**Long Chain Models**

here’s the **clean LangChain map** exactly in the buckets you asked:

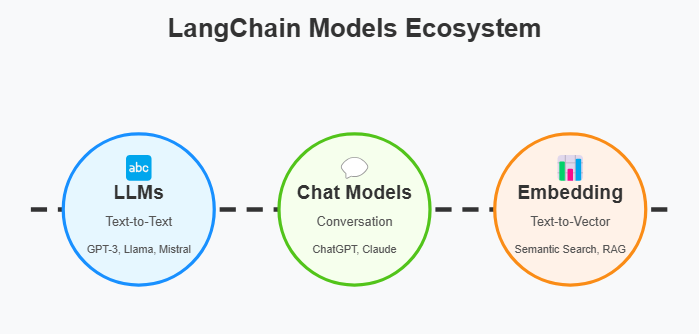
**1) Models**

* **Language models -> LLMs / Chat models**: ChatOpenAI, ChatAnthropic, local LLMs (HF).
* **Embeddings**: OpenAIEmbeddings, HuggingFaceEmbeddings.
* (Optional) **Other modalities**: vision, audio models (varies by provider).

**What are Models in LangChain?**

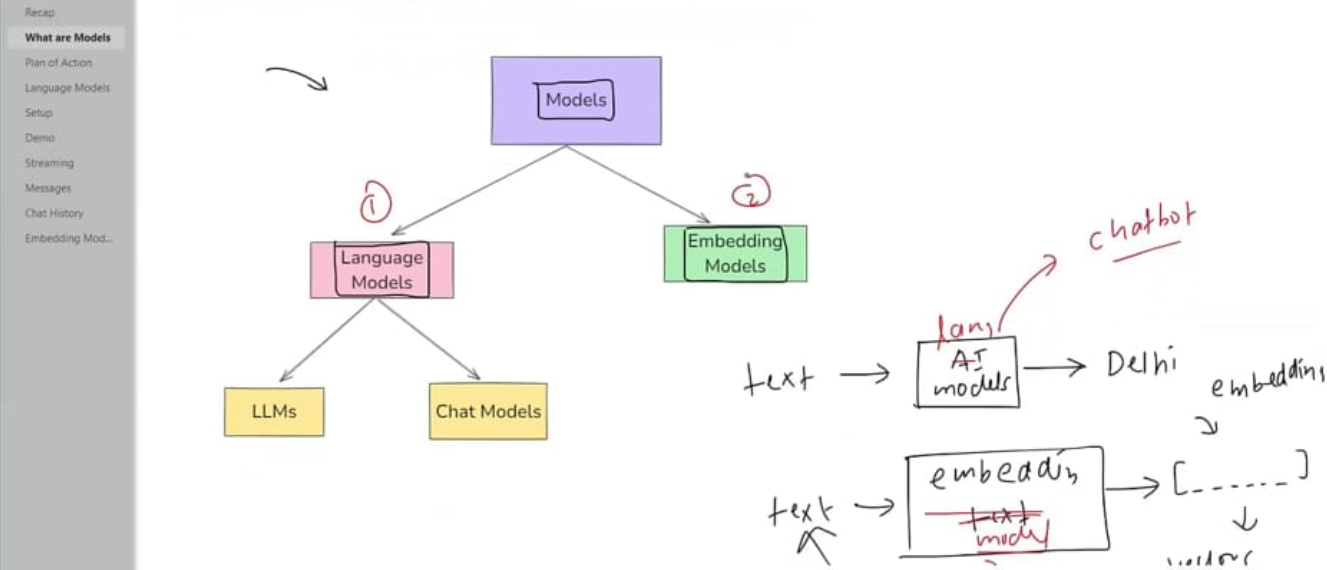
In LangChain, **Models are the core AI components** that process user inputs and generate intelligent outputs. It is designed to facilitate interactions with various language models and embedding models.

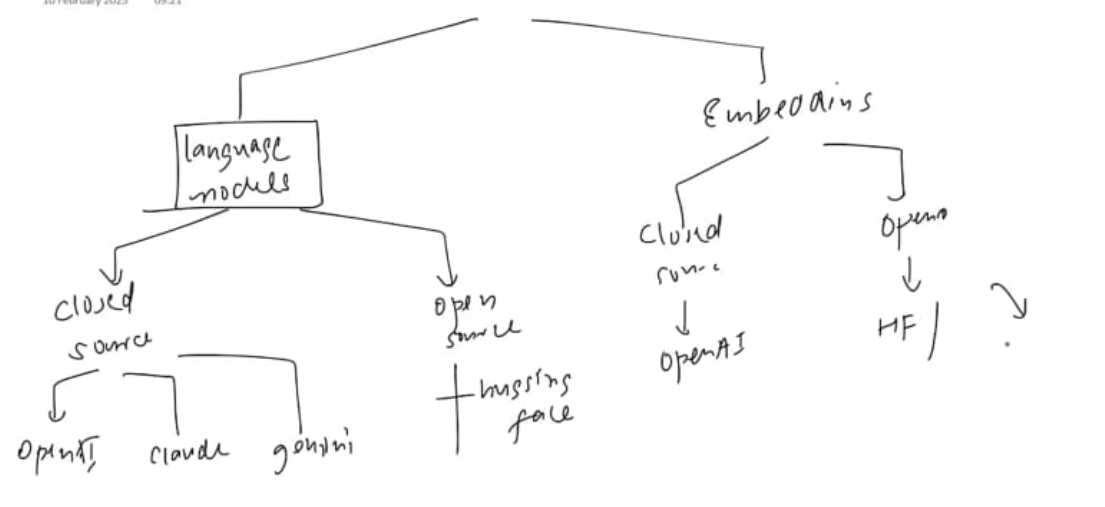
It abstracts the complexity of working directly with different LLMs, Chat Models, and embedding models, providing a uniform interface to communicate with them. This makes it easier to build applications that rely on AI-generated text, text embedding for similarity search, and retrieval-augmented generation (RAG).

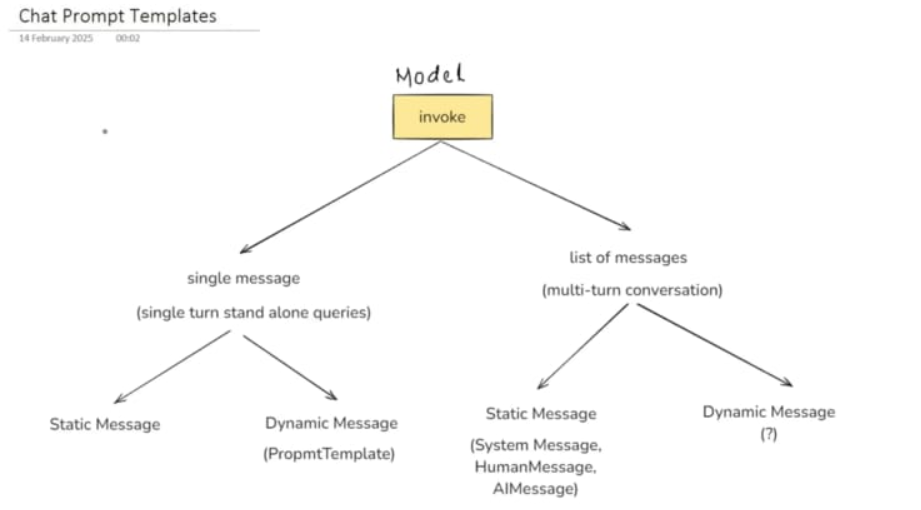


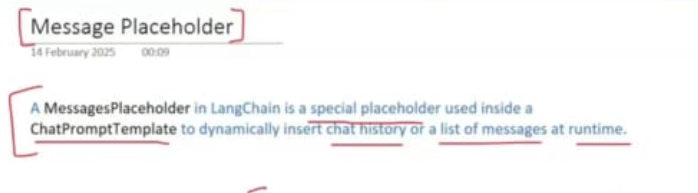
LangChain supports multiple types of models, including:

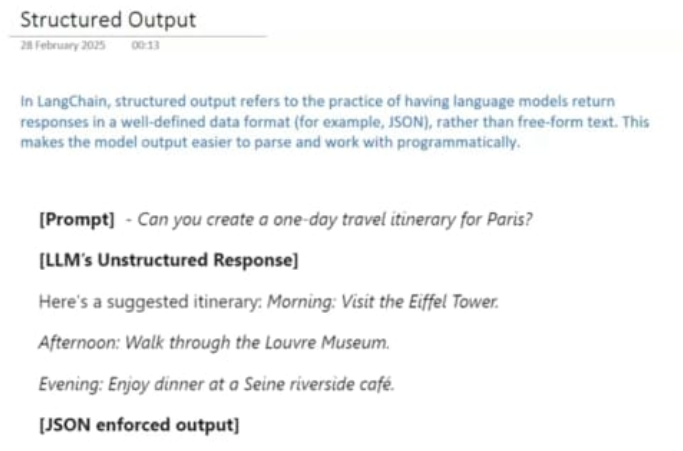
* **Language Models (LLMs)**: Generate text (e.g., OpenAI’s GPT-4, Hugging Face models).
* **Chat Models**: Specialized for multi-turn conversations (e.g., ChatGPT, Claude).
* **Embedding Models**: Convert text into vector representations for semantic search (e.g., OpenAI’s Embeddings, Hugging Face Transformers).

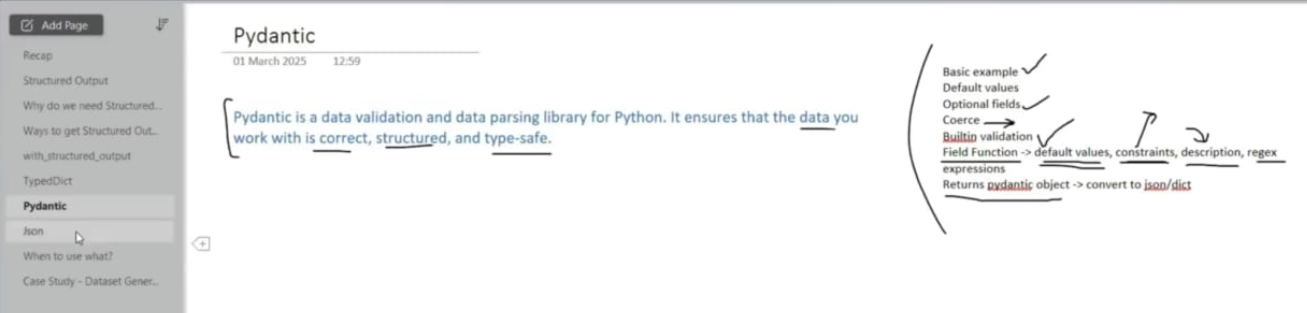


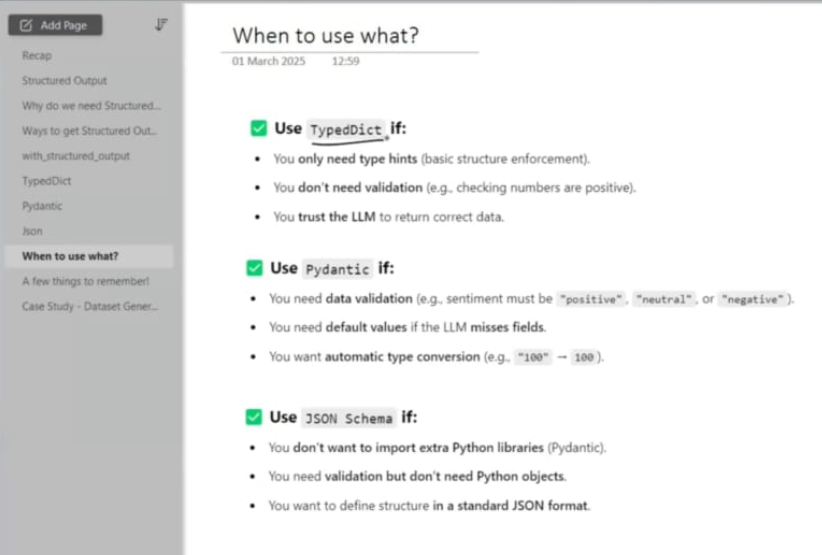


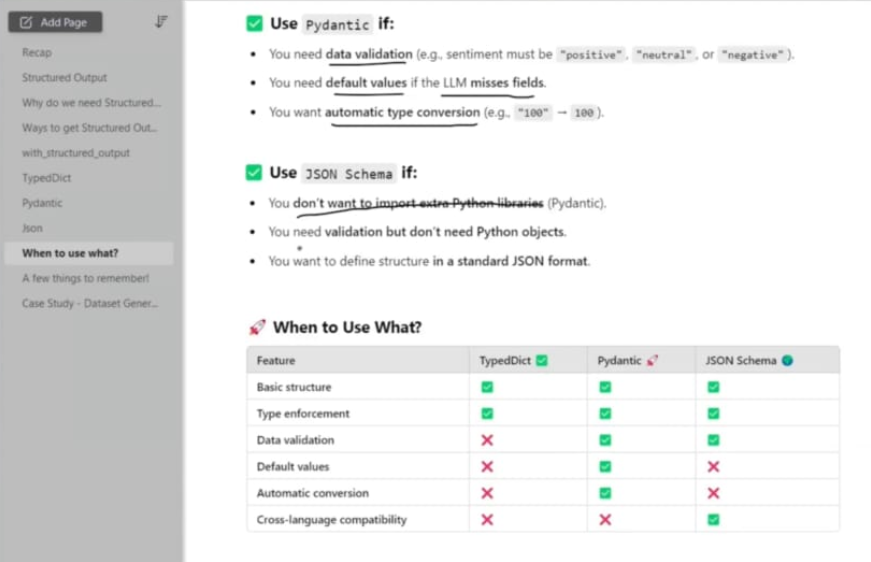


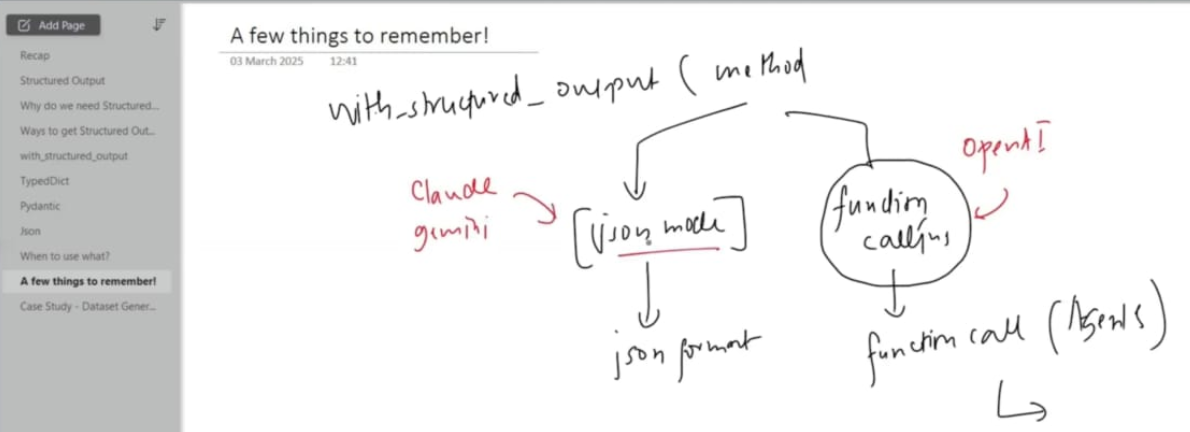










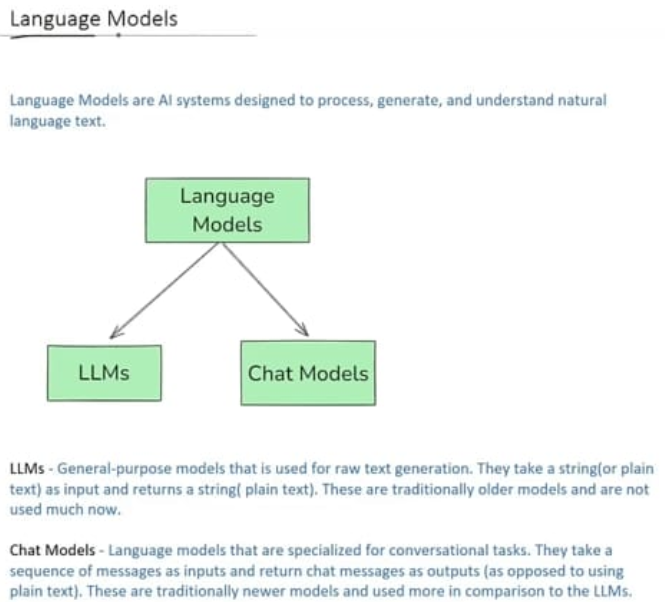


**🚀Language Models**

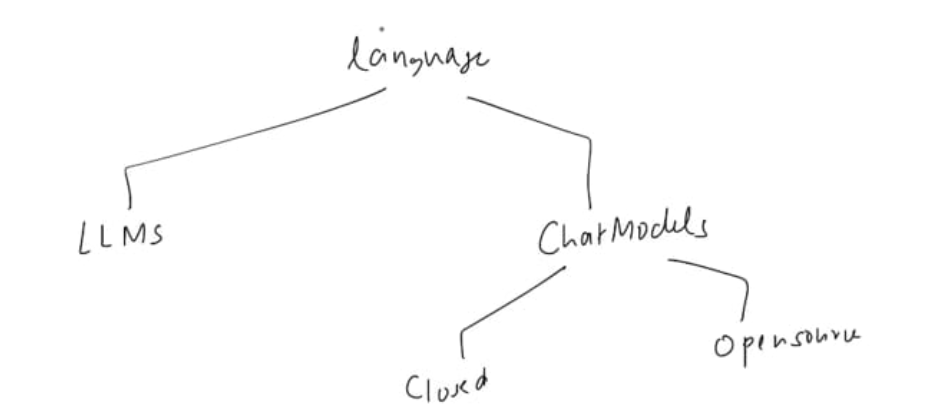
**🌐 What are Language Models?**

Language Models are AI systems designed to process, generate, and understand natural language text. It takes text input and generate text output. They are used for a wide range of tasks, including:

* Text generation (story writing, email drafting)
* Language translation
* Question answering
* Text classification







**Language Models are of two types:**

* LLMs
* Chat Models

**💬LLMs :**

LLMs are general-purpose models that are used for raw text generation. They take a string(or plain text) as input and return a string(plain text). These are traditionally older models and are not used much now.

**✅ Supported LLMs in LangChain**

* OpenAI (GPT-3.5, GPT-4)
* Hugging Face Transformers
* Anthropic (Claude)
* Ollama (Local Models)
* Custom Models (via API)

**✅ Using Language Models in LangChain**

**🚦 Example: Using OpenAI’s GPT-4**

from langchain\_openai import OpenAI  
from dotenv import load\_dotenv  
  
load\_dotenv()  
  
llm = OpenAI(model='gpt-3.5-turbo-instruct')  
  
result = llm.invoke("Explain LangChain in simple terms.")  
  
print(result)

NOTE: LLMs are old, langchain 0.3 is also saying to use Chat Models for chatbots that are specially built for conversational tasks

**💬 Chat Models — Making Conversations Context-Aware**

**🌐 What are Chat Models?**

Chat Models are specialized LLMs designed for **multi-turn conversations**, where they can maintain context across interactions. Examples include:

In **LangChain**, **chat models** are a type of Large Language Model (LLM) interface designed to work with **multi-turn, structured conversations** instead of plain, single-turn text completions.

Think of them as LLMs that understand **"messages"** rather than just **raw strings**.  
They handle context, roles (system, user, assistant), and can keep track of conversation history more naturally

**Examples of Chat Models in LangChain**

LangChain supports chat-based APIs from different providers:

| Provider | Class Name | Notes |
| --- | --- | --- |
| OpenAI | ChatOpenAI | Works with GPT-3.5, GPT-4 |
| Anthropic | ChatAnthropic | Works with Claude models |
| Azure | AzureChatOpenAI | GPT models via Azure |
| Google Vertex AI | ChatVertexAI | PaLM / Gemini models |
| Ollama | ChatOllama | Local LLaMA or Mistral models |
| HuggingFace | ChatHuggingFace | Some HF chat LLMs |

**Messages Types based on roles in chat models:**

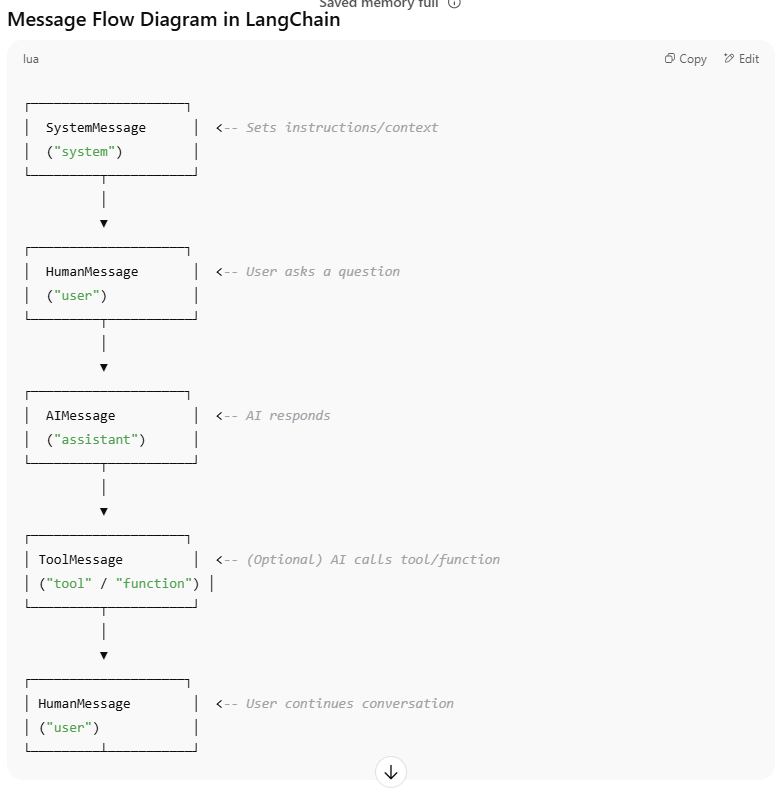
In **LangChain chat models**, conversations aren’t just raw strings — they’re **structured as messages**, and each message has a **role** that represents *who* is speaking.

These message types are inspired by **OpenAI’s chat format** but are standardized in LangChain so they work across providers.

**1️⃣ The Main Message Types**

LangChain defines the following core message classes:

| **Message Class** | **Role** | **Purpose** |
| --- | --- | --- |
| HumanMessage | "human" | Represents user input or anything coming from the human side of the conversation. |
| AIMessage | "ai" | Represents responses generated by the AI/LLM. |
| SystemMessage | "system" | Sets context, instructions, or behavior guidelines for the AI. |
| FunctionMessage | "function" | Used when calling a tool/function and returning results to the AI. |
| ToolMessage | "tool" | Similar to FunctionMessage, used when integrating LangChain Tools. |
| ChatMessage | "custom role" | Generic version where you can specify any custom role (e.g., "moderator", "coach"). |

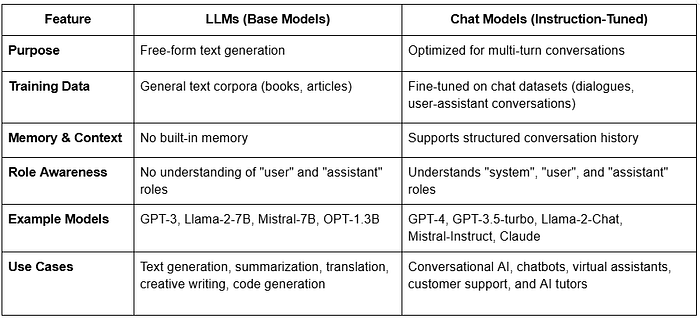


**2. Why Use Chat Models?**

✅ **Role-based prompts** — Distinguishes instructions (system) from user input.  
✅ **Multi-turn conversation** — Maintains context across messages.  
✅ **Better control** — You can insert instructions mid-conversation.  
✅ **Cleaner formatting** — No need to manually concatenate context and user input.

**✅ How Chat Models Differ from Basic LLMs**

Press enter or click to view image in full size



**✅ Using Chat Models in LangChain**

1. **OpenAI GPT-4**

First, generatean OpenAI access key from [OpenAI Platform](https://platform.openai.com/settings/organization/api-keys)

Set the access key in the .env file

OPENAI\_API\_KEY="YOUR\_API\_KEY"

from langchain\_openai import ChatOpenAI  
from dotenv import load\_dotenv  
  
load\_dotenv()  
  
model = ChatOpenAI(model='gpt-4', temperature=1.5, max\_completion\_tokens=10)  
  
result = model.invoke("Write a 5 line poem on cricket")  
  
print(result.content)

2. **Anthropic Claude**

First, generate access key from [Anthropic.](https://console.anthropic.com/)

Set the access key in the .env file

ANTHROPIC\_API\_KEY="YOUR\_API\_KEY"

from langchain\_anthropic import ChatAnthropic  
from dotenv import load\_dotenv  
  
load\_dotenv()  
  
model = ChatAnthropic(model='claude-3-5-sonnet-20241022')  
  
result = model.invoke('What is the capital of India')  
  
print(result.content)

3.**Google Gemini**

First, generatean API key from [Google AI Studio](https://aistudio.google.com/apikey)

Set key in .env file

GOOGLE\_API\_KEY="YOUR\_API\_KEY"

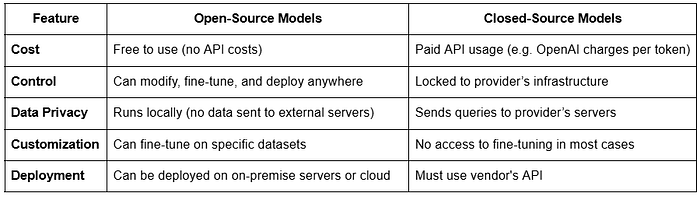
from langchain\_google\_genai import ChatGoogleGenerativeAI  
from dotenv import load\_dotenv  
  
load\_dotenv()  
  
model = ChatGoogleGenerativeAI(model="gemini-1.5-pro")  
result = model.invoke("What is the capital of France?")  
print(result.content)

4. **Open-Source Models (Local Models) :**

Open-source models are freely available AI models that can be downloaded, modified, fine-tuned, and deployed without restrictions from a central provider. Unlike closed-source models such as Open AI’s GPT-4, Anthropic’s Claude, and Google’s Gemini, open-source models allow full control and customization.

**✅ Why Use Open-Source Models?**

Press enter or click to view image in full size



**✅ Supported Open-Source Models in LangChain**

* LLAMA 2 (Meta)
* Falcon (Hugging Face)
* GPT-NeoX (EleutherAI)
* MPT (MosaicML)

**I) Hugging Face Models using API :**

First, generatean access token using [Hugging Face](https://huggingface.co/settings/tokens)

Set key in .env file

HUGGINGFACEHUB\_API\_TOKEN = "YOUR\_API\_KEY"

Copy the model name and use it in repo\_id

from langchain\_huggingface import ChatHuggingFace, HuggingFaceEndpoint  
from dotenv import load\_dotenv  
import os  
  
load\_dotenv()  
  
llm = HuggingFaceEndpoint(  
 repo\_id="TinyLlama/TinyLlama-1.1B-Chat-v1.0",  
 task="text-generation"  
)  
  
model = ChatHuggingFace(llm=llm)  
  
result = model.invoke("What is the capital of India")  
  
print(result.content)

**II) Hugging Face Models Local (downloaded) :**

We can use models that are present in [Hugging Face](https://huggingface.co/models)

We just need to copy the model name and use it in model\_id, first, it will download it and then use it

from langchain\_huggingface import ChatHuggingFace, HuggingFacePipeline  
import os  
  
#To store downloaded models in specific directory  
os.environ['HF\_HOME'] = 'D:\learning\llm\_downloaded\hugging\_face\_cache'  
  
llm = HuggingFacePipeline.from\_model\_id(  
 model\_id='TinyLlama/TinyLlama-1.1B-Chat-v1.0',  
 task='text-generation',  
 pipeline\_kwargs=dict(  
 temperature=0.5,  
 max\_new\_tokens=100  
 )  
)  
model = ChatHuggingFace(llm=llm)  
  
result = model.invoke("What is the capital of India")  
  
print(result.content)

III) **Ollama**

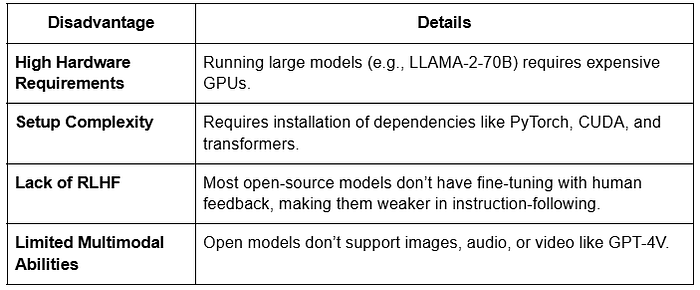
Most simplest way to use a local model is Ollama, here we can use all the models available in [Ollama](https://ollama.com/search) by just changing the model name.

You can check my other [blog](https://medium.com/@mishra.sagar25/exploring-agent-ai-the-future-of-autonomous-systems-with-autogen-and-ollama-running-locally-e7c88bd21f01) for step by step installation of Ollama.

from langchain\_ollama import OllamaLLM  
  
# Using mistral via Ollama (Local)  
model = OllamaLLM(model = "mistral")  
  
response = model.invoke("What is the capital of India")  
print(response)

**Disadvantages of using local models:**

Press enter or click to view image in full size



**📊 Embedding Models — Making Text Searchable**

**🌐 What are Embedding Models?**

**What is an Embedding Model?**

An **embedding model** is a type of machine learning model that converts **input data (text, image, audio, etc.)** into a **vector representation** — essentially, a list of numbers (floating-point values).

* These vectors are called **embeddings**.
* The key idea: **semantically similar inputs have similar vector representations**.
* This allows machines to reason about meaning, not just raw words or pixels.

Example:

"cat" → [0.21, -0.14, 0.93, ...]

"dog" → [0.19, -0.10, 0.91, ...] # Close to "cat"

"car" → [-0.45, 0.80, -0.12, ...] # Far from "cat" and "dog"

**2️⃣ Why Use Embeddings?**

They allow us to do **semantic search, clustering, recommendation, and classification** without hard-coded rules.

For example:

* If you embed a search query and all documents, you can retrieve the **most semantically similar documents** even if they don’t share exact words.
* In chatbots, embeddings are used for **retrieval-augmented generation (RAG)**.

**3️⃣ How Are Embeddings Generated?**

Internally:

1. **Tokenization** – Convert input (e.g., text) into tokens (words, subwords, or characters).
2. **Mapping to vectors** – Each token gets a dense vector from an **embedding layer**.
3. **Contextualization (for large models)** – Transformer layers refine token embeddings using context.
4. **Pooling** – The token embeddings are aggregated into a single vector (e.g., by averaging, or using the [CLS] token in BERT).
5. **Normalization** – Often, vectors are normalized so similarity can be measured with cosine similarity.

**4️⃣ Similarity Metrics**

Once you have embeddings, you can compare them:

* **Cosine similarity** → Measures the angle between vectors (most common for semantic similarity).
* **Euclidean distance** → Measures straight-line distance in vector space.
* **Dot product** → Measures projection of one vector onto another.

**5️⃣ Types of Embedding Models**

Depending on the **input modality**:

| **Type** | **Example Models** | **Use Cases** |
| --- | --- | --- |
| **Text embeddings** | OpenAI text-embedding-3-small, HuggingFace all-MiniLM-L6-v2, Cohere embed-multilingual | Semantic search, clustering, RAG |
| **Image embeddings** | CLIP, OpenAI clip-vit, Google Vision Embeddings | Reverse image search, similarity |
| **Audio embeddings** | OpenAI whisper + projection layers, SpeechBrain embeddings | Speaker identification, audio search |
| **Multimodal embeddings** | CLIP, BLIP, Florence | Linking images ↔ text |

**6️⃣ Embedding Model Providers**

Common providers in production:

* **OpenAI** → text-embedding-3-small, text-embedding-3-large
* **HuggingFace** → Sentence Transformers (e.g., all-MiniLM-L6-v2, multi-qa-mpnet-base-dot-v1)
* **Cohere** → embed-english-v3.0, embed-multilingual-v3.0
* **Google Vertex AI** → textembedding-gecko
* **AWS Bedrock** → Titan embeddings
* **Mistral AI** → Embedding API
* **Ollama** → Local LLaMA or Mistral models fine-tuned for embeddings

**1️⃣ Commercial Cloud AI Providers**

| **Provider** | **Popular Embedding Models** | **Key Features** |
| --- | --- | --- |
| **OpenAI** | text-embedding-ada-002, text-embedding-3-large, text-embedding-3-small | Very high quality, optimized for semantic search, RAG, clustering. |
| **Anthropic** | (Currently focuses more on chat models; embeddings expected in future) | Ethical AI focus, can be integrated with LangChain. |
| **Azure OpenAI** | Same as OpenAI’s embeddings, hosted on Azure | Enterprise integration, compliance-ready. |
| **Google Cloud Vertex AI** | textembedding-gecko | Good multilingual support, optimized for Google’s ecosystem. |
| **AWS Bedrock** | Titan Embeddings (amazon.titan-embed-text-v1) | Fully managed, integrated with AWS services. |
| **Cohere** | embed-english-v3, embed-multilingual-v3 | High-quality embeddings for English & multilingual text. |
| **IBM Watsonx.ai** | Proprietary embeddings | Enterprise-level governance. |

**2️⃣ Open-Source Model Providers (Hosted or Self-Hosted)**

| **Provider** | **Popular Models** | **Notes** |
| --- | --- | --- |
| **Hugging Face** | sentence-transformers/all-MiniLM-L6-v2, multi-qa-mpnet-base-dot-v1, all-mpnet-base-v2 | Huge model hub; can run locally or via API. |
| **Mistral AI** | (Primarily LLMs now, but embeddings possible via adapters) | Open-weight friendly. |
| **Ollama** | Can run embedding-capable models like nomic-embed-text locally | No internet needed. |
| **nomic.ai** | nomic-embed-text-v1.5 | Specializes in high-performance embeddings; open weights available. |
| **BAAI** | bge-large-en, bge-m3 | State-of-the-art on many retrieval benchmarks. |
| **Snowflake Arctic** | Arctic Embeddings | Designed for enterprise RAG. |

**3️⃣ Specialized & Multimodal Embedding Providers**

| **Provider** | **Embedding Type** | **Example Models** |
| --- | --- | --- |
| **CLIP (OpenAI)** | Text & Image | clip-vit-base-patch32 |
| **Salesforce BLIP** | Text & Image | blip-image-captioning |
| **LAION** | Image & text embeddings | Used in Stable Diffusion pipelines |
| **OpenAI Audio** | Audio embeddings | From Whisper or future APIs |
| **Vespa.ai** | Text embeddings | Often used for large-scale search |

**4️⃣ Vector Databases With Built-in Embeddings**

Some vector DBs can auto-generate embeddings without you directly calling an embedding model API:

* **Pinecone (with Cohere/OpenAI integration)**
* **Weaviate (built-in text2vec-\* models)**
* **Milvus (integrates with Hugging Face & others)**
* **Qdrant (plugin-based embedding loaders)**

**7️⃣ Key Characteristics of Embeddings**

* **Dimensionality** – Number of values in the vector (e.g., 768, 1024, 1536).
* **Domain specificity** – Some models are trained for general knowledge, others for specific domains like code or medical.
* **Language coverage** – Monolingual vs multilingual models.
* **Context window** – How much input text they can handle.

**8️⃣ Limitations**

* **Static embeddings** (like word2vec) don’t handle polysemy well ("bank" = river bank vs financial bank).
* **Contextual embeddings** (like BERT) solve this, but can be computationally expensive.
* High-dimensional vectors require specialized **vector databases** (FAISS, Pinecone, Weaviate) for fast search.
* Meaning can still degrade if input is **out-of-distribution** (something model wasn’t trained on).

**OpenAI(...)?**

That’s because **OpenAI in LangChain is for the older completion-style API**, which defaults to OpenAI’s legacy models like text-davinci-003.

* If you don’t pass a model argument, it uses the **default model** (often text-davinci-003 unless overridden by your environment variables).
* It works with single text prompts (like "Summarize this"), not multi-message chat history.

Example:

from langchain\_openai import OpenAI

llm = OpenAI(temperature=0, model="text-davinci-003")

**2️⃣ What if you used ChatOpenAI instead?**

Yes — if your use case involves **chat-style** prompts (structured as messages), or you want to use modern OpenAI models like gpt-3.5-turbo or gpt-4o, you should use ChatOpenAI.

Example:

from langchain\_openai import ChatOpenAI

llm = ChatOpenAI(temperature=0, model="gpt-3.5-turbo")

**✅ Using Embedding Models in LangChain**

1. **Embedding query (String) using OpenAI**

from langchain\_openai import OpenAIEmbeddings  
from dotenv import load\_dotenv  
  
load\_dotenv()  
  
embedding = OpenAIEmbeddings(model='text-embedding-3-large', dimensions=32)  
  
result = model.invoke('What is the capital of India')  
  
print(result.content)

**2.** **Embedding Documents (List of String) using OpenAI**

from langchain\_openai import OpenAIEmbeddings  
from dotenv import load\_dotenv  
  
load\_dotenv()  
  
embedding = OpenAIEmbeddings(model='text-embedding-3-large', dimensions=32)  
  
documents = [  
 "Delhi is the capital of India",  
 "Kolkata is the capital of West Bengal",  
 "Paris is the capital of France"  
]  
  
result = embedding.embed\_documents(documents)  
  
print(str(result))

**3. Embedding Query using HuggingFace local**

from langchain\_huggingface import HuggingFaceEmbeddings  
  
embedding = HuggingFaceEmbeddings(model\_name='sentence-transformers/all-MiniLM-L6-v2')  
  
text = "Delhi is the capital of India"  
  
vector = embedding.embed\_query(text)  
  
print(str(vector))

**4. Embedding Document using HuggingFace local**

from langchain\_huggingface import HuggingFaceEmbeddings  
  
embedding = HuggingFaceEmbeddings(model\_name='sentence-transformers/all-MiniLM-L6-v2')  
  
documents = [  
 "Delhi is the capital of India",  
 "Kolkata is the capital of West Bengal",  
 "Paris is the capital of France"  
]  
  
vector = embedding.embed\_documents(documents)  
  
print(str(vector))

**✅ Real-World Use Case: Document Similarity with Embeddings**

Imagine you are building a **Customer Support Assistant** that helps users find the most relevant support article based on their query. The assistant needs to understand which document is most similar to the user’s question.

**🚦 Problem Without Embedding Models**

* If you only use keyword matching, the assistant may not understand user queries properly.
* For example, if a user asks **“How to reset my password?”**, it should match with a document titled **“Password Recovery Guide”**, even if the words don’t exactly match.

**✅ Solution: Using Embedding Models for Semantic Search**

Using **Hugging Face’s Sentence Transformers** with LangChain, we can achieve document similarity with high accuracy.

from langchain\_huggingface import HuggingFaceEmbeddings  
from dotenv import load\_dotenv  
from sklearn.metrics.pairwise import cosine\_similarity  
  
# Load environment variables (API keys)  
load\_dotenv()  
  
# Initialize Embedding Model  
embedding = HuggingFaceEmbeddings(model\_name='sentence-transformers/all-MiniLM-L6-v2')  
  
# Sample Documents (Support Articles)  
documents = [  
 "How to reset your password quickly and securely.",  
 "Troubleshooting login issues on your account.",  
 "Understanding account security and two-factor authentication.",  
 "How to delete your account permanently.",  
 "How to change your registered email address."  
]  
  
# User Query  
query = "I forgot my password, what should I do?"  
  
# Generate Document and Query Embeddings  
doc\_embeddings = embedding.embed\_documents(documents)  
query\_embedding = embedding.embed\_query(query)  
  
# Calculate Similarity using Cosine Similarity  
scores = cosine\_similarity([query\_embedding], doc\_embeddings)[0]  
index, score = sorted(list(enumerate(scores)), key=lambda x: x[1])[-1]  
  
# Display the most similar document  
print("User Query:", query)  
print("Most Relevant Document:", documents[index])  
print("Similarity Score:", score)

Output:

User Query: I forgot my password, what should I do?  
Most Relevant Document: How to reset your password quickly and securely.  
Similarity Score: 0.89

**✅How It Works:**

* The user query is converted into a **vector (embedding)**.
* Each document is also converted into a vector.
* The **Cosine Similarity** between the query vector and each document vector is calculated.
* The document with the highest similarity score is selected as the most relevant.

**✅ Why This is Powerful**

* It works even if the user’s query does not match the document text exactly (semantic search).
* Can be scaled to thousands of documents with efficient search.
* Can be extended to handle multi-lingual text (if the embedding model supports it).

**✅ More Practical Use Cases:**

* **Customer Support:** Finding the most relevant help article.
* **E-Learning Platforms:** Matching user questions to course content.
* **Legal Research:** Finding similar cases or documents.
* **Code Search:** Matching user queries to code snippets

**1) Language Models (LMs)**

**Definition**

A **Language Model** is an AI model that learns the statistical patterns of language to predict the next token (word, subword, or character) given a sequence of tokens.  
Think of it as:

“Given what I’ve seen so far, what is the most likely next word?”

**Core Purpose**

* Predict **next tokens** based on prior context.
* Generate text sequences that are **syntactically and semantically correct**.
* Work in both **generative** and **classification** contexts.

**How It Works**

1. **Training data** → Large corpus of text (Wikipedia, books, code, articles, etc.).
2. **Tokenization** → Break text into tokens.
3. **Neural architecture** (e.g., Transformers, RNNs) learns:
   * Word probabilities
   * Grammar patterns
   * Contextual meanings
4. **Inference** → Given a prompt, predict the next token repeatedly until:
   * A stopping condition is reached
   * Or max length is hit

**Examples**

* **OpenAI GPT series**
* **Google BERT** (though BERT is bidirectional & not generative)
* **Meta OPT**
* **EleutherAI GPT-NeoX**

**Key Characteristics**

| **Feature** | **Description** |
| --- | --- |
| **Input** | Plain text (prompt) |
| **Output** | Predicted next tokens / completed text |
| **Architecture** | Transformer-based (mostly) |
| **Pretraining Task** | Language modeling (next token prediction or masked token prediction) |
| **Size** | Can range from millions to hundreds of billions of parameters |

✅ **In short:**  
Language Models are the **core building block** of GenAI. They learn the statistical rules of language and can generate coherent text — but they are not necessarily trained to be “helpful” or “conversational.” That’s where **LLMs** and **Chat Models** come in later.

**2. Embedding Models – The Deep Dive**

**1. What are Embeddings?**

Think of embeddings as **numerical fingerprints of data**.

* They are **vectors** (arrays of numbers) that represent meaning.
* Similar meanings → vectors close to each other in space.
* Different meanings → vectors far apart.

📌 Example:

"King" → [0.21, 0.53, -0.14, ...]

"Queen" → [0.19, 0.51, -0.16, ...] → Similar direction in vector space

"Apple" → [0.87, -0.44, 0.31, ...] → Far from King/Queen

**2. Why are Embeddings Important?**

* **Semantic search** → "doctor" and "physician" match without exact word match.
* **Recommendation systems** → find similar movies/products.
* **Clustering** → group similar content.
* **Context understanding** for LLMs.

**3. How They Work**

1. **Model Training** – Embedding models learn from large datasets.
2. **Vectorization** – Each input (text, image, audio) → vector in a high-dimensional space.
3. **Distance Calculation** – Measure similarity with cosine similarity or Euclidean distance.

**4. Popular Embedding Models**

| **Model** | **Provider** | **Use Case** |
| --- | --- | --- |
| text-embedding-3-large | OpenAI | High accuracy semantic search |
| text-embedding-3-small | OpenAI | Cheaper, smaller search |
| all-MiniLM-L6-v2 | HuggingFace | Free, fast, smaller |
| e5-large-v2 | HuggingFace | Excellent for multi-lingual |
| bge-large-en | HuggingFace | Optimized for English |

**5. Example: Using OpenAI’s Embedding API**

from openai import OpenAI

client = OpenAI(api\_key="YOUR\_API\_KEY")

response = client.embeddings.create(

model="text-embedding-3-small",

input="FastAPI is a high-performance Python web framework."

)

vector = response.data[0].embedding

print(len(vector), vector[:5]) # length, first 5 values

📌 Output:

1536 [-0.014, 0.032, -0.002, 0.055, ...]

**6. Measuring Similarity**

import numpy as np

def cosine\_similarity(vec1, vec2):

return np.dot(vec1, vec2) / (np.linalg.norm(vec1) \* np.linalg.norm(vec2))

# Example:

v1 = [0.1, 0.2, 0.3]

v2 = [0.1, 0.2, 0.4]

print(cosine\_similarity(v1, v2))

**7. Full Stack Usage in FastAPI**

Imagine a **semantic search API**:

* User searches: "Best Italian restaurants in New York"
* We embed the query
* Compare it with stored restaurant embeddings
* Return top matches

**Backend FastAPI (embeddings.py)**:

from fastapi import FastAPI

from openai import OpenAI

import numpy as np

app = FastAPI()

client = OpenAI(api\_key="YOUR\_API\_KEY")

# Pre-stored restaurant embeddings (pretend these come from a DB)

restaurant\_data = [

{"name": "Luigi's Pasta", "embedding": [...]},

{"name": "Roma's Pizzeria", "embedding": [...]},

{"name": "Tuscany Deli", "embedding": [...]},

]

def cosine\_similarity(vec1, vec2):

return np.dot(vec1, vec2) / (np.linalg.norm(vec1) \* np.linalg.norm(vec2))

@app.get("/search")

async def search\_restaurants(query: str):

query\_embedding = client.embeddings.create(

model="text-embedding-3-small",

input=query

).data[0].embedding

# Rank restaurants by similarity

ranked = sorted(

restaurant\_data,

key=lambda r: cosine\_similarity(query\_embedding, r["embedding"]),

reverse=True

)

return {"results": ranked[:3]}

**8. Where to Store Embeddings**

* 1. **Vector Databases (Best for Large Scale Search)**

**Best for**: Semantic search, RAG (Retrieval-Augmented Generation), large datasets.

* Popular options:
  + **FAISS** (Facebook AI) — Fast, local, in-memory or disk index.
  + **Chroma** — Easy local/remote vector store.
  + **Weaviate**, **Pinecone**, **Milvus**, **Qdrant** — Cloud or self-hosted.
* Pros:
  + Built-in similarity search.
  + Scales to millions+ vectors.
* Cons:
  + Learning curve and setup needed.

**Relational or NoSQL DB**

**Best for**: When you already use a DB and embeddings are small.

* Store embeddings as JSON arrays or binary blobs.
* Use Postgres + **pgvector** for vector search.
* MongoDB Atlas also supports vector search.

**Rule of Thumb**:

* Small test → In-memory / .npy file.
* Up to hundreds of thousands → FAISS / Chroma locally.
* Millions+ or cloud scale → Pinecone / Weaviate / Milvus / Qdrant.

✅ **Summary**:

* Embeddings are *vector representations of meaning*.
* They enable semantic search, recommendations, clustering.
* They’re key for connecting **LLMs with memory & knowledge**.
* FastAPI can expose an embedding-based API easily.

**Full project Project layout**

semantic-search-pgvector/

├── app.py

├── ingest.py

├── requirements.txt

├── docker-compose.yml

├── init-db.sql

├── .env.example

├── my\_notes.txt

└── README\_RUN.md (run instructions are below too)

**1) requirements.txt**

fastapi

uvicorn[standard]

python-dotenv

openai

sqlalchemy

psycopg2-binary

pgvector

numpy

requests

* pgvector is the Python package that provides the Vector type for SQLAlchemy integration.
* openai is used to call the OpenAI embeddings API.
* psycopg2-binary is the Postgres driver for SQLAlchemy.

**2) docker-compose.yml**

This spins up Postgres and initializes the pgvector extension.

version: "3.8"

services:

postgres:

image: postgres:15

environment:

POSTGRES\_USER: postgres

POSTGRES\_PASSWORD: postgres

POSTGRES\_DB: semantic\_db

volumes:

- ./init-db.sql:/docker-entrypoint-initdb.d/init-db.sql:ro

- pgdata:/var/lib/postgresql/data

ports:

- "5432:5432"

volumes:

pgdata:

init-db.sql will run at container startup to enable the vector extension.

**3) init-db.sql**

-- init-db.sql

-- run in the POSTGRES\_DB (semantic\_db) at first time

CREATE EXTENSION IF NOT EXISTS vector;

**4) .env.example**

Create a .env from this and fill your OpenAI key:

OPENAI\_API\_KEY=sk-REPLACE\_WITH\_YOUR\_KEY

DATABASE\_URL=postgresql://postgres:postgres@localhost:5432/semantic\_db

EMBED\_MODEL=text-embedding-3-small

EMBED\_DIM=1536

* EMBED\_MODEL and EMBED\_DIM: set model + dimensionality. text-embedding-3-small dimension is 1536 (change if you use other model).
  + If you use a different embedding model, update EMBED\_DIM accordingly.

**5) my\_notes.txt (sample)**

In Q1 2024, our company launched two new products that exceeded sales expectations.

In Q2 2024, we expanded into three new markets: India, Brazil, and South Korea.

Customer satisfaction increased by 15% due to better support response times.

In Q3 2024, we plan to focus on AI-powered product recommendations.

In Q4 2024, our target is to increase revenue by 20% compared to Q4 2023.

**6) app.py — FastAPI server (full, explained)**

# app.py

"""

FastAPI semantic search with PostgreSQL + pgvector.

Endpoints:

- POST /ingest -> ingest a piece of text (generates embedding + stores)

- POST /ingest\_bulk -> ingest many texts from payload or files (helper)

- GET /search?q=...&k=5 -> semantic search for query q, returns top-k docs

- GET /health -> simple health check

"""

import os

from typing import List

from dotenv import load\_dotenv

from fastapi import FastAPI, HTTPException, Request

from pydantic import BaseModel

import openai

import numpy as np

# SQLAlchemy + pgvector

from sqlalchemy import create\_engine, Column, Integer, Text

from sqlalchemy.orm import sessionmaker, declarative\_base

from pgvector.sqlalchemy import Vector

# -- load config from .env --

load\_dotenv()

OPENAI\_API\_KEY = os.getenv("OPENAI\_API\_KEY")

DATABASE\_URL = os.getenv("DATABASE\_URL", "postgresql://postgres:postgres@localhost:5432/semantic\_db")

EMBED\_MODEL = os.getenv("EMBED\_MODEL", "text-embedding-3-small")

EMBED\_DIM = int(os.getenv("EMBED\_DIM", "1536"))

if not OPENAI\_API\_KEY:

raise RuntimeError("Set OPENAI\_API\_KEY in environment or .env file")

openai.api\_key = OPENAI\_API\_KEY

# -- SQLAlchemy setup --

engine = create\_engine(DATABASE\_URL, echo=False)

SessionLocal = sessionmaker(bind=engine)

Base = declarative\_base()

class Document(Base):

\_\_tablename\_\_ = "documents"

id = Column(Integer, primary\_key=True, index=True)

content = Column(Text, nullable=False)

embedding = Column(Vector(EMBED\_DIM), nullable=False) # pgvector Vector column

# Create tables (idempotent)

Base.metadata.create\_all(bind=engine)

# -- FastAPI app --

app = FastAPI(title="Semantic Search (pgvector)")

# -- Pydantic models for requests --

class IngestRequest(BaseModel):

content: str

class SearchResponseItem(BaseModel):

id: int

content: str

distance: float

class SearchResponse(BaseModel):

query: str

results: List[SearchResponseItem]

# ---------- Helpers ----------

def embed\_text(text: str) -> List[float]:

"""

Call OpenAI embeddings API and return a vector (list of floats).

"""

# NOTE: If you prefer LangChain embeddings, swap this call.

resp = openai.Embedding.create(model=EMBED\_MODEL, input=text)

vec = resp["data"][0]["embedding"]

return vec

# ---------- Endpoints ----------

@app.get("/health")

def health():

return {"ok": True}

@app.post("/ingest", status\_code=201)

def ingest(payload: IngestRequest):

"""

Ingest a single text document:

1) compute embedding via OpenAI

2) insert into documents table

"""

content = payload.content.strip()

if not content:

raise HTTPException(status\_code=400, detail="content empty")

vec = embed\_text(content)

# convert to numpy array (pgvector accepts list of floats via SQLAlchemy)

session = SessionLocal()

try:

doc = Document(content=content, embedding=vec)

session.add(doc)

session.commit()

session.refresh(doc)

return {"id": doc.id, "content": doc.content}

finally:

session.close()

@app.post("/ingest\_bulk", status\_code=201)

def ingest\_bulk(payload: List[IngestRequest]):

"""

Ingest multiple documents in a single call (naive implementation).

Use this to bulk load a file split into chunks.

"""

if not payload:

raise HTTPException(status\_code=400, detail="empty payload")

session = SessionLocal()

created = []

try:

for item in payload:

text = item.content.strip()

if not text:

continue

vec = embed\_text(text)

doc = Document(content=text, embedding=vec)

session.add(doc)

session.flush() # get id

created.append({"id": doc.id, "content": text})

session.commit()

return {"created": created}

finally:

session.close()

@app.get("/search", response\_model=SearchResponse)

def search(q: str, k: int = 5):

"""

Semantic search:

- compute embedding for query q

- use pgvector distance operator '<->' to order by Euclidean distance (smaller = closer)

- return top-k results

"""

if not q or q.strip() == "":

raise HTTPException(status\_code=400, detail="query missing")

q\_vec = embed\_text(q)

# raw SQL using pgvector operator for ordering is easiest:

# SELECT id, content, embedding <-> '<vector here>' AS distance FROM documents ORDER BY distance LIMIT k;

# We'll pass the vector as a parameter.

session = SessionLocal()

try:

# SQLAlchemy text-based query:

sql = "SELECT id, content, embedding <-> :qvec AS distance FROM documents ORDER BY distance LIMIT :k"

result = session.execute(sql, {"qvec": q\_vec, "k": k})

rows = result.fetchall()

results = []

for r in rows:

results.append(SearchResponseItem(id=r[0], content=r[1], distance=float(r[2])))

return SearchResponse(query=q, results=results)

finally:

session.close()

**Explanation — highlights**

* Document.embedding is Vector(EMBED\_DIM) from pgvector library — stores vector efficiently in Postgres.
* embed\_text() calls OpenAI Embeddings API (text-embedding-3-small by default). You can change EMBED\_MODEL in .env.
* /ingest inserts a single document and its computed embedding.
* /ingest\_bulk ingests a list of documents (useful when chunking a file into many pieces).
* /search computes a query embedding and performs semantic nearest-neighbor search using the embedding <-> query operator (pgvector). It orders by distance ascending (smaller is better).

**7) ingest.py — helper script to chunk a file and ingest**

Simple splitter: split by blank lines and long lines into chunks of ~200 characters with overlap. You can replace with LangChain’s splitter for more advanced chunking.

python

# ingest.py

"""

Read my\_notes.txt, split into chunks, and POST to /ingest\_bulk

Run: python ingest.py

"""

import os

import requests

from dotenv import load\_dotenv

load\_dotenv()

API\_URL = os.getenv("API\_URL", "http://127.0.0.1:8000")

FILE = "my\_notes.txt"

CHUNK\_SIZE = 300

CHUNK\_OVERLAP = 50

def simple\_chunk\_text(text, chunk\_size=CHUNK\_SIZE, overlap=CHUNK\_OVERLAP):

# naive chunker: split by sentences if possible, otherwise cut

text = text.replace("\r\n", "\n").strip()

if not text:

return []

parts = []

start = 0

while start < len(text):

end = min(start + chunk\_size, len(text))

chunk = text[start:end]

parts.append(chunk.strip())

start = end - overlap if end - overlap > start else end

return parts

def main():

with open(FILE, "r", encoding="utf-8") as f:

text = f.read()

chunks = simple\_chunk\_text(text)

payload = [{"content": c} for c in chunks]

resp = requests.post(f"{API\_URL}/ingest\_bulk", json=payload)

print("status:", resp.status\_code)

print(resp.json())

if \_\_name\_\_ == "\_\_main\_\_":

main()

* Set API\_URL in your .env if your server runs elsewhere (default http://127.0.0.1:8000).

**8) How to run — step-by-step**

1. Copy the project files into a folder (e.g., semantic-search-pgvector).
2. Create and edit .env from .env.example:

cp .env.example .env

# edit .env -> set OPENAI\_API\_KEY to your key

1. Start Postgres with pgvector:

docker-compose up -d

* Wait a few seconds for DB to initialize.

1. Create a Python virtualenv and install dependencies:

bash

python -m venv venv

source venv/bin/activate # mac/linux

# venv\Scripts\activate # windows

pip install -r requirements.txt

1. Start the FastAPI app:

bash

uvicorn app:app --reload --port 8000

1. In a new terminal, run the ingest helper to populate DB:

bash

python ingest.py

This will chunk my\_notes.txt and call /ingest\_bulk to store embeddings and content in Postgres.

1. Test search with curl:

bash

curl "http://127.0.0.1:8000/search?q=plans%20for%20Q3%202024&k=3"

You should get a JSON list of top matches (id, content, distance).

1. Ingest individual docs:

bash

curl -X POST "http://127.0.0.1:8000/ingest" -H "Content-Type: application/json" \

-d '{"content":"We will open a new office in Berlin in Q2 2025."}'

**Notes, tips & next steps**

* **Persistence**: Documents and embeddings are stored in Postgres (pgdata volume) — they persist between restarts.
* **Embedding cost**: Each ingest and each search (if embedding query on-the-fly) calls the OpenAI embeddings endpoint. For many queries consider caching embeddings for common queries or use approximate nearest neighbor engines.
* **Chunking**: The simple chunker works, but for production use LangChain or RecursiveCharacterTextSplitter (better handling of natural boundaries).
* **pgvector**:
  + embedding <-> query\_vector returns Euclidean distance (smaller is better).
  + pgvector also supports other metrics (inner product, cosine) if you prefer. To get cosine similarity you can normalize vectors and use inner product, or use embedding <#> query\_vector depending on pgvector version.
* **Security**: Never commit .env with your real OpenAI key. Use environment variables or secret management.
* **Performance**: For large datasets (millions of vectors), consider specialized vector DBs (Pinecone, Weaviate, Milvus) or use Postgres + pgvector with IVF/HNSW indexing (pgvector supports ivfflat and hnsw indexes).

If you want, I can next:

* Add an **example React frontend** to query /search and show results.
* Replace the simple chunker with **LangChain splitters** and show how to ingest a folder of PDFs/CSVs/HTML.
* Add **pgvector HNSW index** creation to speed up k-NN at scale.
* Add **reranking**: retrieve top 20 candidates then rerank with an LLM using cross-encoder style.

**3) Large Language Models (LLMs) — Deep Dive**

**What exactly is an LLM?**

A **Large Language Model** (LLM) is a language model that is:

* **Large** in parameter count (hundreds of millions → hundreds of billions),
* **Pretrained** on massive corpora, and
* **Capable** of performing a wide set of tasks (generation, summarization, translation, code, reasoning) often *without* task-specific training (zero-/few-shot).

Think of an LLM as a highly capable, general-purpose text engine that can be guided by prompts to perform many tasks.

**How LLMs differ from “language models” & embeddings**

* **Language Model (LM)** — general concept: learns token probabilities.
* **LLM** — a large-scale LM with broad capabilities and often tuned for instruction-following.
* **Embedding model** — produces vectors that capture semantics; *not* for text generation.  
  So: embeddings → retrieval/semantic tasks; LLMs → reasoning and fluent generation (possibly using retrieved context).

**Core pieces & internals (high-level)**

**Architecture**

* Modern LLMs use the **Transformer** architecture (self-attention blocks).
* Usually decoder-only (GPT-style) or encoder-decoder (T5-style). Decoder-only are the most common for open-ended generation.

**Training phases**

1. **Pretraining**: next-token prediction on large text corpora.
2. **Instruction tuning**: further training on (input → desired output) pairs so model follows instructions better.
3. **Alignment / Safety tuning**: human feedback (RLHF / RLAIF) to make outputs safer and more helpful.

**Tokenization**

* Text is split into tokens (subwords, BPE, byte-level). Tokenization affects costs and length limits.
* Always be mindful of tokens when setting max lengths and measuring cost.

**Important inference knobs (parameters you set at runtime)**

* **temperature (0.0–1.2+)** — randomness. 0.0 = deterministic (greedy); 0.7 = more creative.
* **top\_p (nucleus sampling)** — sample from smallest set with cumulative prob p. Often used with temperature.
* **max\_tokens** — max tokens to generate (controls length & cost).
* **presence\_penalty / frequency\_penalty** — discourage reusing same tokens.
* **stop sequences** — give strings to stop generation early (e.g., "\n\nHuman:" in chat simulators).
* **best\_of / n / logprobs** — multiple completions, likelihood scoring, etc.

**LLM capabilities & common applications**

* **Text generation**: creative writing, marketing copy.
* **Question answering**: when paired with retrieval (RAG).
* **Summarization**: long→short text summaries.
* **Code generation**: produce or fix code.
* **Translation / rewriting / formatting**.
* **Tool usage / agents**: LLM decides to call tools (APIs, python exec, DB).

**Failure modes & mitigations**

* **Hallucination** (inventing facts) — mitigate with RAG (retrieve real context), prompt engineering, and verification.
* **Bias / unsafe outputs** — use safety filters, RLHF, guardrails, and content moderation.
* **Token limits / truncated context** — chunking, long-context models, or retrieval/compression.
* **Cost & latency** — use smaller models for low-cost tasks, batching, caching, quantized local models.

**Fine-tuning & customization options**

* **Full fine-tuning** — expensive; trains many parameters.
* **Parameter-efficient fine-tuning**:
  + **LoRA** (Low-Rank Adapters) — lightweight add-on weights.
  + **Adapters**, **prefix tuning** — alternatives that require less compute.
* **Instruction tuning** — fine-tune with instruction → response datasets.
* **Prompt tuning** — keep model weights fixed, tune prompts or soft tokens.

**Deployment & inference optimizations**

* **Cloud APIs** (OpenAI, Anthropic, Cohere, etc.) — easiest to start, manage scaling & updates.
* **Self-hosted** (Llama, Mistral, etc.) — cheaper at scale or required for privacy, but need infra (GPU/quantization).
* **Quantization** — reduce precision (8-bit, 4-bit) to run bigger models locally with acceptable quality.
* **Batched inference + concurrency** — maximize throughput.
* **vLLM / DeepSpeed** — optimized inference engines for throughput and memory.

**Evaluation & metrics**

* **Perplexity** — core LM metric (lower is better).
* **ROUGE / BLEU** — for summarization/translation (but limited).
* **Exact match / F1** — for QA tasks.
* **Human eval** — often required for quality, helpfulness, safety.
* **Automated safety checks & toxicity scores.**

**Practical examples**

**A) Raw OpenAI ChatCompletion (simple)**

python

# pip install openai

import os

from openai import OpenAI

client = OpenAI(api\_key=os.environ["OPENAI\_API\_KEY"])

resp = client.chat.completions.create(

model="gpt-4o-mini",

messages=[

{"role":"system", "content":"You are a helpful assistant."},

{"role":"user", "content":"Summarize the following: In Q3 2024, we plan to focus on AI-powered product recommendations."}

],

temperature=0.0,

max\_tokens=150

)

print(resp.choices[0].message.content)

**B) LangChain LLM usage (simple LLMChain)**

python

# pip install langchain langchain-openai

from langchain\_openai import OpenAI

from langchain.prompts import PromptTemplate

from langchain.chains import LLMChain

llm = OpenAI(temperature=0, model="gpt-4o-mini") # LangChain LLM wrapper

template = "You are a concise assistant. Summarize: {text}"

prompt = PromptTemplate(input\_variables=["text"], template=template)

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

print(chain.run("In Q3 2024, we plan to focus on AI-powered product recommendations."))

**C) LLM with Retrieval (RAG pattern) — short sketch**

* Compute embedding for user query.
* Retrieve top-k documents.
* Construct prompt: system + Context: <retrieved snippets> + user question.
* Call LLM with temperature 0–0.2 for factual answers.

**D) Fine-tuning outline (LoRA example, conceptual)**

* Prepare dataset: JSONL with prompt and completion.
* Use a LoRA fine-tuning tool (e.g., peft, transformers training loop).
* Apply LoRA weights at inference time on base model.

(Implementation depends on the model and stack — I can provide a LoRA example for a chosen local model if you want.)

**Hallucination mitigation checklist (practical)**

1. Use **retrieval** (RAG) for knowledge-heavy queries.
2. Use **low temperature** and **few-shot examples** when you need conservative answers.
3. Add **explicit instruction** like “If you don't know, say 'I don't know'.” to prompt.
4. **Cross-check** important facts with external APIs or databases.
5. Use **output parsers** to constrain format (JSON schema) and validate results.

**Cost & sizing guidance (practical)**

* For quick dev / prototyping use smaller LLMs or gpt-3.5 family.
* For production-critical tasks needing reasoning/completeness, use larger models but optimize by:
  + Pre-filtering prompts with cheaper models,
  + Using retrieval to lower context size,
  + Caching responses for repeated prompts.

**When to host locally vs use API**

* **Use API** if: you want immediate access, easier scaling, less infra ops, and are okay with sending data to provider.
* **Self-host** if: you need data privacy, lower marginal cost at high scale, or want to experiment with novel architectures. Be ready for GPU infra, quantization, and inference engines.

**Quick checklist to build an LLM-powered feature (practical flow)**

1. Decide model: API vs local.
2. Choose prompt design and temperature.
3. Add retrieval if real-world knowledge is required.
4. Add memory if conversational context needed.
5. Add output parser/schema if structured output required.
6. Add logging + evaluation (LangSmith / custom).
7. Deploy and monitor cost & errors.

**Next hands-on options I can provide (pick any)**

* A **small runnable demo**: local LLM inference example (using llama.cpp or transformers with a small model).
* A **LangChain RAG pattern** example tuned for minimizing hallucination (with code you can run).
* A **LoRA fine-tuning walkthrough** on a small dataset (code + commands).
* A **production checklist**: metrics, logging, monitoring, costs, safety gates.

Which would you like me to show next? I recommend: **LangChain RAG example** (full) or **LoRA fine-tune demo** — which one?

**You said:**

first one -Next hands-on options I can provide (pick any)

**ChatGPT said:**

Here’s what I can do for the **first model (Embedding + Cosine Similarity with SentenceTransformers)** in a **hands-on** way.  
You can pick any of these next steps:

**1. Build a Simple Search Tool (Local Only)**

* Keep embeddings in memory (Python list).
* Search through them with cosine similarity.
* Perfect for small datasets or a quick proof of concept.
* No database, no hosting — just run locally.

**2. Store Embeddings in SQLite / Postgres (No Vector Extension)**

* Save documents + embeddings in a relational DB.
* Fetch embeddings from DB and run similarity search in Python.
* Suitable for medium datasets.
* Lets you integrate into existing SQL-based systems.

**3. Use FAISS for Scalable Vector Search**

* Store embeddings in FAISS index for **fast nearest neighbor search**.
* Supports millions of vectors.
* Can be disk-backed for persistence.
* Good for production-scale retrieval.

**4. Integrate with a REST API (FastAPI)**

* Expose endpoints:
  + /add\_document → stores text + embedding.
  + /search → finds top-N similar docs.
* Can run locally or deploy to the cloud.

**5. Add Reranking (Better Search Results)**

* Use a second model (e.g., cross-encoder) to rerank top results.
* Improves relevance when initial similarity matches are close.
* Slightly slower, but higher accuracy.

If you tell me which option you like, I’ll give you **full runnable code** with explanations.

Given you asked for “hands-on” earlier, I’d suggest starting with **#3 FAISS** — it’s fast, scalable, and production-ready.  
Do you want me to go with **FAISS integration** for this first model?

**Full FastAPI backend with HuggingFace embeddings + pgvector search + React UI**

We’ll build an **end-to-end working app** where:

* Backend (FastAPI) handles document upload, embedding creation, and semantic search using **PostgreSQL + pgvector**.
* Frontend (React) provides a **simple search UI** for entering queries and showing results.
* We’ll use **HuggingFace** for text embeddings (can easily swap to OpenAI if needed).

**1️⃣ Architecture Overview**

**Flow**:

1. User uploads text document (or directly pastes content) in UI.
2. FastAPI stores text + vector embedding in **Postgres** table with pgvector.
3. User types search query in UI.
4. FastAPI converts query → embedding → performs **vector similarity search** in Postgres.
5. Returns top matches to UI.

css

[React UI] → (REST API) → [FastAPI Backend] → [Postgres + pgvector]

**2️⃣ Backend (FastAPI)**

**Install dependencies**

bash

pip install fastapi uvicorn psycopg2-binary sqlalchemy sentence-transformers

**Postgres Setup with pgvector**

sql

CREATE EXTENSION IF NOT EXISTS vector;

CREATE TABLE documents (

id SERIAL PRIMARY KEY,

content TEXT,

embedding vector(384) -- dimension matches the embedding model

);

**FastAPI app — main.py**

python

from fastapi import FastAPI, UploadFile, Form

from sentence\_transformers import SentenceTransformer

import psycopg2

import io

app = FastAPI()

# Load model

model = SentenceTransformer("all-MiniLM-L6-v2")

# DB connection

conn = psycopg2.connect(

host="localhost",

database="semantic\_db",

user="postgres",

password="yourpassword"

)

cur = conn.cursor()

@app.post("/upload/")

async def upload\_document(file: UploadFile):

text = (await file.read()).decode("utf-8")

embedding = model.encode([text])[0].tolist()

cur.execute(

"INSERT INTO documents (content, embedding) VALUES (%s, %s)",

(text, embedding)

)

conn.commit()

return {"status": "uploaded"}

@app.get("/search/")

def search(q: str):

query\_embedding = model.encode([q])[0].tolist()

cur.execute(

"""

SELECT content, embedding <=> %s AS distance

FROM documents

ORDER BY distance ASC

LIMIT 5

""",

(query\_embedding,)

)

results = cur.fetchall()

return [{"content": r[0], "score": 1 - r[1]} for r in results]

**Notes:**

* embedding <=> %s uses pgvector’s **cosine distance**.
* If using Euclidean distance, use <-> instead.

**3️⃣ Frontend (React)**

**Setup**

bash

npx create-react-app semantic-search-ui

cd semantic-search-ui

npm install axios

**App.js**

javascript

import React, { useState } from "react";

import axios from "axios";

function App() {

const [query, setQuery] = useState("");

const [results, setResults] = useState([]);

const [file, setFile] = useState(null);

const handleSearch = async () => {

const res = await axios.get("http://localhost:8000/search/", {

params: { q: query },

});

setResults(res.data);

};

const handleUpload = async () => {

const formData = new FormData();

formData.append("file", file);

await axios.post("http://localhost:8000/upload/", formData);

alert("Uploaded successfully!");

};

return (

<div style={{ padding: "20px" }}>

<h2>Semantic Search UI</h2>

<h3>Upload Document</h3>

<input type="file" onChange={(e) => setFile(e.target.files[0])} />

<button onClick={handleUpload}>Upload</button>

<h3>Search</h3>

<input

value={query}

onChange={(e) => setQuery(e.target.value)}

placeholder="Type your search..."

/>

<button onClick={handleSearch}>Search</button>

<h3>Results:</h3>

<ul>

{results.map((r, i) => (

<li key={i}>

<strong>Score:</strong> {r.score.toFixed(4)} — {r.content}

</li>

))}

</ul>

</div>

);

}

export default App;

**4️⃣ Running Everything**

**Backend**

bash

uvicorn main:app --reload

**Frontend**

bash

npm start

✅ **What we get**:

* Upload .txt files to the backend.
* Store content in Postgres with embeddings.
* Search semantically from React UI.

If you want, I can **extend this to support multiple file formats (PDF, DOCX)** and **chunk large documents** so that the search returns smaller, context-rich matches instead of entire files. That will make it **production-grade** like ChatGPT retrieval.

Do you want me to now extend this with **PDF/DOCX chunking + richer search results**? That would make it far more powerful.

**You said:**

yes please - Do you want me to now extend this with PDF/DOCX chunking + richer search results? That would make it far more powerful.

**ChatGPT said:**

Thought for a couple of seconds

Great — let’s upgrade the project to support **PDF / DOCX chunking** and **richer search results** (showing source filename, chunk index, and a snippet). I’ll give you a **complete, runnable FastAPI backend** plus the **React UI changes**. Everything is copy-paste ready.

We’ll keep using:

* **Postgres + pgvector** for vector storage (dimension 384 for all-MiniLM-L6-v2)
* **SentenceTransformers (all-MiniLM-L6-v2)** for embeddings (local, no OpenAI calls)
* **PyPDF2** for PDF text extraction and **python-docx** for DOCX
* A chunker that produces overlap and keeps metadata (source file, chunk index, page when available)
* Search endpoint returns: content snippet, source filename, chunk\_index, and score

**Project layout (updated)**

bash

semantic-search-pgvector/

├── app.py # FastAPI backend (updated)

├── ingest.py # helper script (optional)

├── requirements.txt

├── docker-compose.yml

├── init-db.sql

├── .env.example

├── my\_notes.txt # optional

└── templates/ # optional if you use server-side UI

frontend/

└── src/

└── App.js # React UI (updated)

**1) requirements.txt (backend)**

txt

fastapi

uvicorn[standard]

python-dotenv

sqlalchemy

psycopg2-binary

pgvector

sentence-transformers

numpy

python-multipart

PyPDF2

python-docx

aiofiles

Install with:

bash

python -m venv venv

source venv/bin/activate

pip install -r requirements.txt

**2) Database changes**

Update the documents table to include source metadata:

sql

-- init-db.sql

CREATE EXTENSION IF NOT EXISTS vector;

CREATE TABLE IF NOT EXISTS documents (

id SERIAL PRIMARY KEY,

content TEXT NOT NULL,

embedding vector(384) NOT NULL,

source TEXT, -- filename or 'pasted text'

chunk\_index INTEGER, -- position in file/chunks

page INTEGER -- PDF page when available, NULL otherwise

);

If you already have the table, either drop and recreate for the new schema or ALTER TABLE accordingly.

**3) app.py — Full FastAPI backend (PDF/DOCX chunking + richer results)**

python

# app.py

"""

FastAPI backend with PDF/DOCX chunking and pgvector semantic search.

Run:

1) ensure Postgres+pgvector is running (see docker-compose/init-db.sql)

2) pip install -r requirements.txt

3) set env var DATABASE\_URL and optionally MODEL\_NAME (default uses all-MiniLM-L6-v2)

4) uvicorn app:app --reload

"""

import os, io, math

from typing import List, Optional

from dotenv import load\_dotenv

from fastapi import FastAPI, UploadFile, File, Form, HTTPException

from fastapi.responses import JSONResponse

from pydantic import BaseModel

import numpy as np

# DB

from sqlalchemy import create\_engine, Column, Integer, Text, String

from sqlalchemy.orm import sessionmaker, declarative\_base

from pgvector.sqlalchemy import Vector

# text extraction

from PyPDF2 import PdfReader

import docx

# embeddings

from sentence\_transformers import SentenceTransformer

load\_dotenv()

DATABASE\_URL = os.getenv("DATABASE\_URL", "postgresql://postgres:postgres@localhost:5432/semantic\_db")

EMBED\_MODEL = os.getenv("EMBED\_MODEL", "all-MiniLM-L6-v2")

EMBED\_DIM = 384 # embedding dimension for all-MiniLM-L6-v2

# SQLAlchemy setup

engine = create\_engine(DATABASE\_URL, echo=False)

SessionLocal = sessionmaker(bind=engine)

Base = declarative\_base()

class Document(Base):

\_\_tablename\_\_ = "documents"

id = Column(Integer, primary\_key=True, index=True)

content = Column(Text, nullable=False) # chunk text

embedding = Column(Vector(EMBED\_DIM), nullable=False)

source = Column(String, nullable=True) # filename or 'pasted'

chunk\_index = Column(Integer, nullable=True)

page = Column(Integer, nullable=True)

Base.metadata.create\_all(bind=engine)

app = FastAPI(title="Semantic Search with PDF/DOCX chunking")

# Load sentence-transformers model (local)

model = SentenceTransformer(EMBED\_MODEL)

# -------------------------

# Utility: extract text from files

# -------------------------

def extract\_text\_from\_pdf\_bytes(b: bytes) -> List[dict]:

"""

Returns list of dicts: [{'page': 0, 'text': '...'}, ...]

"""

reader = PdfReader(io.BytesIO(b))

pages = []

for i, page in enumerate(reader.pages):

try:

txt = page.extract\_text() or ""

except Exception:

txt = ""

pages.append({"page": i, "text": txt})

return pages

def extract\_text\_from\_docx\_bytes(b: bytes) -> str:

f = io.BytesIO(b)

doc = docx.Document(f)

texts = []

for para in doc.paragraphs:

texts.append(para.text)

return "\n".join(texts)

# -------------------------

# Utility: chunker

# -------------------------

def simple\_chunker(text: str, chunk\_size: int = 400, overlap: int = 100):

"""

Naive character-based chunker with overlap.

Returns list of strings.

"""

text = text.replace("\r\n", "\n").strip()

if not text:

return []

chunks = []

start = 0

length = len(text)

while start < length:

end = start + chunk\_size

chunk = text[start:end]

chunks.append(chunk.strip())

if end >= length:

break

start = end - overlap

return chunks

# -------------------------

# Helper: compute embeddings (batch)

# -------------------------

def embed\_texts(texts: List[str]) -> List[List[float]]:

# model.encode returns numpy arrays; convert to lists

vecs = model.encode(texts, show\_progress\_bar=False, convert\_to\_numpy=True)

return [v.tolist() for v in vecs]

# -------------------------

# API models

# -------------------------

class IngestResponse(BaseModel):

created: int

class SearchResultItem(BaseModel):

id: int

content: str

source: Optional[str]

chunk\_index: Optional[int]

page: Optional[int]

score: float

class SearchResponse(BaseModel):

query: str

results: List[SearchResultItem]

# -------------------------

# Endpoint: upload file (PDF/DOCX/TXT) and ingest chunks

# -------------------------

@app.post("/upload\_file", response\_model=IngestResponse)

async def upload\_file(file: UploadFile = File(...), chunk\_size: int = Form(400), chunk\_overlap: int = Form(100)):

"""

Accepts PDF, DOCX or plain text file.

Splits into chunks, creates embeddings and stores in DB.

"""

filename = file.filename

content\_bytes = await file.read()

texts\_with\_meta = [] # list of dicts: {'text':..., 'source':..., 'chunk\_index':..., 'page':...}

try:

if filename.lower().endswith(".pdf"):

pages = extract\_text\_from\_pdf\_bytes(content\_bytes)

for p in pages:

page\_text = p.get("text", "")

page\_idx = p.get("page")

# chunk each page separately (keeps page metadata)

chunks = simple\_chunker(page\_text, chunk\_size=chunk\_size, overlap=chunk\_overlap)

for ci, c in enumerate(chunks):

texts\_with\_meta.append({"text": c, "source": filename, "chunk\_index": ci, "page": page\_idx})

elif filename.lower().endswith(".docx"):

whole\_text = extract\_text\_from\_docx\_bytes(content\_bytes)

chunks = simple\_chunker(whole\_text, chunk\_size=chunk\_size, overlap=chunk\_overlap)

for ci, c in enumerate(chunks):

texts\_with\_meta.append({"text": c, "source": filename, "chunk\_index": ci, "page": None})

else:

# treat as plain text

text = content\_bytes.decode("utf-8", errors="ignore")

chunks = simple\_chunker(text, chunk\_size=chunk\_size, overlap=chunk\_overlap)

for ci, c in enumerate(chunks):

texts\_with\_meta.append({"text": c, "source": filename, "chunk\_index": ci, "page": None})

except Exception as e:

raise HTTPException(status\_code=400, detail=f"Failed to parse file: {e}")

if not texts\_with\_meta:

return IngestResponse(created=0)

# compute embeddings in batches

texts = [t["text"] for t in texts\_with\_meta]

embeddings = embed\_texts(texts)

# insert into DB

session = SessionLocal()

created\_count = 0

try:

for obj, emb in zip(texts\_with\_meta, embeddings):

doc = Document(content=obj["text"], embedding=emb, source=obj["source"], chunk\_index=obj["chunk\_index"], page=obj["page"])

session.add(doc)

created\_count += 1

session.commit()

finally:

session.close()

return IngestResponse(created=created\_count)

# -------------------------

# Endpoint: ingest plain text (paste) - useful for small notes

# -------------------------

@app.post("/ingest\_text", response\_model=IngestResponse)

def ingest\_text(content: str = Form(...), chunk\_size: int = Form(400), chunk\_overlap: int = Form(100), source: str = Form("pasted")):

chunks = simple\_chunker(content, chunk\_size=chunk\_size, overlap=chunk\_overlap)

if not chunks:

return IngestResponse(created=0)

embeddings = embed\_texts(chunks)

session = SessionLocal()

created\_count = 0

try:

for ci, (c, emb) in enumerate(zip(chunks, embeddings)):

doc = Document(content=c, embedding=emb, source=source, chunk\_index=ci, page=None)

session.add(doc)

created\_count += 1

session.commit()

finally:

session.close()

return IngestResponse(created=created\_count)

# -------------------------

# Endpoint: search (semantic)

# -------------------------

@app.get("/search", response\_model=SearchResponse)

def search(q: str, k: int = 5):

"""

Compute query embedding and use pgvector <-> operator (cosine similarity) or <-> depends on setup.

We will use inner product similarity transformed to score between 0..1 for presentation.

"""

if not q or q.strip() == "":

raise HTTPException(status\_code=400, detail="Empty query")

q\_vec = embed\_texts([q])[0] # single vector list -> take 0

# Raw SQL uses pgvector operator. By default pgvector offers:

# - '<->' for Euclidean distance

# - '<#>' / '<=>', '<=>' etc for cosine depending on version. We'll use '<->' (euclidean) in this example.

session = SessionLocal()

try:

sql = "SELECT id, content, source, chunk\_index, page, embedding <-> :qvec AS distance FROM documents ORDER BY distance ASC LIMIT :k"

rows = session.execute(sql, {"qvec": q\_vec, "k": k}).fetchall()

results = []

# convert distance to a pseudo-score: smaller distance -> higher score (0..1)

# This is not a true cosine; for real cosine, store normalized vectors and use inner product.

max\_possible = 1.0 # placeholder

for r in rows:

doc\_id, content, source, chunk\_idx, page, distance = r

# naive conversion to score: invert distance (higher better)

# NOTE: distances range depends on vector; for presentation we map via 1 / (1+dist)

score = 1.0 / (1.0 + float(distance))

results.append({

"id": int(doc\_id),

"content": content,

"source": source,

"chunk\_index": int(chunk\_idx) if chunk\_idx is not None else None,

"page": int(page) if page is not None else None,

"score": float(score)

})

return SearchResponse(query=q, results=results)

finally:

session.close()

**Key points / explanations**

* We extract text from PDFs page-by-page using PyPDF2.PdfReader and from DOCX via python-docx.
* Chunking is character-based with overlap (tune chunk\_size and chunk\_overlap for best retrieval).
* Each chunk stores source (filename), chunk\_index, and page (if PDF).
* We compute embeddings using sentence-transformers entirely offline (no API calls).
* /upload\_file accepts multipart file upload and ingests chunks.
* /ingest\_text lets you paste text into the API and ingest it (useful for quick notes).
* /search returns results with score mapped from distance for human-friendly sorting. For production, consider normalizing vectors and using proper cosine operator or HNSW index for speed.

**4) React UI changes (frontend)**

We’ll update the React UI to:

* Upload files (PDF/DOCX/TXT)
* Show search results with source filename, chunk index, page, and snippet

**App.js (React)**

Replace your current App.js with the following:

javascript

// frontend/src/App.js

import React, { useState } from "react";

import axios from "axios";

function App() {

const [file, setFile] = useState(null);

const [query, setQuery] = useState("");

const [results, setResults] = useState([]);

const [ingestCount, setIngestCount] = useState(null);

const uploadFile = async () => {

if (!file) { alert("select file"); return; }

const fd = new FormData();

fd.append("file", file);

// optional: pass chunk size/overlap

fd.append("chunk\_size", 400);

fd.append("chunk\_overlap", 100);

try {

const res = await axios.post("http://localhost:8000/upload\_file", fd, {

headers: { "Content-Type": "multipart/form-data" },

});

setIngestCount(res.data.created);

alert(`Ingested ${res.data.created} chunks`);

} catch (e) {

console.error(e);

alert("Upload failed");

}

};

const ingestText = async (text) => {

const fd = new FormData();

fd.append("content", text);

fd.append("source", "pasted");

fd.append("chunk\_size", 400);

fd.append("chunk\_overlap", 100);

const res = await axios.post("http://localhost:8000/ingest\_text", fd);

setIngestCount(res.data.created);

alert(`Ingested ${res.data.created} chunks`);

};

const search = async () => {

if (!query) return;

try {

const res = await axios.get("http://localhost:8000/search", {

params: { q: query, k: 8 },

});

setResults(res.data.results);

} catch (e) {

console.error(e);

alert("Search failed");

}

};

return (

<div style={{ padding: 24, fontFamily: "Arial" }}>

<h2>Semantic Search with PDF/DOCX Chunking</h2>

<section style={{ marginBottom: 16 }}>

<h3>Upload file (PDF / DOCX / TXT)</h3>

<input type="file" onChange={(e) => setFile(e.target.files[0])} />

<button onClick={uploadFile}>Upload & Ingest</button>

{ingestCount !== null && <div>Last ingest created: {ingestCount} chunks</div>}

</section>

<section style={{ marginBottom: 16 }}>

<h3>Or paste text to ingest</h3>

<textarea id="paste" placeholder="Paste text here..." rows={6} style={{ width: "100%" }} />

<button onClick={() => ingestText(document.getElementById("paste").value)}>Ingest Pasted Text</button>

</section>

<section style={{ marginBottom: 16 }}>

<h3>Search</h3>

<input value={query} onChange={(e) => setQuery(e.target.value)} style={{ width: "70%" }} />

<button onClick={search}>Search</button>

</section>

<section>

<h3>Results</h3>

<ol>

{results.map((r) => (

<li key={r.id} style={{ marginBottom: 12 }}>

<div><strong>Score:</strong> {r.score.toFixed(4)}</div>

<div><strong>Source:</strong> {r.source || "unknown"} <strong>Chunk:</strong> {r.chunk\_index} {r.page !== null && <><strong>Page:</strong> {r.page}</>}</div>

<div style={{ marginTop: 6, background: "#fff", padding: 8, borderRadius: 6, border: "1px solid #eee" }}>

<div style={{ maxHeight: 120, overflow: "auto", whiteSpace: "pre-wrap" }}>{r.content}</div>

</div>

</li>

))}

</ol>

</section>

</div>

);

}

export default App;

**Notes for frontend**

* The upload uses multipart/form-data.
* ingestText posts form fields (FastAPI will accept Form); the backend endpoint expects content as Form.
* Make sure CORS is configured if your frontend runs on a different origin (see next).

**5) CORS — allow frontend to call backend**

Add CORS in app.py:

python

from fastapi.middleware.cors import CORSMiddleware

app.add\_middleware(

CORSMiddleware,

allow\_origins=["http://localhost:3000"], # adjust if your React dev server runs elsewhere

allow\_credentials=True,

allow\_methods=["\*"],

allow\_headers=["\*"],

)

Add this before endpoints are defined.

**6) Run instructions (recap)**

1. Ensure Postgres + pgvector are running (docker-compose from earlier). init-db.sql must create extension and table schema (or run the CREATE TABLE manually).
2. Create .env and set DATABASE\_URL if not default.
3. Create & activate venv:

bash

python -m venv venv

source venv/bin/activate

pip install -r requirements.txt

1. Start FastAPI:

bash

uvicorn app:app --reload --port 8000

1. Start React dev server (in frontend):

bash

cd frontend

npm install

npm start

1. Upload a PDF or DOCX via UI, wait for ingest alert telling how many chunks were created. Then search.

**7) Tuning tips & production notes**

* **Chunk size**: 400 chars / overlap 100 is a good start. For long technical docs try 1k char chunks or split by paragraphs.
* **Model**: all-MiniLM-L6-v2 (384-dim) is small & fast. For better quality try all-mpnet-base-v2 (768-dim) — adjust Vector dimension in DB and model accordingly.
* **Indexing**: For large datasets, add an index in Postgres using ivfflat or hnsw via pgvector:

sql

CREATE INDEX ON documents USING ivfflat (embedding vector\_cosine\_ops) WITH (lists = 100);

Then run REINDEX or follow pgvector docs for indexing.

* **Normalization**: If you want cosine similarity, normalize vectors before storing and use inner-product ordering.
* **Reranking**: For higher accuracy, retrieve top-50 candidates from DB then rerank with a cross-encoder (or use an LLM to rerank / score).
* **Metadata**: Save file path, upload timestamp, original filename, and optionally text offsets so you can show highlighted passages in UI.
* **Security**: Validate uploaded files; limit sizes; run virus scans if needed.
* **Persistence & Re-ingest**: If you re-upload changed files, implement upsert logic (delete old chunks for same filename before inserting new ones).

If you want, I can:

* Provide the **exact SQL** commands to create a pgvector HNSW index and demonstrate its speed with sample dataset.
* Add a **reranking pipeline**: retrieve 50 candidates from DB, rerank with a smaller cross-encoder or LLM, return top 5.
* Add **highlighted snippet extraction** (return substring with context and highlight matched terms).
* Convert the React UI into a prettier UI (Material-UI / Tailwind) with snippet highlighting and file list.

Which of those would you like next?

**What is a Chat Model?**

A **Chat Model** is an LLM that’s been *packaged and often fine-tuned* specifically for **multi-turn conversational use**. Instead of a single prompt → completion, chat models accept a sequence of messages (roles like system, user, assistant, tool) and return a message (or call a tool). They’re optimized for multi-turn coherence, role semantics, and instruction-following.

**How chat models differ from plain LLMs & embeddings**

* **Structure**: Chat models use *message sequences* with explicit roles. Plain LLMs take a single prompt string.
* **Optimization**: Chat models are instruction-tuned and sometimes trained with human feedback to behave conversationally (e.g., follow system message rules).
* **Features**: Commonly support *function/tool calling* and *streaming* outputs natively.
* **Use**: Best for assistants, bots, and interactive agents; embeddings are for semantic search, not generation.

**Core concepts & primitives**

**Roles (message types)**

* **system** — global instructions / persona / constraints. (Set once or rarely changed.)
* **user** — what the human types.
* **assistant** — model replies (previous responses).
* **tool/function** — messages representing tool output or function calls (in modern tool-calling APIs).

**System message example**

system: "You are a concise professional assistant. If you don't know, say 'I don't know'."

**Conversation context**

* Entire message history can be included up to the model’s context window (token limit).
* Models are stateless between calls; the client supplies history each request (unless server wraps it as memory).

**Architectures & training**

* Usually **decoder-only Transformers** (GPT-style) with instruction tuning and RLHF.
* Fine-tuned on conversations (human–assistant pairs) to produce helpful, safe replies.
* Tokenizers and positional encodings limit the context window (e.g., 8k, 32k, 100k tokens depending on model).

**Important runtime knobs for chat usage**

* **temperature** — randomness (0 = deterministic).
* **top\_p** — nucleus sampling.
* **max\_tokens** — generated length limit.
* **stop sequences** — strings to end generation early.
* **streaming** — receive tokens incrementally.
* **function\_call / tools** — allow model to decide to call a function/tool rather than respond.

**Tool (function) calling — modern, core feature**

* You define a set of **tools/functions** (name, description, JSON schema for arguments).
* The model can either: return text *or* return a **function call** object (name + arguments).
* The client executes the function and returns the result as a tool/assistant message; the model can continue the conversation using that tool output.
* This pattern is great for grounding answers (databases, calculators, web search, code execution).

**Memory & multi-turn strategies**

* **Stateless convo**: send only the current user query + short context or retrieval results (cheap, but limited).
* **Buffer memory**: store whole history and re-send each call (works but can blow token budget).
* **Summary memory**: compress old history into a summary and send that plus last few turns.
* **Vector memory**: store embeddings of facts to retrieve relevant facts when needed.
* **Hybrid**: summary + retrieval for long-term facts + buffer for recent turns.

**Prompt engineering patterns for chat models**

* **System-first**: always include a system message to set persona & constraints.
* **Few-shot messages**: include a couple of example user/assistant pairs for behavior shaping.
* **Context + instruction**: system sets rules; user asks with relevant retrieved context embedded.
* **Output schema**: use JSON schema or output parsers to constrain format (especially for downstream processing).

Example system + user pattern:

system: "You are concise and cite sources when possible."

user: "Using only the provided context, answer: <question>"

context: "<retrieved snippets here>"

**Streaming & token-by-token UX**

* Streaming gives faster perceived latency: start rendering partial tokens on the client while the model is still generating.
* Requires handling partial tokens and finalization (stop conditions).

**Safety & hallucination mitigation**

* **RAG**: retrieve verified context and instruct model to use only provided context for factual answers.
* **Guardrails**: system messages + post-generation filters (toxicity, privacy checks).
* **Function calls**: force model to call a fact-checking tool for claims.
* **Calibration**: low temperature for factual tasks; high for creative tasks.

**Failure modes**

* **Hallucination** — inventing facts. Mitigate with RAG, low temperature, cite sources.
* **Context truncation** — older messages get lost. Use summary + retrieval.
* **Ambiguous role use** — model confuses user/system. Keep system clear and minimal.
* **Tool misuse** — model calls wrong tool; improve tool descriptions and examples.

**Evaluation & metrics**

* **Automated**: BLEU/ROUGE (limited), factuality scores, exact match, token-level log probs.
* **Human**: helpfulness, accuracy, safety, style, and latency.
* **Logging**: record inputs, outputs, tool calls, and sources for debugging and retraining.

**Cost & scaling guidance**

* Use smaller chat models for many small queries; larger models for complex reasoning.
* Cache results for repeated questions; batch similar requests if possible.
* Use retrieval + cheap model to filter and expensive model to finalize.

**Concrete examples**

**1) OpenAI-style Chat (synchronous) — Python (simple)**

python

from openai import OpenAI

client = OpenAI(api\_key="YOUR\_KEY")

messages = [

{"role": "system", "content": "You are a concise assistant. If unknown, say 'I don't know'."},

{"role": "user", "content": "Summarize the company's Q3 2024 plan using only the context below.\n\nCONTEXT: In Q3 2024, we plan to focus on AI-powered product recommendations."}

]

resp = client.chat.completions.create(

model="gpt-4o-mini",

messages=messages,

temperature=0.0,

max\_tokens=150

)

print(resp.choices[0].message.content)

**2) Function-calling example (OpenAI-style)**

python

from openai import OpenAI

client = OpenAI(api\_key="YOUR\_KEY")

functions = [

{

"name": "get\_latest\_sales",

"description": "Return latest sales number for a product",

"parameters": {

"type": "object",

"properties": {

"product\_id": {"type": "string"}

},

"required": ["product\_id"]

}

}

]

messages = [

{"role":"system","content":"You can call tools to fetch real data."},

{"role":"user","content":"What were sales for product 123 last month?"}

]

resp = client.chat.completions.create(

model="gpt-4o-mini",

messages=messages,

functions=functions,

function\_call="auto", # model decides whether to call

)

choice = resp.choices[0]

if choice.message.get("function\_call"):

fname = choice.message["function\_call"]["name"]

args = json.loads(choice.message["function\_call"]["arguments"])

# call your function

result = get\_latest\_sales(args["product\_id"])

# send back result as assistant message with role='tool' or 'assistant' depending on API

messages.append({"role":"assistant","content": None, "function\_call":{"name":fname,"arguments":json.dumps(args)}})

messages.append({"role":"tool","name":fname,"content": json.dumps(result)})

# call model again to get final answer

second = client.chat.completions.create(model="gpt-4o-mini", messages=messages)

print(second.choices[0].message.content)

else:

print(choice.message.content)

**3) LangChain — ChatPromptTemplate + ConversationChain**

python

from langchain.chat\_models import ChatOpenAI

from langchain.prompts import ChatPromptTemplate

from langchain.chains import ConversationChain

from langchain.memory import ConversationBufferMemory

llm = ChatOpenAI(model="gpt-4o-mini", temperature=0.0)

prompt = ChatPromptTemplate.from\_messages([

("system", "You are concise and only use provided context."),

("human","{input}")

])

memory = ConversationBufferMemory(return\_messages=True)

chain = ConversationChain(llm=llm, prompt=prompt, memory=memory)

print(chain.run("What are our Q3 plans?"))

# memory persists last turns in chain

**Best practices & checklist (copyable)**

**Design & prompts**

* Always include a clear system message (persona + constraints).
* Keep system short but explicit about forbidden behaviour.
* Provide examples if you need a specialized format (few-shot in messages).
* Use an **output schema** (JSON schema or output parser) when downstream systems expect structured data.

**Retrieval & grounding**

* Use RAG for factual answers: retrieve top-k docs and include them as context before user message.
* If large context, summarize older content and retrieve important facts.

**Tooling**

* Expose minimal, well-documented tools (name/description) — simpler tools prevent misuse.
* Provide the model with explicit tool schemas for reliable args.

**Memory**

* Buffer for recent turns, summary or vector memory for long-term facts.
* Limit what you resend to the model to control token cost.

**Safety & testing**

* Add content filters post-response.
* Test prompts with adversarial inputs (prompt injection).
* Log inputs, outputs, and tool calls for audit/training.

**Performance**

* Use streaming to reduce perceived latency.
* Cache repeated QA queries and reruns.
* Use cheaper models for pre-processing / filtering.

**Debugging tips**

* Inspect full message history sent to the model (especially system message).
* If model ignores system: tighten phrasing, add examples, reduce temperature.
* If hallucinations persist: lower temperature, force model to answer from sources only.
* Use function calls to constrain actions — easier to verify.

**Quick UX patterns**

* **Assistant that cites**: include source snippets and instruct model to cite the snippet id.
* **Step-by-step chain**: use plan → act → verify approach for complex tasks — either via LLM planning or LangChain agents.
* **Human-in-the-loop**: require human confirmation for risky actions (financial transfers, deletions).

**Final short recipe (how to build a robust chat assistant)**

1. Choose chat model & set system instructions.
2. Add retrieval (vector DB + embeddings) for grounding.
3. Attach memory (short-term buffer + long-term vector memory).
4. Add vetted tools for external data; enable function calling.
5. Use output schemas for structured responses.
6. Stream tokens to UI and log everything.
7. Add safety filters and test with adversarial prompts.

If you want now, I can:

* Provide a **runnable minimal FastAPI chat server** that demonstrates system messages, retrieval-grounding, and a function-call tool (complete code).
* Or build a **LangChain agent** example that uses a search tool + python executor + memory and walks through a sample conversation step-by-step.

**fully runnable FastAPI chat server that demonstrates:**

* **System message** to control model behavior
* **Retrieval grounding**: finds top-k relevant document chunks via embeddings & cosine similarity and includes them in the prompt (RAG)
* **Function (tool) calling**: exposes two example local functions the model can call; the server executes the function and returns the tool output to the model for a final answer (complete call loop)
* Clean, commented code and instructions so you can run it locally

I used the official OpenAI Chat API function-calling approach as the model driver (docs/examples referenced). See OpenAI docs for the function-calling pattern and chat completion API. [OpenAI Platform+1](https://platform.openai.com/docs/assistants/tools/function-calling/quickstart?utm_source=chatgpt.com)

**Files you’ll create**

rag\_chat\_server/

├── requirements.txt

├── sample\_docs.txt

└── app.py

**requirements.txt**

txt

fastapi

uvicorn[standard]

python-dotenv

sentence-transformers

numpy

openai

Install with:

bash

python -m venv venv

source venv/bin/activate # or venv\Scripts\activate on Windows

pip install -r requirements.txt

Note: openai is the official Python package; this code uses the current OpenAI client patterns (see docs). [OpenAI Platform](https://platform.openai.com/docs/api-reference/chat?utm_source=chatgpt.com)

**sample\_docs.txt**

Put some short knowledge text here (used for retrieval). Example:

pgsql

Our product X will launch in July 2025 with an initial price of $29.99.

We expanded into India and Brazil in Q2 2024.

Customer satisfaction increased by 15% after support improvements.

We plan to focus on AI-powered recommendations in Q3 2024.

**app.py — full code (copy & run)**

python

"""

RAG Chat Server with Function Calling (FastAPI)

Run:

1) set OPENAI\_API\_KEY in your environment (export OPENAI\_API\_KEY="sk-...")

2) python -m venv venv

source venv/bin/activate

pip install -r requirements.txt

3) uvicorn app:app --reload

4) POST JSON to http://127.0.0.1:8000/chat with {"user\_input": "..."}

"""

import os

import json

from typing import List, Dict, Any

from fastapi import FastAPI, HTTPException

from pydantic import BaseModel

from dotenv import load\_dotenv

# embeddings + numeric operations

from sentence\_transformers import SentenceTransformer

import numpy as np

# OpenAI client

from openai import OpenAI

# load .env if present

load\_dotenv()

# ---------------------------

# Config / init

# ---------------------------

OPENAI\_API\_KEY = os.getenv("OPENAI\_API\_KEY")

if not OPENAI\_API\_KEY:

raise RuntimeError("Set OPENAI\_API\_KEY in environment before running")

# instantiate OpenAI client (uses OPENAI\_API\_KEY)

client = OpenAI(api\_key=OPENAI\_API\_KEY)

# sentence-transformers model for embeddings (local)

EMBED\_MODEL\_NAME = "all-MiniLM-L6-v2" # small & fast; 384-dim

embedder = SentenceTransformer(EMBED\_MODEL\_NAME)

# read sample docs and create chunk list (very small example)

DOC\_FILE = "sample\_docs.txt"

if not os.path.exists(DOC\_FILE):

raise RuntimeError(f"Create {DOC\_FILE} with some sample knowledge text.")

with open(DOC\_FILE, "r", encoding="utf-8") as f:

raw\_text = f.read().strip()

# naive chunking: split by lines for demo (replace with better splitter in prod)

doc\_chunks = [line.strip() for line in raw\_text.split("\n") if line.strip()]

# precompute embeddings for chunks (numpy arrays)

chunk\_embeddings = np.vstack([embedder.encode(c, convert\_to\_numpy=True) for c in doc\_chunks])

# helper: cosine similarity

def cosine\_similarity\_matrix(query\_vec: np.ndarray, matrix: np.ndarray) -> np.ndarray:

# returns 1D array of cosine similarity

q = query\_vec / (np.linalg.norm(query\_vec) + 1e-12)

M = matrix / (np.linalg.norm(matrix, axis=1, keepdims=True) + 1e-12)

return (M @ q).reshape(-1)

# ---------------------------

# Example functions (tools) you expose to the model

# ---------------------------

def get\_current\_time(arguments: Dict[str, Any]) -> Dict[str, Any]:

"""Example tool - returns a simple timestamp string (no external APIs)."""

from datetime import datetime, timezone

tz = timezone.utc

now = datetime.now(tz).isoformat()

return {"timestamp": now, "note": "UTC time returned"}

def lookup\_product(arguments: Dict[str, Any]) -> Dict[str, Any]:

"""

Example product lookup tool. Input args expected:

{ "product\_id": "X" }

This is a demo stub: returns hardcoded info for product\_id "X".

"""

pid = arguments.get("product\_id")

if not pid:

return {"error": "product\_id missing"}

# demo data

demo\_db = {

"X": {"name": "Product X", "price": "$29.99", "launch": "July 2025"},

"Y": {"name": "Product Y", "price": "$59.99", "launch": "Dec 2024"},

}

return demo\_db.get(pid, {"error": f"product {pid} not found"})

# map function name -> python callable for execution

FUNCTION\_REGISTRY = {

"get\_current\_time": get\_current\_time,

"lookup\_product": lookup\_product,

}

# function specs we expose to the model (JSON schema per OpenAI function-calling pattern)

# The model will use these specs to decide which function to call and to generate args

FUNCTION\_SPECS = [

{

"name": "get\_current\_time",

"description": "Return current UTC time and a short note",

"parameters": {

"type": "object",

"properties": {},

},

},

{

"name": "lookup\_product",

"description": "Lookup product metadata by product\_id (demo stub).",

"parameters": {

"type": "object",

"properties": {

"product\_id": {"type": "string", "description": "Product id, e.g., 'X' or 'Y'"},

},

"required": ["product\_id"],

},

},

]

# ---------------------------

# FastAPI app + request models

# ---------------------------

app = FastAPI(title="RAG Chat + Function Calling Demo")

class ChatRequest(BaseModel):

user\_input: str

top\_k: int = 3 # how many chunks to retrieve for grounding

class ChatResponse(BaseModel):

answer: str

used\_chunks: List[Dict[str, Any]]

function\_calls: List[Dict[str, Any]] = []

# ---------------------------

# Utility: build messages with retrieval context

# ---------------------------

def build\_messages\_with\_context(user\_input: str, top\_k: int) -> (List[Dict[str,str]], List[Dict[str,Any]]):

"""

1) compute query embedding and retrieve top\_k chunks

2) build messages list: system message -> user message that includes retrieved context

Returns: (messages, used\_chunk\_info)

"""

q\_vec = embedder.encode(user\_input, convert\_to\_numpy=True)

sims = cosine\_similarity\_matrix(q\_vec, chunk\_embeddings)

top\_idx = np.argsort(-sims)[:top\_k] # highest similarity first

used\_chunks = []

retrieved\_texts = []

for i in top\_idx:

used\_chunks.append({"index": int(i), "text": doc\_chunks[int(i)], "score": float(sims[int(i)])})

retrieved\_texts.append(f"- {doc\_chunks[int(i)]} (score={float(sims[int(i)]):.3f})")

# system message to control model behavior

system\_msg = {

"role": "system",

"content": (

"You are a helpful assistant. Use ONLY the provided CONTEXT when giving factual answers. "

"If the user asks for an action that needs a tool, and if a tool is available, you may call it. "

"When calling tools, please follow the function signature precisely."

)

}

# include the retrieved context inside the user prompt so the model is grounded

context\_block = "\n".join(retrieved\_texts) if retrieved\_texts else "No context available."

user\_msg = {

"role": "user",

"content": f"CONTEXT:\n{context\_block}\n\nUSER QUESTION: {user\_input}\n\nAnswer using only the CONTEXT above. If you need to call an available function, do so."

}

return [system\_msg, user\_msg], used\_chunks

# ---------------------------

# Core chat endpoint with function calling loop

# ---------------------------

@app.post("/chat", response\_model=ChatResponse)

def chat\_endpoint(req: ChatRequest):

user\_input = req.user\_input

top\_k = int(req.top\_k or 3)

if not user\_input.strip():

raise HTTPException(400, "user\_input is empty")

# 1) retrieval + build messages

messages, used\_chunks = build\_messages\_with\_context(user\_input, top\_k=top\_k)

# 2) first model call: allow it to call functions (tools)

try:

response = client.chat.completions.create(

model="gpt-4o-mini", # replace with a chat-capable model you have access to

messages=messages,

functions=FUNCTION\_SPECS,

function\_call="auto", # let the model decide whether to call a function

temperature=0.0,

max\_tokens=500,

)

except Exception as e:

raise HTTPException(status\_code=500, detail=f"OpenAI API error: {e}")

# The API returns choices; grab the assistant message

choice = response.choices[0]

assistant\_message = choice.message

# Track function calls we made

function\_calls\_log: List[Dict[str, Any]] = []

# If model wants to call a function, it'll include 'function\_call' in the assistant message

if assistant\_message.get("function\_call"):

fn = assistant\_message["function\_call"]

fn\_name = fn["name"]

fn\_args\_json = fn.get("arguments") or "{}"

# model generates arguments as a JSON string; parse safely

try:

fn\_args = json.loads(fn\_args\_json) if isinstance(fn\_args\_json, str) else fn\_args\_json

except Exception:

fn\_args = {}

# Record that model requested this function

function\_calls\_log.append({"requested": fn\_name, "arguments": fn\_args})

# Execute the function locally (if registered)

func = FUNCTION\_REGISTRY.get(fn\_name)

if not func:

tool\_result = {"error": f"Function {fn\_name} not implemented on server."}

else:

try:

tool\_result = func(fn\_args)

except Exception as e:

tool\_result = {"error": f"function execution failed: {e}"}

# Append the tool output as a message with role "tool" (or "function") so the model can see the result

# Historically APIs use role "function" or "tool" depending on product; we include as assistant-visible message:

messages.append({

"role": "function",

"name": fn\_name,

"content": json.dumps(tool\_result)

})

# 3) call the model again so it can produce a final answer using the function output + context

try:

followup = client.chat.completions.create(

model="gpt-4o-mini",

messages=messages,

temperature=0.0,

max\_tokens=500,

)

except Exception as e:

raise HTTPException(status\_code=500, detail=f"OpenAI API error on followup: {e}")

final\_choice = followup.choices[0]

final\_msg = final\_choice.message

answer\_text = final\_msg.get("content") or ""

# log the function output too

function\_calls\_log[-1]["result"] = tool\_result

else:

# model produced a direct assistant reply; no function call needed

answer\_text = assistant\_message.get("content") or ""

# return answer, what chunks were used and function call log

return ChatResponse(answer=answer\_text, used\_chunks=used\_chunks, function\_calls=function\_calls\_log)

**How it works — short walkthrough**

1. **Startup**
   * Loads sample documents from sample\_docs.txt.
   * Embeds each chunk with sentence-transformers and caches embeddings in memory.
2. **/chat** POST
   * You send {"user\_input":"...","top\_k":3}.
   * Server computes embedding for the query, finds top-k similar chunks by cosine similarity.
   * Builds a message list: system message (instructions) + user message that **contains the retrieved context**. This is the RAG grounding step.
   * Calls OpenAI chat completions with functions= (the JSON specs), and function\_call="auto". The model may:
     + Return an ordinary assistant reply (done), or
     + Return a function\_call object (name + arguments). The server:
       - Parses arguments, executes the matching local function, appends the function output as a function message, then calls the model again to get the final answer (the function-calling loop). This matches the recommendation in OpenAI docs for tool/function calling. [OpenAI Platform](https://platform.openai.com/docs/assistants/tools/function-calling/quickstart?utm_source=chatgpt.com)[OpenAI Cookbook](https://cookbook.openai.com/examples/how_to_call_functions_with_chat_models?utm_source=chatgpt.com)
3. **Response**
   * The endpoint returns the final assistant text (answer), the used\_chunks that grounded the answer, and a function\_calls array describing function requests and results.

**Quick test with curl**

bash

curl -X POST "http://127.0.0.1:8000/chat" -H "Content-Type: application/json" \

-d '{"user\_input":"When will product X launch? What is its price?", "top\_k": 3}'

* If the model decides it needs more precise product info it may call lookup\_product (demonstrated).
* If it decides no function is necessary, it will answer directly using the retrieved context.

**Notes, tips & safety**

* **Model selection**: I used "gpt-4o-mini" in the example — replace with a model you have access to (e.g., gpt-4o, gpt-3.5-turbo, or other chat-capable model). The function-calling parameters and response structure follow OpenAI docs. [OpenAI Platform](https://platform.openai.com/docs/api-reference/chat?utm_source=chatgpt.com)
* **Prompt design**: system message instructs the model to **use only provided context** for factual answers — that helps reduce hallucination.
* **Function specs**: make the function descriptions and parameter schemas precise to help the model pick the right tool. The OpenAI Cookbook has many function-calling examples. [OpenAI Cookbook](https://cookbook.openai.com/examples/how_to_call_functions_with_chat_models?utm_source=chatgpt.com)
* **Security**: validate/limit function arguments before executing functions that access sensitive resources. Never execute arbitrary code supplied by the model.
* **Scaling**: this example uses in-memory embeddings and small models for demonstration. For production, persist embeddings in a vector DB (FAISS, Pinecone, pgvector) and scale OpenAI usage via proper batching, caching, and monitoring.
* **Costs**: each chat call + follow-up call (when function called) consumes model tokens — design your loop to minimize unnecessary repeat calls.

If you want I can next:

* Swap retrieval to **pgvector** or **FAISS** and show the persistent + scalable variant.
* Replace the example lookup\_product with a safe SQL-backed product lookup (and show how to validate args before calling).
* Provide a React UI that talks to /chat and streams tokens to show partial model output.

Which one would you like next?