# Adapting MCP Host Tool Use for OpenAI-Compatible Backends (LM Studio & Azure OpenAI)

## 1. Introduction

### 1.1. Objective

This report provides a detailed technical guide for modifying a custom Model Context Protocol (MCP) Host application to utilize LM Studio or Azure OpenAI Service as the Large Language Model (LLM) backend. The primary focus is adapting the tool use mechanism, previously integrated with Ollama, to align with the OpenAI API specification for function calling. This adaptation leverages the widespread adoption of the OpenAI standard, which both LM Studio and Azure OpenAI implement, ensuring broader compatibility and access to advanced model capabilities.1

### 1.2. Context: The Shift Towards OpenAI's Tool Use Standard

Tool use, often referred to as function calling, significantly enhances the capabilities of LLMs by enabling them to interact with external systems, APIs, and data sources.3 This allows models to access real-time information, perform actions in other software, and overcome the limitations of their static training data.5 The OpenAI API specification for defining and handling these tool interactions has emerged as a de facto industry standard. Consequently, numerous platforms, including local development environments like LM Studio 1 and enterprise cloud services like Azure OpenAI Service 8, strive for compatibility with this specification. This standardization contrasts with potentially bespoke or less formalized methods that might be employed by other backends, necessitating adaptation when switching providers.

It is important to contextualize this within the broader MCP framework. MCP itself is an open protocol designed to standardize interactions between AI applications (MCP Hosts) and external tools or data sources (MCP Servers).4 This report specifically addresses the interaction *between the MCP Host application and the LLM backend* concerning tool use, assuming the MCP Host continues to interact with MCP Servers according to the MCP specification for tool discovery and execution.

### 1.3. Scope of Modifications

Migrating the MCP Host application from an Ollama backend to an OpenAI-compatible one requires targeted modifications within the application's orchestration layer. This layer mediates between the MCP interactions and the LLM API calls. The key areas requiring adaptation include:

1. **Schema Translation:** Modifying the logic that translates MCP tool definitions into the format expected by the LLM backend.
2. **LLM Request Formatting:** Ensuring API requests to the LLM include tool definitions in the correct OpenAI format.
3. **LLM Response Parsing:** Updating the logic to correctly interpret the LLM's response when it requests a tool call, according to the OpenAI specification.
4. **Backend Connection & Authentication:** Adjusting connection parameters (endpoints) and implementing appropriate authentication mechanisms for LM Studio and Azure OpenAI Service.

## 2. The OpenAI Tool Use / Function Calling Standard

### 2.1. Overview

OpenAI's function calling mechanism provides a structured way for developers to describe external functions or tools to GPT models.25 Based on the user's input and the descriptions of available tools, the model can intelligently determine if executing one or more functions would be beneficial. If it decides to use a tool, it does not execute the function itself. Instead, it generates a structured JSON object within its response, specifying the name of the function(s) it wishes to call and the arguments it deems appropriate based on the conversation context.7 The responsibility of parsing this JSON, executing the actual function (often involving interaction with external APIs or databases), and returning the result to the model lies entirely with the calling application – in this context, the MCP Host.26

### 2.2. Defining Tools: The tools Parameter Structure

To enable function calling, the application must provide the model with definitions of the available tools. This is done via the tools parameter, an array included in the request payload to the Chat Completions API endpoint (e.g., /v1/chat/completions).27 Each object within the tools array represents a single tool the model can potentially use.

For custom functions, the structure of each tool definition object is as follows:

* **type** (string, required): Must be set to "function" to indicate a custom function tool.25
* **function** (object, required): An object containing the detailed definition of the function.
  + **name** (string, required): The exact name of the function that the application code will execute. The model will use this name in its response if it decides to call the function.25
  + **description** (string, optional but highly recommended): A clear, natural language description of the function's purpose, inputs, and outputs. The model relies heavily on this description to understand when and how to use the function effectively. Well-crafted descriptions are crucial for reliable tool selection.25
  + **parameters** (object, required): A JSON Schema object defining the structure, types, and constraints of the arguments the function accepts.25 This schema guides the model in generating the correct arguments.
    - **type** (string, required): Must be "object" at the top level of the parameters schema.
    - **properties** (object, required): Defines the individual parameters of the function. Each key in this object is a parameter name, and its value is another JSON Schema object specifying the parameter's type (e.g., "string", "number", "boolean", "array", "object"), description, and potentially other constraints like enum for specific allowed values.28 Nested objects and arrays are fully supported, allowing for complex argument structures.
    - **required** (array of strings, optional): Lists the names of parameters within properties that the model *must* provide values for when calling the function.28

**Example tools Parameter Structure:**

JSON

,  
 "description": "The temperature unit to use"  
 }  
 },  
 "required": ["location"]  
 }  
 }  
 },  
 {  
 "type": "function",  
 "function": {  
 "name": "search\_products",  
 "description": "Search for products based on query, category, and optional price constraints",  
 "parameters": {  
 "type": "object",  
 "properties": {  
 "query": {  
 "type": "string",  
 "description": "The search query string"  
 },  
 "category": {  
 "type": "string",  
 "description": "The product category to search within"  
 },  
 "max\_price": {  
 "type": "number",  
 "description": "Optional maximum price filter"  
 },  
 "options": {  
 "type": "object",  
 "description": "Additional search options",  
 "properties": {  
 "sort\_by": { "type": "string", "enum": ["relevance", "price\_asc", "price\_desc"]},  
 "include\_out\_of\_stock": { "type": "boolean", "default": false }  
 }  
 }  
 },  
 "required": ["query", "category"]  
 }  
 }  
 }  
]

*(Example based on structures from 25)*

The adoption of JSON Schema for parameter definition is a significant aspect of this standard.25 JSON Schema is itself a mature and widely understood specification for describing JSON data structures. By leveraging it, OpenAI provides a powerful and standardized method for developers to communicate complex function signatures to the LLM. This allows for precise definitions including nested objects, arrays, enumerations (enums), and clear indication of required fields. This expressiveness helps the model generate more accurate and correctly formatted arguments, reducing the likelihood of errors during function execution compared to less formal or more ambiguous parameter definition methods.

### 2.3. Handling Tool Calls: The tool\_calls Response Structure

When the model determines that one or more functions should be called to fulfill the user's request, the API response signals this intent. Specifically, the finish\_reason field in the response object will have the value tool\_calls.26 The details of the requested call(s) are contained within a tool\_calls array, located in the message object of the response choice (typically accessed via response.choices.message.tool\_calls).26

Each element within the tool\_calls array is an object representing a single function call request. The structure of this tool\_call object is:

* **id** (string, required): A unique identifier generated by the API for this specific tool call instance (e.g., call\_abc123). This ID is essential for correlating the function call request with its execution result when sending the result back to the model in a subsequent API call.26
* **type** (string, required): The type of the tool being called. For custom functions, this will always be "function".26
* **function** (object, required): An object containing the specifics of the function invocation.
  + **name** (string, required): The name of the function that the model intends to call, matching one of the names provided in the tools parameter of the request.26
  + **arguments** (string, required): A JSON *string* representing the arguments for the function, as generated by the model. The application code must parse this string into a JSON object to access the individual argument values.26

**Example Response Snippet with tool\_calls:**

JSON

{  
 "id": "chatcmpl-xxxxxxxxxxxxxxxxxxxxxxx",  
 "object": "chat.completion",  
 "created": 1700000000,  
 "model": "gpt-4o",  
 "choices":  
 },  
 "finish\_reason": "tool\_calls"  
 }  
 ],  
 "usage": {... }  
}

*(Example based on structures from 26)*

A key feature of this design is its inherent support for *parallel function calling*. The tool\_calls field is an array, explicitly allowing the model to request multiple, independent function calls within a single response turn.33 The MCP Host application must be architected to handle this possibility, iterating through the array and potentially initiating multiple tool executions concurrently or sequentially as appropriate.

The inclusion of a unique id for each tool\_call is critical for managing these potentially multiple calls. It provides an unambiguous way to link each function execution result back to the specific request made by the model. This is vital in the subsequent step where results are submitted back to the model, especially if function executions complete asynchronously or out of order. This design enables more sophisticated agentic behavior where multiple pieces of external information can be gathered or multiple actions taken within one conversational exchange, significantly enhancing the model's capabilities compared to systems limited to a single function call per turn.

### 2.4. Submitting Tool Results

Following the reception of a tool\_calls response and the successful execution of the requested function(s) by the application (MCP Host), the results must be communicated back to the LLM. This allows the model to incorporate the external information or the outcome of the action into its final response to the user.

The standard OpenAI flow involves making another call to the Chat Completions API. In this subsequent call, the application appends one or more new messages to the conversation history. For each executed function call, a message with role: "tool" is added. This message must include 36:

* **tool\_call\_id** (string, required): The unique id from the corresponding tool\_call object received in the previous response. This links the result back to the specific request.
* **content** (string, required): The return value of the executed function, typically represented as a string (e.g., JSON stringified if the function returns complex data).

After providing these tool results, the model can process them and generate a natural language response for the end-user. While the exact mechanism of getting this result back into the LLM's context within an MCP Host architecture might involve interaction with the MCP Server/Client loop, the fundamental requirement is that the Host ensures the LLM eventually receives the function output correlated with the initial request via the tool\_call\_id.

## 3. Comparative Analysis: OpenAI vs. Ollama Tool Formats

### 3.1. Preamble

This section provides a comparative analysis between the OpenAI tool use format (detailed in Section 2) and the tool/function calling mechanism employed by Ollama. It is important to note that this comparison relies heavily on the specifics of the Ollama implementation as documented in the *previous report* associated with this project, which was not provided as input for this current analysis. The points below are based on common patterns observed in LLM integrations but require validation against the specific details of the prior Ollama setup used in the MCP Host application.

### 3.2. Tool Definition Schema (tools equivalent)

A primary difference often lies in how tools are defined and presented to the LLM.

* **OpenAI:** Uses a structured tools array, where each tool is an object explicitly typed as "function". Parameter definitions strictly adhere to the JSON Schema standard, offering high expressiveness and validation capabilities.25
* **Ollama (Hypothesized):** Depending on the specific model and interface used with Ollama, tool definitions might follow a different structure. Possibilities include:
  + A less standardized format embedded within the prompt itself.
  + A custom JSON or YAML structure specific to the Ollama interface or a particular model's fine-tuning.
  + Potentially simpler parameter definitions not strictly adhering to JSON Schema, possibly lacking support for complex types, nesting, or explicit requirement declarations.
  + The overall structure might be a single object or a different array format compared to OpenAI's tools array.

Key differences to investigate based on the previous report:

* Is the root definition an array or object?
* What are the field names for tool name, description, and parameters?
* Does Ollama utilize JSON Schema for parameters, or a different format?
* How are required parameters, enums, nested structures, and different data types handled?

### 3.3. Tool Call Response Structure (tool\_calls equivalent)

The format in which the LLM signals its intent to call a tool also typically differs.

* **OpenAI:** Returns a specific finish\_reason of tool\_calls and includes a structured tool\_calls array within the message object. Each element in the array represents a distinct call with a unique id, name, and arguments (as a JSON string).26 This structure inherently supports parallel function calls.
* **Ollama (Hypothesized):** The response indicating a tool call might be:
  + Embedded within the main content response, requiring string parsing or regex matching to extract the call details.
  + A custom JSON object within the response, potentially with different field names for the function name and arguments.
  + Possibly limited to requesting only one function call per response turn.
  + Arguments might be provided as a pre-parsed JSON object or in a different string format.
  + A unique call identifier equivalent to OpenAI's tool\_call\_id might be absent or structured differently.

Key differences to investigate based on the previous report:

* How does Ollama signal a tool call (e.g., specific field, content pattern)?
* Where is the tool call information located in the response?
* Does the structure support multiple calls? If so, how?
* What are the field names for function name and arguments?
* Are arguments provided as a JSON string or a parsed object?
* Is there a unique identifier for each call request?

### 3.4. Key Takeaways from Comparison

The migration from Ollama to an OpenAI-compatible backend necessitates a clear understanding of these structural and semantic differences. Developers must adapt the MCP Host's orchestration layer to translate MCP tool definitions into the OpenAI tools format and parse the OpenAI tool\_calls response structure.

While OpenAI's reliance on JSON Schema might require a more complex translation step compared to potentially simpler Ollama formats, it offers greater precision and standardization. Similarly, OpenAI's explicit tool\_calls array provides a robust mechanism for handling function calls, including parallel execution, which might be simpler to parse reliably than methods involving string matching within the main content.

The following table summarizes potential areas of difference, requiring confirmation against the previous report's Ollama details:

**Table 1: Comparison of Potential Tool/Function Calling Formats**

| **Feature** | **Ollama Format (Hypothesized - Verify w/ Prev. Report)** | **OpenAI Format (Section 2)** |
| --- | --- | --- |
| **Tool Definition Root** | Varies (e.g., Prompt injection, Custom JSON/YAML) | tools: Array of Objects |
| **Tool Type Identifier** | May be implicit or custom | type: "function" |
| **Function Name Field** | Varies | function.name |
| **Function Desc. Field** | Varies | function.description |
| **Parameter Definition** | Varies (Potentially simpler, less formal) | function.parameters: JSON Schema Object (type: "object") |
| **Parameter Properties** | Varies | function.parameters.properties: Object of JSON Schemas |
| **Required Param. Syntax** | Varies or unsupported | function.parameters.required: Array of strings |
| **Response Indicator** | Varies (e.g., Content pattern, Custom JSON field) | finish\_reason: "tool\_calls" |
| **Tool Call Structure** | Varies (Potentially single call, embedded in content) | message.tool\_calls: Array of Objects |
| **Call Identifier** | Varies or absent | id (unique string per call) |
| **Called Function Name** | Varies | function.name (within tool\_calls element) |
| **Argument Format** | Varies (e.g., Parsed object, Custom string format) | function.arguments (JSON string needing parsing) |
| **Parallel Call Support** | Varies (Potentially limited or unsupported) | Yes (via tool\_calls array) |

This table serves as a high-level guide. The specific implementation details for the Ollama integration, as documented previously, are essential for accurate mapping during the migration process. Understanding these differences is the first step towards adapting the MCP Host's orchestration logic.

## 4. Adapting the MCP Host Application's Orchestration Layer

### 4.1. Overview of Orchestration Layer Changes

The MCP Host application's orchestration layer serves as the crucial intermediary between the high-level MCP interactions (like discovering tools from MCP Servers 4) and the low-level specifics of the chosen LLM backend's API. With the shift from Ollama to an OpenAI-compatible backend (LM Studio or Azure OpenAI), this layer requires significant adaptation to align with the OpenAI tool use standard detailed in Section 2. The core responsibilities of this layer – translating tool definitions for the LLM and parsing the LLM's requests to use tools – remain, but the specific formats and logic must change.

### 4.2. Modifying Schema Translation (MCP Tool -> OpenAI tools)

The previous implementation likely included a "Schema Translation" step (as referenced from Section 9 of the prior report) responsible for converting MCP Tool definitions into the format expected by Ollama. This existing translation logic must now be modified to target the OpenAI tools array format.

The process involves:

1. **Input:** Receiving MCP Tool definitions. These might be discovered from MCP Servers via mechanisms like the ListToolsRequest.3 The input schema for these tools might be represented using formats like Zod schemas, which are then converted to a standard JSON Schema representation.3
2. **Mapping:** Translating the MCP Tool definition fields to the corresponding OpenAI function definition fields:
   * The MCP Tool's name maps directly to function.name.
   * The MCP Tool's description maps directly to function.description. Providing a clear and accurate description here is vital for the LLM's ability to choose the correct tool.25
   * The MCP Tool's input schema (assumed to be available as a JSON Schema object, potentially after conversion from Zod or another format 3) maps to the function.parameters object. This requires ensuring the schema structure conforms to OpenAI's expectations: a root type: "object", a properties object defining each parameter with its type and description, and an optional required array listing mandatory parameters.28 Careful mapping of data types (string, number, boolean, array, object) and constraints is necessary.
3. **Packaging:** Encapsulating each translated function definition within the standard OpenAI tool structure: { "type": "function", "function": {... } }.
4. **Assembly:** Collecting all translated tool objects into a JSON array, which will serve as the value for the tools parameter in the API request to the LLM.

The presence of this pre-existing schema translation step represents a sound architectural choice. It decouples the MCP tool discovery mechanism from the specifics of the LLM backend's API. This abstraction means the migration primarily involves changing the *output format* of the translation process, rather than requiring a fundamental rewrite of how the MCP Host discovers or represents tools internally. This significantly reduces the complexity and risk associated with switching LLM backends.

### 4.3. Updating LLM Request Generation

The component within the orchestration layer responsible for constructing and sending the API request to the LLM backend must be updated.

* **Include tools Array:** Ensure that the JSON array generated in the modified Schema Translation step (Section 4.2) is correctly included as the value for the tools parameter in the API request payload (e.g., within the JSON body of a POST request to /v1/chat/completions).
* **Consider tool\_choice:** Evaluate and implement a strategy for the tool\_choice parameter.9
  + "auto" (or omitting the parameter): Allows the model to decide whether to call a function and which one(s) to call. This is often the default and most flexible option.
  + "none": Forces the model to generate a standard text response and prevents it from calling any functions.
  + "required": Forces the model to call at least one function from the provided list.41 The model will choose the most relevant one(s).
  + {"type": "function", "function": {"name": "my\_specific\_tool"}}: Forces the model to call only the specified function.9 The appropriate choice depends on the specific interaction logic desired by the MCP Host application. For general-purpose interaction, "auto" is common.

### 4.4. Updating LLM Response Parsing (OpenAI tool\_calls -> MCP Host Logic)

The logic that processes the LLM's response needs substantial modification to handle the OpenAI tool\_calls format, replacing the previous Ollama response parsing.

1. **Detect Tool Call Intent:** Check if the finish\_reason field in the API response is exactly "tool\_calls".26 If not, the model generated a text response, and the existing logic for handling text content should apply.
2. **Access tool\_calls Array:** If finish\_reason is "tool\_calls", access the tool\_calls array located at response.choices.message.tool\_calls.26
3. **Iterate and Parse:** Loop through each tool\_call object within the array. For each object:
   * Extract the unique id string.30 This ID should be stored temporarily, associated with this specific call request, as it will be needed when sending the execution result back to the model.
   * Extract the function name from function.name.26
   * Extract the JSON string from function.arguments.26
   * **Parse Arguments:** Attempt to parse the function.arguments string into a language-specific object or dictionary (e.g., using json.loads() in Python or JSON.parse() in JavaScript).26 Implement robust error handling for this step, as malformed JSON (though less common with newer, well-tuned models) could cause failures.
4. **Trigger MCP Tool Invocation:** Use the extracted function name and the parsed arguments object to initiate the actual tool execution process within the MCP Host. This typically involves:
   * Identifying the corresponding MCP Client associated with the MCP Server that provides this tool.
   * Sending an invocation request (using the MCP protocol's primitives) to the MCP Server via the Client, passing the necessary arguments.6
5. **Handle Multiple Calls:** The parsing logic must correctly handle the tool\_calls array containing multiple elements, potentially triggering multiple MCP tool invocations.33 Depending on the nature of the tools and application requirements, these could be handled sequentially or in parallel.
6. **Correlate Results:** When the result of an MCP tool execution is received back from the MCP Server (via the Client -> Host flow 6), the stored tool\_call\_id associated with the initial request must be used to correctly formulate the role: tool message for the subsequent API call back to the LLM (as described in Section 2.4).

Robust parsing is essential here. The system must gracefully handle potential errors, such as invalid JSON in the arguments string, and must be designed to process every element in the tool\_calls array if multiple calls are requested. Failure to parse arguments correctly or to handle the array structure will lead to incomplete or failed tool execution.

## 5. Integrating Specific Backends: LM Studio and Azure OpenAI

While both LM Studio and Azure OpenAI Service adhere to the OpenAI API specification for tool use, integrating them into the MCP Host application involves distinct configurations for connection and authentication.

### 5.1. LM Studio Integration

LM Studio provides a convenient way to run LLMs locally, offering an API endpoint designed for compatibility with the OpenAI specification.2

* **Compatibility:** LM Studio explicitly supports the OpenAI Function Calling / Tool Use API, acting as a drop-in replacement for requests targeting OpenAI.1 It works with various models, including those known for good tool-use performance like specific versions of Qwen, Mistral, and Llama.1 However, it's noted that smaller models or those not specifically fine-tuned for tool use might generate improperly formatted tool\_calls that LM Studio cannot parse correctly into the standard response structure.7
* **Connection:**
  + LM Studio runs a local HTTP server, typically accessible at http://localhost:1234 by default.2 The port can be configured within the LM Studio application.
  + To connect the MCP Host application, the primary change involves configuring the OpenAI client library (e.g., Python's openai library, JavaScript's openai library) to use the LM Studio server's address as the base\_url or baseURL. The specific path is usually http://localhost:1234/v1.2
  + Example (Python):  
    Python  
    from openai import OpenAI  
    client = OpenAI(  
     base\_url="http://localhost:1234/v1",  
     api\_key="lm-studio" # Placeholder key  
    )  
    *(Based on 2)*
* **Authentication:** For local connections to LM Studio, standard authentication is typically not required. However, many OpenAI client libraries mandate an API key. In such cases, a non-empty placeholder string like "lm-studio", "sk-placeholder", or any arbitrary value can usually be provided as the api\_key.2
* **Model Identifier:** The model parameter in the Chat Completions API call should be set to the specific model identifier of the model currently loaded within the LM Studio application (e.g., "lmstudio-community/Meta-Llama-3.1-8B-Instruct-GGUF").2 This identifier can be found within the LM Studio UI.

Using LM Studio provides significant advantages during development and testing. It acts as a local emulator for the OpenAI API, allowing developers to iterate on the MCP Host's orchestration logic (schema translation, response parsing) using local LLMs without incurring cloud API costs or dealing with network latency.1 The high degree of compatibility ensures that code developed and tested against LM Studio should require minimal changes (primarily configuration) to work with cloud-based OpenAI-compatible services like Azure OpenAI.

### 5.2. Azure OpenAI Service Integration

Azure OpenAI Service provides access to OpenAI's models within the Azure cloud environment, offering enterprise-grade features, security, and scalability.

* **Compatibility:** Azure OpenAI implements the OpenAI API specification, including support for tool use (function calling) via the Chat Completions endpoint.8 Note that older API versions might use deprecated parameters like functions and function\_call; ensure usage of recent API versions (like 2023-12-01-preview or later GA versions like 2024-06-01, 2024-10-21) that support the tools and tool\_choice parameters.9
* **Endpoint Configuration:** Azure OpenAI utilizes a unique endpoint structure that differs from the standard OpenAI API and LM Studio. The URL format for the Chat Completions API is: https://{your-resource-name}.openai.azure.com/openai/deployments/{deployment-id}/chat/completions?api-version={api-version} 8
  + {your-resource-name}: The specific name assigned to your Azure OpenAI resource in the Azure portal.
  + {deployment-id}: The custom name given to the specific model deployment within your Azure OpenAI resource.8 This is crucial as it identifies the exact model instance to use, distinct from the base model name (e.g., you might deploy gpt-4o and name the deployment my-gpt4o-deployment).
  + api-version: A mandatory query parameter specifying the API version being targeted (e.g., 2024-10-21). Check the Azure OpenAI documentation for the latest recommended stable or preview versions.48 The MCP Host's connection logic must be configured to use this specific endpoint structure, either by setting the base URL appropriately in the client library or using Azure-specific SDK features.
* **Authentication Mechanisms:** Azure provides more sophisticated and secure authentication options compared to a simple static API key, aligning with enterprise security requirements.46
  + **API Key:** An API key can be obtained from the "Keys and Endpoint" section of the Azure OpenAI resource in the Azure portal. This key must be included in the api-key HTTP header of each request.46 While simple, this method is generally discouraged for production environments due to the risks associated with static keys.53 Azure API Management can be employed to manage these keys more securely.50
  + **Microsoft Entra ID (Azure Active Directory):** This is the recommended approach for enhanced security, leveraging Azure's identity and access management capabilities.50
    - **Role Assignment:** The identity (user, application, or managed identity) making the API call must be granted an appropriate Azure RBAC role on the Azure OpenAI resource, typically the "Cognitive Services OpenAI User" role.50
    - **Token Acquisition:** The application uses Azure Identity client libraries (e.g., azure-identity for Python) to acquire an OAuth 2.0 bearer token from Entra ID.46
    - **Authorization Header:** The obtained token is included in the Authorization HTTP header, prefixed with Bearer (e.g., Authorization: Bearer <token>).48
    - **Supported Identities:**
      * *Managed Identity:* For applications hosted within Azure (VMs, App Service, AKS, Functions, etc.). The identity is managed by Azure, eliminating the need for storing credentials in code.50 Can be System-Assigned (tied to a specific resource) or User-Assigned (standalone resource assignable to multiple services).
      * *Service Principal:* Represents an application identity within Entra ID. Authentication uses client credentials (client ID and secret) or certificates. Suitable for applications running anywhere, including on-premises or local development machines.51
      * *User Identity:* Authenticates as an interactive user, typically used during development via tools like Azure CLI or IDE logins (DefaultAzureCredential often handles this automatically).52 The MCP Host application should ideally use the Azure Identity libraries, particularly DefaultAzureCredential, which can automatically detect and use various credential sources (environment variables, managed identity, developer credentials) based on the environment it's running in.46 This provides flexibility for different deployment scenarios (local development, testing, production on Azure).

Integrating with Azure OpenAI introduces considerations typical of enterprise cloud services. The requirement to use a deployment-id reflects Azure's resource management model, allowing multiple model deployments within a single resource. The strong emphasis on Microsoft Entra ID authentication provides robust security features like role-based access control, conditional access policies, and centralized credential management, which are often essential for production systems handling sensitive data or performing critical actions via tool use.

### 5.3. Backend Connection Parameter Summary

The following table summarizes the key configuration differences when connecting the MCP Host application to Ollama (hypothesized), LM Studio, and Azure OpenAI Service.

**Table 2: Backend Connection Parameter Comparison**

| **Parameter** | **Ollama Format (Hypothesized - Verify w/ Prev. Report)** | **LM Studio** | **Azure OpenAI Service** |
| --- | --- | --- | --- |
| **Base Endpoint/URL** | Varies (e.g., http://localhost:11434) | http://localhost:1234/v1 (Default) | https://{resource}.openai.azure.com/openai/deployments/{deployment}/chat/completions?api-version={version} |
| **Model Identifier** | Model name (e.g., llama3) | Loaded model identifier from LM Studio UI | Deployment ID (custom name given during deployment) |
| **Authentication** | Typically None for local | Placeholder API Key (e.g., "lm-studio") | 1. API Key (api-key header) <br> 2. Microsoft Entra ID (Bearer Token in Authorization header via Azure Identity) |
| **API Version Param** | N/A or specific to Ollama interface | N/A (Implicit in /v1 path) | Required query parameter (e.g., api-version=2024-10-21) |

This table highlights the specific configuration points that must be adapted in the MCP Host's connection logic and deployment settings when switching between these backends.

## 6. Summary of Required Modifications

Adapting the custom MCP Host application from an Ollama backend to an OpenAI-compatible one (LM Studio or Azure OpenAI) involves targeted changes primarily within the orchestration layer and configuration settings. The key modifications are:

* **6.1. Configuration:**
  + Update backend connection parameters: Modify the endpoint/base URL, API key/authentication credentials, and model/deployment identifiers in the application's configuration to match the target backend (LM Studio or Azure OpenAI), referencing Table 2.
* **6.2. Orchestration Layer - Schema Translation:**
  + Rewrite the existing logic that translates MCP Tool definitions (discovered from MCP Servers) from the Ollama format to the OpenAI tools parameter format, ensuring adherence to the JSON Schema structure detailed in Section 2.2.
* **6.3. Orchestration Layer - Request Generation:**
  + Modify the code that constructs the API request to the LLM to include the correctly formatted tools array.
  + Implement the desired strategy for the tool\_choice parameter (auto, none, required, or specific function).
* **6.4. Orchestration Layer - Response Parsing:**
  + Update the response handling logic to check for finish\_reason: "tool\_calls".
  + If detected, parse the message.tool\_calls array, extracting the id, function.name, and function.arguments (parsing the JSON string) for each requested call.
  + Ensure the logic can handle multiple (parallel) tool calls within a single response.
  + Map the extracted information to trigger the appropriate MCP tool invocation via the corresponding MCP Client/Server. Store the tool\_call\_id for correlating results.
* **6.5. Authentication (Azure Specific):**
  + If targeting Azure OpenAI and choosing Microsoft Entra ID authentication (recommended over API keys for production), implement the necessary token acquisition logic using Azure Identity libraries (e.g., DefaultAzureCredential) and ensure the application's identity has the required RBAC role assigned on the Azure OpenAI resource.

## 7. Conclusion and Recommendations

### 7.1. Recap

Migrating the MCP Host application's tool use mechanism from an Ollama backend to OpenAI-compatible alternatives like LM Studio or Azure OpenAI involves adapting the application's orchestration layer to conform to the widely adopted OpenAI API standard for function calling. This requires modifying the schema translation process to produce the OpenAI tools format, updating API request generation, rewriting response parsing logic to handle the tool\_calls structure (including parallel calls), and configuring the correct connection endpoints and authentication methods for each target backend.

### 7.2. Benefits

This migration offers several advantages:

* **Standardization:** Aligns the application with a prevalent industry standard for tool use, simplifying future integrations and maintenance.
* **Compatibility:** Enables the use of a broader range of LLM providers and platforms that adhere to the OpenAI API, including powerful models available through Azure OpenAI.
* **Enhanced Capabilities:** Leverages the robust features of the OpenAI tool use specification, such as parallel function calling and detailed parameter schemas using JSON Schema.
* **Flexibility:** Facilitates easier switching between local development/testing (LM Studio) and cloud deployment (Azure OpenAI) due to API compatibility.

### 7.3. Recommendations

To ensure a successful transition and robust operation, the following recommendations should be considered:

1. **Thorough Testing:** Conduct comprehensive testing across different scenarios:
   * Use LM Studio for rapid local iteration and validation of the core logic changes (schema translation, response parsing).
   * Test against Azure OpenAI to validate endpoint configuration, authentication, and behavior with specific deployed models.
   * Cover cases involving single tool calls, multiple parallel tool calls, model requests for clarification (no tool call), and model refusal to call tools. Test edge cases like ambiguous user requests or tools with complex parameter structures.
2. **Robust Error Handling:** Implement comprehensive error handling mechanisms for:
   * API calls to the LLM backend (network issues, authentication failures, rate limits, invalid requests).
   * Parsing the function.arguments JSON string (handle potential malformed JSON, although less likely with compliant models).
   * The entire MCP tool execution cycle (errors during communication with MCP Servers or during the tool's execution itself).
3. **Security Best Practices:** Prioritize security, especially when connecting to Azure OpenAI in production environments.
   * Strongly prefer Microsoft Entra ID authentication (Managed Identity or Service Principal) over static API keys.
   * Follow the principle of least privilege when assigning roles (e.g., "Cognitive Services OpenAI User").
   * Securely manage any client secrets or certificates if using Service Principal authentication. Store sensitive configuration outside of source code (e.g., environment variables, Azure Key Vault 51).
4. **Logging and Monitoring:** Implement detailed logging throughout the tool use orchestration flow. Track requests to the LLM, parsed tool\_calls, initiation of MCP tool invocations, received results, and any errors encountered. Monitoring key metrics (e.g., tool call success rate, latency, error types) can help identify issues and understand model behavior.
5. **Internal Documentation:** Update the MCP Host application's technical documentation to reflect the integration with LM Studio and Azure OpenAI, the new tool use mechanism based on the OpenAI standard, and the updated configuration and authentication procedures.

#### Works cited

1. LM Studio 0.3.6, accessed April 8, 2025, <https://lmstudio.ai/blog/lmstudio-v0.3.6>
2. OpenAI Compatibility API | LM Studio Docs, accessed April 8, 2025, <https://lmstudio.ai/docs/api/openai-api>
3. Connecting Astra DB to Claude Desktop Using the Model Context Protocol (MCP), accessed April 8, 2025, <https://jherr2020.medium.com/connecting-astra-db-to-claude-desktop-using-the-model-context-protocol-mcp-e878e3902046>
4. The Model Context Protocol (MCP) by Anthropic: Origins, functionality, and impact - Wandb, accessed April 8, 2025, <https://wandb.ai/onlineinference/mcp/reports/The-Model-Context-Protocol-MCP-by-Anthropic-Origins-functionality-and-impact--VmlldzoxMTY5NDI4MQ>
5. Building Agents with Model Context Protocol - Full Workshop with Mahesh Murag of Anthropic - YouTube, accessed April 8, 2025, <https://www.youtube.com/watch?v=kQmXtrmQ5Zg>
6. Model Context Protocol (MCP) an overview - Philschmid, accessed April 8, 2025, <https://www.philschmid.de/mcp-introduction>
7. Tool Use | LM Studio Docs, accessed April 8, 2025, <https://lmstudio.ai/docs/advanced/tool-use>
8. Work with chat completion models - Azure OpenAI Service - Microsoft Learn, accessed April 8, 2025, <https://learn.microsoft.com/en-us/azure/ai-services/openai/how-to/chatgpt>
9. How to use function calling with Azure OpenAI Service - Microsoft Learn, accessed April 8, 2025, <https://learn.microsoft.com/en-us/azure/ai-services/openai/how-to/function-calling>
10. Implementing Model Context Protocol (MCP) with Claude Desktop: A Developer's Guide, accessed April 8, 2025, <https://guido-salimbeni.medium.com/implementing-model-context-protocol-mcp-with-claude-desktop-a-developers-guide-e36ab746617e>
11. Claude MCP - Model Context Protocol, accessed April 8, 2025, <https://www.claudemcp.com/>
12. An Introduction to Model Context Protocol - MCP 101 - DigitalOcean, accessed April 8, 2025, <https://www.digitalocean.com/community/tutorials/model-context-protocol>
13. Model Context Protocol - GitHub, accessed April 8, 2025, <https://github.com/modelcontextprotocol>
14. Introducing the Model Context Protocol - Anthropic, accessed April 8, 2025, <https://www.anthropic.com/news/model-context-protocol>
15. Model Context Protocol (MCP) - Anthropic API, accessed April 8, 2025, <https://docs.anthropic.com/en/docs/agents-and-tools/mcp>
16. Is Anthropic's Model Context Protocol Right for You? - WillowTree Apps, accessed April 8, 2025, <https://www.willowtreeapps.com/craft/is-anthropic-model-context-protocol-right-for-you>
17. What Is the Model Context Protocol (MCP) and How It Works - Descope, accessed April 8, 2025, <https://www.descope.com/learn/post/mcp>
18. What is Model Context Protocol (MCP): Explained - Composio, accessed April 8, 2025, <https://composio.dev/blog/what-is-model-context-protocol-mcp-explained/>
19. The Model Context Protocol (MCP) — A Complete Tutorial | by Dr. Nimrita Koul - Medium, accessed April 8, 2025, <https://medium.com/@nimritakoul01/the-model-context-protocol-mcp-a-complete-tutorial-a3abe8a7f4ef>
20. A Comprehensive Overview of the Model Context Protocol (MCP) | by Astropomeai, accessed April 8, 2025, <https://medium.com/@astropomeai/a-comprehensive-overview-of-the-model-context-protocol-mcp-f65150da0aa0>
21. Model Context Protocol: Introduction, accessed April 8, 2025, <https://modelcontextprotocol.io/introduction>
22. What is Model Context Protocol (MCP)? How it simplifies AI integrations compared to APIs | AI Agents That Work - Norah Sakal, accessed April 8, 2025, <https://norahsakal.com/blog/mcp-vs-api-model-context-protocol-explained/>
23. Model Context Protocol (MCP): A comprehensive introduction for developers - Stytch, accessed April 8, 2025, <https://stytch.com/blog/model-context-protocol-introduction/>
24. Anthropic's Model Context Protocol (MCP): A Deep Dive for Developers - Medium, accessed April 8, 2025, <https://medium.com/@amanatulla1606/anthropics-model-context-protocol-mcp-a-deep-dive-for-developers-1d3db39c9fdc>
25. OpenAI Platform, accessed April 8, 2025, <https://platform.openai.com/docs/guides/function-calling>
26. openai-cookbook/examples/How\_to\_call\_functions\_with\_chat\_models.ipynb at main - GitHub, accessed April 8, 2025, <https://github.com/openai/openai-cookbook/blob/main/examples/How_to_call_functions_with_chat_models.ipynb>
27. API Reference - OpenAI API, accessed April 8, 2025, <https://platform.openai.com/docs/api-reference/chat/create>
28. json - What is the OpenAI Chat Completion API tools/functions ..., accessed April 8, 2025, <https://stackoverflow.com/questions/78282380/what-is-the-openai-chat-completion-api-tools-functions-property-format>
29. Extended or minimal Schemas for tool parameters? - API - OpenAI Developer Community, accessed April 8, 2025, <https://community.openai.com/t/extended-or-minimal-schemas-for-tool-parameters/578636>
30. There is no mention of tool\_calls anywhere in API reference - OpenAI Developer Community, accessed April 8, 2025, <https://community.openai.com/t/there-is-no-mention-of-tool-calls-anywhere-in-api-reference/819421>
31. Tool Calls placed inside content - Bugs - OpenAI Developer Community, accessed April 8, 2025, <https://community.openai.com/t/tool-calls-placed-inside-content/648334>
32. OpenAI Function Calling Tutorial: Generate Structured Output - DataCamp, accessed April 8, 2025, <https://www.datacamp.com/tutorial/open-ai-function-calling-tutorial>
33. Parallel Function Calling - API - OpenAI Developer Community, accessed April 8, 2025, <https://community.openai.com/t/parallel-function-calling/626868>
34. Parallel Function Calling in OpenAI using ChatGPT - Pragnakalp Techlabs, accessed April 8, 2025, <https://www.pragnakalp.com/parallel-function-calling-in-openai-using-chatgpt/>
35. How to call functions with chat models - OpenAI Cookbook, accessed April 8, 2025, <https://cookbook.openai.com/examples/how_to_call_functions_with_chat_models>
36. Multiple Tools in Assistant Streaming API - Bugs - OpenAI Developer Community, accessed April 8, 2025, <https://community.openai.com/t/multiple-tools-in-assistant-streaming-api/690663>
37. Launch of multiple functions (Assistant v2) - API - OpenAI Developer Community, accessed April 8, 2025, <https://community.openai.com/t/launch-of-multiple-functions-assistant-v2/842636>
38. Assistants API expects multiple tool\_calls, even though only one function provided - Bugs, accessed April 8, 2025, <https://community.openai.com/t/assistants-api-expects-multiple-tool-calls-even-though-only-one-function-provided/509641>
39. What is MCP in AI? Model Context Protocol Simply Explained [No BS] - YouTube, accessed April 8, 2025, <https://www.youtube.com/watch?v=Xs9AwE2lyHg>
40. Microsoft partners with Anthropic to create official C# SDK for Model Context Protocol, accessed April 8, 2025, <https://devblogs.microsoft.com/blog/microsoft-partners-with-anthropic-to-create-official-c-sdk-for-model-context-protocol>
41. New API feature: forcing function calling via `tool\_choice: "required"`, accessed April 8, 2025, <https://community.openai.com/t/new-api-feature-forcing-function-calling-via-tool-choice-required/731488>
42. OpenAI Function Calling Tutorial for Developers - Vellum AI, accessed April 8, 2025, <https://www.vellum.ai/blog/openai-function-calling-tutorial>
43. How to get API Tool Call to choose just one tool accurately, accessed April 8, 2025, <https://community.openai.com/t/how-to-get-api-tool-call-to-choose-just-one-tool-accurately/590315>
44. LM Studio as a Local LLM API Server, accessed April 8, 2025, <https://lmstudio.ai/docs/local-server>
45. OpenAI Compatible Providers: LM Studio - AI SDK, accessed April 8, 2025, <https://sdk.vercel.ai/providers/openai-compatible-providers/lmstudio>
46. Azure Chat Completions example (preview) - OpenAI Cookbook, accessed April 8, 2025, <https://cookbook.openai.com/examples/azure/chat>
47. Azure Chat Completion models with your own data (preview) - OpenAI Cookbook, accessed April 8, 2025, <https://cookbook.openai.com/examples/azure/chat_with_your_own_data>
48. Azure OpenAI Service REST API reference - Microsoft Learn, accessed April 8, 2025, <https://learn.microsoft.com/en-us/azure/ai-services/openai/reference>
49. Azure OpenAI Service REST API preview reference - Microsoft Learn, accessed April 8, 2025, <https://learn.microsoft.com/en-us/azure/ai-services/openai/reference-preview>
50. Authenticate and authorize access to Azure OpenAI APIs using Azure API Management, accessed April 8, 2025, <https://learn.microsoft.com/en-us/azure/api-management/api-management-authenticate-authorize-azure-openai>
51. Authenticate requests to Azure AI services - Learn Microsoft, accessed April 8, 2025, <https://learn.microsoft.com/en-us/azure/ai-services/authentication>
52. LazaUK/AOAI-EntraIDAuth-SDKv1: Authenticating with Entra ID (former Azure AD) to access Azure OpenAI models in Python SDK v1.x - GitHub, accessed April 8, 2025, <https://github.com/LazaUK/AOAI-EntraIDAuth-SDKv1>
53. Enhancing Azure OpenAI Security with Microsoft Entra ID & OAuth 2.0, accessed April 8, 2025, <https://hernandezpaul.wordpress.com/2024/06/21/enhancing-azure-openai-security-with-microsoft-entra-id-oauth-2-0/>
54. Provide custom authentication to Azure OpenAI Service through a gateway - Learn Microsoft, accessed April 8, 2025, <https://learn.microsoft.com/en-us/azure/architecture/ai-ml/guide/azure-openai-gateway-custom-authentication>
55. Authentication in Azure OpenAI Service | Journey Of The Geek, accessed April 8, 2025, <https://journeyofthegeek.com/2023/04/02/authentication-in-azure-openai-service/>
56. Authenticate to Azure OpenAI the right way using Microsoft Entra ID - YouTube, accessed April 8, 2025, <https://www.youtube.com/watch?v=keIs_bpgdXs>