Reference Implementation Skeleton with MCP

Agentic AI - Using Model Context Protocol (MCP)

Curated Teaching PDF with Code & Comparison

1. Introduction

This teaching PDF presents a regenerated implementation skeleton of Agentic AI using the Model Context Protocol (MCP). MCP formalizes tool definitions, enforces schemas for inputs and outputs, and provides better interoperability and observability. We will compare this against the earlier framework-free design.

2. MCP Reference Implementation Code

```
from mcp.server import Server
from mcp.types import Tool, Schema, ToolCall

# ---------- Define Tools ----------
docsearch_tool = Tool(
    name="DocSearch",
    description="Retrieve text chunks with cite_labels",
    input_schema=Schema.object({
        "query": Schema.string(),
        "k": Schema.integer(default=3),
        "section": Schema.string()
    }),
    output_schema=Schema.array(Schema.object({
        "doc_id": Schema.string(),
        "title": Schema.string(),
        "year": Schema.integer(),
        "page": Schema.integer(),
        "text": Schema.string(),
        "cite_label": Schema.string()
    }))
)

saveoutput_tool = Tool(
    name="SaveOutput",
    description="Persist brief and sources",
    input_schema=Schema.object({
        "brief_markdown": Schema.string(),
        "sources": Schema.array(Schema.object({
            "doc_id": Schema.string(),
            "pages_used": Schema.array(Schema.integer())
        }))
    }),
    output_schema=Schema.object({
        "brief_path": Schema.string(),
        "sources_path": Schema.string()
    })
)

# ---------- Agent Controller ----------
class ResearchBriefAgentMCP:
    def __init__(self, server: Server):
        self.server = server
        self.server.add_tool(docsearch_tool, self.docsearch_impl)
        self.server.add_tool(saveoutput_tool, self.saveoutput_impl)

    async def run(self, topic: str):
        plan = await self.server.call_llm("Propose sections", context={"topic": topic})
        retrieval = await self.server.call_tool("DocSearch", {"query": topic, "k": 3})
        draft = await self.server.call_llm("Draft bullets", context={"chunks": retrieval})
        reflection = await self.server.call_llm("Check rubric", context={"draft": draft})
        await self.server.call_tool("SaveOutput", {"brief_markdown": draft, "sources": self._collect_sources(

    async def docsearch_impl(self, call: ToolCall):
        return [{
            "doc_id": "doc123",
            "title": "Role of Ubiquitin in Cancer",
            "year": 2021,
            "page": 4,
            "text": "p53 degradation is tightly controlled by MDM2 ubiquitination.",
            "cite_label": "Smith 2021, p.4"
        }]

    async def saveoutput_impl(self, call: ToolCall):
        return {"brief_path": "out/brief.md", "sources_path": "out/sources.json"}

    def _collect_sources(self, retrieval):
        return [{
            "doc_id": r["doc_id"], "pages_used": [r["page"]]
        } for r in retrieval]
```

3. Differences: Framework-Free vs MCP

• Tool Definition:

- Framework-Free: Python classes (DocSearch, SaveOutput).
- MCP: Tools declared with schemas (inputs/outputs enforced).

• Invocation:

- Framework-Free: Agent calls Python methods directly.
- MCP: Agent uses server.call_tool and server.call_llm.

• Safety:

- Framework-Free: Relies on developer discipline.
- MCP: Enforces contracts, only declared tools usable.

• Observability:

- Framework-Free: TraceEvent logs each tool call.
- MCP: Built-in structured logging and audits.

• Extensibility:

- Framework-Free: New tools = new Python classes.
- MCP: Add Tool with schema, no change in agent core.

• Interoperability:

- Framework-Free: App-specific only.
- MCP: Protocol-driven, reusable across ecosystems.