RAG & Langchain

Models Available

GTP - 3, 3.5 & 4

Generate and Understand Text

Prompt:

Write a tagline for an ice cream shop

Response:

We serve up smiles with every scoop!

GPT-4

Generate text and Understands text and images (multimodal)

Codex

Generate and Understand Code

Prompt:

Table Customers, columns = [CustID, FirstName, LastName, Company, Address, City, State, Country, PostalCode]

Create a SQL statement for selecting customers in texas named Jane

Response:

Select * from customers where state = ‘TX’ and firstname = ‘Jane’;

DaLLe

Generate Images from Text Prompts

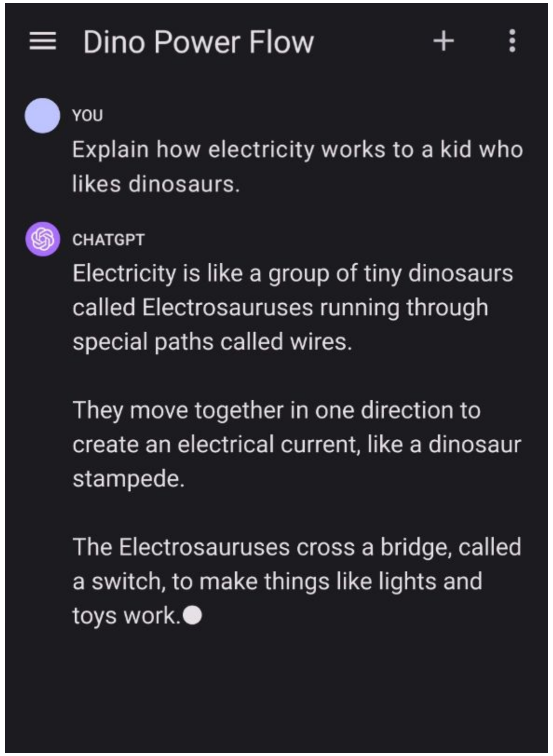
Prompt:

A white siamese cat

Response:

ChatGPT

Generate Conversational Responses

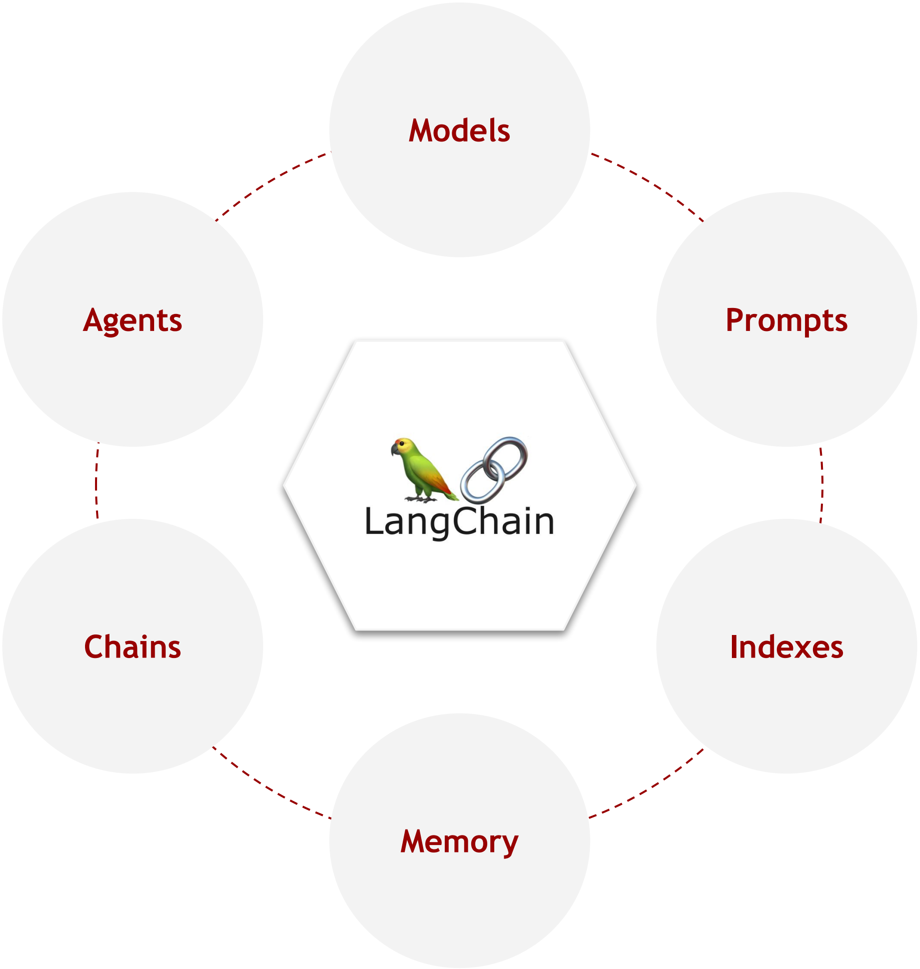


What is LangChain?

- An open-source framework designed to simplify the creation of applications using large language models (LLMs)
- It provides a standard interface for chains, lots of integrations with other tools, and end-to-end chains for common applications
- Allows AI developers to develop applications by combining LLMs such as GPT-4 with external compute and data sources
- Supports Python and JavaScript
- LangChain workflow pipeline:
  - User submits a question to the language model
  - Vector representation of the question is used for a similarity search in the vector database
  - Relevant information is fetched from the vector database
  - Response is returned to the language model
  - Language model generates an answer or takes an action

What is LangChain?

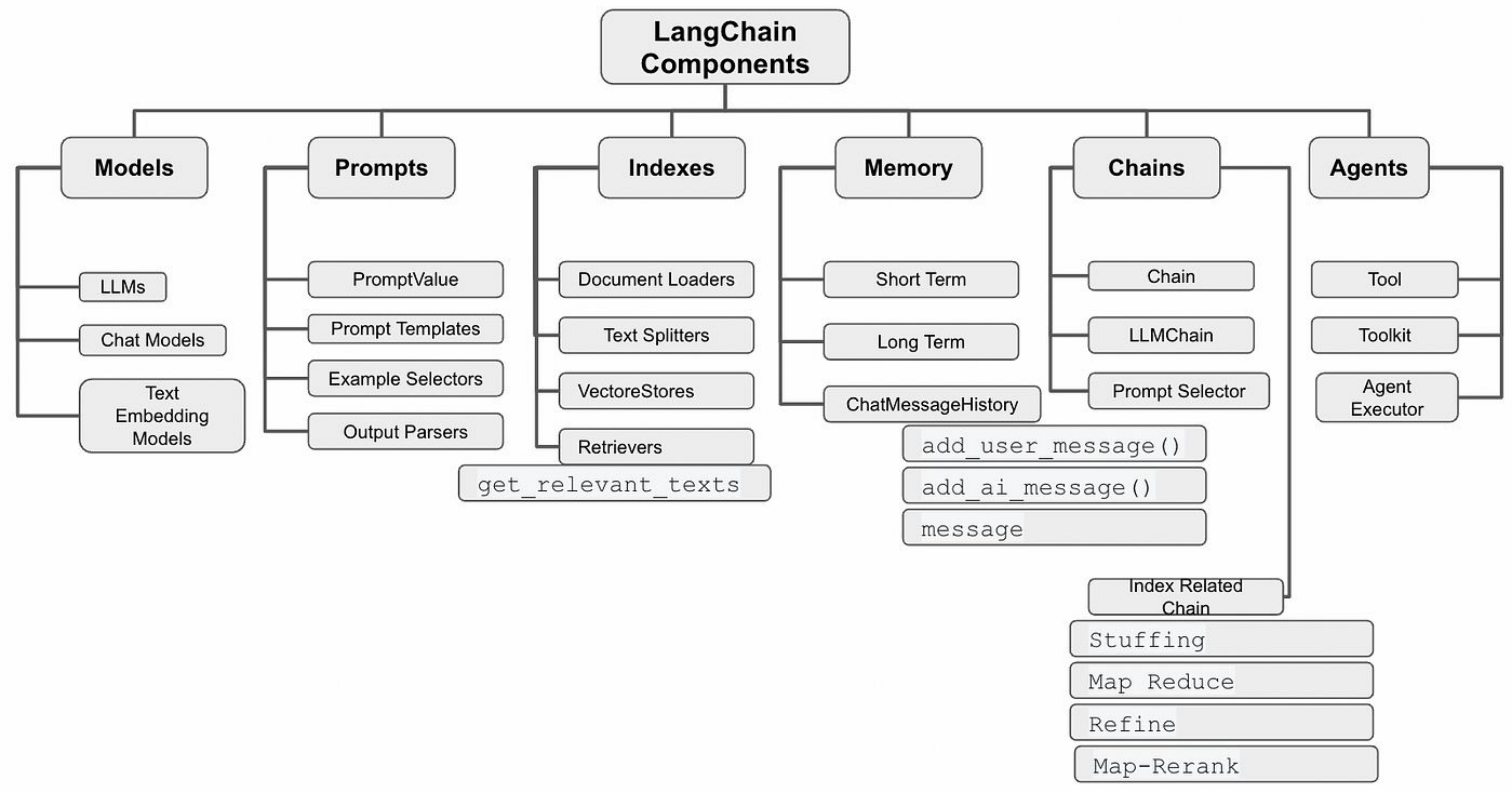
LangChain is composed of six key modules



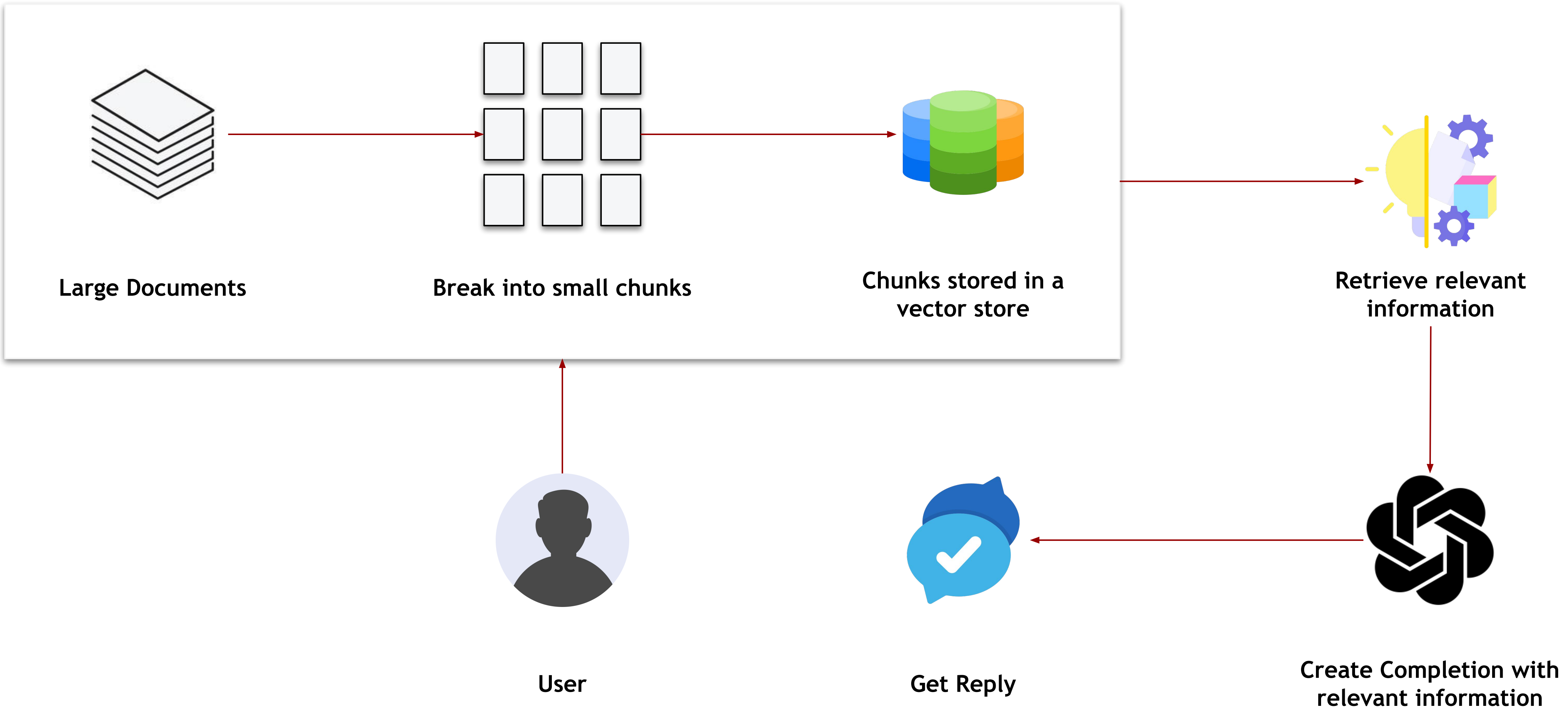
- **LLMs:** LangChain serves as a standard interface that allows interactions with a wide range of LLMs
- **Prompts:** LangChain offers a variety of classes and functions designed to simplify the process of programmatically creating and handling prompts
- **Indexes:** Indexes help organizing reference and underlying documents in a manner that facilitates grounding and accuracy of LLMs
- **Memory:** It has memory modules allowing the management and alteration of past conversations, a key feature for chatbots that need to recall previous interactions
- **Chains:** LangChain provides a standard interface (and some commonly used implementations) for chaining LLMs together for more complex applications, either among themselves or with other specialized modules
- **Agents:** LangChain agents have a comprehensive toolkit and can choose which tools to utilize based on user input

What is LangChain?

LangChain is composed of six key modules

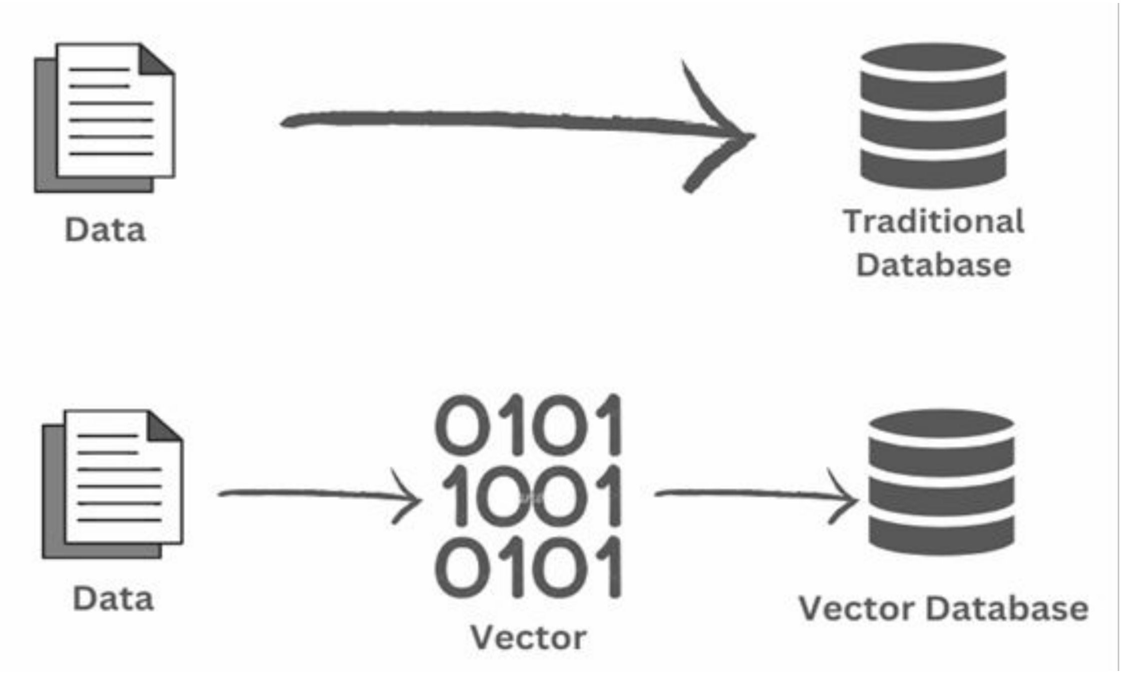


How does LangChain work?



What are Vector Store Databases?

A vector database is a type of database that indexes and stores vector embeddings for fast retrieval and similarity search, with capabilities like CRUD operations, metadata filtering, and horizontal scaling



Why do we need Vector Store Databases?

Because of **Embeddings**

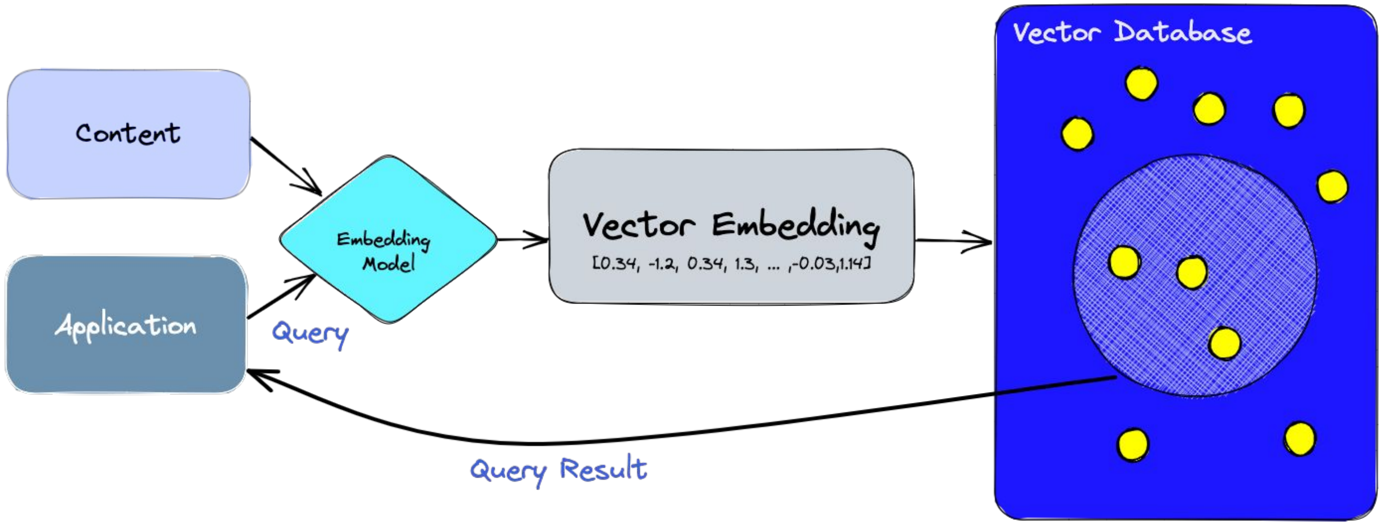
These are generated by AI models (such as Large Language Models) and have many attributes or features, making their representation challenging to manage. In the context of AI and machine learning, these features represent different dimensions of the data that are essential for understanding patterns, relationships, and underlying structures

- The challenge of working with vector data is that traditional scalar-based databases can't keep up with the complexity and scale of such data, making it difficult to extract insights and perform real-time analysis
- Vector databases are specialized database designed for optimized storage and querying capabilities for embeddings
- They have the capabilities of a traditional database that are absent in standalone vector indexes and the specialization of dealing with vector embeddings, which traditional scalar-based databases lack
- Vector databases offer the performance, scalability, and flexibility required by AI models
- With the help of a vector database, AI models can have advanced features like semantic information retrieval, long-term memory, and more

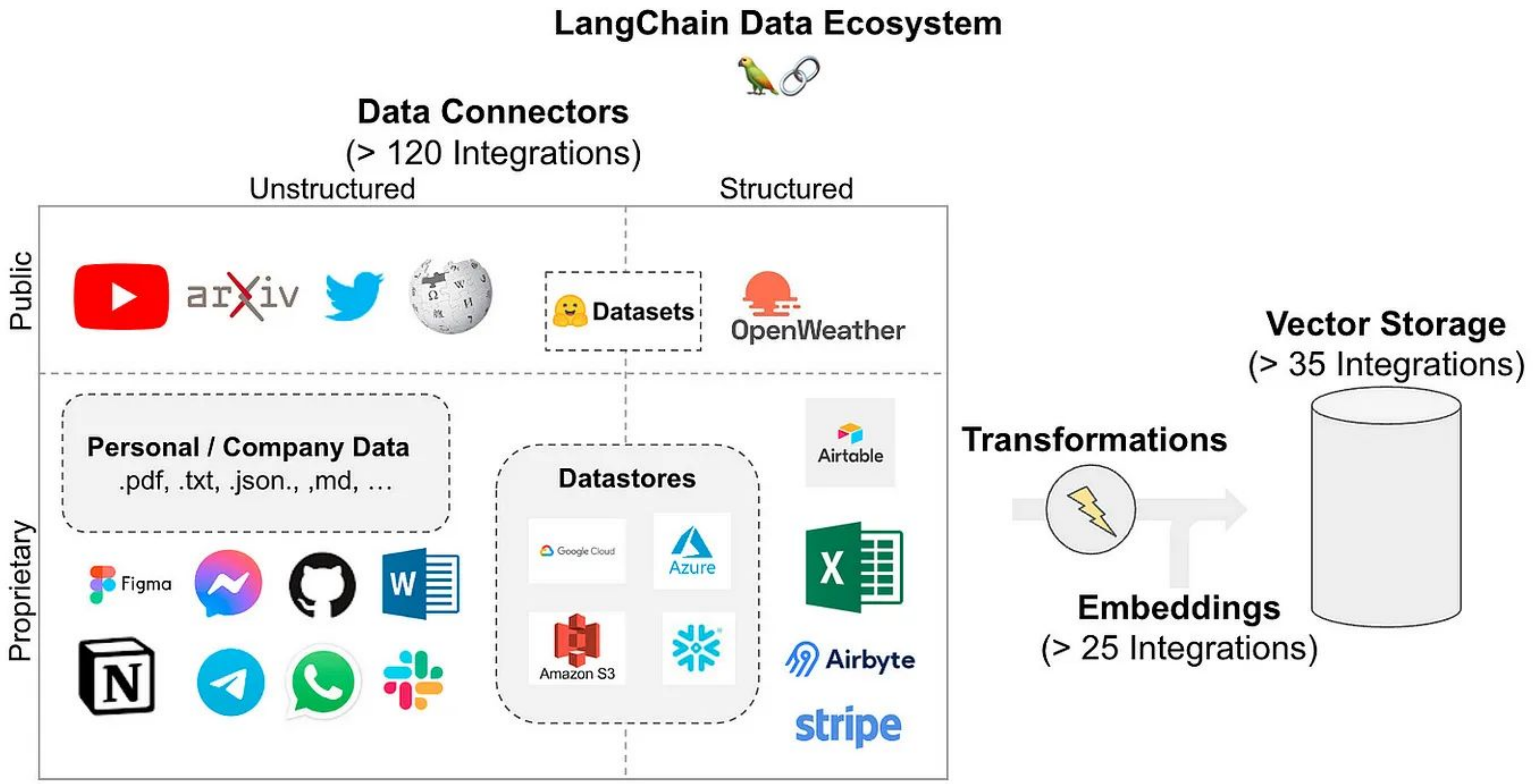
How data is stored with Vector Store Databases?

Let's break this down:

1. First, we use the ***embedding model*** to create ***vector embeddings*** for the content we want to index
2. The ***vector embedding*** is inserted into the ***vector database***, with reference to the original content the embedding was created from
3. Upon querying, the embedding model is used to create embeddings for the query and use those embeddings to query the database for *similar* vector embeddings. The query results (embeddings) are associated with the original content that was used to create them



LangChain Data Landscape

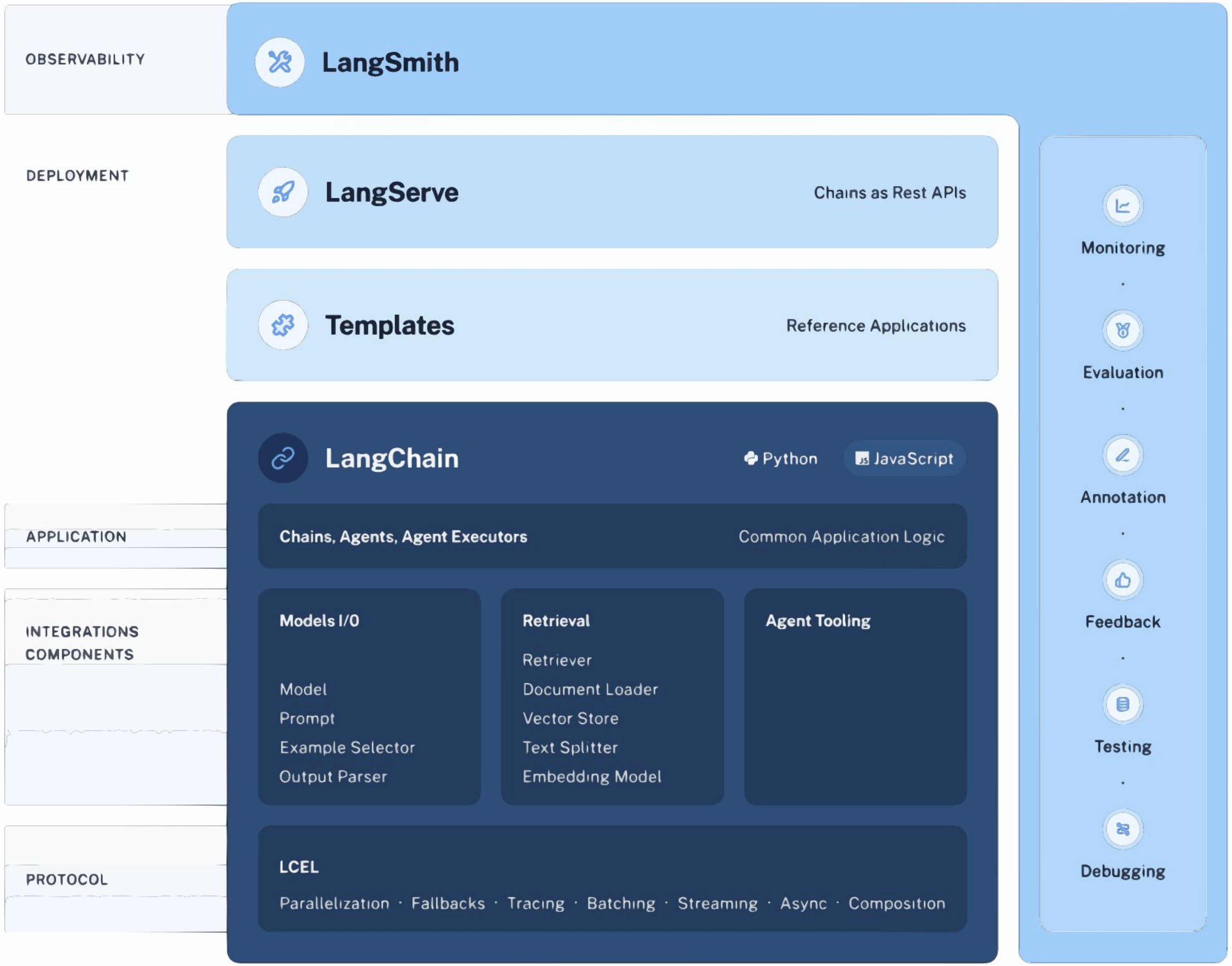


LangChain Framework

This framework consists of several parts

- **LangChain Libraries**: Python and JavaScript libraries contain interfaces and integrations for a myriad of components, a basic run time for combining these components into chains and agents, and off-the-shelf implementations of chains and agents.
- **LangChain Templates**: A collection of easily deployable reference architectures for a wide variety of tasks.
- **LangServe**: A library for deploying LangChain chains as a REST API.
- **LangSmith**: A developer platform that lets you debug, test, evaluate, and monitor chains built on any LLM framework and seamlessly integrates with LangChain.

LangChain Framework



LangChain Framework

DEVELOP

Write your applications in LangChain/LangChain.js. Hit the ground running using templates for reference

PRODUCTIONIZE

Use LangSmith to inspect, test and monitor your chains, so that you can constantly improve and deploy with confidence

DEPLOY

Turn any chain into an API with LangServe

How to create prompts in LangChain

1. Installing Python and LangChain

```
pip install LangChain
```

2. Adding Integrations

```
pip install openai
from LangChain.llms import OpenAI
llm = OpenAI(openai_api_key="...")
```

3. Importing the Prompt Template

```
from LangChain import PromptTemplate
prompt_template = PromptTemplate.from_template(
   "Tell me an {adjective} fact about {content}."
)
prompt_template.format(adjective="interesting", content="zebras")
"Tell me an interesting fact about zebras."
```

How to build applications in LangChain?

1

DEFINE THE APPLICATION

Define a specific use case or objectives for the application, i.e., define its scope, required integrations, components, and LLMs

2

BUILD FUNCTIONALITY

Use prompts to build the functionality or logic of the intended app

3

CUSTOMIZE FUNCTIONALITY

Modify LangChain code to create customized functionality to meet the needs of unique use cases and shape application behavior

4

FINE TUNE LLMs

Choose an appropriate LLM for the job and fine-tune it to get the best results for the use case

5

DATA CLEANSING

Using data cleansing techniques ensures clean and accurate data sets for relevant results

6

TESTING

Regularly test LangChain apps to ensure they continue to run securely and smoothly

Interacting with LLMs in LangChain

Using LangChain to interact with LLMs involves a series of steps to help you leverage the power of these AI models for text generation and various other tasks.

- **Initializing an LLM**
- **Inputting Prompts**
- **Response Retrieval**

Interacting with LLMs in LangChain

- **Initializing an LLM:** First to import the necessary libraries and dependencies. For example, if you're using Python, import the `LangChain` library and specify the language model you want to use:

```python
from langchain import LangModel

# Specify the language model you want to use
model_name = 'gpt3'

# Initialize the LLM
llm = LangModel(model_name)
```

Interacting with LLMs in LangChain

- **Inputting Prompts:** Input prompts to generate text or get responses. Provide a single prompt (for single-turn conversations) or multiple prompts (for multi-turn conversations), depending on your requirements. Here's an example:

```
# Input a single prompt
prompt = "Once upon a time"

# Generate text based on the prompt
generated_text = llm.generate_text(prompt)
```

Interacting with LLMs in LangChain

- **Retrieving Generated Text or Responses:** Retrieve the generated text or responses from the LLM. The generated responses will be based on the capabilities of the language model and the context provided in the prompt(s). For example:

```
# Print the generated text
print(generated_text)

# Print the responses
for response in responses:
    print(response)
```

What can be done with LangChain?

LangChain’s key components — models, prompts, indexes, memory, chains, agents - help us accomplished more.

- **Models:** LangChain offers a uniform interface to integrate even with emerging models providing you access to an extensive collection of powerful LLMs

```
from langchain.llms import OpenAI
llm = OpenAI(model_name="text-davinci-003")

# The LLM takes a prompt as an input and outputs a completion
prompt = "How many days are there in a month"
completion = llm(prompt)
```

What can be done with LangChain?

- **Prompts:** By applying prompt engineering principles you can make good prompt templates. You can also use LangChain provided PromptTemplates to build prompts using various components.

```python
template = "What is a good name for a company that makes {product}?"

prompt = PromptTemplate(
    input_variables=["product"],
    template=template,
)

prompt.format(product="colorful socks")
```

What can be done with LangChain?

- **Indexes:** Vector indexes are used with external data stored as vector embeddings in a Vector Store for faster access to large volumes of data.

```python
retriever = db.as_retriever()

qa = RetrievalQA.from_chain_type(
    llm=llm,
    chain_type="stuff",
    retriever=retriever,
    return_source_documents=True)

query = "What am I never going to do?"
result = qa({"query": query})
```

What can be done with LangChain?

- **Memory:** By default, LLMs lack long-term memory capabilities. To provide this feature agents must have access to previous conversations (chat history). LangChain maintains all conversations and even summarizes the most recent K conversations for effective conversations.

```
conversation = ConversationChain(llm=llm, verbose=True)
conversation.predict(input="Alice has a parrot.")
conversation.predict(input="Bob has two cats.")
conversation.predict(input="How many pets do Alice and Bob have?")
```

What can be done with LangChain?

- **Text Embedding Model:** Text input is used by text embedding model to query the embeddings database and output a list of embeddings *similar* to the input text. Using the output embedding information can be extracted to summarize, compare, or analyze the text inputs.

```
embeddings = OpenAIEmbeddings()

text = "Alice has a parrot. What animal is Alice's pet?"
text_embedding = embeddings.embed_query(text)
```

What can be done with LangChain?

- **<u>Chains:</u>** Chaining is the process of combining LangChain components like LLM, prompts, and agents to create an application. Some examples:
    - Combining prompt templates and LLMs
    - Using the output of one agent (LLM) as the input for another in a sequence of LLMs
    - Combining external data with LLMs to answer domain/org-specific questions
    - Combining long-term memory capabilities (like from chat history) with LLMs

```
chain = LLMChain(llm = llm,
                 prompt = prompt)

chain.run("colorful socks")
```

What can be done with LangChain?

- **<u>Agent/Chat Model:</u>** To set up a conversation between a user and an agent, create an instance of the appropriate agent class. Start the conversation with a system message stating the objective and overall context for the bot . Follow it with a human message(s) seeking a response. For example, here an OpenAI chatbot is created by instantiating the ChatOpenAI class and initially its temperature is set to 0, making its responses focused and deterministic:

```
chat = ChatOpenAI(temperature=0)

chat(
    [
        SystemMessage(content="You are a nice AI bot that helps a user figure out
          what to eat in one short sentence"),
        HumanMessage(content="I like tomatoes, what should I eat?")
    ]
)
```

Evaluation

After fine-tuning the LLM, it's crucial to evaluate its performance. This step helps you assess how well the model has adapted to your specific task. You can evaluate the fine-tuned model using appropriate metrics and a separate test dataset. Here's an example:

```
# Prepare the test dataset
test_dataset = load_dataset('your_test_dataset.txt')
preprocessed_test_dataset = preprocess(test_dataset)

# Evaluate the fine-tuned LLM
evaluation_results = fine_tuned_model.evaluate(preprocessed_test_dataset)
```

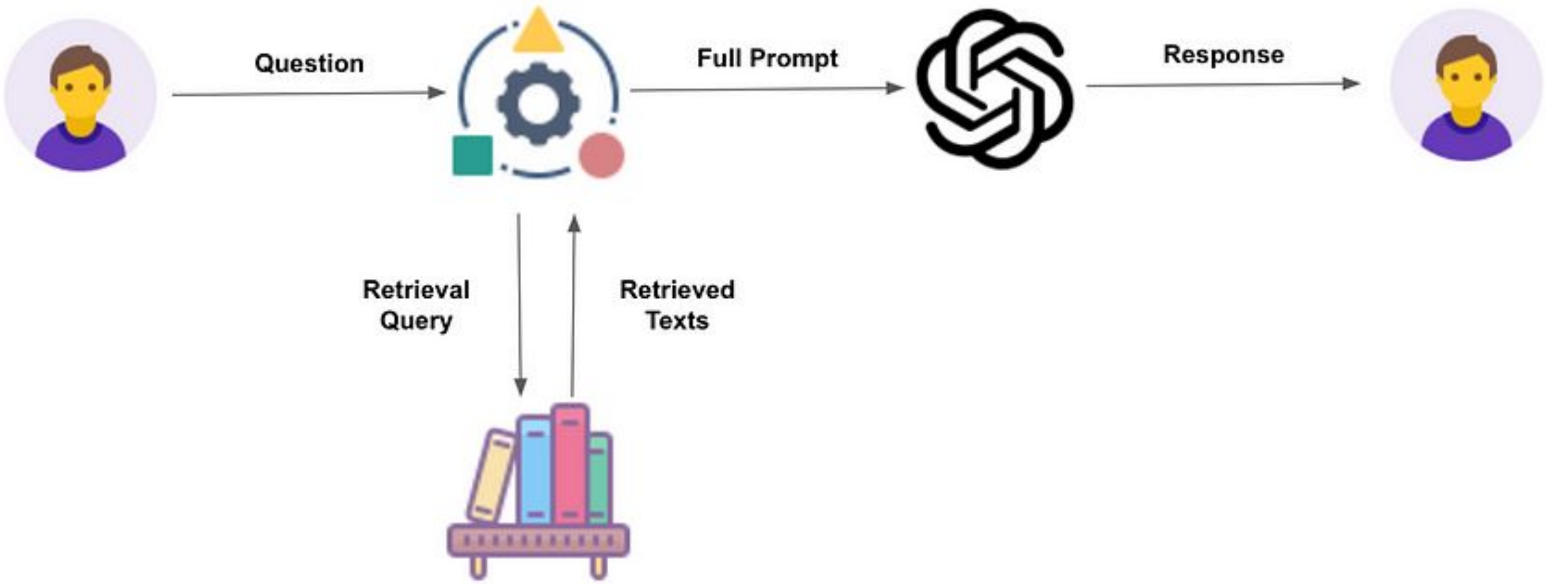
Benefits of using LangChain for Finetuning

- Finetuning LLMs with LangChain improves the model’s accuracy and contextual relevance for specific tasks or domains, resulting in higher-quality outputs
- LangChain allows developers to customize LLMs to handle unique tasks, industry-specific jargon, and domain-specific contexts, catering to specific user needs
- Fine tuned LLMs enable the development of powerful applications with a deeper understanding of domain-specific language, leading to more accurate and contextually aware responses
- Finetuning with LangChain reduces the need for extensive training data and computational resources, saving time and effort while achieving significant performance improvement

Introduction to RAG

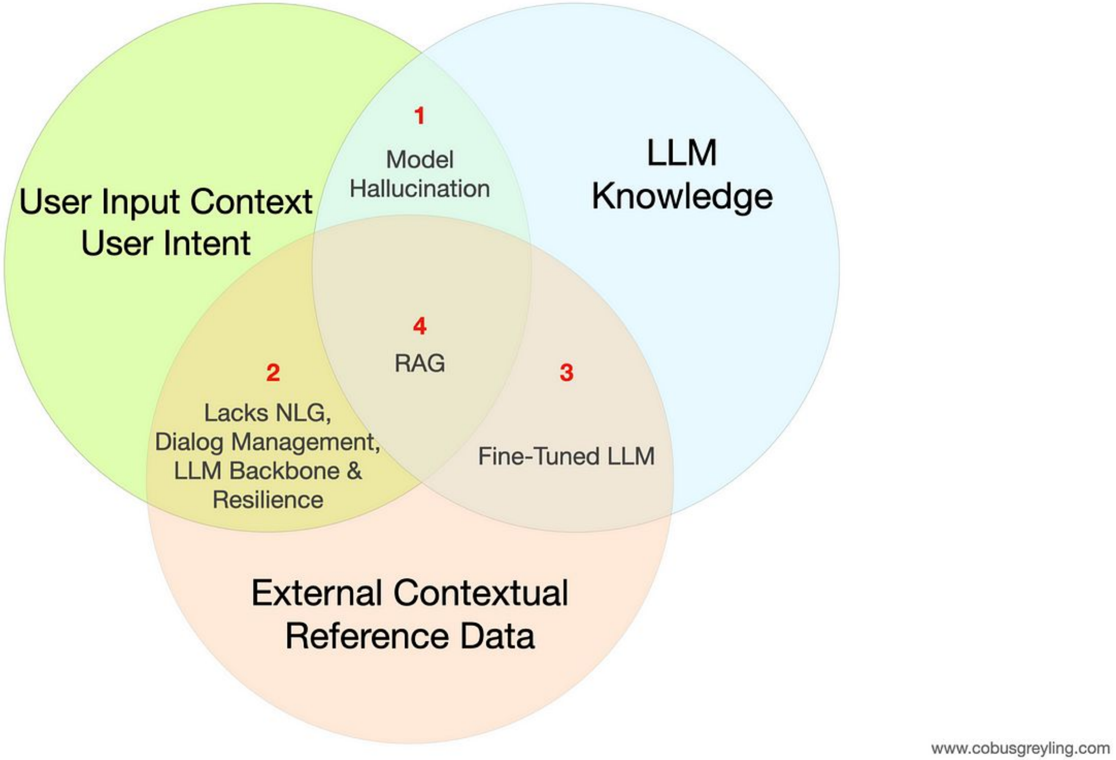
RAG combines retrieval, augmentations, and generation in natural language processing, enabling models to retrieve information from a vast knowledge source and use it to augment their generation capabilities.

- **Retrieval**: Accessing external knowledge sources (e.g., databases, documents, web) for information.
- **Generation**: Creating human-like text or content.
- **Augmentation or Integration**: How RAG integrates these two to enhance language understanding and generation.



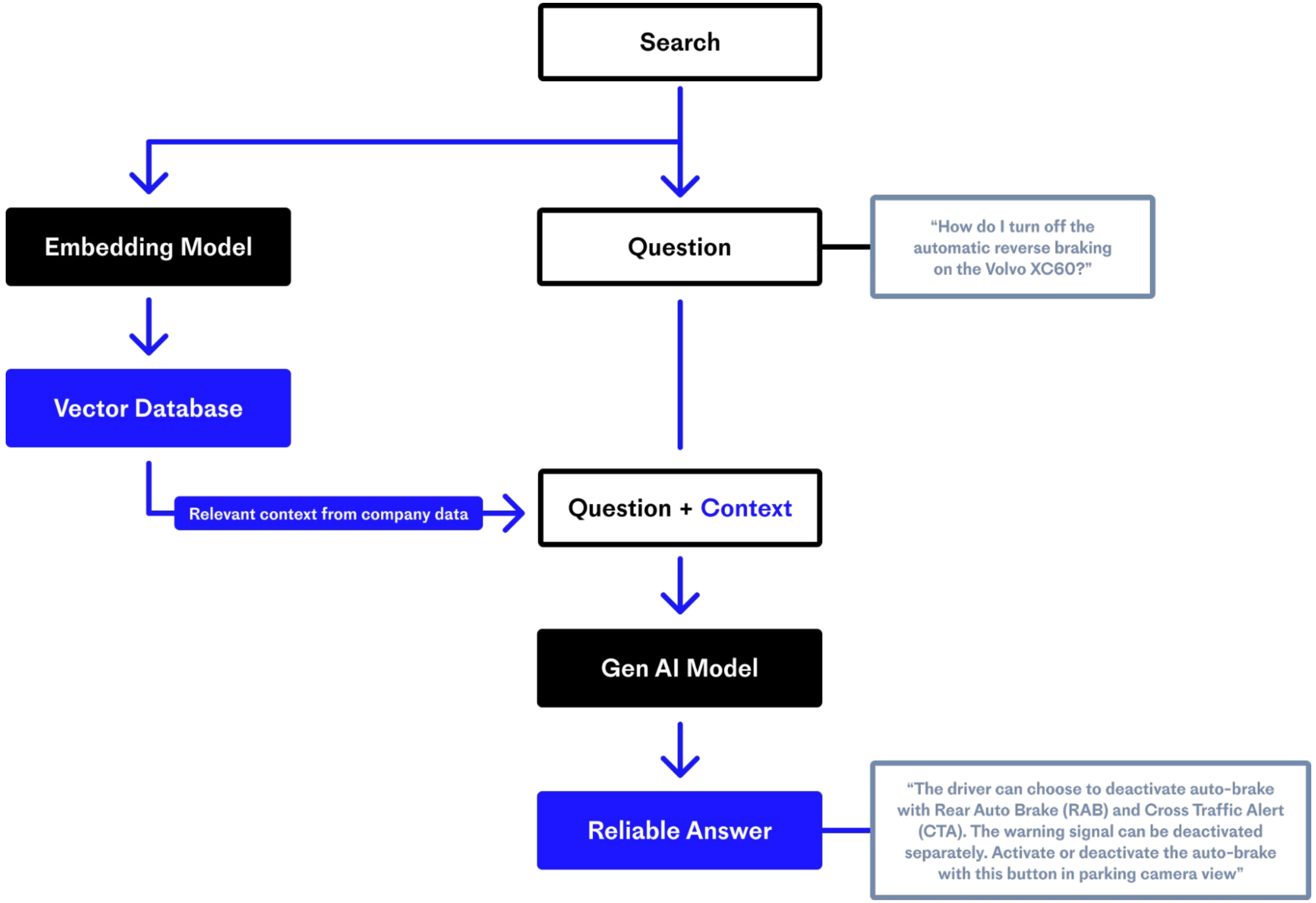
What is the need for RAG?

- Pre-trained LLMs (foundation models) do not learn over time, often hallucinate, and may leak private data from the training corpus
- To overcome these limitations, there has been growing interest in retrieval-augmented generation which incorporate a vector database and/or feature store with their LLM to provide context to prompts, also known as RAG LLMs



Architecture of RAG

- **Retriever**: Searches and retrieves relevant information from external knowledge sources
- **Generator**: Language model responsible for creating human-like text or content
- **Integration**: How the retrieved information influences or augments the generation process



Working of RAG

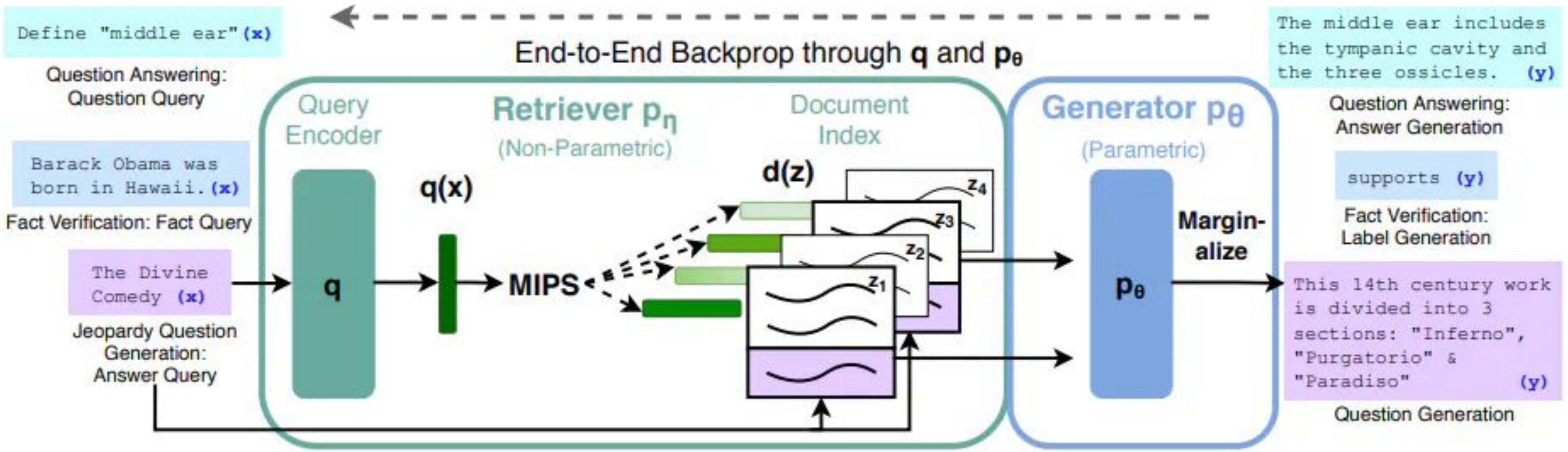


Figure 1: Overview of our approach. We combine a pre-trained retriever (*Query Encoder* + *Document Index*) with a pre-trained seq2seq model (*Generator*) and fine-tune end-to-end. For query x, we use Maximum Inner Product Search (MIPS) to find the top-K documents z_i. For final prediction y, we treat z as a latent variable and marginalize over seq2seq predictions given different documents.

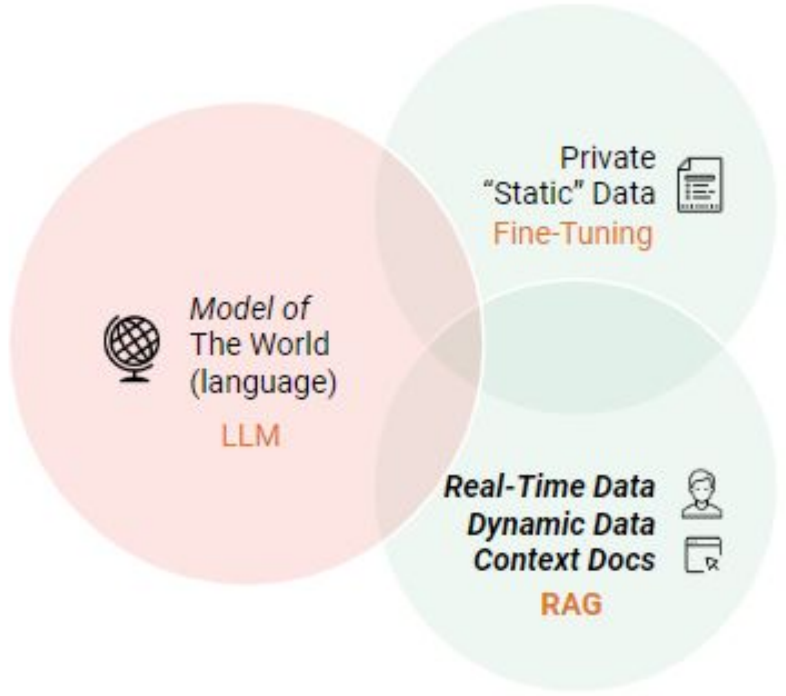
How does RAG compare with Fine Tuning?

- Fine-tuning takes a pre-trained LLM and further trains the model on a smaller task-specific dataset, often with data not previously used to train the LLM, to improve its performance for that task

- RAG integrates retrievers to enhance or augment (in real-time) natural language understanding and generation tasks of the LLM.

- Fine-tuning is appropriate when you want to customize an LLM to perform well in a particular domain using private data. For e.g., fine-tuning an LLM to become better at producing Python programs by further training the LLM on high-quality Python source code

- We should use RAG when we are able to augment LLM prompts with data that was not known to the LLM at the time of training, such as real-time data, personal (user) data, or context information useful for the prompt



Applications of RAG

- **Question Answering**: RAG can retrieve information from a vast knowledge base to answer questions accurately
- **Content Creation**: Generating coherent and informative content by leveraging external information
- **Conversational Agents**: Enhancing chatbots' ability to provide contextually relevant responses by incorporating external knowledge


THANK YOU