Retrieval Augmented Generation

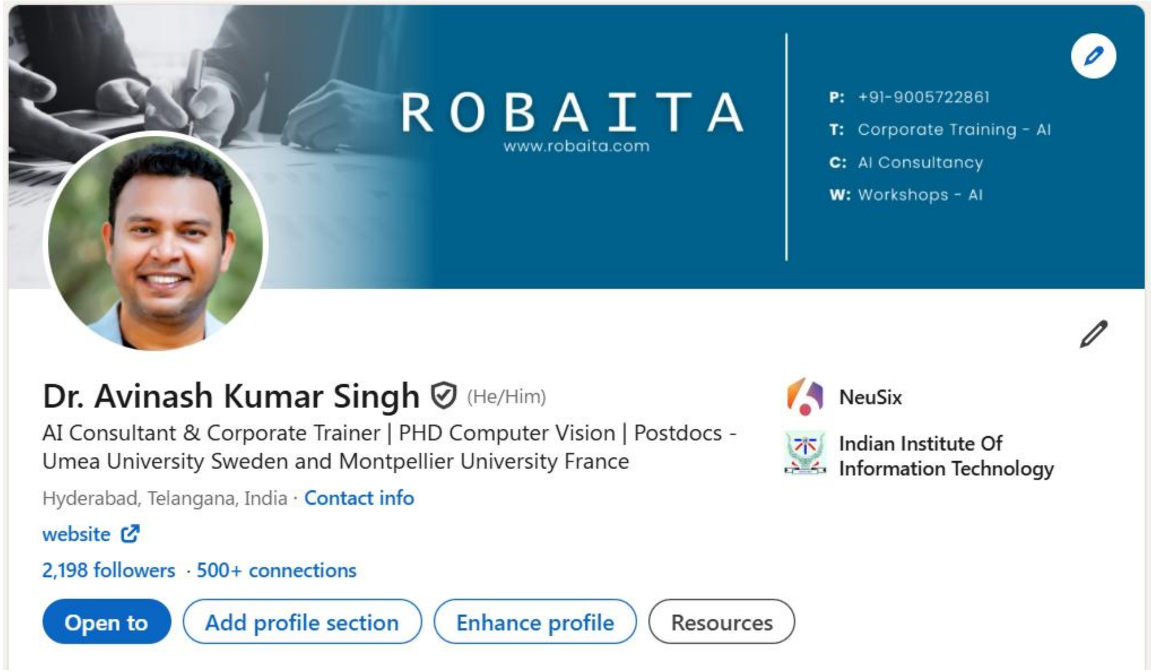
RAG 2.0 – Semantic Search, Re-ranking, and Trustworthy Generation

Dr. Avinash Kumar Singh

AI Consultant and Coach, Robaita

Dr. Avinash Kumar Singh

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



Dr. Avinash Kumar Singh (He/Him)

AI Consultant & Corporate Trainer | PHD Computer Vision | Postdocs - Umea University Sweden and Montpellier University France

Hyderabad, Telangana, India · Contact info

website

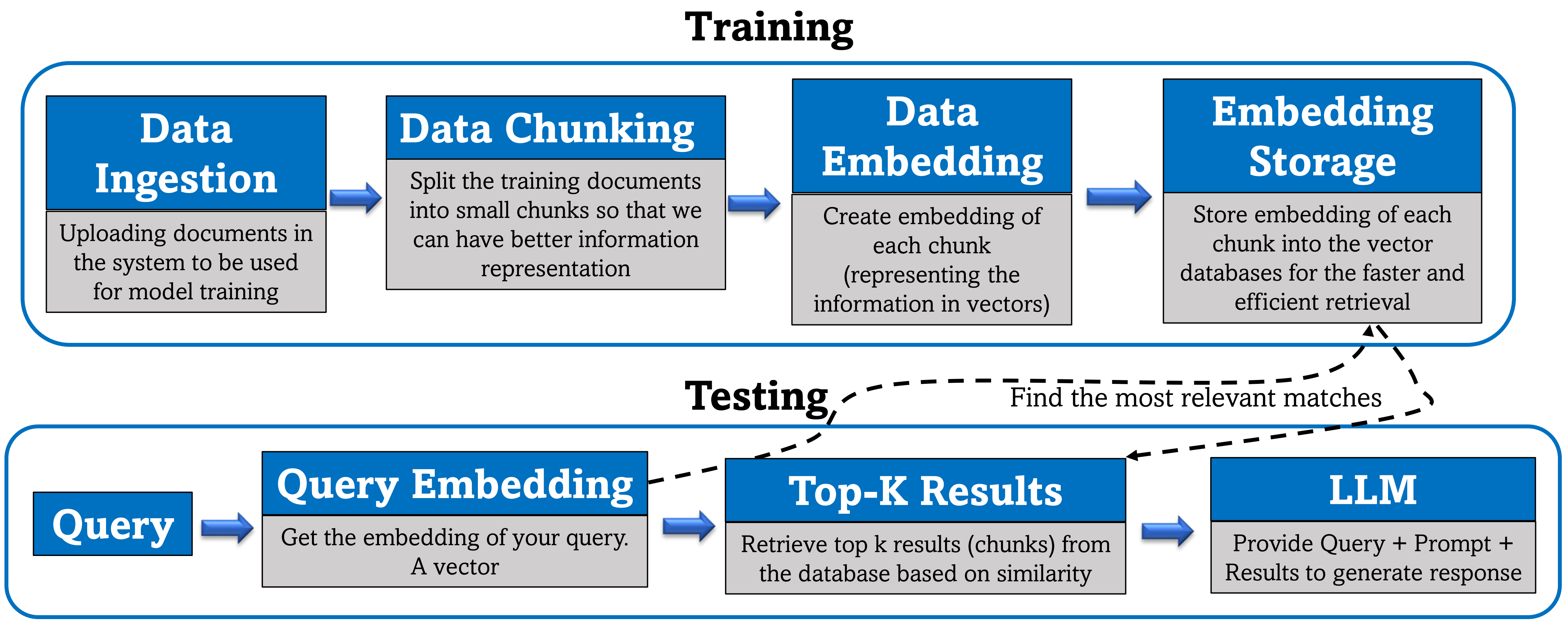
2,198 followers · 500+ connections

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

Discussion Points

- Data Chunking: Dynamic header generation, Embedding-aware section titles, Summarization-based token reduction, Importance-aware trimming, LLM-assisted compression
- Storage and Retrieval: Dense retrieval, Sparse retrieval (BM25), Combining scores from both, Attribute-based filtering, Facet tagging in vector DBs, User-defined query constraints, Paragraph-level scoring, Semantic overlap with query, Focused context injection, Ensemble retrieval strategies
- Prompt Engineering & LLM for Answer Generation: Reducing hallucinations with verification, Confidence scoring via LLMs, Audit trails for traceability, Intelligent reranking, LLM based scoring, Use of RAGAS metrics, Ground truth-based scoring, Human-in-the-loop validation

RAG Pipeline


Training
Data Ingestion
Uploading documents in the system to be used for model training
Data Chunking
Split the training documents into small chunks so that we can have better information representation
Data Embedding
Create embedding of each chunk (representing the information in vectors)
Embedding Storage
Store embedding of each chunk into the vector databases for the faster and efficient retrieval
Testing
Find the most relevant matches
Query
Query Embedding
Get the embedding of your query. A vector
Top-K Results
Retrieve top k results (chunks) from the database based on similarity
LLM
Provide Query + Prompt + Results to generate response

Data Chunking

Data Chunking

Dynamic Header Generation

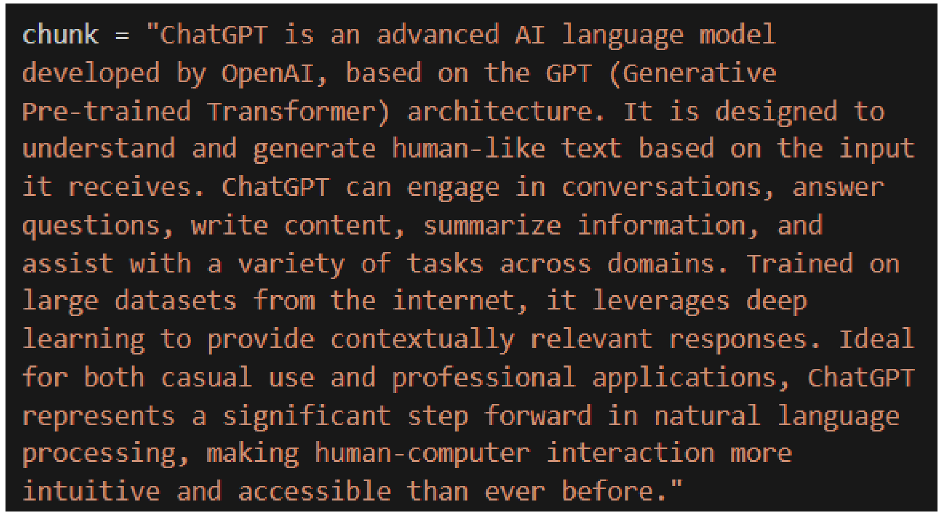
Dynamic header generation refers to the process of creating contextual headings for each chunk based on its content. This provides semantic anchors that guide retrieval and improve understanding during both indexing and generation.

How it works

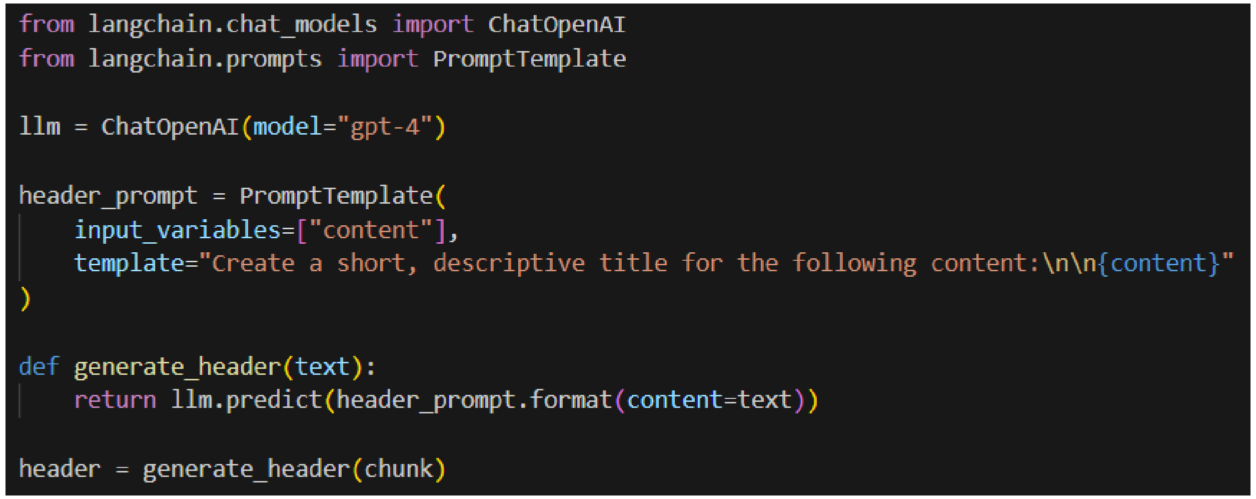
Instead of using static titles like "Section 1", the system scans the content of a chunk and generates a title dynamically. This is especially useful when the source content lacks structure (e.g., raw text or scanned documents).

Benefits

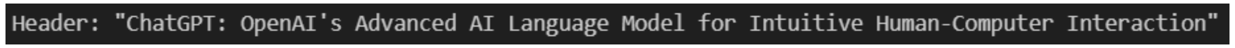
- Adds semantic indexing.
- Enhances retrieval ranking by matching user queries to dynamic headers.
- Provides context to the LLM even before chunk content is read.



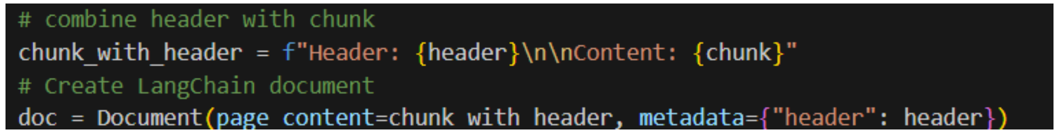
```
chunk = "ChatGPT is an advanced AI language model
developed by OpenAI, based on the GPT (Generative
Pre-trained Transformer) architecture. It is designed to
understand and generate human-like text based on the input
it receives. ChatGPT can engage in conversations, answer
questions, write content, summarize information, and
assist with a variety of tasks across domains. Trained on
large datasets from the internet, it leverages deep
learning to provide contextually relevant responses. Ideal
for both casual use and professional applications, ChatGPT
represents a significant step forward in natural language
processing, making human-computer interaction more
intuitive and accessible than ever before."
```



```python
from langchain.chat_models import ChatOpenAI
from langchain.prompts import PromptTemplate

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

header_prompt = PromptTemplate(
    input_variables=["content"],
    template="Create a short, descriptive title for the following content:\n\n{content}"
)

def generate_header(text):
    return llm.predict(header_prompt.format(content=text))

header = generate_header(chunk)
```



```
Header: "ChatGPT: OpenAI's Advanced AI Language Model for Intuitive Human-Computer Interaction"
```



```python
# combine header with chunk
chunk_with_header = f"Header: {header}\n\nContent: {chunk}"
# Create LangChain document
doc = Document(page_content=chunk_with_header, metadata={"header": header})
```

Data Chunking

Embedding-Aware Section Titles

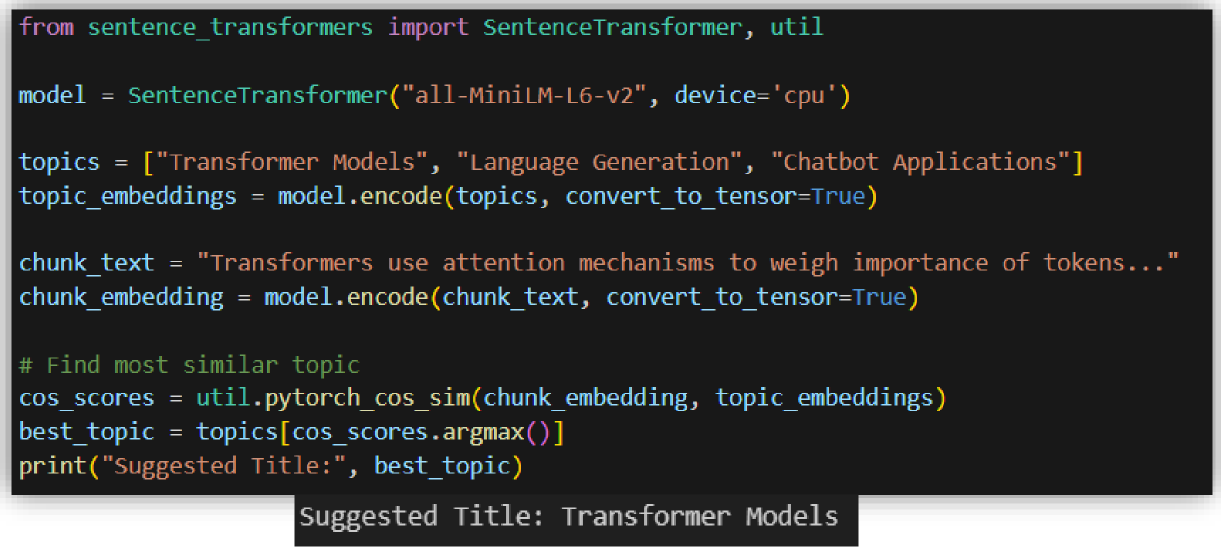
Generates section titles based on the semantic similarity of embeddings, ensuring the title reflects the main theme of the chunk at a vector level.

How it works

Instead of just using raw text, the system compares the chunk's embedding with pre-defined topic vectors or generates a title aligned with the chunk's semantic centroid.

Benefits

- Provides LLM-friendly summaries during indexing.
- Boosts retrieval by aligning chunk headers with user intent semantically.



```
from sentence_transformers import SentenceTransformer, util

model = SentenceTransformer("all-MiniLM-L6-v2", device='cpu')

topics = ["Transformer Models", "Language Generation", "Chatbot Applications"]
topic_embeddings = model.encode(topics, convert_to_tensor=True)

chunk_text = "Transformers use attention mechanisms to weigh importance of tokens..."
chunk_embedding = model.encode(chunk_text, convert_to_tensor=True)

# Find most similar topic
cos_scores = util.pytorch_cos_sim(chunk_embedding, topic_embeddings)
best_topic = topics[cos_scores.argmax()]
print("Suggested Title:", best_topic)
```

```
Suggested Title: Transformer Models
```

Data Chunking

Summarization-Based Token Reduction

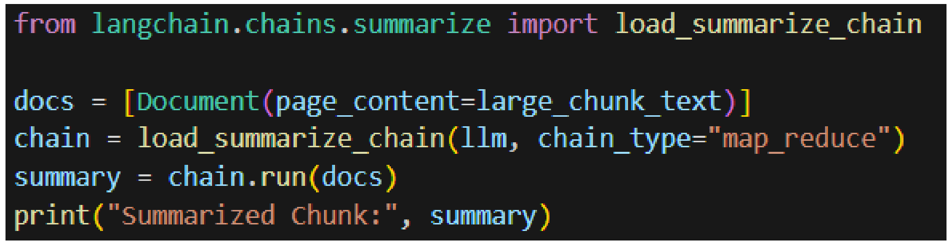
A technique where lengthy chunks are compressed by summarizing them while preserving key information, thus reducing the total token count.

How it works

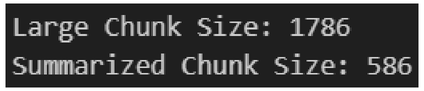
An LLM is used to generate a condensed version of the chunk without losing core context, ideal when you're close to token limits in downstream processing.

Benefits

- Makes room for more chunks in context window.
- Minimizes hallucination due to token truncation.
- Increases efficiency in retrieval and generation stages.



```python
from langchain.chains.summarize import load_summarize_chain

docs = [Document(page_content=large_chunk_text)]
chain = load_summarize_chain(llm, chain_type="map_reduce")
summary = chain.run(docs)
print("Summarized Chunk:", summary)
```



```
Large Chunk Size: 1786
Summarized Chunk Size: 586
```

Data Chunking

Importance-Aware Trimming

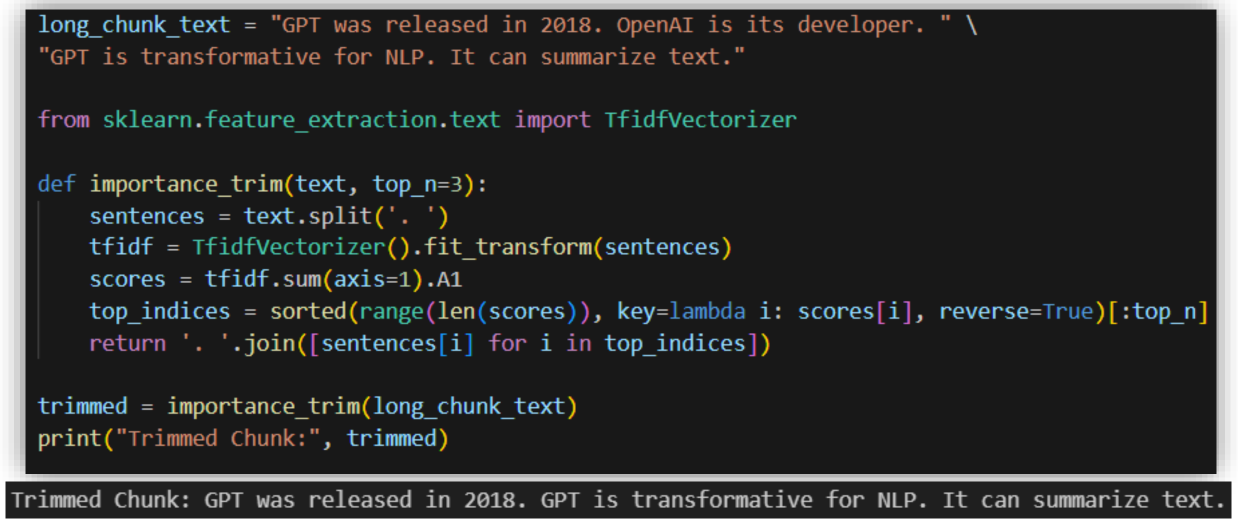
Selective trimming of less important sentences based on scoring techniques or LLM estimation of relevance within the chunk.

How it works

Every sentence in a chunk is scored using TF-IDF, embedding similarity, or LLM-based scoring, and only top-scoring sentences are retained.

Benefits

- Removes noise.
- Keeps only high-signal information.
- Reduces token overload in context windows.



```python
long_chunk_text = "GPT was released in 2018. OpenAI is its developer. " \
"GPT is transformative for NLP. It can summarize text."

from sklearn.feature_extraction.text import TfidfVectorizer

def importance_trim(text, top_n=3):
    sentences = text.split('. ')
    tfidf = TfidfVectorizer().fit_transform(sentences)
    scores = tfidf.sum(axis=1).A1
    top_indices = sorted(range(len(scores)), key=lambda i: scores[i], reverse=True)[:top_n]
    return '. '.join([sentences[i] for i in top_indices])

trimmed = importance_trim(long_chunk_text)
print("Trimmed Chunk:", trimmed)
```

```
Trimmed Chunk: GPT was released in 2018. GPT is transformative for NLP. It can summarize text.
```

Data Chunking

LLM-Assisted Compression

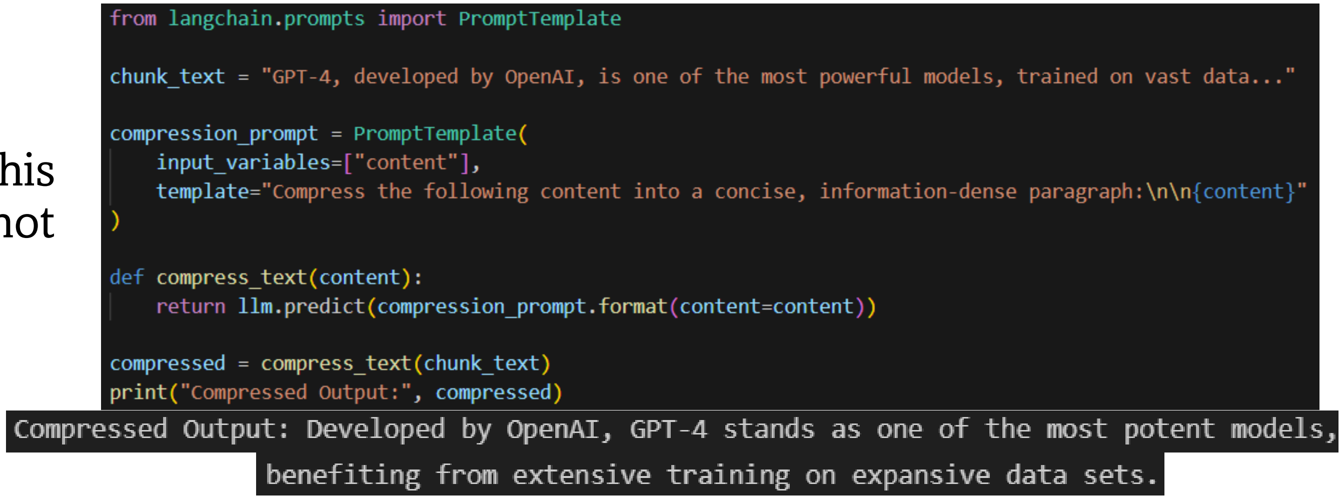
LLMs are used to rewrite the chunk in a compressed, information-dense manner—retaining semantic fidelity but with fewer tokens.

How it works

Unlike summarization, which may omit details, this rewrites the text in a concise style using zero-shot prompts (like “rewrite this more concisely”).

Benefits

- Increases context window utilization.
- Balances detail with brevity.
- Minimizes token usage while keeping informative density high.



```python
from langchain.prompts import PromptTemplate

chunk_text = "GPT-4, developed by OpenAI, is one of the most powerful models, trained on vast data..."

compression_prompt = PromptTemplate(
    input_variables=["content"],
    template="Compress the following content into a concise, information-dense paragraph:\n\n{content}"
)

def compress_text(content):
    return llm.predict(compression_prompt.format(content=content))

compressed = compress_text(chunk_text)
print("Compressed Output:", compressed)
```

```
Compressed Output: Developed by OpenAI, GPT-4 stands as one of the most potent models,
benefiting from extensive training on expansive data sets.
```

Storage and Retrieval

Storage and Retrieval

Dense Retrieval

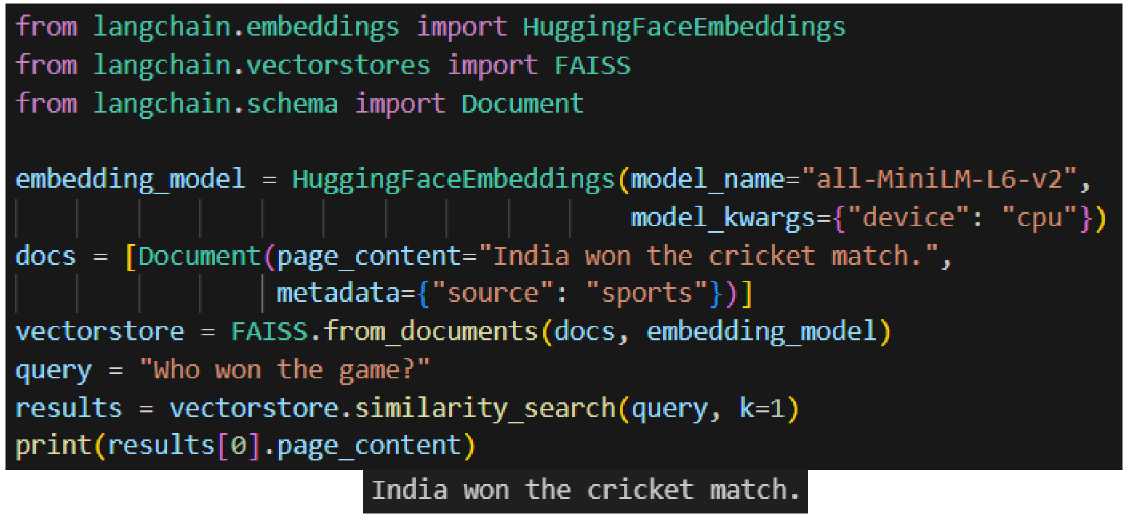
Dense retrieval uses dense vector embeddings generated by transformer models to retrieve documents based on semantic similarity rather than keyword overlap.

How it works

- Convert both documents and query into high-dimensional dense vectors using an embedding model (e.g., all-MiniLM-L6-v2).
- Perform vector similarity search (e.g., cosine similarity) to retrieve top-k relevant documents.

Benefits

Dense retrieval captures contextual meaning and improves accuracy for semantic queries where keywords may not match directly.



```python
from langchain.embeddings import HuggingFaceEmbeddings
from langchain.vectorstores import FAISS
from langchain.schema import Document

embedding_model = HuggingFaceEmbeddings(model_name="all-MiniLM-L6-v2",
                                        model_kwargs={"device": "cpu"})
docs = [Document(page_content="India won the cricket match.",
            metadata={"source": "sports"})]
vectorstore = FAISS.from_documents(docs, embedding_model)
query = "Who won the game?"
results = vectorstore.similarity_search(query, k=1)
print(results[0].page_content)
```

```
India won the cricket match.
```

Storage and Retrieval

Sparse Retrieval (Best Matching - BM25)

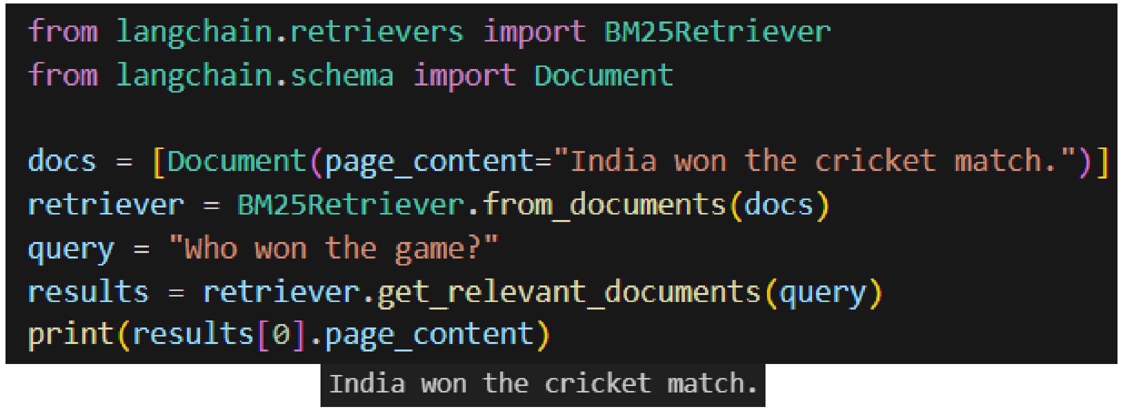
Sparse retrieval relies on traditional term-frequency-based methods like BM25 to rank documents based on keyword overlap.

How it works

- Tokenize documents and compute inverse document frequency (IDF).
- Score based on term frequency and document length normalization.

Benefits

BM25 provides high recall for keyword-based queries, making it useful when semantic models fail.



```python
from langchain.retrievers import BM25Retriever
from langchain.schema import Document

docs = [Document(page_content="India won the cricket match.")]
retriever = BM25Retriever.from_documents(docs)
query = "Who won the game?"
results = retriever.get_relevant_documents(query)
print(results[0].page_content)
```

```
India won the cricket match.
```

Storage and Retrieval

Combining Scores (Hybrid Retrieval)

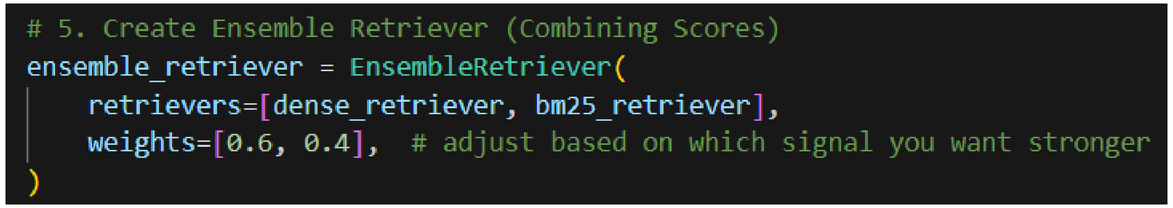
Combines both dense and sparse retrieval scores to balance semantic and keyword-based relevance.

How it works

- Retrieve top-k from both dense and sparse methods.
- Normalize and combine scores (e.g., weighted average).

Benefits

Improves robustness and coverage for diverse query formulations.



```
# 5. Create Ensemble Retriever (Combining Scores)
ensemble_retriever = EnsembleRetriever(
    retrievers=[dense_retriever, bm25_retriever],
    weights=[0.6, 0.4],  # adjust based on which signal you want stronger
)
```

Storage and Retrieval

Attribute-Based Filtering

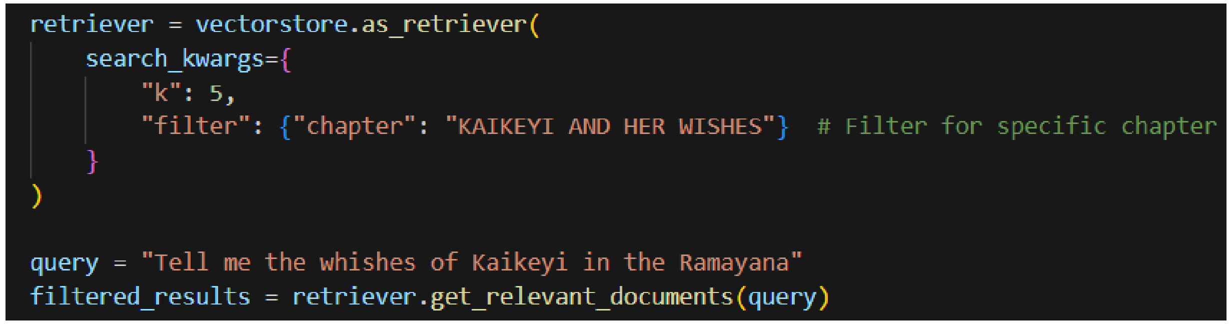
Filters retrieved documents based on metadata attributes like date, author, category, etc.

How it works

- Tag chunks with metadata during ingestion.
- Use query constraints to filter results before/after vector search.

Benefits

Enables targeted retrieval from relevant subsets, improving precision and reducing irrelevant context.



```python
retriever = vectorstore.as_retriever(
    search_kwargs={
        "k": 5,
        "filter": {"chapter": "KAIKEYI AND HER WISHES"}  # Filter for specific chapter
    }
)

query = "Tell me the whishes of Kaikeyi in the Ramayana"
filtered_results = retriever.get_relevant_documents(query)
```

Storage and Retrieval

Facet Tagging in Vector DBs

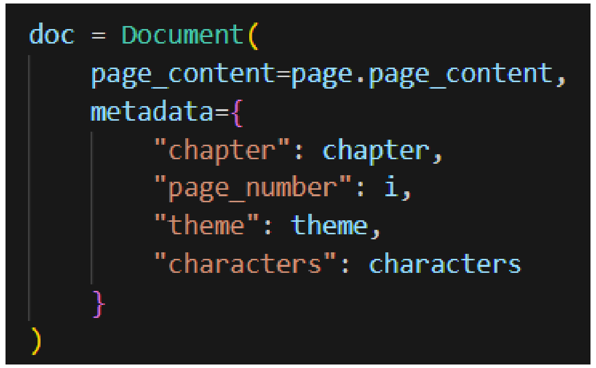
Faceting refers to categorizing documents by multiple tags (facets) like topic, date, type, etc.

How it works

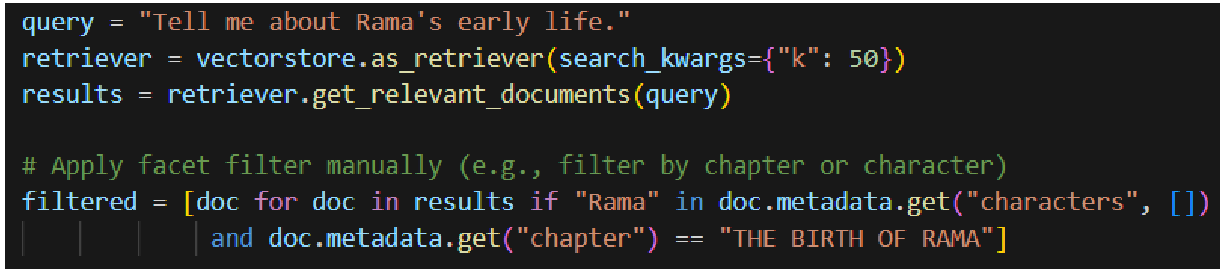
- During ingestion, annotate documents with multiple facet fields.
- Allow multi-faceted queries like: "topic=finance AND year=2023".

Benefits

Enhances retrieval granularity and personalization for complex queries.



```python
doc = Document(
    page_content=page.page_content,
    metadata={
        "chapter": chapter,
        "page_number": i,
        "theme": theme,
        "characters": characters
    }
)
```



```python
query = "Tell me about Rama's early life."
retriever = vectorstore.as_retriever(search_kwargs={"k": 50})
results = retriever.get_relevant_documents(query)

# Apply facet filter manually (e.g., filter by chapter or character)
filtered = [doc for doc in results if "Rama" in doc.metadata.get("characters", [])
            and doc.metadata.get("chapter") == "THE BIRTH OF RAMA"]
```

Storage and Retrieval

User-Defined Query Constraints

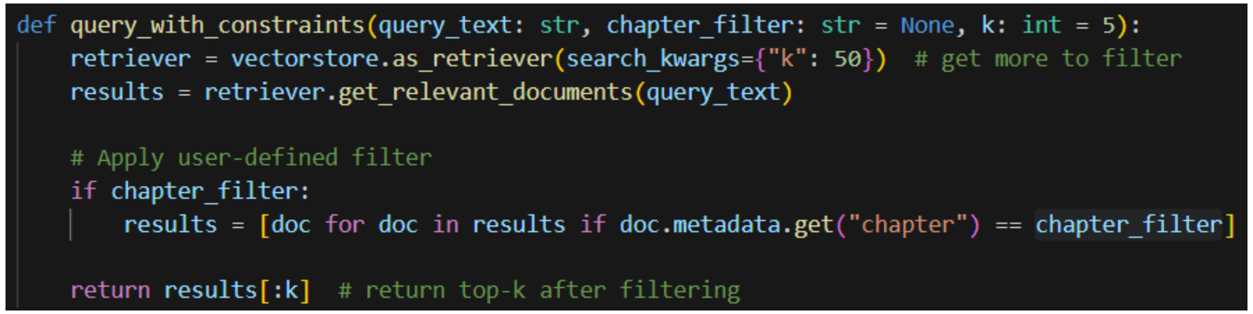
Allows users to specify retrieval constraints manually (e.g., only recent documents, author filters).

How it works

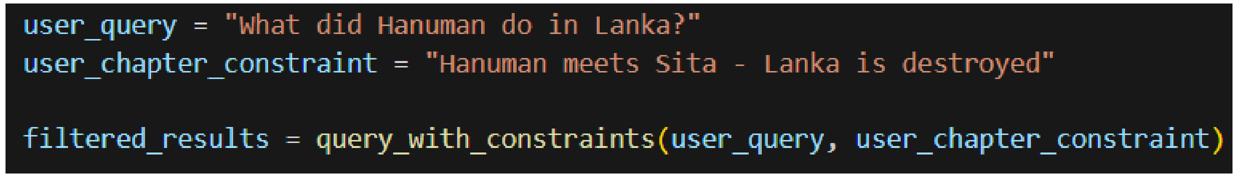
- Accept user-defined parameters along with the query.
- Apply these as filters on vectorstore search.

Benefits

Enables controllability, aligning retrieval with user intent.



```python
def query_with_constraints(query_text: str, chapter_filter: str = None, k: int = 5):
    retriever = vectorstore.as_retriever(search_kwargs={"k": 50})  # get more to filter
    results = retriever.get_relevant_documents(query_text)

    # Apply user-defined filter
    if chapter_filter:
        results = [doc for doc in results if doc.metadata.get("chapter") == chapter_filter]

    return results[:k]  # return top-k after filtering
```



```python
user_query = "What did Hanuman do in Lanka?"
user_chapter_constraint = "Hanuman meets Sita - Lanka is destroyed"

filtered_results = query_with_constraints(user_query, user_chapter_constraint)
```

Storage and Retrieval

Paragraph-Level Scoring

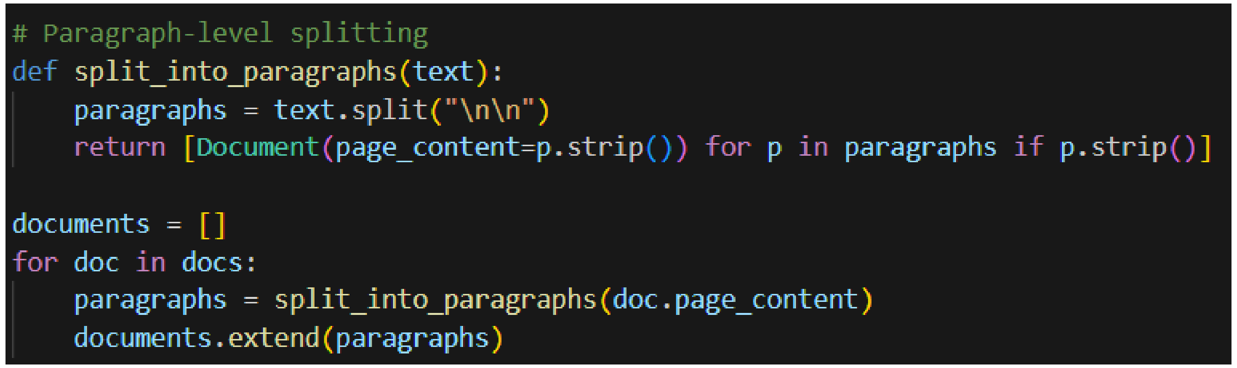
Scores individual paragraphs within a document rather than the entire document.

How it works

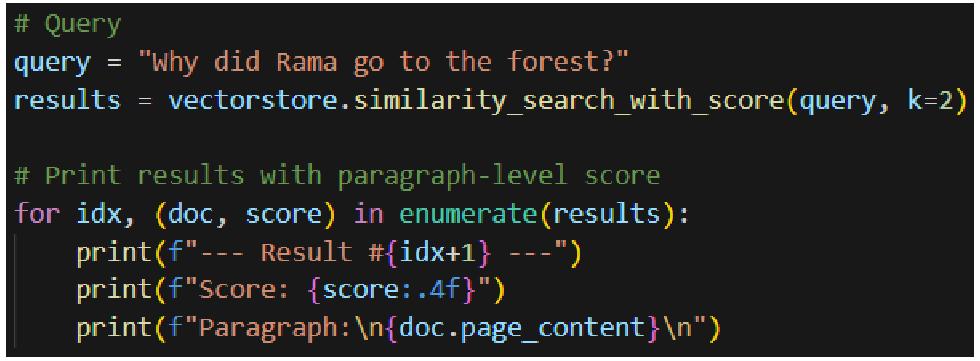
- Split documents into paragraphs.
- Retrieve and rank each paragraph independently.

Benefits

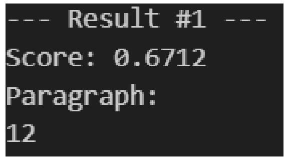
Increases answer granularity and relevance by retrieving only the most relevant parts.



```python
# Paragraph-level splitting
def split_into_paragraphs(text):
    paragraphs = text.split("\n\n")
    return [Document(page_content=p.strip()) for p in paragraphs if p.strip()]

documents = []
for doc in docs:
    paragraphs = split_into_paragraphs(doc.page_content)
    documents.extend(paragraphs)
```



```python
# Query
query = "Why did Rama go to the forest?"
results = vectorstore.similarity_search_with_score(query, k=2)

# Print results with paragraph-level score
for idx, (doc, score) in enumerate(results):
    print(f"--- Result #{idx+1} ---")
    print(f"Score: {score:.4f}")
    print(f"Paragraph:\n{doc.page_content}\n")
```



```
--- Result #1 ---
Score: 0.6712
Paragraph:
12
```

Prompt Engineering and LLM

Prompt Engineering and LLM

Reducing Hallucinations with Verification

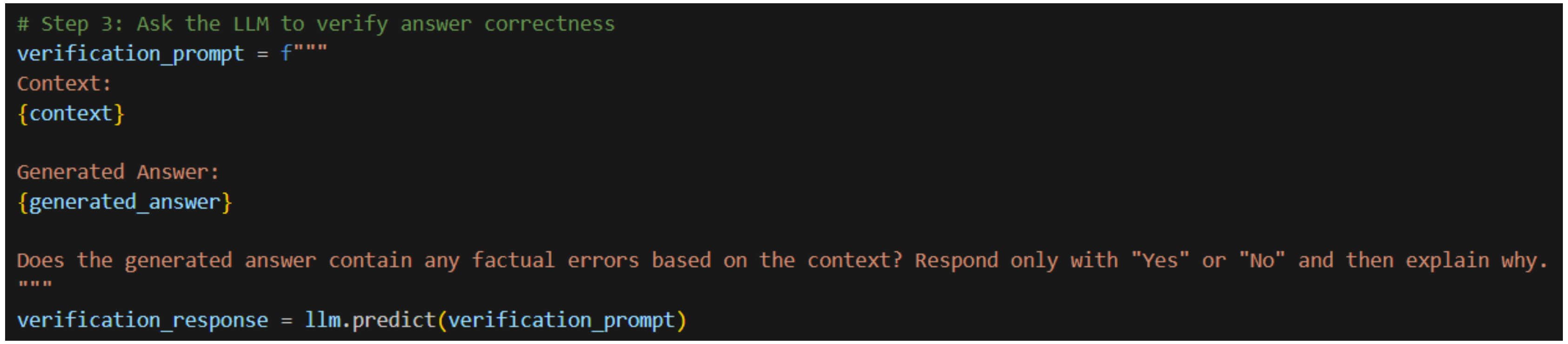
Reducing hallucinations means minimizing fabricated or unsupported responses from LLMs. Verification ensures that the LLM's answer is grounded in retrieved knowledge.

How it works

- Retrieved documents (contexts) are used as evidence.
- The generated answer is validated against the evidence.
- If the content doesn’t align, it's flagged or revised.

Benefits

- Boosts factual accuracy
- Reduces model hallucination
- Builds user trust



```
# Step 3: Ask the LLM to verify answer correctness
verification_prompt = f"""
Context:
{context}

Generated Answer:
{generated_answer}

Does the generated answer contain any factual errors based on the context? Respond only with "Yes" or "No" and then explain why.
"""
verification_response = llm.predict(verification_prompt)
```

Prompt Engineering and LLM

Confidence Scoring via LLMs

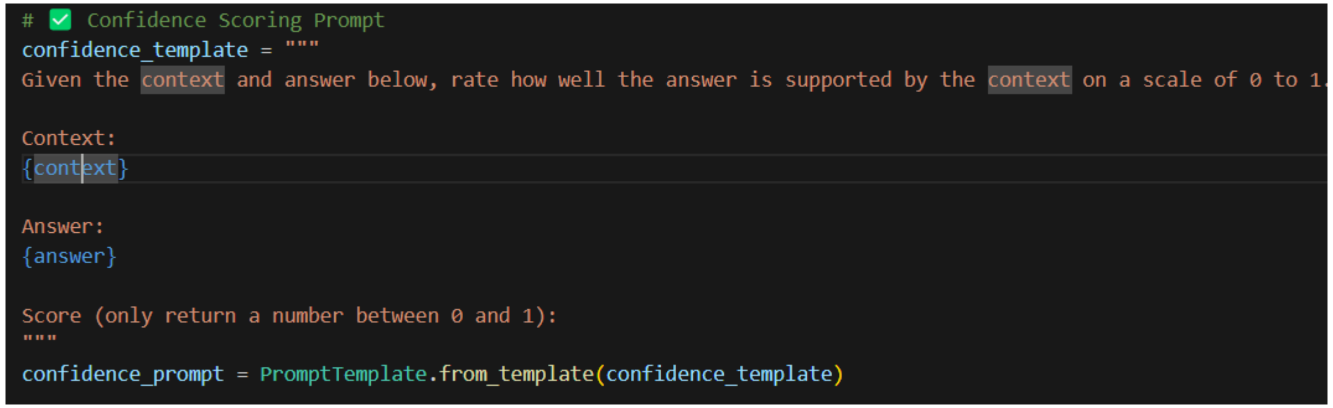
Assigning a confidence score to an LLM's response to estimate its reliability.

How it works

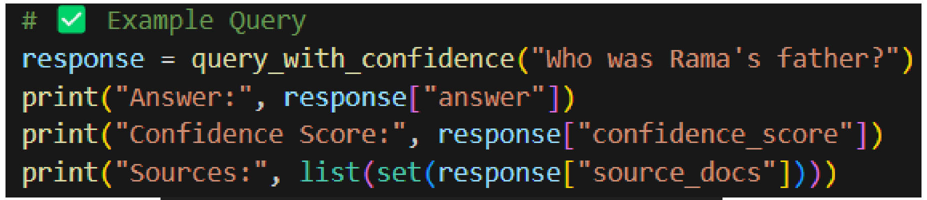
- A secondary LLM evaluates how confident the model is based on answer-context alignment.
- Output is a score (0 to 1 or percentage).

Benefits

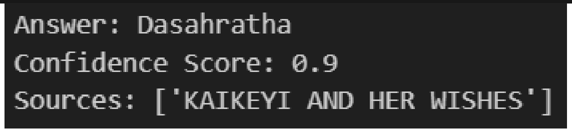
- Provides quality signals to downstream tasks.
- Enables answer filtering



```
# ✅ Confidence Scoring Prompt
confidence_template = """
Given the context and answer below, rate how well the answer is supported by the context on a scale of 0 to 1.

Context:
{context}

Answer:
{answer}

Score (only return a number between 0 and 1):
"""
confidence_prompt = PromptTemplate.from_template(confidence_template)
```



```
# ✅ Example Query
response = query_with_confidence("Who was Rama's father?")
print("Answer:", response["answer"])
print("Confidence Score:", response["confidence_score"])
print("Sources:", list(set(response["source_docs"])))
```



```
Answer: Dasahratha
Confidence Score: 0.9
Sources: ['KAIKEYI AND HER WISHES']
```

Prompt Engineering and LLM

Audit Trails for Traceability

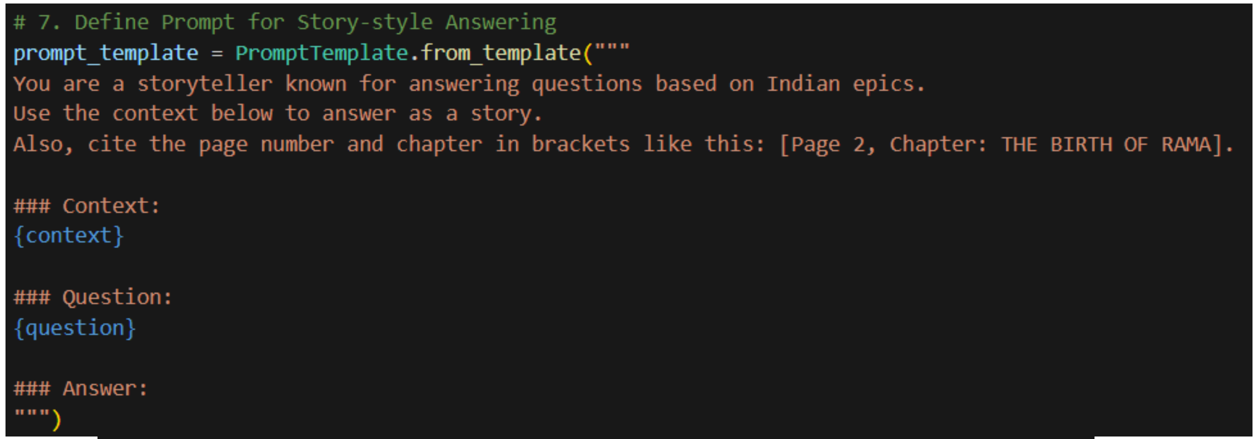
Maintaining a record of retrieved documents, generation steps, and decisions.

How it works

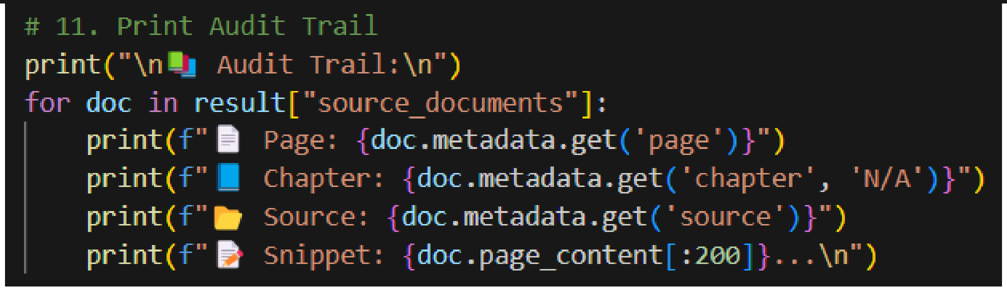
- Store user query, retrieved contexts, model prompts, final answer.
- Enable reviewing for debugging or compliance.

Benefits

- Enables root cause analysis.
- Essential for regulated domains.



```
# 7. Define Prompt for Story-style Answering
prompt_template = PromptTemplate.from_template("""
You are a storyteller known for answering questions based on Indian epics.
Use the context below to answer as a story.
Also, cite the page number and chapter in brackets like this: [Page 2, Chapter: THE BIRTH OF RAMA].

### Context:
{context}

### Question:
{question}

### Answer:
""")
```



```
# 11. Print Audit Trail
print("\n📚 Audit Trail:\n")
for doc in result["source_documents"]:
    print(f"📄 Page: {doc.metadata.get('page')}")
    print(f"📘 Chapter: {doc.metadata.get('chapter', 'N/A')}")
    print(f"📁 Source: {doc.metadata.get('source')}")
    print(f"📝 Snippet: {doc.page_content[:200]}...\n")
```

Prompt Engineering and LLM

Intelligent Reranking

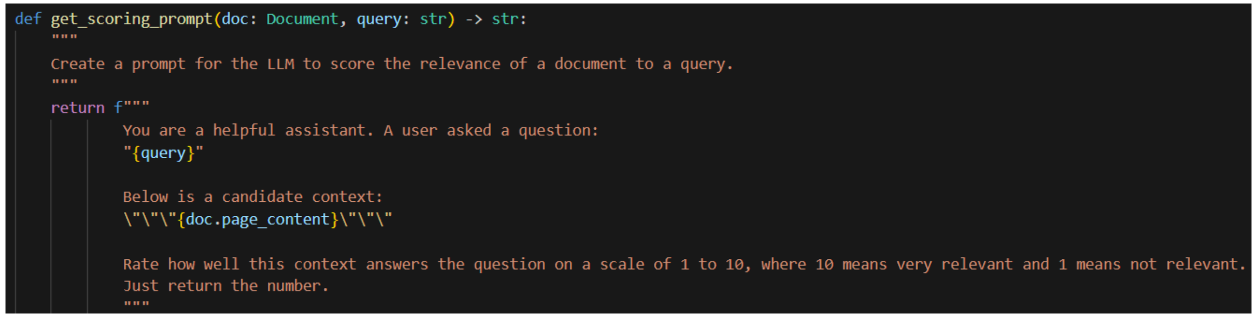
Reordering retrieved documents based on relevance using learned signals.

How it works

- Initial top-k retrieved chunks are reranked using a cross-encoder or scoring LLM.
- Most relevant chunk moved to top.

Benefits

- Enhances relevance and answer quality.



```python
def get_scoring_prompt(doc: Document, query: str) -> str:
    """
    Create a prompt for the LLM to score the relevance of a document to a query.
    """
    return f"""
            You are a helpful assistant. A user asked a question:
            "{query}"

            Below is a candidate context:
            \"\"\"{doc.page_content}\"\"\"

            Rate how well this context answers the question on a scale of 1 to 10, where 10 means very relevant and 1 means not relevant.
            Just return the number.
            """
```

Prompt Engineering and LLM

LLM-Based Scoring

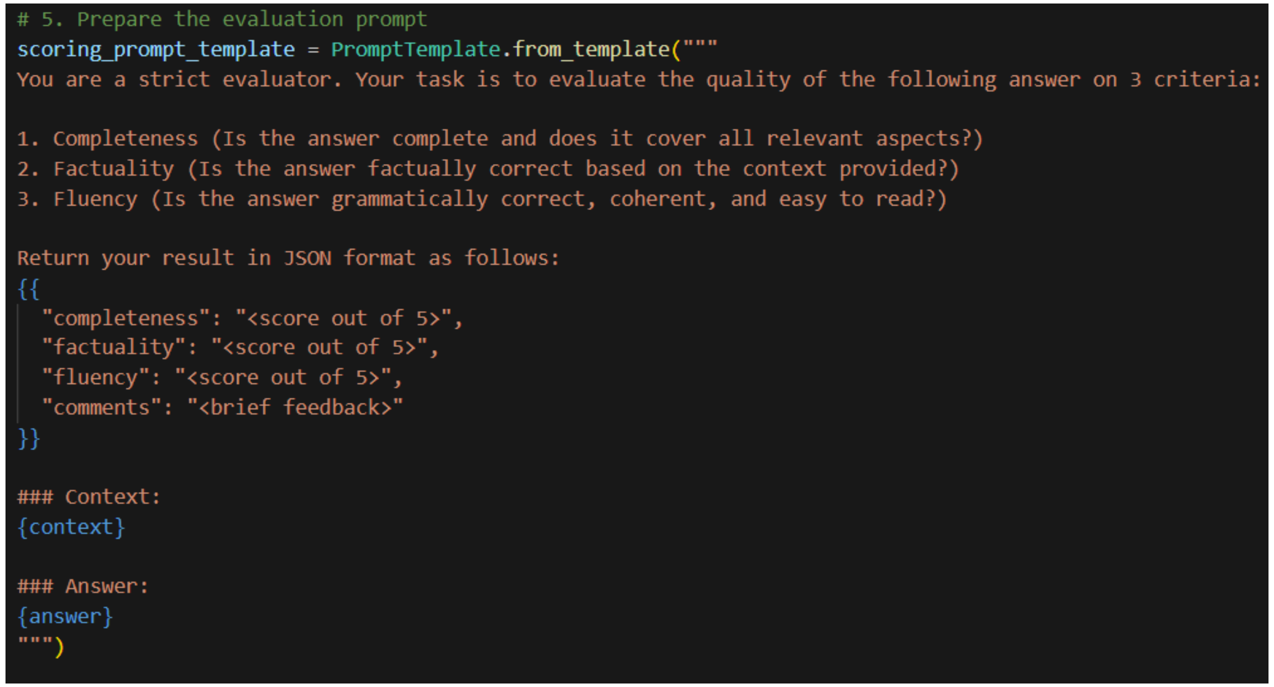
Using an LLM to score answers on various criteria like completeness, factuality, fluency.

How it works

- LLM is prompted to evaluate answer and return scores.

Benefits

- Quality assurance loop for generation.



```
# 5. Prepare the evaluation prompt
scoring_prompt_template = PromptTemplate.from_template("""
You are a strict evaluator. Your task is to evaluate the quality of the following answer on 3 criteria:

1. Completeness (Is the answer complete and does it cover all relevant aspects?)
2. Factuality (Is the answer factually correct based on the context provided?)
3. Fluency (Is the answer grammatically correct, coherent, and easy to read?)

Return your result in JSON format as follows:
{{
  "completeness": "<score out of 5>",
  "factuality": "<score out of 5>",
  "fluency": "<score out of 5>",
  "comments": "<brief feedback>"
}}

### Context:
{context}

### Answer:
{answer}
""")
```

```
=== LLM-Based Scoring ===
{ "completeness": "3 out of 5",
 "factuality": "5 out of 5",
 "fluency": "5 out of 5",
 "comments": "The answer is factually correct and fluently written, but it does not cover all
aspects of the context. It does not mention Rama's mother's sadness, Bharatha's rule over
Ayodhya, or Rama's decision to leave Chitrkut." }
```

Prompt Engineering and LLM

Use of RAGAS Metrics

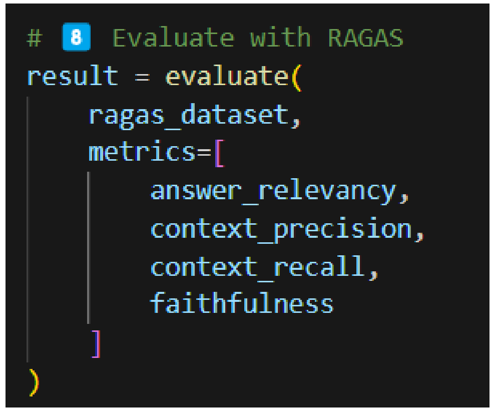
RAGAS (RAG Assessment System) is a framework for evaluating RAG systems using retrieval and generation scores.

How it works

- Measures answer correctness, faithfulness, context precision, etc.
- Uses pre-built RAGAS pipelines.

Benefits

- Quantifies end-to-end system performance.
- Helps optimize pipeline components.



```
# 8 Evaluate with RAGAS
result = evaluate(
    ragas_dataset,
    metrics=[
        answer_relevancy,
        context_precision,
        context_recall,
        faithfulness
    ]
)
```

Prompt Engineering and LLM

Ground Truth-Based Scoring

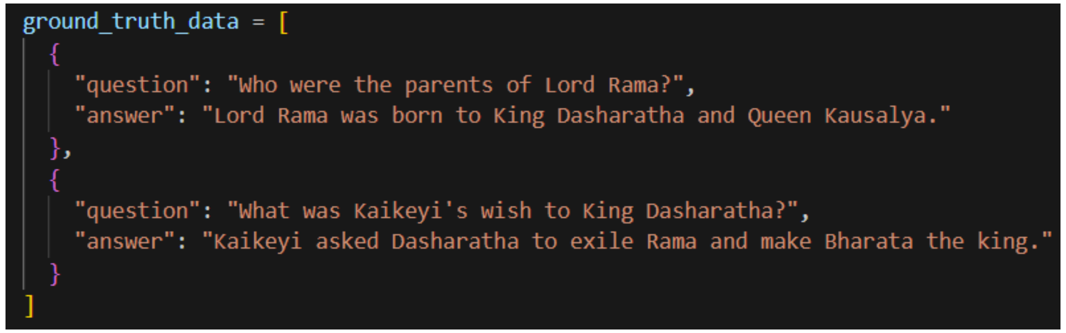
Compare generated answer to known ground-truth answers to assess performance.

How it works

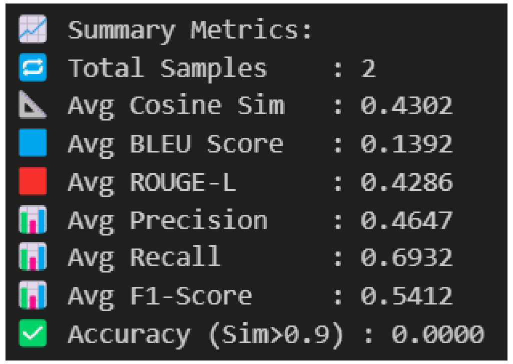
- Use gold standard QA pairs.
- Compute accuracy, BLEU, ROUGE, F1.

Benefits

- Enables objective evaluation.
- Supports model comparison.



```
ground_truth_data = [
  {
    "question": "Who were the parents of Lord Rama?",
    "answer": "Lord Rama was born to King Dasharatha and Queen Kausalya."
  },
  {
    "question": "What was Kaikeyi's wish to King Dasharatha?",
    "answer": "Kaikeyi asked Dasharatha to exile Rama and make Bharata the king."
  }
]
```



```
📈 Summary Metrics:
🔁 Total Samples     : 2
📐 Avg Cosine Sim    : 0.4302
🟦 Avg BLEU Score    : 0.1392
🟥 Avg ROUGE-L       : 0.4286
📊 Avg Precision     : 0.4647
📊 Avg Recall        : 0.6932
📊 Avg F1-Score      : 0.5412
✅ Accuracy (Sim>0.9) : 0.0000
```

Prompt Engineering and LLM

Large Language Models Evaluation

BLEU (Bilingual Evaluation Understudy) – *compares n-gram overlaps between prediction and reference*

$$\text{BLEU} = \text{BP} \cdot \exp\left(\sum_{n=1}^{N} w_n \log p_n\right).$$

Papineni, K., Roukos, S., Ward, T., & Zhu, W. J. (2002).

Let's calculate it for bigram

$$\text{BLEU-2} = \text{BP} \cdot \exp\left(\frac{1}{2}(\log p_1 + \log p_2)\right)$$

Actual: The cat is on the mat

Unigram: the, cat, is, on, the, mat

Bigram: the cat, cat is, is on, on the, the mat

Predicted: The cat sat on the mat $p1 = \frac{5}{6}$

Unigram: the, cat, sat, on, the, mat

Bigram: the cat, cat sat, sat on, on the, the mat $p2 = \frac{3}{6}$

$$BLEU - 2 = 1 * e^{\left(\frac{1}{2}\left(log\frac{5}{6} + log\frac{3}{6}\right)\right)}$$

$$\boxed{0.645}$$

BP stands for **Brevity Penalty**. It is used to penalize machine-generated text that is **too short** compared to the reference

BLEU score is precision-oriented (counts how many n-grams match), but without a length penalty, a model could **cheat** by just outputting short sequences.

BP solves this by lowering the BLEU score when the generated output is shorter than the reference.

$$BP = \begin{cases} 1 & \text{if } c > r \\ e^{(1-\frac{r}{c})} & \text{if } c \leq r \end{cases}$$

Prompt Engineering and LLM

Large Language Models Evaluation

ROUGE (Recall-Oriented Understudy for Gisting Evaluation) Score

Lin, C.-Y. (2004).
ROUGE: A Package for Automatic Evaluation of Summaries

ROUGE (Recall-Oriented) is used in summarization. The most common are:

- **ROUGE-1**: Overlap of unigrams

$$\textbf{ROUGE-1(Precision)} = 5/6,\ \textbf{ROUGE-1(Recall)} = 5/6,\ \textbf{ROUGE-1(F1)} = \frac{2 * \frac{5}{6} * \frac{5}{6}}{\frac{5}{6} + \frac{5}{6}}$$

- **ROUGE-2**: Overlap of bigrams

$$\textbf{ROUGE-2(Precision)} = 3/6,\ \textbf{ROUGE-1(Recall)} = 3/6,\ \textbf{ROUGE-1(F1)} = \frac{2 * \frac{3}{6} * \frac{3}{6}}{\frac{3}{6} + \frac{3}{6}}$$

- **ROUGE-L**: Longest Common Subsequence (LCS)

  Longest Sequence (5) = The cat [mismatch/gap] on the mat

$$\textbf{ROUGE-L(Precision)} = 5/6,\ \textbf{ROUGE-1(Recall)} = 5/6,\ \textbf{ROUGE-1(F1)} = \frac{2 * \frac{5}{6} * \frac{5}{6}}{\frac{5}{6} + \frac{5}{6}}$$

Prompt Engineering and LLM

Human-in-the-Loop Validation

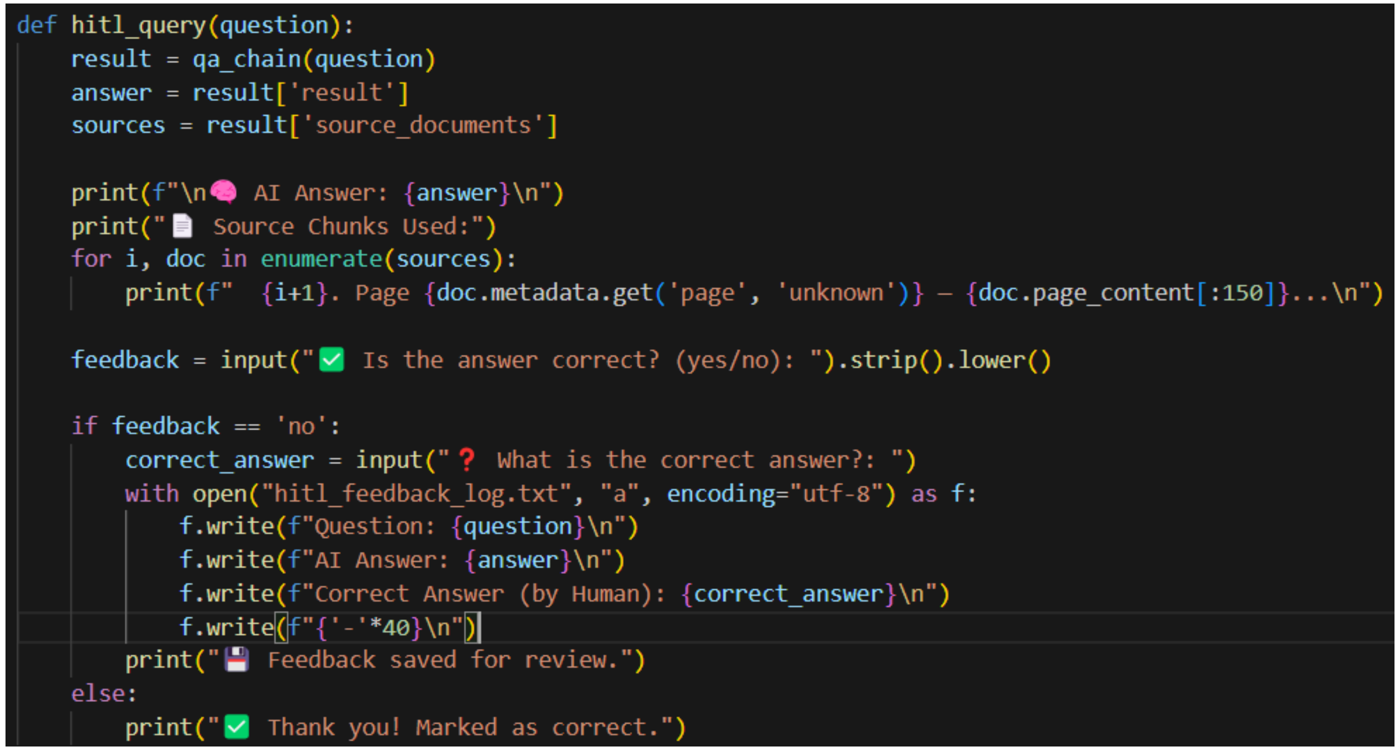
Involves humans reviewing or correcting model outputs.

How it works

- Experts validate model answers.
- Feedback loop refines prompts or retriever config.

Benefits

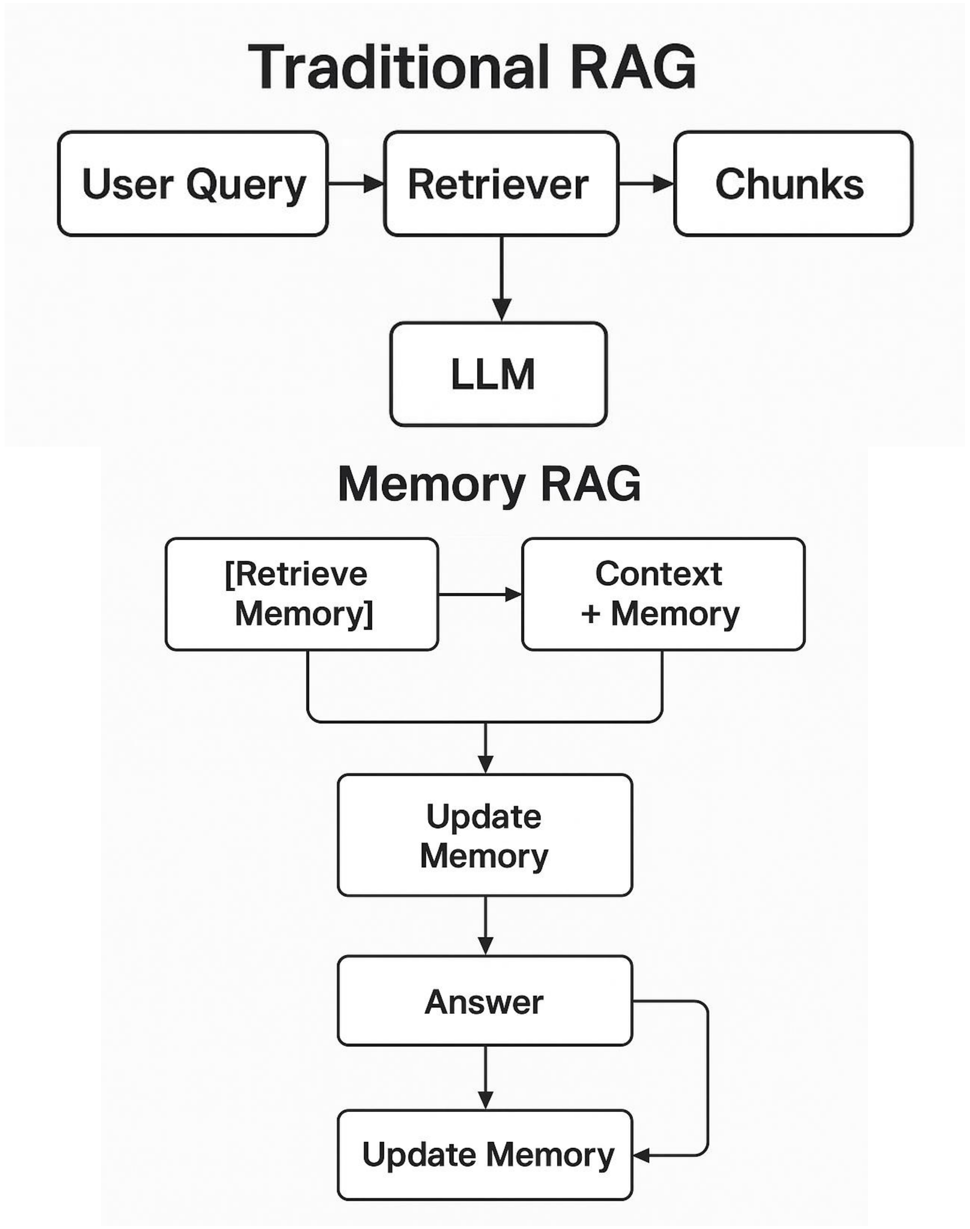
- Improves accuracy via human review.
- Enables continuous learning.



```python
def hitl_query(question):
    result = qa_chain(question)
    answer = result['result']
    sources = result['source_documents']

    print(f"\n🧠 AI Answer: {answer}\n")
    print("📄 Source Chunks Used:")
    for i, doc in enumerate(sources):
        print(f"  {i+1}. Page {doc.metadata.get('page', 'unknown')} – {doc.page_content[:150]}...\n")

    feedback = input("✅ Is the answer correct? (yes/no): ").strip().lower()

    if feedback == 'no':
        correct_answer = input("❓ What is the correct answer?: ")
        with open("hitl_feedback_log.txt", "a", encoding="utf-8") as f:
            f.write(f"Question: {question}\n")
            f.write(f"AI Answer: {answer}\n")
            f.write(f"Correct Answer (by Human): {correct_answer}\n")
            f.write(f"{'-'*40}\n")
        print("💾 Feedback saved for review.")
    else:
        print("✅ Thank you! Marked as correct.")
```

Memory RAG

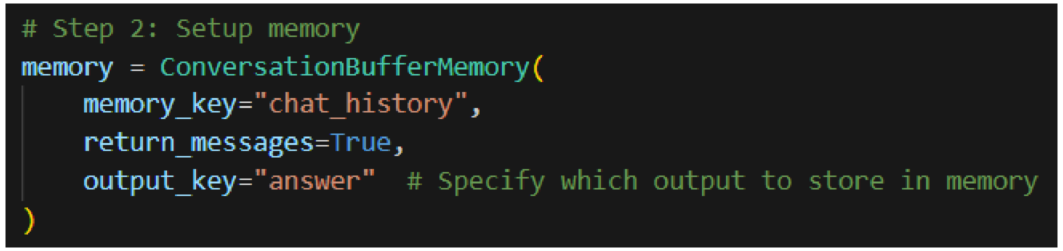
- Memory-augmented Retrieval-Augmented Generation (Memory RAG) extends the RAG pipeline by **incorporating a long-term or short-term memory component into the traditional RAG framework**.
- While standard RAG retrieves chunks from static knowledge bases, **Memory RAG stores conversational history or episodic facts and uses them in future queries**—allowing for more contextual, personalized, and stateful interactions.



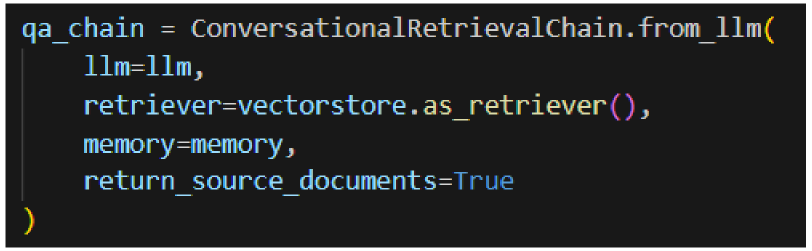
Memory RAG

Benefits

- **Memory Store:** Holds previous interactions or important context (e.g., LangChain’s ConversationBufferMemory, Redis, or Chroma DB).
- **Retriever:** Fetches relevant documents based on the query.
- **Fusion:** Merges memory context and retrieved documents.
- **LLM Response:** Generates an answer grounded in both memory and retrieved facts.
- **Memory Update:** Adds new context (question + answer) to the memory.



```python
# Step 2: Setup memory
memory = ConversationBufferMemory(
    memory_key="chat_history",
    return_messages=True,
    output_key="answer"  # Specify which output to store in memory
)
```



```python
qa_chain = ConversationalRetrievalChain.from_llm(
    llm=llm,
    retriever=vectorstore.as_retriever(),
    memory=memory,
    return_source_documents=True
)
```

Thanks for your time