Telecom KPI Agent: Setup and Workflow Documentation

By: Chetan Sai Borra

The agent is built using LangChain, NVIDIA LLMs, and LangGraph to provide intelligent responses to telecom KPI queries by combining internal analytics with external web search.

As openAi and claude API's are expensive we use nivdia API for prototyping and the agent workflow would be same for other llm's as well.

1. Agent Setup Overview:

Libraries and Tools:

- langchain_nvidia_ai_endpoints.ChatNVIDIA – Interface to NVIDIA-hosted large language models (LLMs).
- langchain_tavily.TavilySearch – Tool for answering open-ended or causal questions using real-time web search.
- ToolNode – A graph node that handles tool execution in LangGraph.
- RunnableLambda – Wraps a custom function (lambda) as a LangChain Runnable.

2. LLM and Tool Binding:

llm = ChatNVIDIA(model="meta/llama-3.1-70b-instruct", streaming=True, api_key=nvidia_key)

search_tool = TavilySearch(max_results=3, tavily_api_key=tavily_key)

I define a powerful **70B LLaMA-3.1 model** from Meta hosted via NVIDIA NIM and bind it with tools like:

- get_site_kpi_extreme
- get_peak_kpi_day_for_site
- compare_kpi_impact
- describe_kpi_dataset
- kpi_anomalies
- search_tool (Tavily)

Then I bind all tools:

llm_with_tools = llm.bind_tools(all_tools)

3. LangGraph Workflow (State Machine):

LangGraph is used to control multi-step agent logic via a finite state machine.

States:

- "agent": The LLM receives messages, decides what to do.
- "tools": If a tool is required, it runs here (via ToolNode).

Logic:

```
def should_continue(state):
    last = state["messages"][-1]
    return "tools" if getattr(last, "tool_calls", None) else "__end__"
```

- If the LLM calls a tool → go to tools node.
- Otherwise → stop (__end__).

Graph Definition:

```
workflow = StateGraph(MessagesState)
workflow.add_node("agent", call_model)
workflow.add_node("tools", tool_node)
workflow.set_entry_point("agent")
workflow.add_conditional_edges("agent", should_continue)
workflow.add_edge("tools", "agent")
```

This creates a loop:
→ LLM (agent) → tools if needed → back to LLM with tool output → stop when no more tool calls.

4. Agent Execution:

```
agent_executor: Runnable = RunnableLambda(lambda x: app.invoke({...}))
```

This lambda:

- Initializes the conversation with a SystemMessage that sets the agent's personality and domain.

- Adds any chat_history.
- Appends the new HumanMessage (user question).
- Invokes the LangGraph app.

MCP Server (mcp_server.py): API Wrapper for Telecom Agent

This script sets up a FastAPI server to expose the telecom KPI agent (agent_executor) over HTTP.

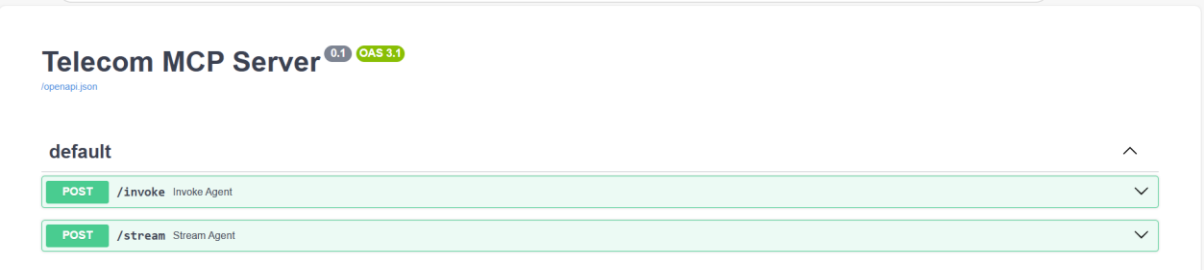
Features Provided

1. Standard Chat Endpoint
   /invoke — handles normal, single-turn message queries.
2. Streaming Chat Endpoint
   /stream — supports real-time streaming of agent responses.

1. FastAPI App Setup:

app = FastAPI(title="Telecom MCP Server", version="0.1")

- Defines a REST API named *Telecom MCP Server*.
- FastAPI provides automatic Swagger docs at /docs.



2. Pydantic Models:

class ChatInput(BaseModel):

    input: str

    chat_history: List[Dict] = Field(default=[])

class ChatOutput(BaseModel):

    output: str

- **ChatInput** expects:

- input: the user question.
- chat_history: previous messages for memory.

- **ChatOutput** returns the final agent output as a string.

3. POST /invoke: Normal Response:

```
@app.post("/invoke", response_model=ChatOutput)
def invoke_agent(input: ChatInput):
    ...
    return {"output": last_ai.content if last_ai else " No response generated."}
```

- Calls the agent executor with input and history.
- Extracts the most recent AI message (of type ai) and returns it.

4. Execution Entry Point:

```
if __name__ == "__main__":
    uvicorn.run("MCP_server:app", host="0.0.0.0", port=8000, reload=True)
```

- Starts a local development server on port 8000.
- Enables auto-reload during development (reload=True).

Frontend: Gradio Chat Interface (app.py)

Purpose: Provides a lightweight web UI for interacting with the Telecom KPI conversational AI agent using a chat interface.

1. API Integration

```
API_URL = "http://localhost:8000/invoke"
```

- Sends POST requests to the FastAPI server (must run MCP_server.py first).

2. Global Chat History

```
chat_history = []
```

- Keeps track of the conversation using LangChain's HumanMessage and AIMessage.

3. chat() Function

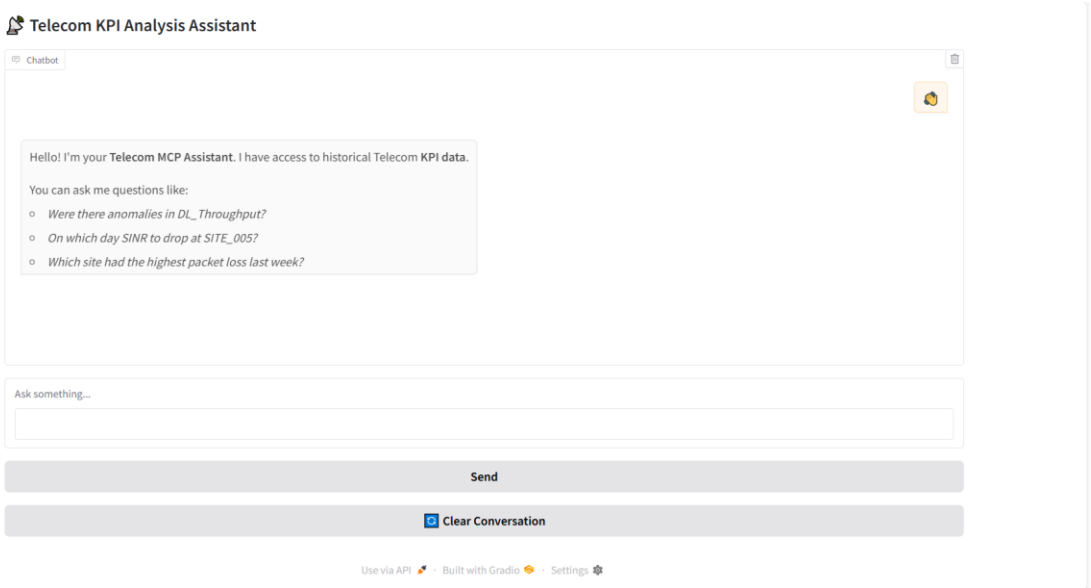
```
def chat(user_input):
    ...
    response = requests.post(API_URL, json=payload)
```

- Converts chat history to JSON.
- Sends user input + history to the backend.
- Receives model response and updates history.

4. Gradio Layout

with gr.Blocks() as demo:

- Markdown: Title
- Chatbot: Preloaded with welcome message
- Textbox: For user input
- Send Button: Calls respond
- Clear Button: Clears history



Thank You!