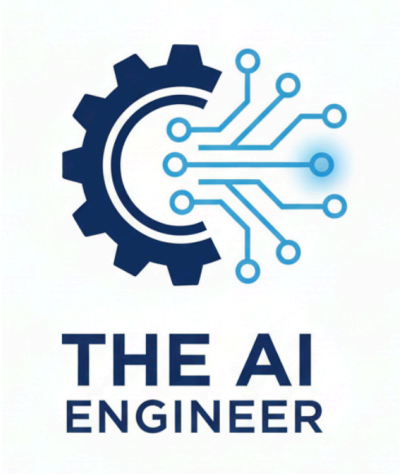
Model Context Protocol (MCP) Agents: Systems Note & Capstone Guide

TAE Program — Core Track

November 23, 2025

Abstract

This Week 4 note makes the Model Context Protocol (MCP) tangible. We connect the Observe → Plan → Act loop from [1] with the systems view of Part III in [2], define what MCP servers and agents must provide, and spell out the capstone deliverables. The featured project—an "Incident Command Agent" that triages operational runbooks—lets you exercise tool contracts, memory, telemetry, and deployment discipline. Colab remains optional: the core workflow assumes a local Python environment, with pointers on how to mirror it in notebooks if needed.



Contents

1 Motivation and Goal

Week 4 shifts from "model reasoning" to "agentic systems". The [1] guide outlines an Observe → Plan → Act → Learn loop, while Part III of [2] stresses retrieval, orchestration, and scaling. The Model Context Protocol ties these pieces together and the points below summarize why MCP matters for builders:

- **Standard contracts** for discovery: clients enumerate tools/resources without bespoke SDK glue.
- **Structured messages** so agents can log, replay, and retry safely—critical for the Learning stage highlighted in [1].
- **Deterministic execution envelope**: MCP keeps tool payloads typed, bounded, and auditable, echoing the guardrail guidance emphasized in [2].

Figure 1, adapted from the Week 4 slides, reinforces why we obsess over budgets before adding autonomy: keeping each phase in a tight latency window forces observability hooks long before exotic tooling.

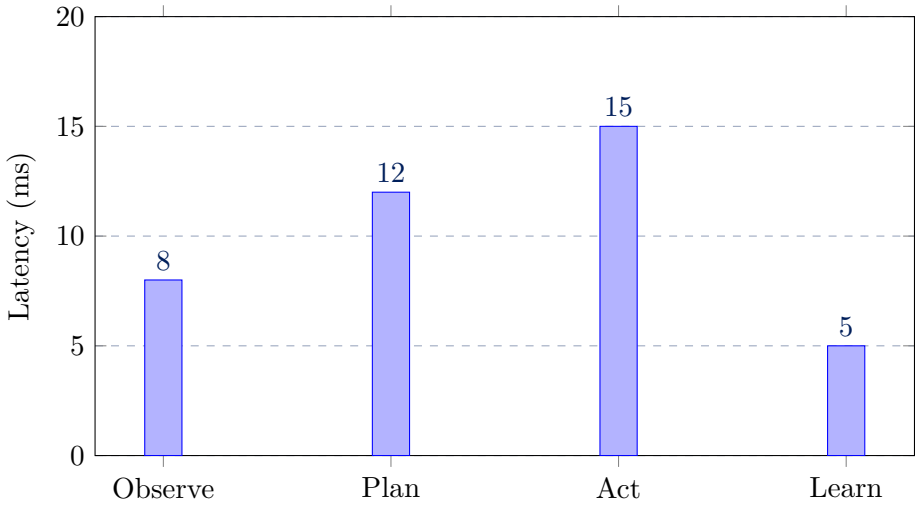


Figure 1: Keeping each phase within its latency budget forces instrumentation and logging before adding more tools or autonomy. Here, total budget is fixed to 40 ms.

The capstone you will build must prove that you can read/write MCP traffic, orchestrate multiple tools, and expose observability hooks that make the broader engineering story credible.

2 MCP Fundamentals

This section unpacks the mechanics of MCP so you can reason about the protocol before writing code. Read each subsection sequentially: discovery and RPC wiring come first, then tool design, and finally telemetry. Figure 2 from the slides keeps the component wiring front and center so you always know which interface (policy, tools, memory, environment) a protocol feature serves.

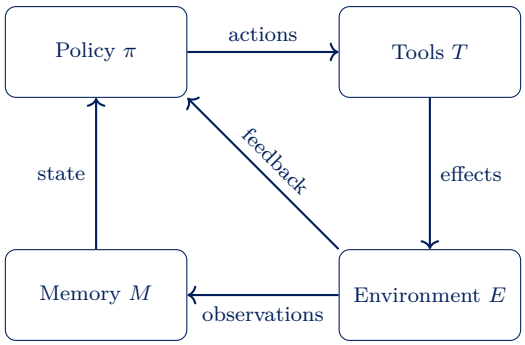


Figure 2: Every component connection stays explicit so you can trace where information is created, transformed, or persisted.

2.1 Protocol Overview

At its core, MCP is a JSON-RPC 2.0 conversation between a *client* (your orchestrator) and a *server* (your tool stack). The protocol stays transport-agnostic (websocket, stdio, pipes), but every exchange follows a predictable cadence:

1. **Capabilities handshake**. The client issues an `initialize` call that includes its name/version. The server responds with `capabilities`: a tool list (name, JSON schema, rate hints) plus resource catalogs. Until this succeeds you cannot plan safely.

2. **Resource access**. Clients call `getResource` (or similar) with a URI, pagination cursor, and optional filters. Servers respond with structured chunks—think of them as typed "Observe" payloads [1].

3. **Tool calls**. Clients invoke `callTool` with a tool name and arguments validated against the advertised schema. Servers return `result` objects that include status, payload, and metrics (latency, cost, token counts) to support the guardrails in [2].

The snippet below shows each phase end-to-end; adapt it inside your orchestrator to keep the control flow explicit.

MCP Handshake, Resource Read, Tool Call

```
import asyncio  # event loop support
import json  # JSON serialization helper
import uuid  # unique correlation IDs
import websockets  # WebSocket client transport

ENDPOINT = "ws://localhost:8765/mcp"  # local MCP server address

async def run_loop():  # orchestrate handshake, resource read, and tool call
    async with websockets.connect(ENDPOINT) as ws:  # open MCP session
        init_id = str(uuid.uuid4())  # unique initialize request id
        init_payload = {
            "jsonrpc": "2.0",
            "id": init_id,
            "method": "initialize",
            "params": {"clientName": "incident-cli", "clientVersion": "0.1.0"},
        }  # capabilities handshake payload
        await ws.send(json.dumps(init_payload))  # transmit handshake request
        caps = await ws.recv()  # capture server capabilities

        res_id = str(uuid.uuid4())  # resource fetch request id
```

```
        resource_payload = {
            "jsonrpc": "2.0",
            "id": res_id,
            "method": "getResource",
            "params": {"uri": "memory://alerts/latest", "cursor": None},
        }  # request latest alert snapshot
        await ws.send(json.dumps(resource_payload))  # send resource request
        alert_snapshot = await ws.recv()  # receive snapshot payload

        tool_id = str(uuid.uuid4())  # tool invocation request id
        tool_payload = {
            "jsonrpc": "2.0",
            "id": tool_id,
            "method": "callTool",
            "params": {
                "name": "run_diagnostic",
                "arguments": {
                    "command": "kubectl get pods",
                    "host": "staging-api",
                },
            },
        }  # run diagnostic tool over staging-api
        await ws.send(json.dumps(tool_payload))  # invoke tool through MCP
        diagnostic = await ws.recv()  # grab tool response

        print(caps, alert_snapshot, diagnostic)  # quick sanity output

if __name__ == "__main__":  # allow direct script execution
    asyncio.run(run_loop())  # kick off async workflow
```

Note how every request carries a unique `id` so replies can be correlated—log these IDs for replay testing and incident auditing. The client expects an MCP server to be reachable at `ws://127.0.0.1:8765/mcp`; you can run the reference server from Section 6 in a second terminal to satisfy this dependency before experimenting with your own stack.

2.2 Tool Contracts and Memory

Every tool you register should answer four questions:

1. **Purpose**: why the agent should call it instead of prompting.
2. **Inputs**: JSON schema with defaults, ranges, and guardrail hints (timeouts, max tokens).
3. **Outputs**: typed payload plus status (success/error) and metrics (latency, cost).
4. **Side effects**: what gets persisted, how to roll back, and what to write into working memory.

Combine this with the Learn/Loop guidance in [1]: after each tool call, summarize outcomes into short "deltas" and append them to memory so the next planning step has fresh context.

2.3 Telemetry and Safety

Part III of [2] emphasizes evaluation, guardrails, and scaling. MCP helps by making every action structured:

- Include **correlation IDs** on each request/response pair to join logs.

- Emit **budget counters** (tokens, milliseconds, dollars) so your orchestrator can halt before breaching quotas.
- Store **replay traces**: serialized MCP transcripts allow deterministic regression tests and red-teaming.

3 Capstone Narrative: Incident Command Agent

To make the protocol concrete, this section frames a realistic Incident Command scenario. Use it to keep the project user-focused: each subsection explains the story, stakeholders, and definition of done before you start diagramming.

3.1 Why This Topic?

Operations teams need fast incident triage. The featured agent acts as an "Incident Command" assistant:

- **Scenario**: your platform emits alerts (CPU spikes, failed deployments). The agent must gather context, consult runbooks, execute diagnostics, and draft handoff notes.
- **Tools**: retrieval over incident runbooks, a sandboxed shell diagnostic runner, a summarizer that references recent tickets, and a memory store for resolved actions.
- **Stakeholders**: on-call engineers who demand reliability, auditability, and concise recommendations.

This keeps the project practical while touching every MCP concept: multi-tool orchestration, memory, and telemetry.

3.2 Success Criteria

To "pass" the capstone, delegates should deliver the following artifacts and guarantees:

1. A runnable MCP server exposing at least three tools plus one resource surface.
2. An agent (client) that cycles through Observe $\rightarrow$ Plan $\rightarrow$ Act $\rightarrow$ Learn with enforced budgets.
3. Structured logging + replay artifacts proving instrumentation.
4. Documentation of optional Colab execution (how to tunnel MCP traffic, mount artifacts, etc.) without making it a hard dependency.

4 Detailed Objectives

After framing the why, we list the concrete objectives and tangibles you must hit. Treat the next subsections as a checklist for both theory and deliverables.

4.1 Theory + Practice Checklist

Work through the following sequence each time you pick up the project so conceptual understanding and implementation stay in lockstep:

1. **Observe**: fetch alert payloads, most recent telemetry, and runbook snippets via MCP resource listings.
2. **Plan**: select which tool to call (retrieval vs. diagnostics vs. summary) using either an LLM policy or heuristics; ensure guardrails per the orchestration chapter in [2].

3. **Act**: invoke tools with typed inputs; capture latency, stdout/stderr, and sanitized outputs.
4. **Learn**: write memory deltas ("Diagnosed service X, ran command Y, outcome Z") and tag them with timestamps for future retrieval.
5. **Loop**: continue until the agent either resolves the incident or escalates with a human-handoff package.

4.2 Deliverables (Recommended Format)

Document the evidence of completion in the following format so reviewers can grade quickly:

- **Code**: MCP server (Python package or module) + orchestrator CLI.
- **Config**: YAML/JSON describing tool registry, budgets, and environment toggles.
- **Docs**: README or notebook describing setup, optional Colab path, and verification commands.
- **Logs**: sample trace demonstrating at least two full loops.

5 Implementation Reference (Step-by-Step)

This section is intentionally hint-like to encourage independent builds while providing a repeatable path.

5.1 Step 0 — Workspace Setup

Lay a clean foundation before touching protocol code by tackling these preparation tasks:

1. Create a Python 3.10+ virtual environment (local preferred). Install `uvicorn`, `fastapi` (or similar) if you want HTTP wrappers, plus any LLM SDKs needed.
2. Define environment variables for secrets and tool limits; never bake tokens into MCP payloads.
3. For Colab: author a lightweight notebook that starts the MCP server inside the session and exposes a public tunnel (e.g., `cloudflared`) so the local agent can connect. Keep this optional.

5.2 Step 1 — Model the Tools

Define the contract surface first so the rest of the stack can compile against stable schemas:

1. Draft JSON schemas for each tool. Example categories:
   - `retrieve_runbook(query: str, top_k: int)`
   - `run_diagnostic(command: str, host: str)`
   - `summarize_incident(alert_id: str, evidence: list[str])`
2. Annotate each schema with cost/time expectations; use this metadata when budgeting inside the planner.

5.3 Step 2 — Build the MCP Server

With schemas locked, implement the server-side plumbing through the following steps:

1. Implement the capability endpoint: respond with the tool list, schemas, and resource catalog (recent alerts, telemetry snapshots, memory entries).
2. Implement tool handlers. Each handler should:
   - Validate input against the schema.
   - Execute the requested action (e.g., call a retriever, run a sandboxed subprocess).
   - Return structured output with `status`, `data`, and `metrics`.
3. Add logging around every request/response pair so you can replay sessions.

5.4 Step 3 — Memory + Resource Surfaces

Persistent state keeps loops coherent; follow these guidelines to expose it responsibly:

1. Store observations and tool outcomes as short JSON documents. Index them by alert ID + timestamp.
2. Expose the store through MCP's resource listing so the client can "Observe" before planning.
3. Keep summaries concise (3–4 sentences) to stay inside token budgets.

Figure 3 from the slides shows the exact read/write choreography we want: observations log into memory immediately, plans read curated slices, and actions feed fresh deltas back for the next loop.

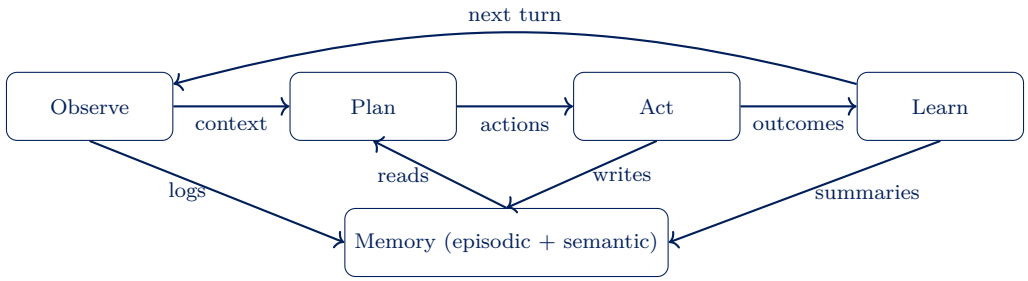


Figure 3: Explicit read/write paths keep episodic and semantic stores trustworthy and audit-friendly.

5.5 Step 4 — Orchestrator

Your orchestrator enforces policy and budgets, so build it deliberately via the steps below:

1. Initialize the MCP client: open a session, fetch capabilities, and cache schemas.
2. Implement the planning policy. Hints:
   - Start with a deterministic finite-state machine (FSM) to enforce guardrails (max depth, allowed tool sequences).
   - Optionally embed an LLM call that scores tool options based on the current observation + memory.
3. Track budgets (tokens, runtime, dollars) per loop. Abort if any limit is exceeded.

5.6 Step 5 — Telemetry and Replay

Instrument everything; the checklist below ensures you can debug and audit after the fact:

1. Assign a correlation ID at the beginning of each loop.
2. Log: observation snapshot, chosen plan, tool request, tool response, memory delta.
3. Provide a CLI flag (e.g., `--replay path.json`) that replays a stored transcript without calling real tools.

5.7 Step 6 — Verification

Finally, verify the system end-to-end with the following tests:

1. Unit test each tool handler with golden inputs/outputs.
2. Write an integration script that simulates an incident, runs the full agent loop, and checks for:
   - At least one retrieval and one diagnostic call.
   - A final summary referencing citations (from resources) plus recommended actions.
3. Capture latency and cost stats—align with the "Scaling" tips in [2].

Minimal Tool Invocation Skeleton

```python
from mcp import Client  # lightweight MCP client helper

client = Client("ws://localhost:8765")  # connect to local MCP endpoint
caps = client.get_capabilities()  # cache advertised schemas

diagnostic = client.call_tool(  # trigger diagnostic run
    name="run_diagnostic",  # tool identifier
    arguments={
        "command": "kubectl get pods",  # diagnostic command
        "host": "staging-api",  # target host
    },  # typed args
)  # end tool invocation

memory_delta = {  # structured memory entry
    "alert_id": "ALRT-2025-07",  # alert being handled
    "action": "run_diagnostic",  # tool that produced the facts
    "command": diagnostic["input"],  # echoed command metadata
    "result": diagnostic["data"],  # sanitized output payload
    "latency_ms": diagnostic["metrics"]["latency_ms"],  # telemetry for budgets
}  # end delta payload
client.append_memory(memory_delta)  # persist the delta for future loops
```

6 Complete Reference Example

Use this self-contained server/client pair before starting your capstone so you can witness the full MCP handshake locally.

1. Save the server code below as `mcp_demo_server.py` and install `websockets` via `pip install websockets`.
2. Run `python mcp_demo_server.py`; it serves MCP over `ws://127.0.0.1:8765/mcp`.
3. In a second terminal, execute the earlier client snippet (or your orchestrator) to watch the handshake, resource fetch, and tool call complete.

Minimal MCP server for local testing

```
import asyncio  # async event loop primitives
import json  # encode/decode JSON-RPC messages
import time  # fake latency measurements
import websockets  # WebSocket server utilities

HOST = "127.0.0.1"  # bind to localhost only
PORT = 8765  # default MCP demo port

RUNBOOK_SNIPPETS = [  # canned runbook guidance
    "Restart the service if CPU > 90% for 5 minutes.",  # first tip
    "If pods crashloop, capture logs before redeploying.",  # second tip
]  # end runbook list

ALERT_PAYLOAD = {  # representative alert object
    "id": "ALRT-2025-07",
    "service": "staging-api",
    "symptom": "CPU spike on node-3",
    "severity": "high",
}  # end alert object

CAPABILITIES = {  # capability advertisement
    "tools": [{
        "name": "run_diagnostic",
        "description": "Return canned diagnostics for a host.",
        "schema": {
            "type": "object",
            "properties": {
                "command": {"type": "string"},
                "host": {"type": "string"},
            },
            "required": ["command", "host"],
        },
    }],
    "resources": [{
        "uri": "memory://alerts/latest",
        "description": "Latest alert snapshot",
    }],
}  # end capability payload

async def handle_session(ws):  # process each client connection
    async for raw in ws:  # stream incoming JSON-RPC frames
        req = json.loads(raw)  # parse request payload
        method = req.get("method")  # requested RPC method
        req_id = req.get("id")  # correlate replies

        if method == "initialize":  # capability handshake
            result = {"capabilities": CAPABILITIES}  # send tool/resource list
        elif method == "getResource":  # resource fetch
            result = {
                "uri": "memory://alerts/latest",
                "data": {
                    "alert": ALERT_PAYLOAD,
                    "recommendations": RUNBOOK_SNIPPETS,
                },
            }  # latest alert snapshot
        elif method == "callTool":  # tool invocation
```

```
            args = req.get("params", {}).get("arguments", {})  # extract args
            latency = round(time.perf_counter() % 0.02, 4)  # fake latency
            result = {
                "status": "ok",
                "data": {
                    "command": args.get("command"),
                    "host": args.get("host"),
                    "stdout": "All pods healthy",
                },
                "metrics": {"latency_ms": latency * 1000},
            }  # canned tool response
        else:
            error = {
                "code": -32601,
                "message": f"Unknown method: {method}",
            }  # signal unsupported method
            await ws.send(json.dumps({
                "jsonrpc": "2.0",
                "id": req_id,
                "error": error,
            }))  # return JSON-RPC error
            continue  # skip to next frame

        payload = {"jsonrpc": "2.0", "id": req_id, "result": result}  # success
        ↪ msg
        await ws.send(json.dumps(payload))  # send response frame

async def main():  # run the websocket server forever
    async with websockets.serve(  # expose MCP endpoint
        handle_session,
        HOST,
        PORT,
        subprotocols=["mcp"],
    ):
        print(
            f"MCP demo server ready on ws://{HOST}:{PORT}/mcp"
        )  # status log
        await asyncio.Future()  # keep server alive indefinitely

if __name__ == "__main__":  # allow CLI execution
    asyncio.run(main())  # start server loop
```

7 Optional Colab Path

Colab remains strictly optional, but if you document it provide the following guidance to delegates:

- Colab notebooks can host an MCP server (Python runtime + `uvicorn`) if you forward ports via `cloudflared` or `ngrok`. Document the steps but keep local execution canonical.
- Store artifacts (logs, memory) in mounted Google Drive folders. Ensure the orchestrator can read the same paths locally to stay portable.
- Highlight limitations: restricted long-running processes, lack of system packages, and the need for persistent secrets management.

8 Stretch Goals

Stretch goals help advanced delegates push further; suggest the following extensions:

- **Adaptive planners**: swap the FSM for a tree-search planner that scores tool sequences.
- **Multi-tenant servers**: serve multiple agents with per-tenant quotas and isolated memories.
- **Evaluation harness**: reuse the evaluation tips from [2] to run scripted assertions per incident type.
- **Production hardening**: package the server as a container, add CI checks that replay transcripts, and deploy to your chosen environment.

9 Recap

MCP operationalizes everything you learned in Weeks 1–3:

- **Math + optimization** (Week 1): still relevant for scoring plans and budgets.
- **Transformers + attention** (Week 3): still the reasoning core, now embedded inside a disciplined agentic loop.
- **Systems thinking** (Week 4): MCP gives you the scaffolding to ship trustworthy agents.

Use this note as your blueprint: define tools carefully, orchestrate responsibly, log everything, and ship an Incident Command Agent that your future self would trust when the pager goes off.

References

[1] Y. J. Hilpisch, *AI Agents & Automation — Engineering Intelligent Workflows*, The AI Engineer Program, 2025. Available at `https://theaiengineer.dev/tae/assets/age.html`.

[2] Y. J. Hilpisch, *Software, ML, and AI Engineering*, The AI Engineer Program, 2025. Available at `https://theaiengineer.dev/tae/assets/eng.html`.