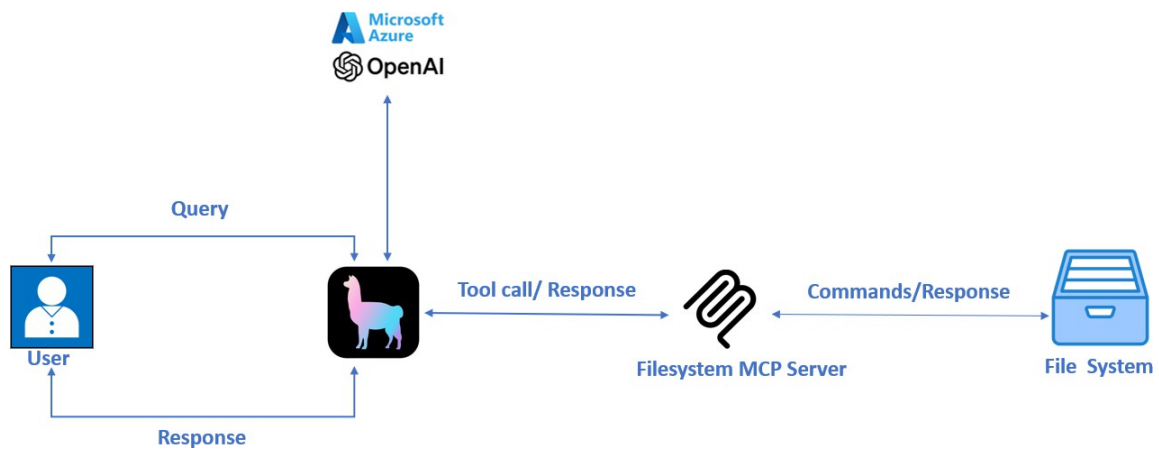


MCP Filesystem Agent

Overview



The application provides an interactive interface where users can:

1. Navigate local directories using natural language commands.
2. List directories and files
3. Read file contents
4. Perform basic file operations

Prerequisites

Before you begin, ensure you have the following installed:

- Python 3.8 or higher
- Node.js and npx (for the MCP filesystem server)
- Azure OpenAI API access
- uv package manager (faster alternative to pip)

Project Setup

Installing uv Package Manager

If you don't have uv installed yet:

```
pip install uv
```



Creating a Project Environment

Create a new project environment using uv:

```
uv init llamaMCP_project
cd llamaMCP_project
```

Installing Required Packages

Install all required packages using uv:

```
uv add dotenv llama-index-llms-azure-openai==0.3.2 llama-index-core==0.12.36 llama-index-tools-mcp==0.1.2 "mcp[cli]>=1.6.0"
```

Environment Configuration

Create a .env file in your project directory with the following Azure OpenAI configuration:

```
AZURE_OPENAI_API_KEY=your_azure_openai_api_key_here
AZURE_OPENAI_ENDPOINT=your_azure_endpoint_here
AZURE_OPENAI_API_VERSION=your_api_version_here
```

Running the Application

1. Place the **llamaMCP.py** file in your project directory
2. Update the path in the StdioServerParameters to match your desired root directory:

```
server_params = StdioServerParameters(
    command="npx",
    args=[
        "-y",
        "@modelcontextprotocol/server-filesystem",
        "D:/YourDesiredDirectory"
    ],
)
```

3. Run the application:

```
python llamaMCP.py
```

4. Enter your queries when prompted

Example Queries

- "List all files in the current directory"
- "Navigate to Documents/Reports"
- "Read config.json"
- "Show me the contents of readme.txt"



- "What files are in the images folder?"

Understanding the Code

Let's break down the main components of the llamaMCP.py script:

1. Imports and Configuration

```
import asyncio
import os
from dotenv import load_dotenv
from llama_index.llms.azure_openai import AzureOpenAI
# ... more imports
```

This section imports necessary libraries and loads environment variables from your .env file.

2. Azure OpenAI Setup

```
llm = AzureOpenAI(
    api_key=AZURE_OPENAI_API_KEY,
    azure_endpoint=AZURE_OPENAI_ENDPOINT,
    deployment_name=AZURE_OPENAI_DEPLOYMENT,
    api_version=AZURE_OPENAI_API_VERSION,
)
```

This section initializes the Azure OpenAI LLM using your API credentials.

3. MCP Filesystem Server

```
server_params = StdioServerParameters(
    command="npx",
    args=["-y", "@modelcontextprotocol/server-filesystem", "allowed-directory"],
)
```

This configures the Model Context Protocol (MCP) server for filesystem access. The server allows the application to interact with files on your system.

4. Chat Memory

```
memory = ChatMemoryBuffer.from_defaults(token_limit=1500)
```

This creates a memory buffer for the agent to maintain conversation context.

5. Main Application Loop

The main() function establishes connections and handles user interaction:

```
async def main():
    # ... implementation details
```

This function:



- Sets up tools for file system by connecting to the MCP server
- Creates a ReActAgent with those tools
- Manages the chat loop, handling user input and displaying responses

Technical Details

MCP Tools

The application uses the Model Context Protocol (MCP) to interact with the filesystem. Available tools include:

1. **read_file**: Read complete file contents
2. **read_multiple_files**: Read multiple files at once
3. **write_file**: Create or overwrite files
4. **edit_file**: Make selective text edits
5. **create_directory**: Create new directories
6. **list_directory**: Show directory contents
7. **move_file**: Move or rename files
8. **search_files**: Find files with patterns
9. **get_file_info**: Get file metadata details
10. **list_allowed_directories**: Show accessible directories

ReAct Agent

The application uses a ReAct (Reasoning and Acting) agent that:

1. Interprets natural language commands
2. Plans appropriate actions using the available tools
3. Executes those actions
4. Provides responses based on the results