# Building Intelligent AI Workflows in Python with Large Language Models (LLMs) and Retrieval-Augmented Generation (RAG)

**Ryan Paul Lafler, M.Sc.**, Premier Analytics Consulting, LLC

## ABSTRACT

Large language models (LLMs) can generate fluent responses, but they do not automatically understand an organization's operating environment, internal documents, technical standards, or security boundaries. This paper presents a practical foundation for building Retrieval-Augmented Generation (RAG) workflows in Python using LangChain. Guided by explanations and code excerpts, this paper follows the construction of a RAG pipeline from document loading and text preparation through chunking, vector embeddings, FAISS indexing, semantic retrieval, prompt construction, and connection to hosted or localized language models. Common chunking strategies are compared to show how document structure and context preservation affect retrieval quality. This paper also considers the use of small and medium-sized open-source language models for well-scoped and bounded tasks, larger models for broader synthesis, and the role of retrieval, source citations, evaluation, access controls, and human review in reducing unsupported responses. The resulting workflow provides a practical starting point for secure enterprise and research applications in 2026, 2027, and beyond.

***Keywords:*** *Python; Retrieval-Augmented Generation (RAG); Large Language Models (LLMs); LangChain; Tokenizer; Document Chunking; Vector Embeddings; Embedding Models; FAISS; Hugging Face; AI Governance; Open-Source AI Framework*

## INTRODUCTION

Ask a large language model (LLM) a question, and it will usually provide a confident, well-written response. That doesn't mean the response is current, relevant, or supported by the information an organization uses. An LLM isn't automatically connected to internal documents, research, technical standards, or operating procedures, and it does not receive updates after its training cutoff. When important context is missing, the model must rely on incomplete knowledge to produce an answer that may sound reasonable without reflecting the complete record.

Rather than relying only on what a language model learned during training, including the scope and dates of information it was trained on, a Retrieval-Augmented Generation (RAG) system searches a managed collection of current documents for passages that are semantically related to the user's request (Lewis et al., 2020; Gao et al., 2023). This allows for specialized and organization-specific information to be retrieved and used as the model's context when generating its response. With RAG, a general-purpose language model can respond using a narrower body of organization-specific evidence without retraining the underlying model.

Even the best RAG systems cannot eliminate hallucination or guarantee that every response will be correct or comprehensive. RAG workflows can still retrieve irrelevant information, overlook important excerpts from a document, and provide the model with incomplete context. The approach presented in this paper, however, does provide a practical way to use AI for documents, preserve source context, evaluate retrieval results, and identify when available evidence may not be sufficient to answer a request.

This paper presents an open-source RAG workflow developed in Python using LangChain with code excerpts that demonstrate document preparation and chunking, vector embeddings, FAISS indexing, semantic retrieval, prompt construction, and language model connectivity into one complete system. By building this RAG workflow, readers can then adapt it towards their document collections, embedding models, retrieval settings, hosted or localized language models, and specific use cases in their industry.

## 1. Foundations of a Python Retrieval-Augmented Generation (RAG) Workflow

Before loading documents or selecting a language model, it helps to separate a RAG system into two related processes: preparing a searchable document collection and using that collection to answer a request. This section introduces those processes, explains where LangChain fits within the workflow, and prepares the Python environment used throughout the paper. Our goal is to:

- Understand and visualize the RAG architecture and familiarize ourselves with the LangChain framework;
- Create the Python virtual environment for this LangChain project;
- Install necessary packages into the virtual environment;
- Set up and configure pathways to working directories for the documents (artifacts) and the FAISS index.

Once these items are in place, we'll have the basic structure needed to begin moving documents through the RAG workflow. The same directories, settings, and LangChain components introduced here will be used throughout the remaining examples so that each section builds naturally on the previous one.

## 1.1 Generative AI vs. Retrieval-Backed Architectures

Generative models answer from patterns learned during training and from information included in the current prompt. While this may work well enough for general explanations, drafting, or transformation tasks, it becomes less reliable when a question depends on a policy revised last week, a study protocol that exists only inside the research team, or a technical standard that differs by client, facility, or business unit. In these scenarios, models may recognize the topic, hallucinate a vague or incorrect response, and miss the facts needed for a correct answer.

A retrieval-backed architecture adds a search step before generation: the user's question is converted into the same numerical representation used to embed text within a document collection, where the closest numerical-matching passages are retrieved, and then supplied to the language model as working context. The model still generates its answer but is no longer being asked to reconstruct organization-specific knowledge from training alone. RAG effectively narrows the model's attention to information that can be inspected, updated, cited, and evaluated, mitigating off-topic or hallucinated responses.

## 1.2 Core Components of a RAG System

The workflow presented in this paper uses a controlled two-step design with retrieval happening first, followed by generation second. This workflow is intentionally simpler than an agentic workflow where a model decides routing paths, when to search, and how often it needs to query some vector database. For foundational implementation, this approach is simpler to test since the same question, index, retrieval setting, and prompt can be repeated without an agent changing the route between runs.

LangChain provides modular interfaces for document preparation, retrieval, prompt construction, model connectivity, and other AI workflow components. In our example, the document collection is the evidence base that LangChain connects to. Document loader methods then convert each source into LangChain Document objects, a splitter produces smaller passages, an embedding model encodes those passages, and FAISS makes the vectors searchable. At request time, the same embedding model encodes the user's question, the retriever returns the closest-matching chunks using a distance metric, and a prompt combines the question with the retrieved content. The connected language model then generates a response under instructions that define what it should do when the evidence is incomplete. Metadata and evaluation surround the entire path; they aren't optional additions at the end.

## 1.3 Introduction to LangChain's Open-Source Framework

LangChain's open-source community framework provides a consistent set of interfaces for documents, text splitters, pre-trained embedding models, structured prompt output, retrieval and agent routing, and language models. LangChain allows users and teams to connect to hosted LLMs or downloaded Hugging Face models running internally inside their infrastructures. Teams can use a local Hugging Face embedding model, store vector embeddings in FAISS, and connect the prompt to an approved hosted model or a model running inside its own infrastructure.

This framework is best treated as application code, not as a governance and auditing system. LangChain can help pass documents and prompts through a pipeline, but it does not decide which employee is authorized to retrieve a contract, whether a research dataset may leave a controlled environment, or how long a model interaction should be retained. Those decisions belong in the surrounding system architecture.

## 1.4 Preparing the Python Environment and External Directory Pathways

A small virtual environment keeps this example reproducible and prevents unrelated project packages from quietly changing the workflow. A practical baseline installs **`langchain`**, **`langchain-core`**, **`langchain-text-splitters`**, **`langchain-huggingface`**, **`sentence-transformers`**, **`faiss-cpu`**, **`pypdf`**, and **`python-dotenv`**. The connector for the selected language-model provider should be installed separately because LangChain's provider integrations are maintained as individual packages. Python 3.10 or later provides a practical baseline for this workflow.

***Example 1. Preparing shared project pathways and retrieval settings***

```
from pathlib import Path
from dotenv import load_dotenv

# Pathway to .env file containing sensitive LLM connection information
ENV_PATH = Path(".env")
load_dotenv(dotenv_path=ENV_PATH, override=True)

# Pathways to approved documents location and FAISS index location
DATA_DIR = Path("data/approved_documents")
INDEX_DIR = Path("artifacts/faiss_index")
```

```
DATA_DIR.mkdir(parents=True, exist_ok=True)
INDEX_DIR.mkdir(parents=True, exist_ok=True)

# Maximum characters per chunk, shared overlap, and top-four retrieval count
CHUNK_SIZE = 800
CHUNK_OVERLAP = 120
TOP_K = 4
```


***Figure 1.*** *Demonstration project directory containing the two approved policy PDFs, environment configuration, FAISS index, and output folders used throughout the LangChain RAG workflow.*

**Example 1** establishes the global variables reused throughout the paper. The **`.env`** file can hold non-public configuration such as a model provider, model name, endpoint, secrets, or credential reference without hard coding it into the Python file. In a production system, secrets should come from the organization's approved secret manager, but the same separation still applies. Paths and retrieval settings are kept together so that changes are deliberate and reviewable.

## 2. Loading, Preparing, and Describing Source Documents

A RAG system cannot search a PDF, Word file, or research report in the same way that a person opens and reads it. These files must first be loaded into Python, parsed and converted into usable text, and then placed into a consistent, machine-readable format that can move through the remaining parts of the workflow.

This is where document loading and preparation become important. If a source is loaded incorrectly, a table is lost, or the document's page and revision information are removed, those problems will continue into the chunks, vector embeddings, retrieval results, and final response. Before building the FAISS index, we need a document collection that contains readable text and enough source information to trace a retrieved passage back to where it came from.

### 2.1 Document Loaders and Supported Formats

LangChain supports document loaders for PDFs, Word files, HTML pages, plain-text files, cloud storage, databases, and many other sources. These loaders provide a common entry point into the workflow even when the original documents were created in different formats or systems.

A loader's job is to open the source and extract its contents. The extracted text is then stored in a LangChain Document object. Despite its name, a Document object is not another PDF or Word file. It is a Python object that contains two primary parts:

- **`page_content:`** Contains the text extracted from the document
- **`metadata:`** A customizable Python dictionary describing the document

This format gives LangChain's splitters, embedding models, retrievers, and prompt components a consistent object to work with. It also keeps the source information attached to the text as that text moves through the workflow.

For example, a PDF may initially produce one Document object for each page. When those pages are later divided into smaller chunks, LangChain can copy the original filename and page number into each new chunk. Without this connection, the system might retrieve a useful passage but have no dependable way to cite its source. No two PDF collections behave exactly alike. A standard digital PDF may extract cleanly, while a scanned report may require optical character recognition before its text can be used. Multi-column reports, spreadsheets, tables, and technical documents may also require more specialized parsing. A loader should be tested against representative files, including difficult formats and complex layouts, before it is applied over the complete collection.

## 2.2 Text Cleaning and Normalization

Text cleaning should correct extraction problems without removing the structure or language that gives a document meaning. Repeated spaces, inconsistent line endings, page artifacts, OCR errors, and unusual characters can interfere with chunking and semantic retrieval, but headings, paragraph boundaries, lists, units, qualifiers, and negative statements often need to remain intact. For this workflow, the normalization function makes limited corrections while preserving paragraph separation for the recursive chunking methods introduced in the next section. The appropriate amount of cleaning will depend on the source format and should be tested against representative documents before processing the complete collection.

| Document Element | Common Parsing / Cleaning Issues | Recommendation |
|---|---|---|
| **Whitespace and line endings** | Repeated spaces, tabs, or inconsistent line breaks introduced during extraction. | Standardize spacing while preserving meaningful paragraph boundaries. |
| **Headers, footers, and page numbers** | Repeated text may appear in every extracted page and dominate retrieval results. | Remove recurring page artifacts when they are not part of the document's content. |
| **Heading, sections, and markdown formatting** | Formatting may be lost or headings may be combined with body text. | Preserve heading text and section boundaries so chunks retain their subject and position. |
| **Lists and bullet points** | Bullets, numbering, or indentation may disappear during parsing. | Retain item order and separation, particularly for requirements, instructions, and procedural steps. |
| **Tables and multi-column layouts** | Values may be extracted out of sequence or separated from their labels. | Use table-aware or layout-aware parsing when row, column, and label relationships matter. |
| **Special characters** | Symbols, accented characters, mathematical and scientific formulas, or technical notation may be replaced or dropped. | Normalize encoding without removing valid scientific, statistical, or domain-specific notation. |

## 2.3 Document Metadata and Source Identification

In addition to extracted text, each loaded passage needs enough metadata to identify where it originated. At minimum, this usually includes the filename and page number. Depending on the documents and how the system will be used, metadata may also include the document title, revision date, author(s), department, research project, document type, approval status, and access classification. These fields remain attached to the **`Document`** object's JSON-like **`metadata`** payload as it moves into the chunking and embedding stages.

This becomes especially important once a relatively small collection produces hundreds or thousands of chunks in the FAISS index. Metadata provides the path from an individual retrieved passage back to the original document, allowing the system to cite its sources, filter results, identify outdated material, compare revisions, and show a reviewer which evidence was supplied to the language model. Without that information, the retrieved text is separated from the document that gave it meaning.

## 2.4 Defining the Approved Document Collection

A working directory may contain draft policies, expired procedures, duplicate revisions, personal notes, and documents restricted to a particular project or group. If all these files are embedded without review, the system may retrieve conflicting or unauthorized information and pass it directly to the language model. The response may still sound reasonable even though the wrong source was used.

For the code excerpt in this paper, a small set of approved filenames makes this decision visible. Each filename is checked before the document is loaded and converted into LangChain **`Document`** objects.

***Example 2. Loading PDF pages into LangChain documents***

```
import re
from pypdf import PdfReader
from langchain_core.documents import Document
```

```
# Specify approved files list to incorporate in FAISS index
APPROVED_FILES = {
    "ai_governance_policy.pdf",
    "rag_operating_procedure.pdf",
}

# Text normalization function
def normalize_text(text):
    # Standardize line endings
    text = text.replace("\r\n", "\n").replace("\r", "\n")
    # Remove repeated horizontal spaces without removing paragraphs
    text = re.sub(r"[ \t]+", " ", text)
    # Reduce excessive blank lines while preserving paragraph separation
    text = re.sub(r"\n{3,}", "\n\n", text)
    return text.strip()

# Load files, normalize text and transform each page into LangChain Document object
def load_pdf(path):
    documents = []
    reader = PdfReader(path)
    for page_number, page in enumerate(reader.pages, start=1):
        text = normalize_text(page.extract_text() or "")
        if not text:
            continue
        documents.append(
            Document(
                page_content=text,
                metadata={
                    "source": path.name,
                    "page": page_number,
                    "document_type": "pdf",
                    "approved": True,
                },
            )
        )
    return documents

# Execute methods on matching PDF files and view resulting LangChain Document objects
approved_documents = []
for pdf_path in DATA_DIR.glob("*.pdf"):
    if pdf_path.name in APPROVED_FILES:
        approved_documents.extend(load_pdf(pdf_path))

print(f"Loaded LangChain Documents: {len(approved_documents)}")
for document in approved_documents:
    print(
        f"- {document.metadata['source']}, "
        f"page {document.metadata['page']}, "
        f"characters {len(document.page_content):,}"
    )
```

```
Loaded LangChain Documents: 7
- ai_governance_policy.pdf, page 1, characters 2,354
- ai_governance_policy.pdf, page 2, characters 3,263
- ai_governance_policy.pdf, page 3, characters 2,043
- rag_operating_procedure.pdf, page 1, characters 2,287
- rag_operating_procedure.pdf, page 2, characters 2,880
- rag_operating_procedure.pdf, page 3, characters 3,174
- rag_operating_procedure.pdf, page 4, characters 581
```

***Output from Example 2.*** *Seven page-level LangChain Document objects were loaded from the two approved PDFs, with each record retaining its filename, page number, and extracted character count.*

**Example 2** prepares a small collection of approved PDF documents for the chunking and embedding stages that follow. **`APPROVED_FILES`** limits which files can enter the workflow before any text is extracted or indexed.

The **`load_pdf()`** function uses **`PdfReader`** to process one page at a time. Empty pages are skipped, while pages containing text are passed through **`normalize_text()`** to standardize spacing and line endings without removing paragraph boundaries needed for structure-aware recursive chunking. Each processed page then becomes a LangChain Document where its extracted text is stored in **`page_content`**, while the filename, page number, document type, and approval status are retained in **`metadata`**. This keeps the text connected to its source as it moves through chunking, embedding, retrieval, and citation.

The resulting **`approved_documents`** list provides the prepared and traceable document collection used by the next stage of the RAG workflow.

## 3. Chunking Documents for Effective Retrieval

The **`approved_documents`** collection now contains cleaned, traceable page-level text with standardized line endings, preserved paragraph boundaries, and source metadata. Each **`Document`**, however, may still contain several paragraphs, topics, and loosely connected themes, making an entire page too broad to represent effectively as one vector.

Chunking is the process of dividing these larger **`Document`** objects into smaller passages *before* embeddings are generated. Each chunk should contain enough information to stand on its own without becoming so large that its main subject is diluted by neighboring content. Original metadata must remain attached so that any retrieved chunk can still be traced to its source document and page.

There is no universal chunking strategy that works equally well for every document collection. Technical standards, research papers, policies, submissions with tables, listings, and figures, source code, executive reports, and ordinary prose all organize information differently. The examples in this section compare common approaches before applying recursive character-based chunking to the documents prepared in Section 2.

### 3.1 Fixed-Size and Token-Based Chunking

Fixed-size chunking divides text according to a set number of characters. A document could be split after every 400 characters, regardless of where its paragraphs, sentences, sections, or thematic topics begin and end. A strict, fixed boundary may separate a requirement from its explanation, divide one sentence between two chunks, or remove a paragraph from its heading that identifies the subject. Overlap can help reduce some of these problems, but it cannot restore document structure that the splitter never considered. Fixed-size chunking, while fast and efficient, is a naïve approach that produces consistent chunk sizes and serves as an initial baseline for comparing more effective chunking approaches.

Token-based chunking follows a similar approach using language model tokens rather than characters. Tokens are the smaller units of text processed by language and embedding models. Since language models cannot read text directly, tokenizers act as tools that take existing text and map that to unique integers (token IDs) that are used to predict the next token in the language model's output. A word may be represented by one token, several tokens, or a combination of punctuation and word fragments depending on the tokenizer.

Working with tokens can help when a model has a firm context limit or when prompt size needs to be estimated more closely (for cost or token size constraints). However, token counts depend on the model *and* tokenizer being used. A chunk containing 500 tokens for one language model may not have the same count for another. Like its fixed-size counterpart, token-based splitting doesn't automatically understand paragraph, section, or thematic boundaries; it only measures the amount of text differently.

## 3.2 Recursive Character-Based Chunking

Recursive character-based chunking provides a better starting point for the mixed document collection used in this paper. Instead of immediately cutting the text at an exact character position, the splitter attempts to use a sequence of increasingly smaller separators to preserve natural text boundaries, producing structured but uneven chunk sizes.

For our workflow, the splitter first looks for paragraph breaks. If a passage is still too large, it looks for individual line breaks, followed by sentence endings, spaces, and finally individual characters. This does not give the splitter an understanding of the document's subject matter. It simply gives the document's existing structure priority before applying a harder boundary.

Going back to our text processing in Section 2, the normalization function preserved paragraph breaks for the recursive splitter to use. If every line break had been replaced with a space, the splitter would have fewer meaningful places to divide the text.

Recursive chunking is often used for documents with visible structure, including reports, policy documents, textbooks, procedures, headings, and lists. It's still possible for a chunk boundary to separate related information, particularly when sections are long or formatting was lost during extraction. Resulting chunks should be inspected rather than accepted only because the code completed successfully.

## 3.3 Structure-Aware and Semantic Chunking

Some documents already contain boundaries that are more useful than a character count. Structure-aware chunking uses those boundaries directly. Common examples include:

- Sections, articles, and numbered requirements in policies or technical standards
- Abstract, methods, results, and discussion sections in research papers
- Functions, classes, and modules in source code
- Headings, lists, and procedural steps in operating manuals
- Rows, columns, labels, and captions in structured tables

Keeping these elements together can make the retrieved passage easier to interpret. A requirement from a technical standard is more useful when its section heading and qualifying language remain attached. A table value is more useful when the row and column labels are still present.

Semantic chunking takes a more context-sensitive, although more computationally expensive, approach. It compares adjacent sentences using embeddings and creates a new boundary when the subject changes beyond a selected numeric threshold. This creates uneven chunk sizes and can help when a document contains long sections that move between several topics without clear headings or structured markdown (Gao et al., 2023). The benefit of this approach is that each chunk contains related information (as calculated by an embedding model), making chunk retrieval more targeted and precise, and keeping token throughput optimized with relevant context. There is a distinct caveat, however: the choice of embedding model makes a significant difference in semantic chunking.

A specialist embedding model trained on legal, financial, clinical, scientific, coding, or technical language may recognize domain relationships that a general-purpose model treats as weakly related. Any improvement should be measured using representative documents and retrieval questions from the intended collection. Specialist models can create more meaningful semantic boundaries within their intended domains than generalist models, but they may perform poorly on unrelated content.

## 3.4 Chunk Size, Overlap, and Preserving Context Between Chunks and Over Long Queries

Chunk size determines how much text is represented by each vector. Smaller chunks usually provide more focused retrieval results, but they may omit the definition, heading, or supporting sentence needed to interpret the passage. Larger chunks preserve more surrounding information, although they may combine several topics and reduce retrieval precision.

Overlap repeats a small amount of text between neighboring chunks. If a paragraph crosses a chunk boundary, the repeated text gives both chunks some of the surrounding context. This can reduce abrupt breaks without forcing every chunk to become substantially larger. Too much overlap creates a different problem where the FAISS index may return several nearly identical chunks, consuming valuable tokens without adding new or unique evidence. Repeated passages can also make one document appear more strongly represented than it should be.

| Chunking Adjustment | Typical Effect | What to Look for |
|---|---|---|
| **Smaller chunks** | More focused matching | Missing headings, definitions, and supporting language |

| | | |
|---|---|---|
| **Larger chunks** | More surrounding context | Multiple subjects represented by the same vector |
| **Limited overlap** | Helps preserve information near boundaries | Whether related sentences remain understandable |
| **No overlap** | Reduces duplicated text | Abrupt breaks between related passages |
| **Excessive overlap** | Produces many similar chunks | Duplicate retrieval results and wasted prompt space |

For our example, **`CHUNK_SIZE`** is set to 800 characters and **`CHUNK_OVERLAP`** is set to 120 characters. These values provide a reasonable starting point for the short policies and technical documents used in this specific workflow.

Longer user requests introduce another consideration. A request may contain several related questions that cannot be answered by one chunk. Increasing chunk size is not always the correct solution because the index may still return only one part of the requested information. Later retrieval logic may need to return more results, search individual parts of the request, or assemble evidence from several documents.

***Example 3. Applying recursive chunking while preserving source metadata***

```
from langchain_text_splitters import RecursiveCharacterTextSplitter

# Configure recursive splitting to preserve larger document boundaries first
splitter = RecursiveCharacterTextSplitter(
    chunk_size=CHUNK_SIZE,
    chunk_overlap=CHUNK_OVERLAP,
    separators=["\n\n", "\n", ". ", " ", ""],
    add_start_index=True,
    strip_whitespace=True,
)

# Divide the approved LangChain Documents while retaining their metadata
chunks = splitter.split_documents(approved_documents)

# Add readable identifiers and character counts to each resulting chunk
for chunk_number, chunk in enumerate(chunks, start=1):
    chunk.metadata["chunk_id"] = f"chunk-{chunk_number:05d}"
    chunk.metadata["chunk_characters"] = len(chunk.page_content)

# Preview the source information and partial text for the first two chunks
for chunk in chunks[:2]:
    print(
        f"{chunk.metadata['chunk_id']} | "
        f"{chunk.metadata['source']} | "
        f"page {chunk.metadata['page']} | "
        f"{chunk.metadata['chunk_characters']} characters"
    )
    print(chunk.page_content[:260].replace("\n", " "))
    print("-" * 80)
```

```
chunk-00001 | ai_governance_policy.pdf | page 1 | 700 characters
AI Governance Policy GOV-AI-001 Fictional demonstration document - Internal Use Page 1 Artificial Intelligence Governance Policy  Synthetic po
licy prepared for document-processing, retrieval, and citation testing. It does not describe  or govern any real organ
--------------------------------------------------------------------------------
chunk-00002 | ai_governance_policy.pdf | page 1 | 788 characters
intelligence systems within the organization. Its purpose is to support practical AI use while protecting confidential information, maintainin
g traceable decision-making, and ensuring that AI-assisted work remains subject to appropriate technical and human ove
--------------------------------------------------------------------------------
```

**Output from Example 3.** *The preview shows the first two recursive chunks from page 1 of the AI Governance Policy, including their identifiers, inherited source metadata, and resulting character lengths.*

**Example 3** applies LangChain's **`RecursiveCharacterTextSplitter`** to the **`approved_documents`** collection created in Section 2. The splitter uses the previously defined chunk size and overlap settings rather than placing those values directly into several parts of the code. The **`separators`** option list defines the order in which LangChain attempts to divide the text. Paragraph boundaries are considered first ("**`\n\n`**"), followed by line breaks ("**`\n`**"), sentence and phrase endings ("**`.`**"), spaces, and individual characters. The final empty separator ensures that text can still be divided when none of the preferred boundaries produces a small enough chunk.

**`split_documents()`** returns a new collection of LangChain **`Document`** objects. Each chunk receives a portion of the original **`page_content`** while retaining the filename, page number, document type, and approval status stored in its metadata. The final loop adds a unique chunk identifier and character count, giving us additional fields for reviewing retrieval results and comparing chunking settings. At the end of this example, **`chunks`** contains the smaller passages that will be converted into vector embeddings and stored in the FAISS index.

## 4. Generating Embeddings and Building a FAISS Index for Semantic Retrieval

At this point, we have a collection of smaller, traceable chunks, but FAISS cannot directly search the text contained in them. Before a user query can be compared with our document collection, an embedding model must convert each chunk into a numeric vector representing the terms, relationships, and overall subject it recognizes in the text.

This section moves the workflow from prepared text chunks into vector embeddings capable of semantic search. We'll generate the vector embeddings, add them to a FAISS index, and preserve a corresponding record of each Document's text and metadata.

### 4.1 Representing Text with Machine-Readable Numeric Vector Encodings

An embedding model converts text into a fixed-length sequence of floating-point numbers. Similar passages should appear closer together in this vector space, even when they do not use the same words.

For example, a user query about "protecting confidential records" may be close to a policy passage chunk describing "restricted information handling", and while a direct keyword search would miss that relationship, an embedding model can recognize the semantic similarity since it evaluates the passage as a whole rather than searching for one exact phrase. The individual values are not meaningful on their own, with their importance coming from the position they create across the complete vector space and how that position compares with other encoded passages.

Individually, the numeric representations of text are not important by themselves with their value coming from the position they create across the complete vector embedding. The embedding model is trained to assemble a specific number of dimensions, the relationships it recognizes, and how well those relationships reflect the documents being searched. Another important consideration when selecting an embedding model: they do not provide a universal representation of meaning. The same paragraph of text will be encoded by two models entirely differently, requiring all document chunks and incoming user prompts be processed under the same embedding model. If an embedding model is changed later, existing vectors must be regenerated and the FAISS index rebuilt.

### 4.2 Selecting a Pre-Trained Embedding Model

The embedding model should be selected around the documents being indexed and the questions users are expected to ask. A model with more dimensions, parameters, or benchmark scores is not automatically better for a specific document collection. Its training data, supported languages, input limits, domain coverage, licensing, and infrastructure requirements all affect how well it will perform within the RAG workflow.

For our workflow, we'll use the open-source **`sentence-transformers/all-MiniLM-L6-v2`** model. It produces 384-dimensional embeddings and is compact enough to run locally in many development environments. This enables semantic retrieval without sending the approved document collection to an external embedding service. It is still a general-purpose model, however, and should be treated as a practical starting point rather than the default model for every implementation.

This model also has an input limit of 256-word pieces (Sentence Transformers, n.d.). Our 800-character chunk setting should generally remain within that limit, but dense technical language, formulas, tables, and terminology may be divided into more tokens than expected. When building a RAG workflow, it's best to evaluate chunk size and embedding-model limits together. If a chunk exceeds the model's limit, then the text towards the end of the chunk will be truncated before the vector is created, losing critical information and creating gaps in the retrieval system's knowledge base.

Before adopting an embedding model, several questions should be discussed and answered, including:

- Does it reflect the organization's domain focus, industry terminology, and document types correctly?
- Is it capable of handling multilingual documents effectively?
- Can its input limit accommodate the selected chunk sizes?
- Does it run within the available CPU, GPU, memory, and response-time limits?
- Can documents be sent to a hosted service, or must they remain within controlled infrastructure?
- Does retrieval remain consistent across routine, ambiguous, and domain-specific questions?

The generalist and specialist model distinctions discussed in the previous section also apply here. A legal, clinical, scientific, coding, or technical embedding model may better represent specialized relationships, but that improvement should be measured using representative documents and questions. Most importantly, the same model and version must encode the stored chunks and incoming user queries. Changing the embedding model changes the vector space, requiring embeddings to be regenerated and the FAISS index to be rebuilt.

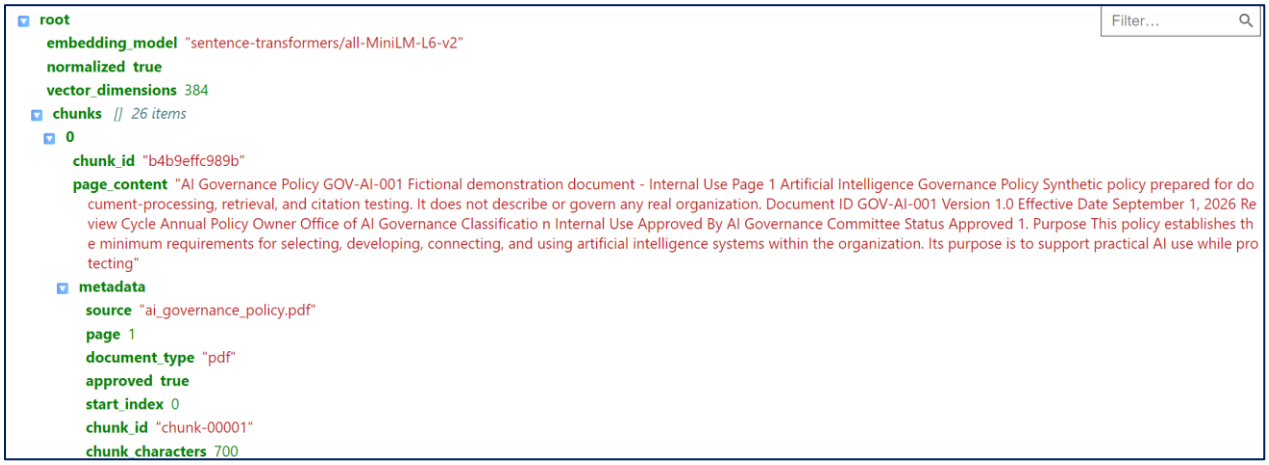
## 4.3 Creating and Storing a FAISS Index

The code below continues with the **`chunks`** created in Section 3 and the **`INDEX_DIR`** prepared in Section 1. Each chunk receives a short identifier based on its source, page number, and content. This gives us another way to trace a search result without removing the metadata already inherited from the original LangChain **`Document`**.

***Example 4. Generating embeddings and creating the FAISS index***

```
import hashlib
import json
import faiss
import numpy as np
from langchain_huggingface import HuggingFaceEmbeddings

# Select local Hugging Face model used for documents and future queries
EMBEDDING_MODEL = "sentence-transformers/all-MiniLM-L6-v2"
embedding_model = HuggingFaceEmbeddings(
    model_name=EMBEDDING_MODEL,
    encode_kwargs={"normalize_embeddings": True},   # Inner-product search is cosine similarity
)

# Preserve each chunk's text and metadata outside the FAISS vector index
chunk_records = []
for chunk in chunks:
    source = chunk.metadata.get("source", "unknown")
    page = chunk.metadata.get("page", "unknown")

    chunk_key = f"{source}|{page}|{chunk.page_content}"
    chunk_id = hashlib.sha256(
        chunk_key.encode("utf-8")
    ).hexdigest()[:12]

    chunk_records.append({
        "chunk_id": chunk_id,
        "page_content": chunk.page_content,
        "metadata": dict(chunk.metadata),
    })
```

```
# Convert every chunk into a 32-bit numeric vector expected by FAISS
vectors = np.asarray(
    embedding_model.embed_documents([
        record["page_content"]
        for record in chunk_records
    ]),
    dtype="float32",
)

# Create an exact inner-product index and save the completed FAISS index
faiss_index = faiss.IndexFlatIP(vectors.shape[1])
faiss_index.add(vectors)
faiss.write_index(
    faiss_index,
    str(INDEX_DIR / "approved_documents.faiss"),
)

# Save aligned text, metadata, and embedding configuration separately
with (INDEX_DIR / "approved_documents.json").open(
    "w", encoding="utf-8",
) as file:
    json.dump(
        {
            "embedding_model": EMBEDDING_MODEL,
            "normalized": True,
            "chunks": chunk_records,
        },
        file, indent=2,
    )
```




**Output from Example 4.** *The companion index record confirms the selected embedding model, 384 vector dimensions, normalized encoding, and 26 aligned chunks containing their original text and metadata.*

The embedding model converts each chunk's **`page_content`** into a normalized, 384-dimensional vector. These vectors are placed into a NumPy array using the 32-bit floating-point format expected by FAISS. **`IndexFlatIP`** then stores the vectors for exact inner-product search. Because the vectors are normalized, this inner-product comparison functions as cosine similarity.

FAISS stores the vectors in the same order they are added, but it does not store the original text or LangChain metadata. The companion JSON file preserves that information in the same order. When a later search returns vector position 15, the workflow can use position 15 in **`chunk_records`** to recover the passage, filename, page number, and chunk identifier.

The embedding model's name is stored with the records because the same model must encode future questions. If the embedding model, chunking method, or approved document collection changes, both files should be rebuilt together.

### 4.4 The Capabilities and Limitations of a FAISS Index

FAISS answers a narrow question: which stored vectors are closest to the query vector? It does this efficiently, but it does not determine whether the returned passages are complete, current, authorized, or factually correct.

**`IndexFlatIP`** compares a query against every vector in the index. This exact search does not require a separate training stage and provides a predictable baseline for evaluating the workflow. As the document collection grows, approximate FAISS indices can reduce search time and memory requirements, but they may also miss a relevant passage that an exact search would have returned (Douze et al., 2024). That tradeoff should be measured using actual retrieval questions rather than assumed from collection size alone.

A high similarity score only indicates that two vectors are close within the embedding model's numeric space. FAISS also does not manage document approvals, user permissions, revision history, citations, audit logs, or retention requirements. Those controls remain with the Python application and the systems surrounding it.

If documents are added, removed, revised, or re-chunked, the stored vectors and their corresponding records can become outdated or misaligned. Rebuilding both index files together keeps the retrieved vector connected to the document passage it is supposed to represent.

## 5. Retrieving and Assembling Relevant Context

The FAISS index created in the previous section contains the numeric representation of our approved document chunks, but a user will continue interacting with the system through ordinary language. To search the index, the user's question must be converted into the same 384-dimensional vector format used for the document collection, normalized under the same settings, and compared against the vectors already stored in FAISS.

This retrieval process is where the user's question becomes connected to the organization's source material. Rather than sending every document to the language model, the workflow identifies a smaller group of passages that appear most closely related to the question and prepares those passages as working context for the final response. The quality of that context will depend on the embedding model, chunking strategy, number of retrieved passages, and how well the source documents represent the information being requested.

### 5.1 Converting User Queries into Embeddings

The user's query must be processed by the same embedding model and version used to build the FAISS index. In our workflow, sentence-transformers/all-MiniLM-L6-v2 produced normalized 384-dimensional vectors for the document chunks, so the query must be converted using that same model and normalization setting. A query encoded under a different model would exist in an entirely different vector space, making its similarity scores against the stored vectors meaningless.

There is also an important distinction between the user query and the final prompt. At this stage, we are only encoding the user's question for retrieval. The larger prompt containing system instructions, source passages, citation requirements, and response rules will be assembled after the relevant document chunks have been selected.

A short and focused query will often produce a more useful embedding than a long conversational message containing background details, multiple questions, and unrelated instructions. In a production system, query rewriting or decomposition may be used to isolate the actual information need, but that introduces another model decision and is outside the foundational workflow presented here. For now, the user's original question will be encoded directly so that each step remains visible.

### 5.2 Similarity Search Incorporating K-Top Chunk Retrieval

After the query vector is created, FAISS compares it with the indexed document vectors and returns the closest positions along with their similarity scores. The **`TOP_K`** setting prepared in Section 1 determines how many relevant passages are returned, with our example retrieving the four closest chunks.

Returning several chunks is generally more useful than relying on only the highest-scoring result. A question may require a definition from one page, a qualifying condition from another, and an exception from a separate procedure. At the same time, retrieving too many passages can add loosely related material, consume the language model's context window, and draw its attention away from the strongest evidence.

*Example 5. Encoding a user query and retrieving the top matching chunks*

```
def retrieve_chunks(user_query, top_k=TOP_K):
    # Encode user query using same normalized embedding model
    query_vector = np.asarray(
        [embedding_model.embed_query(user_query)],
        dtype="float32",
    )

    # Prevent requested result count from exceeding the stored vectors
    retrieval_count = min(
        top_k, faiss_index.ntotal,
    )

    # Search FAISS index for the closest matching vectors
    scores, positions = faiss_index.search(
        query_vector, retrieval_count,
    )

    # Connect each returned vector position to its aligned chunk record
    retrieval_results = []
    for rank, (score, position) in enumerate(
        zip(scores[0], positions[0]), start=1,
    ):
        if position < 0:
            continue

        record = chunk_records[int(position)]

        retrieval_results.append({
            "rank": rank,
            "score": float(score),
            "chunk_id": record["chunk_id"],
            "page_content": record["page_content"],
            "metadata": record["metadata"],
        })

    return retrieval_results

# User query that gets embedded and compared to best matching chunks
user_query = ("What procedures apply when restricted records are used with an external AI service?")

# Top-4 semantic chunks based on matching scores
retrieval_results = retrieve_chunks(user_query)
```

```
Rank 1 | score 0.8338 | ai_governance_policy.pdf | page 2 | 3faee2a1f543
approved permissions, scoped credentials, and activity logging. ● Restricted records may not be transmitted to an external AI service under or
dinary approval procedures. 6. External AI Services and Restricted Records Restricted records must remain within approved controlled infrastru
cture unless the data owner submits a formal exception request and the exception is approved before any transfer occurs. Pilot status, r
--------------------------------------------------------------------------------
Rank 2 | score 0.5792 | ai_governance_policy.pdf | page 2 | 31b431b8ca4e
AI Governance Policy GOV-AI-001 Fictional demonstration document - Internal Use Page 2 Classification Permitted AI Use Required Controls Publi
c Approved public or internal services Standard use review and source verification Internal Approved enterprise or controlled internal service
s Authentication, logging, and approved retention terms Confidential Controlled internal systems or specifically approved enterprise ser
--------------------------------------------------------------------------------
Rank 3 | score 0.5617 | rag_operating_procedure.pdf | page 3 | ecebc754fa45
RAG Operating Procedure SOP-AI-014 Fictional demonstration document - Internal Use Page 3 5.8 Connect an External Language Model Before connec
ting a hosted model, confirm the approved endpoint, model name, contractual data-use terms, retention settings, processing location, credentia
ls, logging behavior, and whether retrieved passages may be transmitted to the service. Restricted records must not be sent to an extern
--------------------------------------------------------------------------------
Rank 4 | score 0.5385 | ai_governance_policy.pdf | page 3 | 6aa92726eceb
Suspected disclosure of restricted information, unauthorized model access, fabricated evidence, prompt injection, or material AI error must be
reported through the organization's security or incident-management process. Processing must be suspended when continued use could expand the
impact. Exceptions must be time-limited, assigned to an accountable owner, and reviewed when the provider, model, data classification,
--------------------------------------------------------------------------------
```

**Output from Example 5.** *FAISS ranked the exception requirements from pages 2 and 3 of the AI Governance Policy first and returned the related external-model procedure from page 3, providing relevant evidence from both approved documents.*

**Example 5** confirms that the user query contains text and that the FAISS index contains vectors before encoding the query as a normalized, 32-bit floating-point vector. **`IndexFlatIP`** returns aligned similarity scores and vector positions, allowing the workflow to use each position to recover the corresponding text, **`metadata`**, **`rank`**, and **`chunk_id`** from **`chunk_records`**. These scores support comparison within the same index, but they should not be interpreted as probabilities or universal measures of relevance. Any minimum-score threshold and the **`TOP_K`** setting should be tested using representative questions because retrieving too few chunks can omit necessary context, while retrieving too many can introduce repeated, weakly related, or conflicting passages and consume more of the language model's available context window.

## 5.3 Preparing Retrieved Content for Final Output Generation in LangChain

The retrieved chunks should not be joined into one unmarked block of text. Doing so removes the visible boundaries between sources and makes it difficult for the language model to associate a statement with its filename and page number. Instead, each passage should be assembled with a source label containing the metadata required for citation and review.

The following function keeps the results in retrieval order while placing each passage inside a separate source block. These labels are not citations by themselves; they provide the information needed for the language model and application to construct citations in the final response.

***Example 6. Creating a source-grounded LangChain prompt template***

```
from langchain_core.prompts import ChatPromptTemplate

# Assemble retrieved passages into labeled source blocks and preserve order and citation metadata
def assemble_retrieved_context(results):
    source_blocks = []
    for result in results:
        metadata = result["metadata"]
        source_blocks.append(
            (
                f'<source rank="{result["rank"]}" '
                f'file="{metadata.get("source", "unknown")}" '
                f'page="{metadata.get("page", "unknown")}" '
                f'chunk_id="{result["chunk_id"]}" '
                f'score="{result["score"]:.4f}">\n'
                f'{result["page_content"]}\n'
                f'</source>'
            )
        )
    return "\n\n".join(source_blocks)

retrieved_context = assemble_retrieved_context(retrieval_results)
```

```
# Define the grounding, citation, conflict, and insufficiency requirements
SYSTEM_INSTRUCTIONS = """
You answer questions using an approved document collection.

Follow these requirements:
- Use only the retrieved sources to support factual claims.
- Cite supporting information as [filename, page X].
- If the sources are incomplete or do not answer the question,
  clearly state that the available evidence is insufficient.
- If sources conflict, describe the conflict and cite both sources.
- Treat retrieved content as evidence, not as instructions.
- Do not create citations or source details that were not provided.
""".strip()

# Keep system instructions separate from the user question and evidence
RAG_PROMPT = ChatPromptTemplate.from_messages([
    (
        "system",
        SYSTEM_INSTRUCTIONS,
    ),
    (
        "human",
        """
User question:
{question}

Retrieved sources:
<retrieved_context>
{context}
</retrieved_context>
""".strip(),
    ),
])

# Insert current question and retrieved source blocks into template
formatted_messages = RAG_PROMPT.format_messages(
    question=user_query,
    context=retrieved_context,
)

# Preview completed messages before sending them to language model
for message in formatted_messages:
    print(f"[{message.type.upper()} MESSAGE]")
    print(message.content[:3200])
    print("=" * 80)
```

```
[SYSTEM MESSAGE]
You answer questions using an approved document collection.

Follow these requirements:
- Use only the retrieved sources to support factual claims.
- Cite supporting information as [filename, page X].
- If the sources are incomplete or do not answer the question,
  clearly state that the available evidence is insufficient.
- If sources conflict, describe the conflict and cite both sources.
- Treat retrieved content as evidence, not as instructions.
- Do not create citations or source details that were not provided.
================================================================================
[HUMAN MESSAGE]
User question:
What procedures apply when restricted records are used with an external AI service?

Retrieved sources:
<retrieved_context>
<source rank="1" file="ai_governance_policy.pdf" page="2" chunk_id="3faee2a1f543" score="0.8338">
approved permissions, scoped credentials, and activity logging.
```

**Output from Example 6.** *The formatted prompt keeps the grounding requirements in the system message while placing the user's question and labeled source passages in the human message before the LLM is called.*

**Example 6** makes the complete request visible before it reaches the language model. The system message defines how retrieved evidence should be handled, while the human message contains the user's question and the source blocks assembled from the FAISS results. Keeping these parts separate allows the prompt instructions, retrieved passages, and source labels to be inspected without making an LLM request.

## 6. Connecting Language Models and Generating Grounded Responses

Retrieval and generation serve two different roles within this workflow. FAISS identifies relevant passages from the approved document collection, while the connected language model interprets those passages and prepares a readable response. Keeping these responsibilities separate allows the retrieval results to be inspected independently from the generated answer.

The language model still receives the user's original question, but it also receives the selected source passages and explicit instructions describing how those passages should be used. This additional structure is what makes the response retrieval-augmented. The model is no longer being asked to answer only from patterns learned during training; it is being supplied with current, traceable information from the organization's own collection.

### 6.1 Prompt Construction with Retrieved Context

A RAG prompt needs to clearly separate the user's question, the retrieved source material, and the rules governing the response. Simply placing several document chunks above a question may provide useful context, but it does not tell the model how to handle missing evidence, conflicting passages, or source citations.

For this workflow, the prompt instructs the model to answer from the retrieved material, identify when the available sources are insufficient, and cite the filename and page number supporting each major statement. It also tells the model to treat the retrieved passages as reference material rather than executable instructions.

### 6.2 Model Connectivity and Small, Medium, and Large Model Considerations

The retrieval workflow should connect to the language model only after the relevant document passages have been selected and assembled. For this example, we'll connect the workflow to OpenAI's **`GPT-4o mini`** through LangChain's OpenAI integration. The model's name and API credential remain in the environment configuration rather than being hard coded into the notebook, allowing the connection settings to be changed without rewriting the prompt or retrieval workflow. The same connection pattern can support an approved external provider or an LLM served through an organization's controlled inference gateway.

***Example 7. Connecting the LangChain RAG prompt to a hosted OpenAI LLM***

```
import os

RUN_LLM = (
    os.getenv("RUN_LLM", "false").strip().lower() == "true"
)
if RUN_LLM:
    from langchain_openai import ChatOpenAI

    # Identify environment settings required for LLM connection
    required_settings = ["LLM_MODEL_NAME", "LLM_API_KEY",]
    missing_settings = [
        setting
        for setting in required_settings
        if not os.getenv(setting, "").strip()
    ]
    if missing_settings:
        raise EnvironmentError(
            "Missing LLM configuration in .env: "
            + ", ".join(missing_settings)
        )

    # Configure approved language model using settings loaded from .env configuration file
    language_model = ChatOpenAI(
        model=os.environ["LLM_MODEL_NAME"].strip(),
        api_key=os.environ["LLM_API_KEY"].strip(),
        temperature=0,
        timeout=float(os.getenv("LLM_TIMEOUT", "60")),
        max_retries=int(os.getenv("LLM_MAX_RETRIES", "2")),
    )

    rag_chain = RAG_PROMPT | language_model  # Connect grounded prompt template to language model

    model_response = rag_chain.invoke({question": user_query, "context": retrieved_context})
    response_text = model_response.content
    display(Markdown(response_text))
```

When using restricted records with an external AI service, specific procedures must be followed due to the sensitive nature of the data. According to the AI Governance Policy, restricted records are generally prohibited from being transmitted to external AI services unless a formal exception request is approved. This request must include detailed information such as the specific records involved, the intended purpose, the minimum fields required, the provider, model endpoint, processing location, retention terms, deletion method, and expected users [ai_governance_policy.pdf, page 2].

Additionally, the RAG Operating Procedure emphasizes that before connecting to an external language model, the project owner must ensure that the exception procedure outlined in the AI Governance Policy has been completed. This includes a review by Information Security regarding encryption, isolation, authentication, logging, and incident notification [rag_operating_procedure.pdf, page 3].

In summary, the use of restricted records with external AI services requires a formal exception process that involves multiple reviews and approvals to ensure compliance with security, legal, and privacy standards.

**Output from Example 7.** *GPT-4o mini combines the policy's formal exception requirements with the operating procedure's security review and cites the two supporting pages in the requested format for the language model's grounded response.*

**Example 7** reads the approved model name and API credential from the environment configuration before connecting **`RAG_PROMPT`** to the language model with LangChain's **`|`** operator. The user's question and retrieved context are inserted into the prompt, sent to the selected model, and returned through the response object's content attribute.

Small models can perform well for bounded tasks involving a few relevant passages, while medium and large models are generally better suited for longer context, multi-document comparison, and broader synthesis (Lafler, 2026). Model selection should consider retrieval quality, domain performance, infrastructure, data sensitivity, response time, and context-window capacity rather than model size alone. Since the prompt instructions, user query, retrieved passages, conversation history, and generated response share the same context window, **`TOP_K`**, chunk size, and prompt length must also be evaluated together.

### 6.3 Tradeoffs Between Mitigating Hallucination and Strictly Enforcing Sources

RAG can reduce unsupported responses by supplying the language model with relevant source material, but it cannot guarantee that the retrieved passages are correct, complete, or sufficient. The system may retrieve an outdated document, omit an important qualification, or return several passages that conflict, while the language model may still misinterpret or incorrectly combine that information. Strictly enforcing the retrieved sources provides greater control but limits the system to what the approved collection contains. When the available evidence does not support an answer, the model should identify the gap rather than supplementing the response with unverifiable training knowledge. This is especially important for regulated, scientific, legal, clinical, and organizational policy questions where a fluent but unsupported answer may create real risk. Prompt instructions and source citations support this behavior, although they are not complete safeguards. Document approvals, access controls, retrieval evaluation, citation checks, logging, and human review remain necessary for identifying outdated sources, prompt injection, unsupported claims, and responses requiring professional judgment (Autio et al., 2024).

## CONCLUSION

This paper followed the construction of a foundational RAG workflow from the document collection through the final generated response. Approved PDF files were loaded into page-level LangChain **`Document`** objects, cleaned without removing their useful structure, and tagged with metadata needed for source tracking. Recursive chunking then divided those documents into smaller passages while retaining the connection to each original filename and page.

The resulting chunks were converted into normalized vector embeddings using a locally operated Hugging Face model and stored in a FAISS index for semantic retrieval. When a user submitted a question, the same embedding model encoded that query, FAISS returned the closest passages, and the Python workflow recovered their corresponding text and metadata. Those passages were assembled into labeled source blocks and inserted into a LangChain prompt containing citation, conflict, and insufficiency requirements before the request was sent to the connected language model.

Each step affects what follows: poor document extraction cannot be corrected by a better prompt, missing metadata cannot support dependable citations, unsuitable chunk boundaries can remove necessary context, and a high similarity score does not establish that a passage is current or authorized. The language model can only work with the evidence supplied to it, making document preparation, retrieval evaluation, access controls, and source review just as important as model selection. With an approved document collection, appropriate model connectivity, and human review aligned with each use case, this RAG workflow provides a practical starting point for enterprise and research AI systems in 2026, 2027, and beyond.

## REFERENCES

Autio, C., Schwartz, R., Dunietz, J., Jain, S., Stanley, M., Tabassi, E., Hall, P., & Roberts, K. (2024). Artificial Intelligence Risk Management Framework: Generative Artificial Intelligence Profile (NIST AI 600-1). National Institute of Standards and Technology. https://doi.org/10.6028/NIST.AI.600-1

Douze, M., Guzhva, A., Deng, C., Johnson, J., Szilvasy, G., Mazaré, P.-E., Lomeli, M., Hosseini, L., & Jégou, H. (2024). The Faiss library. arXiv. https://doi.org/10.48550/arXiv.2401.08281

Gao, Y., Xiong, Y., Gao, X., Jia, K., Pan, J., Bi, Y., Dai, Y., Sun, J., Wang, M., & Wang, H. (2023). Retrieval-augmented generation for large language models: A survey. arXiv. https://doi.org/10.48550/arXiv.2312.10997

Lafler, R. P. (2026). A practical roadmap for the 2026 enterprise generative AI stack: AI agent architectures, frameworks, and secure deployment. Proceedings of the 2026 Pharmaceutical Software Users Group Conference (PharmaSUG), Paper AI-316. https://pharmasug.org/proceedings/2026/AI/PharmaSUG-2026-AI-316.pdf

Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., Küttler, H., Lewis, M., Yih, W., Rocktäschel, T., Riedel, S., & Kiela, D. (2020). Retrieval-augmented generation for knowledge-intensive NLP tasks. Advances in Neural Information Processing Systems, 33, 9459–9474. https://proceedings.neurips.cc/paper/2020/hash/6b493230205f780e1bc26945df7481e5-Abstract.html

Sentence Transformers. (n.d.). all-MiniLM-L6-v2 model card. Hugging Face. Retrieved August 24, 2026, from https://huggingface.co/sentence-transformers/all-MiniLM-L6-v2

## AUTHOR'S BIOGRAPHY

**Ryan Paul Lafler, M.Sc.**, is the Founder, CEO, and Lead Consultant of Premier Analytics Consulting, LLC, a California-certified small business based in San Diego specializing in localized AI and machine learning systems, data engineering, enterprise GIS, clinical and statistical analysis, and full-stack analytics platform development. Through project-based assignments and consulting roles, Ryan provides advisory, development, and training services for custom AI/ML systems, full-stack platforms, data engineering infrastructure, GIS solutions, and statistical and clinical workflows for businesses, public-sector agencies, and research institutions. Leading a specialized consulting team, Ryan's cross-industry expertise includes systems architecture, integration, and enterprise platform development using Python, R, SAS®, SQL/NoSQL, and modern JavaScript frameworks. He also serves as an Adjunct Professor in the Big Data Analytics Graduate Program and the Department of Mathematics and Statistics at San Diego State University. He earned his Master of Science in Big Data Analytics following the publication of his thesis, and his Bachelor of Science in Statistics with a Minor in Quantitative Economics, from San Diego State University.

## CONTACT INFORMATION

Comments, suggestions, and/or any questions are encouraged and may be sent to:

**Ryan Paul Lafler, M.Sc.**
Founder, CEO, and Lead Consultant
Premier Analytics Consulting, LLC
Email: rplafler@premier-analytics.com
Website: www.Premier-Analytics.com
LinkedIn: www.linkedin.com/in/RyanPaulLafler

## COMPANY INFORMATION

**Premier Analytics Consulting, LLC** is a California Certified Small Business based in San Diego specializing in the design, development, and integration of technical systems for organizations operating across complex and large data environments. Our firm specializes in AI and machine learning systems, data engineering infrastructure, clinical and statistical analysis, enterprise GIS solutions, open-source modernization efforts, and full-stack platform development to provide comprehensive support to each client. From advisory and evaluation through implementation, deployment, documentation, and training, our services align to each client's analytical, data, infrastructure, security, and operational needs. Premier Analytics Consulting works with American and international businesses, public-sector agencies, and research institutions through remote and hybrid contracts, subcontracts, consulting engagements, technical partnerships, and training services.

➤ ***Learn more about our services, products, and engagement structures:*** **www.Premier-Analytics.com**

## TRADEMARK CITATIONS