Generative AI

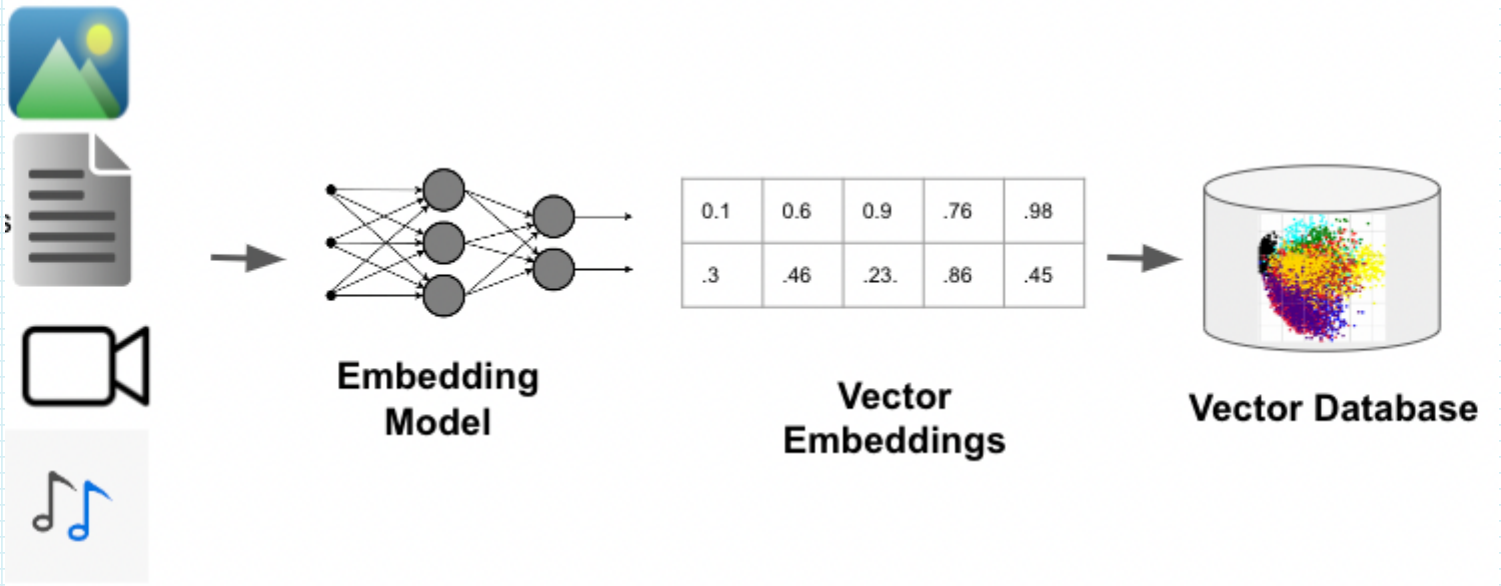
Trainer: Nilesh Ghule

Nearest Neighbor Search

- Problem: You have a query embedding and a collection of document embeddings. Find the closest ones.
- Solution: Nearest Neighbor (NN) Search.
- Brute-Force: Compare query to every vector, sort by similarity, take top-k.
  - Brute-force breaks down with scale: 1M docs * 1536 dimensions = 1.5B operations per query.
- Vector databases (Chroma, FAISS, Pinecone) exist to make NN search fast.
  - They use Approximate Nearest Neighbor (ANN) algorithms.
  - NN search is just math + ranking. No AI, no magic.
  - Why Top-k? Context is fuzzy. RAG needs several relevant chunks, not just one.
  - Trade-off: Accept a tiny, often imperceptible, loss in accuracy for a massive speed gain.
- Note: ANN is like a smart GPS vs. walking to every house (brute-force).

Chroma DB - A Vector Database, Demystified

- At its core: Chroma is a database for fast nearest neighbor search on vectors.
- Chroma is a high-performance index over 'meaning-space.'
- What Chroma DOES:
  - Stores embeddings,
    associated text,
    and metadata.
  - Performs fast ANN search.
  - Persists data to disk.
- What Chroma does NOT do:
  - Create embeddings (you provide them).
  - Understand language or generate text.



Chroma Core Concepts

- 1. Collection: A container for a set of vectors, like a database table.
- 2. What's Stored Per Entry:
  - ids given by you.
  - document given by you.
  - embeddings created by the embedding models.
  - metadata given by you - very helpful in searching and management.

```
{
  "id": "doc1",
  "embedding": [0.01, 0.98, ...],
  "document": "Soccer players train daily",
  "metadata": {"topic": "sports"}
}
```

- 3. Similarity Search: Its only runtime job. Query with an embedding, get back top-k closest documents + metadata.
- 4. Persistence: Saves vectors to disk. Reloads instantly. Essential for production.

Chroma in action - Getting ready

```
cmd> pip install langchain chromadb sentence-transformers
```

```python
import chromadb
from langchain.text_splitter import RecursiveCharacterTextSplitter
from langchain.embeddings import HuggingFaceEmbeddings
```

```python
# Embedding model using LangChain
embed_model = HuggingFaceEmbeddings(model_name="sentence-transformers/all-MiniLM-L6-v2")
# Chunking using LangChain
text_splitter = RecursiveCharacterTextSplitter(chunk_size=500,chunk_overlap=100)
```

```python
client = chromadb.Client(settings=chromadb.Settings(persist_directory="./chroma_db"))
# chromadb.Client() default is in-memory store
collection = client.get_or_create_collection(name="demo")
```

```python
chunks = text_splitter.split_text(raw_text)
embeddings = embed_model.embed_documents(chunks)
# Prepare metadata & IDs
ids = [f"doc_{i}" for i in range(len(chunks))]
metadatas = [ {"source": "example.txt", "chunk_id": i} for i in range(len(chunks)) ]
```

Chroma in action - CRUD operations

```
# Add to Chroma
collection.add(ids=ids, documents=chunks, embeddings=embeddings metadatas=metadatas)
client.persist()
```

```
# READ (Similarity Search)
query = "How do soccer players train?"
query_embedding = embed_model.embed_query(query)

results = collection.query(query_embeddings=[query_embedding], n_results=2)
# get top 2 results -- with similarity to given query_embedding

# Inspect the results
for doc, meta in zip(results["documents"][0], results["metadatas"][0]):
    print(meta, "→", doc)
```

```
# Optional - Observe the differences
for distance in results["distances"][0]:
    print(distance)
```

Chroma in action - CRUD operations

```
# Update = Delete + Re-insert

# Vector DBs do **not** support in-place updates.
collection.delete(ids=["doc_1"])

# Deletes the embedding with given id
updated_text = "Professional soccer players train daily with coaches."
updated_embedding = embed_model.embed_documents([updated_text])

# Build embedding for updated input data and persist it
collection.add(ids=["doc_1"], documents=[updated_text], embeddings=updated_embedding,
metadatas=[{"source": "example.txt", "chunk_id": 1}])
client.persist()

# Note: Embeddings are immutable
# Change text → regenerate embedding
```

```
# DELETE by metadata filter
collection.delete(where={"source": "example.txt"})
```

Langchain Document Loaders

- Data is fetched from the source (so that it can be converted to embedding & store into vector store). Fetched data is collected as langchain Documents.
- File-based Loaders
  - TextLoader: Loads plain text (.txt) files. Use when you just have simple text.
  - CSVLoader: Loads structured tabular data from CSV files. Data with metadata columns.
  - PDF Loaders - PyPDFLoader / PDFLoader: for standard PDF text extraction.
  - UnstructuredPDFLoader: for richer extraction with better layout handling.
  - LangChainDocxLoader: Loads Word .docx files into Documents.
  - LangChainDirectoryLoader: A wrapper loader that walks a directory and applies another loader to all matching files (e.g., PDF + TXT + DOCX).
- Audio & Video Transcript Loaders:
  - YouTubeLoader: Fetches and parses transcripts from YouTube videos.
  - AssemblyAIAudioLoaderById / SonixLoader: Transcribe audio/video (via external APIs like AssemblyAI or Sonix).

Langchain Document Loaders

- Web & URL Loaders: Load content from the web or online sources:
  - WebBaseLoader / UnstructuredURLLoader: Load text from web pages by URL. Useful for scraping single pages.
  - RecursiveURLLoader: Recursively crawls links under a root domain (great for deeper scraping).
  - SitemapLoader: Reads all linked pages from a sitemap
- Tips on Choosing Loaders
  - For file loading:
    - To ingest local files, begin with TextLoader, CSVLoader, or a PDF loader.
    - For directories: Use DirectoryLoader to batch-load many files in one shot.
  - For web content:
    - Use WebBaseLoader or more advanced crawlers like RecursiveURLLoader for deeper scraping.
  - For rich/unstructured extraction:
    - Loaders based on the Unstructured ecosystem (e.g., UnstructuredPDFLoader)

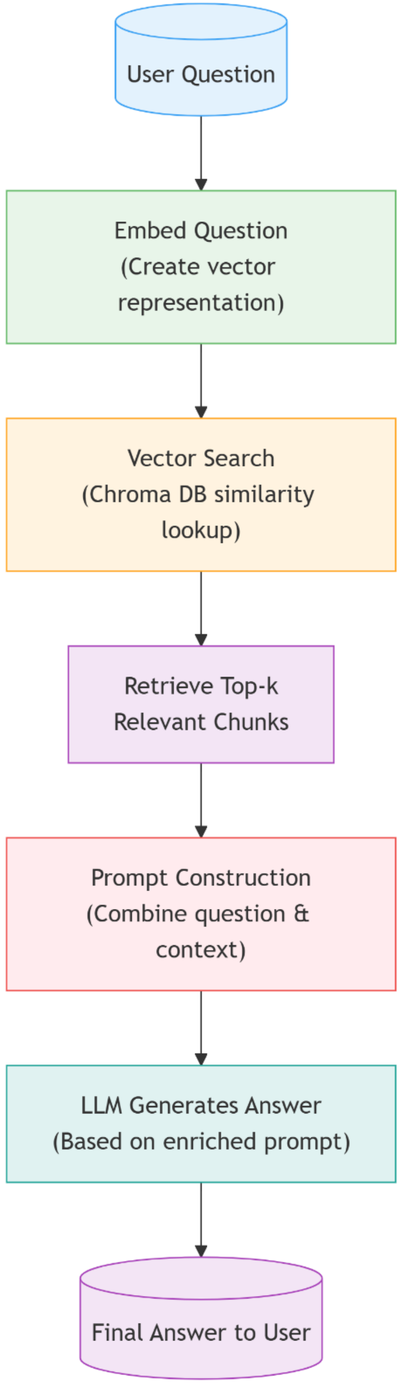
Langchain Document Loader in Action

```
from langchain.document_loaders import PyPDFLoader
from langchain.text_splitter import RecursiveCharacterTextSplitter

# load a PDF document
loader = PyPDFLoader("/path/to/example.pdf")
docs = loader.load()

# read document pages one by one directly
for page in docs:
  print(page.page_content)
  print(page.metadata) # { "source": "example.pdf", "page": 0 }

# common practice to use text splitter after loading docs
text_splitter = RecursiveCharacterTextSplitter(chunk_size=500, chunk_overlap=50)
chunks = text_splitter.split_documents(docs)
# read chunks
for chunk in chunks:
  print(chunk.page_content)
```



RAG Architecture

- RAG = Retrieval-Augmented Generation
- Definition: Using retrieved text as context for an LLM.
  - The LLM does no searching.
  - The vector DB does no reasoning.

<= Core Pipeline

- RAG is composition of 4 systems
  1. **Ingestion (offline)**: Raw docs -> Chunks -> Embeddings -> Chroma
  2. **Retrieval (runtime)**: Question -> Embedding -> Similarity Search
  3. **Prompt Assembly (critical)**: Context + Question + Rules
  4. **Generation (LLM)**: LLM answer using context only (ideally)

Nilesh Ghule

The Heart of RAG: The Prompt

- Prompt assembly is where most RAG systems fail. You must give the LLM explicit rules.
- Bad Prompt (Leads to Hallucination):

```
Answer the question: {question}
Context: {context}
```

- Good Prompt (Minimum Viable - Provides Guardrails):

```
You are a helpful assistant.

Answer the question using ONLY the context below.
If the answer is not in the context, say "I don't know".

Context:
{context_chunks_joined_with_newlines}

Question:
{user_question}

Answer:
```

Rules beat intelligence.
Smaller LLM with strong, explicit instructions will outperform a giant, unfettered model.
This prompt structure forces grounding and reduces 'hallucinations'.

Nilesh Ghule

Key RAG Truths

- Truth #1: RAG quality is dominated by retrieval, not the LLM.
  - Garbage in, garbage out. Even for GPT-4 or Claude.
  - Bigger models just hallucinate better
- Truth #2: RAG is not "asking the LLM to search".
  - The LLM does zero retrieval.
  - If Chroma returns irrelevant chunks,
    the LLM will confidently answer wrong based on them.
- Truth #3: RAG answers are only as good as your chunks
  - Chunking > embeddings > prompt > model

Summary: Your Solid Foundation

- You've built a layered understanding:
  - Embeddings convert meaning to vectors (geometry).
  - Similarity Search (Cosine) finds close vectors.
  - Vector Databases (Chroma) scale this search via ANN.
  - Chunking creates the right-sized 'knowledge units' for retrieval.
  - RAG cleanly composes retrieval + generation with explicit rules.
- Now you build, debug, and optimize real RAG systems.

Thank You!

Nilesh Ghule

<nilesh@sunbeaminfo.com>