gradio, we can access quick start guides, comprehensive documentation, a playground for experimentation, and custom components to extend functionality.

Gradio allows users to easily demo their machine learning models so that anyone can use them, anywhere.

The platform emphasizes ease of use and accessibility, making it the fastest way to create and share delightful machine learning apps.

Users can get started with Gradio, explore example apps like "Hello World," "Airbnb Map," "Chatbot Streaming," and "Diffusion Faces," and access various community resources and support.

One interesting thing about Gradio is that it has been acquired by Hugging Face and is now an element in the Hugging Face ecosystem. You can easily prototype your ML demo with Gradio and then push it to Hugging Face Spaces to share it with everyone.

**Slide 39: Gradio's Documentation**

As always I would recommend you checking out Gradio’s documentation page.

It will give you an overview of the resources available for building and sharing machine learning demos and web applications using Gradio.

The documentation is divided into several key sections:

* **Gradio**: Core library for creating ML demos and web apps with various components like Interface, Blocks, ChatInterface, Textbox, Image, Audio, and Dataframe.
* **Python Client**: Tools for making programmatic requests to Gradio applications from Python environments.
* **JavaScript Client**: Tools for making programmatic requests to Gradio applications from JavaScript (TypeScript) in the browser or server-side.
* **JavaScript Components**: Using Gradio's UI components in standalone JavaScript applications.
* **Gradio Lite**: Running Gradio's Python code serverless (entirely in your browser) using WebAssembly.
* **Custom Components**: Creating, using, and sharing custom components within a Gradio application.

This documentation provides comprehensive guides and tools to help developers effectively utilize Gradio for their machine learning and web application projects.

**Slide 39: Some simple demos with Gradio**

Now let’s see some simple demos with Gradio

**Blocks, Row, Image, HTML**

In this demo, we will learn how to use Gradio to create a custom header for an application.

The code snippet illustrates the process of setting up a Gradio interface with a soft theme using gr.Blocks.

Within this interface, various elements are added using a gr.Row() to organize them horizontally.

The elements include images for the product logo and company logo (gr.Image), and an HTML block for the header (gr.HTML).

This example showcases how to combine different Gradio components to build a cohesive and visually appealing user interface for machine learning applications.

**Column, Dropdown, Slider, Button**

In this example, we will learn how to use Gradio to create an interactive interface with a column layout, dropdown menu, slider, and button. The code snippet demonstrates how to set up these elements:

gr.Column(): Organizes the elements in a vertical column.

gr.Dropdown(): Creates a dropdown menu with options like "Pizza Margherita" and "Spaghetti Carbonara," labeled "Dish" with a default value.

gr.Slider(): Adds a slider to select a number within a specified range, labeled "Number of persons."

gr.Button(): Adds a button with the label "Tell me about dish's ingredient, recipe & interesting stories."

These components are arranged to create an intuitive user interface for selecting a dish, specifying the number of persons, and triggering an action to display more information.

**Textbox, Markdown blocks**

In this demo, we will learn how to create an interactive interface in Gradio with a textbox, button, and markdown component:

* **Textbox**: The gr.Textbox element is used to create an input field for users to enter a webpage URL. It includes parameters like placeholder, label, lines, and interactive.
* **Button**: The gr.Button element labeled "Pre-processing Webpage" allows users to trigger actions, such as processing the entered URL.
* **Markdown**: The gr.Markdown component is used to display dynamic status updates, such as confirming that a webpage has been successfully loaded.

These components are organized within a column layout to create a cohesive user interface for entering and processing webpage URLs.

**Textbox blocks for multi-inputs as texts**

In this demo, we will learn how to create multiple textbox components in Gradio to gather different types of input:

* **Product Textbox**: A gr.Textbox for entering the product name, labeled "Product" with a placeholder "Enter product".
* **Features Textbox**: Another gr.Textbox for listing the product features, labeled "Features" with a placeholder "Enter features".
* **Target Clients Textbox**: A gr.Textbox for specifying the target clients, labeled "Target clients" with a placeholder "Enter target clients".

These textboxes are organized within a column layout to create a user-friendly form for inputting details about a product, its features, and the target audience.

**Textbox blocks for displaying responses**

In this example, we will learn how to create a textbox component in Gradio for displaying generated text responses. The gr.Textbox component is configured as follows:

* **Label**: "Generated text" to indicate the purpose of the textbox.
* **Placeholder**: "Generated text will appear here" to guide the user.
* **Lines**: Set to 10 to provide enough space for displaying longer text.
* **Value**: Initially empty.
* **Interactive**: Set to False to make the textbox read-only.

This setup allows the generated responses to be displayed neatly in a non-editable textbox, ensuring that users can view the content without altering it.

**Upload block**

In this demo, we will learn how to add a file upload component in Gradio. The gr.File component is set up to allow users to upload documents with the following configuration:

* **Label**: "Upload a document" to guide users on the action.
* **File Types**: Restricted to .pdf, .docx, and .txt to ensure only these formats are accepted.
* **Type**: Set to "filepath" to handle the uploaded files as file paths.

This setup provides a user-friendly interface for uploading documents, making it easy to integrate file upload functionality into your application.

**Interface block**

In this demo, we will learn the creation of an interface block using Gradio. The code sets up an input textbox and an output textbox, then creates a Gradio interface that connects these components to a function for processing the input text.

* **Input Textbox**:
  + lines=10: Allows up to 10 lines of text.
  + label="Enter the Text": Label for the textbox.
  + max\_lines=20: Maximum lines allowed for the input.
* **Output Textbox**:
  + lines=5: Displays 5 lines of text.
  + label="Summarization": Label for the output.
* **Gradio Interface**:
  + fn=prediction: Function to process the input and generate the output.
  + inputs=inputs: Connects the input textbox.
  + outputs=outputs: Connects the output textbox.
  + theme="soft": Applies a soft theme to the interface.
  + examples=[EXAMPLE\_TEXT]: Provides example text for demonstration.

This setup provides a user-friendly interface for text summarization, making it easy to input text and see the summarized output.

**Chat Interface block**

In this example, we will learn the setup of a Chat Interface block using Gradio. The code creates a chat interface, providing an intuitive way for users to interact with the model through text-based conversations.

* **Chat Interface Setup**:
  + gr.ChatInterface(): Initializes a chat interface in Gradio.
  + predict: The function to handle predictions based on user input.
  + examples: Provides predefined examples to guide users on possible queries.

The example queries such as "What is climate change?" and "What are the benefits of renewable energy?" illustrate the type of interactions users can have with the chat interface. This setup is ideal for creating interactive, conversational AI applications.

**Slide 40: In Summary**

In summary, what we have learned so far in this part.

We have introduced various components essential for building LLM-based web applications.

We covered LLM orchestration frameworks like LangChain and LlamaIndex, discussed the differences between open-source and proprietary LLMs, and explored concepts like vector embedding and vector databases.

We also explored prompt engineering

Furthermore, we have demonstrated how to use Python frameworks such as Streamlit and Gradio to quickly prototype and tool LLM applications.

Alongside these topics, we provided simple demos and source-code examples to illustrate the implementation process.