RAG for Agentic AI

⭐ What is RAG (Retrieval-Augmented Generation)?

RAG — *Retrieval-Augmented Generation* — is an AI technique where a Large Language Model (LLM) combines **external knowledge** with its own reasoning to produce accurate, reliable answers.

LLM + Your Documents = RAG

RAG fixes the biggest limitations of LLMs:

- They **don't know** your PDF, website, or database.
- They **forget** facts after 2023 cutoff (in many models).
- They **hallucinate** when unsure.

RAG solves this by providing **real, verified information** from your documents at the time of answering.

⭐ Why RAG is Important?

RAG makes AI:

✓ More accurate
✓ Less hallucinating
✓ More trustworthy
✓ Capable of using your custom data
✓ Scalable for enterprise knowledge systems

🧩 PART 1 — DOCUMENTS (Knowledge Source)

Short Theory (Simple Enough for Notes)

Documents = any knowledge your RAG system uses to answer questions.

Examples:

- PDFs
- Markdown, .txt
- Websites
- Databases
- CSV, JSON

- API responses

LLMs **cannot** store your private/company knowledge → so RAG uses **your documents** as the source of truth.

Before RAG can use a document, it must be:

1️⃣ Loaded
2️⃣ Cleaned
3️⃣ Structured
4️⃣ Chunked
5️⃣ Embedded
6️⃣ Stored in vector DB

🧪 Code: Loading Documents (PDF, Text, Website)

🍀 1. Load PDF (most common)

```
from langchain.document_loaders import PyPDFLoader


loader = PyPDFLoader("policy.pdf")

documents = loader.load()


print(documents[0].page_content[:300])
```

📝 2. Load Text / Markdown

```
from langchain.document_loaders import TextLoader


loader = TextLoader("notes.txt")

documents = loader.load()
```

🌐 3. Load Website Page

```
from langchain.document_loaders import WebBaseLoader


loader = WebBaseLoader("https://example.com/docs")

documents = loader.load()
```

🗄️ 4. Load Data from CSV / DB Rows

```
import pandas as pd
from langchain.schema import Document

df = pd.read_csv("data.csv")

documents = [
    Document(page_content=row["text"], metadata={"id": i})
    for i, row in df.iterrows()
]
```

🔑 5. Clean the Document (remove junk)

Basic cleaning improves RAG accuracy.

```
def clean(text):
    text = text.replace("\n", " ")
    text = " ".join(text.split())
    return text

cleaned_docs = []
for d in documents:
    d.page_content = clean(d.page_content)
    cleaned_docs.append(d)
```

🏷️ Metadata (very important for Agentic RAG)

Metadata helps agents' reason better.

```
for i, doc in enumerate(cleaned_docs):
    doc.metadata["chunk_id"] = i
    doc.metadata["source"] = "policy.pdf"
```

🧩 PART 2 — CHUNKING

🔥 What is Chunking? (Short Theory)

Chunking = breaking large documents into **small readable pieces** (chunks) so that:

- LLM can understand the text
- embeddings become more meaningful
- retrieval becomes accurate
- agent can reason step-by-step

Ideal Chunk Size:

- **350–800 tokens** (most used)
- With **10–20% overlap**

Why overlap?
To prevent loss of meaning between paragraphs.

🎯 Why Chunking Is Critical for Agentic AI?

Agents need context that is:
✓ Accurate
✓ Relevant
✓ Reusable
✓ Consistent

Good chunking = Good retrieval → Better decisions by agents.

🧪 CODE: Simple Fixed-Size Chunking

```
from langchain.text_splitter import CharacterTextSplitter


splitter = CharacterTextSplitter(
    chunk_size=500,
    chunk_overlap=100,
    separator="\n"
)


chunks = splitter.split_documents(cleaned_docs)
print("Total Chunks:", len(chunks))
```

This is the **best & most stable** chunking method.

```
from langchain.text_splitter import RecursiveCharacterTextSplitter


splitter = RecursiveCharacterTextSplitter(
    chunk_size=700,
    chunk_overlap=100,
    separators=["\n\n", "\n", " ", ""]
)
chunks = splitter.split_documents(cleaned_docs)
```

This helps agents understand where the chunk came from.

```
for i, chunk in enumerate(chunks):
    chunk.metadata["chunk_id"] = i
    chunk.metadata["length"] = len(chunk.page_content)
```

🔎 Chunk Quality Improvement Tricks

✔ 1. Use Sentence-Aware Chunking (optional)

Good when documents have long paragraphs.

```
from langchain.text_splitter import NLTKTextSplitter


splitter = NLTKTextSplitter(chunk_size=600)
chunks = splitter.split_documents(cleaned_docs)
```

✔ 2. Clean Junk Words Before Chunking

Otherwise junk spreads into every chunk's embedding.

✔ 3. Remove Small Chunks (<200 chars)

They reduce retrieval accuracy.

```
chunks = [c for c in chunks if len(c.page_content) > 200]
```

📝 Example

```
Chunk ID: 42

Source: policy.pdf

Text: "Employees are allowed 24 paid leaves per year..."

Length: 532
```

⭐ PART 3: Retrieval Engine

This part explains **how your agent finds the right chunk** when you ask a question.

RAG Retrieval =
User Query → Embedding → Vector DB Search → Relevant Chunks

🌟 1. What is Retrieval?

Retrieval = "The process of pulling correct knowledge from your DB".

It has 2 parts:

1. **Vector Search (ANN - Approx Nearest Neighbor)**
2. **Re-ranking** (optional but improves accuracy)

🌟 2. Retrieval Flow

```
User Question

    ↓

Convert to Embedding (numbers)

    ↓

Vector DB searches nearest vectors

    ↓

Returns Top-k Chunks

    ↓

LLM uses them to answer
```

🌟 3. Retrieval Parameters You Must Know

Parameter	Meaning	Recommended
k	number of chunks to retrieve	3–5
similarity metric	cosine / euclidean / dot	cosine
score_threshold	ignore low-quality matches	0.2–0.3
reranking	reorder best chunks using LLM	optional but powerful

🌟 4. Types of Retrieval

1️⃣ Dense Vector Search (FAISS / Chroma)

- Best for most RAG systems
- Uses embeddings
- Simple & fast

2️⃣ Sparse Retrieval (BM25)

- Word-based search
- Great for code + exact matching
- Often used in hybrid RAG

3️⃣ Hybrid Retrieval

Embedding + BM25 combined
🔥 *Best accuracy, used in advanced Agentic AI*

⭐ PART 3 — CODE SECTION

✔ **Code #1 — Vector Search with FAISS**

```
from langchain_community.vectorstores import FAISS
from langchain_openai import OpenAIEmbeddings

emb = OpenAIEmbeddings(model="text-embedding-3-small")
# Create vector store
vectorstore = FAISS.from_documents(chunks, embedding=emb)

# Save locally
vectorstore.save_local("faiss_rag_db")
```

✔ **Code #2 — Search Relevant Chunks**

```
query = "What is RAG and how does retrieval work?"

results = vectorstore.similarity_search(
    query=query,
    k=3
)
for r in results:
    print("\n---Chunk---\n", r.page_content)
```

✔ Code #3 — Search With Score Threshold

```
docs = vectorstore.similarity_search_with_score(
    query="Explain agentic AI",
    k=5
)
filtered = [d for d, score in docs if score < 0.25]

print("Returned:", len(filtered))
```

⭐ 5. Add Re-ranking (Boost Accuracy)

Your LLM can reorder results based on relevance.

✔ Code #4 — Simple Reranking with LLM

```
from langchain_openai import ChatOpenAI
llm = ChatOpenAI(model="gpt-4o-mini")

def rerank(query, retrieved_chunks):
    prompt = f"""
    Query: {query}
    Rank the following chunks by relevance:
    {retrieved_chunks}
    Return top 3.
    """
    return llm.invoke(prompt)

reranked = rerank(query, [c.page_content for c in results])
print(reranked)
```

⭐ 6. Hybrid Retrieval (Best for Agentic AI)

Dense + Sparse combined → more accurate.

✔ Code #5 — Hybrid Search

```
from langchain_community.retrievers import BM25Retriever

from langchain.retrievers import EnsembleRetriever

dense_retriever = vectorstore.as_retriever(search_kwargs={"k": 3})

sparse_retriever = BM25Retriever.from_documents(documents)

hybrid = EnsembleRetriever(

   retrievers=[dense_retriever, sparse_retriever],

   weights=[0.6, 0.4]

)

out = hybrid.invoke("how does embedding work?")

📌 Retrieval Best Practices


• Use FAISS for local RAG experiments

• Use Chroma for production

• Always retrieve top-k=3 to avoid hallucination
```

🔥 RAG Skill Tree – PART 4
"LLM Integration + Answer Generation Pipeline"

⭐ PART 4: LLM + Generation Engine

This is where RAG becomes *useful*.
You already learned:

- Load Docs
- Chunk Docs
- Create Embeddings

- Build Vector DB
- Retrieve Relevant Chunks

Now → we plug it into the **LLM** to generate accurate answers.

RAG =
Retrieve (External Knowledge) + Generate (LLM reasoning)

🌟 1. How Answer Generation Works (Simple Visual)

```
Query
    ↓
 Retrieve Chunks
    ↓
 Build RAG Prompt
    ↓
     LLM
    ↓
   Final Answer
```

🌟 2. What is a RAG Prompt?

It is a special prompt where we give the LLM:

✔ user query
✔ retrieved chunks
✔ instructions
✔ restrictions

Example:

```
Use ONLY the provided context to answer.

If answer not found, say "Not in document".


Context:

[chunk1]

[chunk2]

[chunk3]


User question:

Explain RAG retrieval.
```

This eliminates hallucination.
This turns the LLM into a **grounded agent**.

🌟 3. Code: Build a Simple RAG Pipeline

This is the **standard RAG code** used everywhere.

✔ Code #1 — Setup LLM

```
from langchain_openai import ChatOpenAI

llm = ChatOpenAI(model="gpt-4o-mini")
```

✔ Code #2 — Build RAG Chain

```
def build_rag_prompt(query, retrieved_docs):
    context = "\n\n".join([d.page_content for d in retrieved_docs])
    prompt = f"""
    Use ONLY the context below to answer the question.

    Context:
    {context}

    Question:
    {query}

    If not found in context, reply: 'Information not available.'
    """
    return prompt
```

✔ Code #3 — RAG Execution

```
query = "What is agentic AI?"

# Step 1: Retrieve
docs = vectorstore.similarity_search(query, k=3)

# Step 2: Build prompt
rag_prompt = build_rag_prompt(query, docs)
```

This is a **complete RAG answer generator**.

🌟 4. Optional but Powerful: RAG with Chain

LangChain gives a ready-built pipeline.

✔ Code #4 — LangChain RAG Chain

```
from langchain.chains import RetrievalQA


rag_chain = RetrievalQA.from_chain_type(
   llm=llm,
   retriever=vectorstore.as_retriever(search_kwargs={"k": 3}),
   return_source_documents=True
)
out = rag_chain.invoke({"query": "Explain embeddings"})
print(out["result"])
```

Th

- builds prompt
- merges text
- produces answer

🌟 5. Improve Output with Instructions (Most Important)

Your LLM produces better results with strong guiding instructions.

RAG System Instructions Template

```
You are a highly accurate retrieval-focused AI.
Rules:
- Use ONLY the provided context.
- Do NOT add external knowledge.
- Cite the chunk if necessary.
- If answer not found, say "Not found in the document."
- Keep the answer clear and concise.
```

🌟 6. Add Safety: Prevent Hallucination

Add this rule:

```
If context does not contain relevant info:

Return: "Not available in the provided document."
```

This makes your RAG **robust** for Agentic AI.

🌟 7. Outputs You Can Generate

RAG can produce:

- Summary
- Q&A
- Explanations
- Step-by-step instructions
- Convert PDF → Structured data
- Reasoning based on retrieved chunks

🌟 8. Full RAG Pipeline

⭐ Full RAG Answer Generation Flow

1️⃣ User asks a question

2️⃣ Convert query → embedding

3️⃣ Vector DB performs semantic search

4️⃣ Top-k chunks returned

5️⃣ Build final RAG prompt

6️⃣ LLM generates grounded answer

7️⃣ Optional: citations + re-ranking

🌟 9. Full Working RAG Script (Simple)

```
from langchain_community.vectorstores import FAISS

from langchain_openai import OpenAIEmbeddings, ChatOpenAI

from langchain.text_splitter import RecursiveCharacterTextSplitter

from langchain_community.document_loaders import PyPDFLoader


# 1 Load

loader = PyPDFLoader("myfile.pdf")

docs = loader.load()


# 2 Chunk

splitter = RecursiveCharacterTextSplitter(chunk_size=400, chunk_overlap=50)

chunks = splitter.split_documents(docs)


# 3 Embed

emb = OpenAIEmbeddings(model="text-embedding-3-small")


# 4 Vector DB

vectorstore = FAISS.from_documents(chunks, emb)


# 5 LLM

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


# 6 RAG Pipeline

def rag(query):

    docs = vectorstore.similarity_search(query, k=3)

    context = "\n\n".join([d.page_content for d in docs])
```

```
prompt = f"""
    Answer using ONLY this context.
    If not found, say 'Not available.'

Context:
    {context}


    Question: {query}
    """
    return llm.invoke(prompt).content


print(rag("Explain RAG retrieval"))
```

🎯 Part 4 — What You Learned

LLM + Retrieval = RAG Engine

✓ How generation works

✓ Prompt engineering for RAG

✓ Build full RAG chain

✓ Safety rules

✓ Complete working code

✓ Perfect for Agentic AI answering systems
