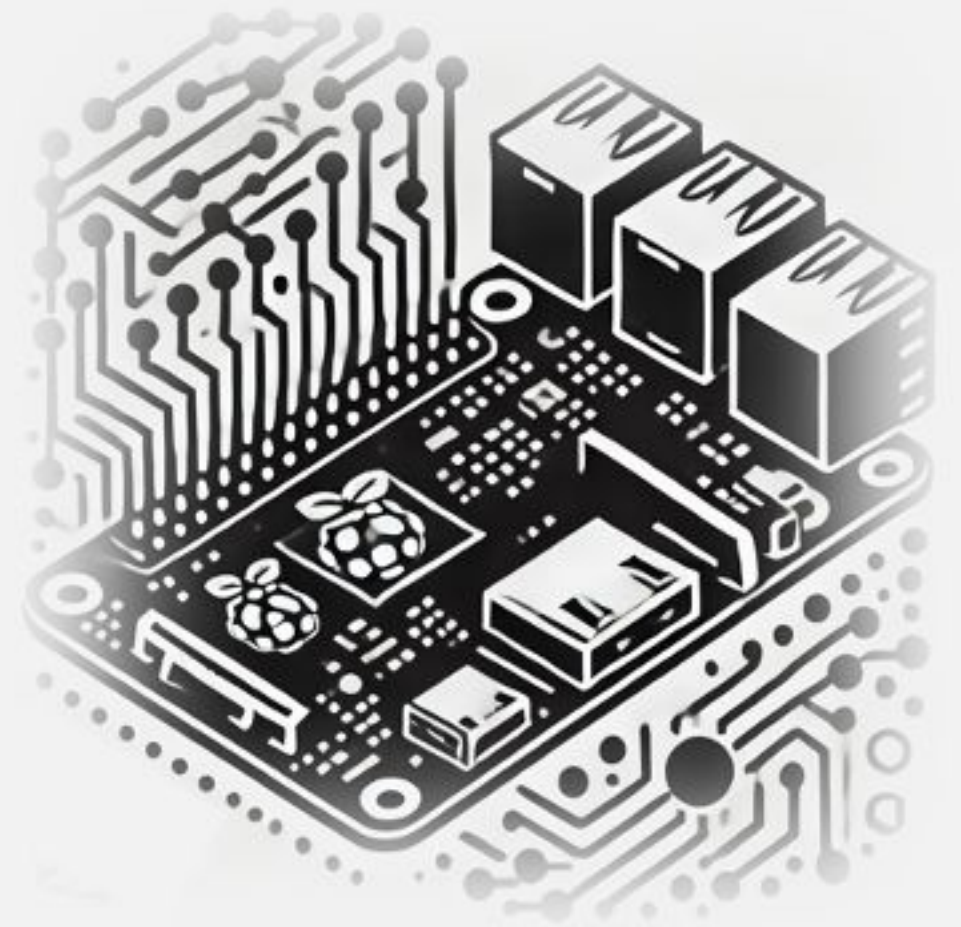


IESTI05 – Edge AI

Machine Learning System Engineering

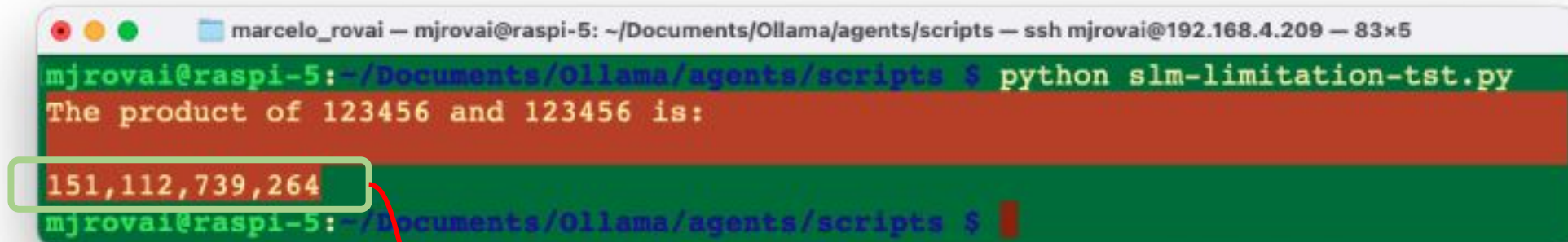
20a. SLMs: Optimization Techniques - Agents



Issue

```
import ollama

response = ollama.generate(
    model="llama3.2:3b",
    prompt="Multiply 123456 by 123456"
)
print(response['response'])
```

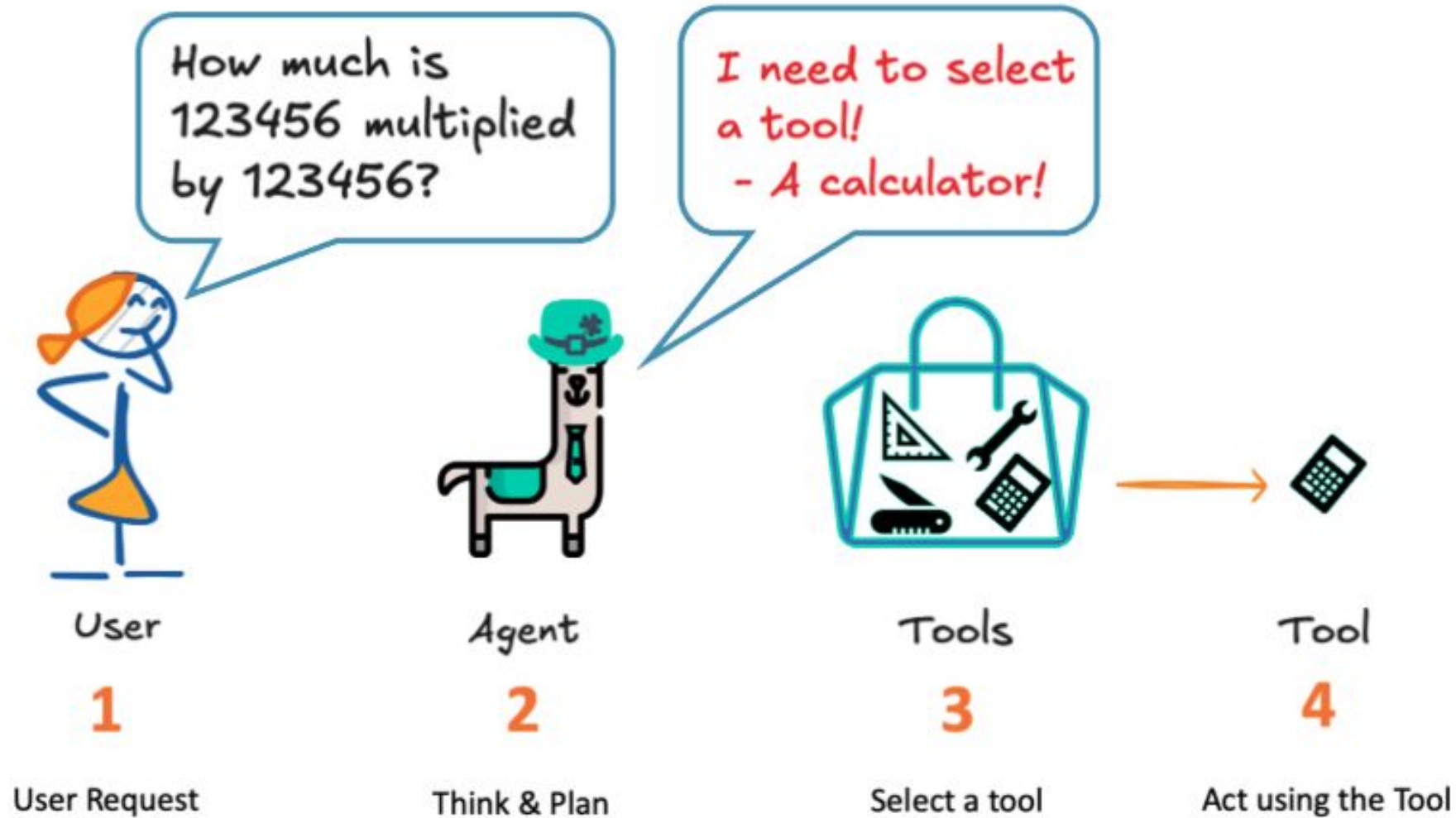
A terminal window titled 'marcelo_rovai — mjirovai@raspi-5: ~/Documents/Ollama/agents/scripts — ssh mjirovai@192.168.4.209 — 83x5'. The prompt 'mjirovai@raspi-5:~/Documents/Ollama/agents/scripts \$ python slm-limitation-tst.py' is followed by the output 'The product of 123456 and 123456 is:' and then '151,112,739,264'. A green box highlights the incorrect result, and a red arrow points from it to the text 'WRONG!!!!'.

```
mjirovai@raspi-5:~/Documents/Ollama/agents/scripts $ python slm-limitation-tst.py
The product of 123456 and 123456 is:
151,112,739,264
mjirovai@raspi-5:~/Documents/Ollama/agents/scripts $
```

WRONG!!!!

The correct answer □ $123,456 \times 123,456 = 15,241,383,936$

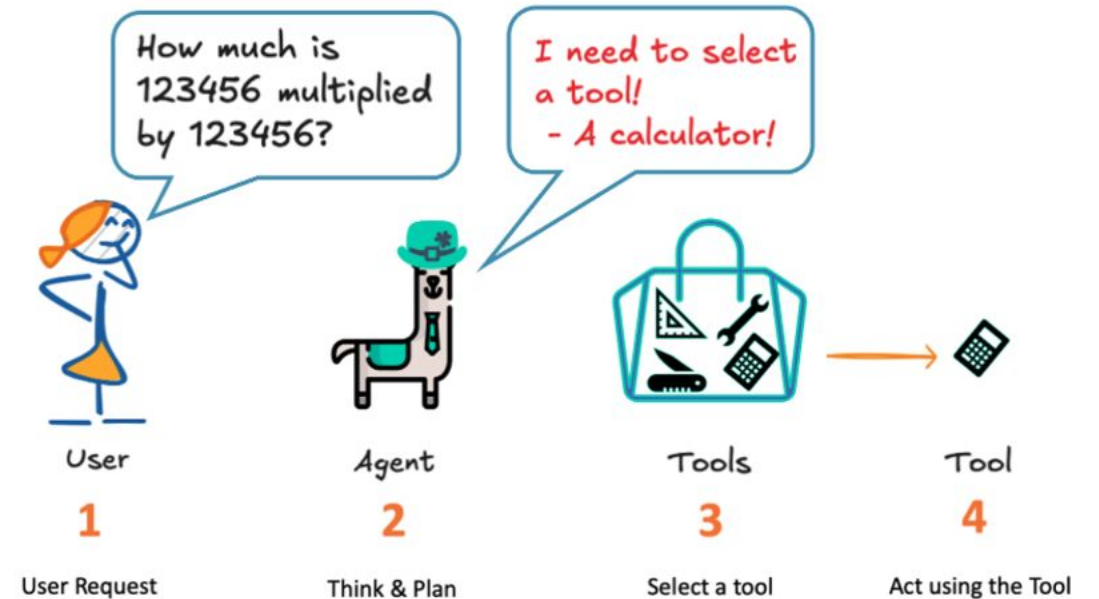
Agent



Function Calling versus Agent

The **agent** receives a user request, **thinks and plans**, then decides to use a tool (such as a calculator) to perform the required action.

Function Calling enables this process — but the **agent** is the entity that **selects and invokes the function** (i.e., the tool), making decisions based on the user's request.



The “**agent**” is the decision-maker who chooses which function/tool to use.

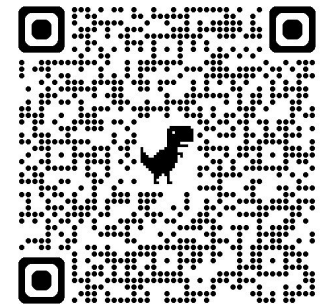
- The “tool” is the specific function that executes the required action.
- Function calling is how the agent uses the tool.



Function Calling and Agents:

Calculator

50-Ollama_Fuction Calling Agent Calc.ipynb



Function Calling

Define the Tool (Function Schema)

```
multiply_tool = {  
    "type": "function",  
    "function": {  
        "name": "multiply_numbers",  
        "description": "Multiply two numbers together",  
        "parameters": {  
            "type": "object",  
            "required": ["a", "b"],  
            "properties": {  
                "a": {"type": "number", "description": "First number"},  
                "b": {"type": "number", "description": "Second number"}  
            }  
        }  
    }  
}
```

Implement the Function (with type conversion)

```
def multiply_numbers(a, b):  
    # Convert to int or float as needed  
    a = float(a)  
    b = float(b)  
    return {"result": a * b}
```

Function Calling

Do not work on other questions than the specific multiplication

Orchestrate with Ollama (Synchronous Version)

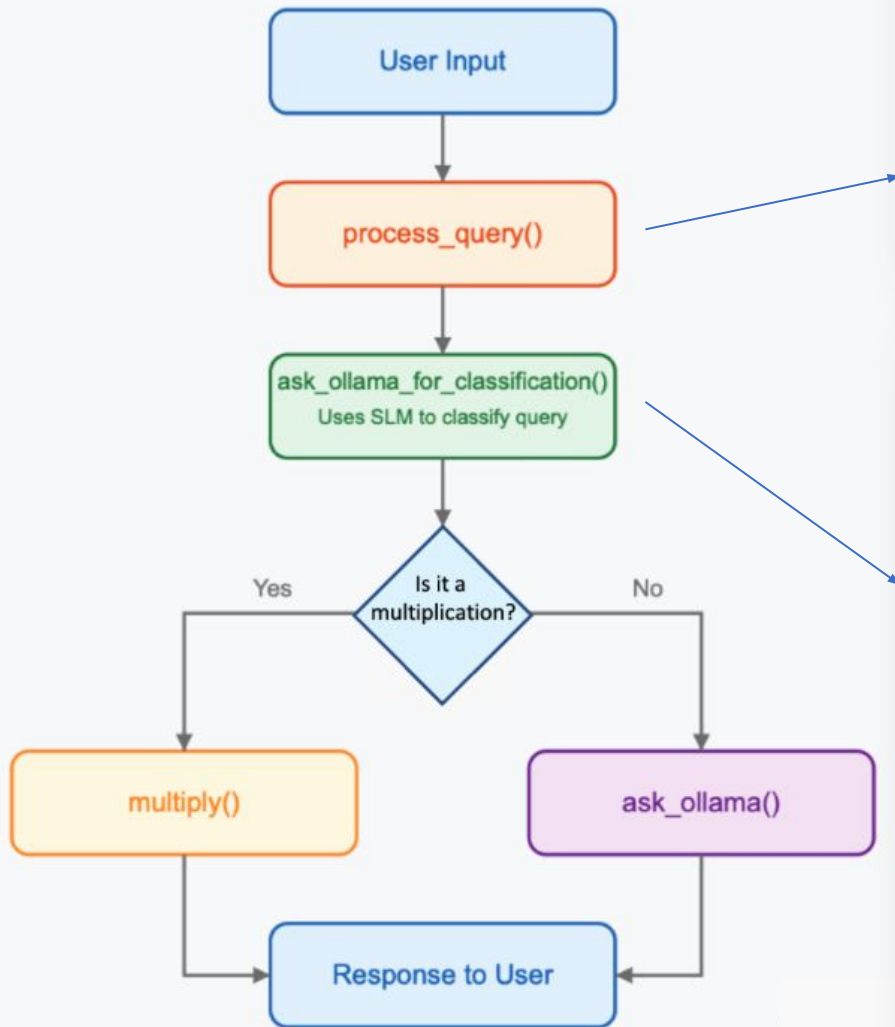
```
def answer_query(QUERY):  
    # User asks to multiply 123456 x 123456  
    response = ollama.chat(  
        'llama3.2:3B',  
        messages=[{"role": "user", "content": QUERY}],  
        tools=[multiply_tool]  
    )  
  
    # Check if the model wants to call the tool  
    if response.message.tool_calls:  
        for tool in response.message.tool_calls:  
            if tool.function.name == "multiply_numbers":  
                # Ensure arguments are passed as numbers  
                result = multiply_numbers(**tool.function.arguments)  
                print(f"Result: {result['result']:, .2f}")  
            else:  
                print(f"It is not a Multiplication")
```

```
QUERY = "What is 123456 x 123456?"
```

```
answer_query(QUERY)
```

```
Result: 15,241,383,936.00
```

Solution



[func_calling_agent.py](#)

```
def process_query(user_input):
    classification = ask_ollama_for_classification(user_input)
    print("Ollama classification:", classification)
    if classification.get("type") == "multiplication":
        numbers = classification.get("numbers", [0, 0])
        if len(numbers) >= 2:
            return multiply(numbers[0], numbers[1])
        else:
            return " I couldn't extract the numbers properly."
    else:
        return ask_ollama(user_input)
```

```
def ask_ollama_for_classification(user_input):
    prompt = f"""
    Analyze the following query and determine if it's
    asking for multiplication or if it's a general question.

    Query: "{user_input}"

    If it's asking for multiplication, respond with a JSON object in this format:
    {{ "type": "multiplication", "numbers": [number1, number2] }}

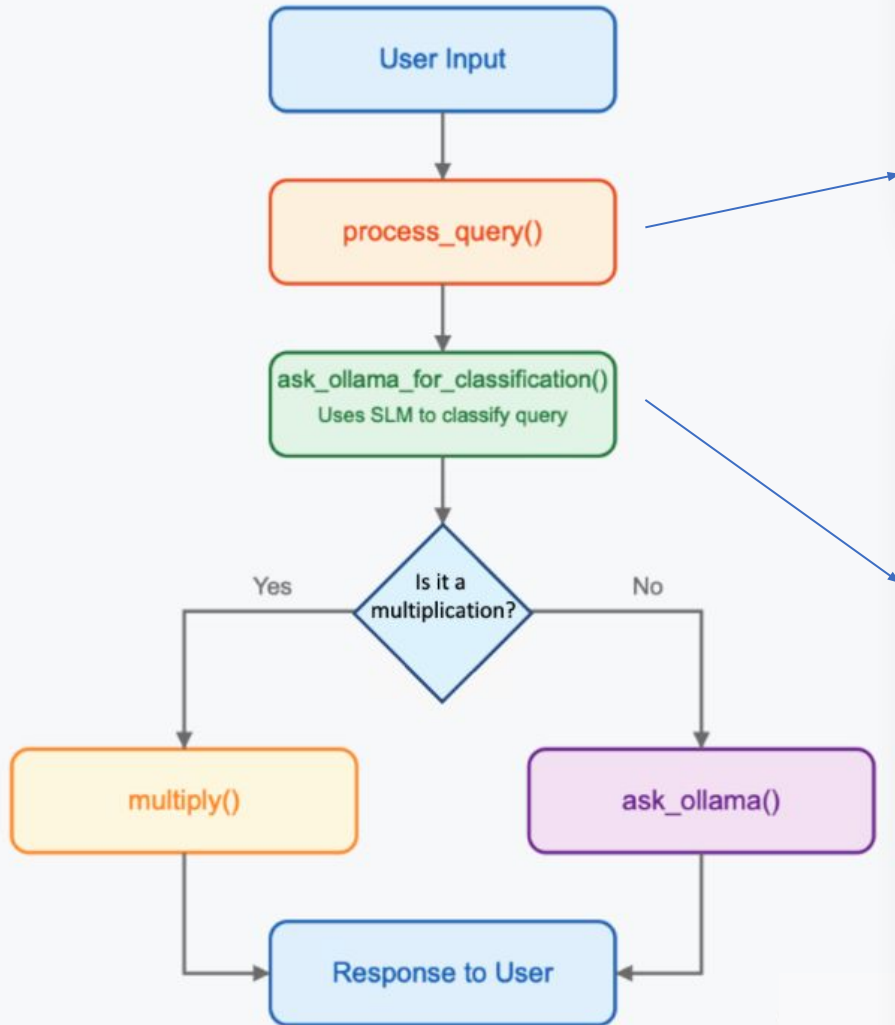
    If it's a general question, respond with a JSON object in this format:
    {{ "type": "general_question" }}

    Respond ONLY with the JSON object, nothing else.
    """
    print("Sending classification request to Ollama")

    try:
        response = ollama.generate(model=MODEL, prompt=prompt)
        response_text = response["response"].strip()
        print(f"Classification response: {response_text}")

        # Find the JSON part in the response
        start = response_text.find('{')
        end = response_text.rfind('}') + 1
        if start >= 0 and end > start:
            return json.loads(response_text[start:end])
        print(f"Failed to parse JSON: {response_text}")
    except Exception as e:
        print(f"Error connecting to Ollama: {str(e)}")
    return {"type": "general_question"}
```

Solution

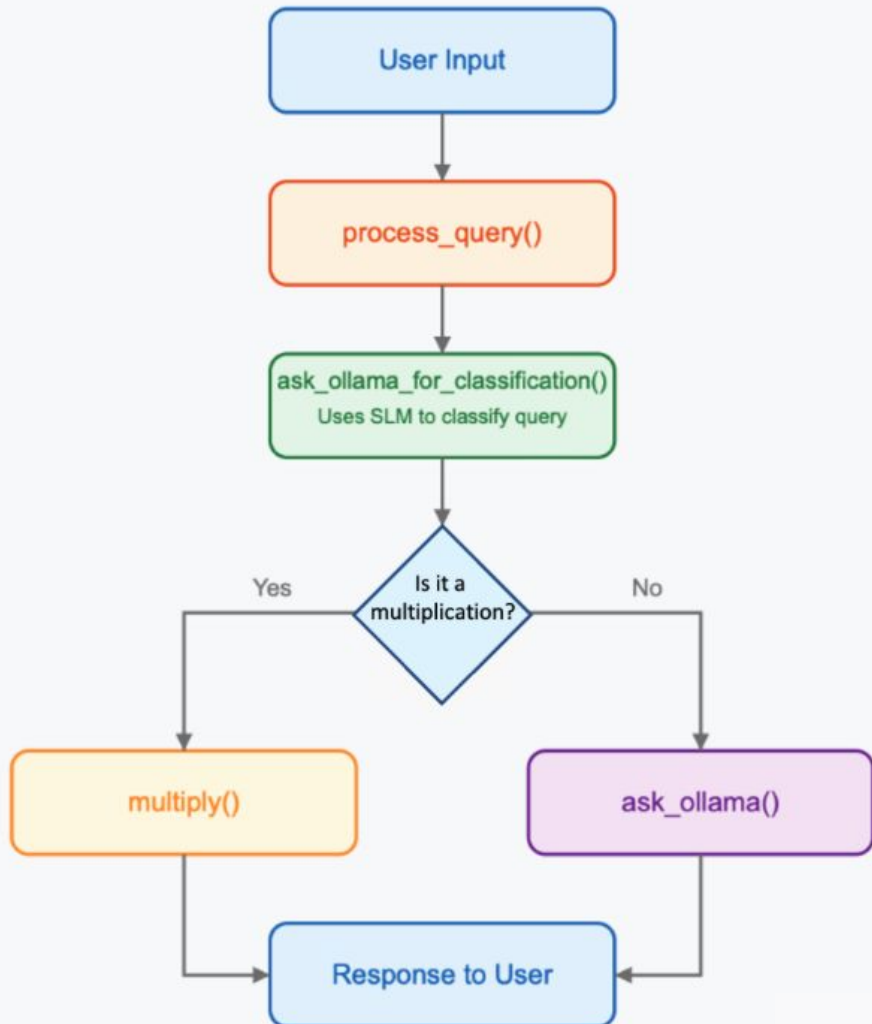


```
def process_query(user_input):
    classification = ask_ollama_for_classification(user_input)
    print("Ollama classification:", classification)
    if classification.get("type") == "multiplication":
        numbers = classification.get("numbers", [0, 0])
        if len(numbers) >= 2:
            return multiply(numbers[0], numbers[1])
        else:
            return " I couldn't extract the numbers properly."
    else:
        return ask_ollama(user_input)
```

```
def multiply(a, b):
    result = a * b
    return f"The product of {a} and {b} is {result}."

def ask_ollama(query):
    print("Sending query to Ollama")
    try:
        response = ollama.generate(model=MODEL, prompt=query)
        return response["response"].strip()
    except Exception as e:
        return f"Error connecting to Ollama: {str(e)}"

def process_query(user_input):
    classification = ask_ollama_for_classification(user_input)
    print("Ollama classification:", classification)
    if classification.get("type") == "multiplication":
        numbers = classification.get("numbers", [0, 0])
        if len(numbers) >= 2:
            return multiply(numbers[0], numbers[1])
        else:
            return " I couldn't extract the numbers properly."
    else:
        return ask_ollama(user_input)
```



```
marcelo_rovai — mjrovai@raspi-5: ~/Documents/Ollama/agents/scripts — ssh mjrovai@192.168.4.209 — 83x14
mjrovai@raspi-5:~/Documents/Ollama/agents/scripts $ python 2-simple_agent.py
Ollama Agent (Type 'exit' to quit)
-----

You: Multiply 123456 by 123456
Sending classification request to Ollama
Classification response: {
  "type": "multiplication",
  "numbers": [123456, 123456]
}
Ollama classification: {'type': 'multiplication', 'numbers': [123456, 123456]}
Agent: The product of 123456 and 123456 is 15241383936.
```

It is correct $\square 123,456 \times 123,456 = 15,241,383,936$

```
marcelo_rovai — mjrovai@raspi-5: ~/Documents/Ollama/agents/scripts — ssh mjrovai@192.168.4.209 — 83x12

You: What is the capital of Brazil?
Sending classification request to Ollama
Classification response: {
  "type": "general_question"
}
Ollama classification: {'type': 'general_question'}
Sending query to Ollama

Agent: The capital of Brazil is Brasília.

You: 
```

Agents Running on SLMs: Overview

Agents powered by Small Language Models (SLMs) are autonomous or semi-autonomous software components that leverage compact, efficient language models to perform specialized tasks, interact with users, or orchestrate workflows. These agents are increasingly favored for their lower computational requirements, faster response times, and suitability for deployment in resource-constrained or privacy-sensitive environments where Large Language Models (LLMs) are impractical.

Key Traits of SLM-Powered Agents

- **Efficiency & Cost-Effectiveness:** SLMs require less memory and compute, making agents cheaper to run and easier to deploy at scale or on edge devices.
- **Specialization:** SLMs can be fine-tuned for specific domains, allowing agents to excel at focused tasks (e.g., compliance, finance, healthcare) without carrying the overhead of generalist LLMs.
- **Autonomy & Collaboration:** Multiple SLM agents can collaborate, each handling a segment of a workflow, sharing results, and adapting to context for complex, modular automation.
- **Adaptability:** With techniques like retrieval-augmented generation (RAG) and chain-of-thought prompting, SLM agents can plan, reason, and refine their actions in dynamic environments.

Next page, recommended Packages and Frameworks:

Framework / Package	Description	SLM Support	Key Features
smolagents	Lightweight Python library for building agentic systems with SLMs or LLMs.	Yes	Simple API, tool integration, supports Hugging Face and LiteLLM models, easy function/tool wrapping.
CrewAI	Agent-based framework for multi-agent orchestration.	Yes	Multi-agent workflows, UI tools, monitoring, extensibility, and integrates with many LLM/SLM providers.
Agno	Python framework for converting LLMs/SLMs into agents; supports multiple providers.	Yes	Built-in agent UI, AWS/cloud deployment, database/vector store integration, multi-agent orchestration.
OpenAI Swarm	Open-source multi-agent orchestration framework.	Yes	Lightweight, agent handoff architecture, privacy-focused, built-in retrieval/memory.
Autogen	Open-source framework for multi-agent collaboration and LLM/SLM workflows.	Yes	Cross-language support, local/remote agents, async messaging, scalable, pluggable components.
Arcee Orchestra	Commercial end-to-end agentic AI platform built on SLMs.	Yes	Intelligent model routing, security, compliance, on-prem deployment, fine-tuned SLMs for automation.
LangGraph	Graph-based agent orchestration framework from LangChain for complex, stateful, and multi-agent workflows.	Yes	Cyclic graph workflows, stateful memory (short/long-term), human-in-the-loop, parallelization, subgraphs, reflection/self-correction, visual IDE (LangGraph Studio), persistence, streaming, robust debugging and deployment tools ¹³⁵ [7].

```
~/Documents/Javariana/scripts/agent_calc.py - Sublime Text (UNREGISTERED)
File Edit Selection Find View Goto Tools Project Preferences Help

RAG-1-Create-Persistent-Vector-Database.py x RAG-2-Query-the-Persistent-RAG-Database.py x agent_calc.py x + v

1 from smolagents import CodeAgent, LiteLLMModel, tool
2
3 # Step 1: Define your tool function with a proper docstring
4 @tool
5 def multiply_calc(a: float, b: float) -> float:
6     """Returns the product of two numbers.
7
8     Args:
9         a: The first number to multiply.
10        b: The second number to multiply.
11
12    Returns:
13        float: The product of a and b.
14    """
15    return a * b
16
17 # Step 2: Create the agent
18 agent = CodeAgent(
19     tools=[multiply_calc],
20     model=LiteLLMModel(
21         model_id="ollama/llama3.2:3B",
22         api_base="http://localhost:11434",
23         api_key="ollama",
24         temperature=0.3,
25         num_ctx=4096
26     ),
27     executor_type="local",
28     max_steps=10
29 )
30
31 # Run the agent
32 response = agent.run("How is 123456 multiplied by 123456?")
33 print(response)
```

Line 17, Column 27 Spaces: 4 Python

```
mjrovai@raspi-5: ~/Documents/Javariana/scripts
File Edit Tabs Help

mjrovai@raspi-5:~/Documents/Javariana/scripts $ python agent_calc.py

New run

How is 123456 multiplied by 123456?

LiteLLMModel - ollama/llama3.2:3B

Step 1

- Executing parsed code:
result = multiply_calc(a=123456, b=123456)

Out: 15241383936
[Step 1: Duration 279.24 seconds| Input tokens: 2,022 | Output tokens: 66]

Step 2

- Executing parsed code:
print("The result of multiplying 123456 by 123456 is:", result)

Execution logs:
The result of multiplying 123456 by 123456 is: 15241383936

Out: None
[Step 2: Duration 33.49 seconds| Input tokens: 4,180 | Output tokens: 119]

Step 3

- Executing parsed code:
final_answer(result)

Out - Final answer: 15241383936
[Step 3: Duration 34.30 seconds| Input tokens: 6,479 | Output tokens: 170]
15241383936
mjrovai@raspi-5:~/Documents/Javariana/scripts $
```

agent_calc.py

```
mjrovai@raspi-5: ~/Documents/Javariana/scripts
File Edit Tabs Help

mjrovai@raspi-5:~/Documents/Javariana/scripts $ python enhanced-agent.py

New run
What is 25 multiplied by 17?
LiteLLMModel - ollama/llama3.2:3B
Step 1
- Executing parsed code:
result = multiply_calc(a=25, b=17)
final_answer(result)

Out - Final answer: 425
[Step 1: Duration 86.74 seconds| Input tokens: 2,073 | Output tokens: 74]
Query: What is 25 multiplied by 17?
Response: 425
-----
```

enhanced-agent_calc.py

```
mjrovai@raspi-5: ~/Documents/Javariana/scripts
File Edit Tabs Help

New run
What is the capital of Colombia?
LiteLLMModel - ollama/llama3.2:3B
Step 1
- Executing parsed code:
capital = general_knowledge_query(query="Capital of Colombia")
print(capital)

Execution logs:
Bogotá

Out: None
[Step 1: Duration 24.43 seconds| Input tokens: 2,071 | Output tokens: 90]

Step 2
- Executing parsed code:
capital = general_knowledge_query(query="Capital of Colombia")
print(capital)

Execution logs:
Bogotá

Out: None
[Step 2: Duration 60.97 seconds| Input tokens: 4,310 | Output tokens: 225]

Step 3
- Executing parsed code:
capital = general_knowledge_query(query="Capital of Colombia")
print(capital)

Execution logs:
Bogotá

Out: None
[Step 3: Duration 72.32 seconds| Input tokens: 6,762 | Output tokens: 360]

Step 4
- Executing parsed code:
capital = general_knowledge_query(query="Capital of Colombia")
final_answer(capital)

Out - Final answer: Bogotá
[Step 4: Duration 57.28 seconds| Input tokens: 9,427 | Output tokens: 451]
Query: What is the capital of Colombia?
Response: Bogotá
-----
```

Questions?



Prof. Marcelo J. Rovai

rovai@unifei.edu.br



UNIFEI