🧠 Building AI Agent - Research Assistant with LangChain & Gemini

Type	@datasciencebrain

🚀 Overview

This assistant is designed for **multi-turn research conversations**, where the AI:

- Understands natural-language queries
- Fetches up-to-date info from **DuckDuckGo**
- Retrieves factual content from **Wikipedia**

- Summarizes everything using **Google Gemini (via LangChain)**
- Saves structured results to a local `.txt` file

This allows for repeatable, documented, and automated research workflows—ideal for bloggers, students, researchers, and AI experimenters.

📁 Project Structure

```
research-assistant/
│
├── main.py              # Runs the agent logic with chat loop
├── tools.py             # All external tools integrated (search, wiki, save)
├── .env                 # API keys used by the LLMs
├── requirements.txt     # Required libraries
└── research_output.txt  # Auto-generated summaries saved here
```

Each file serves a specific purpose:

- `main.py` handles the **conversation flow and reasoning**
- `tools.py` wraps up **actions the agent can perform**
- `.env` securely loads keys for **Gemini / OpenAI / Anthropic**

📦 `requirements.txt`

```
langchain
wikipedia
langchain-community
langchain-openai
langchain-anthropic
python-dotenv
pydantic
duckduckgo-search
```

To install:

```
pip install -r requirements.txt
```

💡 Tip: Use a virtual environment (`venv` or `conda`) for isolation.

🔑 Step 1: Environment Setup

LangChain requires API keys for LLMs. Create a `.env` file:

```
OPENAI_API_KEY = "your_openai_key"
ANTHROPIC_API_KEY = "your_anthropic_key"
GOOGLE_API_KEY = "your_google_gemini_key"
```

Then load them in your code:

```
from dotenv import load_dotenv
load_dotenv()
```

✅ This ensures credentials are never hardcoded and safe for version control (Git).

🛠️ Step 2: Tools Setup (`tools.py`)

LangChain supports external "tools" that the AI can *call* as needed. You define the tools and describe when they should be used.

---

1. 🌐 Web Search Tool (DuckDuckGo)

Used to fetch **recent or lesser-known** facts from the internet.

```
from langchain_community.tools import DuckDuckGoSearchRun
search = DuckDuckGoSearchRun()

search_tool = Tool(
    name="search",
    func=search.run,
```

```
    description="Search the web for information",
)
```

2. 📚 Wikipedia Tool

For authoritative, factual content on well-known topics.

```
from langchain_community.tools import WikipediaQueryRun
from langchain_community.utilities import WikipediaAPIWrapper

api_wrapper = WikipediaAPIWrapper(top_k_results=1, doc_content_chars_max
=100)

wiki_tool = WikipediaQueryRun(api_wrapper=api_wrapper)
```

3. 💾 Save-to-File Tool

A custom function that writes output to a `.txt` file, appending a timestamp.

```
from datetime import datetime

def save_to_txt(data: str, filename: str = "research_output.txt"):
    timestamp = datetime.now().strftime("%Y-%m-%d %H:%M:%S")
    formatted_text = f"--- Research Output ---\nTimestamp: {timestamp}\n\n{d
ata}\n\n"
    with open(filename, "a", encoding="utf-8") as f:
        f.write(formatted_text)
    return f"Data successfully saved to {filename}"
```

Exposed as a tool:

```
save_tool = Tool(
    name="save_text_to_file",
    func=save_to_txt,
    description="ALWAYS call this tool after generating a research summary. T
```

```
his tool saves your result.",
)
```

🤖 Step 3: Building the Agent (`main.py`)

LangChain agents work by combining:

- A prompt template (instructions + structure)
- An LLM (e.g., Gemini)
- External tools
- A loop to process input/output

🧾 Step 3.1: Output Schema with Pydantic

This defines the *structure* of the output we expect. Helps catch formatting errors.

```
from pydantic import BaseModel, Field
from typing import List

class AgentResponse(BaseModel):
    topic: str
    summary: str
    sources: List[str] = Field(default_factory=list)
    tools_used: List[str] = Field(default_factory=list)
```

🔮 Step 3.2: Gemini LLM via LangChain

```
from langchain_google_genai import ChatGoogleGenerativeAI

llm = ChatGoogleGenerativeAI(
    model="gemini-2.5-flash",
    temperature=0.1,
```

```
    max_output_tokens=1000,
)
```

- Low `temperature` = more factual
- `max_output_tokens` limits verbosity

🧠 Step 3.3: Prompt Template

A structured conversation scaffold:

```
from langchain_core.prompts import ChatPromptTemplate

prompt = ChatPromptTemplate.from_messages(
    [
        (
            "system",
            """
            You are a research assistant that generates detailed summaries using s
earch and Wikipedia tools.
            After you complete your answer, you MUST call the tool named `save_t
ext_to_file` to store the result in a file. Never skip this step.

            Always output ONLY a valid JSON matching this schema:
            {format_instructions}
            Do not include any explanation or extra text.
            """,
        ),
        ("placeholder", "{chat_history}"),
        ("human", "{query}"),
        ("placeholder", "{agent_scratchpad}"),
    ]
).partial(format_instructions=parser.get_format_instructions())
```

LangChain fills `{chat_history}` and `{agent_scratchpad}` automatically.

🤖 Step 3.4: Agent Creation & Executor

```
from langchain.agents import create_tool_calling_agent, AgentExecutor

tools = [search_tool, wiki_tool, save_tool]

agent = create_tool_calling_agent(
    llm=llm,
    prompt=prompt,
    tools=tools,
)

agent_executor = AgentExecutor(
    agent=agent,
    tools=tools,
    verbose=True,
)
```

The **executor** handles the entire logic: prompt → reasoning → tool calls → result.

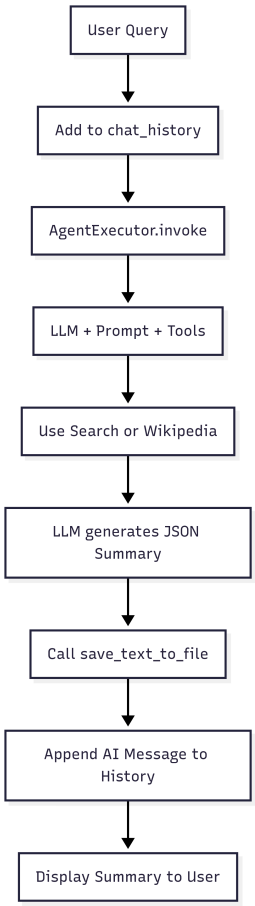
🔁 Step 4: Multi-turn Chat Loop

```
chat_history = []

while True:
    query = input("You: ")
    if query.lower() in ["exit", "quit"]:
        break

    chat_history.append(HumanMessage(content=query))

    response = agent_executor.invoke(
        {
            "query": query,
            "chat_history": chat_history,
```

```
        }
    )

    try:
        structured_response = parser.parse(response.get("output"))
        print("\nAI:", structured_response.summary)
        chat_history.append(AIMessage(content=structured_response.summar
y))
    except Exception as e:
        print("Error parsing:", e)
        print("Raw output:", response.get("output"))
```

- Maintains **conversation memory**
- Outputs **summaries**
- Saves each result to a file via `save_tool`

🧭 Agent Workflow Diagram



✅ Final Output Example (in `research_output.txt`)

```
--- Research Output ---
Timestamp: 2025-07-06 13:22:12

Topic: LangChain Wikipedia vs DuckDuckGo
Summary: Wikipedia provides detailed knowledge on well-known topics; Duck
DuckGo helps find recent and lesser-known info.
Sources: ["https://en.wikipedia.org/...", "https://duckduckgo.com/..."]
Tools Used: ["search", "wiki_tool"]
```

📌 Notes & Ideas

- 🎯 **Extendable**: Add PDF readers, code runners, CSV parsers, database tools, etc.
- 🔐 **Secure**: All credentials are hidden in `.env`
- 📊 **Structured**: JSON output can be easily converted to HTML or used in dashboards

---

✅ Next Steps (Optional Enhancements)

Feature	Benefit
Add memory (LangChain)	Maintain context beyond chat loop
Add vector search	Search your own documents / PDFs
Add UI (Streamlit)	Make it interactive & visual
Export to JSON/CSV	For data analysis / integration
Add voice input	Hands-free research (e.g., with SpeechRecognition)

```
#main.py

from dotenv import load_dotenv
from pydantic import BaseModel, Field
from langchain_google_genai import ChatGoogleGenerativeAI
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.output_parsers import PydanticOutputParser
from langchain.agents import create_tool_calling_agent, AgentExecutor
from tools import search_tool, wiki_tool, save_tool  # Your tools.py should export
from langchain_core.messages import HumanMessage, AIMessage
from typing import List

load_dotenv()

# Robust AgentResponse with Pydantic defaults
class AgentResponse(BaseModel):
    topic: str
    summary: str
```

```
    sources: List[str] = Field(default_factory=list)
    tools_used: List[str] = Field(default_factory=list)

llm = ChatGoogleGenerativeAI(
    model="gemini-2.5-flash",
    temperature=0.1,
    max_output_tokens=1000,
)

parser = PydanticOutputParser(pydantic_object=AgentResponse)

prompt = ChatPromptTemplate.from_messages(
    [
        (
            "system",
            """
            You are a research assistant that generates detailed summaries using sea
            After you complete your answer, you MUST call the tool named `save_text

            Always output ONLY a valid JSON matching this schema:
            {format_instructions}
            Do not include any explanation or extra text.
            """,
        ),
        ("placeholder", "{chat_history}"),
        ("human", "{query}"),
        ("placeholder", "{agent_scratchpad}"),
    ]
).partial(format_instructions=parser.get_format_instructions())

tools = [search_tool, wiki_tool, save_tool]

agent = create_tool_calling_agent(
    llm=llm,
    prompt=prompt,
    tools=tools,
```

```
)

agent_executor = AgentExecutor(
    agent=agent,
    tools=tools,
    verbose=True,
)

# ---- Multi-turn chat loop ----

chat_history = []

print("AI Research Assistant (type 'exit' to quit)\n")

while True:
    query = input("You: ")
    if query.lower() in ["exit", "quit"]:
        print("Exiting chat. Goodbye!")
        break

    # Add user message to chat_history as a HumanMessage
    chat_history.append(HumanMessage(content=query))

    # Invoke agent with chat history as a list of messages
    response = agent_executor.invoke(
        {
            "query": query,
            "chat_history": chat_history,
        }
    )

    # Robust output parsing & fallback
    try:
        structured_response = parser.parse(response.get("output"))
        print("\nAI:", structured_response.summary)
        # Add AI response to chat_history as an AIMessage
```

```
        chat_history.append(AIMessage(content=structured_response.summary))
    except Exception as e:
        print(f"\n[Warning] Error parsing response as JSON: {e}")
        print("[Raw output was]:", response.get("output"))
        # Optionally, append raw output so context isn't lost
        chat_history.append(AIMessage(content=response.get("output")))
```

```
#tools.py

from langchain_community.tools import WikipediaQueryRun, DuckDuckGoSearch
from langchain_community.utilities import WikipediaAPIWrapper
from langchain.tools import Tool
from datetime import datetime
import os

def save_to_txt(data: str, filename: str = "research_output.txt"):
    print("Current working directory:", os.getcwd())
    print("Saving the following data:", data)
    timestamp = datetime.now().strftime("%Y-%m-%d %H:%M:%S")
    formatted_text = f"--- Research Output ---\nTimestamp: {timestamp}\n\n{data
    with open(filename, "a", encoding="utf-8") as f:
        f.write(formatted_text)
    return f"Data successfully saved to {filename}"

save_tool = Tool(
    name="save_text_to_file",
    func=save_to_txt,
    description="ALWAYS call this tool after generating a research summary. This
)

search = DuckDuckGoSearchRun()
search_tool = Tool(
    name="search",
    func=search.run,
```

```
    description="Search the web for information",
)

api_wrapper = WikipediaAPIWrapper(top_k_results=1, doc_content_chars_max=1
wiki_tool = WikipediaQueryRun(api_wrapper=api_wrapper)
```