ENERGY GeO 5th Hackathon

Society of Petroleum Engineers

#DatafyingEnergy

From October 3, 2025

LLMs and RAG for Geosciences – Session 2

Dr Diego Corona-Lopez

Italian Section | Netherlands Section | London Section | Romanian Section

Croatian Section | SPE University of Wyoming Chapter | Stavanger Section | Geothermal Technical Section | Data Science and Engineering Analytics Technical Section

www.spegeohackathon.com

INTRODUCTION TO RAG

PRESENTER

SESSION-02

MODULE-01

AI SPECIALIST

DR DIEGO CORONA-LOPEZ

CLDiego/SPE_GeoHackathon_2025.git

AGENDA

- **Why RAG?** *Understand the limitations of standard LLMs and how RAG solves them*
- **The Indexing Pipeline:** *Learn the offline process to prepare our knowledge base:* **Load, Split, Embed, and Store**
- **The RAG Chain:** *Build the real-time query pipeline with a* **Retriever, Prompt, and LLM**
- **Putting It All Together:** *Compose the full chain using the LangChain Expression Language (LCEL)*
- **Hands-On & Troubleshooting**

S2 / / M1

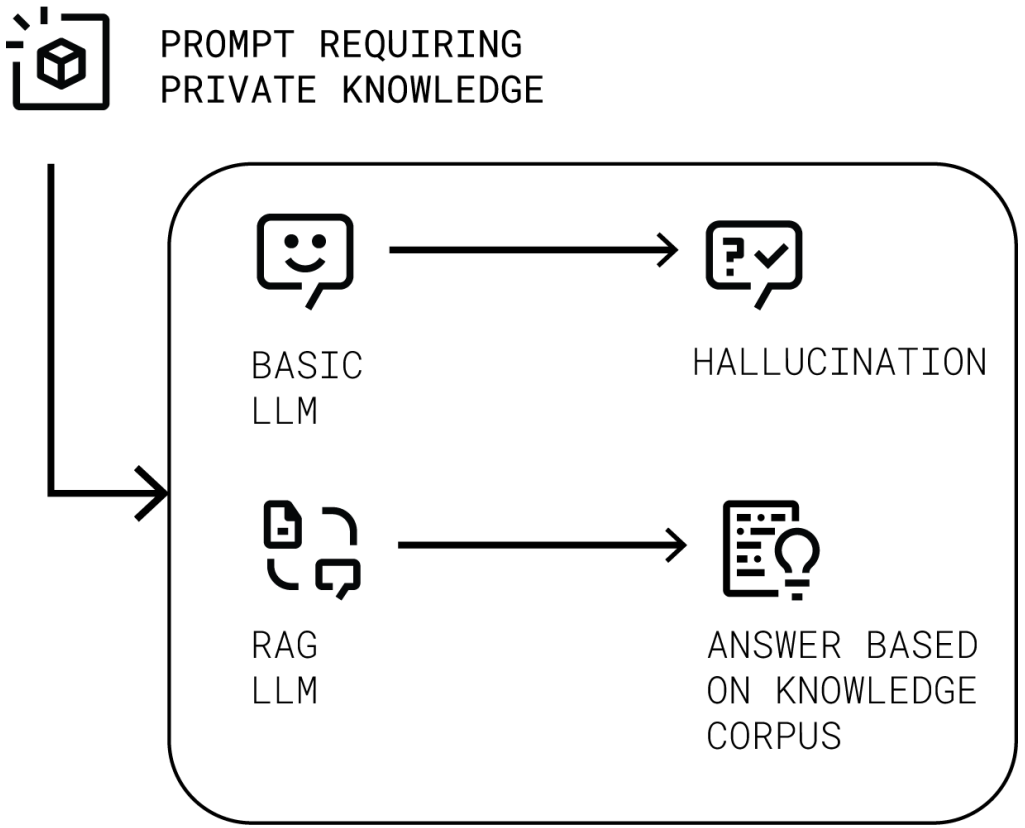
CLDiego/SPE_GeoHackathon_2025.git

WHY USE RAG?

Standard LLMs are powerful but have fundamental limitations:

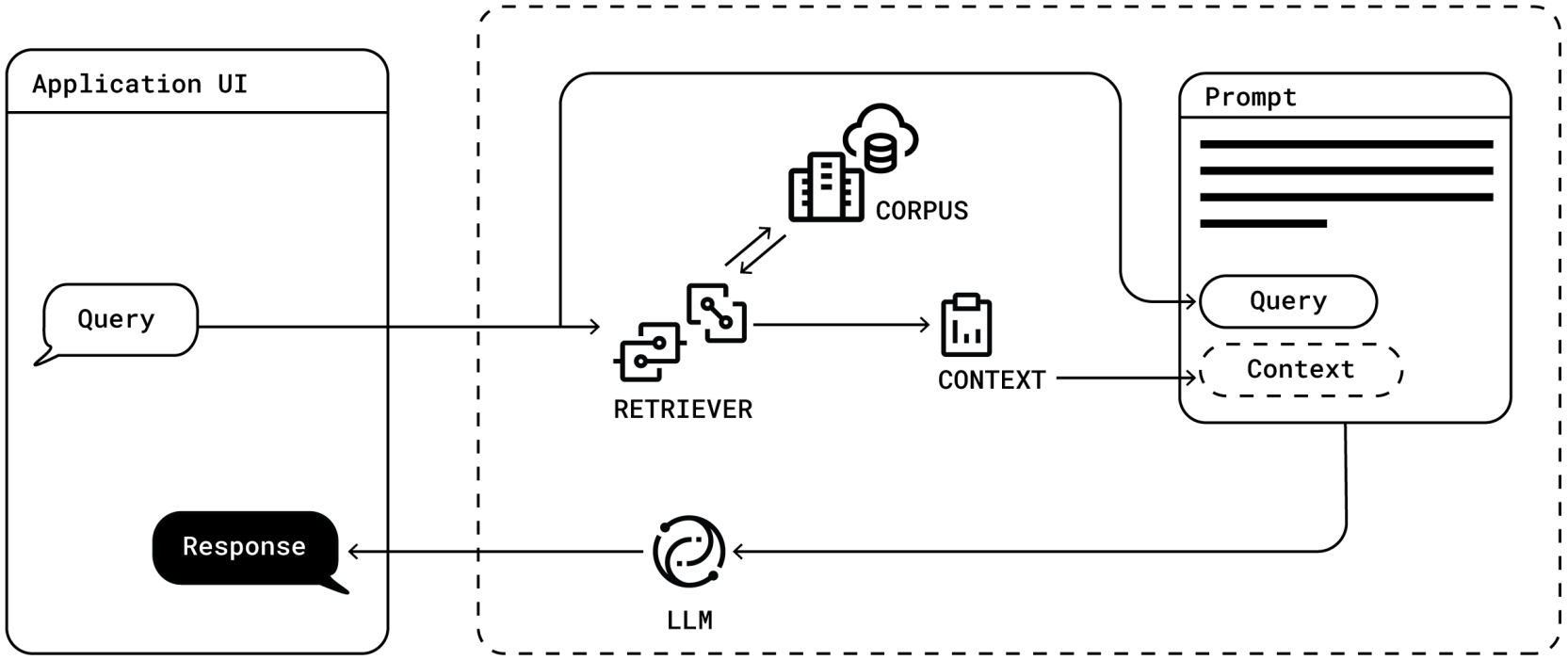
- **Knowledge is Static:** They don't know about recent events or data created after their training.
- **Knowledge is Generic:** They have no access to your private, proprietary, or domain-specific data (e.g., internal well reports, confidential research).
- **Answers are Opaque:** They can't cite sources, making their answers difficult to verify (a "black box").

RAG directly addresses these issues by giving the LLM an "open book" to consult.



WHAT IS RAG?

RAG is a technique for providing an LLM with external knowledge *at the moment it's asked a question*.

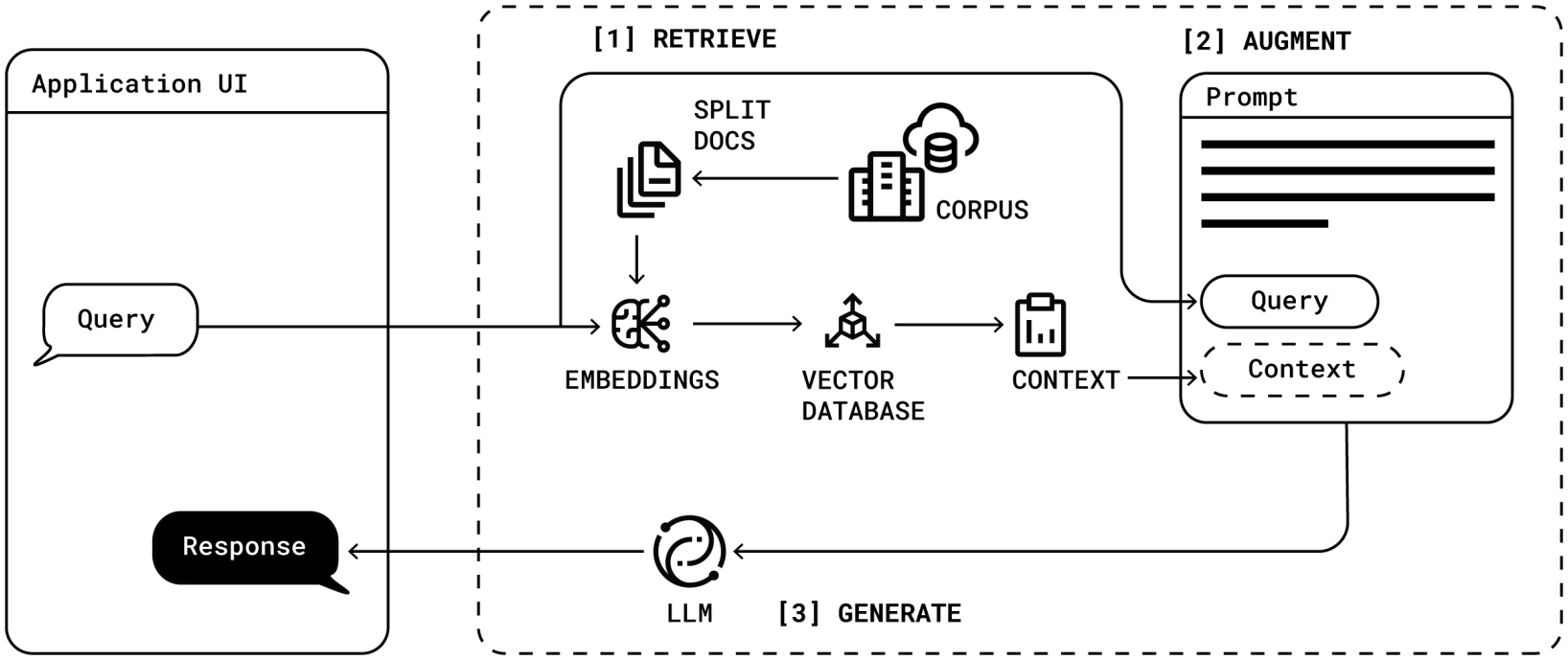


The Two-Stage Process:

Retrieve: Given a user's question, search a knowledge base (your documents) for relevant text chunks.

Generate: Pass the original question *and* the retrieved chunks to the LLM and ask it to synthesise an answer based *only* on the provided context.

RAG ARCHITECTURE



Building a RAG system involves two distinct workflows:

1. **The Indexing Pipeline (Offline):** A one-time process to prepare your knowledge base.

   Load -> Split -> Embed -> Store

2. The RAG Chain (Online): The real-time process that answers user queries.

Retrieve -> Generate

CLDiego/SPE_GeoHackathon_2025.git

ST-1: LOADING DATA

.py

```python
from langchain_core.documents import Document
import json

# From S2_M1_RAG.ipynb
with open(json_path, "r") as f:
    data = json.load(f)

docs: list[Document] = []
for rec in data:
    text = (rec.get("content") or "").strip()
    if not text:
        continue
    meta = {
        "id": rec.get("id"),
        "topic": rec.get("topic"),
        "title": rec.get("title"),
    }
    docs.append(Document(page_content=text, metadata=meta))

print(f"Built {len(docs)} LangChain Documents.")
```

The first step is to load your raw data (PDFs, JSON, text files) into a standard format.

LangChain's DocumentLoaders handle this.

Document objects contain the text (page_content) and metadata (like the source file), which is crucial for citations later.

CLDiego/SPE_GeoHackathon_2025.git

PRESENTER → CORONA-LOPEZ

ST-2: CHUNK SPLIT

.py

```python
from langchain_text_splitters import RecursiveCharacterTextSplitter

# From S2_M1_RAG.ipynb
CHUNK_SIZE = 1000
CHUNK_OVERLAP = 200

splitter = RecursiveCharacterTextSplitter(
    chunk_size=CHUNK_SIZE,
    chunk_overlap=CHUNK_OVERLAP
)
chunks = splitter.split_documents(docs)

print(f"Split {len(docs)} documents into {len(chunks)} chunks.")
```

LLMs have a **limited context window**, so we must split large documents into smaller, manageable chunks

The RecursiveCharacterTextSplitter is a smart way to do this. It tries to split on paragraphs and sentences first to keep related text together

chunk_size: The maximum number of characters in each chunk. A good starting point is 500-1000

chunk_overlap: The number of characters to overlap between adjacent chunks. This helps maintain context across the splits

CLDiego/SPE_GeoHackathon_2025.git

ST-3: EMBEDDINGS

An **Embedding Model** converts each text chunk into a numerical vector. This vector captures the chunk's **semantic meaning**

.py

```python
from langchain_huggingface import HuggingFaceEmbeddings

# From S2_M1_RAG.ipynb
MODEL_EMBED = "sentence-transformers/all-MiniLM-L6-v2"

embeddings = HuggingFaceEmbeddings(model_name=MODEL_EMBED)

# Example of embedding a single piece of text
sample_embedding = embeddings.embed_query("What is porosity?")
print(f"Embedding vector dimension: {len(sample_embedding)}")
```

Text with similar meanings will have vectors that are mathematically **"close"** to each other

This enables **semantic search**: finding documents by meaning, not just keywords

We'll use a fast, local model from Hugging Face for this

CLDiego/SPE_GeoHackathon_2025.git

ST-4: VECTOR STORE

A **Vector Store** is a specialized database designed for storing and efficiently searching millions of these vectors

.py

```python
from langchain_chroma import Chroma

# This single command does the heavy lifting:
# 1. It takes our document chunks.
# 2. It uses the embedding_model to convert each chunk to a vector.
# 3. It stores the chunks and their vectors in a local Chroma database.
# `persist_directory` saves the DB to disk for reuse.
vectorstore = Chroma.from_documents(
    documents=chunks,
    embedding=embedding_model,
    persist_directory="./local_geo_db"
)

print("Vector store created successfully!")
```

We use **Chroma**, an open-source vector database that runs locally

- The from_documents command automates the process: it takes our chunks, uses the embedding model to create vectors, and stores everything in the database
- persist_directory saves the database to disk so we don't have to rebuild it every time we run our code

CLDiego/SPE_GeoHackathon_2025.git

RETRIEVAL

The **Retriever** is the workhorse of the live RAG process. It connects to your Vector Store to find the most relevant documents for a given query.

.py

```python
# Create a retriever from our populated vector store.
# This object knows how to take a query string and find similar documents.
retriever = vectorstore.as_retriever(
    search_kwargs={"k": 5}  # "k" specifies how many documents to return.
)

# Let's test it! The retriever will embed the query and find the top 5
# most similar chunks from the vector store.
query = "What is the void space in a rock called?"
relevant_docs = retriever.invoke(query)

# The output is a list of Document objects.
print(f"Retrieved {len(relevant_docs)} documents.")
print("Top result:", relevant_docs[0].page_content[:100] + "...")
```

It takes the user's question, embeds it into a vector, and performs a similarity search in the Vector Store

- search_kwargs={"k": 5} tells the retriever to return the top 5 most relevant document chunks

CLDiego/SPE_GeoHackathon_2025.git

PROMPT AND GO

This is where we instruct the LLM. We design a specific **Prompt Template** that tells the LLM how to behave.

.py

```python
from langchain_core.prompts import ChatPromptTemplate

# This prompt template is the "instruction manual" for the LLM.
# It has placeholders for the retrieved context and the user's input.
RAG_PROMPT_TEMPLATE = """
SYSTEM: You are a helpful geoscience assistant.
Use the following context to answer the question. If you don't know the answer from the context, say so.

CONTEXT:
{context}

HUMAN:
{input}
"""

rag_prompt = ChatPromptTemplate.from_template(RAG_PROMPT_TEMPLATE)
```

It has two key placeholders:

- {context}: This is where we will inject the relevant document chunks found by the retriever
- {input}: This is where the user's original question will go

Crucially, we instruct the LLM to answer based **only** on the provided context, which reduces hallucinations and keeps the answer grounded in our data

CLDiego/SPE_GeoHackathon_2025.git

RAG CHAIN

.py

```python
from langchain_core.runnables import RunnablePassthrough
from langchain_core.output_parsers import StrOutputParser

# This function formats the retrieved documents into a single string.
def format_docs(docs):
    return "\\n\\n".join(doc.page_content for doc in docs)

# Now we build the full chain with LCEL.
# It reads like a data pipeline from left to right.
rag_chain = (
    # The input to the chain is the user's question.
    # This dictionary runs the retriever and passes the question through.
    {"context": retriever | format_docs, "input": RunnablePassthrough()}
    | rag_prompt          # The dictionary output is fed into our prompt.
    | llm                 # The formatted prompt is sent to the LLM.
    | StrOutputParser()   # The LLM's response is parsed into a string.
)

# Let's invoke the full chain!
answer = rag_chain.invoke("What is porosity?")
print(answer)
```

LangChain Expression Language (LCEL) lets us define the entire RAG data flow declaratively using the pipe | operator

It reads like a data pipeline: the user's question is passed to the **retriever**, the retrieved context is **formatted** and fed into the prompt, the prompt is sent to the **LLM**, and the final answer is **parsed** as a string

This approach is modular, readable, and makes it easy to see exactly how data flows through the system.

CLDiego/SPE_GeoHackathon_2025.git

TROUBLESHOOTING

.py

```python
# A helper function to see what the retriever is finding.
# This is the most important step for debugging your RAG system.
def preview_retrieval(query: str, k: int = 3):
    print(f"Previewing retrieval for query: '{query}'")
    # Use the retriever to get the most similar documents.
    docs = retriever.invoke(query, k=k)
    for i, doc in enumerate(docs, 1):
        source = doc.metadata.get("source", "Unknown")
        # Print a snippet of the retrieved content.
        print(f"  [{i}] From: {source}")
        print(f"      Content: {doc.page_content[:150]}...")
        print("-" * 20)

preview_retrieval("What is seismic resolution?")
```

The quality of your RAG system depends heavily on the **retrieval** step. **"Garbage in, garbage out."**

Use preview_retrieval to check:

- **Relevance:** Are the retrieved chunks actually relevant to the query?
- **Chunking Strategy:** Are the chunks too small (lacking context) or too big (too much noise)? Adjust chunk_size and chunk_overlap.
- **Number of Documents (k):** Is k too low (missing info) or too high (distracting the LLM)?

CLDiego/SPE_GeoHackathon_2025.git

ADVANCED AGENTS

SESSION-02

MODULE-02

PRESENTER

AI SPECIALIST

DR DIEGO CORONA-LOPEZ

CLDiego/SPE_GeoHackathon_2025.git

AGENDA

- **From Chains to Agents:** Grasp the leap from fixed pipelines to dynamic reasoning with the ReAct (Reason + Act) loop
- **Building Our Toolbox:** Package our RAG pipeline and a new SQL Data Analyst into callable tools
- **Guarded Execution:** Understand why safety is critical and how we prevent risky behavior from our agent's tools
- **The Master Agent:** Assemble a "router" agent that can intelligently choose the right tool for any given question
- **Hands-On**

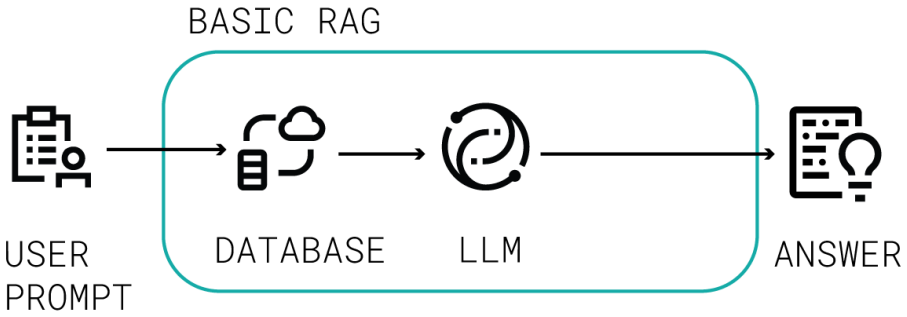
S2 / / M2

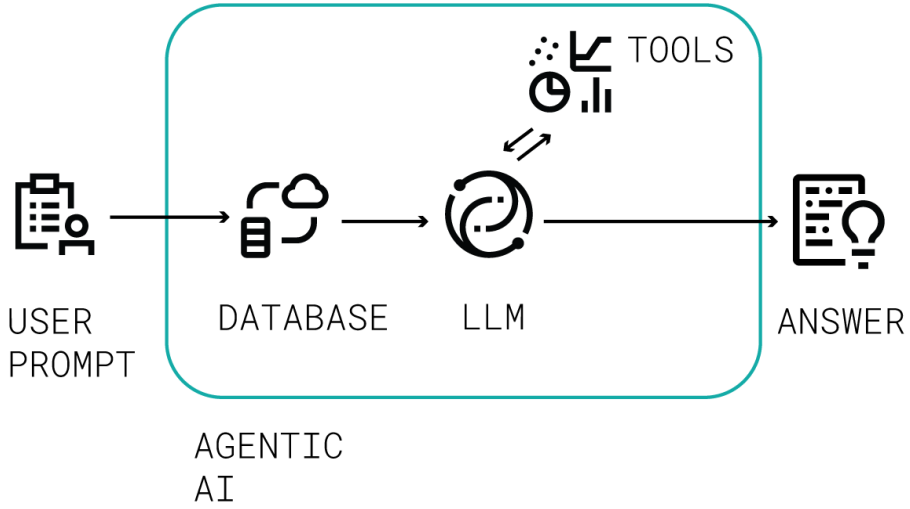
CLDiego/SPE_GeoHackathon_2025.git

FROM CHAINS TO AGENTS

Where We Are: We built a powerful RAG pipeline. It's excellent for answering questions *from our documents*.

The Limitation: What if a question requires a calculation from a csv file? Our RAG chain can't do math or query structured data.





The Goal: Build an **Agent**—a system where the LLM acts as a "reasoning engine" to choose the right tool for the job from a toolbox we provide.

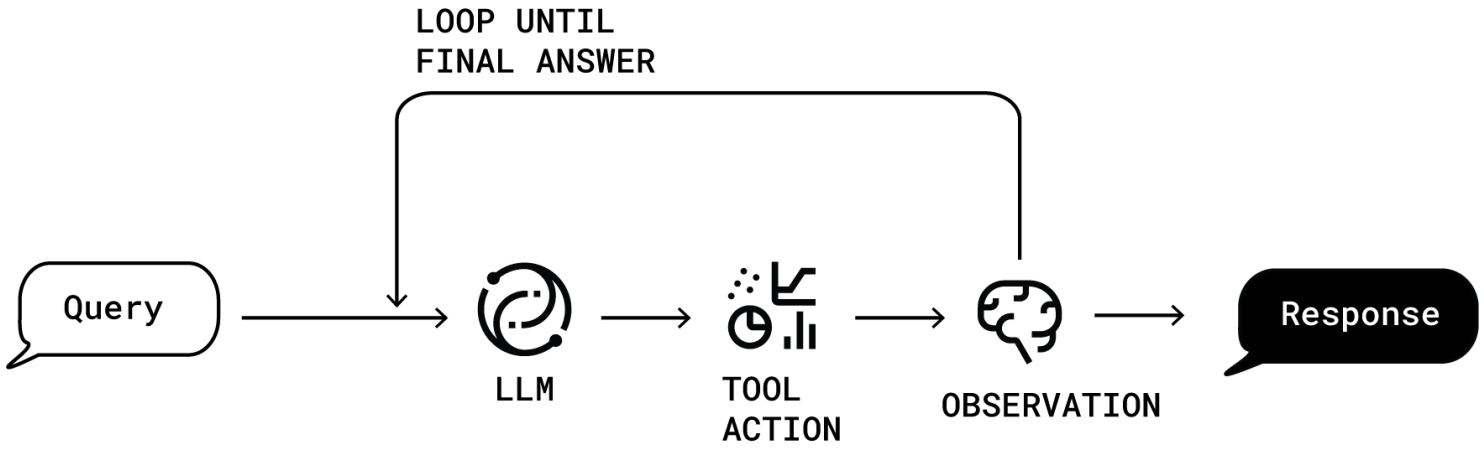
CLDiego/SPE_GeoHackathon_2025.git

PRESENTER → CORONA-LOPEZ

REACT LOOP

Agents operate on a loop, most commonly
ReAct (Reason + Act)

The LLM doesn't just answer; it verbalizes its thought process and decides on an action



The Loop:

1.Thought: The LLM analyzes the user's query and thinks, "To answer this, I need to calculate an average from the well data. I should use the **csv_sql_analyst tool**."

2.Action: The LLM decides to call a specific tool with specific inputs

3.Observation: The system executes the tool and gets a result (e.g., a table with the calculated average). The loop repeats until a final answer is synthesized

AGENT TOOLSET

.py

```python
from langchain.tools import tool

# Assume 'rag_chain' is the chain we built in S2_M1
@tool("geoscience_retriever")
def geoscience_retriever(query: str) -> str:
    """
    Use this tool for conceptual or qualitative geoscience questions about
    geology, geophysics, or reservoir engineering. It searches a
    specialized corpus of geoscience documents and provides citations.
    """
    try:
        result = rag_chain.invoke({"input": query})
        answer = result.get("answer", "No answer found.")
        # ... (citation logic from the notebook) ...
        return answer_with_citations
    except Exception as e:
        return f"Error searching documents: {e}"
```

An agent is only as good as its tools. We will equip our agent with a diverse set of capabilities.

We can wrap our entire RAG chain from Module 1 into a single, callable tool using the **@tool decorator.**

The docstring is critical! The agent reads it to understand what the tool does. We'll tell it to use this tool for conceptual questions about geoscience.

CLDiego/SPE_GeoHackathon_2025.git

ASSEMBLY

.py

```python
from langchain.agents import initialize_agent, AgentType

tools = [geoscience_retriever, csv_sql_analyst]

system_prompt = (
    "You are a geoscience assistant. Decide whether to use "
    "'geoscience_retriever' for concepts or 'csv_sql_analyst' for data."
)

agent_executor = initialize_agent(
    tools=tools,
    llm=chat_model,
    agent=AgentType.STRUCTURED_CHAT_ZERO_SHOT_REACT_DESCRIPTION,
    verbose=True, # Essential for debugging!
    handle_parsing_errors=True,
    agent_kwargs={"system_message": system_prompt}
)
```

Now we create the "master" agent. Its job is to **decide which tool to use** based on the user's question.

We use LangChain's initialize_agent function, providing it with our list of tools and a clear system prompt.

The AgentExecutor is the runtime that powers the agent's **ReAct loop**.

CLDiego/SPE_GeoHackathon_2025.git

GUARDED EXECUTION

What happens if the LLM tries to call a tool with malformed input? Or tries to execute dangerous code?

An agent with access to tools like a Python interpreter or a database connection is a potential security risk.

Guarded Execution means putting safety checks in place.

Our Approach:

- **Input Validation:** Use Pydantic or type hints in our tools (def my_tool(query: str)).
- **Output Parsing:** Use handle_parsing_errors=True in the AgentExecutor to gracefully handle when the LLM produces a malformed action.
- **Restricted Tools:** Only provide the agent with the specific, safe tools it needs. Avoid giving it a general-purpose Python or shell tool unless absolutely necessary and sandboxed.

TESTING THE AGENT

.py

```python
# Test 1: A conceptual question (should use RAG tool)
agent_executor.invoke({
    "input": "Explain the concept of seismic resolution."
})
# > Thought: The user is asking a conceptual geoscience question. I should use the geoscience_retriever tool.
# > Action: ...

# Test 2: A quantitative question (should use SQL tool)
agent_executor.invoke({
    "input": "What is the average water cut for Well-15/9-F-11-A?"
})
# > Thought: The user wants to calculate an average from the well data. I should use the csv_sql_analyst tool.
# > Action: ...
```

CLDiego/SPE_GeoHackathon_2025.git

TESTING THE AGENT

.py

```python
# Test 1: A conceptual question (should use RAG tool)
agent_executor.invoke({
    "input": "Explain the concept of seismic resolution."
})
# > Thought: The user is asking a conceptual geoscience question. I should use the geoscience_retriever tool.
# > Action: ...

# Test 2: A quantitative question (should use SQL tool)
agent_executor.invoke({
    "input": "What is the average water cut for Well-15/9-F-11-A?"
})
# > Thought: The user wants to calculate an average from the well data. I should use the csv_sql_analyst tool.
# > Action: ...
```

CLDiego/SPE_GeoHackathon_2025.git

Q&A

THANK YOU!

GitHub Repo: https://github.com/CLDiego/SPE_GeoHackathon_2025