Understanding and Building RAG Applications using Langchain

Dr. Shailesh Sivan

December 12, 2025

Outline

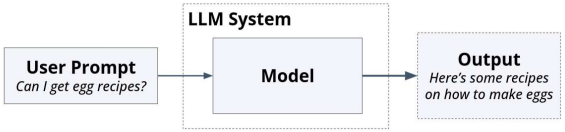
Disclaimer

- Some content and images used in this presentation are sourced from the Internet and publicly available resources.
- Certain materials (text, diagrams, and examples) are generated using AI tools.
- Some content is originally created by the presenter.
- All third-party materials are used as-is from the public domain or open-access sources.
- Full credit and acknowledgment go to the respective original creators and contributors.

This presentation is intended solely for educational and academic purposes.

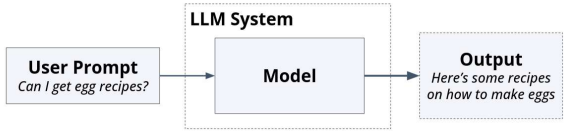
Motivation

LLMs and Domain Knowledge



- **General LLMs lack domain-specific knowledge**
  - Models are trained on broad internet text.
  - Missing specialized vocabulary, rules, and workflows.
- **Improves accuracy in specialized tasks**
  - Finance, healthcare, law, engineering, scientific domains.
  - Reduces wrong assumptions and hallucinations.
- **Aligns model with domain style and terminology**
  - Domain writing patterns, abbreviations, formulas.

LLMs and Domain Knowledge



- **Enhances reliability and safety**
  - Avoids unsafe suggestions in medical/technical domains.
  - Ensures outputs follow domain constraints.
- **Necessary for real-world enterprise use**
  - Internal documents, proprietary workflows.
  - Domain-specific decision support.
- **Reduces cost of over-prompting**
  - Less need to force context through long prompts.

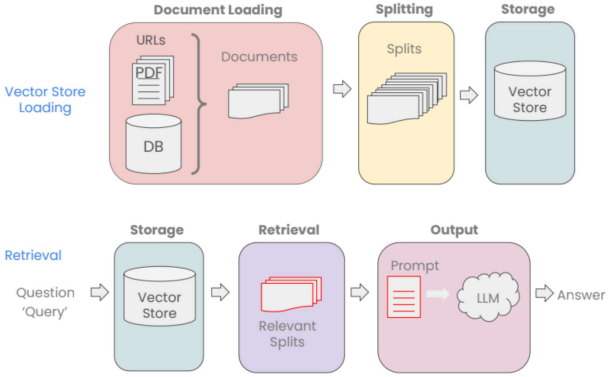
Domain Adaptation Techniques for LLMs

Key Techniques

- **1. Prompt Engineering**
  - Inject domain context via carefully designed prompts.
  - Zero training cost, quick adaptation.
- **2. Pre Traning**
  - Initial stage where an LLM learns general language patterns.
  - Trained on massive text corpora: web pages, books, articles.
- **3. Fine-Tuning**
  - Train the model further on labeled domain datasets.
  - Improves reasoning and domain correctness.
- **4. Retrieval-Augmented Generation (RAG)**
  - Combine LLM with domain documents retrieved at runtime.
  - Avoids hallucinations; supports dynamic updates.

Why RAG and Agentic AI?

- LLMs are powerful but limited: hallucination, context window, stale knowledge.
- RAG (Retrieval-Augmented Generation) grounds generation using retrieved documents.
- Retrieve relevant documents and load into `"working memory"` / context window.



LangChain Fundamentals

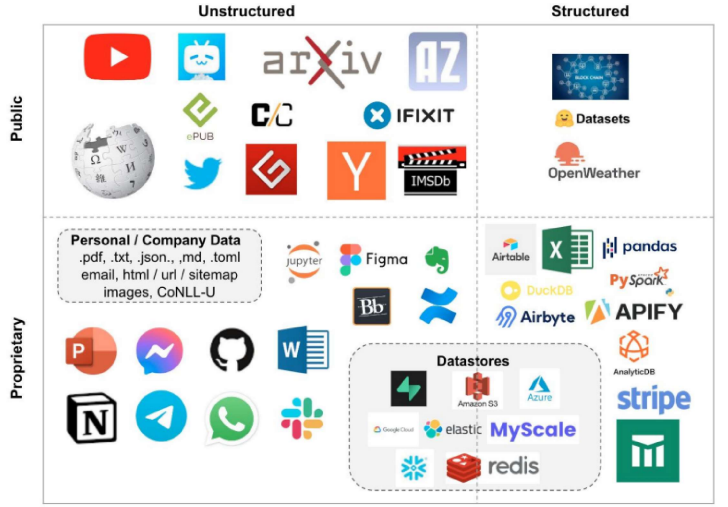
LangChain : Primitives

LangChain provides modular building blocks: **Loaders**, **Splitters**, **Embeddings**, **VectorStores**, **Retrievers**, **Chains**, **Agents**.

Loaders

- **Loaders** deal with the specifics of accessing and converting data
  - **Accessing**
    - Web Sites
    - Data Bases
    - YouTube
    - arXiv
    - ...
  - **Data Types**
    - PDF
    - HTML
    - JSON
    - Word, PowerPoint...
- Returns a list of `Document` objects:

Document Loaders

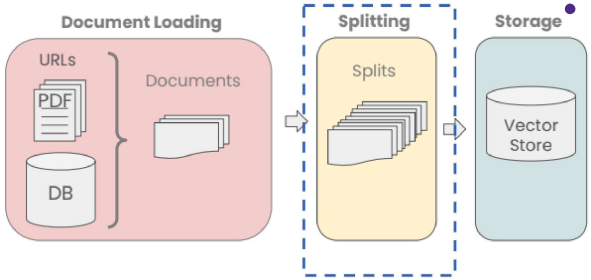


Document Loading (Python)

```python
# Load PDFs -> Document objects
from langchain_community.document_loaders import PyPDFLoader

loader = PyPDFLoader("/content/LoveStories.pdf")
docs = loader.load()
print(len(docs), docs[0].metadata)
```


Document Splitting

- Splitting retains meaningful relationships within text while producing chunks that fit LLM context windows .i.e splitting documents into smaller chunks

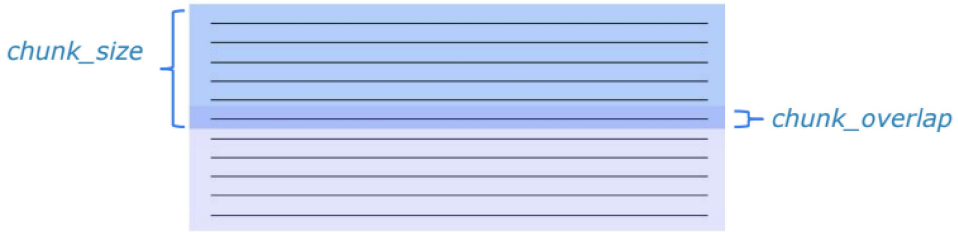


- ...on this model. The Toyota Camry has a head-snapping 80 HP and an eight-speed automatic transmission that will
  - Chunk 1: on this model. The Toyota Camry has a head-snapping
  - Chunk 2: 80 HP and an eight-speed automatic transmission that will

Question: What are the specifications on the Camry?

Splitter Example (Python)

```python
from langchain_text_splitters import RecursiveCharacterTextSplitter

splter = RecursiveCharacterTextSplitter(chunk_size=1000, chunk_overlap=200)
chunks = splter.split_documents(docs)

print("chunks:", len(chunks))
print("first chunk content preview:", chunks[0].page_content[:300])
```



Types of Text Splitters

Classes within `langchain.text_splitter`:

- `CharacterTextSplitter()` - Implementation of splitting text that looks at characters.
- `MarkdownHeaderTextSplitter()`- Implementation of splitting markdown files based on specified headers.
- `TokenTextSplitter()` - Implementation of splitting text that looks at tokens.
- `SentenceTransformersTokenTextSplitter()` - Implementation of splitting text that looks at tokens.
- `RecursiveCharacterTextSplitter()` - Implementation of splitting text that looks at characters. Recursively tries to split by different characters to find one that works.
- `Language()` for CPP, Python, Ruby, Markdown etc
- `NLTKTextSplitter()` - Implementation of splitting text that looks at sentences using NLTK (Natural Language Tool Kit)

Embeddings & Vector Stores

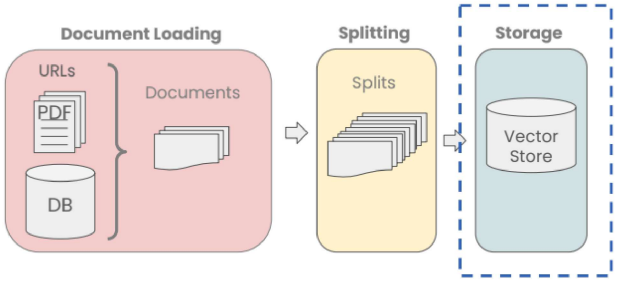
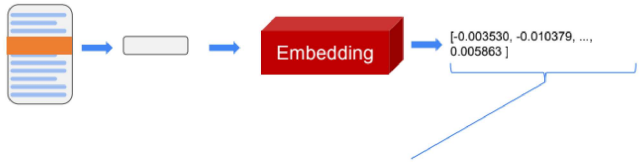


Figure: Vector Stores

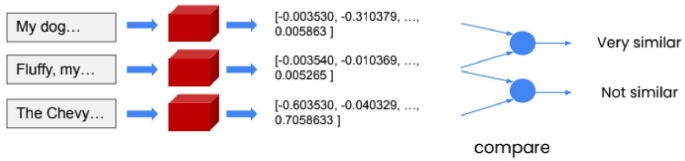
- Convert chunks into embeddings and store in vector DBs (FAISS, Chroma, Pinecone, Weaviate).
- Include metadata for filtering and LLM-aided retrieval.

Embeddings

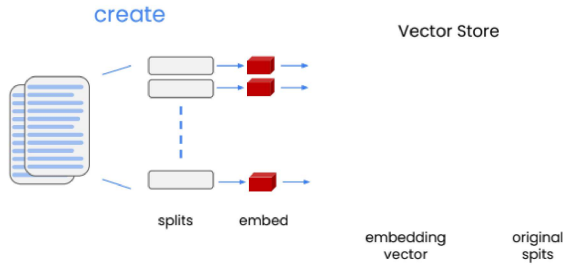


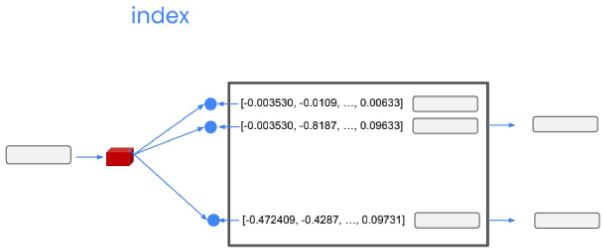
- Embedding vector captures content/meaning
- Text with similar content will have similar vectors

1) My dog Rover likes to chase squirrels.
2) Fluffy, my cat, refuses to eat from a can.
3) The Chevy Bolt accelerates to 60 mph in 6.7 seconds.

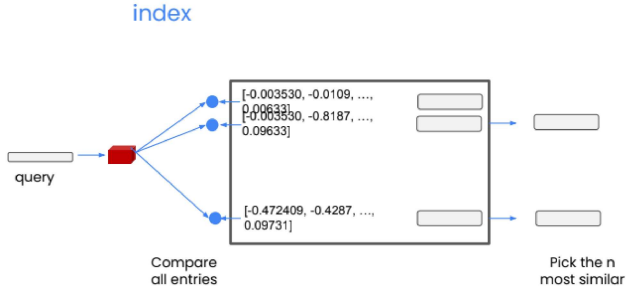


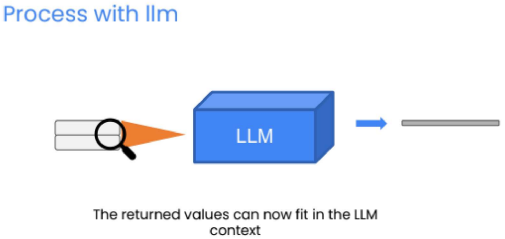
Vector Stores





Vector Stores/Databases



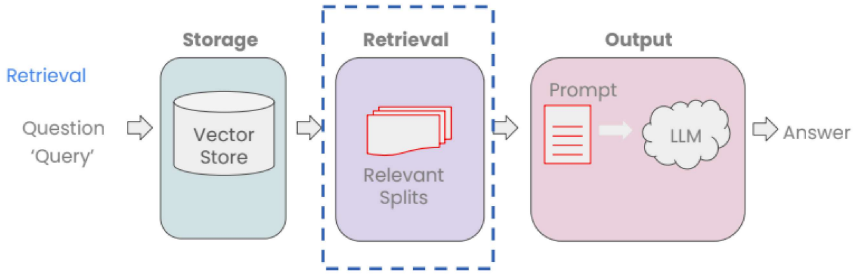


Create & Query Vector Store

```
from langchain_community.embeddings import HuggingFaceEmbeddings
from langchain_community.vectorstores import FAISS

# Choose any HuggingFace embedding model
emb = HuggingFaceEmbeddings(model_name="sentence-transformers/all-MiniLM-L6
    -v2")

# Create FAISS index from your chunks
index = FAISS.from_documents(chunks, emb)

retriever = index.as_retriever(search_kwargs={"k": 5})
results_with_scores = index.similarity_search_with_score("A STRANGE TWIST
    OF FATE ", k=5)
for doc, score in results_with_scores:
   print(doc.metadata.get("source"), score)
```


Retrieval Strategies

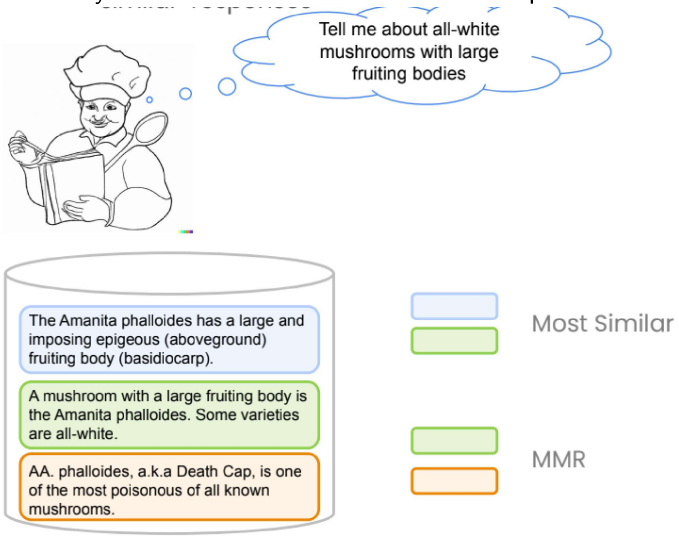
Retrieval modes



- Accessing/indexing the data in the vector store
- Modes:
  - Semantic similarity (embedding cosine)
  - Maximum Marginal Relevance (MMR) for diversity
  - LLM-aided retrieval / Self-query for structured filters
  - Compression LLM to condense retrieved splits for LLM context

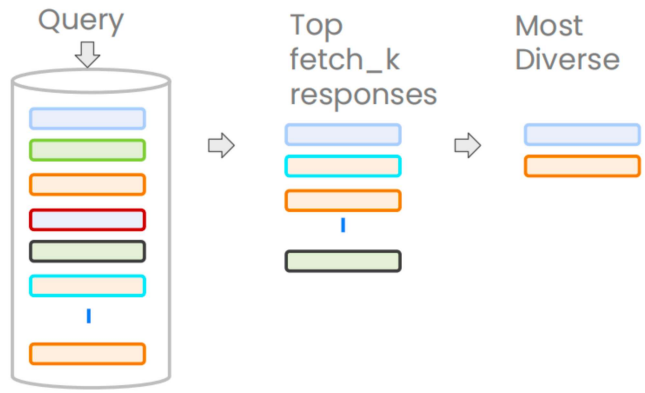
Maximum marginal relevance (MMR)

- You may not always want to choose the most similar responses



Maximum marginal relevance (MMR)

- Query the Vector Store
- Choose the fetch k most similar responses
- Within those responses choose the k most diverse

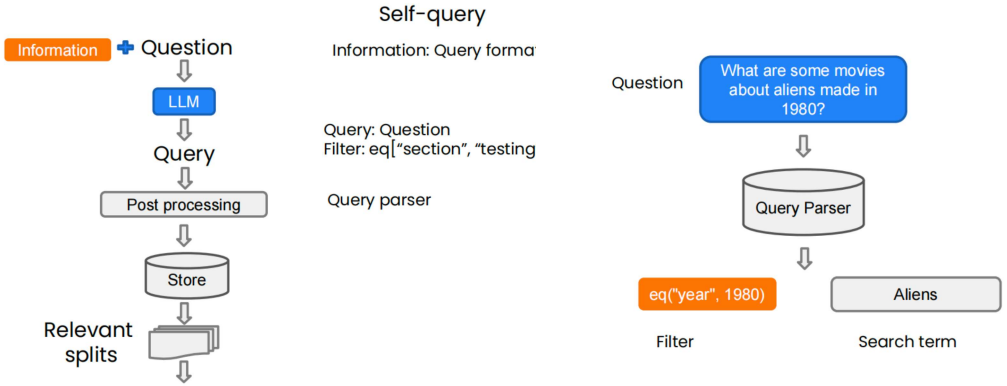


MMR Sketch (Python-style pseudocode)

```
# Pseudo-code: select k items that maximize relevance and diversity
def mmr_select(query_vec, candidates, k, lambda_param=0.5):
    selected = []
    while len(selected) < k:
        best = None
        best_score = -1e9
        for c in candidates:
            relevance = cosine_sim(query_vec, c.vec)
            diversity_penalty = max(cosine_sim(c.vec, s.vec) for s in
                selected) if selected else 0
            score = relevance - lambda_param * diversity_penalty
            if score > best_score:
                best_score, best = score, c
        selected.append(best)
    return selected
```


LLM-aided Retrieval (Self-query)

- Use an LLM to transform a free-form question into a query spec (filters + keywords) to apply to a vector store with metadata support

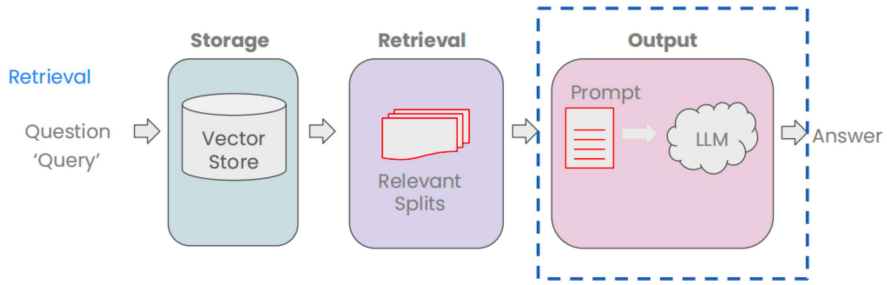


Self-query Example

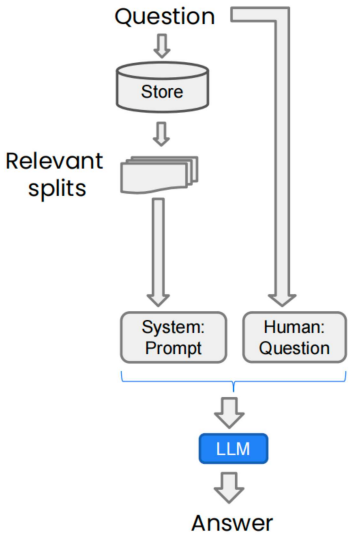
```
from langchain_google_genai import ChatGoogleGenerativeAI
import os

os.environ["GOOGLE_API_KEY"] = "YOUR_GOOGLE_API_KEY"

llm = ChatGoogleGenerativeAI(model="gemini-2.0-flash",temperature=0,)
prompt = "Create a JSON query for: 'movies about aliens in 1980'"
response = llm.invoke(prompt)
if isinstance(response.content, str) and response.content.strip():
    query_spec = response.content
else:
    query_spec = response.response_metadata["candidate_texts"][0]
print(query_spec)
```

RAG: Retrieval Augumented Generation

Question Answering



RetrievalQA Chain



Question is applied to the Vector Store as a query

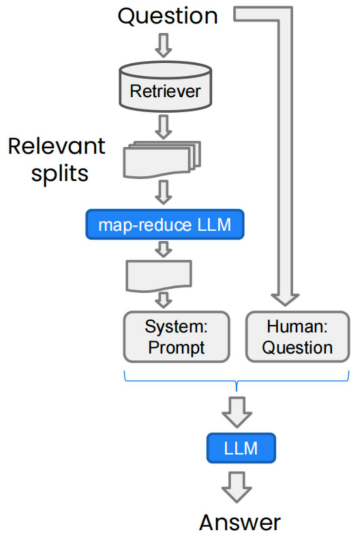
Vector store provides k relevant documents

Docs and original question are sent to an LLM

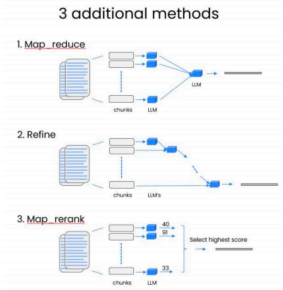
RetrievalQA chain patterns

```python
from langchain_classic.chains import RetrievalQA
from langchain_google_genai import ChatGoogleGenerativeAI
# Gemini LLM
llm = ChatGoogleGenerativeAI(
    model="gemini-2.0-flash", # or gemini-1.5-pro
    temperature=0
)
# Create QA chains
qa_stuff = RetrievalQA.from_chain_type(
    llm=llm,
    chain_type='stuff',
    retriever=retriever
)
# Query
print(qa_stuff.run('Title of the chapters'))
```

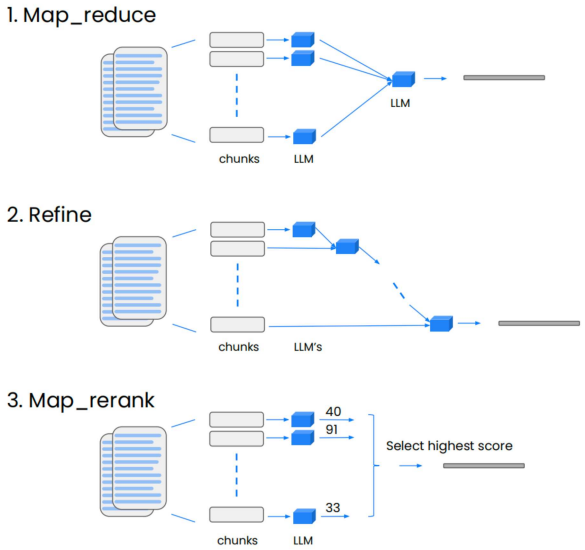
RetrievalQA Chain



You many have too many docs to fit into an LLM context. The solution is to use an LLM to select the 'most relevant' information



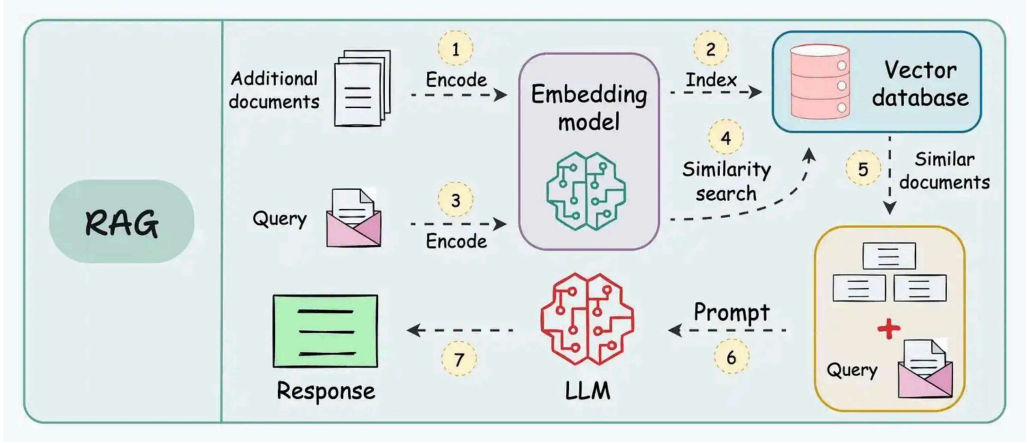
RetrievalQA Chain



RetrievalQA chain patterns

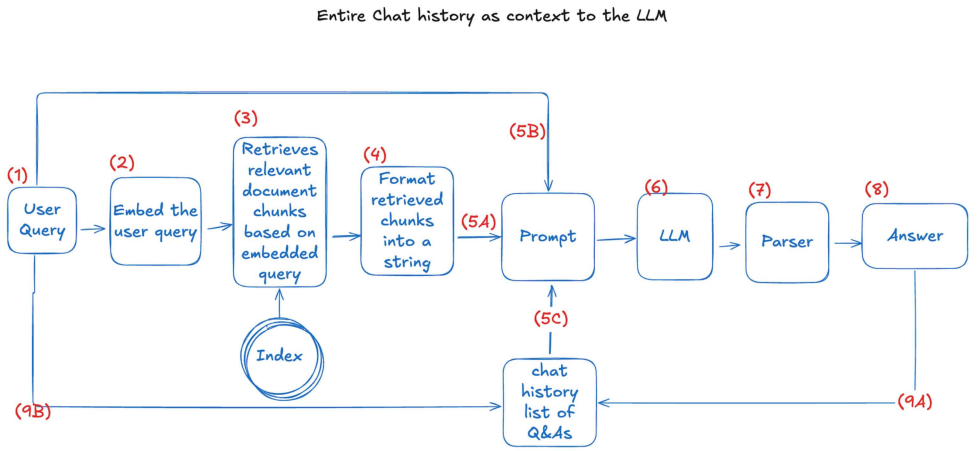
```python
from langchain_classic.chains import RetrievalQA
from langchain_google_genai import ChatGoogleGenerativeAI
# Gemini LLM
llm = ChatGoogleGenerativeAI(
    model="gemini-2.0-flash", # or gemini-1.5-pro
    temperature=0
)
# Create QA chains
qa_stuff = RetrievalQA.from_chain_type(
    llm=llm,
    chain_type='map_reduce',
    retriever=retriever
)
# Query
print(qa_stuff.run('Title of the chapters'))
```


Retrieval Augmented Generation

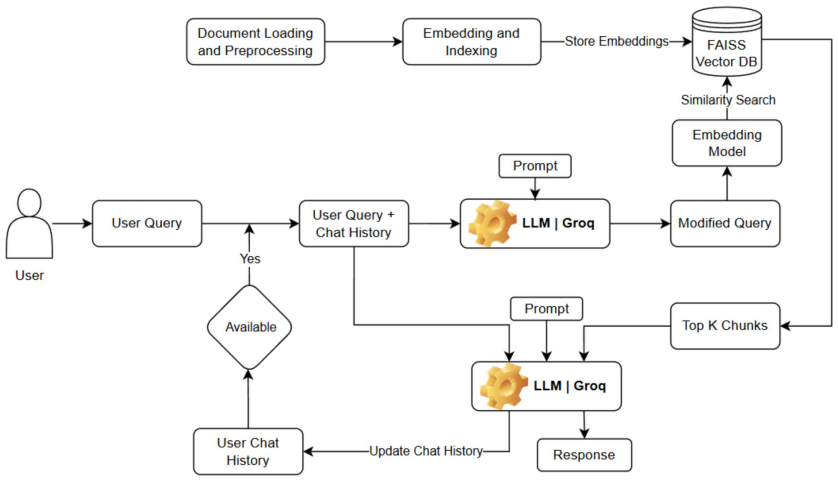


Types of RAG

Conversational RAG



Conversational RAG



Memory Types in LangChain

- **ConversationBufferMemory**
  - Stores full conversation history verbatim.
  - Injects entire chat history into every LLM prompt.
  - Useful for short or simple conversations.
  - Ideal for quick development and prototyping.
  - Can lead to large context windows and higher cost.
- **ConversationSummaryMemory / SummaryBufferMemory**
  - Summarizes older conversation turns using the LLM.
  - Keeps recent messages verbatim for short-term context.
  - Maintains a rolling summary to control token usage.
  - Scales efficiently for long-running dialog systems.
  - Produces concise state representations.

Memory Types in LangChain

- **ConversationBufferWindowMemory**
  - Retains only the last k conversation turns.
  - Ensures fixed-size, predictable memory usage.
  - Suitable for systems relying only on recent context.
  - Avoids summarization overhead.
- **Entity Memory / Key–Value Memory**
  - Stores structured facts such as:
    - User names and identities.
    - Preferences and personal details.
    - Task-specific variables and states.
  - Supports personalization in agents and chatbots.
  - Easily retrieved and injected into prompts.

Memory Types in LangChain

- **Vector-Based Memory (RAG-Style Memory)**
  - Stores embeddings of conversation chunks or summaries.
  - Uses vector databases: Chroma, FAISS, Pinecone, Weaviate.
  - Enables semantic retrieval of past interactions.
  - Scales to very long-term or multi-session memory.
  - Ideal for: Conversational RAG, Domain-specific chat applications.

Hybrid Memory Architectures

- Combine multiple memory types:
  - Buffer memory (recent context)
  - Summary memory (long-term compressed context)
  - Vector memory (semantic retrieval)
- Common in production conversational agents.

RetrievalQA chain patterns

```
from langchain_google_genai import ChatGoogleGenerativeAI
from langchain_classic.chains import ConversationChain
from langchain_classic.memory import ConversationBufferMemory

llm = ChatGoogleGenerativeAI(model="gemini-2.0-flash", temperature=0)
mem = ConversationBufferMemory(memory_key="history", return_messages=True)
chat = ConversationChain(llm=llm, memory=mem, verbose=False)

print("Me : Hi, my name is Shailesh.")
print("Bot : ",chat.predict(input="Hi, my name is Shailesh."))
print("Me : Remember that I like sci-fi movies.")
print("Bot : "+chat.predict(input="Remember that I like sci-fi movies."))
print("Me : What is my name and what do I like?")
answer = chat.predict(input="What is my name and what do I like?")
print("Bot : "+answer)
```


Tool Calling in LangChain

What is Tool Calling?

- A method that allows an LLM to call external functions (“tools”).
- Tools are Python functions wrapped using LangChains `tool` decorator.
- The model decides when to call the tool and with which arguments.

Why Use Tools?

- Add real capabilities: calculations, API calls, DB queries.
- Keep the LLM grounded and reliable.
- Build Agentic AI workflows.

Basic Structure in LangChain

1. Define a Python function.
2. Wrap it using `@tool`.
3. Pass the tool list into the LLM.
4. LLM responds with either: a normal text answer, or a `tool_call` block with arguments.

Example: Tool Calling using Gemini

Step 1: Define a Tool

```
from langchain.tools import tool
@tool
def add(a: int, b: int):
    "Return a + b."
    return a + b
```

Step 2: Use Gemini with Tools

```
from langchain_google_genai import ChatGoogleGenerativeAI
llm = ChatGoogleGenerativeAI(
    model="gemini-2.0-flash",temperature=0,
    tools=[add] # attach tool
)
response = llm.invoke("What is 12 + 9?")
print(response)
```


GraphRAG: Retrieval-Augmented Generation using Graphs

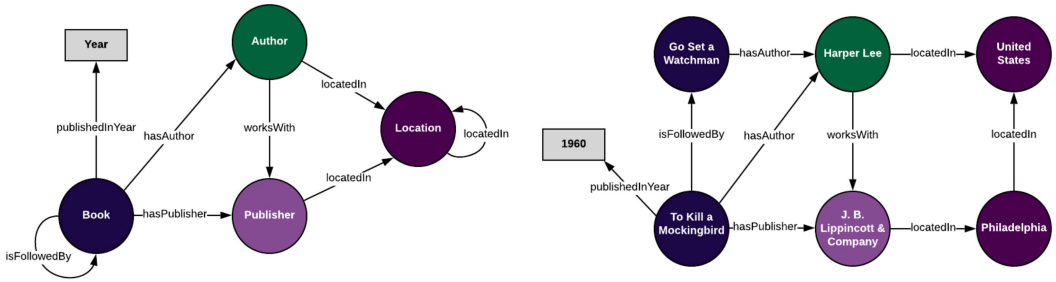
What is GraphRAG?

- A RAG approach where knowledge is stored as a **graph** instead of independent text chunks.
- Nodes represent entities (people, topics, products).
- Edges represent relationships (uses, cites, belongs-to).
- Retrieval is performed over relevant **subgraphs**, not just keyword or vector similarity.

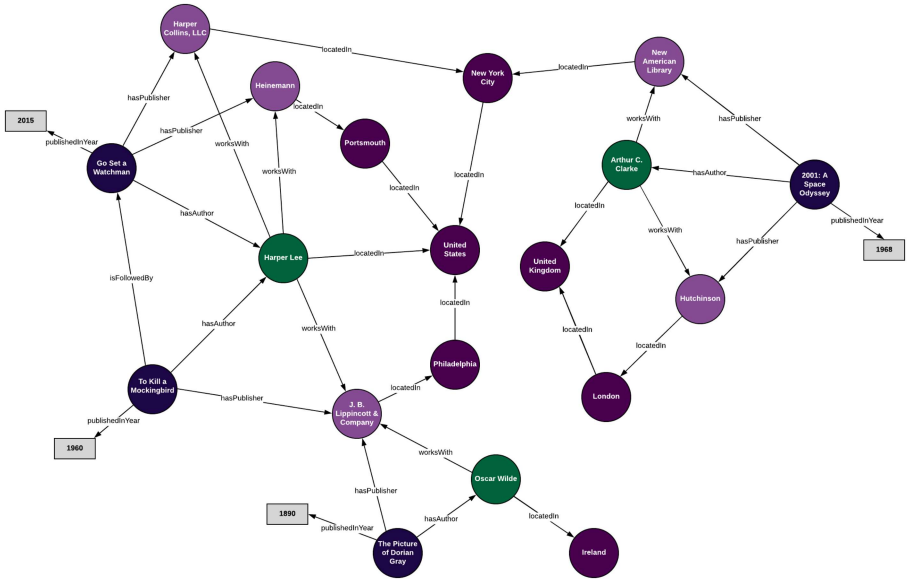
Why GraphRAG?

- Captures structure, relationships, and multi-hop reasoning.
- Better for queries like:
  - "How is X related to Y ?"
  - "Explain the chain of events connecting A and B."
- Improves context precision by retrieving connected graph neighborhoods.

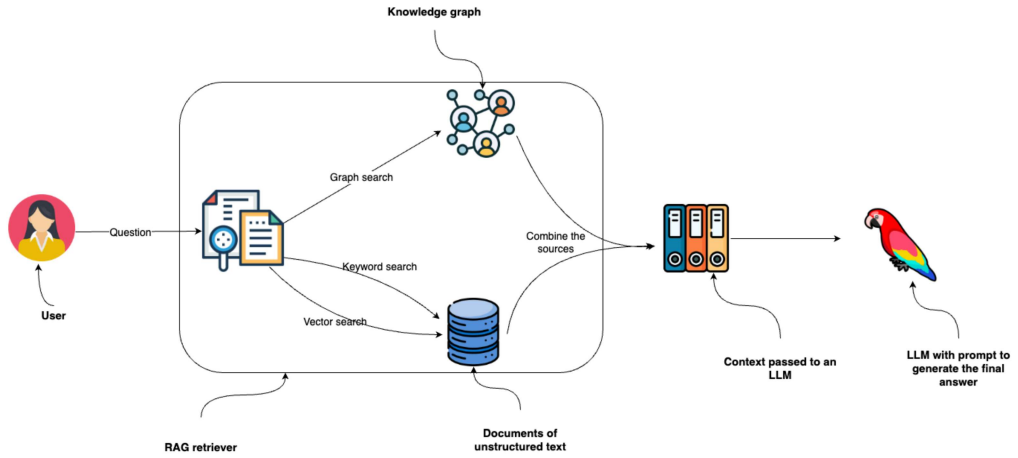
Ontology



Knowledge Graph



Graph RAG



How GraphRAG Works

Pipeline

1. **Ingest Data**
   - Extract entities and relations using NER / IE.
   - Build a Knowledge Graph (KG) or property graph.
2. **Graph Indexing**
   - Create embeddings for nodes / edges.
   - Store in a graph database (Neo4j, Memgraph, ArangoDB).
3. **Graph Retrieval**
   - Expand subgraph around query entity.
   - Retrieve neighbors, paths, or community clusters.
4. **LLM Reasoning**
   - Feed the subgraph context to the LLM.
   - Produce multi-hop, relationship-aware answers.

More RAG Architectures

More RAG Architectures

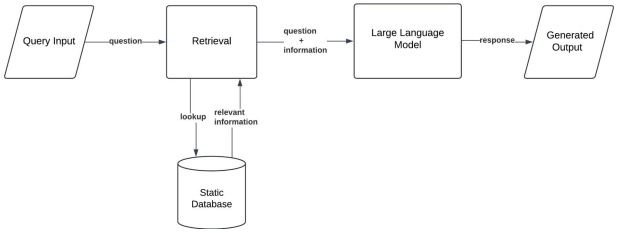


Figure: Simple RAG

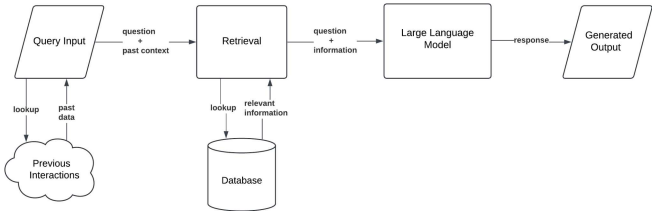


Figure: Simple RAG with Memory

More RAG Architectures

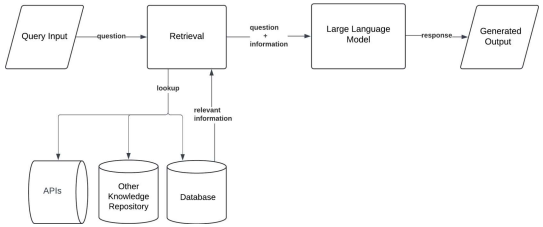


Figure: Branched RAG

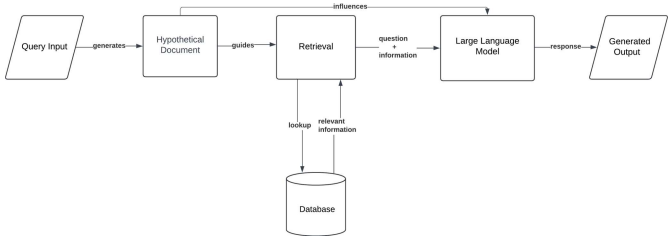


Figure: HyDe (Hypothetical Document Embedding)

More RAG Architectures

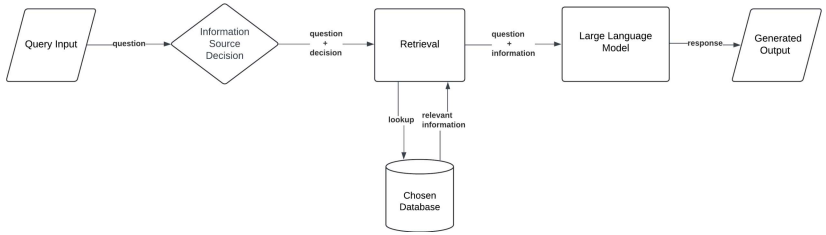


Figure: Adaptive RAG

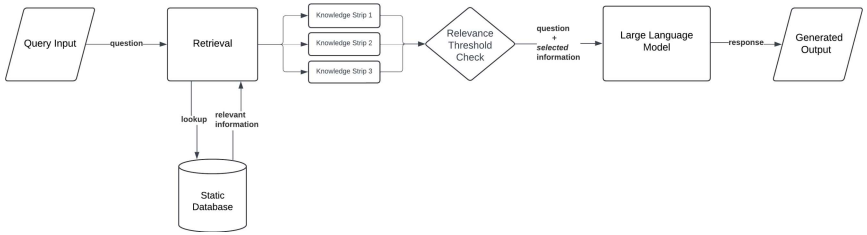


Figure: Corrective RAG (CRAG)

More RAG Architectures

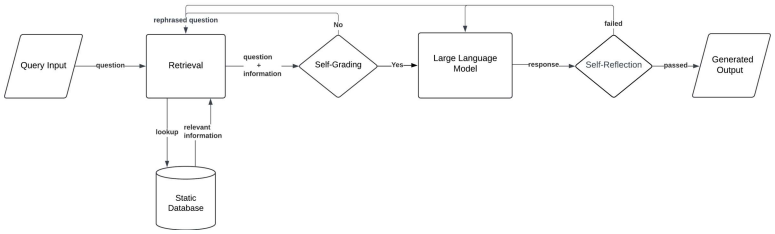


Figure: Self RAG

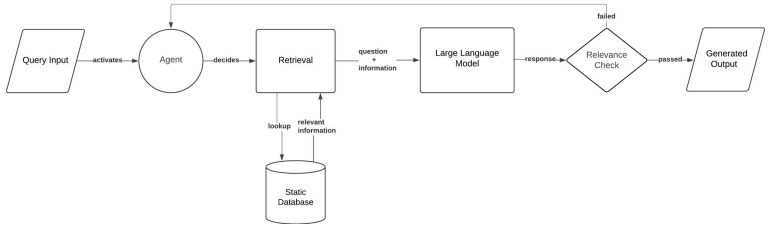


Figure: Agentic RAG

RAG Evaluation

RAG Evaluation: What Do We Measure?

Retrieval-Augmented Generation (RAG) must be evaluated at:

1. **Retrieval Quality**
2. **Context Quality**
3. **Answer (Generation) Quality**
4. **End-to-End System Quality**

Why evaluate RAG?

- Detect retrieval failures
- Reduce hallucinations
- Improve grounding and completeness
- Compare retrievers / LLMs / pipelines

Retrieval Evaluation Metrics

1. **Recall@k**
   - Fraction of relevant documents retrieved in top-k.
2. **Precision@k**
   - Fraction of retrieved documents that are relevant.
3. **MRR (Mean Reciprocal Rank)**

$$MRR = \frac{1}{N} \sum \frac{1}{rank}$$

4. **mAP (Mean Average Precision)**
   - Averages precision across relevant retrieval positions.
5. **nDCG (Normalized Discounted Cumulative Gain)**
   - Discounts relevant results that appear deeper in ranking.

Context Quality Metrics

Why Context Matters?

- LLM answers are only as good as retrieved context.

1. **Context Relevance**
   - Is the context truly related to the query?
2. **Context Coverage**
   - Does the context contain all key facts needed?
3. **Context Precision**
   - How much of the context is useful vs. noise?

Answer Quality Metrics

1. **Faithfulness (Hallucination Rate)**
   - Does the answer stay grounded in retrieved context?
2. **Answer Relevance**
   - Does the answer address the query?
3. **Answer Correctness**
   - Semantic correctness (LLM-as-a-judge or human).
4. **Answer Completeness**
   - Are all important points included?

System-Level RAG Metrics

1. **RAGAS Score**
   - Combines faithfulness, relevance, context recall, and precision.
2. **Latency**
   - Total time per query (retriever + LLM).
3. **Cost per Query**
   - Token cost + retrieval cost.
4. **End-to-End Accuracy**
   - Does the system produce usable and correct answers?

Advanced RAG Evaluation Metrics

1. **Source Attribution Accuracy**
   - Are claims correctly linked to a document?
2. **Citation Quality**
   - Are the cited passages accurate and relevant?
3. **Multi-hop Reasoning Score**
   - Performance on questions requiring multiple steps.
4. **Drift Detection**
   - Detect degradation in retrieval quality over time.

Conclusion

- Conversational AI becomes significantly more powerful when enriched with context and memory.
- LangChain provides multiple memory mechanisms suited for different use-cases: buffer-based, summary-based, windowed, entity-level, and vector-backed.
- Integrating Google Gemini with LangChain enables fast, multimodal, and highly context-aware conversational systems.
- Combining Memory + Retrieval (RAG) gives a scalable architecture for personalized, knowledge-grounded assistants.
- The right memory choice depends on requirements such as token budget, conversation length, personalization depth, and system latency.

Thank You!

For your attention and participation !!!

Dr. Shailesh Sivan

Email: shaileshsivan@gmail.com
Phone: +91-8907230664
Website: www.shaileshsivan.info