Exercise Session 13

Building Local LLM Agents with Tool Calling and MCP

Prepared by Eeshaan Jain

1 Background & Motivation

Modern AI systems almost never operate completely on their own. Even though today's language models are remarkably capable, they still have some important limitations:

- their knowledge is fixed at the moment they were trained,
- they cannot perform exact or complex computations on their own, and
- they cannot directly interact with software, devices, or the physical world.

If we want to build AI assistants that are genuinely useful in real applications, we need a way for models to overcome these built-in constraints.

This is where **tool calling** (often referred to as function calling) becomes crucial. Tool calling lets an LLM recognize when it needs outside help, construct a structured request for a specific tool, and then use the returned results to continue the conversation or reasoning process. You can think of tool calling as giving the model a pair of hands: suddenly it can check the weather, look up flight information, perform accurate math, interact with databases, or even control smart devices.

But if every developer implements tools in their own ad-hoc way, we end up with a messy ecosystem—tools that don't work together, duplicated effort, and systems that are hard to maintain or extend.

The Model Context Protocol (MCP) solves this problem by defining a standardized way to expose tools to AI models. MCP makes tools discoverable, supports tools that are hosted remotely, and enables tools to be composed together. In practice, this means that any AI agent can connect to any tool server that speaks the MCP standard, creating a far more modular, reusable, and scalable tool ecosystem.

In this exercise, you will build a complete local agent system from the ground up. You will work on:

- Running language models locally with tool-calling support
- Implementing the agent loop that orchestrates model-tool interactions
- Creating MCP servers that expose tools via a standard protocol
- Integrating multiple tool sources into a unified agent system

2 Assignment Overview

Your system will consist of three main components:

2.1 Local LLM Server

You will begin by running Qwen3-1.7B locally using vLLM, a high-performance inference engine that exposes an OpenAI-compatible API. vLLM supports structured tool calling out of the box, which means your locally hosted model can request tools in the same way cloud-hosted models do.

2.2 Agent Loop

Next, you'll implement the agent loop, the coordinating logic that ties everything together. The agent loop is responsible for:

- sending user queries to the model,
- detecting when the model wants to call a tool,
- executing that tool and returning the results,
- and repeatedly feeding those results back to the model until it reaches a final answer.

This iterative process continues until the model has gathered enough information to provide a final answer.

2.3 MCP Integration

Finally, you will integrate Model Context Protocol (MCP) components into your system. MCP provides a standardized way to expose tools (via MCP servers) and to discover and invoke them (via MCP clients). Instead of writing custom integrations for each tool, you will rely on the shared MCP protocol to connect your agent to any compliant tool server.

3 Part 1: Setting Up Local LLM Inference

In the first part of the exercise, you will set up a fully local language model that your agent system can communicate with. To make this possible, we will use vLLM, a modern and highly optimized inference engine designed specifically for running large language models efficiently.

3.1 Introduction to vLLM

vLLM is a high-performance library that dramatically simplifies running LLMs on your own machine. Unlike many traditional inference frameworks, vLLM is built around a technology called PagedAttention, which enables fast, memory-efficient execution even for models with long context windows. This allows you to experiment with advanced model behaviors—like tool calling, multi-turn conversations, or long reasoning traces—without needing expensive hardware or cloud APIs.

In this exercise, you'll use vLLM to host Qwen3-1.7B, a lightweight open-source model that supports OpenAI-style tool calling. Because vLLM exposes an OpenAI-compatible API, your agent can interact with the local model in exactly the same way it would interact with a cloud-based model

3.2 Tasks

3.2.1 Task 1.1: Install vLLM

Install vLLM using pip. For CPU-only systems, use the `vllm-cpu` package instead. Consult the vLLM documentation for platform-specific installation instructions if you encounter issues.

3.2.2 Task 1.2: Launch the Server

Write a script that starts the vLLM server with the Qwen3-1.7B model. Be sure to: (1) enable automatic tool-choice detection, (2) use the "hermes" tool-call parser, (3) select a port (default: 8000), and (4) adjust context length if your system has limited memory.

3.2.3 Task 1.3: Verify Functionality

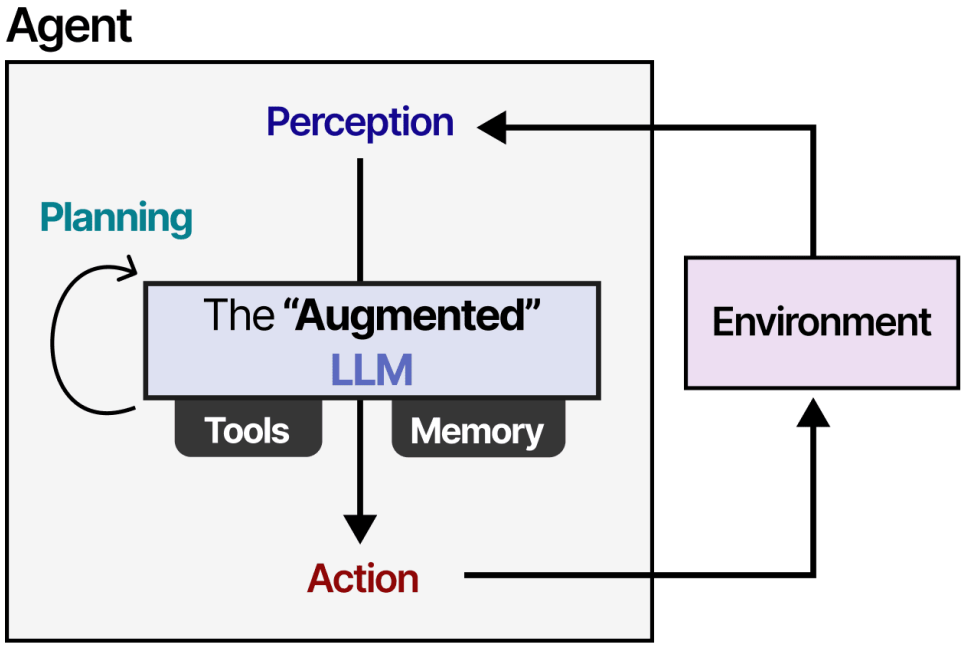
Test that your server is running correctly by making a simple HTTP request to the models endpoint. The server should respond with information about the loaded model.

4 Part 2: Building the Agent Loop

4.1 Understanding the Agent Loop

An agent is fundamentally an iterative process that alternates between reasoning (via the LLM) and acting (via tools). The agent loop orchestrates this, maintaining context across iterations and deciding when to continue or terminate.

The loop follows this flow:



Each iteration adds to the conversation history, allowing the model to build upon previous tool results. The loop terminates when the model produces a final text response instead of tool calls.

4.2 Tasks

4.2.1 Task 2.1: Design the Agent Architecture

Implement an agent class that manages the full agent loop. It should include:

- A client for communicating with your vLLM server
- Storage for conversation history (messages list)
- A registry of available tools with their schemas
- A main `run()` method that implements the agent loop

Consider how you'll structure tool storage. Will you use a dictionary, a list, or a custom tool registry class? How will you handle tool naming conflicts?

4.2.2 Task 2.2: Implement Local Tools

Create at least five Python tools that your agent can use. Each tool requires:

- A Python function that implements the tool's logic
- An OpenAI-compatible JSON schema describing the tool (name, description, parameters)
- Input validation and error handling
- Appropriate return types that the LLM can process

Required tools:

1. `get_current_time`: Returns current date and time (consider timezones)
2. `calculate`: Safely evaluates mathematical expressions (use `ast.literal_eval` or similar, never `eval()`)
3. `read_file`: Reads contents of a local file (handle encoding, large files)
4. `write_file`: Writes content to a file (consider append vs overwrite)
5. `search_web`: Simulated web search returning mock data (or integrate a real API)

When designing tool schemas, be explicit and detailed in your descriptions. The quality of tool descriptions directly impacts the model's ability to use them correctly.

4.2.3 Task 2.3: Implement Tool Execution Logic

Build the machinery to execute tool calls:

- Parse tool calls from the LLM response (handle the specific format your model uses)
- Look up the corresponding Python function
- Extract and validate arguments
- Execute the function and capture results or errors
- Format results in an LLM-readable form

Consider edge cases: What if the model requests a non-existent tool? What if arguments are malformed? How will you communicate errors back to the model?

4.2.4 Task 2.4: Add Safety Mechanisms

Implement safeguards to ensure reliability:

- Maximum iteration limit (e.g., 10 iterations) to prevent infinite loops
- Timeouts for individual tool executions
- Input validation for all tools (type checking, range validation)
- Safe evaluation for the calculator (no arbitrary code execution)

4.3 Testing Requirements

Create comprehensive test cases covering:

- Simple queries that don't require any tools
- Single tool usage (e.g., "What time is it?")
- Multi-step reasoning requiring multiple tools
- Error cases: invalid tool calls, tool failures, malformed arguments
- Boundary cases: maximum iterations, long conversations

5 Part 3: Building an MCP Server

5.1 Understanding MCP

The Model Context Protocol (MCP) provides a standardized way for AI agents to discover and call external tools without hard-coding them into the agent. Instead of bundling tools directly into your system, you expose them through an MCP server that the agent can connect to. This architecture offers:

- **Modularity**: Tools can be developed, tested, and deployed independently
- **Reusability**: One MCP server can serve multiple agents
- **Remote hosting**: Tools can run locally, remotely, or in the cloud
- **Version control**: Tool implementations can be updated without changing agent code
- **Language independence**: Tools can be implemented in any language

MCP uses JSON-RPC 2.0 for communication, which can run over WebSockets (for network communication) or stdio (for local processes). Key protocol methods include:

- `initialize`: Handshake and capability negotiation
- `tools/list`: Discover available tools and their schemas
- `tools/call`: Execute a specific tool with provided arguments

5.2 Tasks

5.2.1 Task 3.1: Work with the MCP server

Use a library such as `fastmcp` or the official MCP Python SDK to build your server. These libraries follow best practices and greatly simplify MCP development by handling protocol details for you.

5.2.2 Task 3.2: Create a Notes Management MCP Server

Implement an MCP server that exposes tools for managing markdown notes in a designated directory.

Required tools:

1. `list_notes`: List all markdown files in a configured directory
2. `read_note`: Read contents of a specific note by filename
3. `create_note`: Create a new markdown file with given content
4. `search_notes`: Search note contents by keyword or regex pattern
5. `update_note`: Append content to an existing note

5.2.3 Task 3.3: Test Your Server

Build a small test client that connects to your MCP server and verifies that all functionality works as expected. Your client should:

- establish a connection and confirm the MCP handshake,
- list all available tools and display their schemas,
- call each tool with valid inputs,

6 Part 4: Integrating MCP with Your Agent

6.1 Bridging Local and Remote Tools

In this part of the assignment, you will integrate everything you have built so far: the local agent from Part 2 and the MCP server from Part 3. Your goal is to create a unified system in which local tools (regular Python functions) and remote MCP tools (accessed over a protocol) appear identical from the agent's perspective. This means the agent must be able to discover tools dynamically, decide which backend to call, and handle errors gracefully—regardless of whether a tool is executed locally or on a remote server.

This requires careful design around:

- **Discovery**: Dynamically fetching tool schemas from MCP servers
- **Execution**: Routing tool calls to the appropriate backend (local function vs MCP RPC)
- **Error handling**: Gracefully handling network failures, timeouts, and unavailable servers
- **Namespacing**: Preventing conflicts when tools from different sources have the same name

6.2 Tasks

Update your agent from Part 2 so that it supports both local tools and MCP-provided tools. Your agent should include methods to establish and manage connections to one or more MCP servers. When the agent loads tools, it should merge remote tool schemas with local ones into a unified registry, while retaining metadata indicating each tool's origin. You must then modify your tool execution logic so that tool calls are routed correctly: local calls should execute Python functions, while remote calls should be delegated to your MCP client. Your agent should also handle mixed workflows where the model uses both local and remote tools within the same reasoning process

7 Part 5: Multiple MCP Servers

7.1 Building a Tool Ecosystem

Real-world agents often need access to diverse capabilities from multiple specialized sources. In this part, you'll extend your system to connect to multiple MCP servers simultaneously.

7.2 Tasks

Develop a second MCP server that provides tools in a different functional domain from your notes server (such as a tool for the weather, or a calculator).

8 Bonus Exercises

8.1 Challenge 1: Tool Recommendation System

Before querying the LLM, analyze the user's input and pre-select relevant tools to include in the prompt. This reduces token usage and can improve response time.

8.2 Challenge 2: Tool Result Caching

Implement caching for deterministic tools. If a tool is called with identical arguments within a time window, return the cached result instead of re-executing.

8.3 Challenge 3: Parallel Tool Execution

When the LLM requests multiple independent tools simultaneously, execute them in parallel using threading or async/await rather than sequentially.

8.4 Challenge 4: Connect to External MCP Servers

Find MCP servers available in public repositories. Try connecting your agent to use tools from servers you didn't create.