<u>Part A</u>: Theoretical Questions

Q.1, Space_Time_Core(STC):

(Ans)➣ Customer Support Agent

Space

Can access (three):

1. Knowledge base / FAQs
2. User's ticket history
3. Order or service status

Cannot access by default ():

1. User passwords
2. Internal confidential company strategies

Time

Stores:

- Ticket state and resolution history
- User preferences (language, channel)

Forgets:

- ☐ Old resolved tickets
- ☐ Temporary frustration signals

Retention rule:

- Tickets retained per compliance policy
- Emotional signals not stored long-term

Core (Trust Rules)

1. Explanation: Explains causes and steps clearly
2. Approval: Escalates to human when needed
3. Refusal: Refuses unsafe or policy-breaking requests
4. Uncertainty: Clearly states when it doesn't know and escalates

Q.2, Tool-use pattern reasoning:

➣ **(Ans)**

A. **- What is tool-use**
  Tool-use is the ability of an AI agent to invoke external systems (such as APIs, databases, search engines, calculators, or code execution environments) integrated with LLM and other AI model to obtain real-time information, perform precise computations, or take actions that cannot be done reliably through internal reasoning alone.

B. – 2 Cases where the agent must call tools :

1. Real-time or dynamic information is required – e.g., checking current flight prices, weather conditions, system status, or account balances. The agent's internal knowledge may be outdated, so tools are necessary for accuracy.

2. High-precision computation or execution is needed – e.g., calculating taxes, generating financial forecasts, running simulations, or validating large datasets, where correctness and determinism matter.

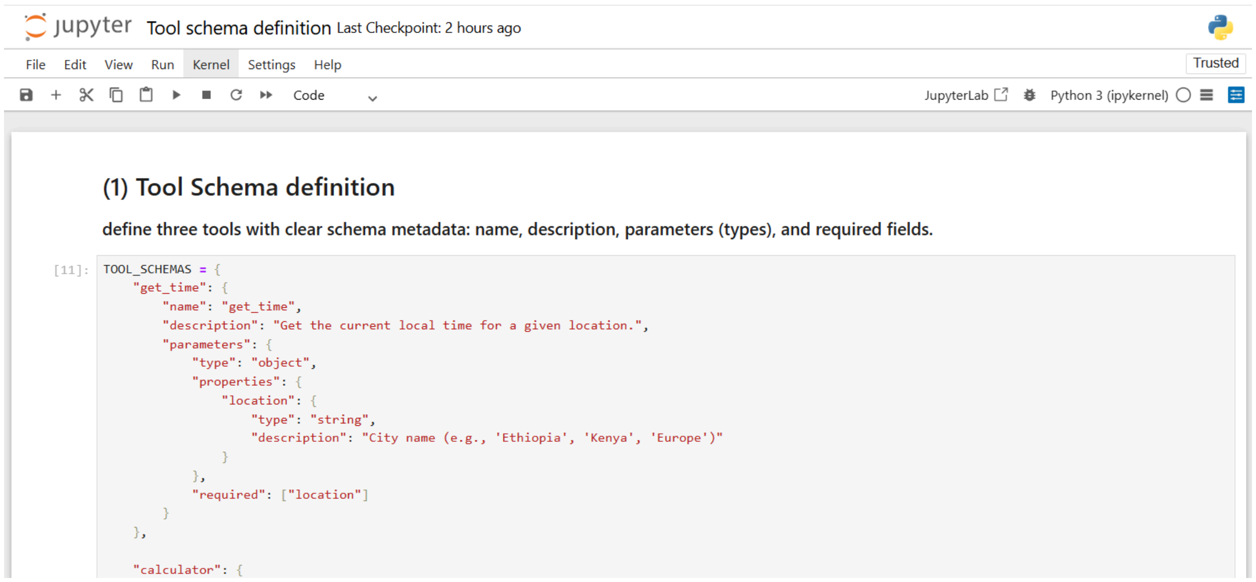
C. – 1 Case where the agent must not call tools :

- Pure conceptual reasoning or explanation – e.g., explaining a concept like "what is compound interest" or "how machine learning works."
  Why: No external data or execution is required; calling tools would add unnecessary latency, cost, and complexity without improving correctness.

PART B: Implementation and Configuration:

Q.1 Define tool schemas:

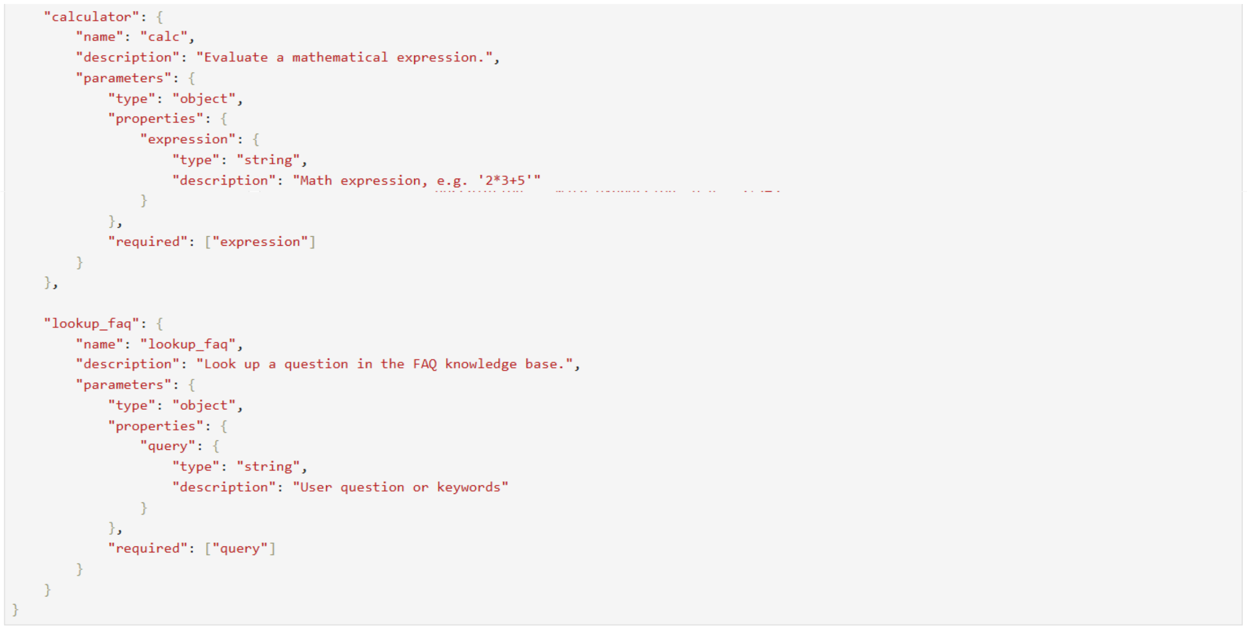
(Ans)➳


jupyter
Tool schema definition Last Checkpoint: 2 hours ago
File Edit View Run Kernel Settings Help
Trusted
Code
JupyterLab
Python 3 (ipykernel)

(1) Tool Schema definition

define three tools with clear schema metadata: name, description, parameters (types), and required fields.

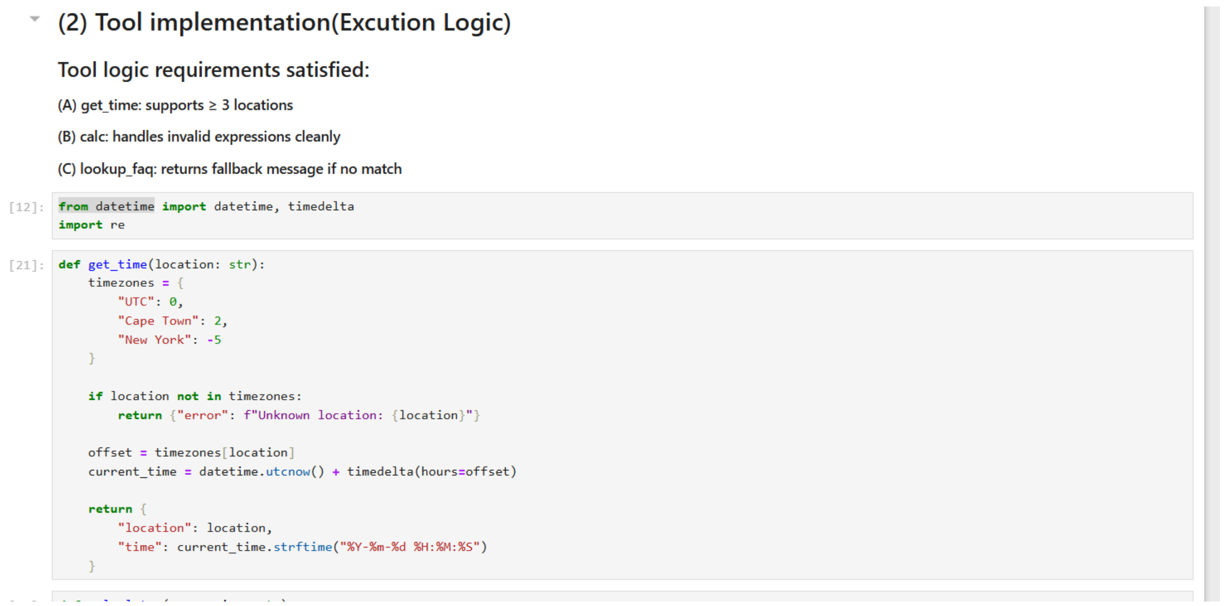
```
[11]: TOOL_SCHEMAS = {
    "get_time": {
        "name": "get_time",
        "description": "Get the current local time for a given location.",
        "parameters": {
            "type": "object",
            "properties": {
                "location": {
                    "type": "string",
                    "description": "City name (e.g., 'Ethiopia', 'Kenya', 'Europe')"
                }
            },
            "required": ["location"]
        }
    },

    "calculator": {
```



```
    "calculator": {
        "name": "calc",
        "description": "Evaluate a mathematical expression.",
        "parameters": {
            "type": "object",
            "properties": {
                "expression": {
                    "type": "string",
                    "description": "Math expression, e.g. '2*3+5'"
                }
            },
            "required": ["expression"]
        }
    },

    "lookup_faq": {
        "name": "lookup_faq",
        "description": "Look up a question in the FAQ knowledge base.",
        "parameters": {
            "type": "object",
            "properties": {
                "query": {
                    "type": "string",
                    "description": "User question or keywords"
                }
            },
            "required": ["query"]
        }
    }
}
```

➣ Q.2. Implement the tools (execution logic):

(Ans).



(2) Tool implementation(Excution Logic)

Tool logic requirements satisfied:

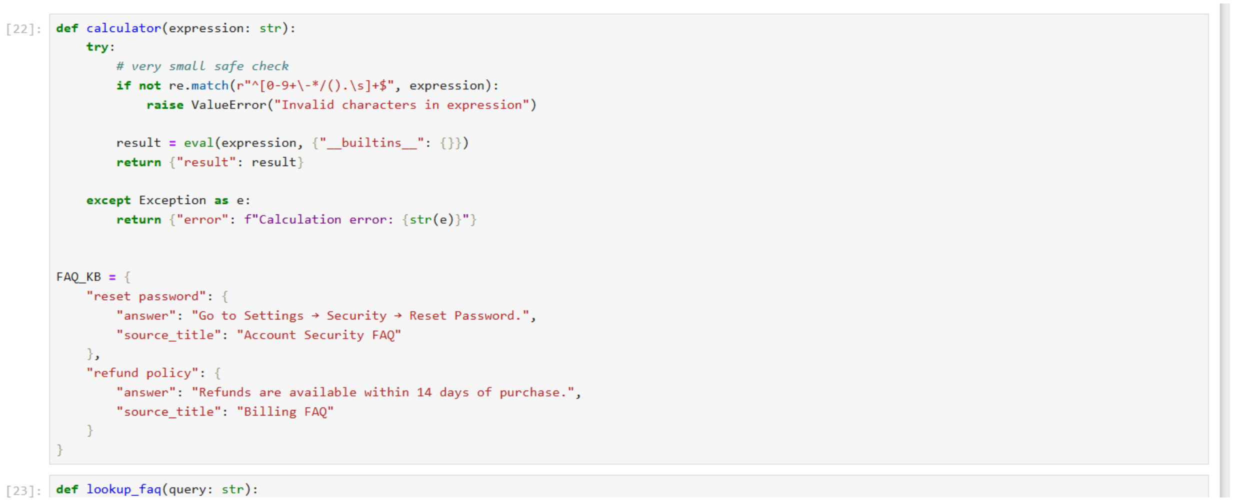
(A) get_time: supports ≥ 3 locations

(B) calc: handles invalid expressions cleanly

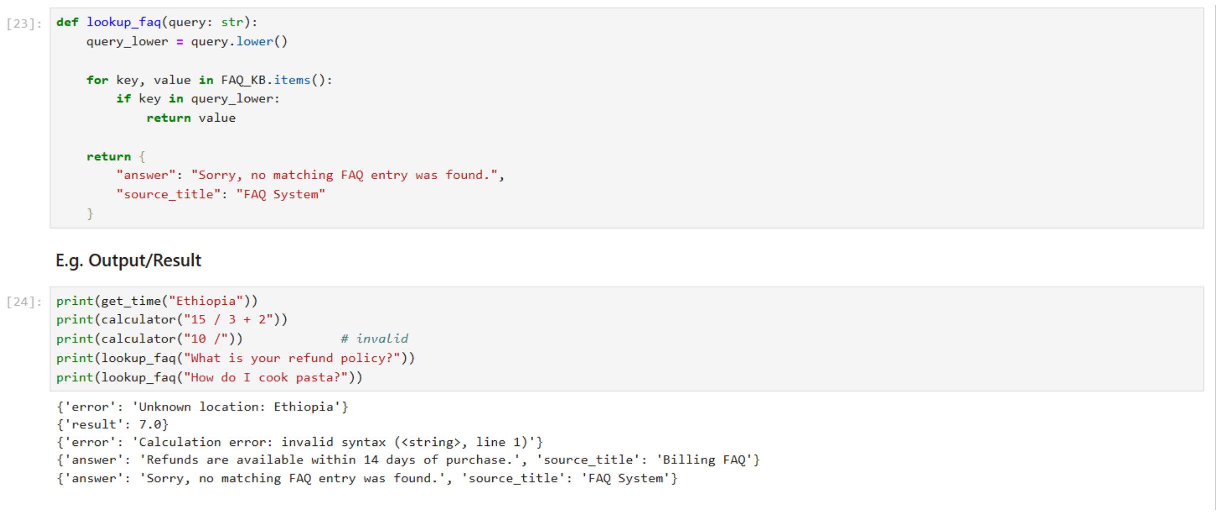
(C) lookup_faq: returns fallback message if no match

```
[12]: from datetime import datetime, timedelta
      import re
```

```
[21]: def get_time(location: str):
          timezones = {
              "UTC": 0,
              "Cape Town": 2,
              "New York": -5
          }

          if location not in timezones:
              return {"error": f"Unknown location: {location}"}

          offset = timezones[location]
          current_time = datetime.utcnow() + timedelta(hours=offset)

          return {
              "location": location,
              "time": current_time.strftime("%Y-%m-%d %H:%M:%S")
          }
```



```
[22]: def calculator(expression: str):
          try:
              # very small safe check
              if not re.match(r"^[0-9+\-*/().\s]+$", expression):
                  raise ValueError("Invalid characters in expression")

              result = eval(expression, {"__builtins__": {}})
              return {"result": result}

          except Exception as e:
              return {"error": f"Calculation error: {str(e)}"}


      FAQ_KB = {
          "reset password": {
              "answer": "Go to Settings → Security → Reset Password.",
              "source_title": "Account Security FAQ"
          },
          "refund policy": {
              "answer": "Refunds are available within 14 days of purchase.",
              "source_title": "Billing FAQ"
          }
      }

[23]: def lookup_faq(query: str):
```



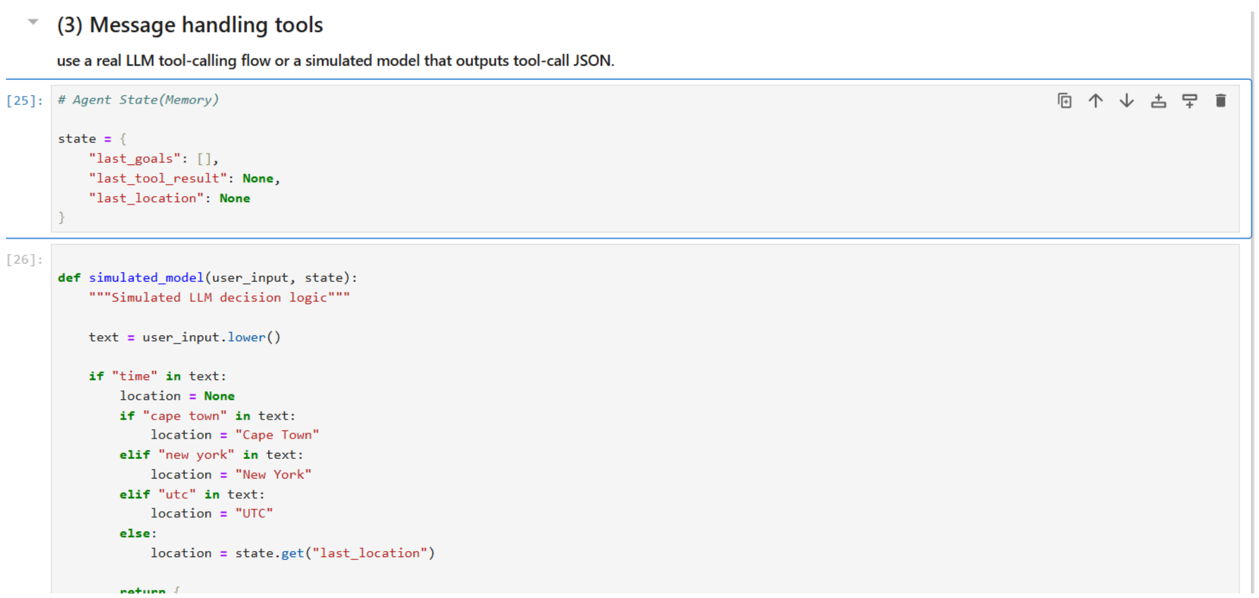
```
[23]: def lookup_faq(query: str):
          query_lower = query.lower()

          for key, value in FAQ_KB.items():
              if key in query_lower:
                  return value

          return {
              "answer": "Sorry, no matching FAQ entry was found.",
              "source_title": "FAQ System"
          }
```

E.g. Output/Result

```
[24]: print(get_time("Ethiopia"))
      print(calculator("15 / 3 + 2"))
      print(calculator("10 /"))             # invalid
      print(lookup_faq("What is your refund policy?"))
      print(lookup_faq("How do I cook pasta?"))

      {'error': 'Unknown location: Ethiopia'}
      {'result': 7.0}
      {'error': 'Calculation error: invalid syntax (<string>, line 1)'}
      {'answer': 'Refunds are available within 14 days of purchase.', 'source_title': 'Billing FAQ'}
      {'answer': 'Sorry, no matching FAQ entry was found.', 'source_title': 'FAQ System'}
```

Q.3. Build the message-handling loop:

➣ **(Ans),**

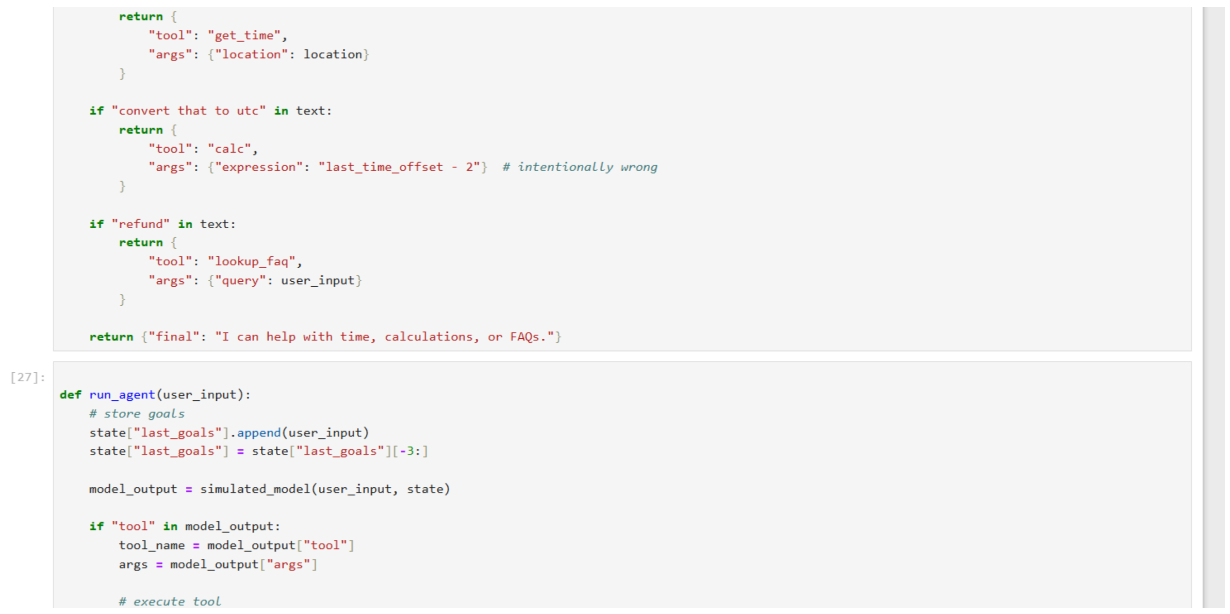


(3) Message handling tools

use a real LLM tool-calling flow or a simulated model that outputs tool-call JSON.

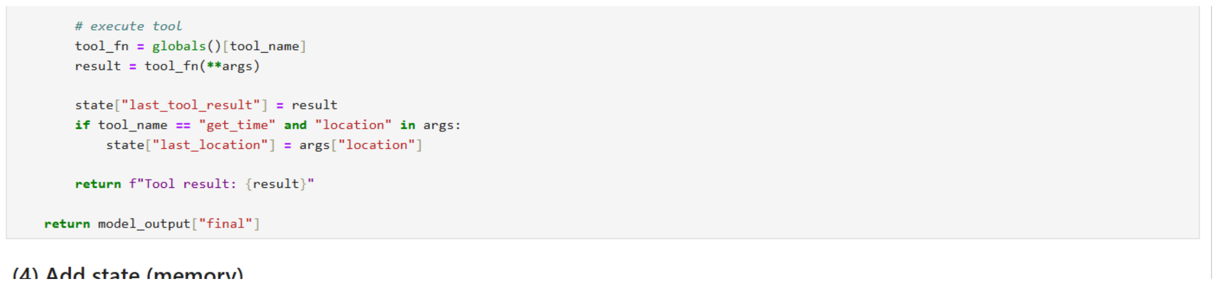
```
[25]: # Agent State(Memory)

      state = {
          "last_goals": [],
          "last_tool_result": None,
          "last_location": None
      }
```

```
[26]:
      def simulated_model(user_input, state):
          """Simulated LLM decision logic"""

          text = user_input.lower()

          if "time" in text:
              location = None
              if "cape town" in text:
                  location = "Cape Town"
              elif "new york" in text:
                  location = "New York"
              elif "utc" in text:
                  location = "UTC"
              else:
                  location = state.get("last_location")

              return {
```



```
              return {
                  "tool": "get_time",
                  "args": {"location": location}
              }

          if "convert that to utc" in text:
              return {
                  "tool": "calc",
                  "args": {"expression": "last_time_offset - 2"}  # intentionally wrong
              }

          if "refund" in text:
              return {
                  "tool": "lookup_faq",
                  "args": {"query": user_input}
              }

          return {"final": "I can help with time, calculations, or FAQs."}
```

```
[27]:
      def run_agent(user_input):
          # store goals
          state["last_goals"].append(user_input)
          state["last_goals"] = state["last_goals"][-3:]

          model_output = simulated_model(user_input, state)

          if "tool" in model_output:
              tool_name = model_output["tool"]
              args = model_output["args"]

              # execute tool
```

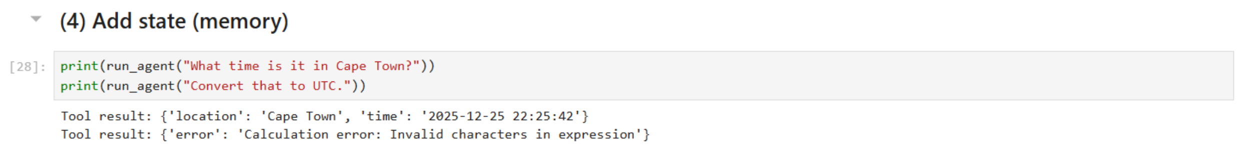


```
              # execute tool
              tool_fn = globals()[tool_name]
              result = tool_fn(**args)

              state["last_tool_result"] = result
              if tool_name == "get_time" and "location" in args:
                  state["last_location"] = args["location"]

              return f"Tool result: {result}"

          return model_output["final"]
```

(4) Add state (memory)

Q.4. Add state (memory).

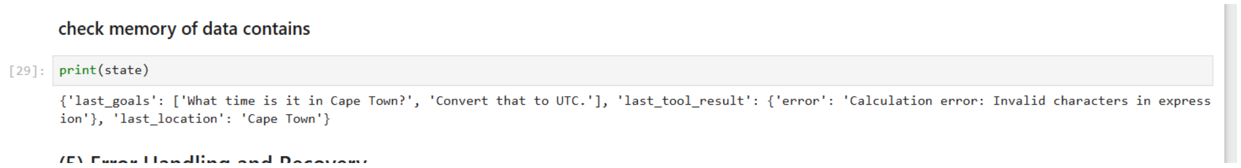
➣ (Ans)



(4) Add state (memory)

```
[28]: print(run_agent("What time is it in Cape Town?"))
      print(run_agent("Convert that to UTC."))

      Tool result: {'location': 'Cape Town', 'time': '2025-12-25 22:25:42'}
      Tool result: {'error': 'Calculation error: Invalid characters in expression'}
```

And then



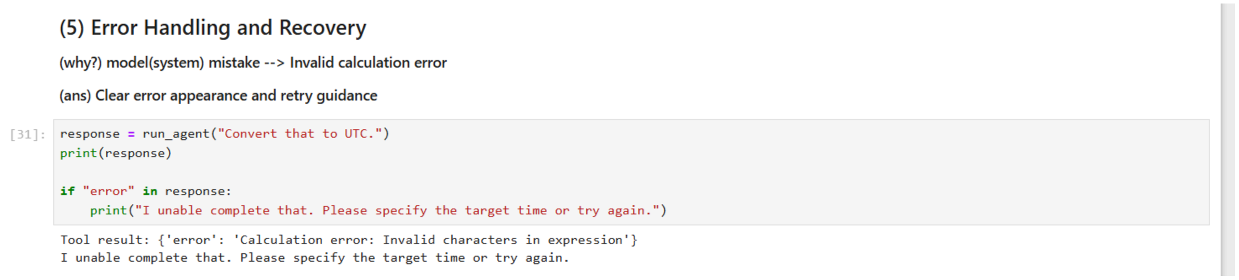
check memory of data contains

```
[29]: print(state)

      {'last_goals': ['What time is it in Cape Town?', 'Convert that to UTC.'], 'last_tool_result': {'error': 'Calculation error: Invalid characters in express
      ion'}, 'last_location': 'Cape Town'}
```

(5) Error Handling and Recovery

Q.5. Error handling and Recovery

⤐ (Ans)



(5) Error Handling and Recovery

(why?) model(system) mistake --> Invalid calculation error

(ans) Clear error appearance and retry guidance

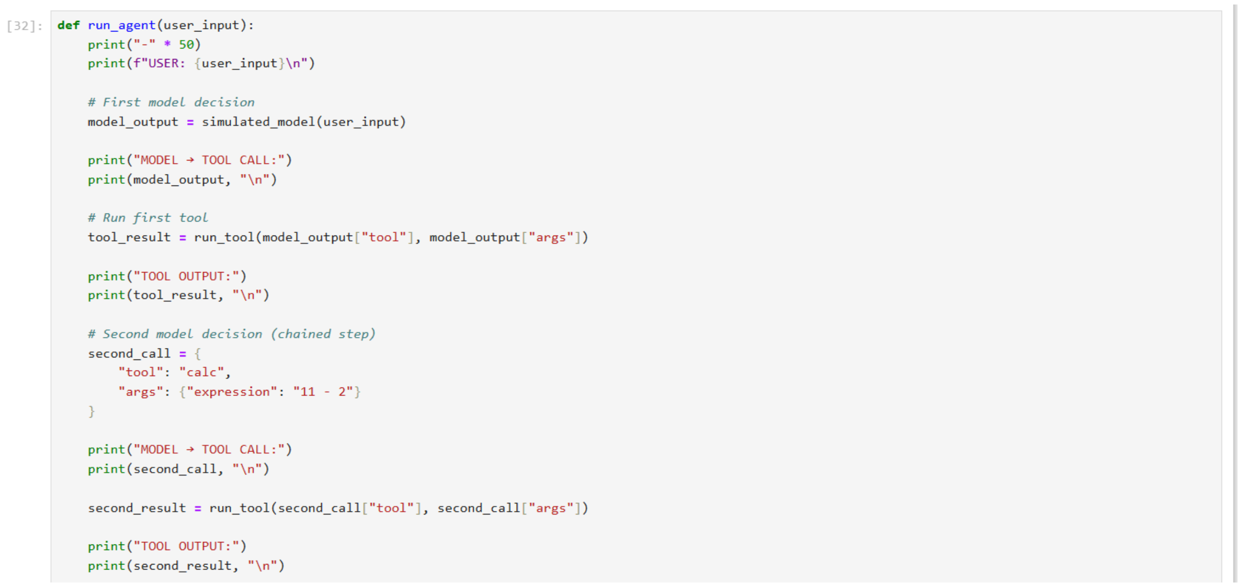
```
[31]: response = run_agent("Convert that to UTC.")
      print(response)

      if "error" in response:
          print("I unable complete that. Please specify the target time or try again.")

      Tool result: {'error': 'Calculation error: Invalid characters in expression'}
      I unable complete that. Please specify the target time or try again.
```

Problem C: Tool Triggering and QA Evidence:

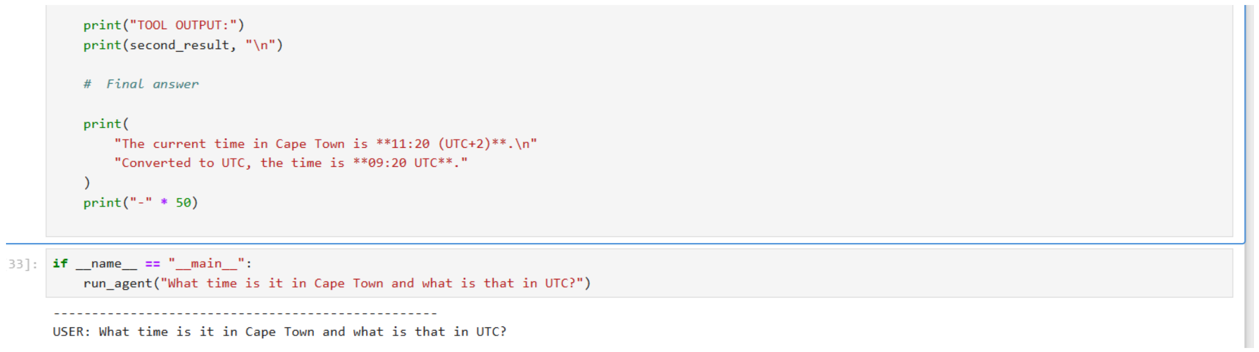
Q.1. Trigger two multi-step tasks:

⤐ (Ans).

<u>Task 1</u>⤐

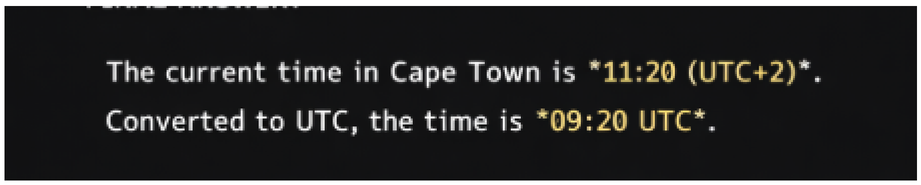


```
[32]: def run_agent(user_input):
          print("-" * 50)
          print(f"USER: {user_input}\n")

          # First model decision
          model_output = simulated_model(user_input)

          print("MODEL → TOOL CALL:")
          print(model_output, "\n")

          # Run first tool
          tool_result = run_tool(model_output["tool"], model_output["args"])

          print("TOOL OUTPUT:")
          print(tool_result, "\n")

          # Second model decision (chained step)
          second_call = {
              "tool": "calc",
              "args": {"expression": "11 - 2"}
          }

          print("MODEL → TOOL CALL:")
          print(second_call, "\n")

          second_result = run_tool(second_call["tool"], second_call["args"])

          print("TOOL OUTPUT:")
          print(second_result, "\n")
```



```
    print("TOOL OUTPUT:")
    print(second_result, "\n")

    #  Final answer

    print(
        "The current time in Cape Town is **11:20 (UTC+2)**.\n"
        "Converted to UTC, the time is **09:20 UTC**."
    )
    print("-" * 50)

33]: if __name__ == "__main__":
         run_agent("What time is it in Cape Town and what is that in UTC?")

     --------------------------------------------------
     USER: What time is it in Cape Town and what is that in UTC?
```

Q.2, The Result



```
The current time in Cape Town is *11:20 (UTC+2)*.
Converted to UTC, the time is *09:20 UTC*.
```

Part D: Reflection

Q.1. Trust breakdowns. List 3 ways your agent could lose user trust, and how you'd fix each using Space–Time–Core.

⤜ (Ans):

User trust is fragile in agent systems. Below are three realistic trust failure modes, each mapped to a Space–Time–Core fix.

⇐==>

Trust Breakdown 1: Acting Beyond Allowed Access (Space Violation)

What goes wrong
The agent implies it can access private or real-world systems (e.g., bank accounts, emails, live calendars) when it cannot or should not.

Why trust is lost
Users feel misled or unsafe when the agent overclaims its reach.

STC Fix – Space

- **Explicitly declare accessible vs non-accessible resources**
- **Enforce tool whitelisting**
- **Surface access boundaries in responses**

Example Fix

> **"I can analyze expenses you provide, but I cannot access your bank account directly."**

✔ Clear capability boundaries
✔ No implied surveillance
✔ Predictable behavior

⇐==>

Trust Breakdown 2: Inconsistent or Incorrect Follow-Ups (Time Failure)

What goes wrong
The agent forgets context or misuses stored state (e.g., converting “that” to the wrong location or time).

Why trust is lost
Users perceive the agent as unreliable or careless.

STC Fix – Time

- Store structured summaries, not raw logs
- Track reference anchors (last location, last number)
- Add validation before reuse

Example Fix

```
If last_location exists → reuse

Else → ask for clarification
```

✔ Reduced hallucinated continuity
✔ Transparent memory use
✔ Safer follow-up resolution

<===>

Trust Breakdown 3: Silent Errors or Confident Wrong Answers (Core Failure)

What goes wrong
The agent returns a plausible but incorrect answer or hides uncertainty when a tool fails.

Why trust is lost
Confidence without correctness damages credibility faster than refusal.

STC Fix – Core

- Mandatory explanation for derived results

- Explicit uncertainty signaling
- Refusal on low-confidence states

Example Fix

> "I may be mistaken because the location was ambiguous. Could you confirm?"

✔ Honest uncertainty
✔ Predictable refusal behavior
✔ User-aligned confidence

Q2. Biggest implementation bottleneck. Pick one: schemas, orchestration loop, state, or error handling. Briefly describe what broke first and what you changed.

2. Biggest Implementation Bottleneck

Chosen bottleneck: Error handling

What Broke First

At early stages, the agent:

- Passed invalid arguments to tools
- Crashed or returned raw exceptions
- Repeated failures without recovery

Example failure:

```
{ "tool": "calc", "args": { "expression": "10 /" } }
```

Result:

- **Tool failure**
- **No user guidance**
- **Trust erosion**

>>(?)Why Error Handling Was the Hardest

Unlike schemas or state:

- **Errors occur across layers (model ↠ tool ↠ memory)**
- **One failure can cascade**
- **Not all errors are equal or recoverable**

Error handling required:

- **Classification**
- **Policy decisions**
- **Recovery strategies**

What I Changed

1. **Introduced error taxonomy**
    - **ModelError, ToolError, ValidationError, SystemError**

2. **Standardized error envelopes**

```
{
  "status": "error",
  "recoverable": True,
  "hint": "Try a valid expression like 10 / 2"
}
```

3. **Added pre-execution validation**

- **Stop bad tool calls early**

4. **Implemented retry & fallback rules**

- **No infinite loops**
- **Escalate to user when needed**

Resulting Improvement

Before	**After**
Crashes	**Graceful recovery**
Silent failures	**Clear guidance**

Repeated errors	**Controlled retries**

Fixing error handling stabilized the entire agent.

Thank You!