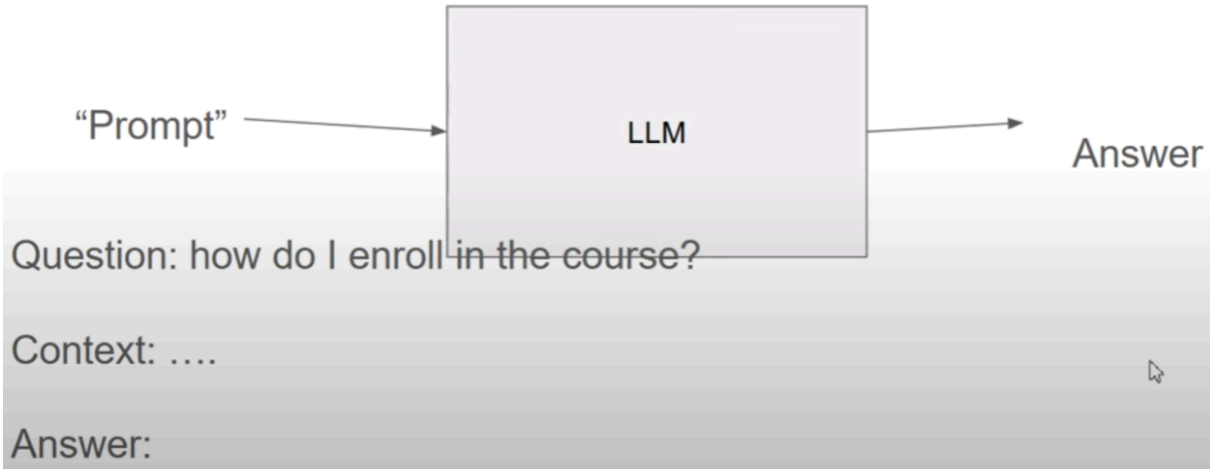
# LLM ZoomCamp

## Week 1

### 1. Introduction to LLM and RAG

#### 1.1)What is LLM

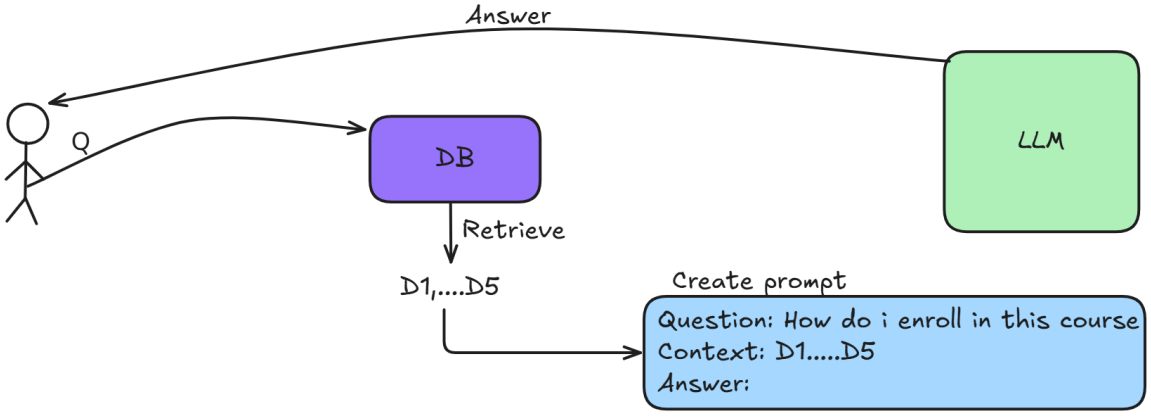
A **Large Language Model (LLM)** is an advanced AI model trained on vast amounts of text data to understand and generate human-like language,It uses deep learning, particularly transformer architectures, to perform tasks like translation, summarization, and conversation.




#### 1.2) What is RAG

it is a technique that retrieves relevant documents from a knowledge base and uses them to generate more accurate and informed responses.

**RAG Architecture:**



## 2. Configuring Your Environment

In this video, we will set up the environment required for the course. This includes installing necessary libraries and tools. checkout the video

## 3. Retrieval and Search

in this video we will have small introduction to RAG, where we will use MinSearch Library, for more details about it check video:

- Video: https://www.youtube.com/watch?v=nMrGK5QgPVE

then we will use OpenAI (DeepSeek in my case) to answer with provided context.

### 3.1) Documents Indexing

before we search we have to index documents, we have to load documents from the Json file, then we choose the field we want to index and finally we initialize the MinSearch and fit it with loaded json file

```
import minsearch
import json

with open('documents.json', 'rt') as f_in:
    docs_raw = json.load(f_in)

documents = []

for course_dict in docs_raw:
    for doc in course_dict['documents']:
        doc['course'] = course_dict['course']
        documents.append(doc)

index = minsearch.Index(
    text_fields=["question", "text", "section"],
    keyword_fields=["course"]
)

index.fit(documents)
```

where

- `text_fields` are fields containing the main textual content of the documents.
- `keyword_fields` are fields containing keywords for filtering, such as course names.

### 3.2) Search

after indexing we can start searching , we first define the query

```
query = 'the course has already started, can I still enroll?'
```

now we start searching:

```
results = index.search(
    query,
    filter_dict={'course': 'data-engineering-zoomcamp'},
    boost_dict={'question': 3.0, 'section': 0.5},
    num_results=5
)

print(results)
```

where

- `filter_dict` is used to filter the search results based on keyword fields.
- `boost_dict` is used to give higher importance to certain fields during the search.
- `num_results` specifies the number of top results to return.

The  search method returns a list of results, each containing the matched document and its relevance score.

```
for result in results:
    print(f"Document: {result['document']}, Score: {result['score']}")
```

## 4. Generating Answers with gpt-4o(DeepSeek-chat)

we will use OpenAi(DeepSeek in my case) to generate response based on the context we provided, we can use other LLMs including open source llms with

ollama

### 4.1 Build the prompts

A prompt is basically what you type in to ask the AI something — like a question, a request, or some instructions. The better and clearer your prompt is, the better the answer you'll get. So, learning how to write good prompts really helps you get the most out of these language models.

we need to build prompte to get an answer from our LLM, here is a way to build your prompte:

```
from openai import OpenAI

client = OpenAI()

def build_prompt(query, search_results):
    prompt_template = """
    You're a course teaching assistant. Answer the QUESTION based on the CO
    Use only the facts from the CONTEXT when answering the QUESTION.

    QUESTION: {question}

    CONTEXT:
    {context}
    """.strip()

    context = ""
    for doc in search_results:
        context += f"section: {doc['section']}\nquestion: {doc['question']}\nansw

    prompt = prompt_template.format(question=query, context=context).strip()
    return prompt
```

First, we import the OpenAI library so we can communicate with their language model.

Then, we define a function called `build_prompt`, which takes two inputs: `query` (the user's question) and `search_results` (a list of documents retrieved from a search).

Inside this function, we use a `prompt_template` to define how our final prompt should look. To build the prompt, we start with an empty string called `context`.

We loop through each document in the search results, extract key parts like the section, question, and answer, then format and add that info to the `context`.

Finally, we insert both the `query` and the full `context` into the template using Python's `format` method. This gives us the complete prompt, ready to be sent to the OpenAI model for generating a response.

### 4.2. Generate The Answer

after having the prompte, now we can use it to get response from OpenAI

```
def llm(prompt):
    response = client.chat.completions.create(
        model='gpt-4o',
        messages=[{"role": "user", "content": prompt}]
    )
    return response.choices[0].message.content
```

we can test or function this way

```
def rag(query):
    search_results = search(query)
    prompt = build_prompt(query, search_results)
    answer = llm(prompt)
    return answer

rag('the course has already started, can I still enroll?')
```

## 5. **The RAG Flow Cleaning and Modularizing Code**

the video is just about cleaning the code

# 6. Searching with Elasticsearch

In this chapter, we will replace the custom search engine with Elasticsearch for more robust search capabilities. We will set up Elasticsearch, index the documents, and perform searches.

### 6.1. Setting Up Elasticsearch

```
pip install elasticsearch
```

### 6.2.Running Elasticsearch with Docker

```
docker run -it \
    --rm \
    --name elasticsearch \
    -p 9200:9200 \
    -p 9300:9300 \
    -e "discovery.type=single-node" \
    -e "xpack.security.enabled=false" \
    docker.elastic.co/elasticsearch/elasticsearch:8.4.3
```

use `-m 4GB \` flag to specify the memory limit for the Docker container.

Use the GET /_cluster/health endpoint to get detailed information about the cluster health, including the status, number of nodes, and number of active shards and the cluster information by running:

```
curl -X GET "localhost:9200/_cluster/health?pretty"
curl localhost:9200
```

### 6.3. Remove the Existing Container

If you get conflict errors, you might need to remove (or rename) that container to be able to reuse that name

```
docker stop elasticsearch
docker rm elasticsearch
```

### 6.4.Indexing Documents

To index the documents in Elasticsearch, we need to define the index settings and mappings, and then index each document.

### 6.5.Index Settings and Mappings

First, create an instance of the Elasticsearch client on Jupyter, which connects to the Elasticsearch server. Index settings include configurations such as the

number of shards and replicas. Shards help distribute the data and load across the cluster, while replicas provide redundancy for fault tolerance.

We are interested in Mappings that define the structure of the documents within an index. Each field in the documents can have specific data types and properties.

```
from elasticsearch import Elasticsearch

es_client = Elasticsearch('http://localhost:9200')

index_settings = {
    "settings": {
        "number_of_shards": 1,
        "number_of_replicas": 0
    },
    "mappings": {
        "properties": {
            "text": {"type": "text"},
            "section": {"type": "text"},
            "question": {"type": "text"},
            "course": {"type": "keyword"}
        }
    }
}

index_name = "course-questions"
es_client.indices.create(index=index_name, body=index_settings)

for doc in documents:
    es_client.index(index=index_name, document=doc)
```

We use the "keyword" type for a field like "course" when we want to filter documents based on the course name or run aggregations to count the number of documents per course.

### 6.6. Performing Searches

Once the documents are indexed, we can perform searches using Elasticsearch. This section explains how to construct and execute search

queries to retrieve relevant documents from the index.

**Search Query**

The following example demonstrates how to create a search query in Elasticsearch using the Python client.

```
query = 'I just discovered the course. Can I still join it?'

def elastic_search(query):
    search_query = {
        "size": 5,
        "query": {
            "bool": {
                "must": {
                    "multi_match": {
                        "query": query,
                        "fields": ["question^3", "text", "section"],
                        "type": "best_fields"
                    }
                },
                "filter": {
                    "term": {
                        "course": "data-engineering-zoomcamp"
                    }
                }
            }
        }
    }

    response = es_client.search(index=index_name, body=search_query)
    result_docs = [hit['_source'] for hit in response['hits']['hits']]
    return result_docs

search_results = elastic_search(query)
print(search_results)
```

The search query in Elasticsearch is composed of several key components that work together to retrieve relevant documents.

- The `size` parameter limits the number of search results to 5, ensuring that the output remains manageable and focused.
- The `multi_match` query searches across multiple fields—specifically `question`, `text`, and `section`—and boosts the relevance of the `question` field with a factor of 3, making matches in this field more significant.
- The `query` option holds the user's search input, while the `fields` option lists the fields to search in, applying boosting as defined.
- The `type` parameter of the `multi_match` query is set to `best_fields`, which finds the single best matching field.
- A `filter` is applied to restrict the search results to documents where the `course` field equals `data-engineering-zoomcamp`.

This combination of components ensures that the search results are both relevant and precise, with a specific focus on certain fields and criteria.

### 6.7.Integrating Elasticsearch with OpenAI API

Finally, we integrate Elasticsearch with the OpenAI API to generate answers based on the search results.

```
search_results = elastic_search(query)
prompt = build_prompt(query, search_results)
answer = llm(prompt)
print(answer)
```