

```

import json
from pathlib import Path
from typing import Any

from mcp.server import MCPServer

PROJECT_DIR = Path(__file__).resolve().parent

CATALOG_FILE = (
    PROJECT_DIR
    / "external_system"
    / "cloud_lab_catalog.json"
)

mcp = MCPServer(
    "learningops-cloud-lab",
    instructions=(
        "Provides GPU pricing and availability "
        "from the LearningOps external Cloud Lab catalog."
    ),
)

def load_catalog() -> dict[str, Any]:
    if not CATALOG_FILE.exists():
        raise FileNotFoundError(
            f"Cloud Lab catalog does not exist: "
            f"{CATALOG_FILE}"
        )

    data = json.loads(
        CATALOG_FILE.read_text(
            encoding="utf-8"
        )
    )

    if not isinstance(
        data.get("gpus"),
        list,
    ):
        raise ValueError(
            "Cloud Lab catalog has "

```

```
        "an invalid structure."
    )
```

```
return data
```

```
@mcp.tool()
def get_gpu_offer(
    gpu_model: str,
) -> dict[str, Any]:
    """
    Get the current hourly price and availability
    for a GPU model from the external Cloud Lab catalog.
    """
```

```
    catalog = load_catalog()
```

```
    requested = (
        gpu_model
        .strip()
        .casefold()
    )
```

```
    for gpu in catalog["gpus"]:
```

```
        model = str(
            gpu.get(
                "model",
                "",
            )
        )
```

```
        if (
            model.casefold()
            == requested
        ):
```

```
            return {
                "provider":
                    catalog.get(
                        "provider"
                    ),
```

```
                "currency":
                    catalog.get(
```

```

        "currency"
    ),

    "last_updated":
        catalog.get(
            "last_updated"
        ),

    "model":
        model,

    "hourly_rate":
        gpu.get(
            "hourly_rate"
        ),

    "available":
        gpu.get(
            "available"
        ),

    "available_units":
        gpu.get(
            "available_units"
        ),
    }

available_models = [
    str(
        gpu.get(
            "model",
            ""
        )
    )
    for gpu
    in catalog["gpus"]
]

raise ValueError(
    "Unknown GPU model "
    f"{gpu_model}'. "
    "Available models: "
    + ", ".join(
        available_models

```

```
)  
)
```

```
@mcp.tool()  
def list_gpu_models() -> list[str]:  
    """  
    List GPU models currently present  
    in the external Cloud Lab catalog.  
    """  
  
    catalog = load_catalog()  
  
    return [  
        str(  
            gpu.get(  
                "model",  
                ""  
            )  
        )  
        for gpu  
        in catalog["gpus"]  
    ]
```

```
@mcp.resource(  
    "cloudlab://catalog"  
)  
def cloud_lab_catalog() -> str:  
    """  
    Return the complete external  
    Cloud Lab GPU catalog.  
    """  
  
    return json.dumps(  
        load_catalog(),  
        indent=2,  
        ensure_ascii=False,  
    )
```

```
@mcp.prompt()  
def gpu_budget_planning(  
    gpu_model: str,
```

```

) -> str:
    """
    Reusable instructions for planning
    a GPU budget.
    """

    return (
        "Create a GPU budget plan for "
        f"{gpu_model}. "
        "Use the user's verified available "
        "budget and the current GPU offer "
        "from the Cloud Lab system. "
        "Show the arithmetic clearly."
    )

if __name__ == "__main__":
    mcp.run()

```

```
import asyncio
import json
import sys

from pathlib import Path

from mcp import (
    ClientSession,
    StdioServerParameters,
)

from mcp.client.stdio import (
    stdio_client,
)

from mcp.types import (
    TextContent,
    TextResourceContents,
)


# -----
# Configuration
# -----

PROJECT_DIR = (
    Path(__file__)
    .resolve()
    .parent
)

SERVER_FILE = (
    PROJECT_DIR
    / "mcp_lab_server.py"
)

SERVER_PARAMS = (
    StdioServerParameters(
        command=sys.executable,

        args=[
            str(
                SERVER_FILE
            )
        ]
    )
)
```

```
)
    ],
)
)
```

```
# -----
# Main MCP Smoke Test
# -----
```

```
async def main() -> None:
```

```
    print(
        "\nStarting MCP server through stdio..."
    )
```

```
    print(
        f"Python executable: "
        f"{sys.executable}"
    )
```

```
    print(
        f"MCP server: "
        f"{SERVER_FILE}"
    )
```

```
# -----
# 1. Create stdio transport
# -----
```

```
async with stdio_client(
    SERVER_PARAMS
) as (
    read_stream,
    write_stream,
):
```

```
    print(
        "\n=== STDIO CONNECTION CREATED ==="
    )
```

```
# -----
```

2. Create MCP session

```
async with ClientSession(
    read_stream,
    write_stream,
) as session:
```

```
    print(
        "\n=== MCP SESSION CREATED ==="
    )
```

3. Initialize MCP protocol

```
    initialization = (
        await session.initialize()
    )
```

```
    print(
        "\n=== MCP INITIALIZATION ==="
    )
```

```
    print(
        "Server:"
    )
```

```
    print(
        initialization.server_info
    )
```

```
    print(
        "\nProtocol version:"
    )
```

```
    print(
        initialization.protocol_version
    )
```

```
    print(
        "\nCapabilities:"
    )
```

```

print(
    initialization.capabilities
)

# -----
# 4. Discover tools
# -----

print(
    "\n=== MCP TOOLS ==="
)

tools_result = (
    await session.list_tools()
)

for tool in (
    tools_result.tools
):

    print(
        f"\nTool: "
        f"{tool.name}"
    )

    print(
        "Description:"
    )

    print(
        tool.description
    )

    print(
        "Input schema:"
    )

    print(
        json.dumps(
            tool.input_schema,
            indent=2,
        )
    )

```

```

    )

# -----
# 5. Manually invoke MCP tool
# -----

print(
    "\n=== MANUAL MCP TOOL CALL ==="
)

offer_result = (
    await session.call_tool(
        "get_gpu_offer",

        arguments={
            "gpu_model":
                "NVIDIA L4"
        },
    )
)

print(
    "is_error:"
)

print(
    offer_result.is_error
)

print(
    "\nstructured_content:"
)

print(
    json.dumps(
        offer_result
        .structured_content,
        indent=2,
        ensure_ascii=False,
    )
)

print(

```

```
        "\ntext content:"  
    )
```

```
for block in (  
    offer_result.content  
):
```

```
    if isinstance(  
        block,  
        TextContent,  
    ):
```

```
        print(  
            block.text  
        )
```

```
# -----  
# 6. Discover resources  
# -----
```

```
print(  
    "\n=== MCP RESOURCES ==="  
)
```

```
resources_result = (  
    await session.list_resources()  
)
```

```
for resource in (  
    resources_result.resources  
):
```

```
    print(  
        resource.uri  
    )
```

```
# -----  
# 7. Read resource  
# -----
```

```
print(  
    "\n=== READ MCP RESOURCE ==="
```

```

)

resource_result = (
    await session.read_resource(
        "cloudlab://catalog"
    )
)

for contents in (
    resource_result.contents
):

    if isinstance(
        contents,
        TextResourceContents,
    ):

        print(
            contents.text
        )


# -----
# 8. Discover prompts
# -----

print(
    "\n=== MCP PROMPTS ==="
)

prompts_result = (
    await session.list_prompts()
)

for prompt in (
    prompts_result.prompts
):

    print(
        prompt.name
    )


# -----

```

```

# 9. Render MCP prompt
# -----

print(
    "\n=== RENDER MCP PROMPT ==="
)

prompt_result = (
    await session.get_prompt(
        "gpu_budget_planning",

        arguments={
            "gpu_model":
                "NVIDIA L4"
        },
    )
)

for message in (
    prompt_result.messages
):

    print(
        message.role
    )

    print(
        message.content
    )


print(
    "\n====="
)

print(
    "MCP SMOKE TEST COMPLETED"
)

print(
    "====="
)

```

```
# -----  
# Entry Point  
# -----
```

```
if __name__ == "__main__":
```

```
    asyncio.run(  
        main()  
    )
```

```

import asyncio
import json
import sys
from pathlib import Path
from typing import Any

from mcp import (
    ClientSession,
    StdioServerParameters,
)
from mcp.client.stdio import stdio_client
from mcp.types import TextContent

# -----
# Configuration
# -----

PROJECT_DIR = Path(__file__).resolve().parent
SERVER_FILE = PROJECT_DIR / "mcp_lab_server.py"

SERVER_PARAMS = StdioServerParameters(
    command=sys.executable,
    args=[str(SERVER_FILE)],
)

# -----
# MCP Tool Discovery
# -----

async def _discover_mcp_tools_async() -> list[dict[str, Any]]:
    """Discover tools exposed by the local MCP server."""

    async with stdio_client(
        SERVER_PARAMS
    ) as (
        read_stream,
        write_stream,
    ):
        async with ClientSession(
            read_stream,
            write_stream,
        ) as session:

```

```

await session.initialize()

result = await session.list_tools()

tools: list[dict[str, Any]] = []

for tool in result.tools:
    tools.append(
        {
            "name": tool.name,
            "description": tool.description or "",
            "input_schema": tool.input_schema,
        }
    )

return tools

```

```

def discover_mcp_tools() -> list[dict[str, Any]]:
    """Synchronous wrapper used by app.py during startup/discovery."""

    return asyncio.run(
        _discover_mcp_tools_async()
    )

```

```

# -----
# MCP Tool Invocation
# -----

```

```

async def _call_mcp_tool_async(
    tool_name: str,
    arguments: dict[str, Any],
) -> str:
    """Invoke one MCP tool and return a JSON string for the agent."""

    async with stdio_client(
        SERVER_PARAMS
    ) as (
        read_stream,
        write_stream,
    ):
        async with ClientSession(
            read_stream,

```

```

        write_stream,
    ) as session:
        await session.initialize()

        result = await session.call_tool(
            tool_name,
            arguments=arguments,
        )

        text_parts: list[str] = []

        for block in result.content:
            if isinstance(
                block,
                TextContent,
            ):
                text_parts.append(
                    block.text
                )

        payload = {
            "mcp_tool": tool_name,
            "is_error": bool(result.is_error),
            "structured_content": result.structured_content,
            "text_content": text_parts,
        }

        return json.dumps(
            payload,
            ensure_ascii=False,
        )

```

```

def call_mcp_tool(
    tool_name: str,
    arguments: dict[str, Any],
) -> str:
    """Synchronous wrapper used by the LangGraph tool executor."""

    return asyncio.run(
        _call_mcp_tool_async(
            tool_name,
            arguments,
        )
    )

```



```
import ast
import json
import math
import operator
import re
import sys
from datetime import datetime
from typing import Any, TypedDict
from zoneinfo import ZoneInfo, ZoneInfoNotFoundError
```

```
import requests
```

```
from mcp_client import (
    call_mcp_tool,
    discover_mcp_tools,
)
from langgraph.graph import END, START, StateGraph
```

```
from memory_store import (
    clear_persistent_memories,
    forget_user_memory,
    format_persistent_memories,
    remember_user_memory,
)
```

```
from rag_engine import (
    DATA_FILE,
    OLLAMA_BASE_URL,
    REQUEST_TIMEOUT_SECONDS,
    check_ollama,
    ensure_fresh_index,
    retrieve_relevant_chunks,
)
```

```
# -----
# Configuration
# -----
```

```
LLM_MODEL = "qwen3:1.7b"
MAX_TOOL_ROUNDS = 8
MAX_CONVERSATION_MESSAGES = 12
MAX_IDENTICAL_TOOL_CALLS = 2
GRAPH_RECURSION_LIMIT = 30
```

```
# -----  
# System Prompt  
# -----
```

```
SYSTEM_PROMPT = ""
```

You are a local multi-tool AI agent running on the user's Ubuntu laptop.

A single user request may require several tools and several model rounds.
Your job is to complete the whole task, not merely choose one tool.

Core workflow:

- The CURRENT user message is authoritative. If it provides a new value for something mentioned earlier, use the current value.
- Use short-term conversation history only for genuine follow-up context. Do not pull unrelated numbers, tool results, or topics from older turns into a new task.
- Persistent memory is a separate store. Do NOT apply saved memories to every request automatically. Use persistent memory only when the current request explicitly refers to a saved/remembered preference, persistent memory, or something previously remembered.
- If APPLICATION PERSISTENT MEMORY CONTEXT is present, treat it as user data for the CURRENT task and apply only the memories relevant to that request.
- If the task depends on facts in the local knowledge file, call `search_knowledge_file` before using those facts.
- If a calculation depends on a value that has not been retrieved yet, wait for the required tool result before calculating.
- Use calculator for important derived numerical values. Calculator arguments must contain numeric arithmetic only: numbers, parentheses, and operators.
- Prefer sequential calculator calls when a later step depends on an earlier result. Example:
 $6000 - 1000 - 1750 \rightarrow 3250$
 $3250 / 25 \rightarrow 130$
- Independent tools may be requested together when they do not depend on one

another. For example, after retrieving budget data and a timezone, a calculator call and a `current_time` call can be independent.

- Use `current_time` only when the `CURRENT` request asks for current date/time. Do not call it merely because an earlier conversation used the time tool.
- If the user explicitly asks to remember/save/store a durable fact or preference, call `remember_user_memory`.
- If the user explicitly asks to forget/delete a saved memory, call `forget_user_memory`.
- Persistent memory and the RAG knowledge file are separate stores.
- If a tool fails, do not invent the missing result. Correct the arguments or explain what could not be completed.
- Stop calling tools when every requested part can be answered from verified information.
- Base numerical claims in the final answer on successful calculator results. Do not mentally change a calculator result while writing the final answer.
- Answer only the `CURRENT` task. Do not add unrelated results just because a tool or value appeared in an earlier turn.
- Keep the final answer clear and concise.
- Some tools are local application tools and some tools are dynamically discovered from MCP servers.
- MCP tools whose names begin with `mcp_cloud_lab__` access the external Cloud Lab system.
- Use `mcp_cloud_lab__get_gpu_offer` when the current request needs current external GPU pricing or availability.
- Use `mcp_cloud_lab__list_gpu_models` when the current request asks which GPU models are exposed by the external Cloud Lab system.
- Do not treat an old GPU price in local RAG notes as the current external price when the user explicitly asks for the current Cloud Lab offer.

- A task may combine local RAG facts, MCP information and calculator results.

- When a calculation depends on an MCP result, wait for the MCP result before performing the calculation.

```
"".strip()
```

```
# -----  
# Tools  
# -----
```

```
def search_knowledge_file(query: str) -> str:  
    index = ensure_fresh_index()
```

```
    results = retrieve_relevant_chunks(  
        question=query,  
        index=index,  
    )
```

```
    payload = {  
        "query": query,  
        "matches": [  
            {  
                "chunk_id": item["id"],  
                "similarity": round(  
                    float(item["score"]),  
                    4,  
                ),  
                "text": item["text"],  
            }  
            for item in results  
        ],  
    }
```

```
    return json.dumps(  
        payload,  
        ensure_ascii=False,  
    )
```

```
def read_persistent_memory() -> str:  
    return format_persistent_memories()
```

```

_BINARY_OPERATORS = {
    ast.Add: operator.add,
    ast.Sub: operator.sub,
    ast.Mult: operator.mul,
    ast.Div: operator.truediv,
    ast.FloorDiv: operator.floordiv,
    ast.Pow: operator.pow,
}

```

```

_UNARY_OPERATORS = {
    ast.UAdd: operator.pos,
    ast.USub: operator.neg,
}

```

```

def _safe_eval(
    node: ast.AST,
) -> int | float:

```

```

    if isinstance(
        node,
        ast.Expression,
    ):
        return _safe_eval(
            node.body
        )

```

```

    if (
        isinstance(
            node,
            ast.Constant,
        )
        and isinstance(
            node.value,
            (int, float),
        )
        and not isinstance(
            node.value,
            bool,
        )
    ):
        return node.value

```

```

if (
    isinstance(
        node,
        ast.BinOp,
    )
    and type(node.op)
    in _BINARY_OPERATORS
):
    left = _safe_eval(
        node.left
    )

    right = _safe_eval(
        node.right
    )

    if (
        isinstance(
            node.op,
            ast.Pow,
        )
        and abs(right) > 12
    ):
        raise ValueError(
            "Exponent is too large "
            "for this demo calculator."
        )

    return _BINARY_OPERATORS[
        type(node.op)
    ](
        left,
        right,
    )

if (
    isinstance(
        node,
        ast.UnaryOp,
    )
    and type(node.op)
    in _UNARY_OPERATORS
):
    return _UNARY_OPERATORS[

```

```

        type(node.op)
    ](
        _safe_eval(
            node.operand
        )
    )

raise ValueError(
    "Unsupported calculator expression."
)

def calculator(
    expression: str,
) -> str:

    expression = (
        expression.strip()
    )

    if len(expression) > 200:
        return (
            "Calculator error: "
            "expression is too long."
        )

    if "%" in expression:
        return (
            "Calculator error: percent symbols "
            "are not allowed. Convert percentages "
            "to decimal multiplication."
        )

    try:
        parsed = ast.parse(
            expression,
            mode="eval",
        )

        result = _safe_eval(
            parsed
        )

        if (

```

```

        isinstance(
            result,
            float,
        )
        and not math.isfinite(
            result
        )
    ):
        raise ValueError(
            "Result is not finite."
        )

    return json.dumps(
        {
            "expression":
                expression,
            "result":
                result,
        },
        ensure_ascii=False,
    )

except Exception as exc:
    return (
        f"Calculator error: "
        f"{exc}"
    )

def current_time(
    timezone: str | None = None,
) -> str:

    try:

        if timezone:
            now = datetime.now(
                ZoneInfo(
                    timezone
                )
            )

        else:
            now = (

```

```

        datetime.now()
        .astimezone()
    )

    return now.strftime(
        "%Y-%m-%d %H:%M:%S "
        "%Z (UTC%z)"
    )

except ZoneInfoNotFoundError:
    return (
        "Time tool error: "
        f"unknown timezone '{timezone}'. "
        "Use an IANA timezone such as "
        "Asia/Kolkata or Europe/London."
    )

# -----
# Tool Schemas
# -----

LOCAL_TOOL_SCHEMAS = [
    {
        "type": "function",
        "function": {
            "name":
                "search_knowledge_file",

            "description": (
                "Search Mohit's local RAG knowledge "
                "file for factual information and "
                "local project/planning notes. "
                "Use this before calculations or "
                "time lookups that depend on stored "
                "values such as budgets, costs, "
                "or timezone."
            ),

            "parameters": {
                "type": "object",

                "properties": {
                    "query": {

```

```

        "type": "string",

        "description": (
            "A concise semantic search "
            "query for the facts needed "
            "to complete the current task."
        ),
    }
},

    "required": [
        "query"
    ],
},
},

{
    "type": "function",

    "function": {
        "name":
            "read_persistent_memory",

        "description": (
            "Read the user's saved persistent "
            "memories and preferences. Use when "
            "the user refers to a saved or "
            "remembered preference."
        ),

        "parameters": {
            "type": "object",
            "properties": {},
        },
    },
},

{
    "type": "function",

    "function": {
        "name":
            "calculator",

```

```

    "description": (
      "Safely evaluate numeric arithmetic. "
      "Use for important derived numerical "
      "values. Expressions must contain "
      "only numbers, parentheses, and "
      "arithmetic operators. Do not include "
      "currency labels, units, or "
      "natural-language words."
    ),

    "parameters": {
      "type": "object",

      "properties": {
        "expression": {
          "type": "string",

          "description": (
            "Numeric arithmetic such as "
            "6000 - 1000 - 1750 or "
            "3250 / 25."
          ),
        }
      },

      "required": [
        "expression"
      ],
    },
  },
{
  "type": "function",

  "function": {
    "name":
      "current_time",

    "description": (
      "Get the current date and time, "
      "optionally in a specific IANA "
      "timezone."
    )
  }
}

```

```

    ),
    "parameters": {
      "type": "object",

      "properties": {
        "timezone": {
          "type": "string",

          "description": (
            "Optional IANA timezone "
            "such as Asia/Kolkata."
          ),
        },
      },
    },
  },
},
{
  "type": "function",

  "function": {
    "name":
      "remember_user_memory",

    "description": (
      "Persist one durable user fact or "
      "preference across application "
      "restarts. Use only when the user "
      "explicitly asks to remember, save, "
      "or store it."
    ),

    "parameters": {
      "type": "object",

      "properties": {
        "memory": {
          "type": "string",

          "description": (
            "A concise standalone "
            "fact or preference to save."

```

```

        },
      },
    },
    "required": [
      "memory"
    ],
  },
},
{
  "type": "function",

  "function": {
    "name":
      "forget_user_memory",

    "description": (
      "Delete a persistent memory when "
      "the user explicitly asks to forget "
      "or remove a saved fact or preference."
    ),

    "parameters": {
      "type": "object",

      "properties": {
        "query": {
          "type": "string",

          "description": (
            "Text identifying the "
            "memory to forget."
          ),
        }
      },
    },

    "required": [
      "query"
    ],
  },
},
},

```

```
]
```

```
AVAILABLE_TOOLS = {  
    "search_knowledge_file":  
        search_knowledge_file,  
  
    "read_persistent_memory":  
        read_persistent_memory,  
  
    "calculator":  
        calculator,  
  
    "current_time":  
        current_time,  
  
    "remember_user_memory":  
        remember_user_memory,  
  
    "forget_user_memory":  
        forget_user_memory,  
}
```

```
MCP_TOOL_SCHEMAS: list[  
    dict[str, Any]  
] = []
```

```
MCP_TOOL_NAME_MAP: dict[  
    str,  
    str,  
] = {}
```

```
MCP_DISCOVERY_ATTEMPTED = False  
MCP_DISCOVERY_ERROR = ""
```

```
def _safe_mcp_tool_name(  
    tool_name: str,  
) -> str:
```

```
    safe_name = re.sub(  
        r"^[a-zA-Z0-9_]",  
        "_",  
        tool_name,  
    )
```

```

    return (
        "mcp_cloud_lab__"
        f"{safe_name}"
    )

def initialize_mcp_tools(
) -> None:

    global MCP_DISCOVERY_ATTEMPTED
    global MCP_DISCOVERY_ERROR

    if MCP_DISCOVERY_ATTEMPTED:
        return

    MCP_DISCOVERY_ATTEMPTED = True

    try:

        discovered_tools = (
            discover_mcp_tools()
        )

        for tool in discovered_tools:

            original_name = str(
                tool["name"]
            )

            agent_name = (
                _safe_mcp_tool_name(
                    original_name
                )
            )

            MCP_TOOL_NAME_MAP[
                agent_name
            ] = original_name

            MCP_TOOL_SCHEMAS.append(
                {
                    "type":
                    "function",

```

```

"function": {
    "name":
        agent_name,

    "description": (
        "MCP tool provided by "
        "the external LearningOps "
        "Cloud Lab server. "
        + str(
            tool.get(
                "description",
                "",
            )
        )
    ),

    "parameters":
        tool.get(
            "input_schema",
            {
                "type":
                    "object",

                "properties":
                    {},
            },
        ),
    },
}

)

print(
    " [mcp] discovered "
    f"{len(MCP_TOOL_SCHEMAS)} "
    "tool(s)"
)

for name in (
    MCP_TOOL_NAME_MAP
):

    print(
        f" [mcp] {name}"
    )

```

except Exception as exc:

```
MCP_DISCOVERY_ERROR = str(
    exc
)
```

```
print(
    " [mcp] discovery failed: "
    f"{exc}"
)
```

```
def get_all_tool_schemas(
) -> list[dict[str, Any]]:
```

```
    initialize_mcp_tools()
```

```
    return (
        LOCAL_TOOL_SCHEMAS
        + MCP_TOOL_SCHEMAS
    )
```

```
# -----
# Dynamic System Prompt
# -----
```

```
def build_system_prompt() -> str:
```

```
    """
```

```
    Return the base agent prompt.
```

```
    Persistent memory is intentionally NOT
    injected into every request.
```

```
    """
```

```
    return SYSTEM_PROMPT
```

```
# -----
# Ollama Chat Call
# -----
```

```
def call_ollama(
    messages: list[
        dict[str, Any]
    ]
)
```

```

    ],
) -> dict[str, Any]:

    payload = {
        "model":
            LLM_MODEL,

        "messages":
            messages,

        "tools":
            get_all_tool_schemas(),

        "stream":
            False,

        "think":
            False,

        "keep_alive":
            "10m",

        "options": {
            "temperature": 0,
        },
    }

    response = requests.post(
        f"{OLLAMA_BASE_URL}/chat",
        json=payload,
        timeout=(
            REQUEST_TIMEOUT_SECONDS
        ),
    )

    response.raise_for_status()

    data = response.json()

    message = data.get(
        "message"
    )

    if not isinstance(

```

```

        message,
        dict,
    ):
        raise RuntimeError(
            "Ollama response did not "
            "contain a valid message object."
        )

    clean_message: dict[
        str,
        Any,
    ] = {
        "role":
            "assistant",

        "content":
            message.get(
                "content",
                "",
            ),
    }

    tool_calls = (
        message.get(
            "tool_calls"
        )
        or []
    )

    if tool_calls:
        clean_message[
            "tool_calls"
        ] = tool_calls

    return clean_message

# -----
# Tool Execution Helpers
# -----

def _parse_tool_arguments(
    arguments: Any,
) -> dict[str, Any]:

```

```
if isinstance(
    arguments,
    dict,
):
    return arguments
```

```
if isinstance(
    arguments,
    str,
):
```

```
    parsed = json.loads(
        arguments
    )
```

```
    if isinstance(
        parsed,
        dict,
    ):
        return parsed
```

```
    raise ValueError(
        "Tool arguments were not "
        "a JSON object."
    )
```

```
def _extract_tool_request(
    tool_call:
        dict[str, Any],
) -> tuple[
    str,
    dict[str, Any],
]:
```

```
    function = tool_call.get(
        "function",
        {},
    )
```

```
    tool_name = str(
        function.get(
            "name",

```

```
    """,
    )
).strip()
```

```
if not tool_name:
    raise ValueError(
        "Tool call did not include "
        "a function name."
    )
```

```
arguments = (
    _parse_tool_arguments(
        function.get(
            "arguments",
            {},
        )
    )
)
)
```

```
return (
    tool_name,
    arguments,
)
```

```
def _tool_signature(
    tool_name: str,
    arguments: dict[str, Any],
) -> str:
```

```
    return json.dumps(
        {
            "tool":
                tool_name,

            "arguments":
                arguments,
        },

        sort_keys=True,
        ensure_ascii=False,
    )
```

```

def execute_tool(
    tool_name: str,
    arguments: dict[str, Any],
) -> str:

    if (
        tool_name
        in AVAILABLE_TOOLS
    ):

        try:

            result = (
                AVAILABLE_TOOLS[
                    tool_name
                ](
                    **arguments
                )
            )

            return str(
                result
            )

        except TypeError as exc:

            return (
                "Tool error: invalid "
                f"arguments for {tool_name}: "
                f"{exc}"
            )

        except Exception as exc:

            return (
                f"Tool error: {exc}"
            )

    if (
        tool_name
        in MCP_TOOL_NAME_MAP
    ):

```

```

original_name = (
    MCP_TOOL_NAME_MAP[
        tool_name
    ]
)

try:

    return call_mcp_tool(
        original_name,
        arguments,
    )

except Exception as exc:

    return (
        "MCP tool error: "
        f"{exc}"
    )

return (
    "Tool error: unknown tool "
    f"{tool_name}."
)

# -----
# Conversation Memory
# -----

def _remember_conversation_turn(
    conversation_history:
        list[dict[str, str]],

    user_input: str,
    assistant_answer: str,
) -> None:

    conversation_history.append(
        {
            "role":
                "user",

```

```

        "content":
            user_input,
    }
)

conversation_history.append(
    {
        "role":
            "assistant",

        "content":
            assistant_answer,
    }
)

if (
    len(conversation_history)
    > MAX_CONVERSATION_MESSAGES
):

    del conversation_history[
        :-MAX_CONVERSATION_MESSAGES
    ]

# -----
# Application Guardrails
# -----

def looks_like_knowledge_question(
    user_input: str,
) -> bool:

    text = (
        user_input.lower()
    )

    knowledge_signals = [
        "mohit",
        "my age",
        "my name",
        "where do i stay",
        "where i stay",
        "my cloud",
    ]

```

```
"my job",
"my work",
"my local knowledge",
"local knowledge",
"knowledge file",
"stored notes",
"my notes",
"my local cloud lab notes",
"my cloud lab budget",
"cloud lab budget",
"lab budget",
"fixed reserve",
"fixed lab cost",
"gpu lab cost",
"default lab timezone",
]
```

```
return any(
    signal in text
    for signal
    in knowledge_signals
)
```

```
def looks_like_saved_memory_request(
    user_input: str,
) -> bool:
```

```
    text = (
        user_input.lower()
    )
```

```
    memory_signals = [
        "saved preference",
        "saved preferences",
        "remembered preference",
        "remembered gpu preference",
        "using my remembered",
        "persistent memory",
        "saved memory",
        "what you remember",
        "previously remembered",
    ]
```

```
return any(
    signal in text
    for signal
    in memory_signals
)
```

```
def current_request_needs_time(
    user_input: str,
) -> bool:
```

```
    text = (
        user_input.lower()
    )
```

```
    time_signals = [
        "current time",
        "what time",
        "time in ",
        "time at ",
        "current date",
        "date and time",
        "today's date",
        "today date",
    ]
```

```
    return any(
        signal in text
        for signal
        in time_signals
    )
```

```
def is_memory_save_only_request(
    user_input: str,
) -> bool:
```

```
    text = (
        user_input
        .lower()
        .strip()
    )
```

```
    save_signals = [
```

```
"remember that",
"remember my",
"save that",
"save this",
"store that",
"store this",
]
```

```
other_task_signals = [
    "calculate",
    "how much",
    "how many",
    "current time",
    "what time",
    "using my",
    "then ",
    "also ",
]
```

```
return (
    any(
        signal in text
        for signal
        in save_signals
    )
    and not any(
        signal in text
        for signal
        in other_task_signals
    )
)
```

```
def is_memory_forget_only_request(
    user_input: str,
) -> bool:
```

```
    text = (
        user_input
        .lower()
        .strip()
    )
```

```
    forget_signals = [
```

```
"forget that",
"forget my",
"delete that memory",
"remove that memory",
"delete my saved",
"remove my saved",
]
```

```
other_task_signals = [
    "calculate",
    "how much",
    "how many",
    "current time",
    "what time",
    "using my",
    "then ",
    "also ",
]
```

```
return (
    any(
        signal in text
        for signal
        in forget_signals
    )
    and not any(
        signal in text
        for signal
        in other_task_signals
    )
)
```

```
def _build_persistent_memory_guardrail_context(
    persistent_memory: str,
) -> str:
```

```
return (
    "APPLICATION PERSISTENT MEMORY CONTEXT "
    "(user data for the CURRENT task):\n"

    "<current_task_memory>\n"

    f"{persistent_memory}\n"
```

```
"</current_task_memory>\n\n"
```

```
"The current request explicitly refers "  
"to saved memory. Apply relevant saved "  
"preferences exactly as written and "  
"ignore unrelated memories."
```

```
)
```

```
def _inject_forced_knowledge_result(  
    messages:  
        list[dict[str, Any]],
```

```
    user_input: str,  
    ) -> str:
```

```
    tool_result = (  
        search_knowledge_file(  
            user_input  
        )  
    )
```

```
    messages.append(  
        {  
            "role":  
                "user",  
  
            "content": (  
                "APPLICATION GUARDRAIL RESULT:\n"  
                "The application performed the "  
                "required local knowledge lookup "  
                "because the model did not emit "  
                "the required structured tool call."  
                "\n\n"
```

```
                f"Knowledge result:\n"  
                f"{tool_result}\n\n"
```

```
                "Continue solving the ORIGINAL "  
                "user request. You are still inside "  
                "the agent workflow. Call any "  
                "additional calculator, time, memory, "  
                "or MCP tool that is still required."
```

```
    ),  
    }  
)
```

```
return tool_result
```

```
# -----  
# LangGraph State  
# -----
```

```
class AgentState(TypedDict):  
    messages: list[dict[str, Any]]  
    user_input: str  
    agent_rounds: int  
    knowledge_lookup_done: bool  
    identical_call_counts: dict[str, int]  
    final_answer: str  
    finish_after_tool: bool
```

```
# -----  
# LangGraph Nodes  
# -----
```

```
def agent_node(  
    state: AgentState,  
) -> dict[str, Any]:
```

```
    round_number = (  
        state["agent_rounds"]  
        + 1  
    )
```

```
    if (  
        round_number  
        > MAX_TOOL_ROUNDS  
    ):
```

```
        raise RuntimeError(  
            "Agent exceeded the maximum "  
            f"of {MAX_TOOL_ROUNDS} "  
            "model/tool rounds."  
        )
```

```
print(
    "\n [graph] entering agent node"
)
```

```
assistant_message = (
    call_ollama(
        state["messages"]
    )
)
```

```
messages = [
    dict(message)
    for message
    in state["messages"]
]
```

```
messages.append(
    assistant_message
)
```

```
tool_calls = (
    assistant_message.get(
        "tool_calls"
    )
    or []
)
```

```
if tool_calls:
```

```
    print(
        f" [round "
        f"{round_number}] "
        f"model requested "
        f"{len(tool_calls)} "
        "tool call(s)"
    )
```

```
    return {
        "messages":
            messages,

        "agent_rounds":
            round_number,
```

```

        "final_answer":
            "",

        "finish_after_tool":
            False,
    }

    answer = str(
        assistant_message.get(
            "content",
            "",
        )
    ).strip()

    if not answer:

        answer = (
            "I could not produce "
            "a final response."
        )

    return {
        "messages":
            messages,

        "agent_rounds":
            round_number,

        "final_answer":
            answer,

        "finish_after_tool":
            False,
    }

def tool_node(
    state: AgentState,
) -> dict[str, Any]:

    print(
        " [graph] entering tool node"
    )

```

```

messages = [
    dict(message)
    for message
    in state["messages"]
]

identical_call_counts = dict(
    state[
        "identical_call_counts"
    ]
)

knowledge_lookup_done = (
    state[
        "knowledge_lookup_done"
    ]
)

final_answer = ""

finish_after_tool = False

last_message = (
    messages[-1]
    if messages
    else {}
)

tool_calls = (
    last_message.get(
        "tool_calls"
    )
    or []
)

for tool_call in tool_calls:

    try:

        (
            tool_name,
            arguments,
        ) = (
            _extract_tool_request(

```

```
        tool_call
    )
)
```

except Exception as exc:

```
    tool_name = (
        "invalid_tool_call"
    )
```

```
    arguments = {}
```

```
    tool_result = (
        f"Tool error: "
        f"{exc}"
    )
```

```
    print(
        f" [tool] "
        f"{tool_name}"
    )
```

```
    print(
        f" [result] "
        f"{tool_result}"
    )
```

```
    messages.append(
        {
            "role":
                "tool",

            "tool_name":
                tool_name,

            "content":
                tool_result,
        }
    )
```

```
    continue
```

```
signature = (
    _tool_signature(
```

```

        tool_name,
        arguments,
    )
)

identical_call_counts[
    signature
] = (
    identical_call_counts.get(
        signature,
        0,
    )
    + 1
)

if (
    identical_call_counts[
        signature
    ]
    > MAX_IDENTICAL_TOOL_CALLS
):

    tool_result = (
        "Tool guardrail error: "
        "this identical tool call "
        "has already been executed "
        "too many times. Use the "
        "existing result, change the "
        "arguments, choose another "
        "tool, or finish the answer."
    )

elif (
    tool_name
    == "current_time"

    and not
    current_request_needs_time(
        state[
            "user_input"
        ]
    )
):

```

```

    tool_result = (
        "Tool guardrail error: "
        "current_time was not requested "
        "by the current user message. "
        "Do not use time data from an "
        "unrelated earlier task."
    )

elif (
    tool_name
    == "read_persistent_memory"

    and not
    looks_like_saved_memory_request(
        state[
            "user_input"
        ]
    )
):

    tool_result = (
        "Tool guardrail error: "
        "persistent memory was not "
        "requested or referenced by "
        "the current user message."
    )

else:

    tool_result = (
        execute_tool(
            tool_name,
            arguments,
        )
    )

print(
    f" [tool] "
    f"{tool_name}"
)

print(
    " [args] "
    + json.dumps(

```

```

        arguments,
        ensure_ascii=False,
    )
)

print(
    f" [result] "
    f"{tool_result}"
)

messages.append(
    {
        "role":
            "tool",

        "tool_name":
            tool_name,

        "content":
            tool_result,
    }
)

if (
    tool_name
    == "search_knowledge_file"
):

    knowledge_lookup_done = True

if (
    tool_name
    == "remember_user_memory"

    and
    is_memory_save_only_request(
        state[
            "user_input"
        ]
    )
):

    final_answer = (
        tool_result

```

```

    )

    finish_after_tool = True

    break

    if (
        tool_name
        == "forget_user_memory"

        and
        is_memory_forget_only_request(
            state[
                "user_input"
            ]
        )
    ):

        final_answer = (
            tool_result
        )

        finish_after_tool = True

        break

    if tool_name in {
        "remember_user_memory",
        "forget_user_memory",
    }:

        messages[0][
            "content"
        ] = (
            build_system_prompt()
        )

    return {
        "messages":
            messages,

        "identical_call_counts":
            identical_call_counts,

```

```

    "knowledge_lookup_done":
        knowledge_lookup_done,

    "final_answer":
        final_answer,

    "finish_after_tool":
        finish_after_tool,
}

```

```

def forced_knowledge_node(
    state: AgentState,
) -> dict[str, Any]:

```

```

    print(
        " [graph] entering "
        "forced_knowledge node"
    )

```

```

    messages = [
        dict(message)
        for message
        in state["messages"]
    ]

```

```

    tool_result = (
        _inject_forced_knowledge_result(
            messages,
            state[
                "user_input"
            ],
        )
    )

```

```

    print(
        " [tool] "
        "search_knowledge_file"
    )

```

```

    print(
        " [args] "
        + json.dumps(
            {

```

```

        "query":
            state[
                "user_input"
            ]
    },

    ensure_ascii=False,
)
)

print(
    f" [result] "
    f"{tool_result}"
)

return {
    "messages":
        messages,

    "knowledge_lookup_done":
        True,

    "final_answer":
        "",

    "finish_after_tool":
        False,
}

# -----
# LangGraph Routing
# -----

def route_after_agent(
    state: AgentState,
) -> str:

    last_message = (
        state[
            "messages"
        ][-1]

        if state[

```

```

        "messages"
    ]

    else {}
)

tool_calls = (
    last_message.get(
        "tool_calls"
    )
    or []
)

if tool_calls:

    print(
        " [graph] route: "
        "agent -> tools"
    )

    return "tools"

if (
    looks_like_knowledge_question(
        state[
            "user_input"
        ]
    )
    and not state[
        "knowledge_lookup_done"
    ]
):

    print(
        f" [round "
        f"{state['agent_rounds']}] "
        "model attempted to finish "
        "before required knowledge lookup"
    )

    print(
        " [graph] route: "
        "agent -> forced_knowledge"
    )

```

```

    )

    return (
        "forced_knowledge"
    )

    print(
        f" [round "
        f"{state['agent_rounds']}] "
        "model finished"
    )

    print(
        " [graph] route: "
        "agent -> END"
    )

    return END

def route_after_tools(
    state: AgentState,
) -> str:

    if state[
        "finish_after_tool"
    ]:

        print(
            " [graph] route: "
            "tools -> END"
        )

        return END

    print(
        " [graph] route: "
        "tools -> agent"
    )

    return "agent"

# -----

```

```
# Build LangGraph
```

```
# -----
```

```
def build_agent_graph():
```

```
    builder = StateGraph(  
        AgentState  
    )
```

```
    builder.add_node(  
        "agent",  
        agent_node,  
    )
```

```
    builder.add_node(  
        "tools",  
        tool_node,  
    )
```

```
    builder.add_node(  
        "forced_knowledge",  
        forced_knowledge_node,  
    )
```

```
    builder.add_edge(  
        START,  
        "agent",  
    )
```

```
    builder.add_conditional_edges(  
        "agent",  
        route_after_agent,  
        {  
            "tools":  
                "tools",  
  
            "forced_knowledge":  
                "forced_knowledge",  
  
            END:  
                END,  
        },  
    )
```

```

builder.add_conditional_edges(
    "tools",
    route_after_tools,
    {
        "agent":
            "agent",

        END:
            END,
    },
)

```

```

builder.add_edge(
    "forced_knowledge",
    "agent",
)

```

```

return builder.compile()

```

```

AGENT_GRAPH = (
    build_agent_graph()
)

```

```

# -----
# Run One Agent Request
# -----

```

```

def run_agent(
    user_input: str,

    conversation_history:
        list[dict[str, str]],
) -> str:

    system_content = (
        build_system_prompt()
    )

    if looks_like_saved_memory_request(
        user_input
    ):

```

```

persistent_memory_result = (
    read_persistent_memory()
)

print(
    " [guardrail] "
    "saved preference referenced; "
    "loading persistent memory"
)

print(
    " [tool] "
    "read_persistent_memory"
)

print(
    f" [result] "
    f"{persistent_memory_result}"
)

system_content = (
    f"{system_content}\n\n"

    + _build_persistent_memory_guardrail_context(
        persistent_memory_result
    )
)

messages: list[
    dict[str, Any]
] = [
    {
        "role":
            "system",

        "content":
            system_content,
    }
]

messages.extend(
    conversation_history
)

```

```

messages.append(
    {
        "role":
            "user",

        "content":
            user_input,
    }
)

initial_state: AgentState = {
    "messages":
        messages,

    "user_input":
        user_input,

    "agent_rounds":
        0,

    "knowledge_lookup_done":
        False,

    "identical_call_counts":
        {},

    "final_answer":
        "",

    "finish_after_tool":
        False,
}

final_state = (
    AGENT_GRAPH.invoke(
        initial_state,

        config={
            "recursion_limit":
                GRAPH_RECURSION_LIMIT,
        },
    )
)

```

```
answer = str(
    final_state.get(
        "final_answer",
        "",
    )
).strip()
```

```
if not answer:
```

```
    answer = (
        "I could not produce "
        "a final response."
    )
```

```
_remember_conversation_turn(
    conversation_history,
    user_input,
    answer,
)
```

```
return answer
```

```
# -----
# CLI Helpers
# -----
```

```
def print_history(
    conversation_history:
        list[dict[str, str]],
) -> None:
```

```
    if not conversation_history:
```

```
        print(
            "No short-term conversation "
            "history yet."
        )
```

```
    return
```

```
    print(
        "\nShort-term conversation history:"
    )
```

```

for message in (
    conversation_history
):

    role = (
        message[
            "role"
        ].capitalize()
    )

    print(
        f"{role}: "
        f"{message['content']}"
    )

```

```

def print_mcp_status() -> None:

```

```

    initialize_mcp_tools()

    print(
        "\nMCP status:"
    )

    if MCP_TOOL_NAME_MAP:

        print(
            " External server: "
            "LearningOps Cloud Lab"
        )

        print(
            " Discovered tools:"
        )

        for (
            agent_name,
            server_name,
        ) in MCP_TOOL_NAME_MAP.items():

            print(
                f" - {agent_name} "
                f"-> {server_name}"
            )

```

```

    )

    return

print(
    " No MCP tools discovered."
)

if MCP_DISCOVERY_ERROR:

    print(
        " Discovery error: "
        f"{MCP_DISCOVERY_ERROR}"
    )

def print_help() -> None:

    print(
        "\nCommands:"
    )

    print(
        " /history      "
        "Show short-term conversation history"
    )

    print(
        " /new           "
        "Clear short-term conversation history only"
    )

    print(
        " /memory        "
        "Show persistent user memory"
    )

    print(
        " /memory-clear  "
        "Delete all persistent user memory"
    )

    print(
        " /mcp           "

```

```

        "Show MCP connection/discovered tools"
    )

    print(
        " /help      "
        "Show these commands"
    )

    print(
        " exit | quit  "
        "Stop the application"
    )

# -----
# Main
# -----

def main() -> None:

    print(
        "=" * 72
    )

    print(
        "Local Multi-Tool AI Agent — "
        "Part 6: External Systems with MCP"
    )

    print(
        "=" * 72
    )

    print(
        f"Knowledge file: "
        f"{DATA_FILE}"
    )

    print(
        f"LLM model: "
        f"{LLM_MODEL}"
    )

    print(

```

```

    "Maximum agent rounds: "
    f"{MAX_TOOL_ROUNDS}"
)

print(
    "LangGraph recursion limit: "
    f"{GRAPH_RECURSION_LIMIT}"
)

print(
    "Short-term memory: "
    "bounded in-process "
    "conversation history"
)

print(
    "Persistent memory: "
    "storage/user_memory.json"
)

print(
    "Graph/tool trace: enabled"
)

print(
    "MCP integration: "
    "LearningOps Cloud Lab over stdio"
)

print_help()

check_ollama()

initialize_mcp_tools()

conversation_history: list[
    dict[str, str]
] = []

while True:

    try:

        user_input = input(

```

```
        "\nYou: "
    ).strip()

    if not user_input:
        continue

    if (
        user_input.lower()
        in {
            "exit",
            "quit",
        }
    ):

        print(
            "Goodbye."
        )

        break

    if (
        user_input
        == "/help"
    ):

        print_help()

        continue

    if (
        user_input
        == "/history"
    ):

        print_history(
            conversation_history
        )

        continue

    if (
        user_input
        == "/new"
    ):
```

```
conversation_history.clear()

print(
    "Short-term conversation "
    "history cleared."
)

continue

if (
    user_input
    == "/memory"
):

    print(
        "\nPersistent "
        "user memory:"
    )

    print(
        format_persistent_memories()
    )

    continue

if (
    user_input
    == "/memory-clear"
):

    clear_persistent_memories()

    print(
        "Persistent user "
        "memory cleared."
    )

    continue

if (
    user_input
    == "/mcp"
):
```

```

        print_mcp_status()

        continue

    answer = (
        run_agent(
            user_input,
            conversation_history,
        )
    )

    print(
        f"\nAgent: "
        f"{answer}"
    )

except KeyboardInterrupt:

    print(
        "\nGoodbye."
    )

    break

except requests.RequestException as exc:

    print(
        f"\nOllama request error: "
        f"{exc}",
        file=sys.stderr,
    )

except Exception as exc:

    print(
        f"\nError: {exc}",
        file=sys.stderr,
    )

if __name__ == "__main__":
    main()

```

