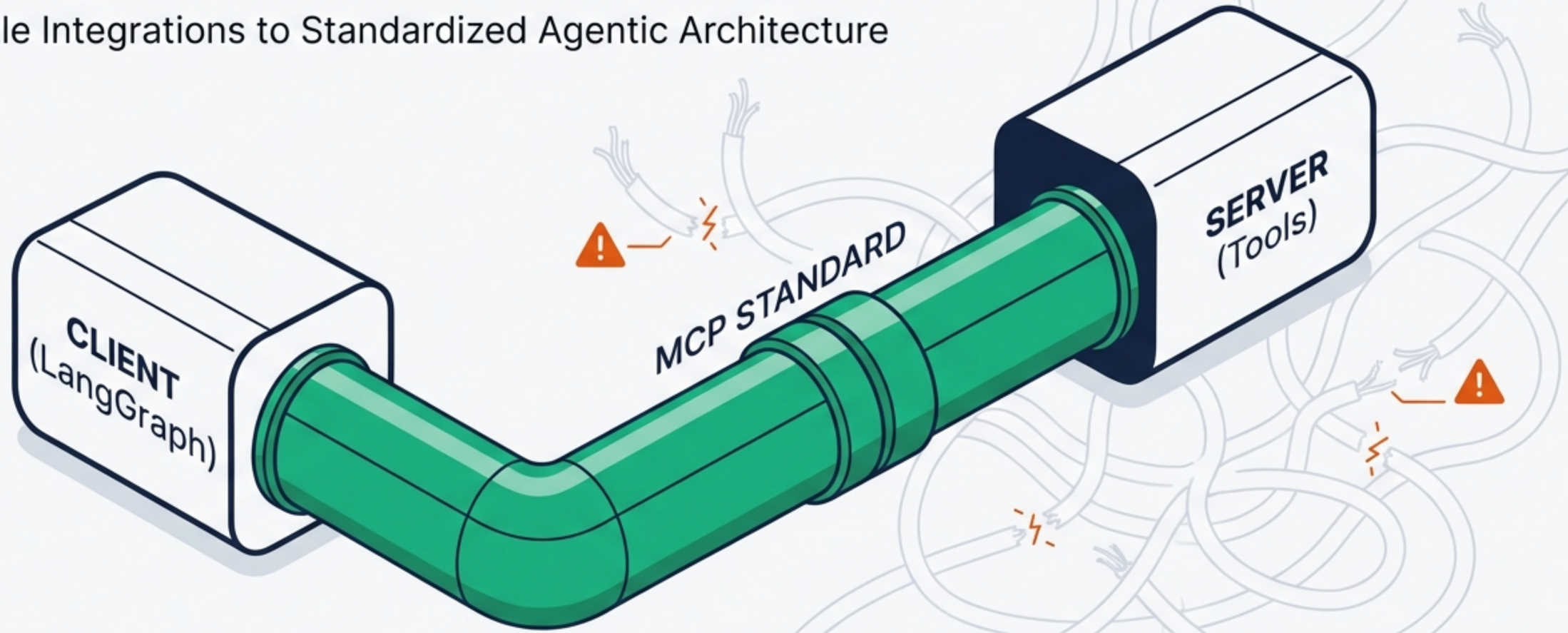
AGENTIC AI SERIES: MODULE 19

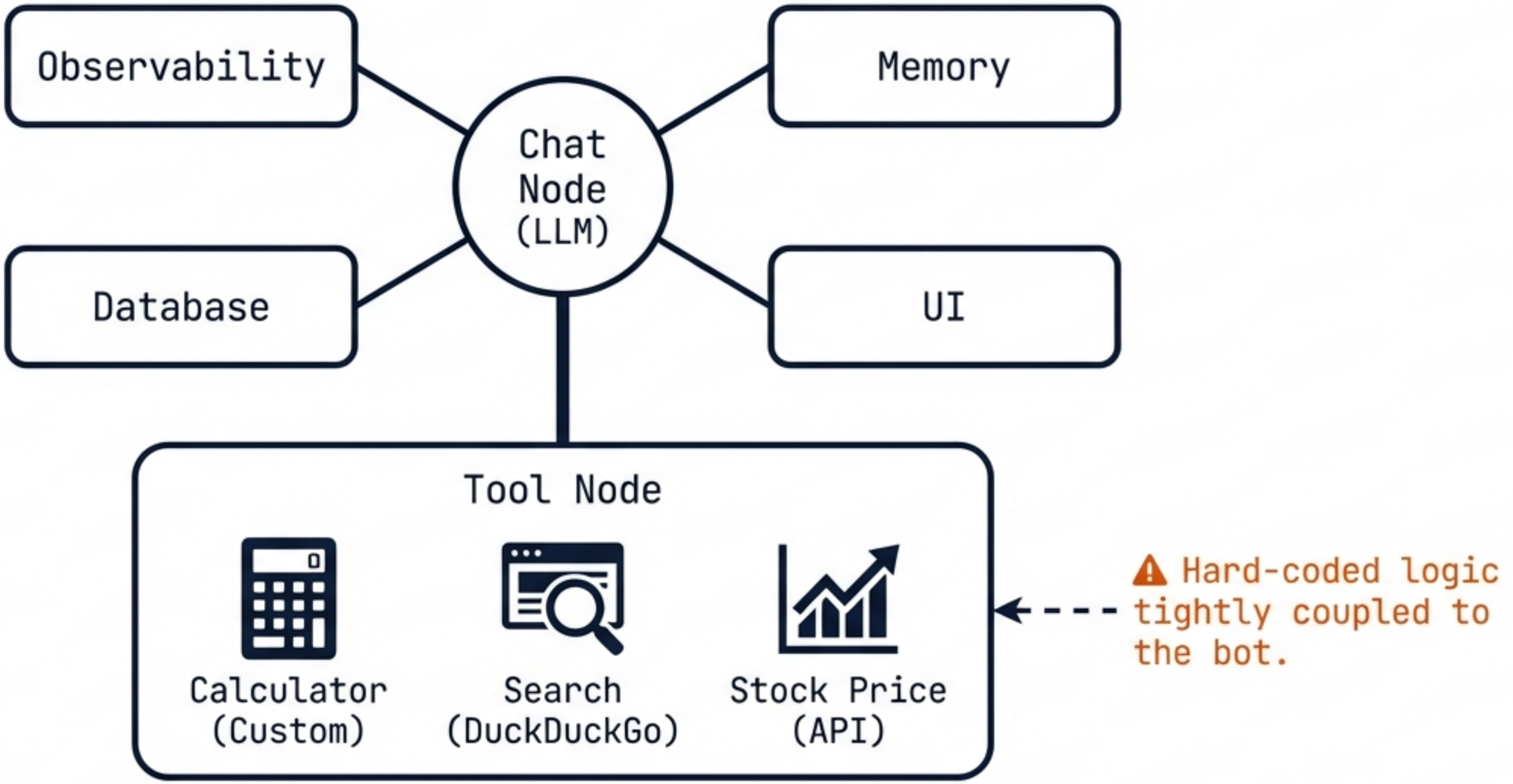
Escaping the Tool Trap: Building MCP Clients in LangGraph

From Brittle Integrations to Standardized Agentic Architecture



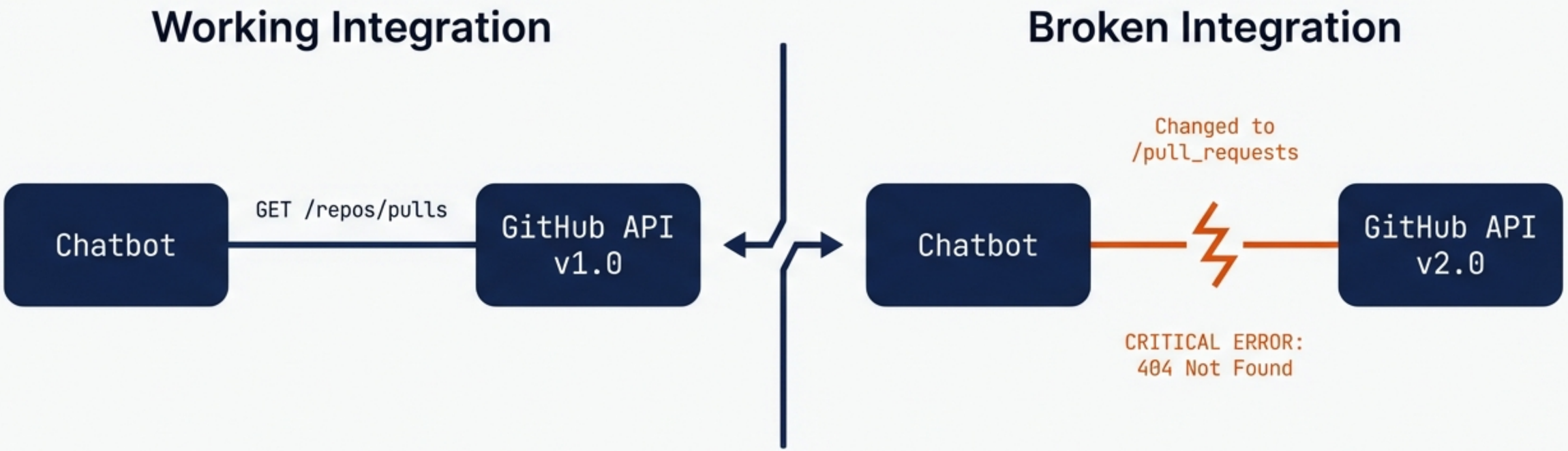
Presented by CampusX

The Status Quo: A Capable but Flawed Agent



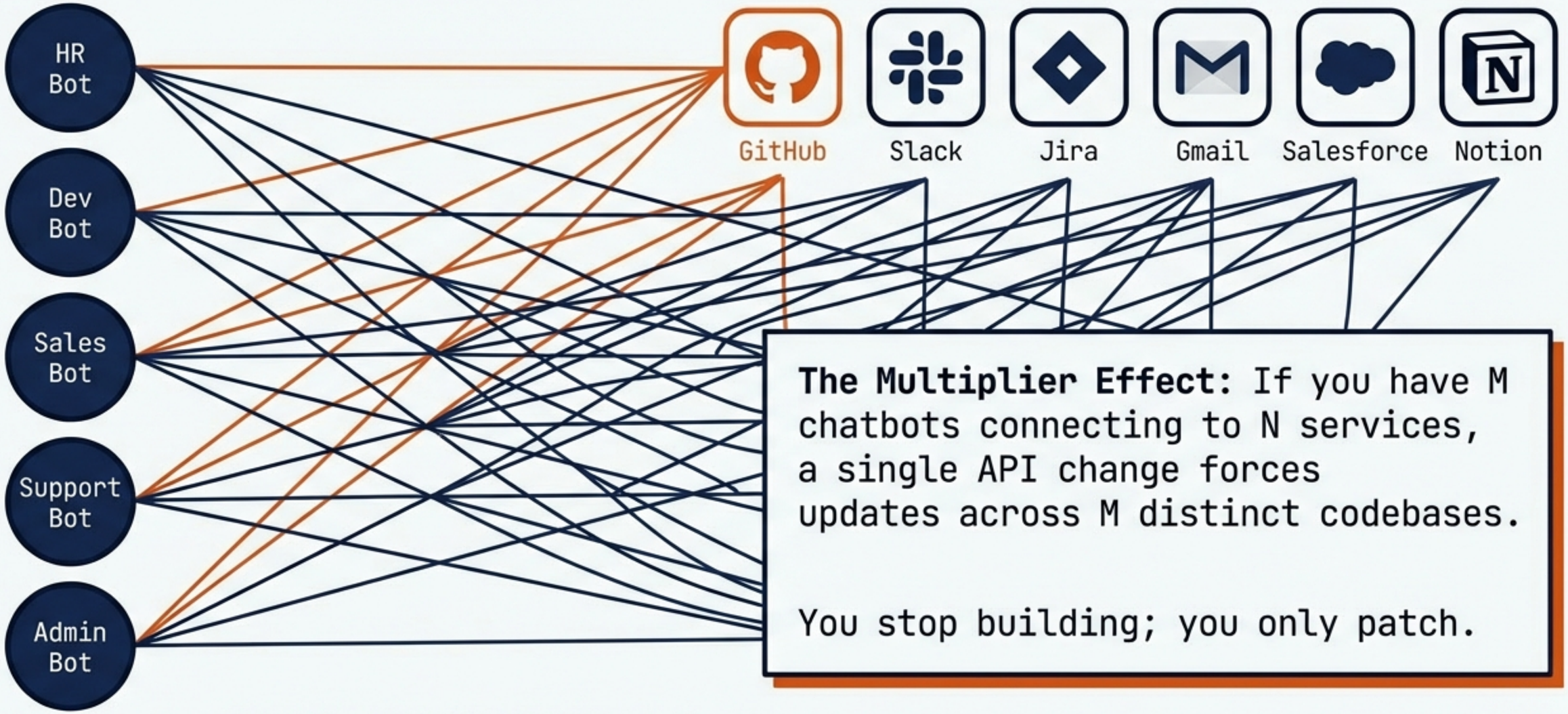
Current State: We have a functional agent with memory and persistence. However, the method of connecting tools is manual and brittle.

The Problem: Brittle Integrations

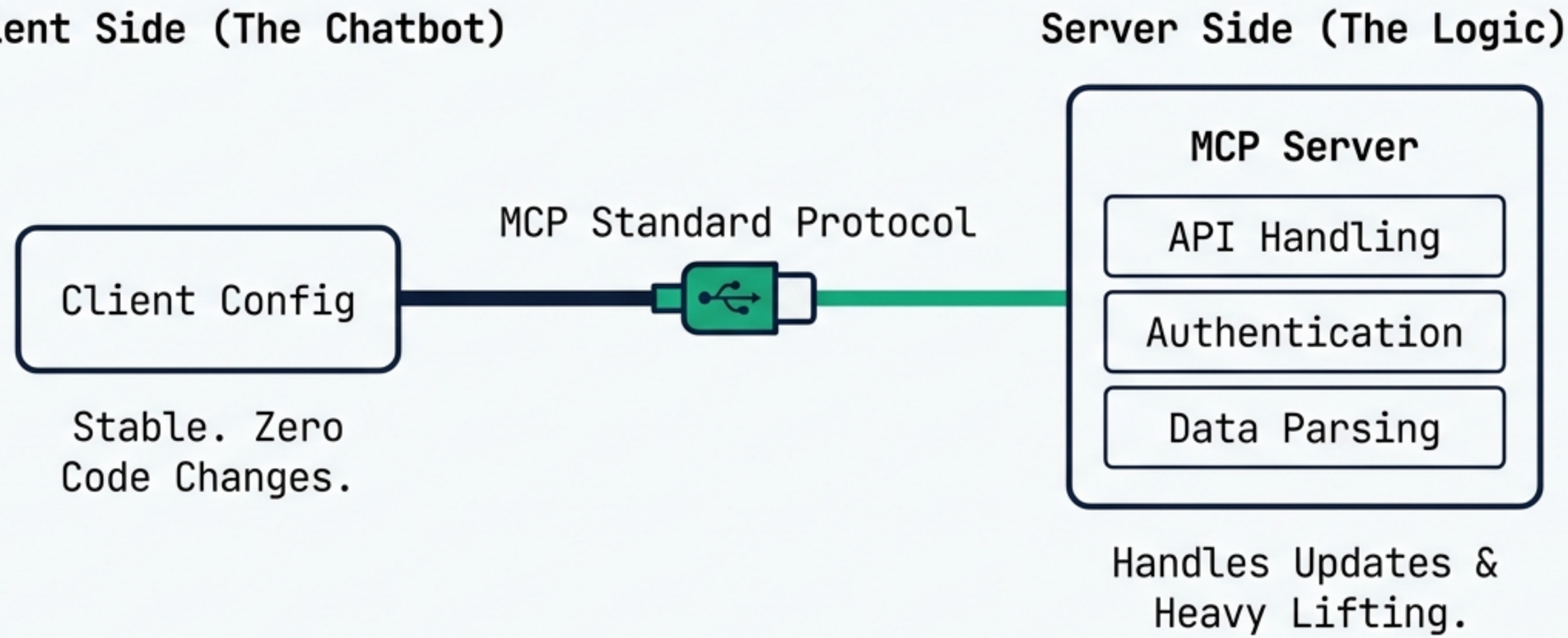


The GitHub Scenario: When a service provider updates their API, hard-coded tools inside the chatbot crash instantly. Constant monitoring is required.

The Maintenance Nightmare: The N × M Problem



The Solution: Model Context Protocol (MCP)



Separation of Concerns: The Client only asks "What can you do?". The Server handles "How I do it." If the API changes, only the Server updates.

The Prerequisite: Shifting to Async

Synchronous Flow (Greyed Out)

Asynchronous Flow (Highlighted Blue)

Constraint: `langchain-mcp-adapters` is an Async-only library.

Action: We must refactor LangGraph from `invoke` to `ainvoke`.

Benefit: Parallel execution and non-blocking I/O.

Refactoring for Async: The Code Transformation

Code Diff

Before (Sync)

```
def main():
    result = graph.invoke(inputs)
    print(result)


# Node Definition
def chat_node(state):
    return llm.invoke(state)
```

After (Async)

```
import asyncio

async def main():
    result = await graph.ainvoke(inputs)
    print(result)

# Node Definition
async def chat_node(state):
    return await llm.ainvoke(state)
```

Note: LangGraph's built-in ToolNode is natively async compatible. Only custom nodes need manual refactoring.

Implementation: The Multi-Server MCP Client

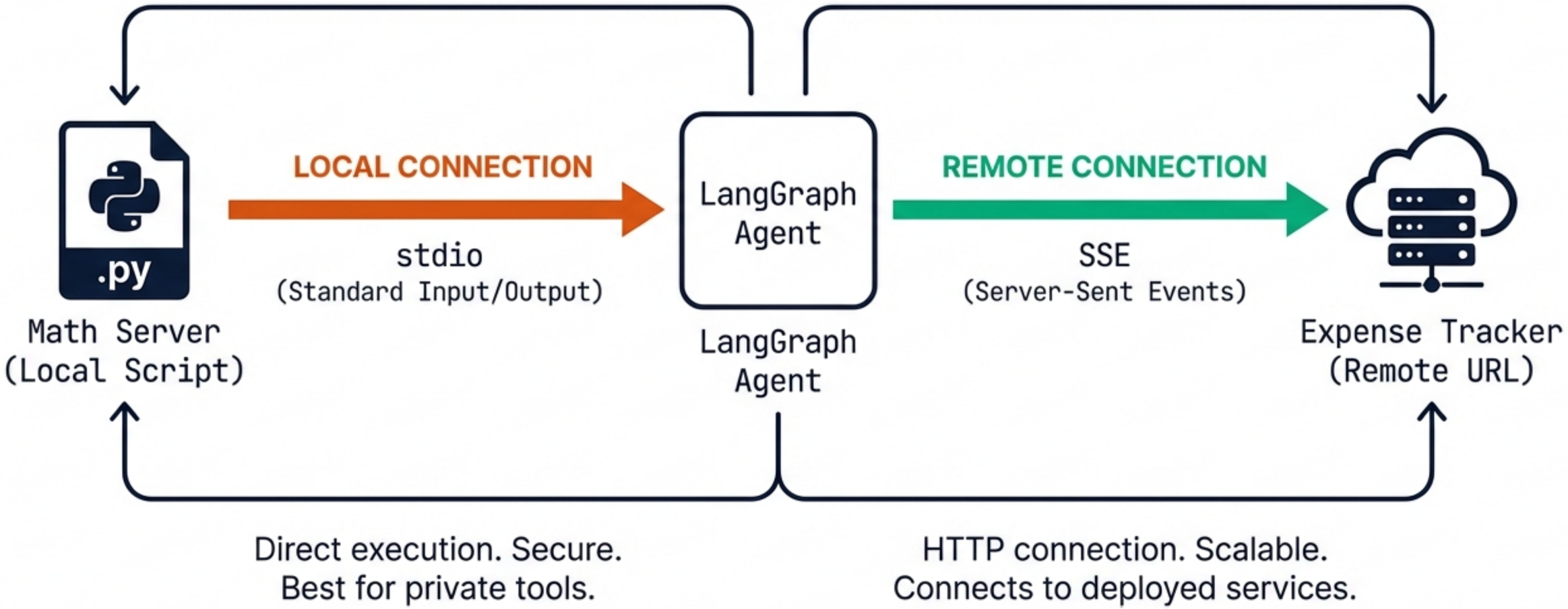
```python
from langchain_mcp_adapters.client import MultiServerMCPClient

# Initialize the Client
client = MultiServerMCPClient(
    servers={
        "math_server": {
            "command": "python",
            "args": ["path/to/math_server.py"],
            "transport": "stdio"
        }
    }
)

# Dynamic Tool Binding
tools = await client.get_tools()
llm_with_tools = llm.bind_tools(tools)
```

The adapter library that bridges LangGraph and MCP.

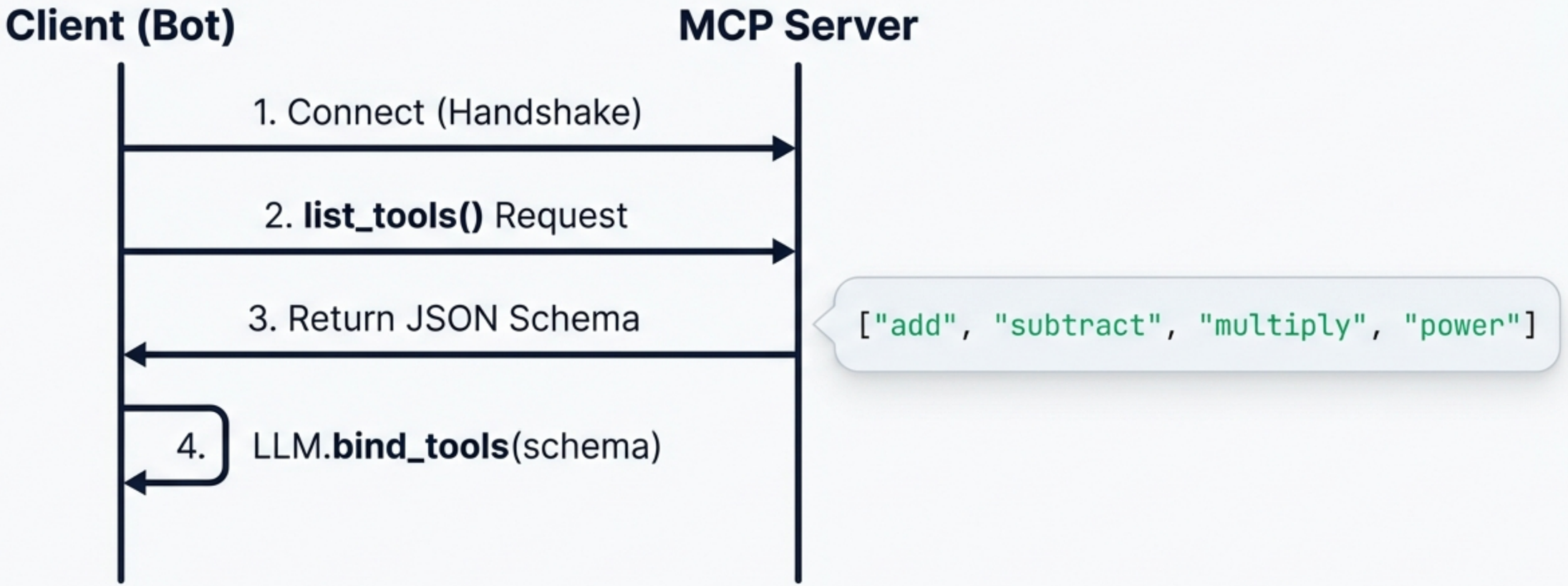
Defining the server config. No tool logic here.

Runtime discovery. The bot asks the server what it can do.

Connection Protocols: Local vs. Remote

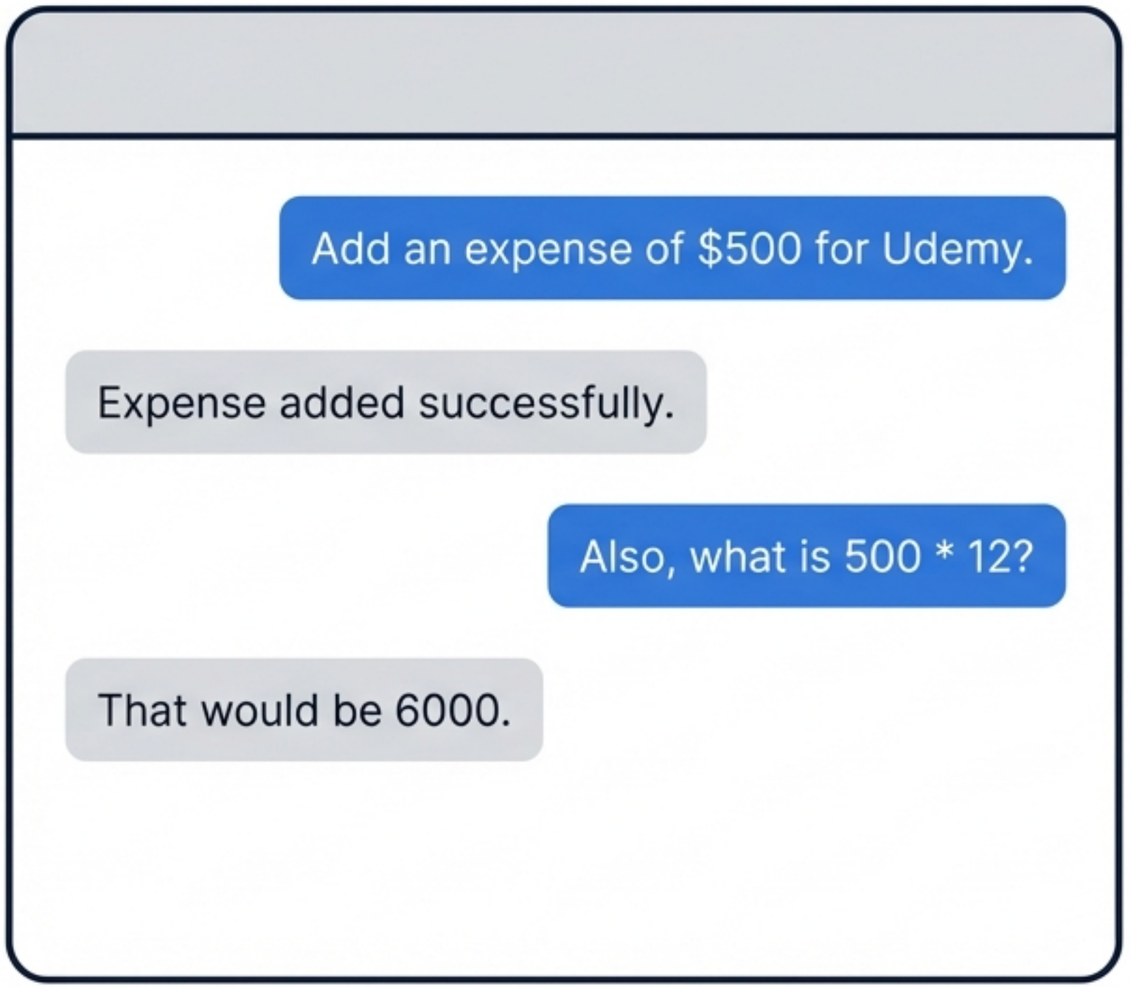


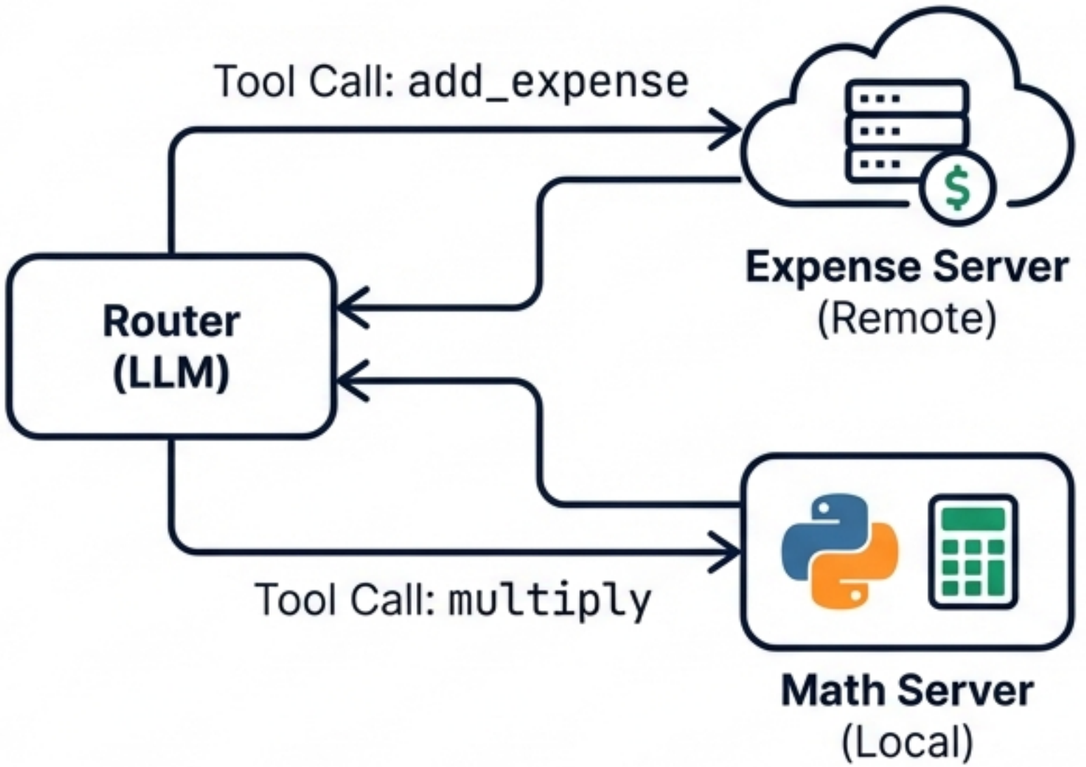
Dynamic Discovery: ‘What Can You Do?’



Key Insight: Zero Manual Definition: The bot adapts instantly. If the Server adds “Square Root” tomorrow, the Client supports it automatically.

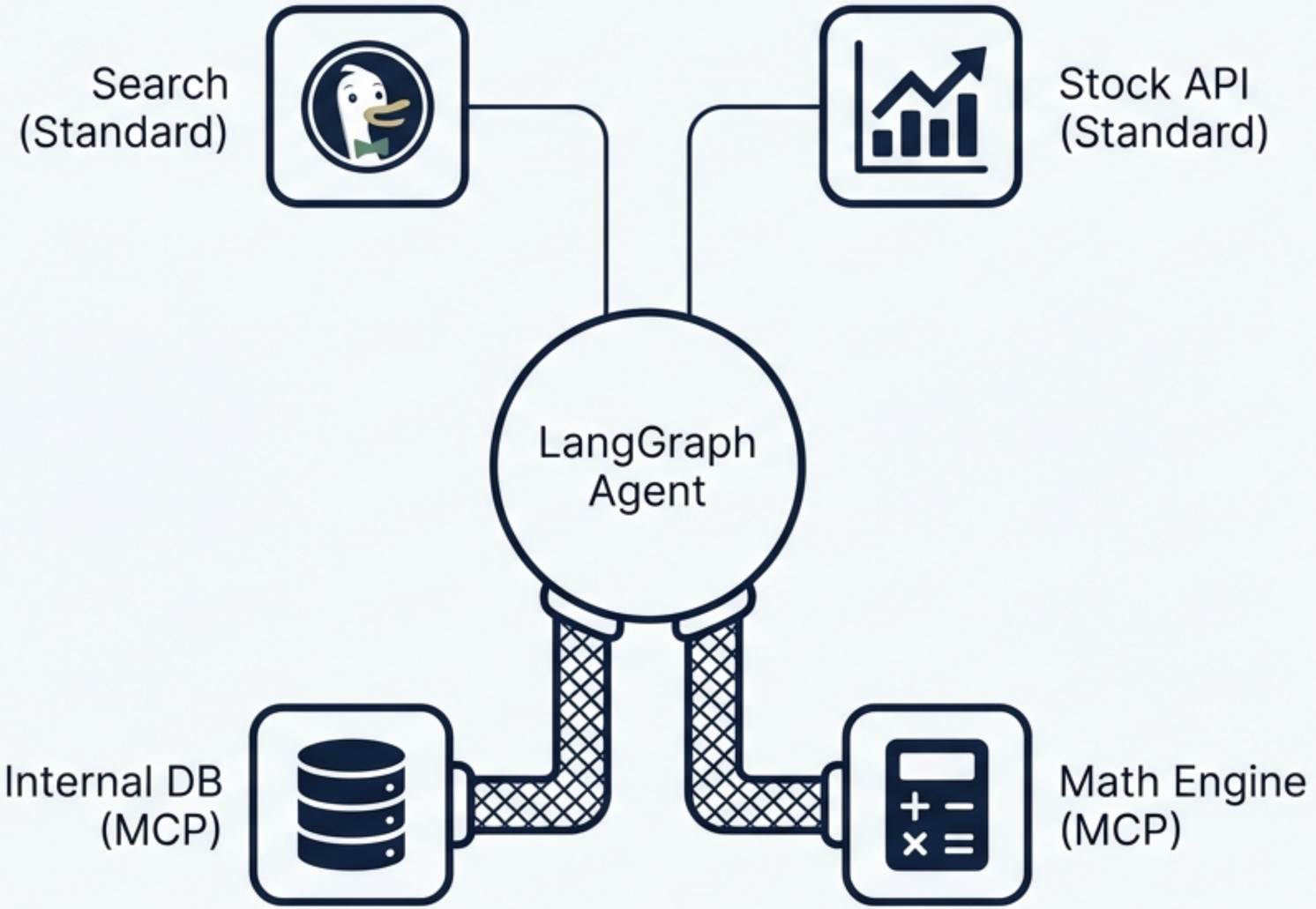
Multi-Server Architecture in Action





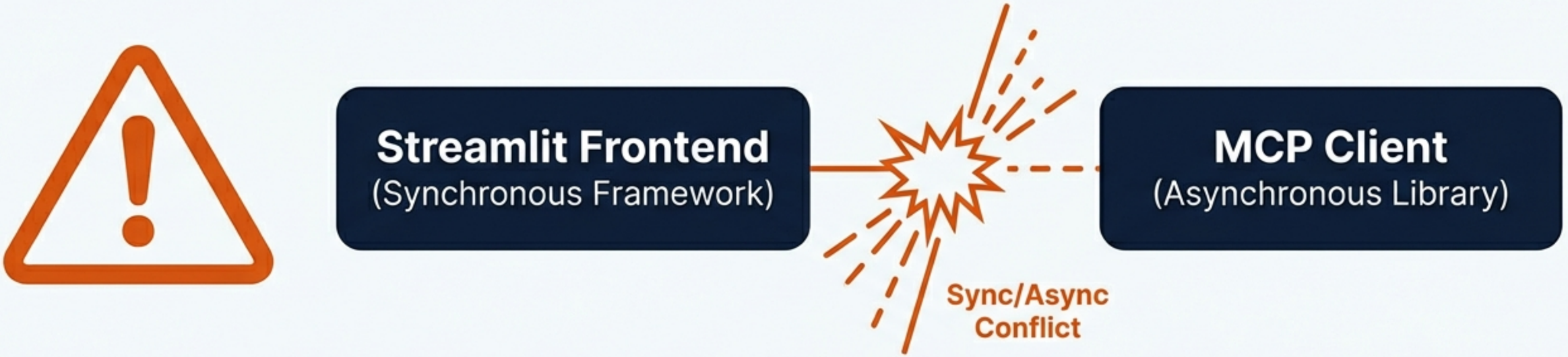
Unified Interface. The LLM intelligently routes queries to the correct MCP server based on the discovered schemas.

Hybrid Architecture: Best of Both Worlds



Flexibility: Keep stable, simple tools as standard integrations. Move complex, evolving, or proprietary integrations to MCP servers.

Real-World Friction: The Sync/Async Conflict

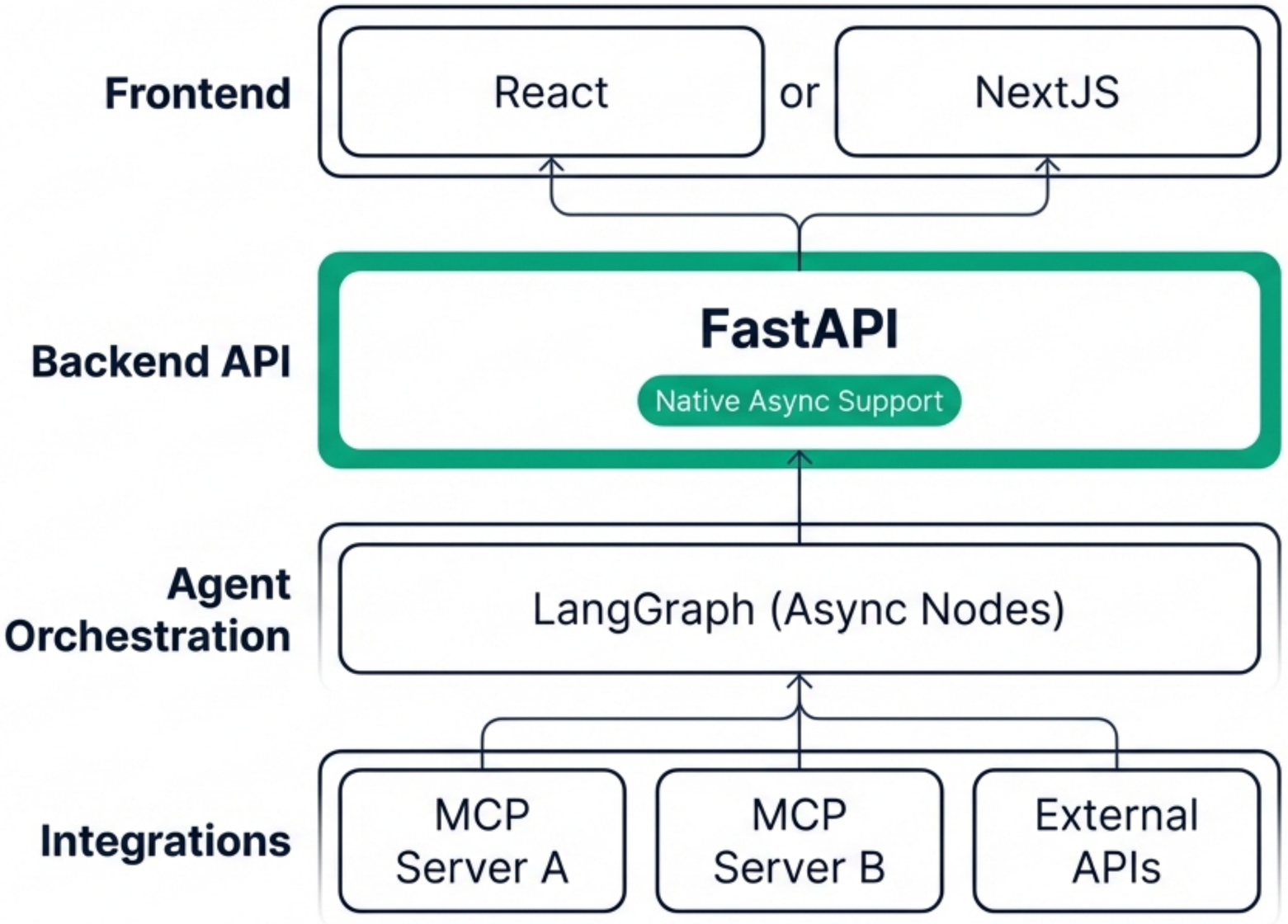


Workaround

```
asyncio.run(main())
AIOSQLite (Async Database)
```

The **Hack:** Bridging the gap requires forcing an async event loop within Streamlit and migrating standard SQLite to AIOSQLite. Functional, but not optimal.

The Production Stack Recommendation



Why FastAPI?

It handles asynchronous MCP connections natively, removing the friction found in Streamlit.

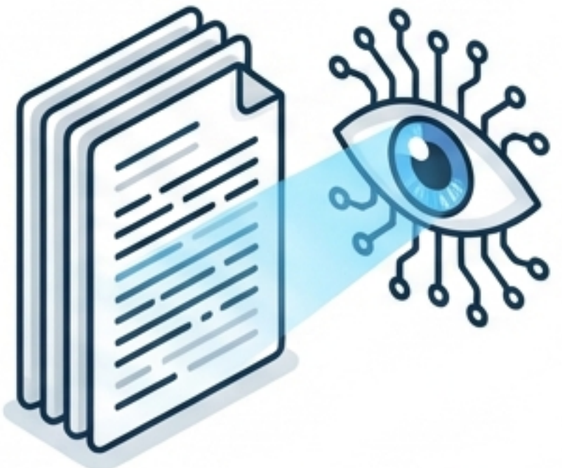
Summary & What's Next

[✅] Basic Chat & Memory

[✅] Web Search & Tools

[✅] Standardized Integrations (MCP)

[▢] Internal Knowledge (RAG)



Retrieval Augmented Generation

Achievement Unlocked: We have decoupled our agent from its tools, solving the maintenance headache. Next, we teach the agent to KNOW things using RAG.