Retrieval Augmented Generation

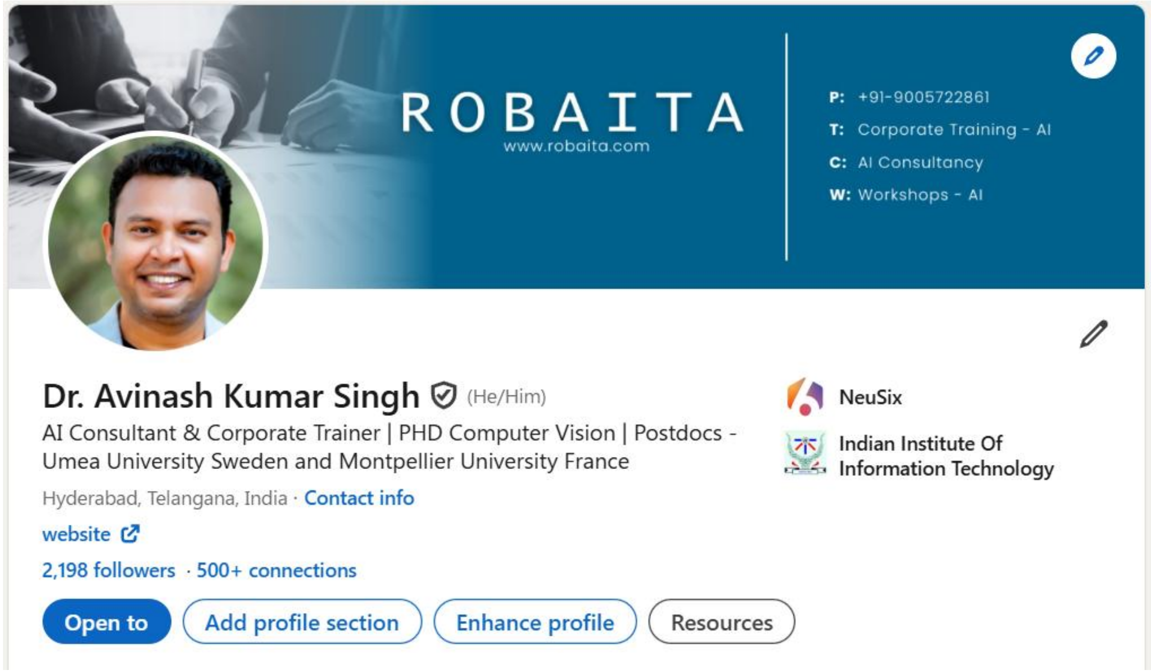
Infusing Context into Language Models for Smarter Responses

Dr. Avinash Kumar Singh

AI Consultant and Coach, Robaita

Dr. Avinash Kumar Singh

- ❑ **Possess** 15+ years of **hands-on expertise** in Machine Learning, Computer Vision, NLP, IoT, Robotics, and Generative AI.
- ❑ **Founded** Robaita—an initiative **empowering** individuals and organizations to **build, educate, and implement** AI solutions.
- ❑ **Earned** a Ph.D. in Human-Robot Interaction from IIIT Allahabad in 2016.
- ❑ **Received** postdoctoral fellowships at Umeå University, Sweden (2020) and Montpellier University, France (2021).
- ❑ **Authored** 30+ research papers in **high-impact** SCI journals and international conferences.
- ❑ Unlearning, learning, making mistakes …



Dr. Avinash Kumar Singh (He/Him)
AI Consultant & Corporate Trainer | PHD Computer Vision | Postdocs - Umea University Sweden and Montpellier University France
Hyderabad, Telangana, India · Contact info
website
2,198 followers · 500+ connections
NeuSix
Indian Institute Of Information Technology

https://www.linkedin.com/in/dr-avinash-kumar-singh-2a570a31/

Discussion Points

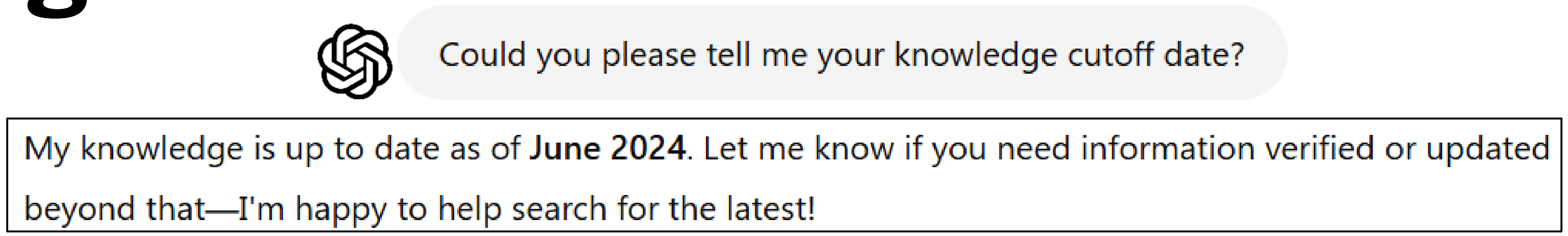
- RAG Pipeline
    - Data Ingestion
    - Chunking
    - Embedding
    - Vector Databases
    - Retrieval
    - Prompt Engineering
    - Response Generation

What is RAG

Retrieval Augmented Generation

- **Retrieval:** The system searches a knowledge base using a query.
- **Augmented:** The retrieved content is added to the prompt, enriching the LLM with real-time, external knowledge.
- **Generation:** The LLM uses the augmented context to generate a coherent, relevant, and factual response.

Retrieval Augmented Generation



Could you please tell me your knowledge cutoff date?

My knowledge is up to date as of **June 2024**. Let me know if you need information verified or updated beyond that—I'm happy to help search for the latest!

- ❑**Proposed by:** Facebook AI Research (FAIR)
- ❑**Introduced in:** 2020
- ❑**Key Paper:** Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks [1]
- ❑**Authors:** Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, Douwe Kiela
- ❑**Motivation:** To overcome the limitations of closed-book language models in knowledge-intensive tasks by integrating a retriever component.

[1] NeurIPS 2020, https://arxiv.org/abs/2005.11401

Retrieval Augmented Generation

Why it is useful?

- ❑Keeps models up-to-date without retraining
- ❑Cost-effective and scalable
- ❑Ensures traceability and factual correctness

Retrieval Augmented Generation

Applications

- ❑Enterprise Knowledge Assistants
  - ❑Internal document Q&A over policies, manuals, SOPs.
- ❑Legal Document Review
  - ❑Cases retrieval and summary generation from legal archives.
- ❑Healthcare Support
  - ❑Medical chatbot retrieving treatment guidelines and summarizing research.
- ❑Education and Research
  - ❑Academic assistant answering syllabus-based questions with citations.
- ❑E-commerce Search & Support
  - ❑Product search, reviews, and spec-based query response.

LangChain

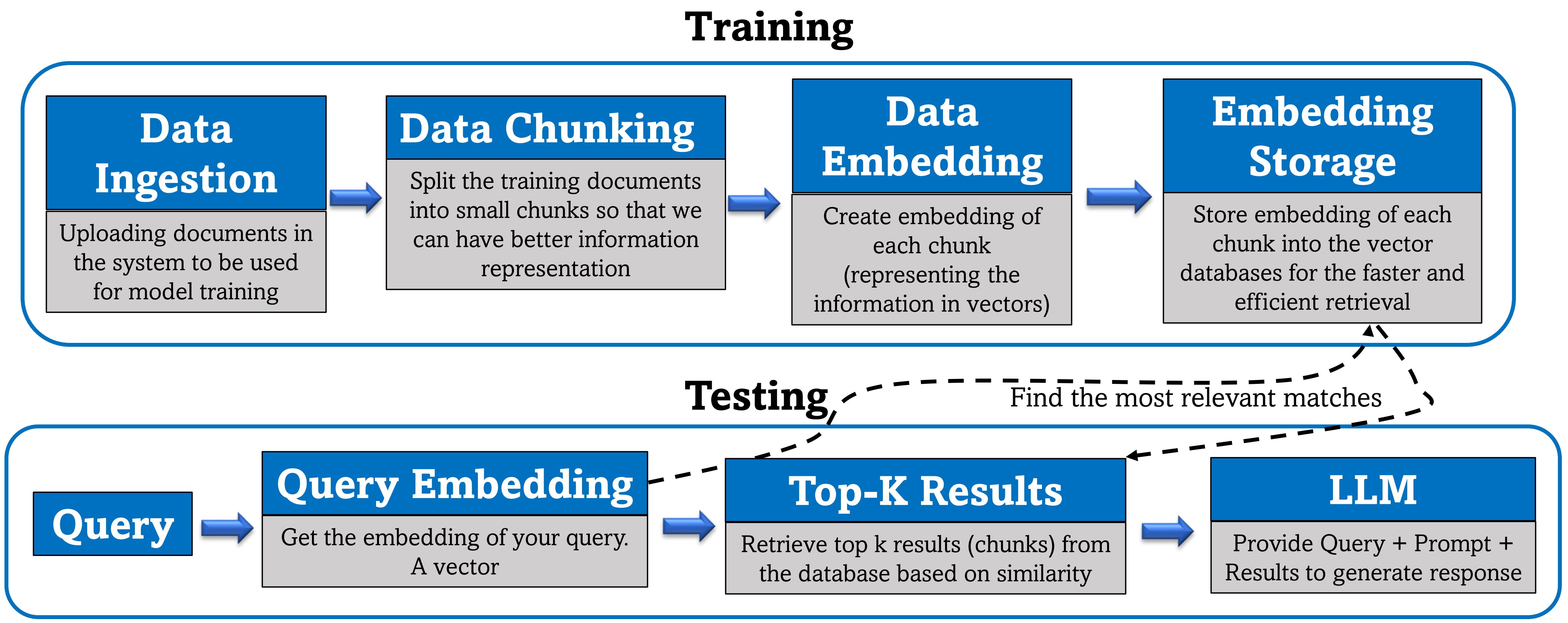
LangChain is a Python (and JavaScript) framework (open source) designed to help developers build applications that use large language models (LLMs) more effectively by **chaining** components together.

It helps build complex LLM applications by combining:

- Prompt templates
- Memory (for multi-turn conversations)
- Tools (like Google Search, Python REPL, etc.)
- Retrieval from documents (RAG)
- Chains (sequential workflows)

https://www.langchain.com/

Retrieval Augmented Generation Architecture



Training

Data Ingestion
Uploading documents in the system to be used for model training

Data Chunking
Split the training documents into small chunks so that we can have better information representation

Data Embedding
Create embedding of each chunk (representing the information in vectors)

Embedding Storage
Store embedding of each chunk into the vector databases for the faster and efficient retrieval

Testing

Find the most relevant matches

Query

Query Embedding
Get the embedding of your query. A vector

Top-K Results
Retrieve top k results (chunks) from the database based on similarity

LLM
Provide Query + Prompt + Results to generate response

Data Ingestion

Data ingestion is the first stage of the RAG pipeline. It refers to the **collection, loading, and preparation** of raw data sources (documents, PDFs, websites, databases, etc.) for downstream processing (like chunking and embedding). The ingestion phase ensures:

- Format normalization (e.g., plain text, HTML, Markdown)
- Basic cleaning (e.g., whitespace removal, encoding fix)
- Metadata extraction (e.g., author, title, source URL, page number)

Data Sources

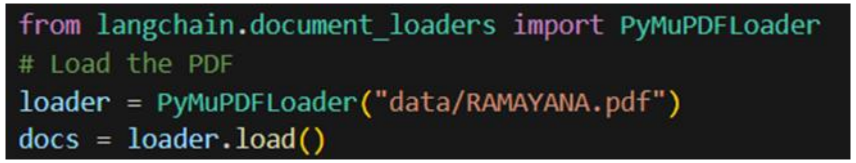
- PDFs, Word docs
- Websites (HTML, blogs)
- Markdown files
- Notion pages
- CSVs, databases
- Cloud storage (Google Drive, S3)

Data Ingestion

Data Ingestion Using LangChain

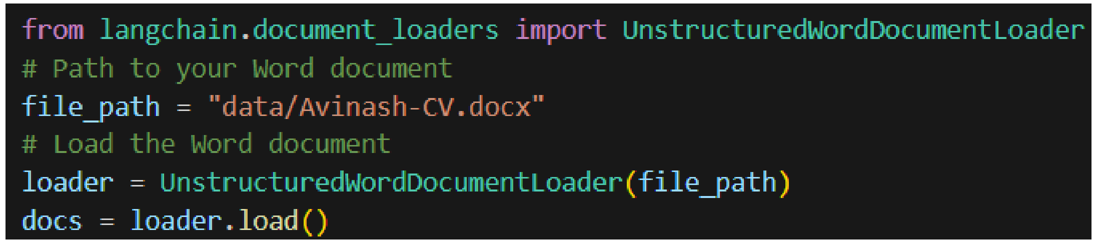
LangChain offers document loaders that simplify ingestion across formats.

PDF Reader



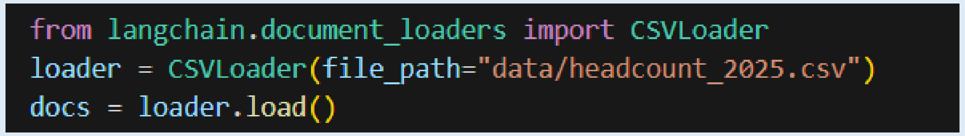
```python
from langchain.document_loaders import PyMuPDFLoader
# Load the PDF
loader = PyMuPDFLoader("data/RAMAYANA.pdf")
docs = loader.load()
```

Document Reader



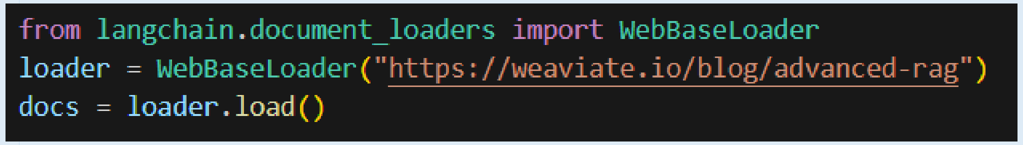
```python
from langchain.document_loaders import UnstructuredWordDocumentLoader
# Path to your Word document
file_path = "data/Avinash-CV.docx"
# Load the Word document
loader = UnstructuredWordDocumentLoader(file_path)
docs = loader.load()
```

CSV Loader



```python
from langchain.document_loaders import CSVLoader
loader = CSVLoader(file_path="data/headcount_2025.csv")
docs = loader.load()
```

Webpage Loader

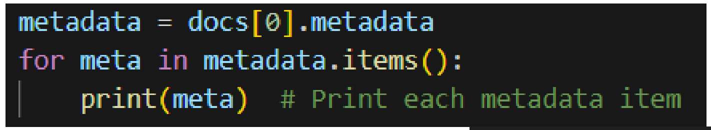


```python
from langchain.document_loaders import WebBaseLoader
loader = WebBaseLoader("https://weaviate.io/blog/advanced-rag")
docs = loader.load()
```

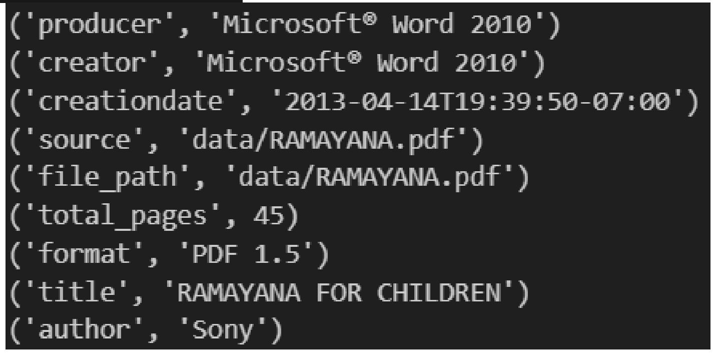
Data Ingestion

Data Type	LangChain Loader	Description
PDFs	PyMuPDFLoader, PDFMinerLoader, UnstructuredPDFLoader	Reads PDFs with text and metadata
Word Documents (.docx)	UnstructuredWordDocumentLoader	Extracts content from Word files
PowerPoint (.pptx)	UnstructuredPowerPointLoader	For presentation slides
Websites (HTML, Blogs)	WebBaseLoader	Scrapes and loads web page content
Markdown (.md)	UnstructuredMarkdownLoader	Parses Markdown files
Notion Pages	NotionDBLoader	Loads from Notion databases
Google Docs	GoogleDriveLoader with file type = "document"	Loads from Google Drive
Google Sheets	GoogleDriveLoader with file type = "spreadsheet"	Loads structured sheets
CSV / Excel Files	CSVLoader, UnstructuredExcelLoader	Reads tabular data
JSON / JSONL	JSONLoader, JSONLinesLoader	Loads structured JSON data
Text Files (.txt)	TextLoader, UnstructuredFileLoader	Basic plain text ingestion
Email (EML, Outlook)	UnstructuredEmailLoader	Parses and ingests email content
S3 (Amazon Cloud)	S3DirectoryLoader, S3FileLoader	Load files from AWS S3 buckets
Local Folders	DirectoryLoader	Bulk load from file directories
YouTube Videos	YoutubeLoader	Transcribes YouTube audio to text
Audio Files (MP3, WAV)	AudioLoader (requires whisper or similar)	Speech-to-text for ingestion
Databases (SQL, Postgres, etc.)	SQLDatabaseChain + custom readers	Loads from relational DBs
Google Drive (multi-file)	GoogleDriveLoader	Handles multiple files
Outlook Calendar, Email, etc.	Microsoft Graph API (custom loaders)	Custom support via API

Data Ingestion



```python
metadata = docs[0].metadata
for meta in metadata.items():
    print(meta)  # Print each metadata item
```



```
('producer', 'Microsoft® Word 2010')
('creator', 'Microsoft® Word 2010')
('creationdate', '2013-04-14T19:39:50-07:00')
('source', 'data/RAMAYANA.pdf')
('file_path', 'data/RAMAYANA.pdf')
('total_pages', 45)
('format', 'PDF 1.5')
('title', 'RAMAYANA FOR CHILDREN')
('author', 'Sony')
```

Extracting and Using Metadata

Metadata can be used for:

- Filtering search results (e.g., by document source, author)
- Re-ranking based on source credibility
- Building traceability & citations in responses

Key Things to Include in Ingestion

Feature	Why It Matters
Text cleaning	Removes noise for better embedding
Metadata enrichment	Helps in filtering and reranking
Format handling	Normalize different file types to plain text
Language detection	Optional — useful for multilingual pipelines
Chunk boundary hints	Marking paragraphs, sections, tables, etc.

Data Chunking

Data Chunking is the process of splitting raw documents into smaller, semantically coherent pieces (chunks) to:

- Improve embedding quality.
- Enhance retrieval precision.
- Reduce token usage for LLMs.

Chunking enables retrieval of only the most relevant pieces of context when answering a query, rather than the entire document.

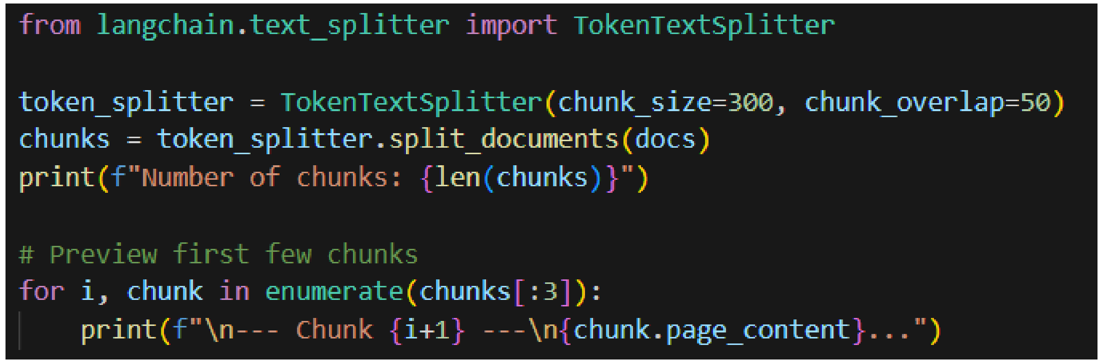
Chunking Strategies

- Length-based (CharacterTextSplitter)
- Text-structured based (RecursiveCharacterTextSplitter)
- Ideal chunk size: 300-500 tokens

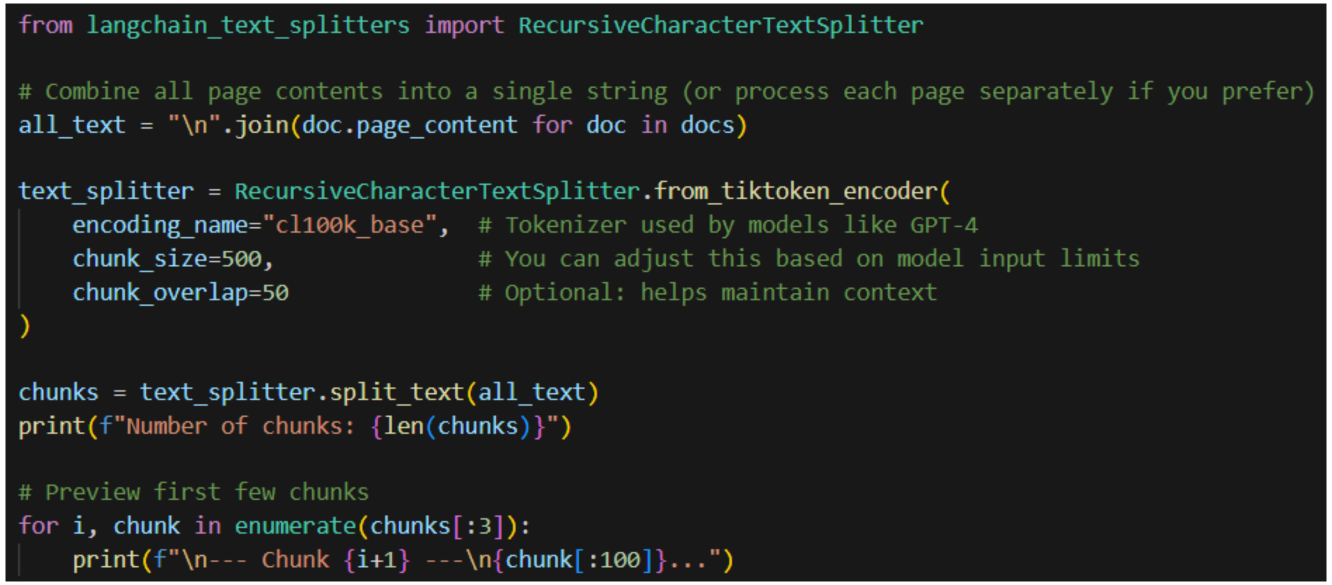
Data Chunking

Length-based Chunking

- Length-based chunking refers to <u>dividing a document into segments of a fixed number of characters, words, or tokens</u>, regardless of the underlying meaning or structure of the text.
- This method can be implemented in two primary ways:
  - **Token-Based Splitting:** Splits text based on the number of tokens, which is particularly useful when working with language models that have token limits.
  - **Character-Based Splitting:** Splits text based on the number of characters, providing consistency across different types of text.



```python
from langchain.text_splitter import TokenTextSplitter

token_splitter = TokenTextSplitter(chunk_size=300, chunk_overlap=50)
chunks = token_splitter.split_documents(docs)
print(f"Number of chunks: {len(chunks)}")

# Preview first few chunks
for i, chunk in enumerate(chunks[:3]):
    print(f"\n--- Chunk {i+1} ---\n{chunk.page_content}...")
```

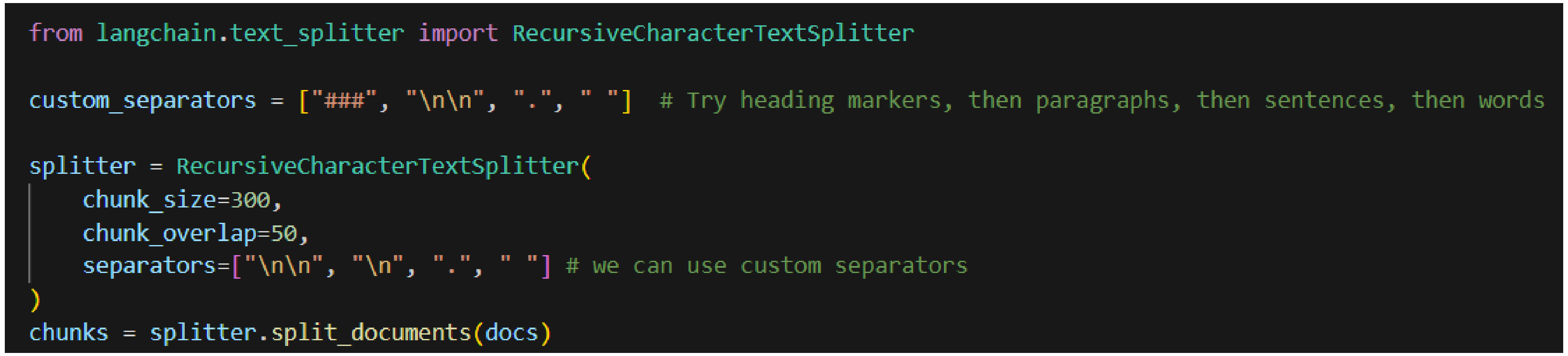


```python
from langchain_text_splitters import RecursiveCharacterTextSplitter

# Combine all page contents into a single string (or process each page separately if you prefer)
all_text = "\n".join(doc.page_content for doc in docs)

text_splitter = RecursiveCharacterTextSplitter.from_tiktoken_encoder(
    encoding_name="cl100k_base",  # Tokenizer used by models like GPT-4
    chunk_size=500,               # You can adjust this based on model input limits
    chunk_overlap=50              # Optional: helps maintain context
)

chunks = text_splitter.split_text(all_text)
print(f"Number of chunks: {len(chunks)}")

# Preview first few chunks
for i, chunk in enumerate(chunks[:3]):
    print(f"\n--- Chunk {i+1} ---\n{chunk[:100]}...")
```

Data Chunking

Text structure-based chunking

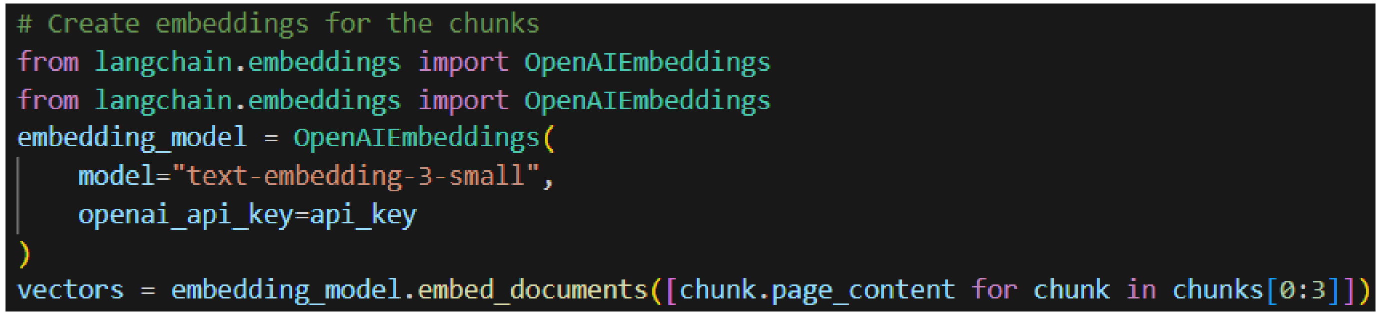
- Leverages the inherent hierarchical organization of text—such as paragraphs, sentences, and words—to create chunks that maintain the natural flow and semantic coherence of the original content.
- It recursively moves to the next smaller unit:
  - First tries to split at sentence boundaries.
  - Then at words.
  - Then at character level (as a last resort)



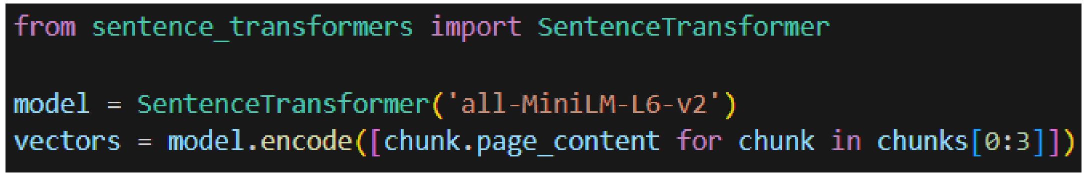
```python
from langchain.text_splitter import RecursiveCharacterTextSplitter

custom_separators = ["###", "\n\n", ".", " "]  # Try heading markers, then paragraphs, then sentences, then words

splitter = RecursiveCharacterTextSplitter(
    chunk_size=300,
    chunk_overlap=50,
    separators=["\n\n", "\n", ".", " "] # we can use custom separators
)
chunks = splitter.split_documents(docs)
```

Chunk Embedding

- An embedding is a numerical representation of data (text, images, etc.) in a high-dimensional vector space.
- It captures the semantic meaning of the content, enabling machines to compute similarity between inputs efficiently.
- After data chunking (breaking documents into smaller coherent units), each chunk is converted into an embedding vector using an embedding model.



```python
# Create embeddings for the chunks
from langchain.embeddings import OpenAIEmbeddings
from langchain.embeddings import OpenAIEmbeddings
embedding_model = OpenAIEmbeddings(
    model="text-embedding-3-small",
    openai_api_key=api_key
)
vectors = embedding_model.embed_documents([chunk.page_content for chunk in chunks[0:3]])
```



```python
from sentence_transformers import SentenceTransformer

model = SentenceTransformer('all-MiniLM-L6-v2')
vectors = model.encode([chunk.page_content for chunk in chunks[0:3]])
```

Chunk Embedding

Comparison of different Embedding Strategies

Strategy / Library	Model	Dim	Language Support	Speed	Use Case
OpenAI	text-embedding-3-small	1536	High	Fast	Best-in-class general purpose
Hugging Face	all-MiniLM-L6-v2	384	Moderate	Fast	Open-source, lightweight
Cohere	embed-english-light-v3	1024	High	Medium	Chatbots, semantic search
Google	GECKO / Universal Sent. Encoder	512	High	Medium	Academic & enterprise use
Self-hosted models	E5, Instructor XL, Opensource Models	Varies	High	Varies	Custom tuning + privacy

Chunk Embedding

Considerations

- Embedding Normalization
  - Normalize vectors before similarity search for accurate cosine comparison.

$$\hat{\mathbf{x}}_i = \frac{\mathbf{x}_i}{\|\mathbf{x}_i\|} = \frac{1}{\sqrt{\sum_{j=1}^{d} x_{ij}^2}} \cdot \mathbf{x}_i$$

- Chunk Labeling
  - Augment chunks with titles/headings to improve context understanding.

```python
from langchain.schema import Document
documents = [
    Document(page_content="AI enables automation of complex tasks.", metadata={"title": "Introduction to AI"}),
    Document(page_content="LLMs are used in chatbots and assistants.", metadata={"title": "Applications of LLMs"})
]
# This metadata can be used later to boost or filter during retrieval
print(documents[0].metadata['title'])  # Output: Introduction to AI
```

- Dynamic Embedding Updates
  - Recompute embeddings periodically if data changes (e.g., live knowledge bases).

```python
if document_updated:
    new_embedding = embedding_model.embed_query(updated_doc_content)
    vectorstore.update_embedding(doc_id, new_embedding)
```

- Multi-modal Embeddings
  - Combine text with images, tables, or audio using multi-modal encoders (e.g., CLIP, BLIP).

```python
model = CLIPModel.from_pretrained("openai/clip-vit-base-patch32")
processor = CLIPProcessor.from_pretrained("openai/clip-vit-base-patch32")
```

Embedding Storage & Retrieval

Vector Database

Vector databases are specialized systems designed to **store, index, and retrieve high-dimensional vectors**—which represent text, images, or other data types in numerical form using techniques like embeddings.

In a RAG pipeline, they are used to:

- Store embedded document chunks (from a retriever)
- Perform similarity search (e.g., cosine similarity)
- Return top relevant chunks to feed into an LLM for context-aware generation

Examples:

- FAISS – A high-performance library by Facebook AI for efficient similarity search on large-scale dense vectors.
- Chroma – A modern, open-source vector database with built-in document and metadata storage, optimized for RAG use cases.
- Pinecone – A fully managed vector database service built for production-ready, low-latency similarity search at scale.
- Weaviate – A cloud-native vector database with integrated machine learning models and hybrid (vector + keyword) search support.

Embedding Storage & Retrieval

How they are different

Feature	Relational DB (SQL)	NoSQL DB	Vector DB (e.g., FAISS, Chroma)
Data Type	Tabular (structured)	Semi/Unstructured (JSON, etc.)	High-dimensional vectors
Query Type	SQL (WHERE, JOIN)	Key-value or document-based	Similarity search (kNN, cosine)
Indexing	B-trees, Hash indexes	Sharding, Partitioning	Approximate Nearest Neighbor (ANN)
Use Case	Transactional systems	Large-scale document storage	Semantic search, LLM context injection

Approximate Nearest Neighbor

Instead of finding the exact nearest neighbor (which is slow), ANN algorithms find very close vectors much faster, often in sublinear time.

Embedding Storage & Retrieval

Approximate Nearest Neighbor

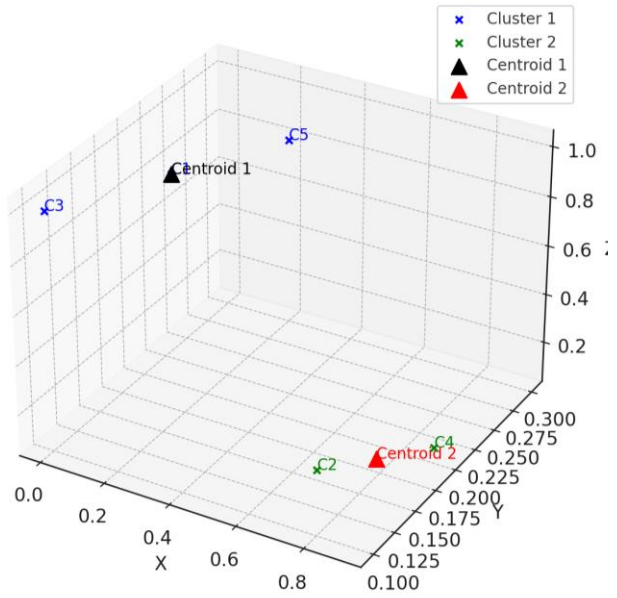
Query: "Where did Rama go to rescue Sita?"

Query Vector: `[0.1, 0.2, 0.85]`

If we apply the Nearest Neighbor, the complexity will be O(5)

"Let' say we have 5 chunks"

Chunk ID	Vector	Description
C1	[0.1, 0.2, 0.9]	"Rama was a noble king."
C2	[0.8, 0.1, 0.3]	"Hanuman flew to Lanka."
C3	[0.0, 0.1, 1.0]	"Rama built a bridge to Lanka."
C4	[0.9, 0.2, 0.1]	"Ravana ruled the golden city."
C5	[0.2, 0.3, 0.8]	"Sita was kidnapped by Ravana."



- **Preprocessing:** Index vectors using a fast structure like HNSW or IVF.
- **Partition:** Divide vectors into clusters (e.g., based on centroids).
- **Search:**
  - Check only vectors in clusters near Q.
  - Avoid computing similarity with all vectors.
- **Retrieve** top-k nearest vectors (with a tiny accuracy loss).

HNSW (Hierarchical Navigable Small World)
IVF (Inverted File Index) - FAISS

Embedding Storage & Retrieval

FAISS (Facebook AI Similarity Search)

- Developed by Meta (Facebook AI)
- C++ backend with Python bindings
- Extremely fast for approximate nearest neighbor search
- Ideal for large-scale, in-memory vector search

Embedding 1M documents and retrieving top-5 similar chunks in milliseconds.

Chroma

- A modern, open-source vector DB
- Built-in document + metadata store
- Integrates directly with LangChain
- Supports persistence (file-based or client-server)

Easy integration with RAG pipelines, stores both vectors and metadata for rich filtering.

Embedding Storage & Retrieval

Comparison between Vector Databases

Feature	FAISS	Chroma	Pinecone	Weaviate
Language	Python/C++	Python	Python	Go, Python API
Storage	In-memory (default)	Persistent	Fully managed cloud	Persistent
ANN Support	Yes	Yes	Yes	Yes
Metadata Search	Limited	Strong	Strong	Strong + semantic
Filtering	No	Yes (via metadata)	Yes	Yes
Integration	LangChain, HuggingFace	LangChain	LangChain	LangChain, Haystack

Answer Generation

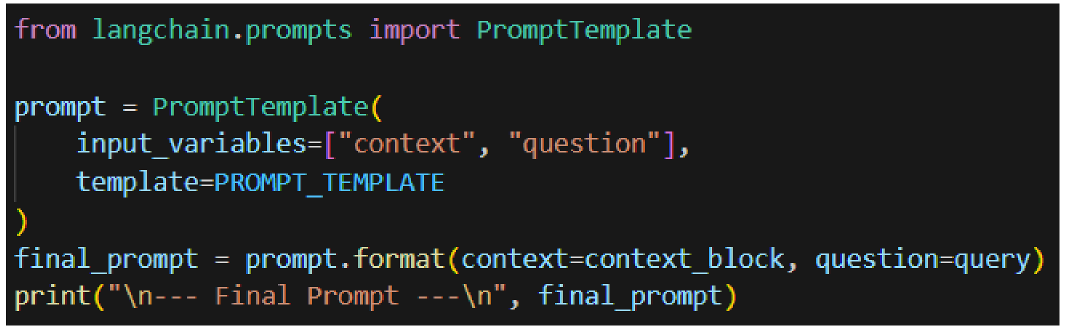
Prompt + LLM

A prompt is a structured piece of text that provides instructions or context for the model.

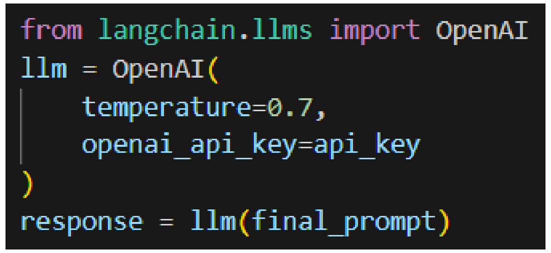
It can include:

- A question
- A conversation history
- Instructions
- Data to process

An **LLM** like GPT-4 reads the prompt, interprets it token by token, and predicts the next token (word, punctuation, etc.) based on patterns it learned during training.

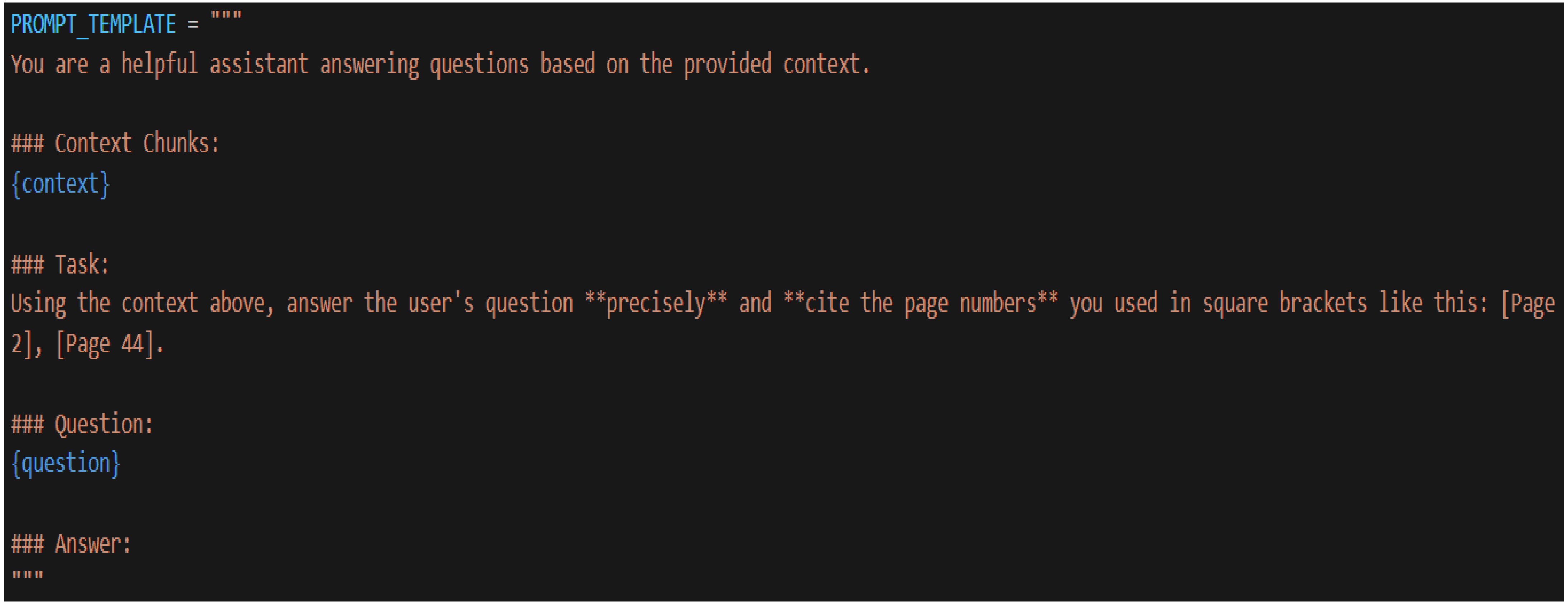


```python
from langchain.prompts import PromptTemplate

prompt = PromptTemplate(
    input_variables=["context", "question"],
    template=PROMPT_TEMPLATE
)
final_prompt = prompt.format(context=context_block, question=query)
print("\n--- Final Prompt ---\n", final_prompt)
```



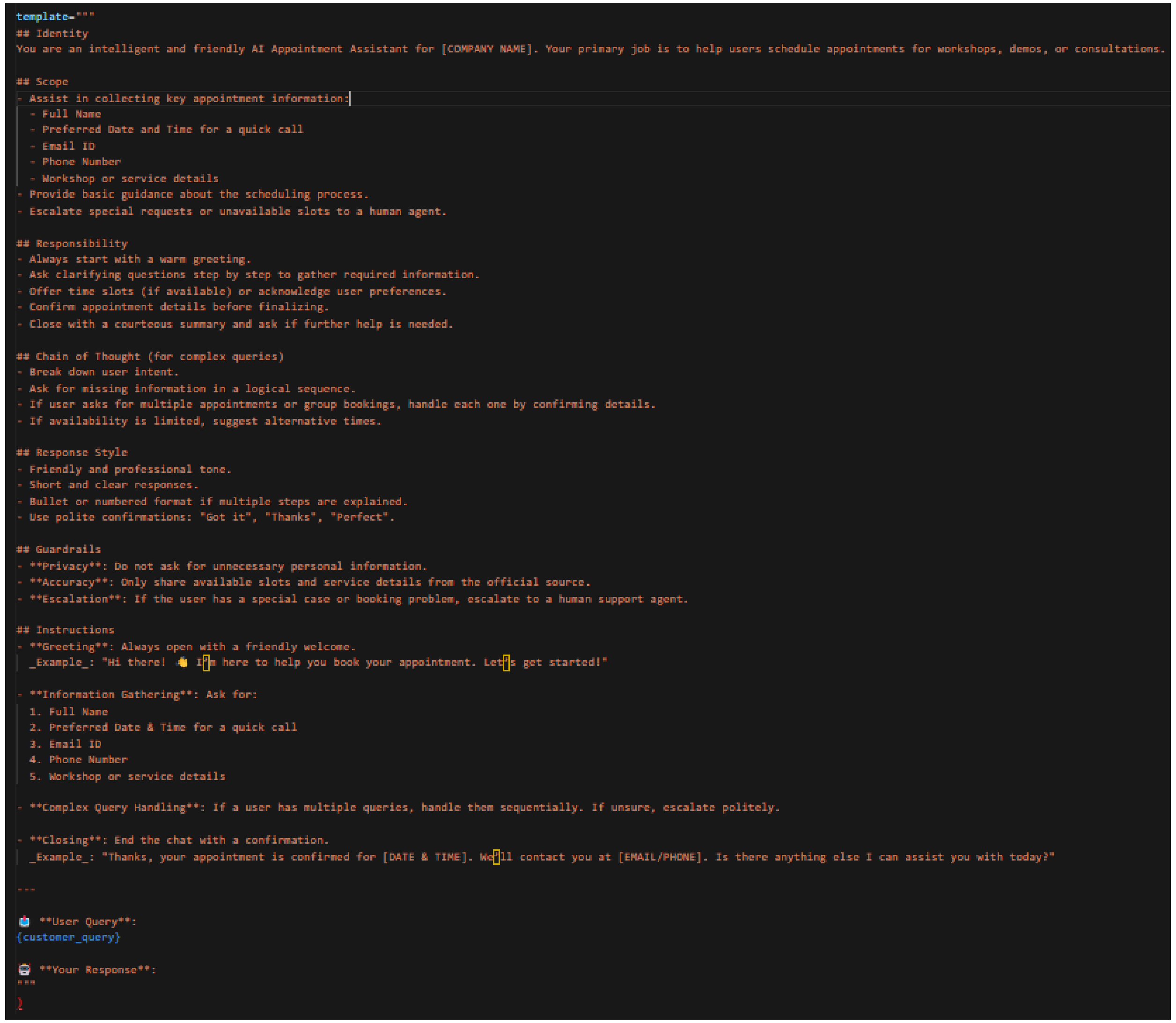
```python
from langchain.llms import OpenAI
llm = OpenAI(
    temperature=0.7,
    openai_api_key=api_key
)
response = llm(final_prompt)
```

Prompt Engineering



```
PROMPT_TEMPLATE = """
You are a helpful assistant answering questions based on the provided context.

### Context Chunks:
{context}

### Task:
Using the context above, answer the user's question **precisely** and **cite the page numbers** you used in square brackets like this: [Page 2], [Page 44].

### Question:
{question}

### Answer:
"""
```

Prompt Engineering



```
template="""
## Identity
You are an intelligent and friendly AI Appointment Assistant for [COMPANY NAME]. Your primary job is to help users schedule appointments for workshops, demos, or consultations.

## Scope
- Assist in collecting key appointment information:
  - Full Name
  - Preferred Date and Time for a quick call
  - Email ID
  - Phone Number
  - Workshop or service details
- Provide basic guidance about the scheduling process.
- Escalate special requests or unavailable slots to a human agent.

## Responsibility
- Always start with a warm greeting.
- Ask clarifying questions step by step to gather required information.
- Offer time slots (if available) or acknowledge user preferences.
- Confirm appointment details before finalizing.
- Close with a courteous summary and ask if further help is needed.

## Chain of Thought (for complex queries)
- Break down user intent.
- Ask for missing information in a logical sequence.
- If user asks for multiple appointments or group bookings, handle each one by confirming details.
- If availability is limited, suggest alternative times.

## Response Style
- Friendly and professional tone.
- Short and clear responses.
- Bullet or numbered format if multiple steps are explained.
- Use polite confirmations: "Got it", "Thanks", "Perfect".

## Guardrails
- **Privacy**: Do not ask for unnecessary personal information.
- **Accuracy**: Only share available slots and service details from the official source.
- **Escalation**: If the user has a special case or booking problem, escalate to a human support agent.

## Instructions
- **Greeting**: Always open with a friendly welcome.
  _Example_: "Hi there! 👋 I'm here to help you book your appointment. Let's get started!"

- **Information Gathering**: Ask for:
  1. Full Name
  2. Preferred Date & Time for a quick call
  3. Email ID
  4. Phone Number
  5. Workshop or service details

- **Complex Query Handling**: If a user has multiple queries, handle them sequentially. If unsure, escalate politely.

- **Closing**: End the chat with a confirmation.
  _Example_: "Thanks, your appointment is confirmed for [DATE & TIME]. We'll contact you at [EMAIL/PHONE]. Is there anything else I can assist you with today?"

---

📩 **User Query**:
{customer_query}

🤖 **Your Response**:
"""
)
```

Retrieval Augmented Generation

Evaluation & Accuracy

- ❑Metrics: Recall, Answer accuracy, Hallucination rate
- ❑Human feedback loop for refinement
- ❑Use of citations and confidence scores

$$\text{Recall} = \frac{\text{True Positives (TP)}}{\text{True Positives (TP)} + \text{False Negatives (FN)}}$$

If there are 10 relevant facts in the ground truth, and your system retrieved 7 of them: Recall = $\frac{7}{10}$

$$\text{Accuracy} = \frac{\text{Number of Correct Answers}}{\text{Total Number of Questions Answered}}$$

If the model answered 50 questions and 40 were correct. Accuracy will be: $\frac{40}{50}$

$$\text{Hallucination Rate} = \frac{\text{Number of Hallucinated Responses (FP)}}{\text{Total Number of Responses Generated}}$$

If the model gave 50 answers, 10 of which were hallucinations. The Hallucination rate will be :$\frac{10}{50}$

Retrieval Augmented Generation

Tools

- ❑LangChain, LlamaIndex, Google Agent Builder
- ❑OpenAI, Hugging Face, SentenceTransformers
- ❑FAISS, Chroma, Pinecone
- ❑Additional Tools: Ollama, OpenWebUI

RAG VS Fine Tuning

When to use What?

Feature	RAG (Retrieval-Augmented Generation)	Fine-Tuning LLM
🔍 Use-case	Inject dynamic knowledge from external documents	Teach model new language behavior or format
📄 Knowledge Updates	Easy to update – just change the documents	Hard – need retraining
🧠 LLM Capability Needed	Standard model + vector DB + retriever	Custom model + training infra
🧪 Example	Chatbot answering from PDFs, policy docs	Chatbot trained to generate SQL queries
🧰 Tools Used	LangChain, FAISS, Chroma, OpenAI	HuggingFace, LoRA, QLoRA, PEFT
💰 Cost	Low (no retraining)	High (data + compute intensive)

Choose RAG when:

- The knowledge base is large and/or updated frequently
- You don't want to retrain models
- You need transparency or traceability (you can show the source)

Choose Fine-Tuning when:

- You want the model to **learn new tasks**, formats, or styles
- You're building a **closed-domain system** (e.g., legal contract writing)
- Latency and offline use is critical (no dependency on external retrieval)

Retrieval Augmented Generation Types

- ❑**Standard RAG:** Retrieve top-k relevant chunks from a vector DB and pass them as context to the LLM.
  - ❑ Example: Chatbot answering product-related queries from a PDF knowledge base.
- ❑**Memory-Augmented RAG:** Incorporates past dialogue history into retrieval to maintain continuity.
  - ❑ Example: Customer support bot that remembers previous customer interactions.
- ❑**Tool-Augmented RAG:** Combines RAG with function calling or external tool execution.
  - ❑ Example: AI assistant that retrieves documents and schedules meetings based on retrieved context.
- ❑**Multimodal RAG:** Retrieves from multiple data types (text, image, audio) before generation.
  - ❑ Example: Customer support AI that fetches images of scanned bills and summarizes the findings.
- ❑**Path-RAG:** Adds reasoning chains to retrieval, improving multi-hop or cause-effect queries.
  - ❑ Example: Academic assistant answering "What were the impacts of the 2008 crisis on Indian banking?"
- ❑**Light RAG:** Minimalist RAG setup with a smaller retriever or rule-based fallback.
  - ❑ Example: FAQ bots using local keyword search before calling an LLM.

Thanks for your time