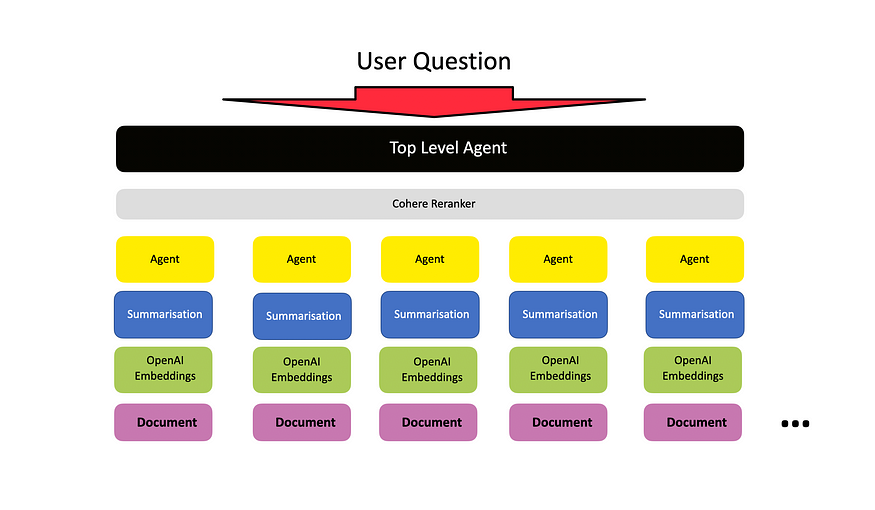
# **Agentic RAG with ApertureDB and Hugging Face SmolAgents**

## **Introduction**

Large Language Models serve as a powerful generative AI tool but can only answer questions within the knowledge of its training datasets. Traditional LLM, if prompted with a focused or a complex question may hallucinate and provide inaccurate responses. Retrieval-Augmented Generation(RAG) is an architecture that helps to add more context to improve the LLM-generated responses.

The input source to these LLMs can be any external data source such as vector databases or search engine records. RAG helps the LLM not only to rely on its internal parameters and training datasets but also introduces RAG pipelines to conduct the retrieval of information from relevant resources. RAG pipelines and data retrieval help in ensuring reduced hallucinations and factual responses.

Traditional RAG follows automated patterns where the LLM directly interacts with the users and query the data store to retrieve relevant responses. Agentic RAG revolutionizes the vanilla RAG by introducing agents in the RAG pipeline flow. Each document is assigned a document agent, then these agents are used to compare and evaluate the summaries generated and decide on the best possible answer. The addition of agents in this overall flow adds some intelligence, allowing it to handle and manage complex queries and tasks.



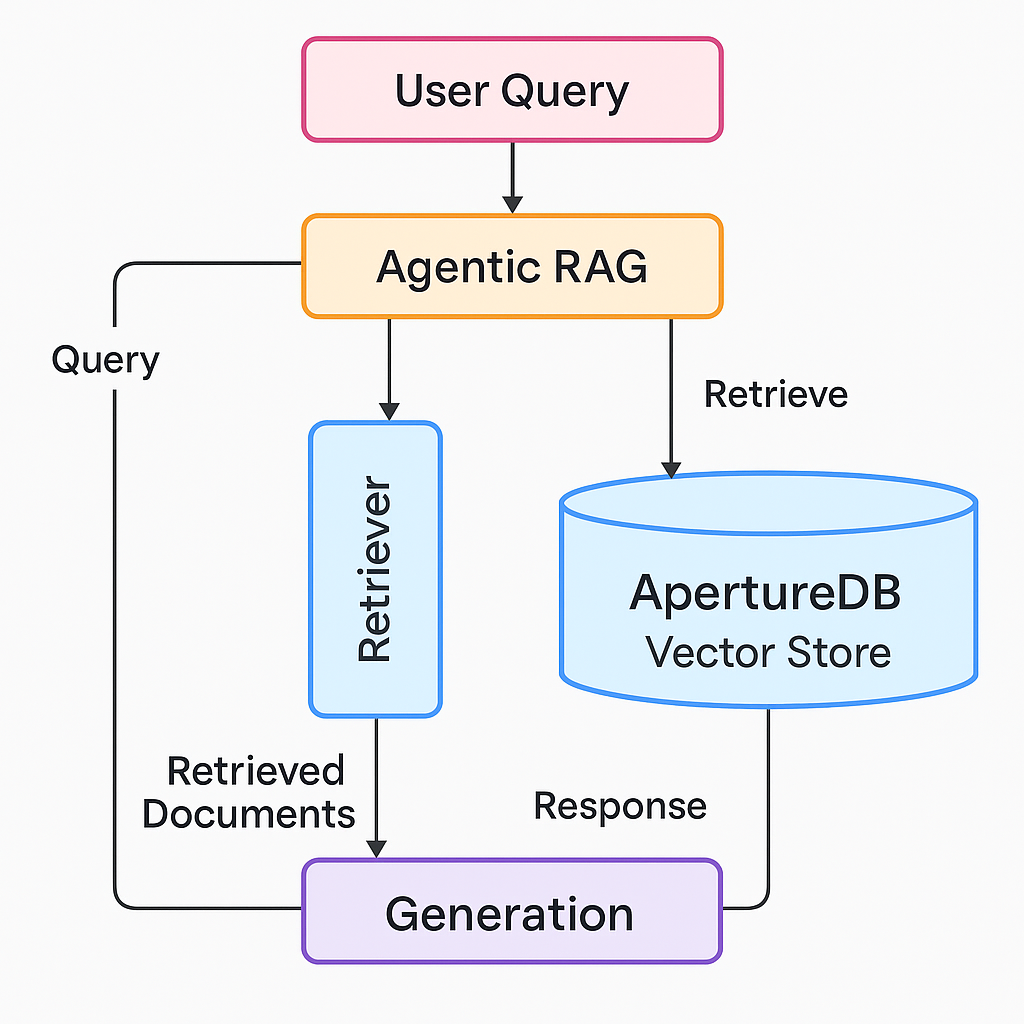
This implementation highlights the Agentic RAG implementation using ApertureDB data store, which is a graph-based multimodal database. Hugging Face SmolAgents will be employed for implementing a multi-agent LLM workflow.

## What is Agentic RAG?

Agentic RAG builds on traditional Retrieval-Augmented Generation by adding autonomous agents into the system to improve how data is retrieved and how responses are generated. These agents bring an extra level of reasoning. They plan, act, and adjust queries based on what’s needed.

Instead of sticking to the user’s original question, these agents rewrite and refine queries multiple times. This helps break down complex questions and filter out irrelevant information, which leads to clearer and more accurate answers. The agents also work together to decide the best way to search for data, making the process more flexible than the straightforward flow of regular RAG.

This approach moves away from one-time retrieval and fixed pipelines. Because the agents keep reasoning and collaborating, the system produces smarter, more relevant responses. Agentic RAG serves as an intelligent agent working towards a specific goal through an iterative flow of reasoning and response generation.



## Implementing Agentic RAG with ApertureDB and Hugging Face SmolAgents

The following steps outline the implementation of Agentic RAG using the Hugging Face SmolAgents.

### **1. Preparing the Data**

For this implementation, we’ll be using an Arxiv structured complex dataset. Which will be first pre-processed into embeddings so that they can be stored in the ApertureDB vector database. The large dataset is divided into small chunks of data, out of which vector embeddings are generated using a model like sentence-transformer/all-MiniLM-L6-v2 from Hugging Face. These vector embeddings are then stored in apertureDB.

import arxiv

from sentence\_transformers import SentenceTransformer

from typing import List

import numpy as np

def get\_arxiv\_papers(query: str, max\_results: int = 10) -> List[str]:

"""Fetch papers from arXiv and return their text content"""

client = arxiv.Client()

search = arxiv.Search(

query=query,

max\_results=max\_results,

sort\_by=arxiv.SortCriterion.SubmittedDate

)

papers = []

for result in client.results(search):

papers.append(f"Title: {result.title}\nAbstract: {result.summary}")

return papers

def chunk\_text(text: str, chunk\_size: int = 512) -> List[str]:

"""Split text into chunks of specified size"""

words = text.split()

chunks = [' '.join(words[i:i+chunk\_size]) for i in range(0, len(words), chunk\_size)]

return chunks

# Fetch and prepare data

arxiv\_query = "graph embeddings"

papers = get\_arxiv\_papers(arxiv\_query)

chunks = []

for paper in papers:

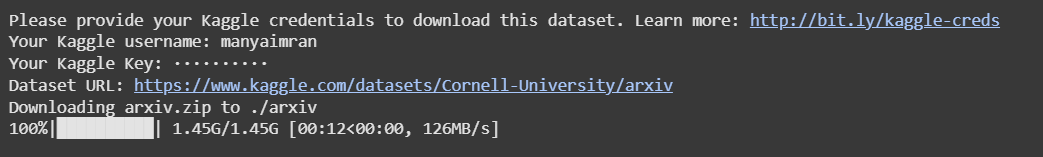
chunks.extend(chunk\_text(paper))

# Generate embeddings

model = SentenceTransformer("sentence-transformers/all-MiniLM-L6-v2")

embeddings = model.encode(chunks, show\_progress\_bar=True)

print(f"Generated {len(embeddings)} embeddings for {len(chunks)} chunks")





### **2. Setting Up ApertureDB**

In this implementation, ApertureDB is used as a vector store that offers high performance and speedy retrievals, also supporting multimodal data. The vector based stores help in maintaining context as vectors while generating responses. ApertureDB also integrates seamlessly with AI pipelines like LangChain and Hugging Face. This is how you store vector embeddings in ApertureDB.

import aperturedb

from aperturedb.Connector import Connector

import time

class ApertureDBVectorStore:

def \_\_init\_\_(self, host: str = "localhost", port: int = 8000):

self.client = Connector({"host": host, "port": port})

def create\_schema(self):

"""Create necessary schema in ApertureDB"""

schema = [ {

"CreateProperty": {

"name": "type",

"type": "string"

}

},

{

"CreateProperty": {

"name": "text",

"type": "string"

}

},

{

"CreateVectorIndex": {

"name": "default\_embedding",

"dimensions": 384,

"metric": "cosine"

}

}

]

response, \_ = self.client.query(schema)

return response

def store\_embeddings(self, chunks: List[str], embeddings: np.ndarray):

"""Store text chunks with their embeddings in ApertureDB"""

records = []

for chunk, embedding in zip(chunks, embeddings):

records.append([

{

"AddObject": {

"properties": {

"type": "document",

"text": chunk

},

"embedding": {

"vector": embedding.tolist(),

"name": "default\_embedding"

}

}

}

])

batch\_size = 50

for i in range(0, len(records), batch\_size):

batch = records[i:i+batch\_size]

response, blobs = self.client.query(batch)

print(f"Inserted batch {i//batch\_size + 1}, response: {response}")

time.sleep(0.1)

return True

# Initialize and populate ApertureDB

aperture\_db = ApertureDBVectorStore()

aperture\_db.create\_schema()

aperture\_db.store\_embeddings(chunks, embeddings)

### **3. Building the Agentic Workflow**

For the agentic flow, we proceed by defining agents using smolAgents in Hugging Face. These agents are responsible for query reformulation, iterative retrievals, and dynamic reranking. All of these are basically an iterative process of rewriting the initial query to get results and prioritizing the results of the iterations for the best result. Here’s how you define the agentic logic for query refinement and retrieval.

from typing import Dict, Any

from langchain.vectorstores import VectorStore

from langchain.embeddings import HuggingFaceEmbeddings

from smolagents import ToolCallingAgent, LiteLLMModel

class ApertureDBRetriever:

def \_\_init\_\_(self, aperture\_db: ApertureDBVectorStore, k: int = 5):

self.db = aperture\_db

self.k = k

def similarity\_search(self, query: str, k: int = None) -> List[Dict[str, Any]]:

"""Perform similarity search in ApertureDB"""

k = k or self.k

query\_vector = model.encode(query).tolist()

search\_query = [

{

"FindObject": {

"with\_vector": {

"name": "default\_embedding",

"vector": query\_vector,

"k": k

},

"properties": ["text"],

"results": {

"list": ["text"]

}

}

}

]

response, \_ = self.db.client.query(search\_query)

if not response or 'FindObject' not in response[0]:

return []

results = []

for i, item in enumerate(response[0]['FindObject']['entities']):

results.append({

"content": item['properties']['text'],

"score": item['vector\_distance'],

"index": i

})

return results

class AgenticRetriever:

def \_\_init\_\_(self, retriever: ApertureDBRetriever):

self.retriever = retriever

def \_\_call\_\_(self, query: str) -> str:

"""Retrieve relevant documents and format the output"""

retrieved\_docs = self.retriever.similarity\_search(query)

if not retrieved\_docs:

return "No relevant documents found."

output = "Retrieved documents:\n"

for doc in retrieved\_docs:

output += f"\n--- Document {doc['index'] + 1} (score: {doc['score']:.3f}) ---\n"

output += doc['content'][:500] + ("..." if len(doc['content']) > 500 else "")

output += "\n"

return output

aperture\_retriever = ApertureDBRetriever(aperture\_db)

agentic\_retriever = AgenticRetriever(aperture\_retriever)

### **4. Integrating with ApertureDB**

The last step is to connect the Hugging Face smolAgents defined to the ApertureDB containing the vector embeddings of the dataset RAG is working on. Here’s how you integrate the database with the Hugging Face Agentic AI pipeline for refined query and retrieving results.

from dotenv import load\_dotenv

import os

# Load environment variables (for API keys)

load\_dotenv()

def main():

# Initialize the agent with our retriever tool

model = LiteLLMModel(model\_id="gpt-4-turbo", api\_key=os.getenv("OPENAI\_API\_KEY"))

agent = ToolCallingAgent(

tools=[agentic\_retriever],

model=model,

system\_message="You are a helpful research assistant. Use the tools provided to retrieve relevant academic papers."

)

queries = [

"why are graph embeddings used for context preservation",

"latest research on knowledge graph embeddings",

"comparison of different graph embedding techniques"

]

for query in queries:

print(f"\n=== Query: {query} ===")

response = agent.run(query)

print("\nResponse:")

print(response)

if \_\_name\_\_ == "\_\_main\_\_":

main()

## Why ApertureDB Matters in RAG

Vector stores are a fundamental component in RAG architecture, providing an efficient way to perform data retrieval on large unstructured datasets. Vector-based data stores help store data as multidimensional vectors to improve relevancy by comparing vector embeddings. These dimensions allow the similarity score metric to have more context while retrieving responses from the dataset.

ApertureDB's has a unique strength to conduct vector similarity retrieval and perform structured text-based data retrieval using a single query. This hybrid retrieval helps with multimodal data processing in RAG, which helps interpret textual metadata or embeddings with images.

Aperture DB low-latency searches makes it an ideal data store for production-scale RAG pipelines. The graph-based data storage helps build associations between the data points to generate more contextually accurate responses. Aperture DB also offers seamless integration with AI agentic frameworks, helping developers incorporate data stores in their agentic workflows.

## Conclusion

Agentic RAG revolutionizes the traditional RAG architecture by introducing iterative querying, refining, and quality assessments of the query and responses. This dynamic reasoning process leads to more accurate and context-aware answers. Integrating ApertureDB enhances this approach by using graph-based data storage to preserve context, making it easier for agents to reason and act effectively.

ApertureDB’s high-performance capabilities make it well-suited for handling complex datasets like Arxiv papers.This flow introduces dynamic reasoning, query refinement and iteration, response improvement, and high retrieval enhances the performance compared to the traditional static pipelines.

## References

Building Agentic RAG Systems - Hugging Face Agents Course. (2025). Huggingface.co. <https://huggingface.co/learn/agents-course/en/unit2/smolagents/retrieval_agents?utm_source=chatgpt.com>

Singh, A., Ehtesham, A., Kumar, S., & Khoei, T. T. (2025). Agentic Retrieval-Augmented Generation: A Survey on Agentic RAG. *arXiv preprint arXiv:2501.09136*.

Shukla, A. K. The Future of AI: How Domain-Specific LLMs, RAG & Agentic AI Are Redefining Intelligence

Gemini 2.0 vs. Agentic RAG: Who wins at Structured Information Extraction? – Unstructured. (2025). Unstructured.io. <https://unstructured.io/blog/gemini-2-0-vs-agentic-rag-who-wins-at-structured-information-extraction?utm_source=chatgpt.com>  
Gemini 2.0 vs. Agentic RAG: Who wins at Structured Information Extraction? – Unstructured. (2025). Unstructured.io. https://unstructured.io/blog/gemini-2-0-vs-agentic-rag-who-wins-at-structured-information-extraction?utm\_source=chatgpt.com

‌

‌