MGS636 Lab 5 Report

1. Summary of Learnings

This lab helped me to learn more about AI systems that go beyond simple prompting

and Retrieval-Augmented Generation, such as agentic AI and web application

development. I developed a ReAct agent with LlamaIndex and Ollama and learnt

how to provide model access to tools such as arithmetic functions and

DuckDuckGo search. We demonstrated that this strategy could help make LLMs

more helpful, by enabling them to reason step by step and choose whether to call

other tools, instead than simply using generated text.

The biggest lesson from this homework was the significance of FunctionTool . I

converted normal Python functions into tools, thus linking classic programming

logic with the agent's thinking process. The agent could perform multi-step tasks

with add, subtract, multiply, divide and search functions. The verbose output helped

me comprehend how the model went through ideas, actions, observations, and

ultimate replies.

I found that model selection and available system memory have direct impact on

performance too. The bigger model did better reasoning, but I had to do additional

troubleshooting as it took more RAM. Clear cues were important, too; the clearer

the prompt concerning tool sequence, the more trustworthy the result.

Another big part of the lab was to turn the agent into a Streamlit web app. This made

the system much more user pleasant, as the user could engage through a browser,

instead of the terminal. I added an extra function, divide, to increase usability, and

added a sidebar with sample prompts. In general, this lab demonstrated the

integration of AI agents, local models, tools and front-end interfaces in a practical AI

application.

2. Agent Design and Tool Setup

The agent was implemented in Python using LlamaIndex, Ollama and a collection of

tools wrapped with FunctionTool. The code structure is shown below in the screenshots,

including imports, math functions, the web search function, tool declarations, and the list

of tools supplied into the ReAct agent.

```
from llama_index.core import Settings
from llama_index.llms.ollama import Ollama
from llama_index.core.agent.workflow import ReActAgent
from llama_index.core.tools import FunctionTool
from duckduckgo_search import DDGS
import asyncio
import streamlit as st

Settings.llm = Ollama(model="llama3.2:latest", request_timeout=360.0)

def multiply(a: int, b: int) -> int:
    """Multiply two integers and return the result integer."""
    return a * b

def add(a: int, b: int) -> int:
    """Add two integers and return the result integer."""
    return a + b

def subtract(a: int, b: int) -> int:
    """Subtract the second integer from the first integer."""
    return a - b

def divide(a: int, b: int) -> float:
    """Divide the first number by the second number and return the result."""
    if b == 0:
        return 0.0
    return a / b

def search(query: str) -> str:
    """
    Args:
        query: user prompt
    Returns:
        context (str): search results to the user query
    """
    req = DDGS()
    response = req.text(query, max_results=3)
    context = ""
    for result in response:
```

Figure 1. webapp.py code

```
def divide(a: int, b: int) -> float:
    """Divide the first number by the second number and return the result."""
    if b == 0:
        return 0.0
    return a / b

def search(query: str) -> str:
    """
    Args:
        query: user prompt
    Returns:
        context (str): search results to the user query
    """
    req = DDGS()
    response = req.text(query, max_results=3)
    context = ""
    for result in response:
        body = result.get("body", "")
        if body:
            context += body + "\n"
    return context

multiply_tool = FunctionTool.from_defaults(fn=multiply)
add_tool = FunctionTool.from_defaults(fn=add)
subtract_tool = FunctionTool.from_defaults(fn=subtract)
divide_tool = FunctionTool.from_defaults(fn=divide)
search_tool = FunctionTool.from_defaults(fn=search)

fntools = [multiply_tool, add_tool, subtract_tool, divide_tool, search_tool]

agent = ReActAgent(tools=fntools, llm=Settings.llm, max_iterations=15, verbose=True)

async def main():
    query =  "First use the subtract tool to subtract 20 from 100. Then use the multiply
tool to multiply 80 by 2. You must use both tools before giving the final answer."
    response = await agent.run(query)
    print(response)

if __name__ == "__main__":
    asyncio.run(main())
```

Figure 2. agent.py code

3. Agent Testing Results

In the first test the agent had to use addition and division. After further refining the query and tweaking the model, the agent was able to finish both phases and produced a final answer of 35.0.

Figure 3. Terminal output for the first agent test, with the correct answer of 35.0

The verbose output reveals that the agent was doing the necessary procedures in order instead of just guessing the answer. This enabled us to verify the workflow and demonstrated how the ReAct pattern can help reduce errors by integrating reasoning with

tool usage.

For the second query, I tried out the agent on another multi-step math assignment

including subtraction and multiplication. The agent properly produced 160 which showed

that it was able to follow a clear sequence of tool calls with better consistency.

Figure 4. Terminal output for the second agent test, showing the correct final answer of 160.

The second test overall was more stable because the prompt explicitly stated both the order of activities as well as the obligation to utilize both tools before answering.

4. Streamlit Web Application

After testing the agent in the terminal, we converted the same logic to a Streamlit web app. We got a sleeker UI with a title, input box, submit button, and sidebar sample prompts and a warning for empty input.

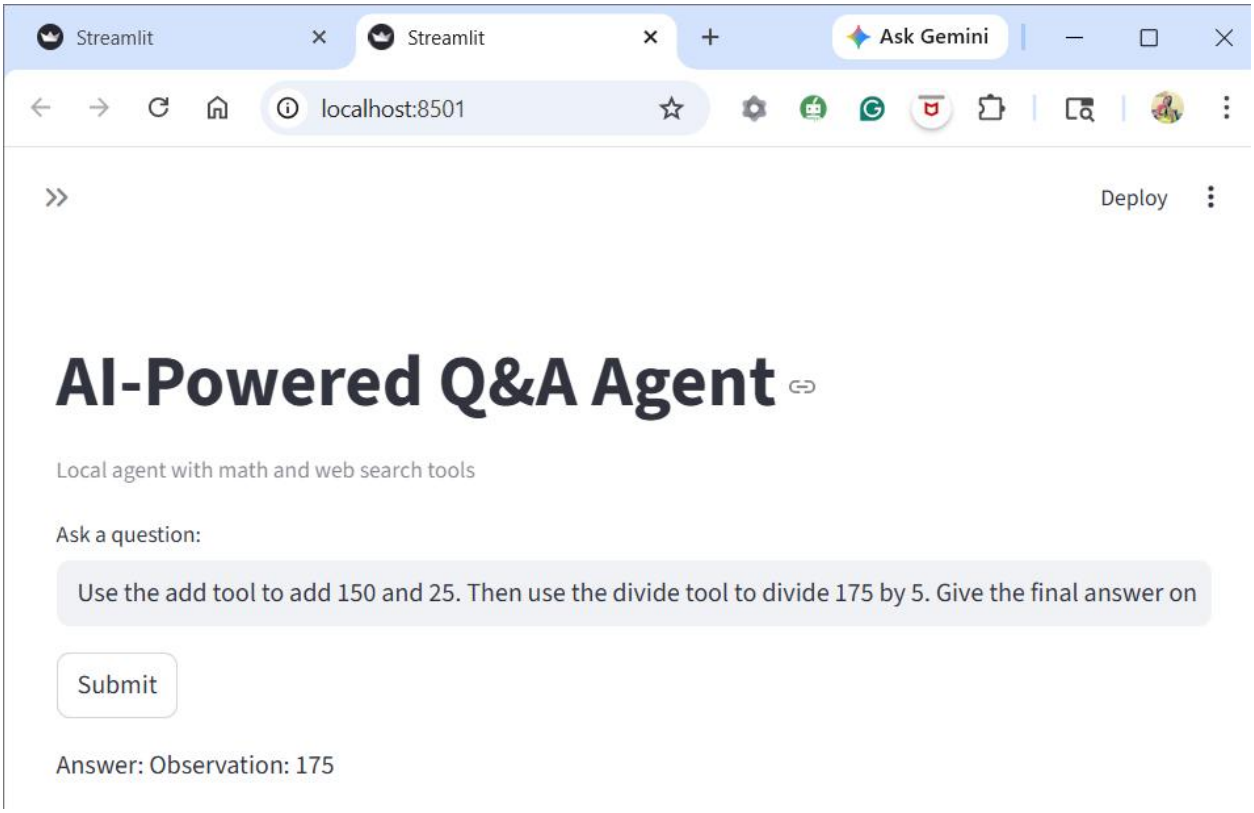


Figure 5. Streamlit web app

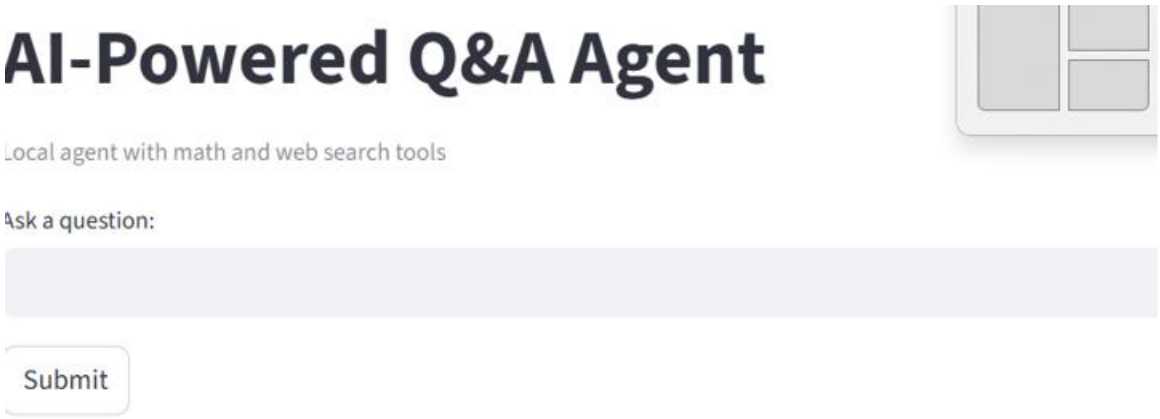
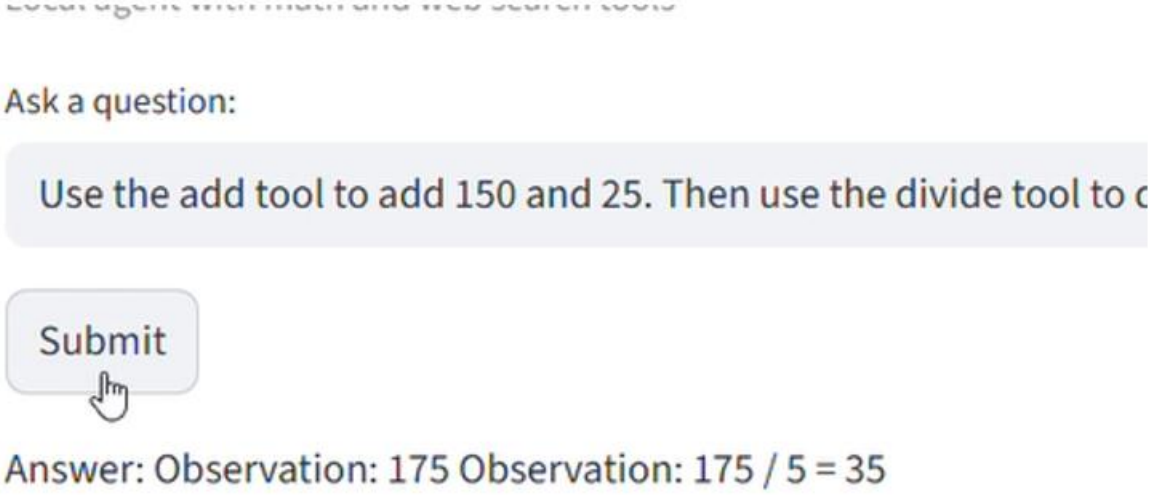


Figure 5. Streamlit landing page after the app loaded successfully.



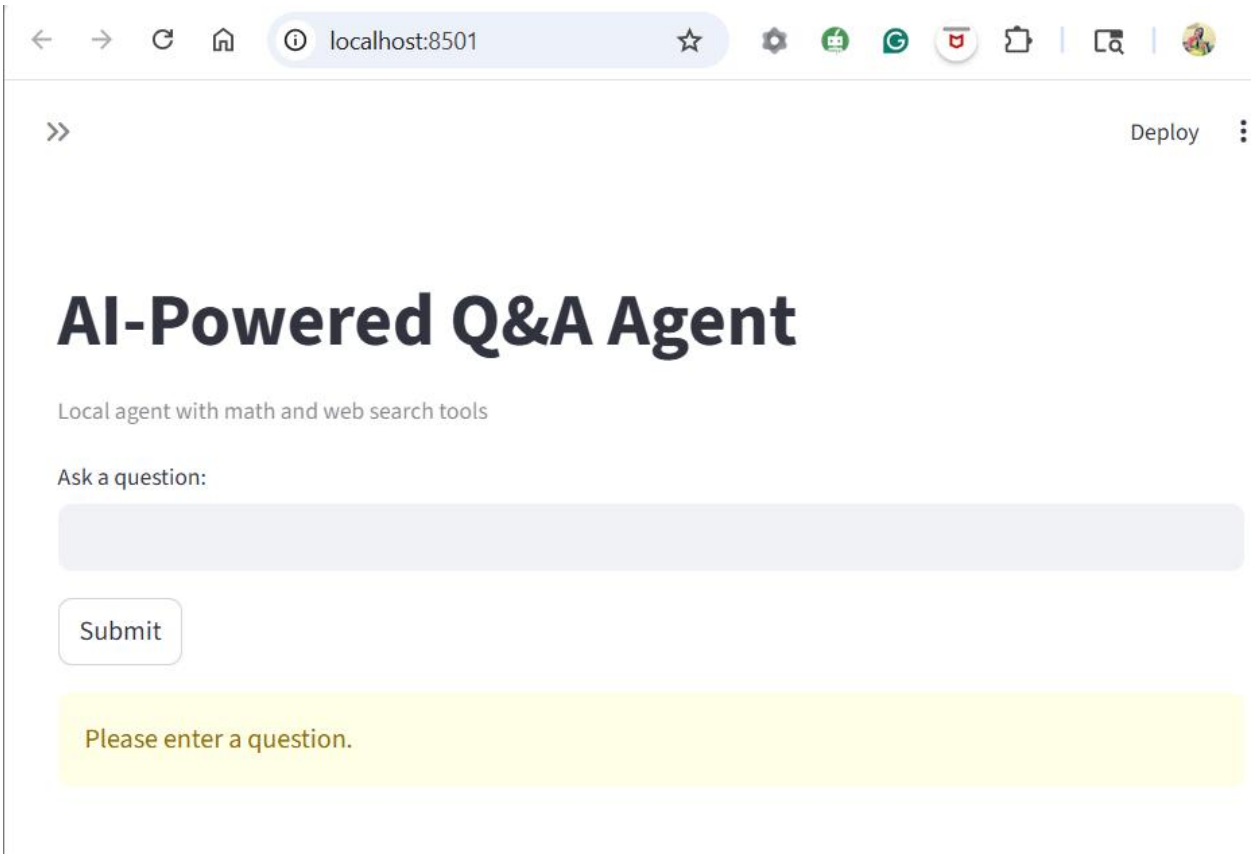


Figure 6. Streamlit web app warning message

The Streamlit web software would take a prompt and provide back a response, but the output was less polished than the terminal version and sometimes showed intermediate observations instead of the final answer completely formatted. The terminal method was better for debugging, because it made the reasoning process more transparent. The online

app did not provide the full reasoning trace and nicer response formatting.

Conclusion

In this assignment, I upgraded the application by adding a new division function and establishing a Streamlit sidebar with sample questions. These improvements make the

tool more interactive and user friendly. This lab enabled me to understand how AI tools may

be improved by adding more functionalities and better UI. The application, in general,

worked and was able to fulfill the lab criteria.