Section 2: Core Components in RAG
Course: Ultimate RAG Bootcamp Using LangChain, LangGraph and LangSmith
Section Number: 2
Total Videos: 2
Date Created: 2024

---

Video 1: Data Ingestion and Parsing
Video Order: 1/2
Topics: Document ingestion, Pre-processing, Chunking, Embeddings, Vector databases, Similarity search, Cosine/Euclidean distance
Difficulty: Beginner

Content
Hello guys.

So we are going to continue the discussion of Retrieval-Augmented Generation (RAG). In this specific video, we'll dive into the **core components** of a RAG pipeline. By now, you already have an intuition for how RAG works at a high level: we have a Large Language Model (LLM), we augment it with external knowledge stored in a **vector database**, and the LLM uses retrieved context from that database to generate better answers.

At a glance, when I provide an input to a plain LLM, it just generates an output from its internal knowledge. In RAG, however, the LLM is **connected to a vector database**. We store information in that database **as vectors** (numerical representations). When we send a question, we first **retrieve** relevant information from the vector database, pass that retrieved context to the LLM, and **then** the LLM generates a summarized, grounded output using that context.

From the architectural diagram we discussed earlier, there are **three main phases** in a full RAG system:

1. **Document Ingestion Phase**
2. **Query Processing Phase**
3. **Generation Phase**

This video focuses on **Phase 1: Document Ingestion and Pre-processing**. We'll break down what data we ingest, how we clean and split it, how we embed it into vectors, and how we store it in a vector database so that later phases (query + generation) can work effectively.

---

## What is Document Ingestion and Pre-processing?

To power the retriever, we first need a **vector database** filled with vectors that represent our knowledge. This knowledge can come from **multiple sources**: company policies, internal documents, PDFs, Word docs, CSVs, websites, databases, images (with extracted text), and more. The ingestion pipeline's job is to:

- **Load** data from these sources
- **Pre-process** the raw text (cleaning, normalization)
- **Split** documents into **chunks**
- **Embed** each chunk into a **vector**
- **Store** the vectors (with metadata) in a **vector database**

That's the core of Phase 1.

---

## Common Data Sources

Your data might be:

- **PDFs**
- **Word Documents (.docx)**
- **Text files (.txt)**
- **Spreadsheets (CSV/Excel)**
- **Web pages** (scraped or via loaders)
- **Databases** (records turned into text fields)

We will read all of these using loaders, normalize them into a consistent text representation, and then pass them through a standardized pipeline.

---

## The Ingestion Pipeline: Step by Step

### Step 1: Load the Documents

Use appropriate loaders per file type to extract text and attach preliminary metadata (e.g., source, filename, section, url).

**Step 2: Pre-process the Text**
Light cleaning (remove boilerplate, headers/footers if noisy, fix line breaks, normalize whitespace, optionally remove very long tables or binary debris) so the chunks are clean and meaningful.

**Step 3: Split into Chunks**
We don't store entire documents as a single vector. We **split** documents into **smaller chunks** (e.g., ~500 tokens with some overlap). This matters because every LLM has a **context window limit**—only so much text can be fed to the model at once. Smaller, coherent chunks improve both **retrieval accuracy** and **downstream answer quality**.

- Think of chunks as paragraphs or sections sized for retrieval.
- Overlap (e.g., 50–100 tokens) helps preserve continuity across chunk boundaries.

**Why chunking is necessary:**
If you try to hand the LLM a thousand-page book as context, it won't fit. Even if it did, the relevant part might be buried. Chunking allows the retriever to **pinpoint** just the passages most relevant to the question, keeping prompts concise and precise.

**Step 4: Embed Each Chunk**
We pass each chunk to an **embedding model** that converts text into a **vector** (a list of numbers). For example, a sentence might turn into a vector like [0.6, 0.5, 0.4, 0.1, 0.7, ...]. These vectors capture semantic meaning, enabling similarity comparisons.

- You can use **OpenAI** embeddings or **Hugging Face** open-source models.
- The embedding model's job: map semantically similar text to nearby points in vector space.

**Step 5: Store in a Vector Database**
We store each vector **along with its metadata** (e.g., source, section, page_number, chunk_id, timestamp, keywords). Popular vector stores include:

- **ChromaDB**
- **FAISS** (library)
- **Pinecone** (managed)
- **DataStax / Cassandra with vector support**

This database supports **similarity search** so we can find the most relevant chunks for a given query later.

---

**Similarity Search and Distance Metrics**

Once vectors are stored, the retriever uses **similarity search** to find chunks related to a query. Common distance/similarity measures:

- **Cosine similarity** (most common for embeddings)
- **Euclidean distance**

Given a user question like, *"What is an LLM?"*, the retriever embeds the question into a vector and searches for the **nearest** vectors in the database. The top-k matching chunks are returned as **context** for the LLM.

---

**End-to-End View of Phase 1**

1. **Data Sources** → PDFs, docs, CSVs, websites, databases
2. **Loaders + Cleaning** → normalize and prepare text
3. **Split into Chunks** → manageable, overlapping units
4. **Embed Chunks** → text → vectors using an embedding model
5. **Store in Vector DB** → vectors + metadata (for retrieval)

When this pipeline completes, you have a **ready-to-query** vector database. In the **next video**, we'll cover **Query Processing**: how the system embeds an incoming user question, how it performs the similarity search, how we select and format the retrieved context, and how that context flows into the LLM for the **Generation Phase**.

Thank you!

---

Video 2: Query Processing and Output Generation Phase
Video Order: 2/2
Topics: Query processing, Query embedding, Similarity search, Retrieval, Context enrichment, Generation, LLMs (OpenAI, Llama, Gemini), Summarized answers
Difficulty: Beginner

Content
Hello guys.

So we are going to continue our discussion with respect to our core components in RAG. We have already discussed about document ingestion and pre-processing. We understood what all specific steps we take in the document ingestion phase, wherein we focus on finding the sources of the data. Then we use document splitter to convert that into chunks. Then with the help of an embedding model, we convert those chunks into vectors, and we store these vectors in the vector database.

So in this video we are going to focus on the second important step—that is, the **query processing phase**. Initially here, we are focusing more on the theoretical concepts, but as we go ahead we will implement each and everything, from the data ingestion pipeline to the query processing phase to the generation phase. Then we will try to build better RAG architectures by using these concepts.

Now in the second important step, we are really going to focus on something called a **query**. If you see over here, these are the steps that we are going to do: the **query processing phase**.

In the query processing phase, what all things will we be learning as we go ahead? Till now you know that from our data ingestion pipeline, **my vector database is ready**. Let's say I draw a vector database—this is the vector database that is ready. You know how this vector database has been created: we used the data ingestion pipeline. In that pipeline, we had our source data, then we did chunking, then we converted chunks into vectors, and finally we stored all of this into the vector database.

Now, in the query processing phase, we first of all take an **input query**. So here I will be having my input query. For example, the input is: *"What is RAG?"* If all the information stored in the vector database is related to RAG, the system will operate as follows.

**Step 1: Embed the input query**
The input query text (e.g., "What is RAG?") will be **converted into a vector** using **embedding models**. After this step, we obtain a numeric vector representation—some

numbers like 0.3, 0.4, 0.6, etc., up to the embedding dimension. This is the initial first phase of query processing: converting text to vectors.

**Step 2: Retrieve similar chunks from the vector database**
Once we have the query vector, we **send a request to the vector database** and apply similarity-based search. Techniques include **cosine similarity** and **Euclidean distance**, among others. The main aim is to **find the most relevant chunks** that match the query vector.

Based on this search, we retrieve multiple chunks—say **Document 1**, **Document 2**, **Document 3**—that best match the input query. These are the **retrieved results** returned by the vector database.

**Step 3: Build the augmented context**
These retrieved results are then used to form a **context** for the LLM. With Retrieval-Augmented Generation, we often **enrich** the retrieved context by adding **metadata** (source, section, page, timestamps, etc.) or applying light transformations so that the LLM receives cleaner, more targeted information. This enrichment helps the LLM produce a **more accurate** and grounded output.

At the end of query processing, we have: **original query + retrieved chunks (+ enriched metadata/context)**. This bundle is what we pass forward to the next stage—**the Generation Phase**.

---

**Generation Phase (overview)**

In the **generation phase**, whatever context we have built from the retriever is passed to the **LLM**. We can use different LLMs: **OpenAI** models, **Llama** (open source), **Google Gemini**, and others. The LLM receives **(original query + enriched context)** and then produces a **summarized output**—a final answer phrased naturally, like how a human would write.

This is how the **entire pipeline** of a RAG system looks in practice:

- **Document Ingestion & Pre-processing** → build the vector database from sources
- **Query Processing** → embed query, retrieve the most relevant chunks, enrich context
- **Generation** → feed query + context to the LLM and produce the final answer

With this, we generate an output and return it to the user. The best part is that the LLM can summarize and structure the answer in a very natural, conversational way.

---