Retrieval-Augmented Generation (RAG)

AI Summer 2025, Vanderbilt Data Science Institute

Umang Chaudhry, Senior Data Scientist

May 19, 2025

Agenda

1. **Introduction to RAG**
   What, why, when
2. **Architecture Deep Dive**
   Key components, pipeline overview
3. **Colab + LangChain Setup**
   Get ready for Wednesday's hands-on coding

Retrieval-Augmented Generation (RAG)

- **Combines:** *retrieval* of relevant knowledge $\rightarrow$ *generation* by an LLM
- Context is injected *at query time* – no model weights are changed
- Works with any LLM (OpenAI, Anthropic, Google, etc.)

Retrieval-Augmented Generation (RAG)

- **Combines:** *retrieval* of relevant knowledge $\rightarrow$ *generation* by an LLM
- Context is injected *at query time* – no model weights are changed
- Works with any LLM (OpenAI, Anthropic, Google, etc.)

Key idea: keep your private or fast-changing data <u>outside</u> the model and still get personalised answers.

Why choose RAG?

- **Lower upfront cost** – no training runs or need for HPCs/large compute.
- **Continuously evolving data** – new/changed documents are re-embedded in seconds; retraining a model could take hours $\rightarrow$ days.
- **Smaller operational risk** – retrieval pipeline tweaks are reversible; model weights stay frozen, so no catastrophic forgetting.
- **Data governance & IP safety**
  When both embeddings *and* the LLM run locally, private data is never exposed.

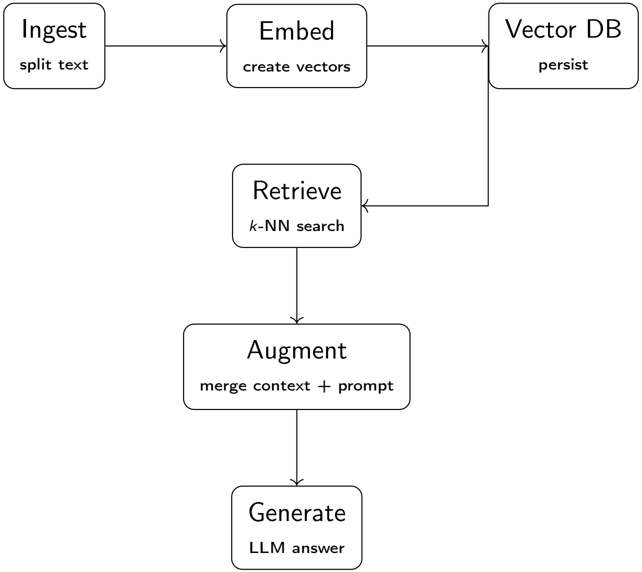
When is fine-tuning the better option?

- **Highly domain-specific language**
  e.g. medical notes whose acronyms vary hospital-to-hospital—too specialised for a generic embedding + prompt to catch every time.
- **Output style must be exact**
  Legal wording, poetry, code patterns that prompt-engineering can't consistently enforce.
- **Strict latency budgets**
  If every millisecond counts, vector search/retrieval may exceed the target; a fine-tuned model can answer in a single forward pass.
- **Context-length limits**
  Very-long-context models help, but when terms appear *in nearly every prompt*, embedding every piece of text is wasteful—bake it into the weights instead.

Where RAG Shines – Use-Case Snapshots

- **Enterprise search assistants** – ask internal policies or research docs
- **Customer support chatbots** – manuals, FAQs, ticket history
- **Academic literature copilots** – retrieve papers, summarise findings
- **Code-base Q&A** – search repo, explain functions

High-Level Pipeline



Document Ingestion & Loaders

- **Goal:** turn raw data (CSV, PDFs, Slack threads, G-Drive files, . . .) into LangChain `Document` objects.
- $>$**100 integrations** – Slack, Notion, Google Drive, S3, webpages, databases, etc.
- `load()` – returns a *list* of documents.
- Once loaded, documents flow to the text-splitting stage.

LangChain PyPDFLoader – Quick Example

```
from langchain_community.document_loaders import PyPDFLoader

loader = PyPDFLoader(file_path)
pages = []
for page in loader.load():
    pages.append(page)
```

Same pattern applies to most document loaders.

Text Splitting: Why & Overview

Why split documents?

- Handle documents of very different lengths.
- Stay within LLM/tokenizer input limits.
- Produce higher-quality, focused embeddings.

Four splitter families (LangChain):

- *Length-based*
- *Text-structured (Recursive)*
- *Document-structured*
- *Semantic*

Text Splitter Strategies

- **Length-based** – fixed token/character windows; fast, predictable.
- **Recursive** – walk the text hierarchy (paragraph $\rightarrow$ sentence $\rightarrow$ word) to keep natural flow.
- **Document-structured** – split on markup (Markdown headers, HTML tags, JSON keys, code blocks).
- **Semantic** – detect topic shifts with sliding-window embeddings; most coherent, highest compute cost.

Recursive Splitter in LangChain

```
from langchain_text_splitters import RecursiveCharacterTextSplitter

splitter = RecursiveCharacterTextSplitter(
    chunk_size = 100,   # tokens/characters per chunk
    chunk_overlap = 0   # overlap between chunks
)
chunks = splitter.split_text(document)
```

Embedding Models: Concept & Role

- Map text $\rightarrow$ fixed-length **vector** ("semantic fingerprint").
- Enable *semantic* search, clustering, recommendation.

Two steps in RAG

1. *Embed* query + chunks.
2. *Compare* vectors to find relevant chunks.

LangChain Embedding Interface

- Uniform API across providers: `embed_documents` & `embed_query`.

```
from langchain_openai import OpenAIEmbeddings
embeddings_model = OpenAIEmbeddings()
embeddings = embeddings_model.embed_documents(
    [
        "Hi there!",
        "Oh, hello!",
        "What's your name?",
        "My friends call me World",
        "Hello World!"
    ]
)
len(embeddings), len(embeddings[0])
# (5, 1536)
```

Measuring Similarity

- **Cosine** — most common for text.
- **Dot-product** — some dense models.
- **Euclidean** — rare for NLP.
- **Note:** Use the similarity metric that your embeddings provider recommends (eg. OpenAI recommends cosine similarity)

```
import numpy as np

def cosine_similarity(vec1, vec2):
    dot_product = np.dot(vec1, vec2)
    norm_vec1  = np.linalg.norm(vec1)
    norm_vec2  = np.linalg.norm(vec2)
    return dot_product / (norm_vec1 * norm_vec2)

similarity = cosine_similarity(query_result, document_result)
print("Cosine Similarity:", similarity)
```

Choosing an Embedding Model

- **Domain fit** – general, code, biomedical, legal. . .
- **Vector size vs latency** – bigger isn't always better.
- **Cost / hosting** – API vs. local inference.

Word2Vec vs Modern Embeddings (RAG Context)

Legacy: Word2Vec-style

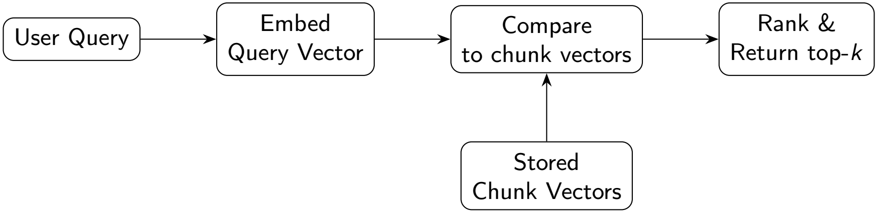
- One vector *per word or token*.
- Limited context — only nearby words influence meaning.
- Requires aggregating many word vectors to compare passages.
- Not ideal for RAG: poor performance on full-document relevance.

Modern: Sentence/Chunk Embeddings

- One vector *per full sentence or chunk of text*.
- Uses Transformer attention $\rightarrow$ understands broader context.
- Encodes semantics of entire paragraph, not just word-level.
- Ideal for RAG: clean document-level retrieval.

Chunk size is tunable (by tokens, sentences, or structure).

Similarity-Search Workflow



Vectors compared with cosine / dot-product etc. Top-k chunks are injected into the LLM prompt.

How Do Embeddings Know What's Similar?

- **Each chunk becomes a vector**: like assigning a point in space based on its meaning.
- **Similar meanings = closer points**: during model training, examples with related meaning are pulled closer together in that space.
- **At runtime**: the model doesn't "compare" documents — it just gives each one a coordinate. You compare those points using math (like cosine similarity).

Why chunk size matters

- Too small → not enough meaning to position well (low-quality vectors).
- Too large → includes irrelevant info, making comparisons noisy.

Vector Stores: Concept & Role

- **What** — specialised data stores that index *embeddings*: vectors capturing semantic meaning.
- **Why** — enable fast, similarity-based search over unstructured data (text, images, audio).

Typical RAG flow

1. Store embedded chunks in a vector DB.
2. Embed user query.
3. Retrieve top-k similar chunks for the LLM.

LangChain Vector Store Interface

Key methods (uniform across integrations)

- `add_documents(docs)`
- `delete(ids)`
- `similarity_search(query, k=4, filter=None)`

```
from langchain_core.vectorstores import InMemoryVectorStore
from langchain_openai import OpenAIEmbeddings
from langchain_core.documents import Document

emb = OpenAIEmbeddings()
vs  = InMemoryVectorStore(embedding=emb)

docs = [
    Document(page_content="Chocolate-chip pancakes", metadata={"src": "tweet"}),
    Document(page_content="Forecast: cloudy, 62°F",    metadata={"src": "news"}),
]
vs.add_documents(docs, ids=["doc1", "doc2"])
```

Similarity Search & Metrics

- **Metric choices**: cosine, Euclidean, dot-product—pick the one recommended by your embedding provider.

Basic call:

```
docs = vs.similarity_search("my query", k=4)
```

Returns a list of `Document` objects, ranked by similarity.

Metadata Filtering

Vector stores can combine *semantic* search with *structured* filters.

```
docs = vs.similarity_search(
    "LangChain abstractions",
    k=2,
    filter={"src": "tweet"}   # only tweets
)
```

When useful

- Restrict by source, author, date range, privacy tag . . .
- Post-processing (e.g. re-ranking) runs on a smaller, relevant set.

Beyond Basic Search

Hybrid search	Combine keyword + vector similarity for cases where exact terms matter (e.g. code).
MMR re-ranking	Maximal Marginal Relevance diversifies results to avoid near-duplicates.
Scalability	Most cloud stores (Pinecone, Qdrant, etc.) expose the same API; swap by changing one line.

LangChain docs list all supported vector-store integrations and their extra parameters.

Retrievers: Concept & Role

- Provide a **uniform interface** over many back-ends (vector DBs, search APIs, SQL/graph DBs, lexical engines).
- **Input**: natural-language query (string)
- **Output**: list of `Document` objects.

LangChain Retriever Interface

- Implement `_get_relevant_documents(query:str)`.
- Exposed via `invoke()` (all LangChain runnables share this).

```
docs = retriever.invoke("What causes tides?")
# docs -> [Document(page_content=..., metadata=...), ...]
```

Each `Document` has:

- `page_content` – the text
- `metadata` – id, source, etc.

Example: Vector Store Retriever

```
vector_store = MyVectorStore()
retriever = vector_store.as_retriever()

docs = retriever.invoke("Photosynthesis steps")
```

- Under the hood: embed query $\rightarrow$ similarity search $\rightarrow$ return top-k docs.
- Supports filters (`k`, metadata) exactly like `similarity_search`.

Augmentation & Prompt Engineering (RAG)

Goal: Inject the *right* external knowledge into the LLM prompt.

- Concatenate retrieved chunks $\rightarrow$ `{context}`.
- Keep total tokens under the model's limit.
- Give the model *clear rules*: tone, length, citations, fallback.

Robust RAG prompt template:

```
SYSTEM:
You are an assistant for question-answering tasks.
Use ONLY the context below. If unsure, say "I don't know".
Answer concisely (3 or fewer sentences) and cite sources.

{context}

USER: {question}
```

Prompt Quality Matters

Weak prompt

```
Answer the question:

{question}

Docs:
{context}
```

- No guidance on using context.
- No instructions for uncertainty.
- High hallucination risk.

Improved prompt

```
SYSTEM: Use the context to answer.
If the answer isn't found, say so.
Limit to 3 sentences. Cite sources.

{context}

USER: {question}
```

- Clear usage rule $\rightarrow$ fewer hallucinations.
- Concise length bound.
- Built-in fallback response.

RAG Pipeline: End-to-End Example (1/2)

```
from langchain_openai import ChatOpenAI
from langchain_core.messages import SystemMessage, HumanMessage

# 1) System prompt template
system_prompt = """You are an assistant for question-answering tasks.
Use the following pieces of retrieved context to answer the question.
If you don’t know the answer, just say that you don’t know.
Use three sentences maximum and keep the answer concise.
Context: {context}:"""

# 2) User question
question = ("What are the main components of an LLM-powered "
            "autonomous agent system?")

# 3) Retrieve relevant documents
docs = retriever.invoke(question)
docs_text = "".join(d.page_content for d in docs)

# 4) Fill prompt with context
system_prompt_fmt = system_prompt.format(context=docs_text)
```

RAG Pipeline: End-to-End Example (2/2)

```
# 5) Create chat model
model = ChatOpenAI(model="gpt-4o", temperature=0)

# 6) Generate answer
answer = model.invoke([
    SystemMessage(content=system_prompt_fmt),
    HumanMessage(content=question)
])

print(answer)
```

Common Pitfalls

1. **Chunking too small / too large** → poor context
2. **Poor embedding model selection**
3. **Hallucinations** from irrelevant/poor quality chunks

Environment Setup

- Google Colab or personal computer with Langchain and Python pre-installed
- LangChain $\geq$ 0.3+ - we'll install this in our demo notebooks
- Python 3+ (default in Colab)
- OpenAI API Key (ideal) or Google AI API key

Optional "Homework" for Wednesday

1. Choose a small corpus of non-private documents (PDFs, .txt, .docx, .csv etc.) you care about
2. Push it to Google Drive or have it available locally
3. Think of queries you'd love a bot to answer using those documents