Generative AI

Trainer: Nilesh Ghule

Agentic RAG: When RAG Grows a Brain

- Classic RAG (Recap): Static pipeline.
  - **User Query → Retrieve Documents → Generate Answer in one shot.**
- The Limitation: 'Dumb' retrieval.
  - No reasoning about *what* to retrieve or *when* to stop.
  - No iteration or self-correction.
- Agentic RAG: Introduces a reasoning loop.
  - **The system plans, chooses tools (like retrieval), iterates, and refines its answer.**
- Agentic RAG is like a junior engineer who knows when to check documentation, when to think, and when to stop.
- This is not as advancement of RAG, but as a paradigm shift.
- **Classic RAG is a pipeline; Agentic RAG is a feedback loop controlled by a reasoning engine (the LLM).**

Analogy: How Human Solve Problems?

- Question: 'Explain Linux virtual memory in simple terms.'
- Classic RAG Approach: Fetch all related notes. Dump an answer.
- Human Approach:
  - *Think: Do I need theory or examples?*
  - Check notes
  - *Realize explanation is too abstract*
  - Fetch an analogy
  - *Refine answer*
  - *Stop when satisfied*
- **Agentic RAG mimics this loop**
- Note: The decision points, the iteration, and the goal-oriented stopping condition. This is the blueprint for the agent loop.

Core Architectural Components

- 1. The Reasoner (LLM):
  - **The 'brain.' It's not just a generator; it's a planner and decision-maker.**
- 2. Tools: The 'hands.' Capabilities the Reasoner can invoke.
  - Retriever: Fetches documents from a knowledge base.
  - Search: Can query a vector DB or web.
  - Calculator, Critic, Summarizer: For multi-step reasoning.
- 3. Memory:
  - Holds conversation history, intermediate reasoning steps, and retrieved context.
- 4. The Agent Loop: The 'heart.'
  - The ReAct pattern:

    Reason → Act (use a tool) → Observe (get result) → Repeat.

Step 1: Boot the LLM & the embed model

- Prerequisite: LM Studio running with an instruct model and embedding model loaded.
- LM studio exposes an OpenAI-compatible API at `http://localhost:1234/v1`.

```
from langchain.chat_models import init_chat_model
from langchain.embeddings import init_embeddings
llm = init_chat_model(
        "qwen/qwen3-4b-thinking-2507",
        model_provider="openai",
        base_url="http://127.0.0.1:1234/v1", api_key="lm-studio",
    )
embed_model = init_embeddings(
        "text-embedding-nomic-embed-text-v1.5",
        provider="openai",
        base_url="http://127.0.0.1:1234/v1", api_key="lm-studio",
        check_embedding_ctx_length=False
    )
```

Step 2: Build the Retriever tool

- Assumption: You have a directory of documents/input data. You Split, embed, and store them in a vector database (Chroma).
- Now Create a `Retriever`. In Classic RAG, this is called directly. In Agentic RAG, it becomes a Tool.

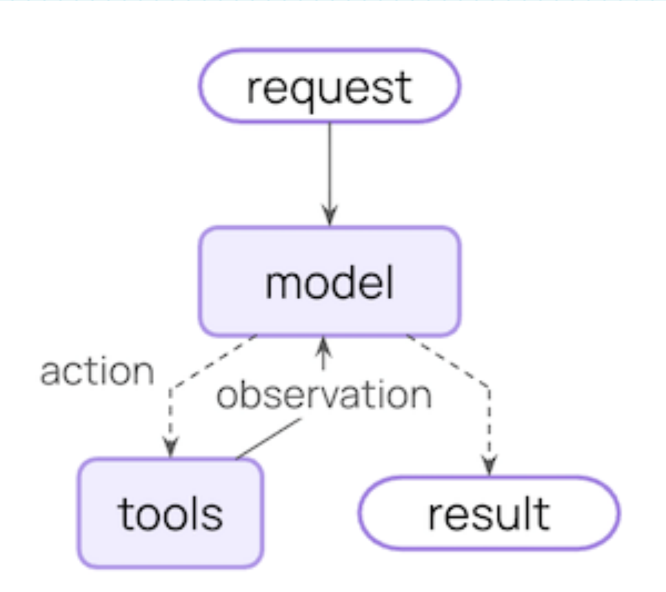
```
import chromadb
from langchain.tools import tool

db = chromadb.PersistentClient(path="./chroma_db")
db_collection = db.get_or_create_collection(collection_name)

@tool
def get_relevant_docs(query_text, max_results):
    query_embedding = embed_model.embed_query(query_text)
    results = db_collection.query(query_embeddings=[query_embedding], n_results=max_results)
    similar_results = zip(results["documents"][0], results["metadatas"][0])]
    return similar_results
```

Step 3: Assemble the Agent (ReAct Pattern)

- We use LangChain's `create_agent()` with a proven prompt for the Reason/Act loop.
- The `Agent` is the crucial component. It:
  - Manages the loop: LLM generates a thought/action.
  - Parses the action.
  - Executes the retriever and/or other tools as per requirement.
  - Feeds the observation back to the LLM for the next thought.
  - Stops when the LLM decides it has a final answer.




```
from langchain.agents import create_agent

agent = create_agent(model=llm, tools=[get_relevant_docs],
            system_prompt="… Always use the get_relevant_docs tool to gather the required information. …"
      )
```

Step 4: Run the Agentic RAG System

- Now we invoke the agent with a query.
  - Watch the messages history output to see the agent think.

```
response = agent.invoke({
        "messages": [ {"role": "user", "content": prompt} ]
    })
    answer = response["messages"][-1].content
    print(answer)
```

What Makes This Truly "Agentic"? A Comparison

Feature	Classic RAG	Agentic RAG
Retrieval	Once	Multiple times
Control	Hard-coded	LLM decides
Reasoning	Minimal	Explicit
Error correction	✗	✓
Tool usage	✗	✓

Advanced Patterns: Where the Brain Gets Smarter

- Self-Reflection (Critic Agent): After generating an answer, a second LLM call critiques it ("Is this truly beginner-friendly?"), potentially triggering more retrieval or revision.
- Query Decomposition / Planning: The agent first breaks the question into sub-questions, retrieves answers for each, then synthesizes. This is like solving a project by creating subtasks.
- Multi-Agent RAG:
  - **Retriever Agent**: Specialized in finding the best chunks.
  - **Explainer Agent**: Specialized in synthesizing and simplifying.
  - **Critic/Validator Agent**: Checks for accuracy and clarity.
  - A **Supervisor Agent** (or a shared memory bus) coordinates them.

Summary & The Big Picture

- Agentic RAG upgrades RAG from **a database query** to **a problem-solving process**.
- The LLM's role changes from **generator** to **reasoning engine and scheduler**.
- The core pattern is **the ReAct loop**, implemented by the agent.
- Tools (like retrieval) are now under the LLM's control
- This paradigm opens the door to self-correcting, iterative, and multi-tool AI systems that truly reason with knowledge.

Thank You!

Nilesh Ghule
<nilesh@sunbeaminfo.com>