COMP4901B: Large Language Models

Assignment 4 Report

HE, Wenqian
Student ID: 20860896

November 29, 2025

1 Step 1: Implement the Python Code Execution Tool

Please see 1, 2, and 3 for the implementation of the code execution tool.
The main logic of my implementation is:

1. Force the agent to print the result using the print() function into stdout.
2. Ban the use of sensitive calls like open(), input(), eval(), exec(), compile(), and __import__().
3. Ban the import of other modules than the pre-imported modules, using regex to match the import statement.
4. Initialize a subprocess of python interpreter using `uv run python` and run the code in the subprocess, so as to capture stdout and stderr.
5. Validation error, stdout, and stderr are put into the response of the tool.
6. If the code execution result is empty, return a system reminder to the agent to notify the agent that the result might not be printed correctly, and remind the agent to retry.

```python
def validate_code(code: str):
    """
    Validates the Python code by checking for forbidden keywords.
    """

    IMPORT_PATTERN = r"^import (\w+)\.?"
    FROM_PATTERN = r"^from (\w+)\.?"
    ALLOWED_MODULES = ["math", "fractions", "itertools", "sympy", "numpy"]
    FORBIDDEN_CALLS = ["open(", "input(", "eval(", "exec(", "compile(", "__import__"]

    if "print(" not in code:
        raise ValueError(
            "Print statement `print()` is required. Otherwise, the code execution will not return any output."
        )

    for line in code.split("\n"):
        if m := re.match(IMPORT_PATTERN, line) or re.match(FROM_PATTERN, line):
            if m[1] not in ALLOWED_MODULES:
                raise ValueError(
                    f"You are not allowed to import {m[1]}. Use the provided modules: math, fractions, itertools, sympy (alias sp), numpy (alias np)."
                )

    for call in FORBIDDEN_CALLS:
        if call in code:
            raise ValueError(f"keyword '{call}' is not allowed.")
```

Figure 1: Code Execution Tool: `validate_code()`

```python
CODE_TEMPLATE = """
import math
import fractions
import itertools
import sympy
import sympy as sp
import numpy
import numpy as np

{code}
"""


def execute_python_code_impl(code: str):
    try:
        validate_code(code)

        code_block = CODE_TEMPLATE.format(code=code)
        result = subprocess.run(
            ["uv", "run", "python", "-c", code_block],
            capture_output=True,
            text=True,
            check=True,
            timeout=30,
        )
        return result.stdout
    except subprocess.TimeoutExpired:
        return "Error: Code execution timed out (limit: 30 seconds)."
    except subprocess.CalledProcessError as e:
        return f"Error executing code:\n{e.stderr}"
    except ValueError as e:
        return f"Security Error: {str(e)}"
    except Exception as e:
        return f"Error executing Python code: {e}"
```

Figure 2: Code Execution Tool: `execute_python_code_impl()`

```python
@tool
def execute_python_code(code: str, runtime: ToolRuntime):
    """
    Execute Python code to perform calculations, verify solutions, and explore patterns.

    This tool allows you to run Python scripts in a sandboxed environment.
    Use it to:
    - Perform complex calculations (symbolic or numerical)
    - Verify mathematical derivations
    - Brute-force small search spaces using itertools
    - Solve equations using sympy

    The following modules are PRE-IMPORTED and available for use:
    - `math`: Standard mathematical functions
    - `fractions`: Rational number arithmetic
    - `itertools`: efficient looping
    - `sympy` (aliased as `sp`): Symbolic mathematics (simplify, solve, etc.)
    - `numpy` (aliased as `np`): Numerical arrays and operations
    ```

    Usage Rules:
    1. **NO Imports**: Do not other modules than the pre-imported modules.
    2. **Print Results**: The tool captures `stdout`. You MUST `print()` variables to see them.
    3. **No I/O**: `input()`, `open()`, and file system access are forbidden.
    4. **Stateless**: Each execution is independent. Variables do not persist between calls. Include all necessary logic/variables in `code`.

    Args:
        code: The Python code to execute.
    """
    result = execute_python_code_impl(code)
    content = result.strip()

    if not content:
        content = "<system_reminder>The code executed successfully but returned no output. Did you print() the result correctly?</system_reminder>"

    return Command[tuple[()]](
        update={
            "messages": [
                ToolMessage(
                    content=content,
                    tool_call_id=runtime.tool_call_id,
)
            ],
            "steps": [
                Step(
                    step_number=runtime.state["current_step"],
                    reasoning=[],
                    actions=[
                        CodeExecutionAction(action="code_execution", code=code, output=result)
                    ],
)
            ],
        }
)
```

Figure 3: Code Execution Tool: `execute_python_code()`
```

2 Step 2: Adapt Your Agent Loop for AIME

I used LangGraph to build a typical agent loop as shown in figure 4.
Here are some important details:

1. I maintained a `answer` field in the agent state, which is originally set to `None`. The loop end condition is that the `answer` field is not `None`.

2. If the agent cannot find the answer within the maximum number of steps, the `answer` field is set to
`boxedfailure`, and the loop ends.

3. If the agent finds the answer, it should call the `submit_answer` tool (see 5) to submit the answer, which will update the `answer` field to the answer to meet the end loop condition. To improve the answer extraction, the `submit_answer` tool requires the answer to be wrapped in
`boxed` format, otherwise the tool will return an error and remind the agent to wrap the answer correctly.

4. In each step, a system reminder to remind the agent what is the question and it should submit the answer if answer is found is appended at the end of the history of messages to the LLM (see 6). This is to ensure the agent does not forget the question and the answer submission requirement after long reasoning.

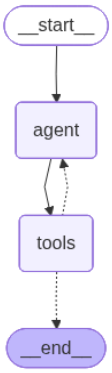

Figure 4: Agent Loop

```python
@tool
def submit_answer(content: str, runtime: ToolRuntime):
    """Provide the final short answer.

    Args:
        content: The short answer text. It must end with the final answer wrapped in a latex box, e.g. "The velocity is $\boxed{42}$".
    """

    # Check if the mandatory \boxed{ format is present
    if "\\boxed{" not in content:
        return (
            "Error: The answer is not formatted with \\boxed{}. "
            'Please wrap your final numeric answer in \\boxed{}, e.g., "The answer is \\boxed{42}".'
        )

    # Basic check to ensure the box isn't empty
    match = re.search(r"\\boxed\{(.*?)\}", content, re.DOTALL)
    if match and not match.group(1).strip():
        return "Error: The \\boxed{} tag is empty. Please put the answer inside."

    return Command[tuple[()]](
        update={
            "messages": [
                ToolMessage(
                    content="The final solution is successfully submitted: " + content,
                    tool_call_id=runtime.tool_call_id,
                )
            ],
            "answer": content,
        }
    )
```

Figure 5: Submit Answer Tool

```python
system_reminder = f"<system_reminder>The current question your are investigating is: [{state['question']}]. If you have not yet found out the answer, please ignore this system reminder and continue to make use of tools provided to you (if any) to gather more information and think about how to answer the question. If you are confident that you have found out the answer, please use `submit_answer` tool to submit the answer concisely.</system_reminder>"

ai_message = invoke_llm(
    [
        SystemMessage(content=system_prompt),
        *state["messages"],
        HumanMessage(content=system_reminder),
    ],
    state=state,
)
return {
    "messages": [ai_message],
    "current_step": state["current_step"] + 1,
    "steps": [
        Step(
            step_number=state["current_step"] + 1,
            reasoning=[ai_message.content],
            actions=[],
        )
    ],
}
```

Figure 6: Agent Reminder In Each Step

3 Step 3: Evaluate your agent w/ and w/o tool calling

I evaluated the agent with and without the python code execution tool. Noticed that the `submit_answer` tool is used in both settings for final answer collection.

3.1 Without Tool (Baseline)

See figure 7 for the evaluation result. The accuracy is 67.50%, which is the baseline accuracy.

```
======================================================================
EVALUATION METRICS
======================================================================

📊 Basic Metrics:
  Total entries: 120
  Correct answers: 81
  Overall accuracy: 67.50%

📈 Rollout Statistics:
  Unique problems: 30
  Avg rollouts per problem: 4.0

🎯 Pass@k Metrics (estimated):
  pass@1: 67.50%
  pass@2: 77.78%
  pass@3: 80.83%
  pass@4: 83.33%
======================================================================
```

Figure 7: Evaluation Result Without Python Code Execution Tool

3.2 With Tool Calling

See figure 8 for the evaluation result. The accuracy is 76.67%, which is the accuracy with the python code execution tool.

```
======================================================================
EVALUATION METRICS
======================================================================

📊 Basic Metrics:
  Total entries: 120
  Correct answers: 92
  Overall accuracy: 76.67%

📈 Rollout Statistics:
  Unique problems: 30
  Avg rollouts per problem: 4.0

🎯 Pass@k Metrics (estimated):
  pass@1: 76.67%
  pass@2: 84.44%
  pass@3: 85.83%
  pass@4: 86.67%
======================================================================
```

Figure 8: Evaluation Result With Python Code Execution Tool