HOW TO BUILD A CHATBOT

Session 3 - Retrieval Augmented Generation

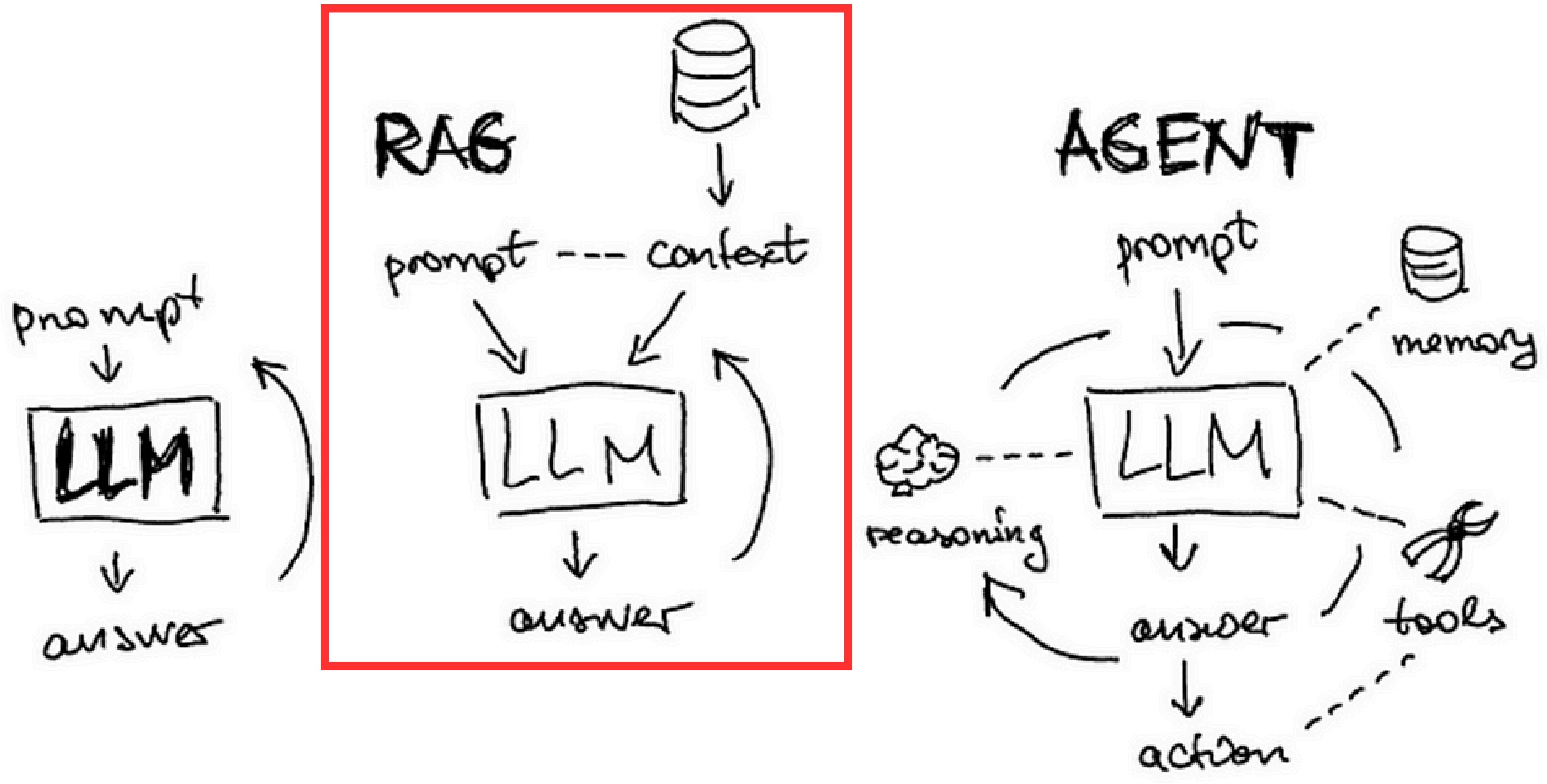
SESSION 3 AGENDA

1. Introduction to RAG
2. Vectors Embeddings
3. Vector Databases
3. Retrieval (Vector Search)

INTRODUCTION TO RAG

Retrieval Augmented Generation (RAG):

- **What?** Technique that enhances text generation of LLMs by retrieving and incorporating external knowledge; output factual information rather than relying on the knowledge that is encoded (learned) in the LLMs parameters.
- **Why?** avoid hallucinations of LLMs; domain knowledge of LLMs is limited; by incorporating outside knowledge the generated text is more useful and factually correct.
- **How?** Passing dynamically relevant domain specific information from external data sources to the prompt as context (prompt injection)

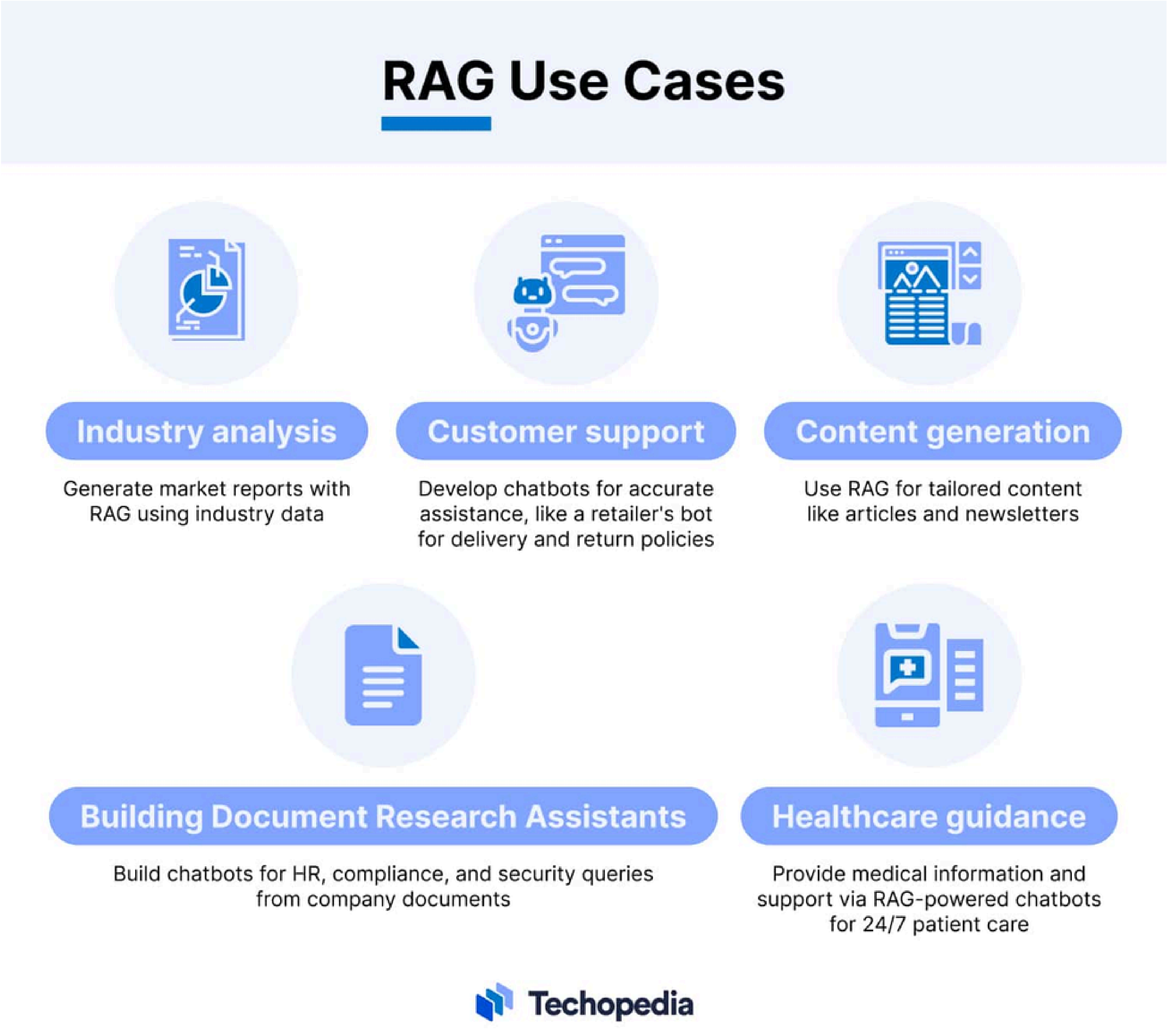


INTRODUCTION TO RAG

Example Customer Support:

Businesses can use RAG to create customer support chatbots that provide users with access to more accurate and reliable domain specific information.

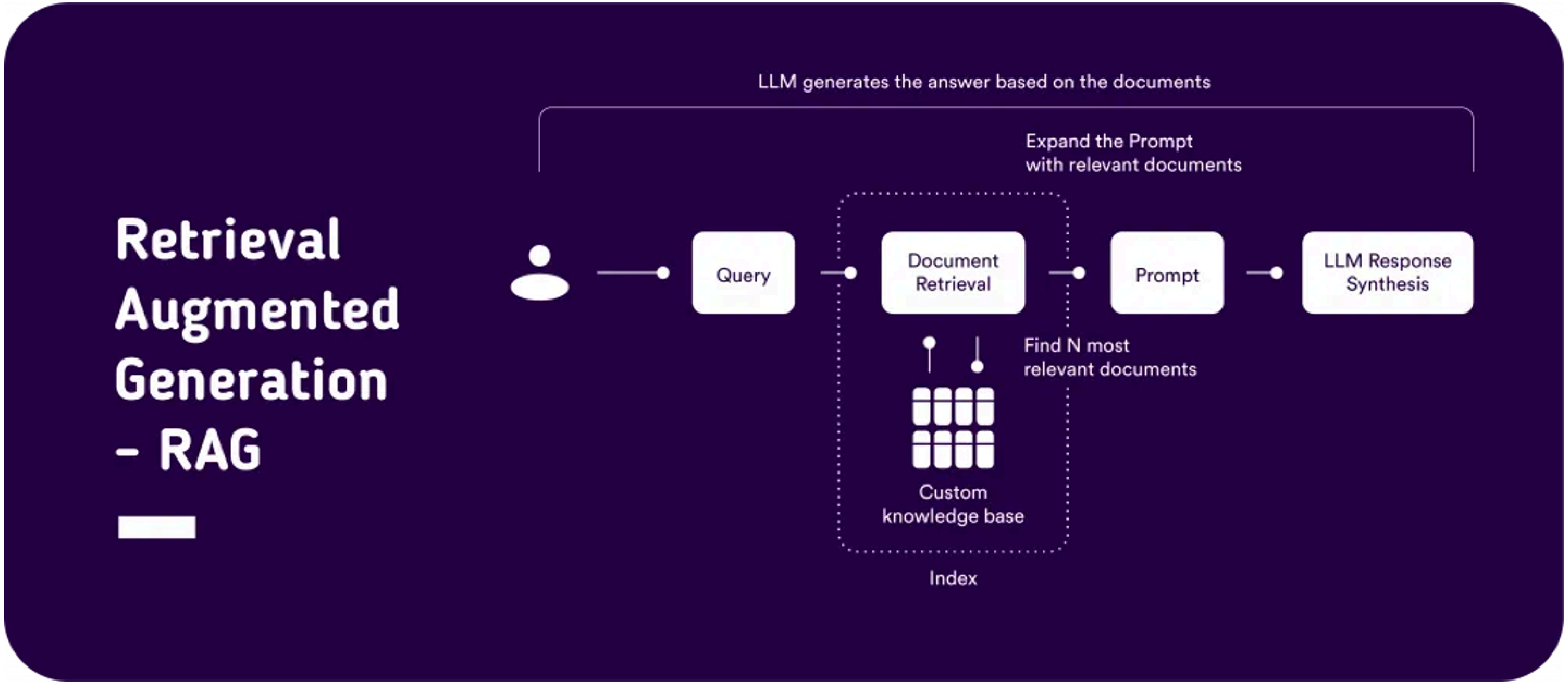
-> For example, a retailer could develop a chatbot that's prepared to answer user questions about delivery and returns policies.



Source

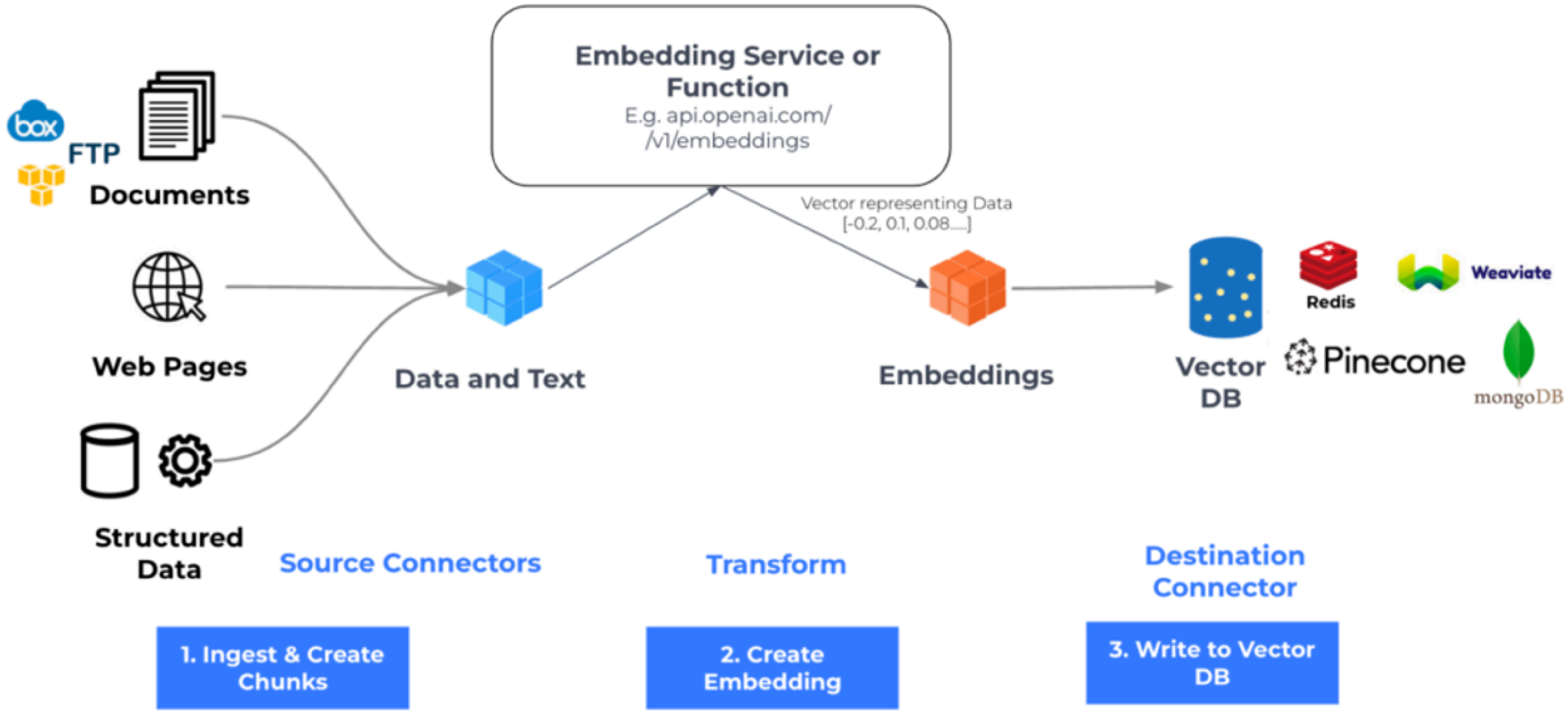
INTRODUCTION TO RAG

General workflow



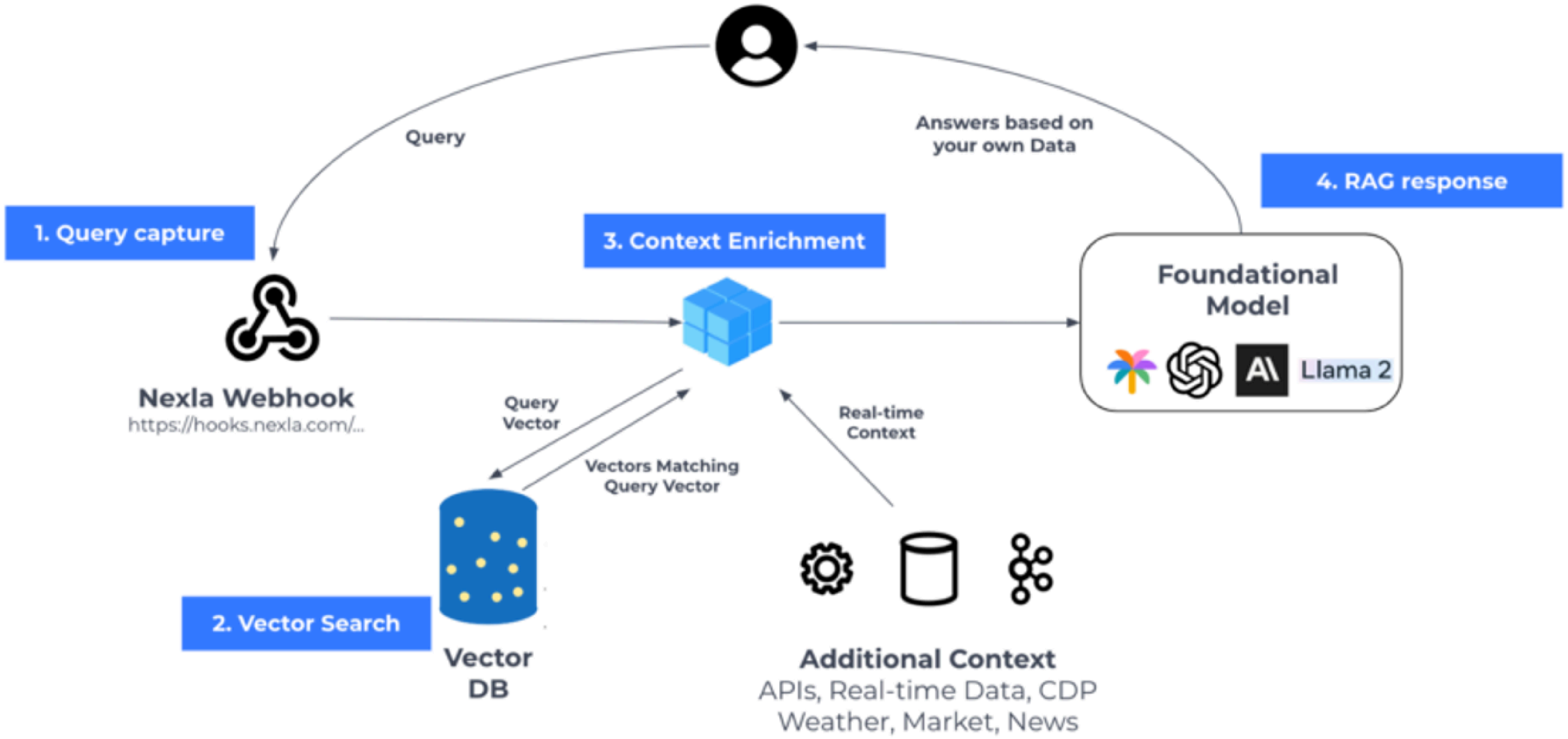
INTRODUCTION TO RAG

Step 1: Store vector embeddings into vector databases as knowledge



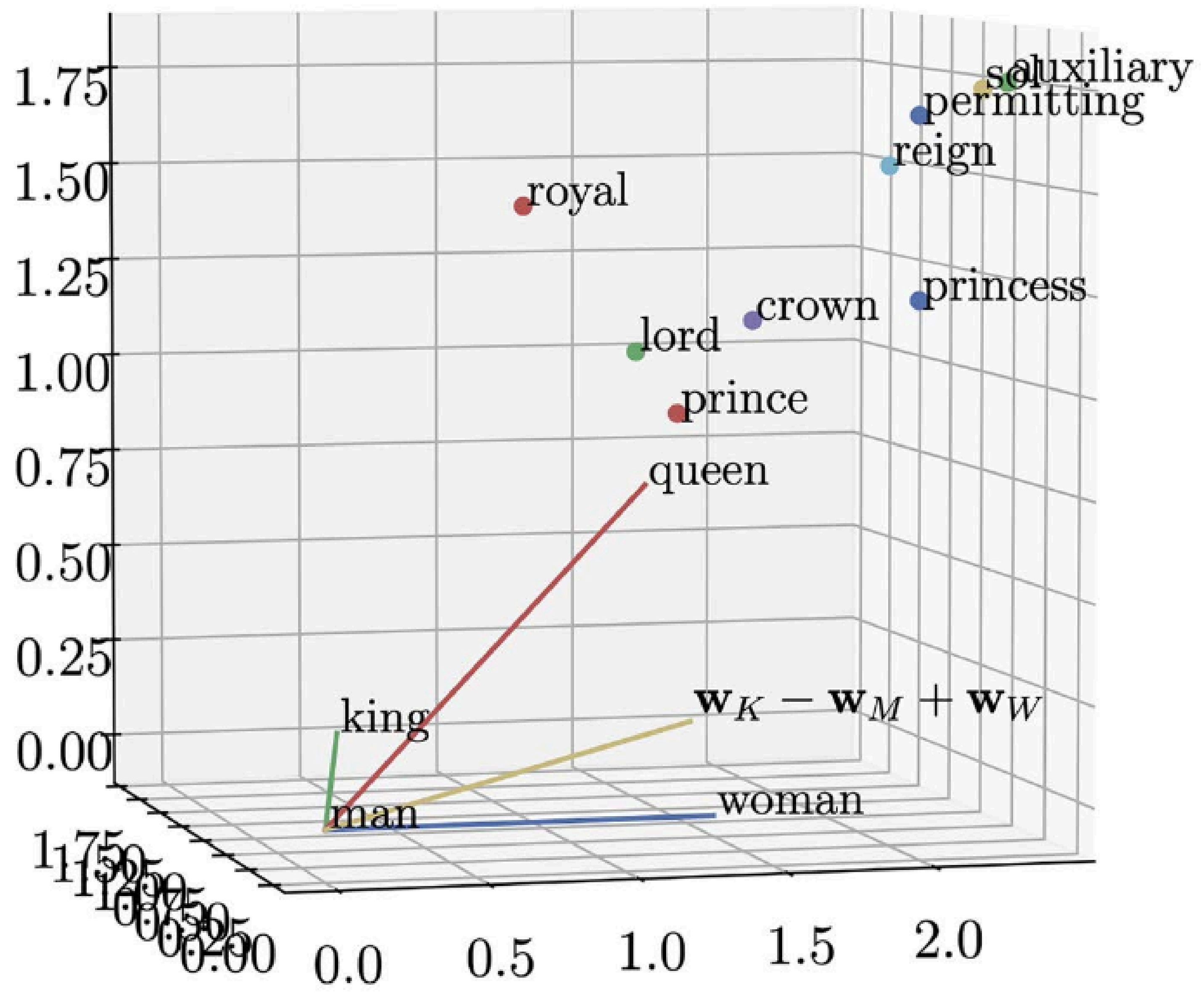
INTRODUCTION TO RAG

Step 2: Information retrieval and answer generation



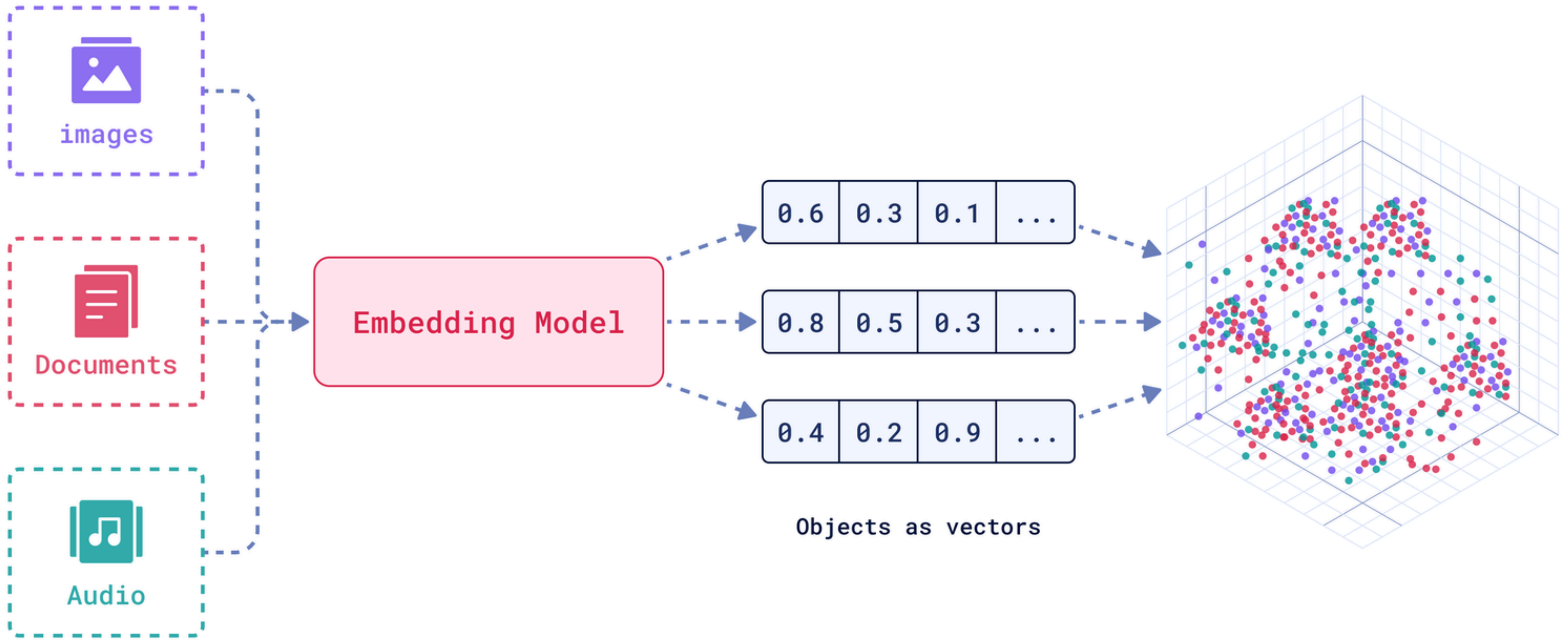
VECTOR EMBEDDINGS

- An embedding is a numerical representation of data in vector format
- An embedding can be a piece of content, such as a word, sentence, or image, and maps it into a multi-dimensional vector space.
- If we convert text into embeddings, the result vectors encapsulate its semantic meaning while discarding irrelevant details as much as possible.



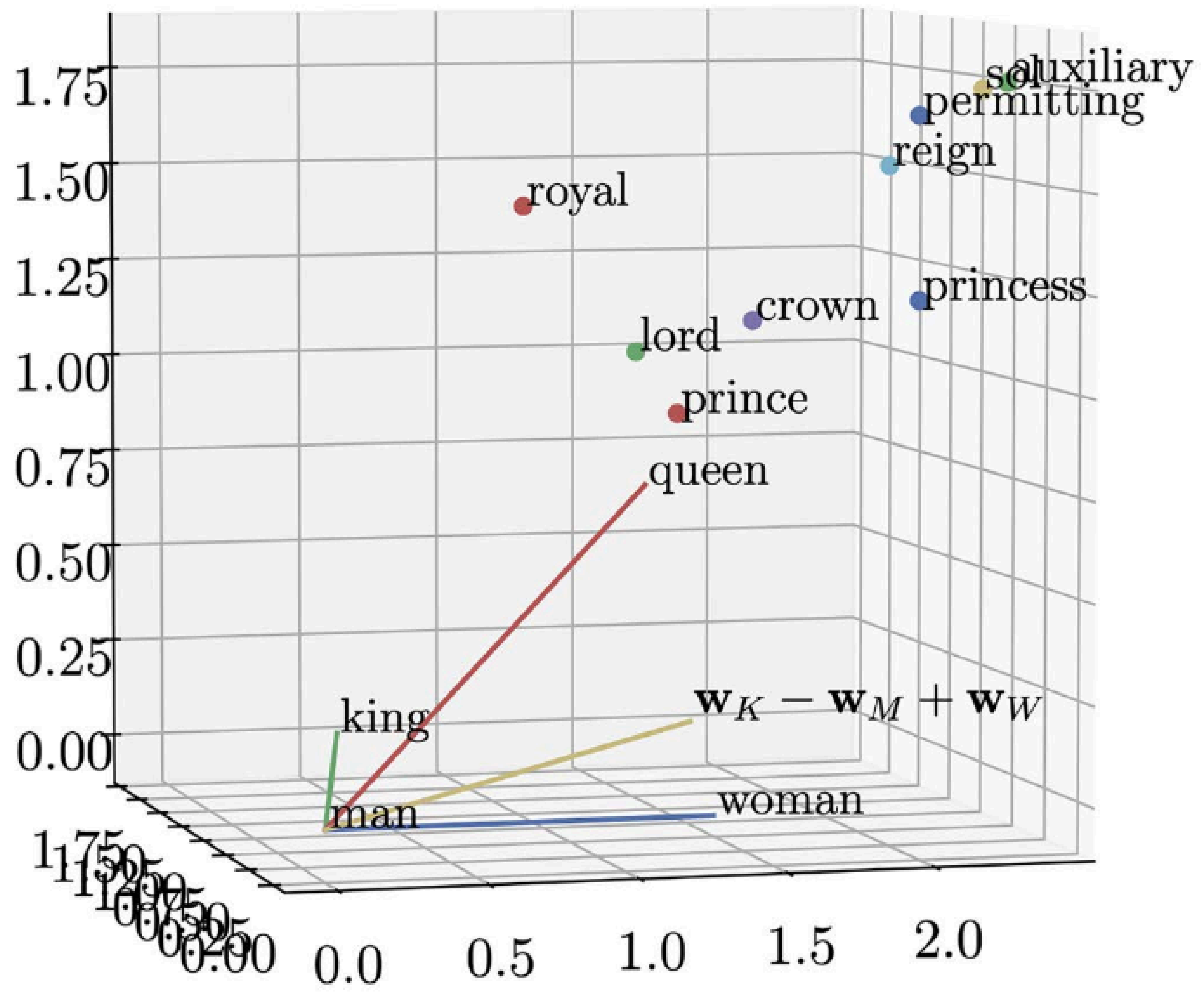
VECTOR EMBEDDINGS

- Today embeddings are usually generated by transformer-based models.
- These models are trained on large text datasets to learn concepts and relationships from languages.
- Transformers can capture the contextual meaning, semantics and order of words.
- Embeddings enable mathematical operations and can be inputs for other ML models.



VECTOR EMBEDDINGS

- Calculating distances between embeddings enables search via similarity scoring.
- The distance between two embeddings indicates the semantic similarity between the corresponding concepts (the original content).
- We can perform simple vector arithmetic with these vectors
- **For example**
    - the vector for king minus man plus the vector for woman gives us a vector that comes close to queen.



VECTOR EMBEDDINGS

Calculating similarity between embeddings:

- Generate embeddings
- Store vectors in a matrix
- Calculate the euclidean distance between each vector

Example result:

- A cat and a dog are indeed closer to an animal than to a computer.

```
from scipy.spatial.distance import pdist, squareform
import numpy as np
import pandas as pd
X = np.array(doc_vectors)
dists = squareform(pdist(X))
```

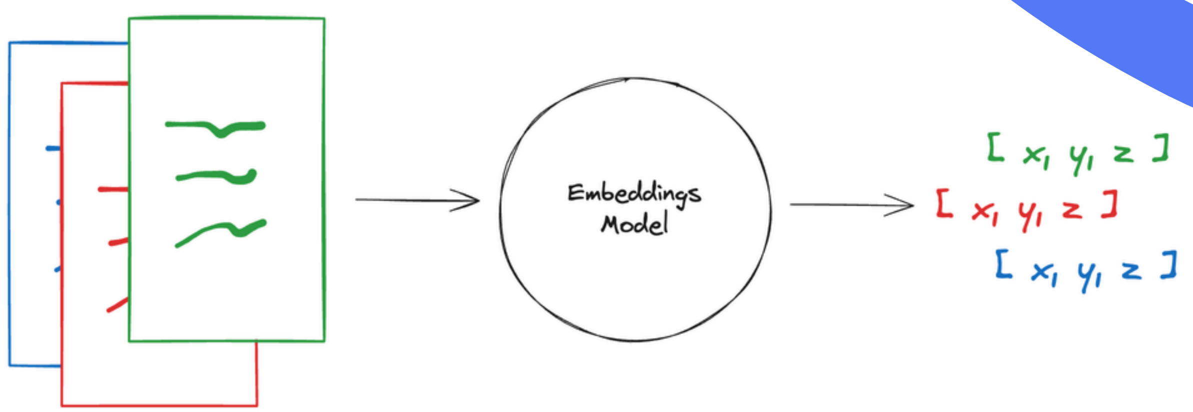
	cat	dog	computer	animal
cat	0.000000	0.522352	0.575285	0.521214
dog	0.522352	0.000000	0.581203	0.478794
computer	0.575285	0.581203	0.000000	0.591435
animal	0.521214	0.478794	0.591435	0.000000

VECTOR EMBEDDINGS

Generate embedding vectors with langchain.

1. Load Embedding Model

```
from langchain_huggingface import HuggingFaceEmbeddings

embeddings_model = HuggingFaceEmbeddings(model_name="sentence-transformers/all-mpnet-base-v2")
```



2. Generate Embeddings

`embed_documents`

Embed list of texts

Use `.embed_documents` to embed a list of strings, recovering a list of embeddings:

```
embeddings = embeddings_model.embed_documents(
    [
        "Hi there!",
        "Oh, hello!",
        "What's your name?",
        "My friends call me World",
        "Hello World!"
    ]
)
len(embeddings), len(embeddings[0])
```

```
(5, 1536)
```

`embed_query`

Embed single query

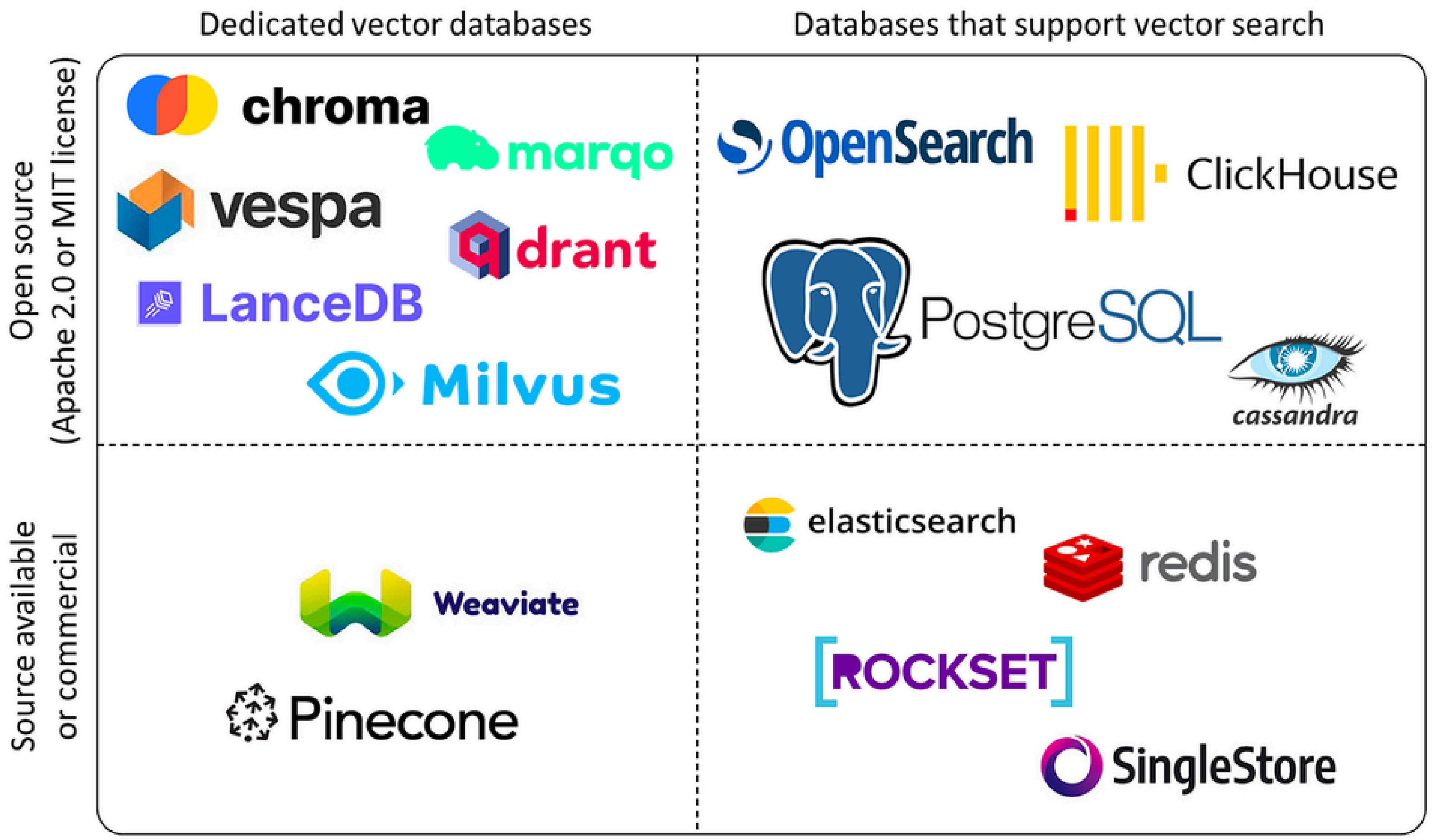
Use `.embed_query` to embed a single piece of text (e.g., for the purpose of comparing to other embedded pieces of texts).

```
embedded_query = embeddings_model.embed_query("What was the name mentioned in the conversation?")
embedded_query[:5]
```

```
[0.0053587136790156364,
 -0.0004999046213924885,
 0.038883671164512634,
 -0.0030010775662958 62,
 -0.009008182212710 38]
```

VECTOR DATABASES

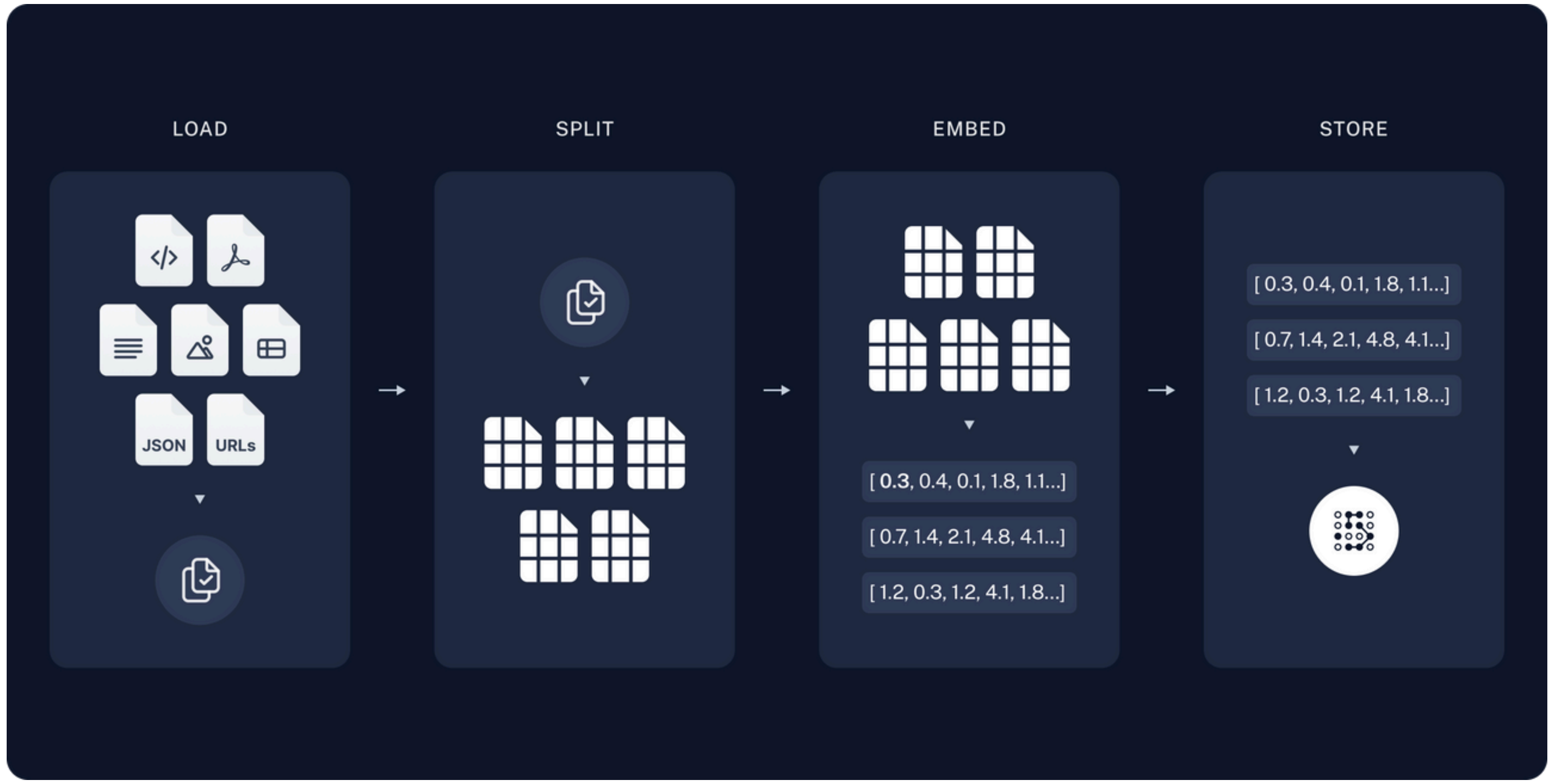
- Vector databases refer to the mechanism for storing and retrieving vector embeddings.
- Database solutions specifically designed for efficient storage and retrieval of vectors.
- Vector storage plays a crucial role in how vector embeddings are retrieved for various applications.
- There many different solutions on the market (commercial, open source, ...)



VECTOR DATABASES

Typical data integration workflow:

1. Collect and load unstructured data (e.g., PDFs).
2. Split data into smaller, topic-specific chunks (model input size limitations; improve search accuracy).
3. Generate embedding vectors for chunks using a embedding model.
4. Store vector embeddings and original text in the database index.



VECTOR DATABASES

Data integration workflow with langchain.

1. Load and split data in chunks

```
from langchain_text_splitters import RecursiveCharacterTextSplitter

# Load example document
with open("state_of_the_union.txt") as f:
    state_of_the_union = f.read()

text_splitter = RecursiveCharacterTextSplitter(
    # Set a really small chunk size, just to show.
    chunk_size=100,
    chunk_overlap=20,
    length_function=len,
    is_separator_regex=False,
)
texts = text_splitter.create_documents([state_of_the_union])
```

2. Load Embedding Model

```
from langchain_huggingface import HuggingFaceEmbeddings

embeddings_model = HuggingFaceEmbeddings(model_name="sentence-transformers/all-mpnet-base-v2")
```

3. Define vector database and pass embedding model

```
from langchain_chroma import Chroma

vector_store = Chroma(
    collection_name="example_collection",
    embedding_function=embeddings,
    persist_directory="./chroma_langchain_db",
)
```

3. Generate embedding vectors from document chunks and store them into vector database index.

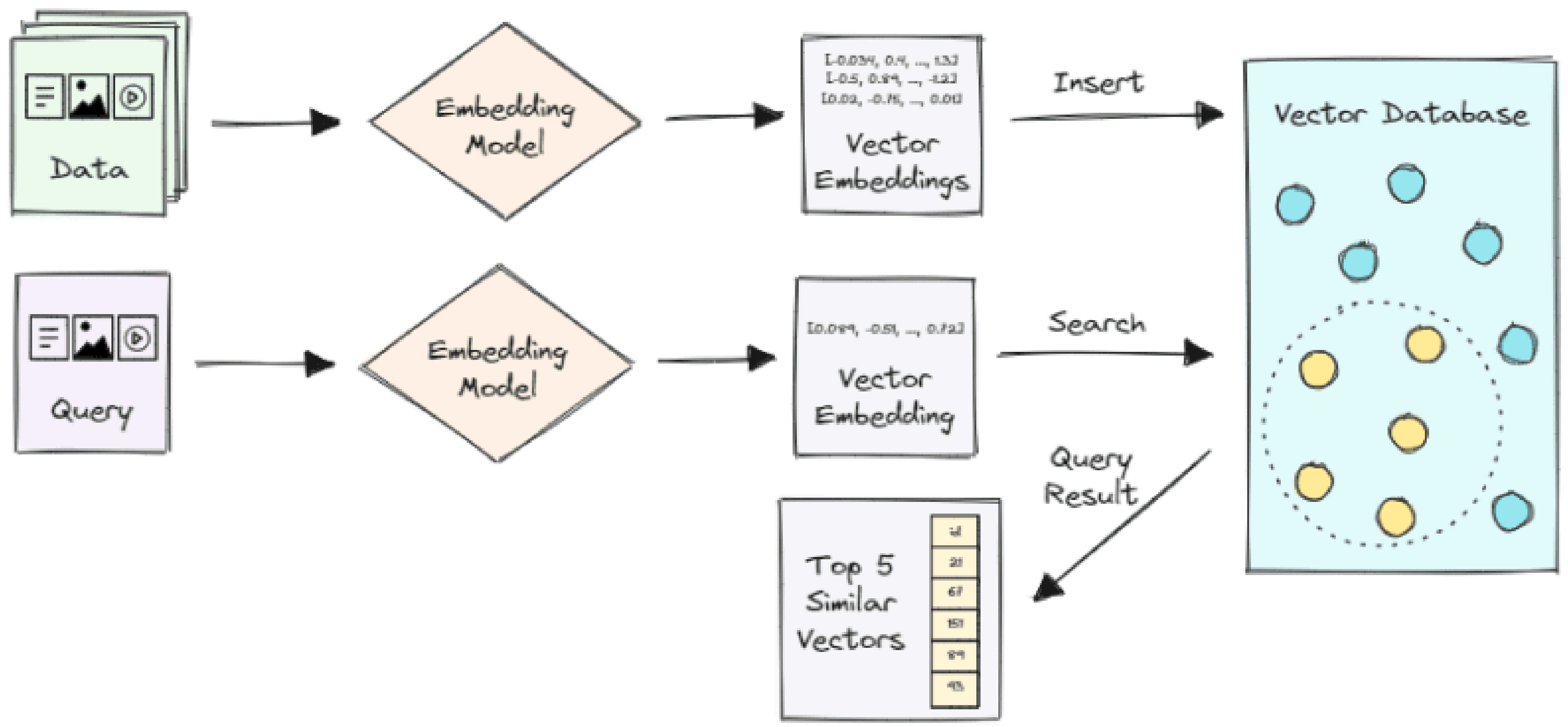
```
document_8 = Document(
    page_content="LangGraph is the best framework for building stateful, agentic applications!",
    metadata={"source": "tweet"},
    id=8,
)

document_9 = Document(
    page_content="The stock market is down 500 points today due to fears of a recession.",
    metadata={"source": "news"},
    id=9,
)

document_10 = Document(
    page_content="I have a bad feeling I am going to get deleted :(",
    metadata={"source": "tweet"},
    id=10,
)

documents = [
    document_1,
    document_2,
    document_3,
    document_4,
    document_5,
    document_6,
    document_7,
    document_8,
    document_9,
    document_10,
]
uuids = [str(uuid4()) for _ in range(len(documents))]

vector_store.add_documents(documents=documents, ids=uuids)
```

RETRIEVAL (VECTOR SEARCH)

In RAG, the goal is to retrieve the most similar vectors to a given query and dynamically pass that information to the answer generation process —this method is known as vector search.

General Process:

1. Generate dynamically a embedding vector from the query
2. Distances between vectors are computed using metrics like cosine similarity or euclidean distance.
3. Based on the distances query vectors are compared to all vectors in the collection; smaller distances indicate higher similarity.
4. Retrieve the k-most similar documents from the vector database index.



RETRIEVAL (VECTOR SEARCH)

Perform vector search in langchain.

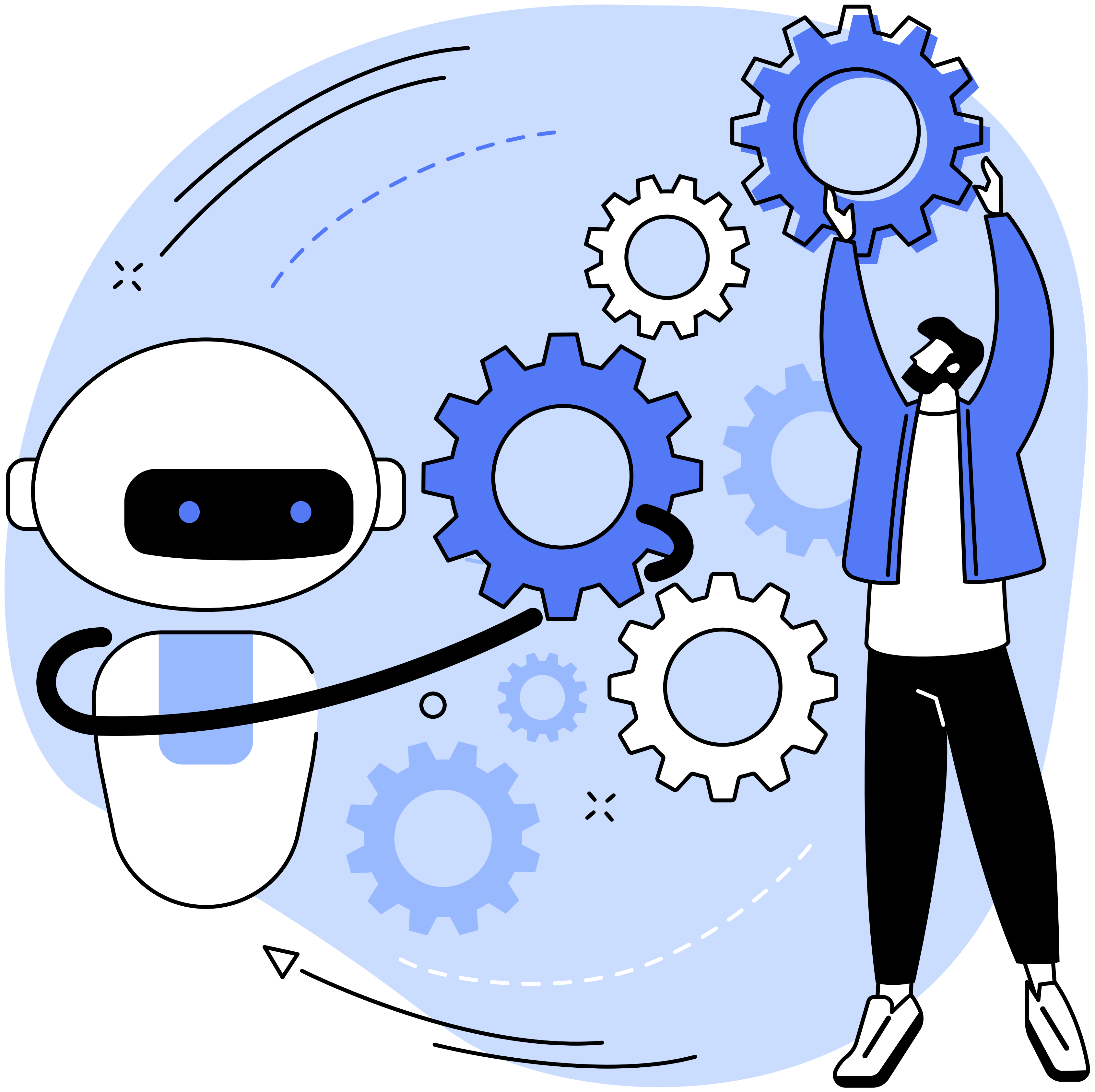
1.Similarity search

- **Generate the query's embedding vector.**
- **Compute similarities using a distance metric.**
- **Retrieve documents with the smallest distances.**
- **Return the top-k documents.**

```python
results = vector_store.similarity_search(
    "LangChain provides abstractions to make working with LLMs easy",
    k=2,
    filter={"source": "tweet"},
)
for res in results:
    print(f"* {res.page_content} [{res.metadata}]")
```

2. Retrieve the k=2 most similar documents from the vector database

```
* Building an exciting new project with LangChain - come check it out! [{'source': 'tweet'}]
* LangGraph is the best framework for building stateful, agentic applications! [{'source': 'tweet'}]
```



IT'S YOUR TURN