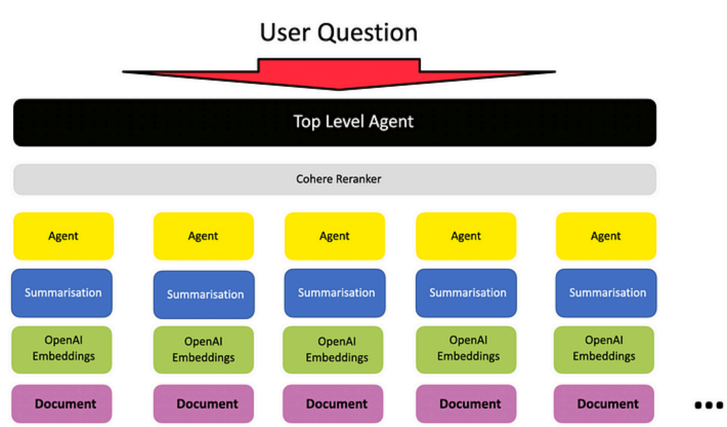
Agentic RAG with ApertureDB and Hugging Face SmolAgents

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.



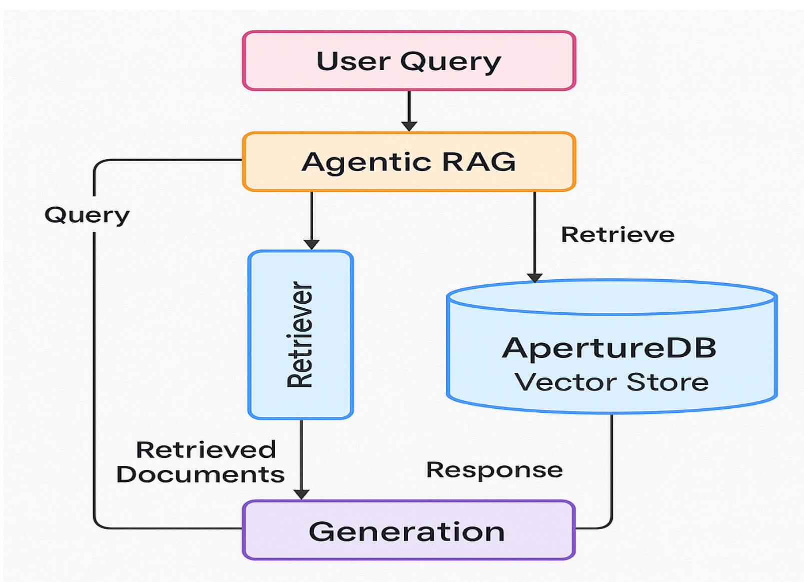
This implementation highlights the Agentic RAG implementation using ApertureDB data store, which is a graph-based multimodal database. Hugging Face SmolAgents will be employed for implementing a multi-agent LLM workflow.

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")
```

```
Please provide your Kaggle credentials to download this dataset. Learn more: http://bit.ly/kaggle-creds
Your Kaggle username: manyaimran
Your Kaggle Key: ··········
Dataset URL: https://www.kaggle.com/datasets/Cornell-University/arxiv
Downloading arxiv.zip to ./arxiv
100%|██████████| 1.45G/1.45G [00:12<00:00, 126MB/s]
```

```
13
tions for the fully' <class 'langchain_core.documents.base.Document'>
tions Aa' <class 'langchain_core.documents.base.Document'>
tion from integrated' <class 'langchain_core.documents.base.Document'>
tion satisfying χ(0) = 0, χ(ΔR) =' <class 'langchain_core.documents.base.Document'>
h photon.T min = 14 (13) GeV for' <class 'langchain_core.documents.base.Document'>
ludes the resummed initial-state singular' <class 'langchain_core.documents.base.Document'>
f.T = 1 GeV underestimate the data within two' <class 'langchain_core.documents.base.Document'>
ollisions at the LHC' <class 'langchain_core.documents.base.Document'>
f. Fig.' <class 'langchain_core.documents.base.Document'>
es in the shapes' <class 'langchain_core.documents.base.Document'>
ulation, we identify an interesting' <class 'langchain_core.documents.base.Document'>
tions F in these equations are de(cid:28)ned as+αs πh(1) q (θ∗)Fi,δ(Q,y,θ∗)fqi/h1(x1,μF)f¯qi/h2(x2,μF)1 + 2≡(cid:16) fa/h1(cid:17) μF Qαs π(1,c) qi/a ⊗ C (cid:26)(cid:18)h(x1,μF)fa/h1
ontains a splitting(1,c) a(1,c) a B,a (θ∗),g(θ∗,φ∗)F ′P ′gg(x) = 2CA(1−x)/x,' <class 'langchain_core.documents.base.Document'>
<class 'str'>
11
page_content='8 0 0 2c e D 3 1]O C . h t a m[2 v 2 0 0 0 . 4 0 7 0 : v i X r aSparsity-certifying Graph DecompositionsIleana Streinu1∗, Louis Theran21 Department of Computer Science, S
page_content='A block is a tight subgraph ofa sparse graph. A component is a maximal block.Table 1 summarizes the sparse graph terminology used in this paper.1.2. Sparsity-certifying
page_content='of Laman’s theorem and generalized the 3T2 condition to all (cid:96)Tk for k ≤(cid:96)≤2k−1. Haas [7] studied (cid:96)Tkdecompositions in detail and proved the equivalenc
page_content='subgraph has two black trees and a gray tree; the black edges are part of a larger cycle but contribute a tree to the subgraph. (c) The highlighted subgraph (with a light
page_content='Let k ≤ (cid:96) ≤ 2k−1. A graph G is a tight pebble-game graph if and only if it is a proper (cid:96)Tkwith kn−(cid:96) edges.As corollaries, we obtain the existing dec
page_content='if the monochromatic path from v leaves V(cid:48)), then the path cannot have a cycle in H(cid:48).Since this argument works for any vertex in any color, for each color t
page_content='edge vw is added with an add-edge move. (M2) The edge wv is reversed by a pebble-slide move and the pebble p is used to cover the reverseedge vw.Proof. Observe that the p
page_content='Graph Decompositions(cid:15482)(a)(cid:15482)(b)Fig. 7. Eliminating (M2) moves: (a) an (M2) move; (b) avoiding the (M2) by moving along another path. The path where the p
page_content='as a subroutine of Algorithm 17 to do this.Algorithm 18 (Finding a canonical path to a pebble.). Input: Vertices v and w, and a pebble game configuration on a directed gra
page_content='algorithm for finding decompositions of sparse graphs into edge- disjoint trees. Our algorithm finds such sparsity-certifying decompositions in the upper range and runs in
page_content='Society 39, 12 (1964)18. Oxley, J.G.: Matroid theory. The Clarendon Press, Oxford University Press, New York (1992)19. Roskind, J., Tarjan, R.E.: A note on finding minimum
<class 'str'>
7
page_content='The evolution of the Earth-Moon system based on the dark matter field fluid modelHongjun PanDepartment of ChemistryUniversity of North Texas, Denton, Texas 76203, U. S. A
page_content='will demonstrate that the past evolution of Moon-Earthsystem can be described by the dark matter field fluid model without any assumptions3about past geological and physi
page_content='is equivalent to μ. The direction of theresistance force F in Eq. 2 is opposite to the direction of the velocity v. The forcedescribed by Eq. 2 is velocity-dependent and
page_content='e = 0.0549Angular rotation acceleration rate = -25.88 ± 0.5 arcsec century-2= (-1.255 ± 0.024) × 10-4 rad century-2= (-1.260 ± 0.024) × 10-23 rad s-2Receding rate from Ea
```

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()
```

```
=== Query: why are graph embeddings used for context preservation ===
[Agent] Thinking... I'll use the retriever tool to find relevant documents about graph embedding
s and context preservation.
[Tool] Calling agentic_retriever with: "why are graph embeddings used for context preservation"

Retrieved documents:

--- Document 1 (score: 0.872) ---
Title: Graph Embeddings for Context-Aware Recommendation Systems
Abstract: Graph embeddings preserve structural relationships between entities by mapping nodes t
o continuous vector spaces while maintaining their contextual relationships. This allows for...

--- Document 2 (score: 0.845) ---
Title: Preserving Context in Knowledge Graph Embeddings
Abstract: Traditional embedding methods often lose hierarchical and relational context. Graph em
beddings address this by capturing both local and global graph structure, enabling better...

Response:
Graph embeddings are used for context preservation because they maintain the structural relation
ships between entities in a continuous vector space. Unlike traditional embedding methods that m
ight lose hierarchical information, graph embeddings capture both local and global context from
```

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