

Lab Title: Build a Multi-Agent System with LangGraph, Tools, and Memory

Prerequisites

1. Python 3.9+ environment
2. OpenAI API key (or any LLM like Mistral, Anthropic via LangChain)
3. Install required packages:

```
pip install langgraph langchain openai faiss-cpu duckduckgo-search
```

Step 1: Understand the Agent Roles

- **Planner Agent:** Interprets the user goal and breaks it down into subtasks.
 - **Retriever Agent:** Queries search tools or vector stores to fetch information.
 - **Summarizer Agent:** Synthesizes all retrieved content into a final output.
-

Step 2: Setup the LangGraph Environment

Create a new Python file (e.g., `multi_agent_langgraph.py`) and import core libraries:

```
from langgraph.graph import StateGraph
from langchain.agents import Tool
from langchain.memory import ConversationBufferMemory
from langchain.chat_models import ChatOpenAI
from langchain.tools import DuckDuckGoSearchRun
from langchain.prompts import PromptTemplate
```

Step 3: Create Shared Memory

```
memory = ConversationBufferMemory(return_messages=True)
```

This memory stores agent interactions, accessible across agents.

Step 4: Define Tools for the Retriever

```
search = DuckDuckGoSearchRun()
search_tool = Tool(
    name="WebSearch",
```

```
func=search.run,  
description="Useful for answering general knowledge queries"  
)
```

You can later expand this with tools like calculator, code executor, vector DBs, etc.

Step 5: Define the LLM for Each Agent

```
llm = ChatOpenAI(temperature=0)
```

This shared LLM instance will be wrapped in each agent function with a specific prompt.

Step 6: Define Agent Prompts and Functions

Planner Agent

```
from langchain.chains import LLMChain  
  
planner_prompt = PromptTemplate.from_template(  
    "You are a planner agent. Break down the task: {input}  
    into a list of steps."  
)  
planner_chain = LLMChain(llm=llm, prompt=planner_prompt)  
  
def planner_node(state):  
    steps = planner_chain.run(state["input"])  
    return {"planner_output": steps, "step": "retrieve"}
```

Retriever Agent

```
retriever_prompt = PromptTemplate.from_template(  
    "Use the web search to find info on: {planner_output}"  
)  
retriever_chain = LLMChain(llm=llm, prompt=retriever_prompt)  
  
def retriever_node(state):  
    query = state["planner_output"]  
    result = search.run(query)  
    return {"retrieved_info": result, "step": "summarize"}
```

Summarizer Agent

```
summarizer_prompt = PromptTemplate.from_template(  
    "Summarize the following information for the user: {retrieved_info}"  
)  
summarizer_chain = LLMChain(llm=llm, prompt=summarizer_prompt)
```

```
def summarizer_node(state):
    final_output = summarizer_chain.run(state["retrieved_info"])
    return {"final_output": final_output, "step": "end"}
```

Step 7: Build the LangGraph

```
graph = StateGraph()
graph.add_node("planner", planner_node)
graph.add_node("retrieve", retriever_node)
graph.add_node("summarize", summarizer_node)

graph.set_entry_point("planner")
graph.add_edge("planner", "retrieve")
graph.add_edge("retrieve", "summarize")
graph.add_edge("summarize", "end")

app = graph.compile()
```

Step 8: Run the Multi-Agent System

```
response = app.invoke({"input": "Tell me what's new about Mars exploration
this month."})
print("\n✅ Final Output:\n", response["final_output"])
```

Optional Enhancements

- Add **Graph memory** to keep track of multi-turn steps
- Use **LangChain Expression Language (LCEL)** for node logic
- Integrate **RAG retriever** instead of DuckDuckGo for structured KBs
- Log each step to build observability into the graph

Outcome

You’ve now built a **fully functional LangGraph multi-agent pipeline** with:

- Distinct agent roles
- Web search as a tool
- Memory buffer across agents
- Deterministic step-by-step execution