Generative AI, Vector Databases, and Retrieval-Augmented Generation (RAG)

Large language models (LLMs)

LLMs are machine learning models that have been trained on large quantities of unlabeled text using self-supervised learning and can perform a variety of natural language processing tasks.

Of course, ChatGPT and Gemini are examples of a generative AI chatbot developed using underlying LLMs.

Generative AI Application Use Cases

Document question/answering; chatbots
Analyzing structured data
Generative question answering
Retrieval-augmented generation (RAG)
Summarization
Personal assistants
Interacting with APIs
Extraction
Code generation

You need to develop AI-powered applications where large language models interact with other large language models, applications and tools. How do you simplify the creation of these applications?

Generative AI Models

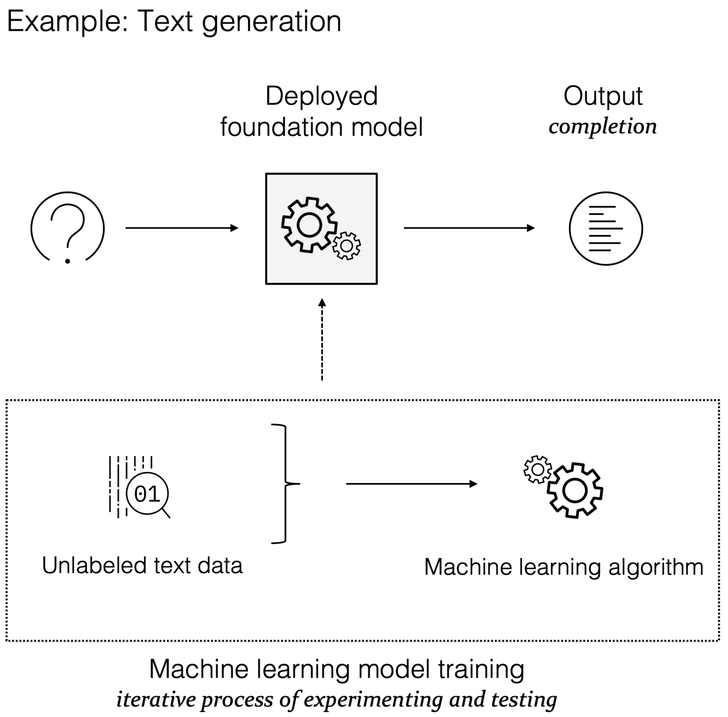
Generative models create new data (completions) in response to input requests (prompts).

Common use cases include:

Text generation: generate new text in response to a prompt, summarizing text, or writing a lengthy essay

Code generation: generate computer code based on a textual description of the proposed program

Image generation: create images based on the prompt



AI Developer Studios, SDKs, and APIs

There are a large number of companies who have some type of AI studio offering, provide an SDK and a general API; most are not free

A large consideration in AI application development become cost

There are a few free resources that we will take a look at and advantage of in this course: Google AI Studio and IBM watsonx.ai

Ollama makes it easy to run LLMs locally on your own laptop or server

A Few Limitations of LLMS

Prone to generation of misinformation ("hallucinations")

"Black box" with lack of transparency

Stale memory, unaware of new data and facts

Lack domain-specific knowledge and terminology

Potential to amplify biases found in training data

Need to be mindful of these limitations when deploying applications

Examples of 2 common issues

Lack of information source

“The bank offers 2.5% interest on accounts with a balance over $20,000.00.”

This sounds great – but where did the information come from?

How can a user verify that this is true? Where is it documented?

Outdated information:“Who is the highest-scoring player in the NBA?”The Llama-3-405b-instruct model returns:

”Kareem Abdul-Jabbar holds the record for the most points scored in the NBA with a total of 38,387 points.”

This is an outdated answer as Lebron James broke that record in 2023.

This means that llama-3-405b-instruct was trained on pre-2023 data.

This is now overcome in Chatgpt & Gemini through integration of internet search but still remains with base models

Retrieval Augmented Generation (RAG)

Grounding the model on additional sources of knowledge to supplement its internal representation of information

RAG addresses these issues:

Where did the LLM get its answer?

Is the answer based on updated material?

RAG does this by:

Working with “external data” (data not used for training the LLM):

Source of answers? From curated, validated, and accurate data

Currency of data? As current as the source

NO model retraining required

Retrieval-augmented generation components

Knowledge base

Can be any collection of information-containing artifacts, such as:

- Internal procedural wiki pages
- Files in GitHub (various formats)
- Messages in a collaboration tool
- Topics in product documentation
- PDF files
- Customer support tickets

Retriever

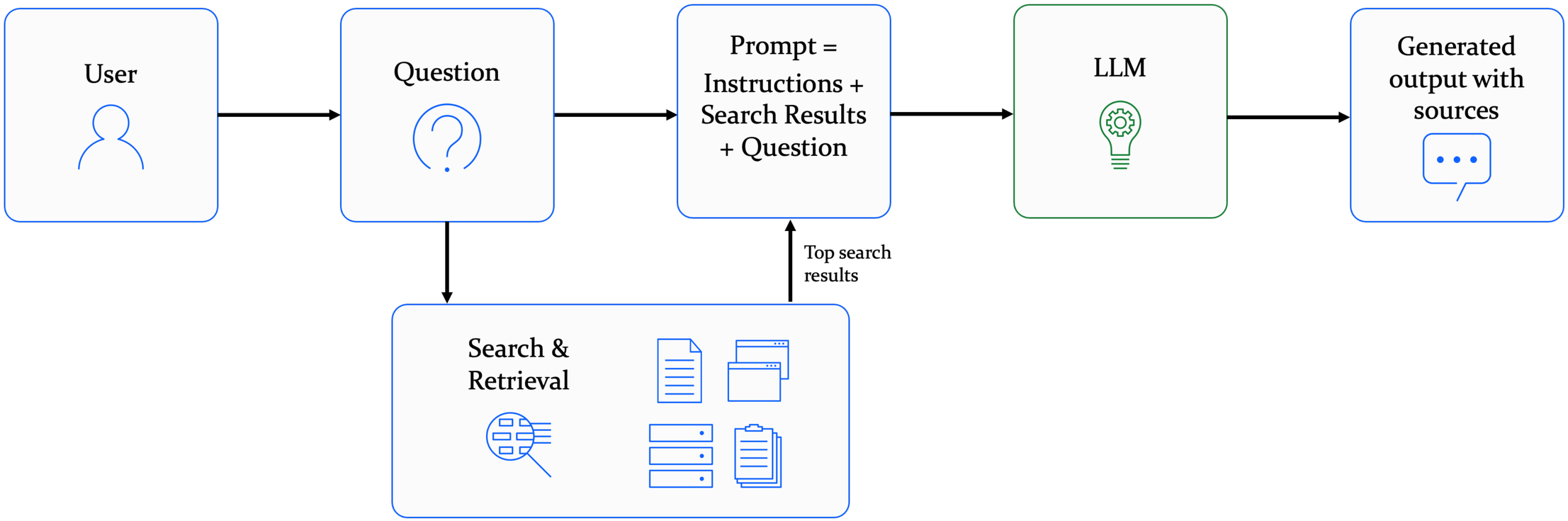
Can be any combination of search and content tools that reliably return relevant content from the knowledge base:

- Search and content APIs (e.g. GitHub APIs)
- Vector databases (such as Milvus and Chroma)

Generator

A generative LLM) that suits your use case, prompt format, and content being pulled in for context

Typical RAG process



RAG - Data Preparation

Validated, filtered, and curated data is stored. These can be files/data in Cloud Object Storage, Box, or other data repositories.

Clients can update them whenever necessary, ensuring that only the latest is used.

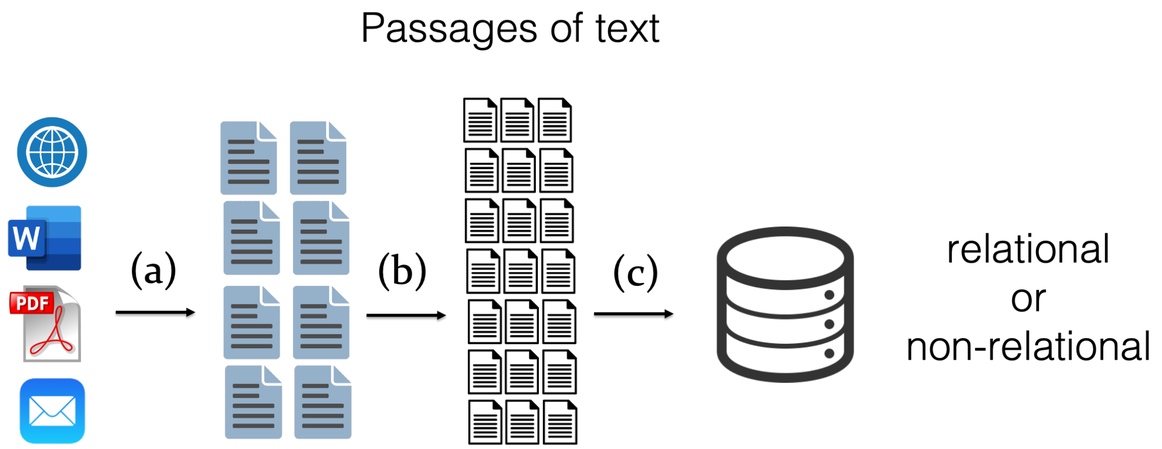
Traditional Data Storage

Data Ingestion

Original files to documents

Split documents into chunks

Store chunks to database



Python PDF Processing (an aside)

A large number of documents exist only as pdfs which could also constitute a large textual data source.

These are not always easy to deal with but there are a number of packages that provide data extraction that have varying fuctionalities and strengths

PyPdf2, PdfPlumber, PyMupdf, Docling to name a few

Docling

Vector Database Data Storage

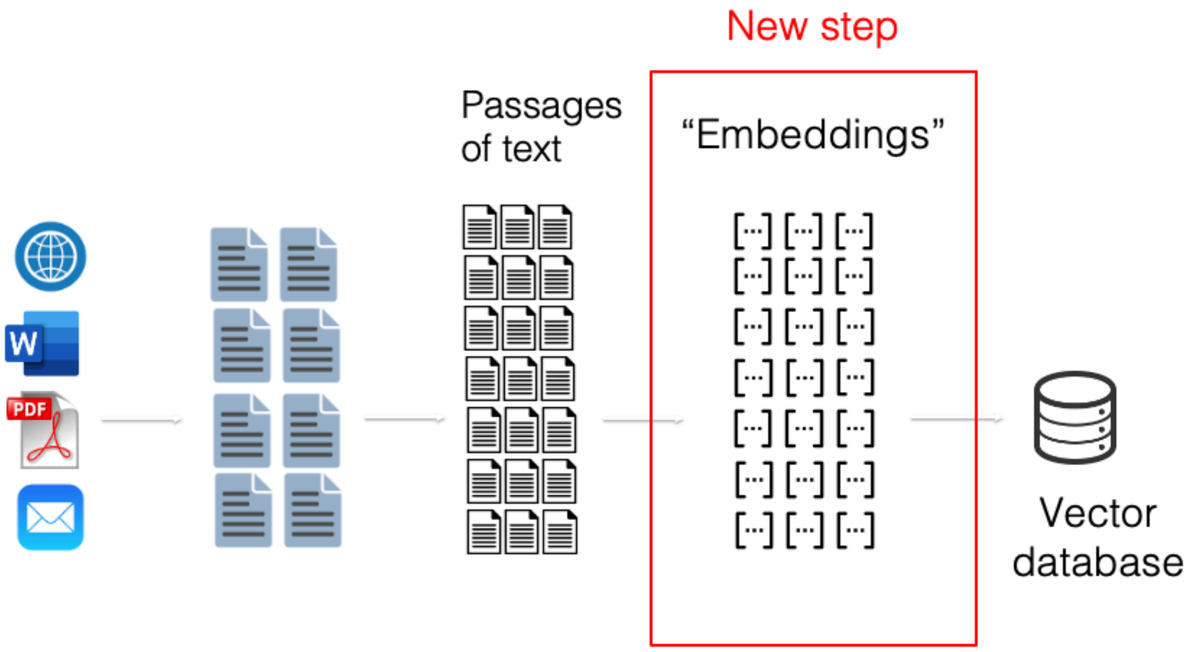
Data Ingestion

Original files to documents

Split documents into chunks

Chunks to embeddings

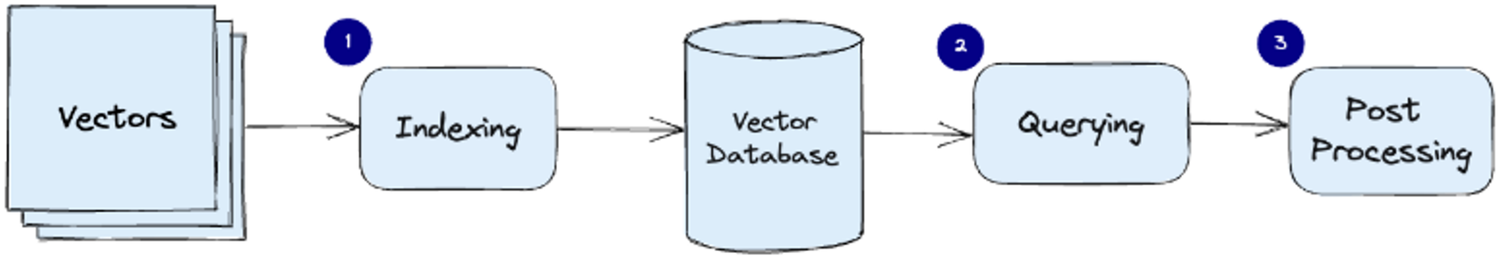
Store chunks to database



Vector Databases

- Commonly used as the back-end knowledge base storage for RAG
- Designed to efficiently store, index,and retrieve vector embeddings (mathematical representations of data)
- Very good at finding things that are similar to each other (similarity search)
- Examples of vector databases include Milvus, Chroma, Pinecone…

Vector Databases



Indexing

A process of mapping the vectors to a data structure that will enable faster search

Indexing helps:

- Avoid writing duplicated content into the vector store
- Avoid re-writing unchanged content
- Avoid re-computing embeddings over unchanged content

Indexing algorithm:

- Flat, Random Projection, Product Quantization, Locality-sensitive hashing

Vector DB

A A vector store takes care of storing embedded data and performing vector search

Similarity measures are mathematical methods for determining how similar two vectors are in a vector space

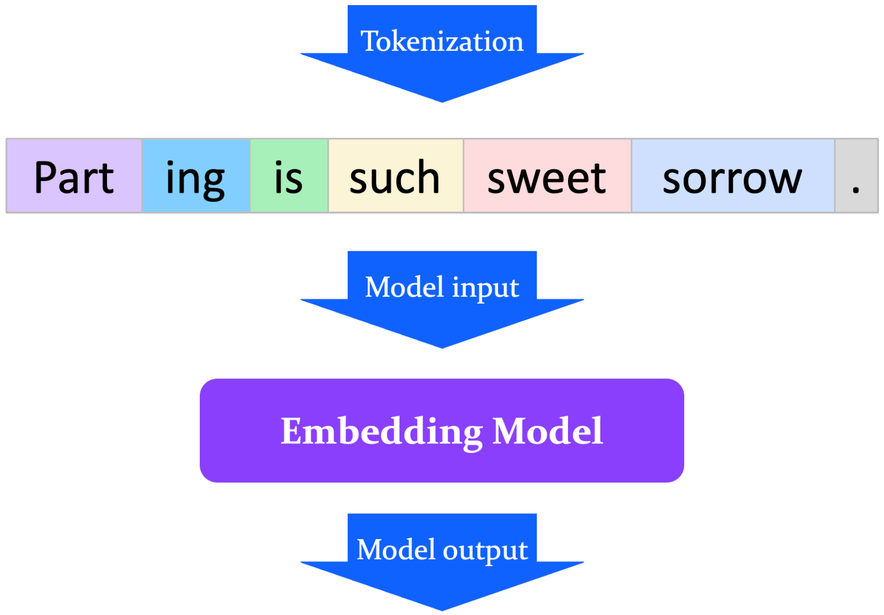
Similarity measures:

- Cosine similarity
- Euclidean distance
- dot product

Vector Embeddings

Input text

"Parting is such sweet sorrow."

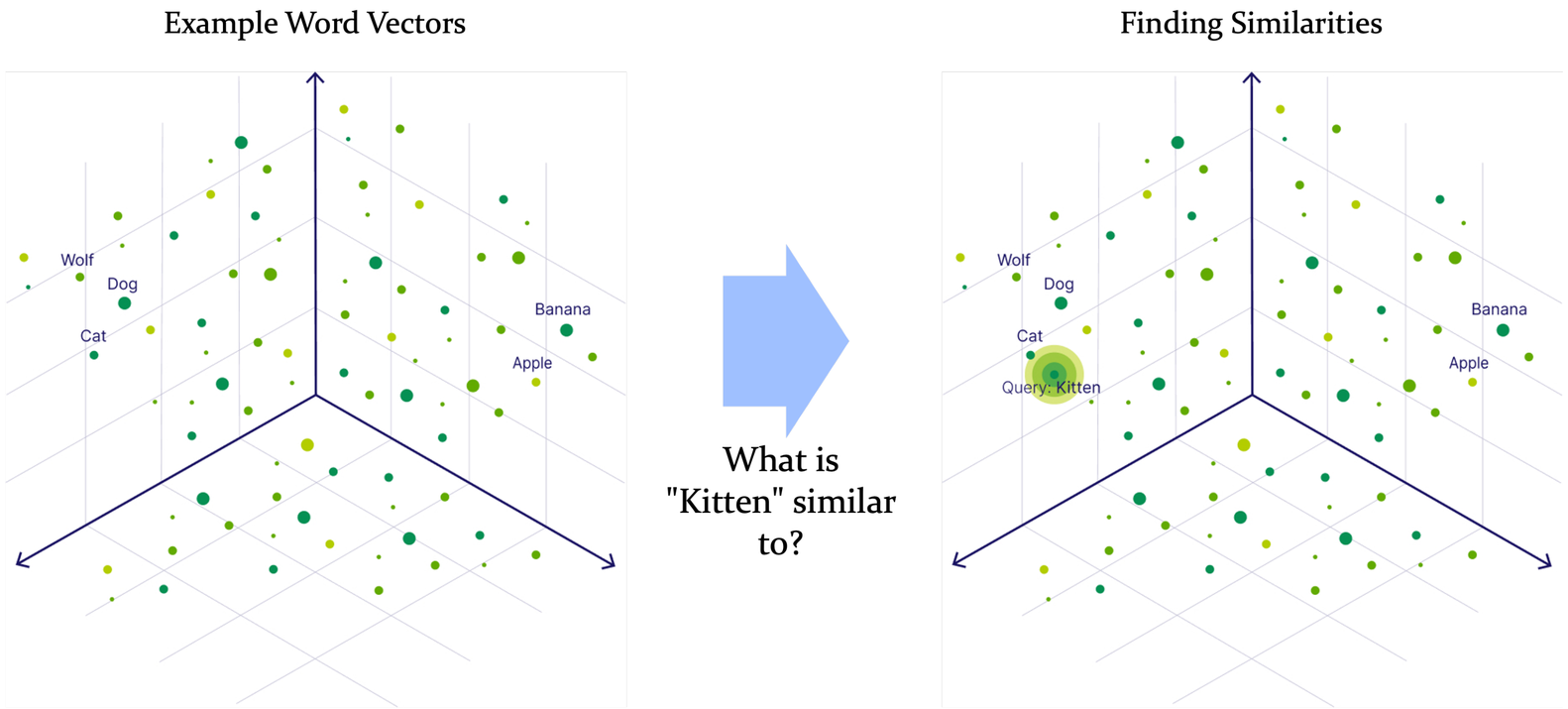


Tokens

Vector Embeddings

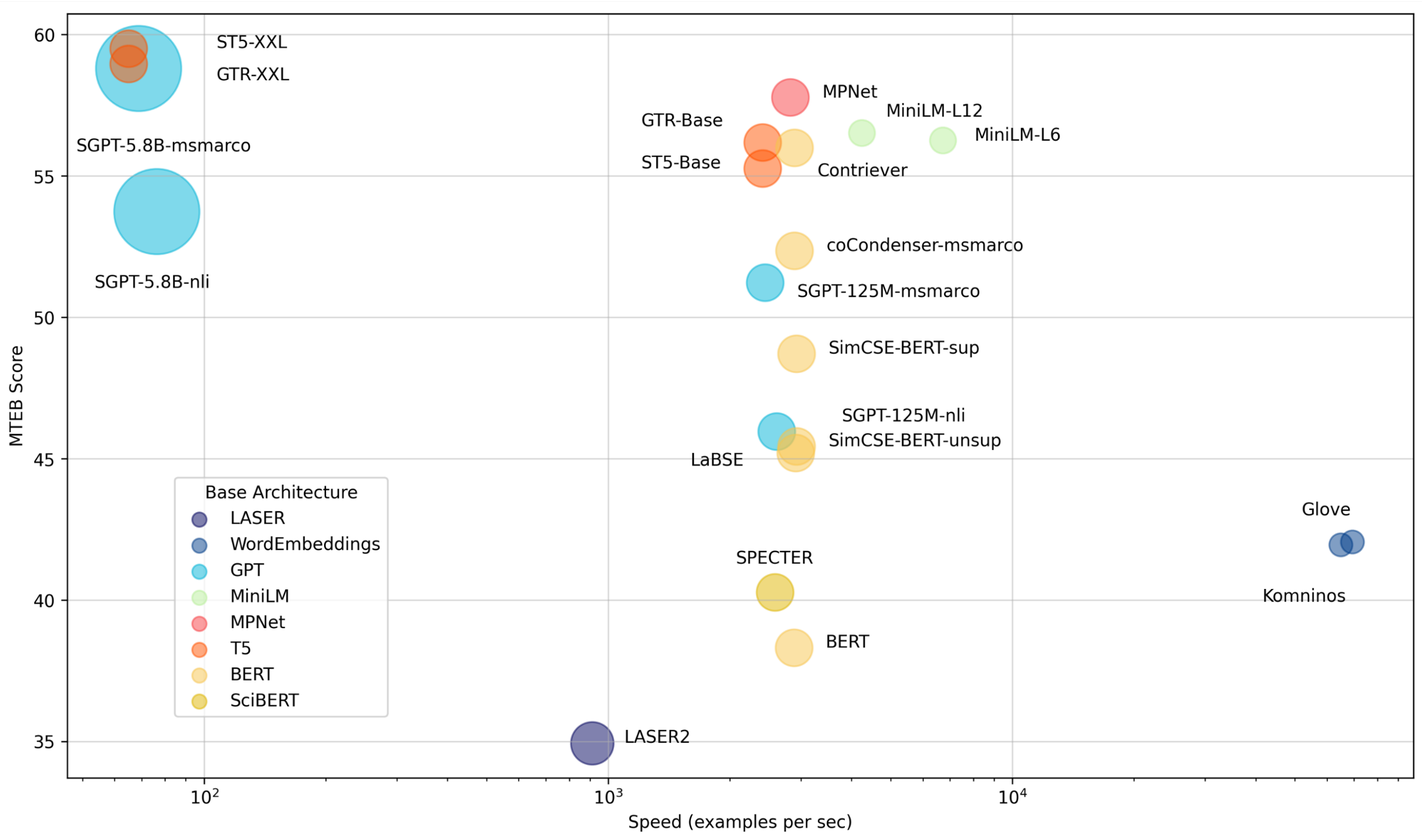
[-0.028, 0.559, 0.142, -0.398, ... , 0.401]

Similarity Search

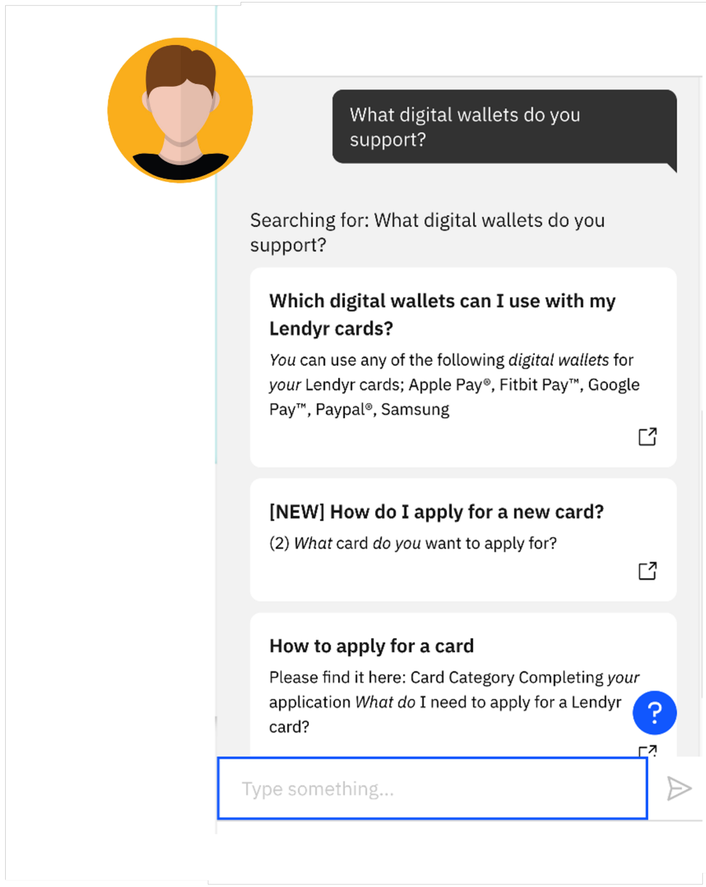


Common similarity measures: Cosine similarity, Euclidean distance, dot product

Retrieval: Embeddings Models



RAG Retrieval



User's question

Few paragraphs related to the question

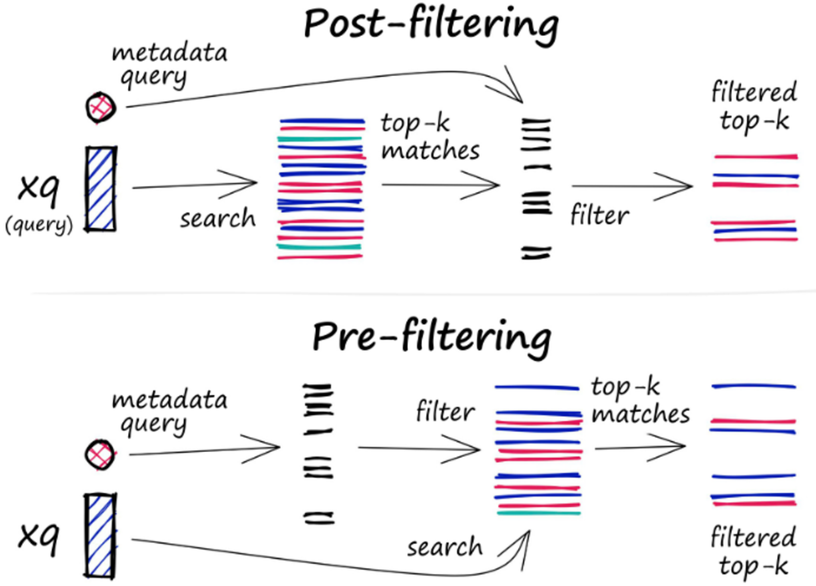
data/information cataloged in a vector database

Read the following paragraphs and answer the question:
[Paragraph 1]
[Paragraph 2]
[Paragraph 3]
Question: [User's question]
Answer:

LLM

Retrieval Filtering

- In addition to the ability to query for similar vectors, vector databases can also filter the results based on a metadata
- Vector database: vector index, metadata index
- **Pre-filtering**: first filter by metadata, then do similarity search against remaining vectors
    - Filtered index is different => have to compare against each vector in new index, i.e., exhaustive instead of approximate search => slow but accurate
- **Post filtering**: First similarity search, then filter by metadata
    - Faster, but might lose accuracy, since similarity search might get rid of hits that would have been relevant by metadata



Semantic vs Syntactic Search

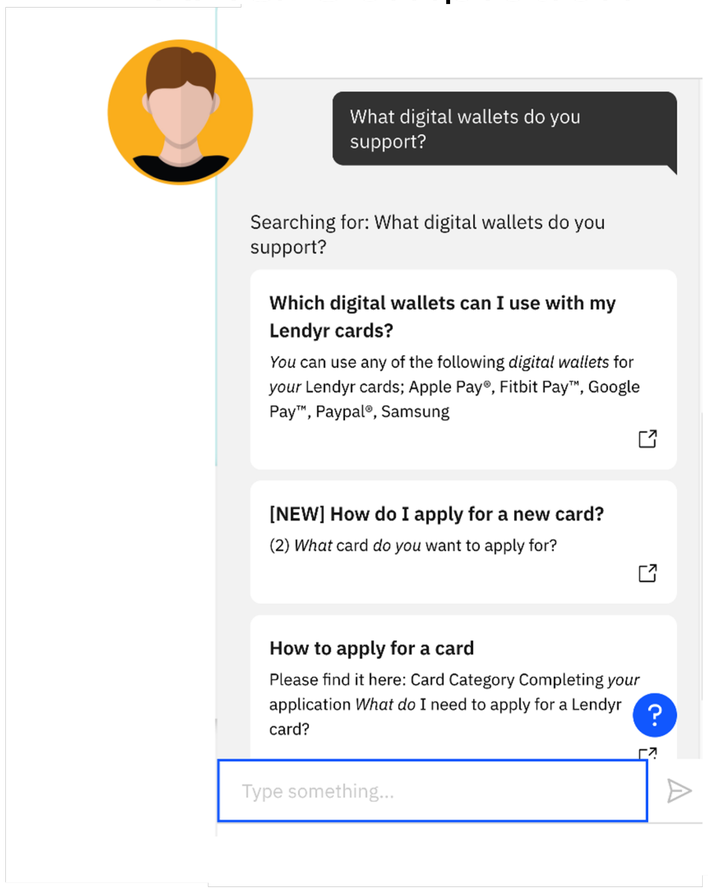
A user expresses themselves in their way, whereas the documents usually use ‘specialized’ terms

Paid leave of absence (HR doc)	Day off
Corporate assets (Code of ethics)	My MacBook
Revenue, profits, benefits (10k form)	Money

Advanced Retrieval Techniques

- Contextual Chunking
- Hybrid Search
- Semantic Reranking
- Prompt Refinement
- Domain-specific Embedding Models

RAG Completion



User's question

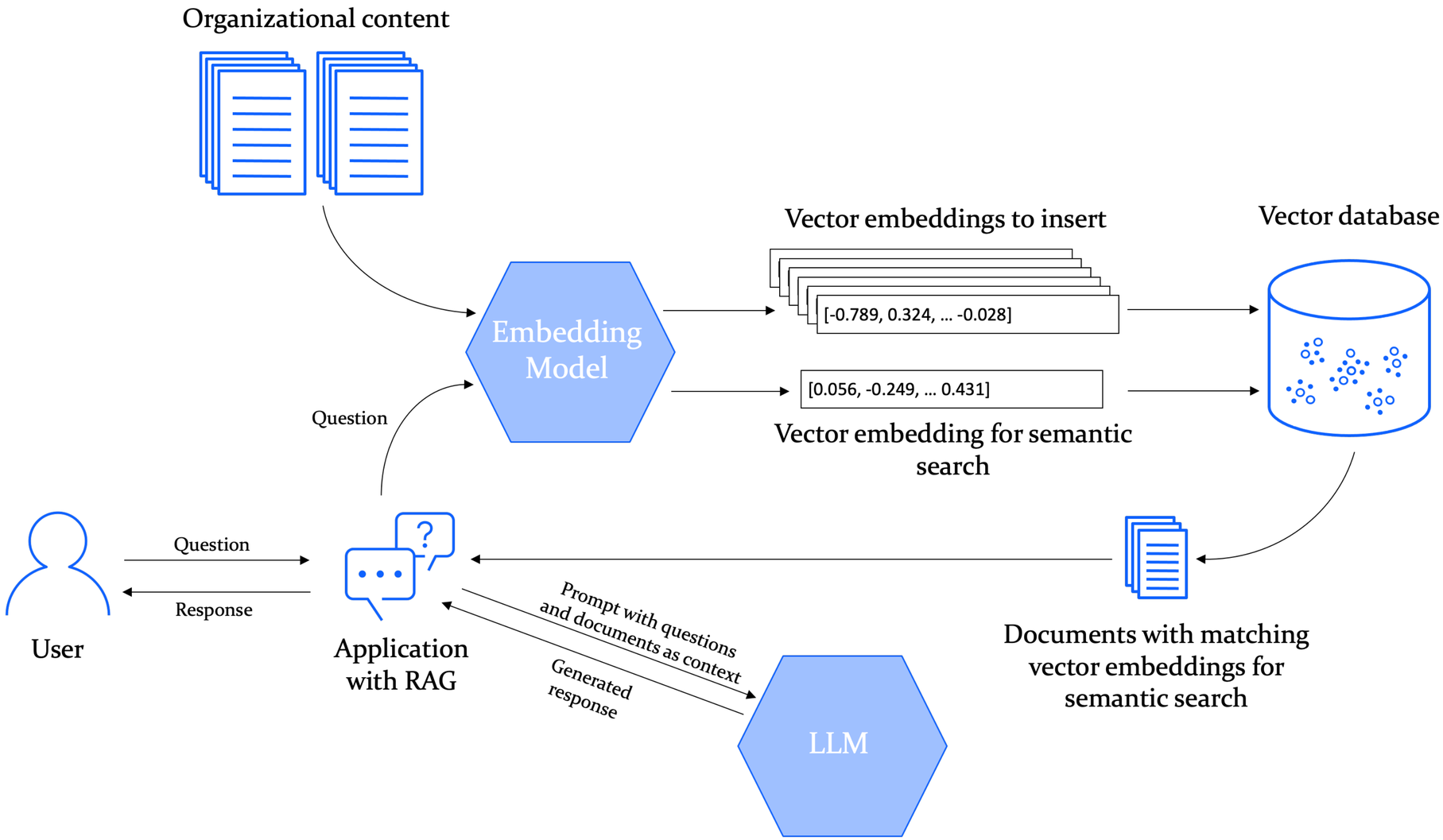
Few paragraphs related to the question

data/information cataloged in a vector database

Read the following paragraphs and answer the question:
[Paragraph 1]
[Paragraph 2]
[Paragraph 3]
Question: [User's question]
Answer:

LLM

RAG Overview



RAG Applications

Summarizing large document

Problem:
Large corpus, potential loss of info and expensive process

Solution:
Leverage RAG for questioning and collect results for summarization

API search agent

Problem:
Querying and processing large corpus of YAML files to feed the LLM

Solution:
Query processed YAML file for Q&A using RAG

Whisper bot

Problem:
Automatic RAG search from the user's historical chat conversations

Solution:
Real-time processing of the user's chat history and utilizing RAG to get the relevant info for the next steps

Evaluating RAG Performance

- Retrieval Quality
- Answer Faithfulness
- Automoated prompt-answer evaluations

RAG Resources

- LangChain currently most widely supported. Used to build various type of applications

- LlamaIndex specifically built for search and retrieval application, more focused on optimization techniques for index querying

- Azure AI Search (GA cloud service)

- Google Vertex AI Embeddings for Text, Matching Engine and PaLM API for Text, Vertex AI Search

- Amazon Kendra similarity search and models from Amazon Bedrock (incl. Amazon Titan) or custom Sagemaker deployments

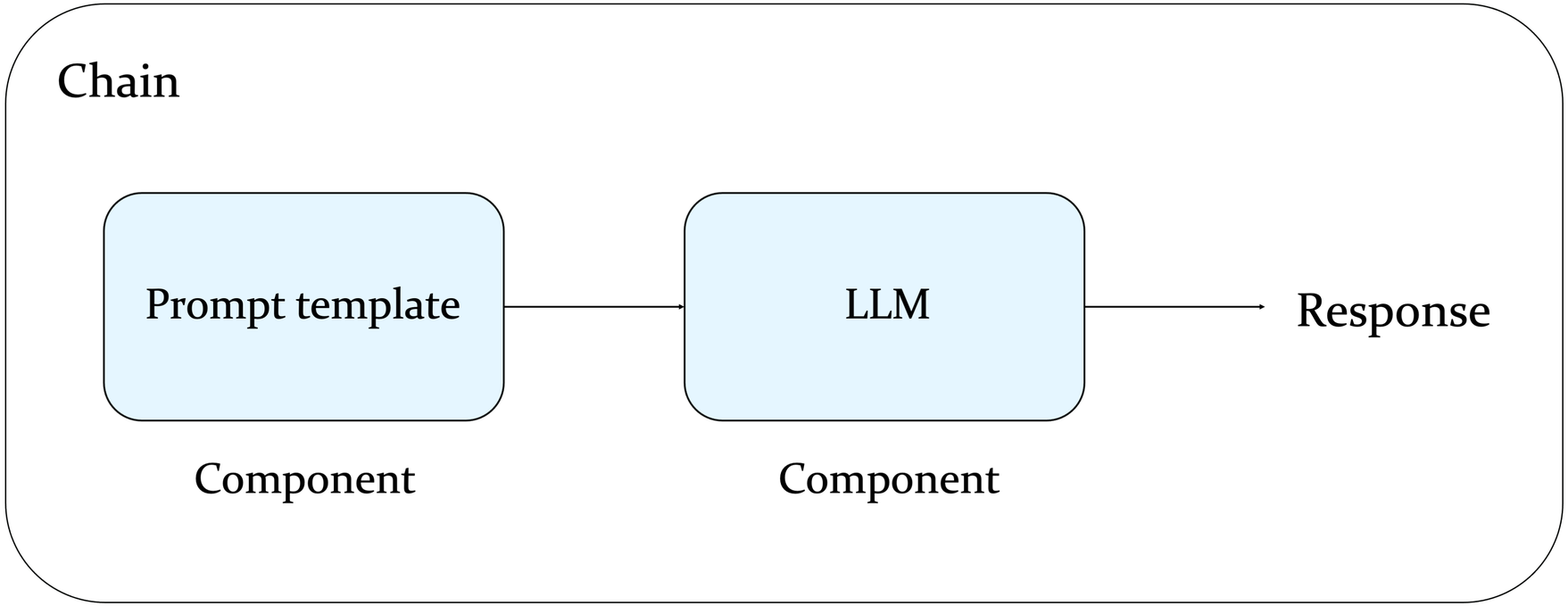
LangChain

- LangChain is a popular open-source framework that’s becoming a de facto standard for LLM application development.

- LangChain simplifies implementation of many tasks that are typical in LLM applications and helps us write “cleaner” code.

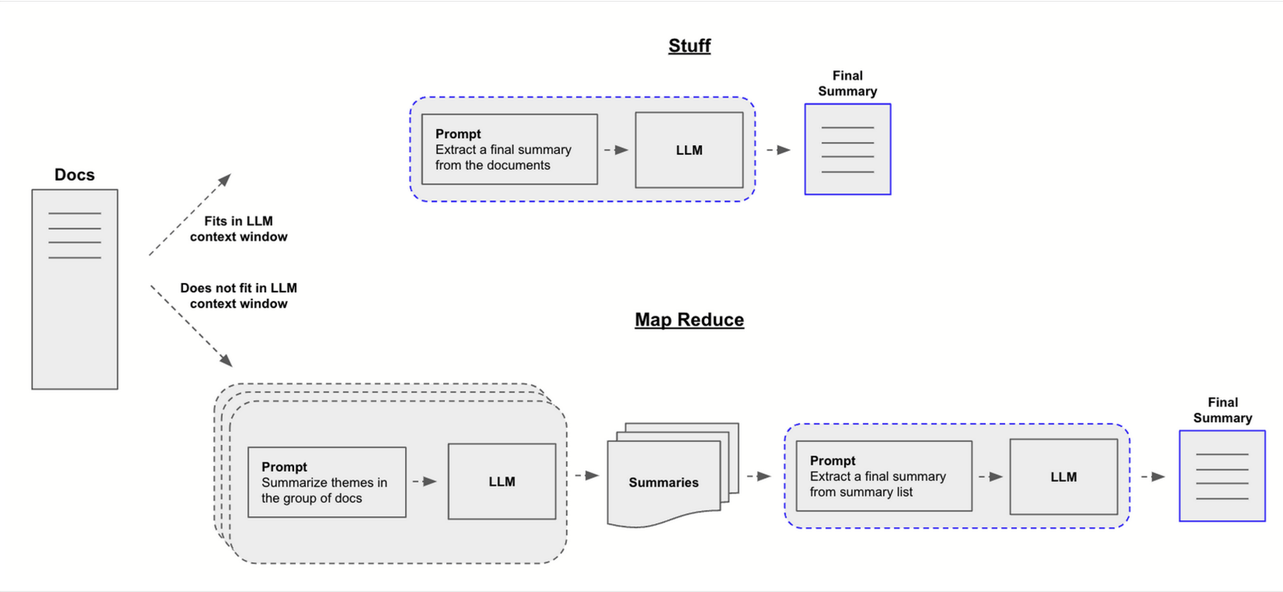
- LangChain does not add new capabilities to LLMs.

A Simple Chain



A simple LLM chain, where a prompt template component is passed to an LLM component to get a response from the language model.

Example



A typical summarization use case requires lots of orchestration and utility code. Langchain provides an API to simplify implementation

LangChain Features

LangChain is:

- A framework for developing applications powered by language models
- A feature-rich, fast-growing
- API

Popular ‘utility’ features are:

- Prompt templates
- LLM output parsers

Wrappers around a series of single components (chains)

Maintaining session state between LLM calls (memory)

- Support for RAG

LangChain can work with LLMS developed by different

vendors, as most LLMs provide a ‘generic API’

LangChain Expression Language (LCEL)

Declarative way to easily compose chains together

LCEL makes it easy to build complex chains from basic components and supports out of the box functionality such as streaming, parallelism, and logging

LangChain extension imlpmentation as a chain

```
chain = prompt | model | output_parser
```

LangChain Expression Language (LCEL)

LangChain Expression Language is a way to create arbitrary custom chains. It is built on the Runnable protocol.

LCEL cheatsheet: For a quick overview of how to use the main LCEL primitives.

Migration guide: For migrating legacy chain abstractions to LCEL.

- How to: chain runnables
- How to: stream runnables
- How to: invoke runnables in parallel
- How to: add default invocation args to runnables
- How to: turn any function into a runnable
- How to: pass through inputs from one chain step to the next

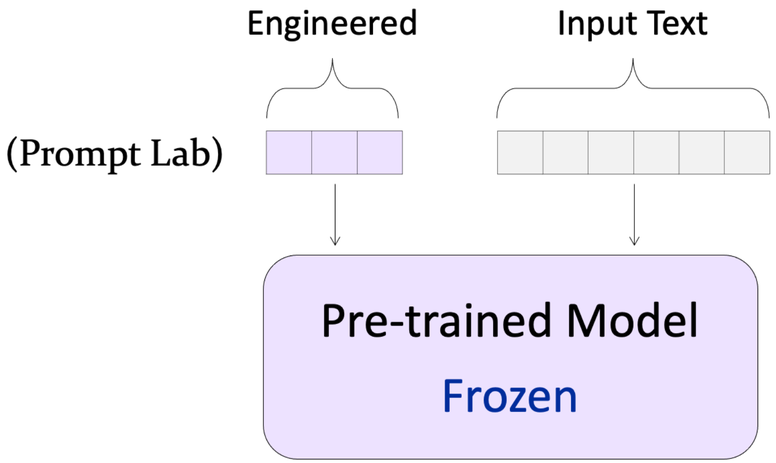
Langchain Expression Language Example

```
# Install LangChain
!pip install langchain

# Import ChatPromptTemplate
from langchain_core.prompts import ChatPromptTemplate

# Create a string that will be formatted to be the system message
system_template = "Translate the following into {language}:“

# Create the PromptTemplate.
prompt_template = ChatPromptTemplate.from_messages(
    [("system", system_template), ("user", "{text}")]
)

# Combine everything using the pipe (|) operator
chain = prompt_template | model | parser
chain.invoke({"language": "italian", "text": "hi"})
```

'ciao'

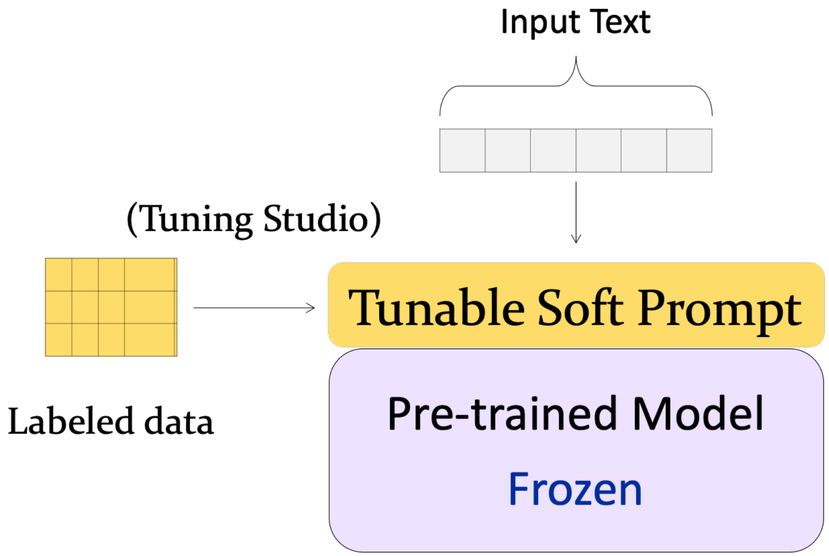
Prompt Tuning

Prompt Engineering



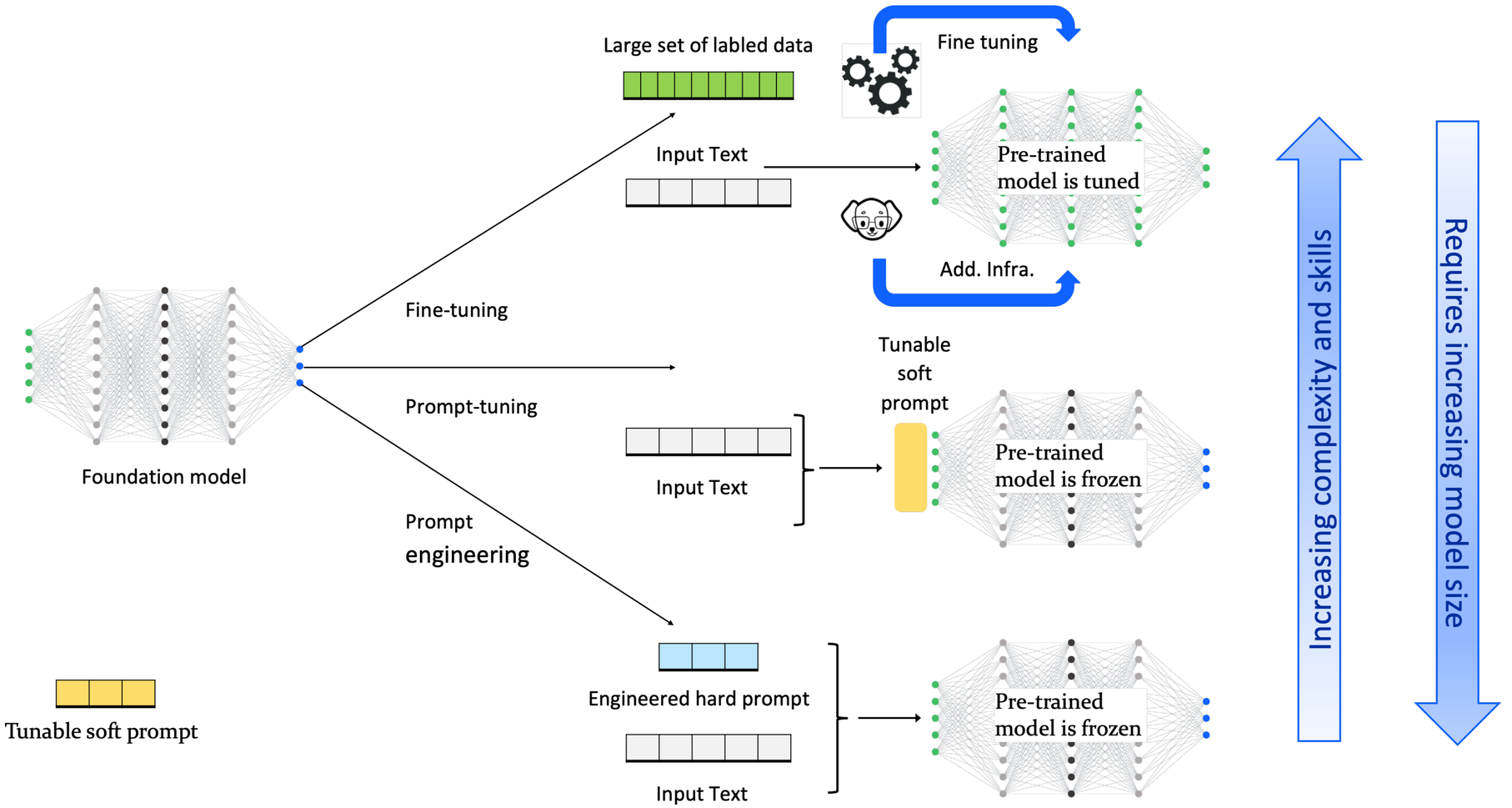
The input prompt, plus any engineered input (such as multi-shot etc.) are passed into the LLM (which remains frozen).

Prompt tuning



A tunable soft prompt is created using the input labeled data and sits on top of the LLM (which remains frozen). This soft prompt is always there for the new model, and can be re-tuned.

Prompt Engineering, Prompt Tuning and Fine Tuning



Prompt Engineering vs Prompt Tuning

Prompt Engineering

- No change to the underlying LLM
- Easier to do, no need for any labeled data input
- Relies more on multi-shot prompting
- Multi-shot is a hard prompt – counts against available tokens
- LLM leverage RAG and a validated corpus of truth to minimize hallucination
- Better for learning

Prompt Tuning

- No change to the underlying LLM
- More resource intensive and time intensive
- Can use multi-shot but shouldn't need
- Soft prompt-does not count against tokens
- Tuned LLM can benefit from RAG, but provides benefits beyond RAG

Prompt Tuning and RAG

Prompt tuning:

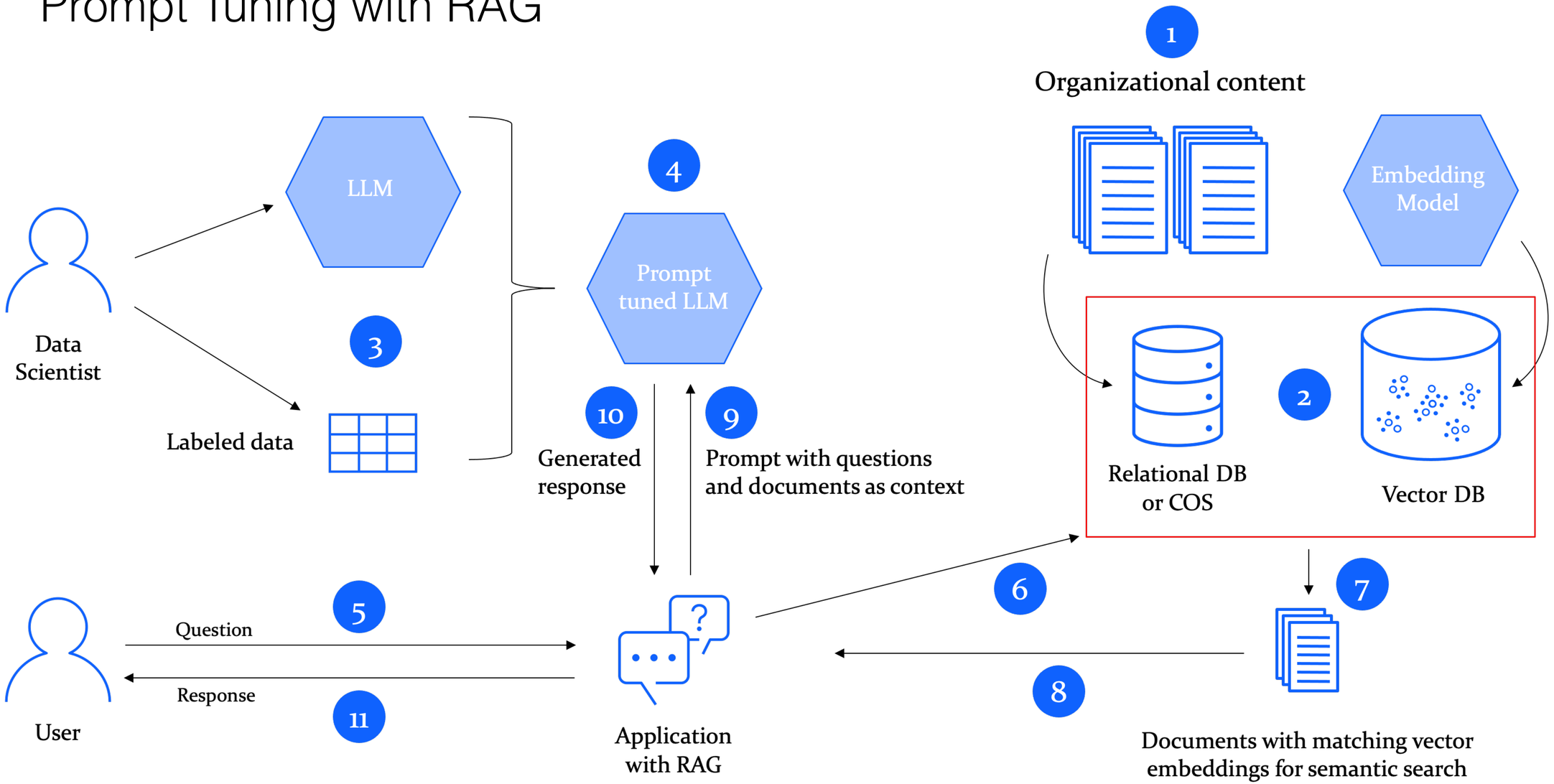
LLM learns business knowledge lacking in its original training.
Augmented (base model not changed) model used to serve downstream tasks such as summarization, classification, generation, etc.
Not required, but can work with a RAG pattern to solve business use cases.

RAG

The RAG pattern is useful when completions should be derived from a corpus of truth.
The LLM generates completions based on a combination of the input prompt as well as extracted info from the corpus of truth.
Its strength is in minimizing hallucination.

Can use either/OR and can use both to achieve even better results

Prompt Tuning with RAG



Prompt Tuning: Smaller vs Larger Model

Smaller Model

- Likely perform less well as-is
- Takes less time to tune
- Less costly to tune
- Easier to re-tune
- Might converge quicker to an optimal solution (for example: a lower value for its loss function given the number of epochs, and the same training data).

Larger Model

- Likely perform better as-is
- Take longer to tune
- More costly (higher CUH requirement) to tune
- More difficult to re-tune
- Tend to exhibit more peaks and valleys in its loss function, might take longer