Lab: Building an Agentic Travel Planning Assistant

Using LangChain and External Tools via MCP

Contents

1 Lab Overview

Title: Agentic AI with External Tools – Travel Planning Assistant

1.1 Prerequisites

- Python fundamentals
- Basic understanding of Large Language Models (LLMs)
- Introductory familiarity with LangChain concepts

1.2 Learning Objectives

By the end of this lab, students will be able to:

- Explain the concept of **agentic AI**
- Describe the role of the **Model Context Protocol (MCP)**
- Build an agent using LangChain that selects tools autonomously
- Connect agents to **external MCP servers**
- Implement a graphical user interface (GUI) for an agentic system

2 Use Case: Agentic Travel Planning Assistant

Planning a trip requires gathering information from multiple sources: destinations, budgets, dates, and constraints. In this lab, students will build an **agentic travel planning assistant** capable of reasoning about a user request and invoking external tools via MCP.

2.1 Example User Request

> "Plan a 5-day trip to Barcelona with an estimated budget and suggested activities."

The agent must:

- Interpret the request
- Decide which tools are needed
- Invoke external tools
- Synthesize a final travel plan

3 Model Context Protocol (MCP)

The **Model Context Protocol (MCP)** is a standardized way for AI agents to:

- Discover tools dynamically
- Call external services in a uniform manner
- Remain decoupled from tool implementations

In this lab:

- Tools run as independent MCP servers
- LangChain agents consume these tools at runtime

4 System Architecture

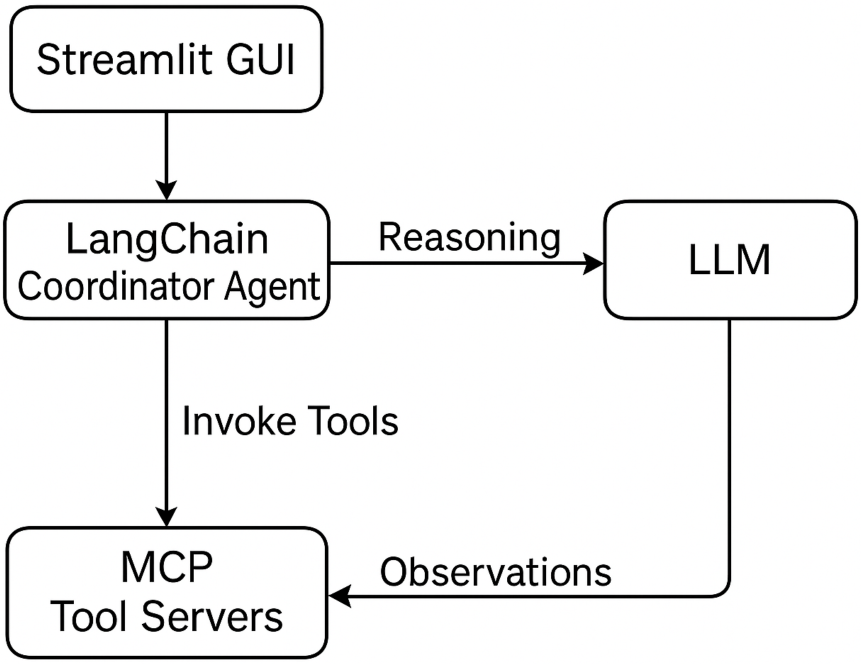

Figure 1: Agentic Travel Planning Assistant Architecture using LangChain and MCP

5 Agents and Tools

5.1 Coordinator Agent

The coordinator agent is responsible for:

- Understanding the user request
- Selecting appropriate tools
- Aggregating tool outputs into a final answer

5.2 External Tools (via MCP)

In addition to destination search and budget estimation, students must extend the agent with the following **mandatory external tools** exposed via MCP servers.

Tool Name	Purpose	MCP Server
Destination Search	Retrieves tourist attractions, landmarks, and activities for a given destination.	travel-search-mcp
Budget Calculator	Estimates total travel cost based on destination and number of days.	finance-mcp
Weather Tool	Provides typical or forecasted weather conditions for the travel dates, used to adapt activities.	weather-mcp
Currency Converter	Converts estimated travel costs into the user's preferred currency.	currency-mcp
Calculator Tool	Performs arithmetic operations required during agent reasoning.	calculator-mcp

Table 1: Mandatory external tools accessed by the agent via MCP

The agent must decide autonomously:

- When weather information is relevant (e.g., outdoor vs indoor activities)
- When currency conversion is required
- When explicit calculation is necessary instead of estimation

6 Technology Stack

- Python 3.10+
- LangChain
- MCP Python SDK
- Ollama or OpenAI-compatible LLM
- Streamlit for GUI

7 Implementation Steps

7.1 Step 1: Environment Setup

```
python -m venv venv
source venv/bin/activate
pip install langchain langchain-community streamlit mcp fastapi uvicorn
```

7.2 Step 2: MCP Tool Server Example

Budget Calculator MCP Server

```
from mcp.server.fastapi import MCPServer

server = MCPServer("budget-tools")

@server.tool()
```

```
def estimate_budget(destination: str, days: int) -> float:
"""Estimate travel budget in USD."""
base_cost = 100
return base_cost * days

server.run(port=3333)
```

Run the server:

```
python budget_mcp_server.py
```

7.3 Step 3: Connecting LangChain to MCP

```
from langchain_community.tools.mcp import MCPToolkit

mcp_toolkit = MCPToolkit.from_server(
server_url="[http://localhost:3333](http://localhost:3333)"
)

tools = mcp_toolkit.get_tools()
```

7.4 Step 4: Creating the LangChain Agent

```
from langchain.agents import create_react_agent
from langchain_openai import ChatOpenAI

llm = ChatOpenAI(model="gpt-4o-mini")

agent = create_react_agent(
llm=llm,
tools=tools,
prompt="""
You are a travel planning agent.
You can search destinations, estimate budgets, and propose itineraries.
Use tools when needed.
"""
)
```

7.5 Step 5: Agent Execution

```
def run_travel_agent(user_request: str):
response = agent.invoke({"input": user_request})
return response["output"]
```

8 GUI Implementation (Streamlit)

```
import streamlit as st

st.title("Agentic Travel Planner (MCP)")

query = st.text_input("Describe your trip")

if st.button("Plan My Trip"):
```

```
output = run_travel_agent(query)
st.subheader("Travel Plan")
st.write(output)
```

Run the GUI:

```
streamlit run app.py
```

9 Exercises

1. Display each tool call in the GUI
2. Add a critic agent to validate itineraries
3. Enforce a budget constraint and re-plan automatically

10 Discussion Questions

- Why is MCP preferable to hard-coded tool calls?
- Where does autonomy emerge in this system?
- What are the security implications of tool-using agents?

11 Extensions

- Add memory to the agent
- Deploy MCP servers using Docker
- Convert to a planner–executor–critic architecture