# Building a Custom MCP Host Application with Local LLMs via Ollama

## 1. Introduction

**Purpose:** This report provides a comprehensive technical guide for developing a custom desktop application in Python. The application aims to replicate core functionalities of existing Model Context Protocol (MCP) Host applications, such as Claude Desktop 1, by acting as an MCP Host itself.

**Core Components:** The central theme is the integration of a locally running Large Language Model (LLM) accessed via Ollama 3, a custom user interface (UI) – conceptually represented by frameworks like Streamlit or those compatible with OpenWebUI – and leveraging the official Model Context Protocol (MCP) Python SDK 5 for interacting with the MCP ecosystem.

**Key Feature:** A critical requirement addressed is the ability to parse a configuration file, similar in structure and purpose to claude\_desktop\_config.json 1, to discover, configure, launch, and manage external MCP server processes, particularly those communicating via standard input/output (stdio).2

**Motivation/Benefits:** This approach offers several advantages over relying on proprietary cloud-based solutions. It grants users local control over the LLM, enhancing data privacy and reducing reliance on external APIs. It allows for deep customization of the user experience and the specific tools integrated. Furthermore, it enables developers to tap into the rapidly expanding open ecosystem of MCP servers and tools.8

**Target Audience & Prerequisites:** This document is intended for Python developers with a foundational understanding of AI concepts, LLMs, APIs, and asynchronous programming. Prerequisites include a working Python environment, a functional Ollama installation with suitable models downloaded, and familiarity with basic command-line operations.

## 2. Understanding the Model Context Protocol (MCP)

What is MCP?

The Model Context Protocol (MCP) is an open standard, initiated by Anthropic 11, designed to standardize the way AI applications, often referred to as "Hosts," connect and communicate with external data sources and tools, known as "Servers".1 It acts as a universal interface, often analogized to a "USB-C port for AI" 8, allowing diverse AI models and applications to interact with a wide range of tools and data without requiring bespoke integration code for each connection.

The primary motivation behind MCP was to address the inherent complexity of integrating AI systems with the multitude of tools and data sources they might need.11 Before MCP, connecting *M* different AI applications to *N* different tools often required *M×N* unique integrations. MCP aims to simplify this "N×M integration problem" by defining a common protocol, reducing the effort to an "M+N" scenario where each application (Host/Client) and each tool (Server) implements the protocol once.8

MCP Architecture

MCP employs a client-server architecture involving three primary components 9:

1. **Host:** The user-facing application that orchestrates the interaction. Examples include dedicated applications like Claude Desktop 12 or Cursor 12, Integrated Development Environments (IDEs) 9, or the custom application being designed in this report. The Host is responsible for managing MCP Clients, integrating with the LLM, providing the UI, and controlling the overall workflow.9
2. **Client:** A component residing within the Host application. Each Client manages a dedicated, stateful, one-to-one connection with a specific MCP Server.9 It handles the specifics of MCP communication, sending requests to the server and receiving responses or notifications according to the protocol specification.19
3. **Server:** An external program, often lightweight, that exposes specific capabilities (like accessing a filesystem, interacting with a Git repository, querying a database, or calling a web API) through the standardized MCP interface.5 Servers can connect to local resources (e.g., files, databases) or remote services.20 A growing number of pre-built servers exist for common tools like filesystems 2, GitHub 1, Brave Search 1, databases 7, and more.12

Communication Protocol

MCP standardizes the communication layer:

* **Message Format:** Interactions are structured using JSON-RPC 2.0 messages.12 This involves three fundamental message types:
  + **Requests:** Sent by either client or server to invoke a method on the other party, requiring a response. They include a method name, parameters, and a unique ID.
  + **Responses:** Sent in reply to a request, containing either a result or an error object, correlated via the original request ID.
  + **Notifications:** One-way messages sent without requiring a response, used for events like state changes.
* **Transport Mechanisms:** MCP supports multiple ways for clients and servers to exchange these JSON-RPC messages:
  + **stdio (Standard Input/Output):** Used primarily for servers running as local processes, often launched by the Host application itself. Communication happens via the process's standard input and output streams. This is the method used by Claude Desktop for local server integration and is the focus of this report.2
  + **HTTP + SSE (Server-Sent Events):** Used for servers accessible over a network. The client sends requests via HTTP POST, and the server can push asynchronous updates and responses back to the client using the SSE protocol over a persistent HTTP connection.7

Core Primitives

MCP Servers expose their capabilities through several defined primitives 6:

* **Tools:** These represent executable functions or actions that the LLM, operating within the Host, can decide to invoke via the Server.6 Tools are considered "model-controlled" because the LLM typically determines *when* to use a tool based on the user's request and the tool's description provided by the server during discovery.19 This is analogous to function calling in other LLM frameworks.15
* **Resources:** These represent data or content that the Server makes available to the Host application, typically to provide context to the LLM.6 Resources are generally read-only and are considered "application-controlled," meaning the Host application decides which resources to fetch and include in the LLM's context.19 They function similarly to GET requests in REST APIs.6
* **Prompts:** Reusable prompt templates or predefined interaction workflows exposed by the Server.6 These are often "user-controlled," meaning the user might explicitly select a prompt to initiate a specific task.19
* **Sampling:** An advanced primitive allowing the *Server* to request an LLM completion *from the Host*.8 This enables more complex, multi-step agentic behaviors initiated by the server but requires explicit user approval within the Host application for security reasons.8
* **Roots:** Define specific locations or entry points within the Host's environment (e.g., filesystem paths) that the Server might be permitted to access.8

MCP Host Application Responsibilities

Building a custom MCP Host application, like the one envisioned to replace Claude Desktop, involves several key responsibilities, derived from the protocol's architecture and the functionalities of existing hosts 9:

* **(a) UI Provision:** Providing the graphical user interface through which the end-user interacts with the application.
* **(b) LLM Integration:** Connecting to the chosen LLM (in this case, a local one via Ollama), sending prompts and context (including tool descriptions), receiving responses, and interpreting the LLM's intent, particularly its decision to use a tool.
* **(c) Configuration Reading:** Parsing a configuration file (e.g., claude\_desktop\_config.json format) to identify which MCP servers the user wants to enable and how to run them (command, arguments, environment variables).1
* **(d) Server Process Management:** Launching MCP server processes (especially via stdio) as defined in the configuration, monitoring their status, managing their lifecycle (termination), and handling their standard error streams for logging/debugging.2
* **(e) MCP Client Logic:** Implementing the client-side of the MCP protocol using an SDK (like modelcontextprotocol/python-sdk). This includes establishing connections to servers, performing the initialization handshake, discovering available tools and resources, sending tool invocation requests, and receiving results.6
* **(f) Orchestration:** Managing the complex flow of interaction between the user (via the UI), the LLM, and the various connected MCP Servers and their tools. This involves translating requests, routing tool calls, managing state, and synthesizing final responses.19

It's important to recognize that MCP standardizes the *communication interface* between the Host (and its embedded LLM) and the external Servers (tools/data). It does not inherently make the LLM capable of complex reasoning or guarantee the correctness of the Server's implementation. The Host application still requires an LLM sophisticated enough to understand *when* a tool is appropriate based on its description provided via MCP, and the Server must correctly execute the logic associated with that tool. MCP acts as the crucial bridge, facilitating discovery and interaction, but the intelligence resides in the LLM and the tool's implementation.

Furthermore, the choice between stdio and SSE transport mechanisms carries significant architectural implications. Stdio, the focus here, simplifies local execution by directly managing server subprocesses but creates a tight coupling – if the Host application terminates, the stdio servers typically terminate as well. SSE enables servers to run independently, potentially on different machines or in containers, offering greater deployment flexibility but introducing network communication, security, and management complexities absent in the stdio model.7

The following table summarizes the core responsibilities of the custom MCP Host application and maps them to the implementation areas discussed later in this report:

| **Responsibility ID** | **Description** | **Key Implementation Area** | **Relevant Report Sections** |
| --- | --- | --- | --- |
| (a) | Provide the user interface for interaction. | UI Framework (Streamlit/OpenWebUI - Conceptual) | 8 |
| (b) | Integrate with a local LLM (Ollama) for chat and tool-use decisions. | Ollama API Client (Python Library) | 3, 4, 5 |
| (c) | Read configuration file (claude\_desktop\_config.json format) to identify and configure MCP servers. | JSON Parsing, Configuration Management | 7 |
| (d) | Launch, manage (monitor, terminate), and communicate (stdio) with MCP server subprocesses. | subprocess Module, Process Management Logic | 7 |
| (e) | Implement MCP Client logic using the Python SDK for connection, discovery, and protocol interaction. | MCP Python SDK (ClientSession, stdio\_client) | 6 |
| (f) | Orchestrate the interaction flow between UI, LLM, and MCP servers, including schema translation. | Core Application Logic, State Management, Async Handling | 9 |

## 3. Setting Up the Local LLM Environment with Ollama

Introduction to Ollama

Ollama is an increasingly popular tool designed to simplify the process of running open-source Large Language Models (LLMs) locally on personal computers.3 By providing a straightforward command-line interface and an API server, Ollama allows developers and users to download, manage, and interact with various LLMs without relying on cloud infrastructure. This local execution model is particularly relevant for applications prioritizing data privacy, offline capability, and fine-grained control over the AI model.

Installation

Ollama supports macOS, Windows, and Linux. Installation typically involves downloading an installer from the official Ollama website or using a command-line script provided in their documentation.3 Detailed, platform-specific instructions can be found on the Ollama website (ollama.com). It is crucial to ensure Ollama is correctly installed before proceeding.

Pulling Models

Once Ollama is installed, LLMs can be downloaded using the ollama pull command followed by the model name and optional tag (e.g., ollama pull llama3.1).3 For building an MCP Host application capable of tool use, selecting an appropriate model is critical. Models need strong instruction-following capabilities and specific fine-tuning for function/tool calling. Recommended models based on recent documentation and community discussions include variants of Llama 3 (specifically Llama 3.1 or newer, noted for tool use improvements 60), Qwen 2.5 61, or potentially other models explicitly supporting tool calling via Ollama.52 It is essential to consult the Ollama model library or model-specific documentation to verify tool-calling support before downloading.64

Running Ollama

After installation, Ollama typically runs as a background service or daemon.3 On macOS and Linux, it might run automatically after installation. On Windows, it might require manual starting or configuration to run as a service. This background process hosts the LLMs and exposes a REST API, by default, at http://localhost:11434 58, which the custom Python application will use to interact with the models.

Verification

To confirm that Ollama is running and models are available, simple command-line checks can be performed:

* ollama list: Shows the models downloaded locally.
* ollama run <model\_name>: Starts an interactive chat session with the specified model in the terminal.58

Successfully executing these commands indicates that the Ollama environment is ready for integration with the Python application.

A key consideration when using local models, especially for complex tasks like tool use, is that performance and reliability can vary significantly based on the chosen model and the hardware running it. Smaller models, while faster and less resource-intensive, may struggle to consistently understand when to use tools or generate the correct parameters for tool calls.52 Larger, more capable models generally perform better but require more powerful hardware (CPU, RAM, potentially GPU). This implies that developers building this custom host application must carefully select and test models to ensure the desired level of tool-use functionality is achieved, potentially incorporating more sophisticated prompting strategies or error handling logic in the host application to compensate for model limitations.

## 4. Integrating Ollama with Python

To enable the custom MCP Host application to communicate with the locally running Ollama service, a Python client is required to interact with Ollama's REST API.

Ollama Python Client

While it's possible to interact with the Ollama API directly using standard Python HTTP libraries like requests 66, the officially supported ollama-python library provides a more convenient and idiomatic interface.3 This library abstracts away the details of constructing API requests and parsing responses, offering simple functions for common tasks like chat completion and model management. It is the recommended approach for integration.

Installation

The official library can be installed using pip:

Bash

pip install ollama

59

Basic Chat Interaction

The core function for interacting with a model is ollama.chat(). This function sends a conversation history to the specified model and returns the model's response. A typical interaction involves constructing a list of message dictionaries, each with a role (system, user, or assistant) and content.

Python

import ollama  
  
messages = [  
 {'role': 'system', 'content': 'You are a helpful assistant.'},  
 {'role': 'user', 'content': 'Why is the sky blue?'},  
]  
  
try:  
 response = ollama.chat(model='llama3.1', messages=messages)  
 print(response['message']['content'])  
except ollama.ResponseError as e:  
 print(f"Error interacting with Ollama: {e.error} (Status Code: {e.status\_code})")

This example sends a system prompt and a user query to the llama3.1 model (assuming it's pulled) and prints the content of the assistant's response message.4 The model and messages parameters are fundamental.4

Streaming Responses

For a more interactive UI, where the response appears token by token, streaming can be enabled by setting stream=True. The ollama.chat() function then returns an iterator (or asynchronous generator if using AsyncClient) yielding response chunks.

Python

import ollama  
  
messages = [  
 {'role': 'user', 'content': 'Explain quantum physics simply.'},  
]  
  
stream = ollama.chat(  
 model='llama3.1',  
 messages=messages,  
 stream=True,  
)  
  
print("Assistant: ", end='', flush=True)  
for chunk in stream:  
 print(chunk['message']['content'], end='', flush=True)  
print() # Newline after streaming finishes

Each chunk typically contains a portion of the response text in chunk['message']['content'].4

Error Handling

Interactions with Ollama can fail (e.g., the service isn't running, the specified model doesn't exist). The ollama-python library raises an ollama.ResponseError in such cases. Basic error handling involves wrapping API calls in try...except blocks to catch these exceptions and provide informative feedback.4 The exception object often contains details like the error message (e.error) and the HTTP status code (e.status\_code).

Async Client

For applications built using Python's asyncio, the library provides an AsyncClient class with asynchronous versions of the API methods (e.g., AsyncClient().chat()). This is particularly relevant given that MCP interactions will also be asynchronous.4

Python

import asyncio  
import ollama  
  
async def run\_async\_chat():  
 message = {'role': 'user', 'content': 'Why is the sky blue?'}  
 async\_client = ollama.AsyncClient() # Can be configured similarly to Client  
 response = await async\_client.chat(model='llama3.1', messages=[message])  
 print(response['message']['content'])  
  
# asyncio.run(run\_async\_chat()) # Example invocation

Custom Client Configuration

If the Ollama API server is running on a different host or port, or requires custom headers, the ollama.Client (or ollama.AsyncClient) can be instantiated with specific parameters:

Python

import ollama  
  
client = ollama.Client(  
 host='http://192.168.1.100:11434',  
 # headers={'Authorization': 'Bearer YOUR\_TOKEN'} # If needed  
)  
# Use 'client' instance for subsequent calls, e.g., client.chat(...)

4

While the official library greatly simplifies interacting with the Ollama REST API 4, developers should be aware that it acts as an abstraction layer. For intricate debugging or when encountering unexpected behavior, understanding the underlying API calls (documented in Ollama's API documentation, e.g., /api/chat, /api/generate 62) and potentially using tools like curl 58 to inspect raw requests and responses can be beneficial. The library handles common cases, but direct API interaction offers maximum transparency and control.

## 5. Local LLM Tool Use with Ollama

Concept

A key capability for enhancing LLMs beyond simple text generation is "tool use," often referred to as "function calling".7 This mechanism allows the LLM, based on the user's prompt and descriptions of available tools, to request the execution of external code (functions, APIs, etc.). The LLM's role is to identify the need for a tool, select the appropriate tool, and generate the necessary arguments for it. The actual execution of the tool is handled by the calling application (the MCP Host in this context), which then feeds the tool's results back to the LLM to formulate a final response.52

Ollama Support

Ollama provides support for tool calling in its API for models that have been specifically trained or fine-tuned for this capability.52 This is a crucial feature for enabling the custom Host application to leverage external functionalities defined by MCP servers through the local LLM.

Defining Tools for Ollama

To enable tool use, the Host application must describe the available tools to the LLM when making a request to the /api/chat endpoint. This is done via the tools parameter, which expects a JSON array where each element describes one tool.62

* **JSON Schema Structure:** The required structure for each tool definition (specifically for type function) is quite precise 61:  
  JSON  
  {  
   "type": "function",  
   "function": {  
   "name": "function\_name\_string",  
   "description": "Concise description of what the function does.",  
   "parameters": {  
   "type": "object",  
   "properties": {  
   "param1\_name": {  
   "type": "string | integer | number | boolean | array",  
   "description": "Description of param1."  
   },  
   "param2\_name": {  
   "type": "string",  
   "description": "Description of param2.",  
   "enum": ["value1", "value2"] // Optional: Allowed values  
   }  
   //... other parameters  
   },  
   "required": ["param1\_name"] // Optional: List of required parameter names  
   }  
   }  
  }  
    
  Key fields include the function's name, a description for the LLM to understand its purpose, and a parameters object defining the expected inputs using JSON Schema syntax (properties defining each parameter's name, type, and description, and an optional required list).
* **Using the Python Library:** The ollama-python library facilitates providing these tool definitions. While earlier versions or documentation might suggest passing Python functions directly 3, the standard and recommended approach, especially aligning with the API documentation and practices using libraries like Pydantic 71, involves constructing this specific JSON schema. The library likely handles the serialization if provided with structured input (like Pydantic models), but the target format remains this JSON structure. The application code will need to generate this JSON based on the tools discovered from MCP servers.

Prompting for Tool Use

Simply providing tool definitions is often insufficient. Effective tool use requires clear instructions within the prompt (typically the system message) guiding the LLM on the available tools and the circumstances under which they should be used.3

Making the API Call

The tool definitions, formatted as the JSON array described above, are passed to the ollama.chat() function via the tools parameter.

Python

import ollama  
import json  
  
# Assume 'messages' is the conversation history  
# Assume 'ollama\_formatted\_tools' is a list of tool dicts adhering to the JSON schema  
  
try:  
 response = ollama.chat(  
 model='llama3.1', # Model known to support tool use  
 messages=messages,  
 tools=ollama\_formatted\_tools,  
 # format='json' # Optional: Might help ensure response structure  
 )  
 #... process response  
except ollama.ResponseError as e:  
 print(f"Error: {e.error}")

3

Parsing the Response for Tool Calls

When the LLM decides to use a tool, the response message from the assistant will contain a tool\_calls field.52 This field holds a list of requested tool invocations.

JSON

// Example structure within response['message']  
{  
 "role": "assistant",  
 "content": "", // Often empty when tool calls are made  
 "tool\_calls":  
}

The Host application must check for the presence and content of tool\_calls. Each element specifies the function.name to be called and the function.arguments as a dictionary (which might be a JSON string needing parsing depending on the model/library version).

Executing the Tool Call

Crucially, Ollama and the LLM do not execute the tool. They only generate the request. The Host application receives the tool\_calls structure and is responsible for:

1. Identifying the requested function (name).
2. Parsing the arguments.
3. Invoking the actual corresponding code/logic (which, in this application's context, will involve calling the appropriate MCP server via the MCP Python SDK).

Returning Results to LLM

After executing the tool, the Host must send the result back to the LLM to allow it to generate the final user-facing response. This is done by appending two messages to the conversation history and making another call to ollama.chat:

1. The assistant's message that contained the tool\_calls.
2. A new message with role: tool, containing the output/result returned by the executed tool, associated with the specific tool call ID if provided by the API.

Python

# Conceptual example after executing tool and getting 'tool\_result\_content'  
  
# Append the assistant's message that requested the tool call  
messages.append(response['message'])  
  
# Append the tool's result  
messages.append({  
 'role': 'tool',  
 'content': tool\_result\_content,  
 # Potentially include tool\_call\_id if the API supports/requires it  
})  
  
# Call Ollama again with the updated history to get the final response  
final\_response = ollama.chat(model='llama3.1', messages=messages)  
print(final\_response['message']['content'])

52

Challenges and Considerations with Local Models

Using local models via Ollama for tool calling presents unique challenges compared to potentially more mature cloud-based APIs:

* **Inconsistency:** Models might not reliably decide when to call a tool, sometimes failing to call when needed, or calling unnecessarily.63
* **Forced Tool Calling:** Some models, when presented with tools, might *always* attempt to call a tool, even for simple questions that don't require one.64 Workarounds, like defining a generic "respond\_directly" tool that the model can call when no other tool fits, might be necessary.64
* **Ignoring Results:** Models might sometimes ignore the provided tool results in the subsequent turn.
* **Model Compatibility:** Not all models available via Ollama support tool calling effectively, or at all. Explicit verification is needed.64

Developers must anticipate these issues and potentially implement more robust orchestration logic, including specific prompting techniques, validation of LLM outputs, and potentially retries or fallback mechanisms.

A significant implication arises from Ollama requiring a specific JSON schema format for tool definitions. Since MCP servers expose tools whose schemas are discovered via the MCP Python SDK (likely inferred from server-side code definitions like type hints and docstrings 6), a translation layer becomes essential. The Host application's orchestrator must convert the tool schemas obtained from ClientSession.list\_tools() 6 into the precise JSON structure expected by Ollama's tools parameter. This mapping process is a critical step in integrating MCP tools with an Ollama-based LLM.

## 6. Implementing the MCP Client with the Python SDK

Introduction to modelcontextprotocol/python-sdk

The official modelcontextprotocol/python-sdk 5 provides the necessary tools to build both MCP servers and clients in Python. This section focuses on the client-side components required for our custom Host application to connect to and interact with MCP servers, particularly those running as local subprocesses communicating via stdio.

Installation

The SDK can be installed using standard Python package managers. Using uv is recommended by the SDK documentation 6, but pip also works:

Bash

# Using uv  
uv pip install mcp  
  
# Or using pip  
pip install mcp

Depending on usage (e.g., running CLI tools provided by the SDK), additional extras might be needed, like mcp[cli].74

Connecting to stdio Servers

The SDK provides specific utilities for managing connections to servers launched as subprocesses communicating over standard input/output.

* **StdioServerParameters:** This class is used to define how the server subprocess should be launched. It encapsulates the necessary information parsed from the configuration file (claude\_desktop\_config.json-like structure).6  
  Python  
  from mcp import StdioServerParameters  
    
  # Example: Parameters derived from a config entry  
  server\_config = {  
   "command": "uv",  
   "args": [  
   "--directory", "/path/to/server/project",  
   "run", "server.py"  
   ],  
   "env": {"API\_KEY": "some\_value"} # Optional  
   # "cwd": "/path/to/server/project" # Optional  
  }  
    
  server\_params = StdioServerParameters(  
   command=server\_config["command"],  
   args=server\_config.get("args",),  
   env=server\_config.get("env"),  
   cwd=server\_config.get("cwd")  
  )  
    
  This structure directly maps to the command, args, env, and potentially cwd fields found in Claude Desktop's MCP server configurations.2
* **stdio\_client Context Manager:** This asynchronous context manager handles the launching of the server subprocess (as defined by StdioServerParameters) and establishes communication pipes to its stdin and stdout.6 It yields asynchronous read and write stream objects upon successful connection.  
  Python  
  from mcp.client.stdio import stdio\_client  
  import asyncio  
    
  async def connect\_to\_server(params: StdioServerParameters):  
   print(f"Attempting to start and connect to server via stdio: {params.command} {' '.join(params.args or)}")  
   try:  
   async with stdio\_client(params) as (read, write):  
   print("Successfully connected to stdio server.")  
   # Proceed with ClientSession using read, write streams  
   #... session logic here...  
   pass  
   except Exception as e:  
   # Basic error handling for connection/startup failure  
   print(f"Failed to connect to stdio server: {e}")  
   # More specific error handling might be needed  
    
  # Example usage:  
  # asyncio.run(connect\_to\_server(server\_params))

Managing the ClientSession

Once the connection streams (read, write) are obtained from stdio\_client, a ClientSession object is used to manage the MCP communication protocol over that connection.

* **Initialization:** The ClientSession is also an asynchronous context manager, ensuring proper session management and cleanup.6  
  Python  
  from mcp import ClientSession  
  #... inside the `async with stdio\_client(...)` block...  
  async with ClientSession(read, write) as session:  
   # Session is ready for initialization and interaction  
   #...
* **Handshake:** The first crucial step after establishing the session is to perform the MCP initialization handshake using await session.initialize().6 This step involves the client and server exchanging capabilities and negotiating the protocol version, ensuring compatibility before further interaction.18 Failure during initialization indicates a problem with the server or the connection.

Discovering Server Capabilities

After successful initialization, the client can query the server about the functionalities it offers.

* **Listing Tools:** The await session.list\_tools() method retrieves the list of tools exposed by the server.6  
  Python  
  #... inside the `async with ClientSession(...)` block after initialize...  
  try:  
   tools\_response = await session.list\_tools()  
   available\_tools = tools\_response.tools # Accessing the list of tools  
   print(f"Discovered {len(available\_tools)} tools:")  
   for tool in available\_tools:  
   print(f"- Name: {tool.name}")  
   print(f" Description: {tool.description}")  
   # print(f" Schema: {tool.inputSchema}") # Accessing schema details  
  except Exception as e:  
   print(f"Error listing tools: {e}")  
    
  The method returns an object (likely ListToolsResult) containing a list of types.Tool objects.6
* **Tool Schema Representation:** Each types.Tool object encapsulates the information needed by the Host and LLM to understand and use the tool.6 Based on server-side definitions 6 and protocol concepts, this object structure likely includes:
  + name: The string name of the tool (e.g., derived from the Python function name on the server).
  + description: A string describing the tool's purpose (e.g., from the function's docstring).
  + inputSchema: A structured representation (likely a dictionary conforming to JSON Schema principles) detailing the expected input parameters, including their names, types (e.g., 'string', 'integer', 'boolean'), descriptions, and whether they are required.6 The exact structure of inputSchema would be defined by the mcp.types module and the MCP specification. As noted previously, this schema information discovered from the MCP server needs to be translated into the format expected by the Ollama API.
* **Listing Resources/Prompts:** Similarly, session.list\_resources() and session.list\_prompts() can be used to discover available resources and prompts, although tools are the primary focus for replicating Claude Desktop's core MCP interaction.6

Interacting with the Server

Once tools are discovered, the client can invoke them.

* **Calling Tools:** The await session.call\_tool(tool\_name, arguments={...}) method sends a request to the server to execute the specified tool with the provided arguments.6  
  Python  
  # Example: Calling a tool named 'add' discovered earlier  
  tool\_name = "add"  
  arguments\_to\_pass = {"a": 5, "b": 7} # Must match the tool's inputSchema  
    
  try:  
   tool\_result = await session.call\_tool(tool\_name, arguments=arguments\_to\_pass)  
   print(f"Result of calling tool '{tool\_name}': {tool\_result}")  
   # Process the tool\_result (structure depends on the tool's return type)  
  except Exception as e:  
   print(f"Error calling tool '{tool\_name}': {e}")  
    
  The arguments dictionary must conform to the inputSchema discovered for that tool. The method returns the result produced by the server-side tool execution. The structure of this result depends entirely on how the tool is implemented on the server.
* **Reading Resources:** For completeness, session.read\_resource(uri) can be used to fetch data exposed as MCP resources.6

Error Handling & Subprocess Management

While ClientSession and stdio\_client handle basic protocol communication and process launching, robust error handling is essential. Errors can occur during initialize, list\_tools, call\_tool (e.g., server-side execution error, invalid arguments), or due to the underlying subprocess failing.42 The SDK likely raises specific exceptions for protocol errors, which should be caught. However, managing the health and lifecycle of the subprocess itself (e.g., unexpected termination) falls outside the direct scope of ClientSession and requires the mechanisms discussed in Section 7. The use of async with for both stdio\_client and ClientSession ensures resources like streams and potentially the subprocess are cleaned up correctly under normal operation or when exceptions within the block occur.6

The following table summarizes key ClientSession methods relevant to this application:

| **Method** | **Description** | **Key Parameters** | **Return Type (Conceptual)** |
| --- | --- | --- | --- |
| initialize() | Performs initial handshake and capability negotiation with the server. | None | InitializeResult or similar |
| list\_tools() | Retrieves the list of tools exposed by the server. | None | ListToolsResult (contains list of types.Tool) |
| call\_tool() | Invokes a specific tool on the server with given arguments. | tool\_name, arguments | CallToolResult (contains tool output) |
| list\_resources() | Retrieves the list of resources exposed by the server. | None | ListResourcesResult |
| read\_resource() | Reads the content of a specific resource from the server. | uri | Resource content (e.g., bytes, str), mime type |
| list\_prompts() | Retrieves the list of prompts exposed by the server. | None | ListPromptsResult |

While the stdio\_client context manager provides a convenient way to launch and connect to the stdio server subprocess 6, it primarily focuses on establishing the communication channel. The Host application remains responsible for more robust lifecycle management. If the server process crashes, fails to start correctly, hangs, or produces unexpected output on stderr, the stdio\_client might raise an error during connection or communication, but detecting and recovering from these scenarios might require additional monitoring logic implemented using the subprocess module directly (as detailed in the next section).

Furthermore, the list\_tools() method provides the crucial bridge to understanding the server's capabilities.6 The schema information contained within the returned types.Tool objects is essential but, as highlighted, is likely in MCP's native format. This necessitates a transformation step within the orchestrator (Section 9) to convert these schemas into the specific JSON format required by the Ollama API's tools parameter 72, enabling the local LLM to correctly understand and request the use of these MCP-provided tools.

## 7. Managing MCP Server Processes

A core requirement for replicating Claude Desktop's functionality is the ability to launch and manage external MCP server processes based on a configuration file. This involves parsing the configuration, using Python's subprocess module to control the server processes, and handling their input/output streams, particularly for stdio-based servers.

Configuration File Parsing

The application needs to read a configuration file defining the MCP servers to be used. This file should resemble the claude\_desktop\_config.json used by Claude Desktop.1

* **Format:** The expected format is a JSON object containing a top-level key, typically "mcpServers". The value of this key is another JSON object where each key is a user-defined name for a server (e.g., "filesystem", "github", "my-custom-server"), and the value is an object specifying how to run that server.2 For stdio servers, this configuration object must contain:
  + "command": The executable to run (e.g., "python", "node", "npx", "uv", or an absolute path to an executable).2
  + "args": An optional list of command-line arguments to pass to the executable.2
  + "env": An optional dictionary of environment variables to set for the server process.2
  + "cwd": An optional string specifying the working directory for the server process.44

JSON  
// Example claude\_desktop\_config.json  
{  
 "mcpServers": {  
 "filesystem": {  
 "command": "npx",  
 "args": [  
 "-y",  
 "@modelcontextprotocol/server-filesystem",  
 "/Users/username/Documents",  
 "/Users/username/Downloads"  
 ]  
 },  
 "my-python-server": {  
 "command": "/path/to/venv/bin/python",  
 "args": ["/path/to/server\_project/main.py"],  
 "env": {  
 "API\_KEY": "secret-key"  
 },  
 "cwd": "/path/to/server\_project"  
 }  
 }  
}

* **Python Implementation:** Python's built-in json module can be used to load and parse this file. The application should read the file at startup and iterate through the servers defined under "mcpServers".  
  Python  
  import json  
  import os  
    
  def load\_mcp\_config(config\_path="claude\_desktop\_config.json"):  
   """Loads MCP server configurations from a JSON file."""  
   try:  
   with open(config\_path, 'r') as f:  
   config = json.load(f)  
   return config.get("mcpServers", {})  
   except FileNotFoundError:  
   print(f"Warning: Configuration file not found at {config\_path}")  
   return {}  
   except json.JSONDecodeError:  
   print(f"Error: Could not decode JSON from {config\_path}")  
   return {}  
    
  # mcp\_servers\_config = load\_mcp\_config()  
  # for server\_name, server\_config in mcp\_servers\_config.items():  
  # # Launch server using server\_config details  
  # pass

Launching Servers with subprocess

While the MCP Python SDK's stdio\_client uses subprocess internally 42, understanding how subprocess works is crucial for robust management and for setting up the StdioServerParameters.

* **Introduction:** The subprocess module allows Python programs to spawn new processes, connect to their input/output/error pipes, and obtain their return codes.80
* **subprocess.Popen:** For managing long-running server processes and interacting with their stdio streams, subprocess.Popen is the appropriate choice.81 Unlike subprocess.run, Popen executes the command asynchronously (non-blocking) and provides fine-grained control over the child process and its streams.
* **Mapping Config to Popen:** The parsed configuration for each stdio server directly maps to Popen arguments:
  + The command and args list form the args parameter for Popen.
  + env maps to the env parameter.
  + cwd maps to the cwd parameter.
  + Crucially, for stdio communication, stdin, stdout, and stderr must be set to subprocess.PIPE.81 This allows the parent Python process (our Host app) to read from the server's stdout/stderr and write to its stdin.
  + Setting text=True (or universal\_newlines=True) is recommended for easier handling of text-based communication, automatically handling encoding/decoding.80 Alternatively, manual byte encoding/decoding is required.
* **Example (Conceptual Launch - SDK handles actual launch via stdio\_client):**  
  Python  
  import subprocess  
  import sys  
    
  def launch\_server\_process(server\_name: str, config: dict):  
   """Launches an MCP server as a subprocess."""  
   command = [config['command']] + config.get('args',)  
   env\_vars = os.environ.copy() # Start with parent environment  
   env\_vars.update(config.get('env', {})) # Add/override specific vars  
    
   print(f"Launching server '{server\_name}': {' '.join(command)}")  
   try:  
   process = subprocess.Popen(  
   command,  
   stdin=subprocess.PIPE,  
   stdout=subprocess.PIPE,  
   stderr=subprocess.PIPE,  
   text=True, # Use text mode for easier communication  
   encoding='utf-8', # Specify encoding  
   errors='replace', # Handle potential decoding errors  
   env=env\_vars,  
   cwd=config.get('cwd'),  
   # On Windows, might need creationflags if launching console apps differently  
   # creationflags=subprocess.CREATE\_NEW\_CONSOLE if sys.platform == "win32" else 0  
   )  
   print(f"Server '{server\_name}' started with PID: {process.pid}")  
   return process  
   except FileNotFoundError:  
   print(f"Error: Command not found for server '{server\_name}': {config['command']}")  
   return None  
   except Exception as e:  
   print(f"Error launching server '{server\_name}': {e}")  
   return None  
    
  # server\_processes = {}  
  # for name, cfg in mcp\_servers\_config.items():  
  # # Note: In practice, stdio\_client(StdioServerParameters(...)) handles this launch.  
  # # This function illustrates the underlying subprocess call.  
  # # proc = launch\_server\_process(name, cfg)  
  # # if proc:  
  # # server\_processes[name] = proc  
  # pass # SDK handles launch

Managing Subprocesses

Once servers are launched (implicitly by stdio\_client), the Host application needs to manage their lifecycles.

* **Storing Process Handles:** Although stdio\_client manages the process it starts during its context, the Host might need awareness of these processes for monitoring or explicit shutdown. Storing references might be necessary if managing processes outside the SDK's context managers.
* **Monitoring:** Regularly check the status of server processes using process.poll(). A non-None return value indicates the process has terminated. This is crucial for detecting crashes.
* **Termination:** When the Host application shuts down, or if a server needs restarting, its process must be terminated cleanly.
  + process.terminate(): Sends a SIGTERM signal (on POSIX) or TerminateProcess (on Windows), requesting graceful shutdown.
  + process.kill(): Sends a SIGKILL signal (POSIX) or TerminateProcess (Windows) for forceful termination if terminate() fails or is ignored.
  + process.wait(timeout=...): Waits for the process to terminate after sending a signal, preventing zombie processes.82

Python  
def shutdown\_processes(processes\_dict):  
 print("Shutting down MCP server processes...")  
 for name, process in processes\_dict.items():  
 if process.poll() is None: # Check if still running  
 print(f"Terminating server '{name}' (PID: {process.pid})...")  
 process.terminate()  
 try:  
 process.wait(timeout=5) # Wait up to 5 seconds  
 print(f"Server '{name}' terminated with code: {process.returncode}")  
 except subprocess.TimeoutExpired:  
 print(f"Server '{name}' did not terminate gracefully, killing...")  
 process.kill()  
 process.wait() # Wait after kill  
 print(f"Server '{name}' killed.")  
 else:  
 print(f"Server '{name}' already terminated with code: {process.returncode}")  
*Note: Graceful shutdown should ideally be handled via MCP protocol messages (shutdown, exit) managed by ClientSession's context exit, but manual termination is a fallback.*

Communicating via stdin/stdout (Manual vs. SDK)

As established, the MCP Python SDK's stdio\_client and ClientSession abstract the low-level reading from stdout and writing to stdin needed for MCP communication.6 Manual interaction using process.stdin.write(), process.stdout.readline(), or process.communicate() 81 is generally not required for MCP protocol messages but understanding the concept reinforces what the SDK manages.

Handling stderr / Logging

Capturing the standard error stream (stderr) from MCP server processes is vital for debugging.2 When stderr=subprocess.PIPE is used in Popen (as done implicitly by the SDK setup), the Host can read from process.stderr. This should be done asynchronously or in a separate thread to avoid blocking, especially if the server produces a lot of error output. The captured errors should be logged using the Host application's logging system. Claude Desktop, for instance, writes server-specific logs.2

Python

import threading  
import time  
  
def log\_stderr(process, server\_name):  
 """Reads stderr from a process and logs it."""  
 try:  
 for line in iter(process.stderr.readline, ''):  
 print(f"[stderr-{server\_name}]: {line.strip()}", file=sys.stderr)  
 process.stderr.close() # Ensure stream is closed after process ends  
 except ValueError:  
 # stderr might be closed already if process terminated abruptly  
 print(f"[stderr-{server\_name}]: Stream closed.", file=sys.stderr)  
 except Exception as e:  
 print(f"[stderr-{server\_name}]: Error reading stderr: {e}", file=sys.stderr)  
  
  
# When launching conceptually (SDK handles this):  
# stderr\_thread = threading.Thread(target=log\_stderr, args=(process, server\_name), daemon=True)  
# stderr\_thread.start()  
# The SDK likely manages stderr reading internally when using stdio\_client.  
# If direct Popen is used, manual stderr handling like this is necessary.

Error Handling

Beyond protocol errors handled by ClientSession, the Host must handle process-level errors:

* FileNotFoundError: If the command in the config doesn't exist.84
* Permission Errors: If the Host lacks permission to execute the command.
* Server Crashes: Detected via poll(). The Host needs a strategy (e.g., log error, notify user, attempt restart).
* Configuration Errors: Invalid JSON, missing required fields (command).

Robust try...except blocks should surround process launching and management logic.

Managing subprocesses effectively is more than just launching them. It requires continuous monitoring for crashes (checking poll()), capturing and logging diagnostic information from stderr 2, and ensuring clean termination using signals and wait() to prevent resource leaks or zombie processes. While the stdio\_client simplifies the initial launch and connection 42, the overall responsibility for the subprocess lifecycle resilience lies with the Host application developer.

Furthermore, executing arbitrary commands specified in a configuration file carries inherent security risks.2 Although in this scenario the user typically controls their own configuration file, it's a critical consideration. The Host application should ideally run these external processes with the minimum necessary privileges. Validating the command against an allowlist or providing clear warnings to the user about the commands being executed are advisable security practices.

## 8. Building the Custom User Interface (Conceptual)

Goal

The primary objective of the User Interface (UI) component is to provide an intuitive and responsive front-end for users to interact with the local LLM and the capabilities exposed through connected MCP servers. It should facilitate sending prompts, displaying conversation history (including indications of tool use), and managing the interaction flow.

Framework Choices

Several Python frameworks are suitable for building such an interface rapidly. Streamlit is a popular choice for creating data-centric web applications with minimal code and offers components suitable for chat interfaces. Alternatives include Gradio, or potentially integrating with web frameworks like Flask or FastAPI combined with front-end technologies compatible with architectures like OpenWebUI. The choice depends on the developer's familiarity and the desired level of UI customization and complexity.

Core UI Components

Regardless of the specific framework, a functional interface would require:

* **Chat Input:** A text input area (e.g., st.text\_input or st.chat\_input in Streamlit) where the user types their prompts or messages. A "Send" button or handling the Enter key triggers message submission.
* **Message Display Area:** A container to display the conversation history chronologically. This should clearly distinguish between user messages and assistant (LLM) responses. Frameworks often support markdown rendering, allowing for formatted text, code blocks, etc..87 It's also beneficial to visually indicate when the LLM is thinking or when an MCP tool is being executed (e.g., displaying "Using tool: *get\_weather*..."). Results from tool calls should ideally be summarized or presented clearly within the chat flow.
* **Status Indicators:** Visual cues to inform the user about the application's state, such as "Connecting to Ollama," "Connecting to MCP servers," "LLM processing," or "Executing tool X."
* **(Optional) Configuration/Settings:** While the primary method for MCP server configuration is the JSON file, a UI section could potentially display the loaded configuration, show the connection status of each server, or offer settings related to the Ollama connection (e.g., selecting the model).
* **(Optional) Tool Visualization:** A dedicated area (perhaps a sidebar or expandable section) could list the MCP tools discovered from connected servers via list\_tools, showing their names and descriptions. This enhances user awareness of the available capabilities.

Integration Approach

The UI acts as the front-end layer and needs to communicate with the backend orchestration logic (detailed in Section 9). This typically involves:

1. **Sending User Input:** When the user submits a message, the UI framework captures the input and passes it to the backend orchestrator function/method.
2. **Receiving Updates:** The backend orchestrator processes the input, interacts with Ollama and MCP servers, and generates responses or status updates. These updates need to be pushed back to the UI. Given the potentially long-running and asynchronous nature of LLM and tool interactions, using callbacks, message queues, or state management patterns appropriate for the chosen UI framework is essential. For streaming responses from Ollama or progress updates from MCP tools, the UI needs to handle these incremental updates efficiently.

Focus of this Report

This report concentrates on the backend architecture and logic required to integrate Ollama, manage MCP servers via stdio, implement the MCP client using the Python SDK, and orchestrate the interactions. Providing detailed, framework-specific UI code is outside the current scope. However, the backend components are designed to be UI-agnostic, allowing developers to plug in their preferred framework (Streamlit, Gradio, etc.) to handle the presentation layer.

A key technical consideration when selecting a UI framework is its compatibility with asynchronous operations. The backend logic involving Ollama API calls 4 and MCP client interactions using the official Python SDK 6 is heavily reliant on Python's asyncio. Choosing a UI framework that either natively supports async/await or provides clear patterns for integrating with an asynchronous backend (e.g., running the async logic in a separate thread or event loop and communicating results back to the UI thread) will significantly simplify the development process and prevent the UI from becoming unresponsive during backend operations.

## 9. Orchestration: Tying It All Together

The Central Controller

The orchestration layer is the heart of the custom MCP Host application. It acts as the central coordinator, managing the flow of information and control between the User Interface (UI), the Ollama LLM client, and the MCP Client Manager (which handles interactions with multiple MCP servers). This component is typically implemented as a Python class or module responsible for receiving user input, querying the LLM, triggering tool calls via MCP, processing results, and sending responses back to the UI.

Application Flow (Step-by-Step)

The core interaction cycle, initiated by user input, follows a sequence similar to this 19:

1. **UI Input:** The user enters a prompt via the UI component.
2. **Orchestrator Receives Input:** The UI forwards the user's message to the orchestrator.
3. **Prepare LLM Request:**
   * The orchestrator retrieves the current conversation history.
   * It iterates through all active MCP server connections (managed via ClientSession objects). For each connection, it calls await session.list\_tools() to get the currently available tools.6 This list might be fetched dynamically on each turn or retrieved from a cache (with appropriate invalidation logic) to balance responsiveness and performance.55
4. **Schema Translation (MCP -> Ollama):** This is a critical step. The tool schemas obtained from session.list\_tools() (structured according to MCP types 6) must be translated into the specific JSON array format required by the Ollama API's tools parameter.88 This involves mapping MCP tool names, descriptions, and parameter definitions (types, descriptions, required status) to the corresponding fields in the Ollama JSON schema structure.
5. **LLM Tool Decision:** The orchestrator sends the user's prompt, the conversation history, and the *Ollama-formatted* tool schemas to the Ollama client (e.g., ollama.chat).
6. **Parse LLM Response:** The orchestrator receives the response from Ollama.
   * **If Tool Call:** It checks for the presence of the tool\_calls field in the assistant's message.52 If present, it extracts the list of requested tool calls, including the function.name and function.arguments for each.
   * **If Direct Answer:** The response contains the final text answer from the LLM. Proceed to step 11.
7. **Route Tool Call:** For each requested tool call, the orchestrator determines which connected MCP server provides the tool with the matching name. A mapping (e.g., a dictionary) created during the tool discovery phase (step 3) can facilitate this routing.
8. **Execute MCP Tool Call:** The orchestrator uses the appropriate ClientSession object corresponding to the identified server and calls await session.call\_tool(tool\_name, arguments=parsed\_arguments).6 The parsed\_arguments are extracted from the tool\_calls structure in the LLM's response.
9. **Prepare Follow-up LLM Request:** After receiving the result(s) from the MCP tool call(s), the orchestrator constructs a new message list. This list includes the original history, the assistant's message containing the tool\_calls, and one or more new messages with role: tool, providing the output returned by each executed tool.52 This updated message list is then sent back to Ollama (repeating step 5, potentially without the tools parameter if no further tool calls are expected in the immediate next turn, or keeping them to allow chained tool use).
10. **Process Final LLM Response:** The orchestrator receives the subsequent response from Ollama, which should now contain the final, user-facing answer synthesized using the tool results.
11. **UI Update:** The orchestrator sends the final text response back to the UI for display. It should also ideally send intermediate status updates (e.g., "Calling tool X...", "Received result from tool X...") to keep the user informed. If streaming is used, response chunks are sent incrementally.

Schema Translation Details

The translation between the MCP tool schema (as represented by types.Tool from the Python SDK) and the Ollama JSON tool format is crucial. A conceptual function might look like:

Python

from mcp import types as mcp\_types # Assuming types definition  
  
def convert\_mcp\_tool\_to\_ollama\_tool(mcp\_tool: mcp\_types.Tool) -> dict:  
 """Converts an MCP Tool object to the Ollama JSON tool format."""  
 ollama\_params = {"type": "object", "properties": {}, "required":}  
 required\_params =  
  
 # Assuming mcp\_tool.inputSchema structure is known (e.g., JSON Schema like)  
 if mcp\_tool.inputSchema and 'properties' in mcp\_tool.inputSchema:  
 for param\_name, param\_schema in mcp\_tool.inputSchema.get('properties', {}).items():  
 ollama\_params["properties"][param\_name] = {  
 "type": param\_schema.get("type", "string"), # Default or map types  
 "description": param\_schema.get("description", "")  
 }  
 # Handle enums if present in MCP schema and supported by Ollama format  
 if "enum" in param\_schema:  
 ollama\_params["properties"][param\_name]["enum"] = param\_schema["enum"]  
  
 # Determine required parameters based on MCP schema conventions  
 required\_params = mcp\_tool.inputSchema.get('required',)  
 if required\_params:  
 ollama\_params["required"] = required\_params  
  
 return {  
 "type": "function",  
 "function": {  
 "name": mcp\_tool.name,  
 "description": mcp\_tool.description,  
 "parameters": ollama\_params  
 }  
 }  
  
# Usage during orchestration:  
# ollama\_tools\_list = [convert\_mcp\_tool\_to\_ollama\_tool(tool) for tool in all\_discovered\_mcp\_tools]

This function needs to accurately map the structure and data types from the MCP representation (obtained via list\_tools) to the nested JSON structure Ollama expects. Handling type conversions (e.g., MCP schema types to JSON schema types) and identifying required parameters correctly are key aspects.

The necessity for this translation arises because MCP aims to be a general protocol, and its SDK likely represents tool schemas in a canonical way 6, while specific LLM providers like Ollama (or OpenAI, Gemini, etc.) have their own distinct API requirements for how tool information must be formatted.45 The orchestrator must bridge this format gap.

The following table provides a conceptual comparison highlighting the mapping needed:

| **Feature** | **MCP Representation (Inferred from SDK/Spec)** | **Ollama JSON Representation (from B3)** |
| --- | --- | --- |
| Tool Name | tool.name (string) | function.name (string) |
| Tool Description | tool.description (string, likely from docstring) | function.description (string) |
| Parameters Structure | tool.inputSchema (dict, likely JSON Schema based) | function.parameters (object) |
| Parameter Definition | tool.inputSchema['properties'][param\_name] (dict) | function.parameters.properties[param\_name] (object) |
| Parameter Name | Key in tool.inputSchema['properties'] | Key in function.parameters.properties |
| Parameter Type | param\_schema['type'] (string, e.g., 'integer') | properties[key].type (string, e.g., 'integer') |
| Parameter Description | param\_schema['description'] (string) | properties[key].description (string) |
| Required Flag | Presence in tool.inputSchema['required'] list | Presence in function.parameters.required list |
| Allowed Values (Enum) | param\_schema['enum'] (list, optional) | properties[key].enum (list, optional) |

State Management

The orchestrator must manage the state of the conversation, including the history of user, assistant, and tool messages, to provide proper context for subsequent LLM calls. It also needs to track the status of connections to MCP servers.

The orchestrator is arguably the most intricate component of the application. It juggles asynchronous operations involving the UI, the LLM, and potentially multiple MCP clients. It must handle errors gracefully across these different components, manage the conversation state accurately, and perform the vital schema translation between the MCP world and the Ollama API. The design and implementation of this layer heavily influence the overall application's stability, performance, and correctness.

Regarding tool discovery, the decision of whether to call list\_tools on every user turn or to cache the results involves a trade-off.48 Dynamic fetching ensures the application always uses the latest tools offered by a server but introduces latency for the list\_tools calls. Caching speeds up the interaction cycle but risks the LLM attempting to use a tool that is no longer available or missing a newly added tool if the cache becomes stale.55 A balanced approach might involve caching with a reasonable time-to-live (TTL) or implementing a mechanism to listen for server notifications about tool changes (notifications/tools/list\_changed, if supported by the server and SDK 48).

## 10. Conclusion and Next Steps

Summary

This report has outlined the essential steps and components required to build a custom Python application serving as an MCP Host, designed as a local alternative to tools like Claude Desktop. Key stages covered include:

* Setting up a local LLM environment using Ollama and selecting appropriate models with tool-use capabilities.
* Integrating the local LLM into the Python application using the Ollama API client library.
* Understanding the Model Context Protocol (MCP) architecture, communication methods (focusing on stdio), and core primitives (Tools, Resources).
* Utilizing the official MCP Python SDK (modelcontextprotocol/python-sdk) to implement client-side logic for connecting to stdio-based MCP servers.
* Parsing a claude\_desktop\_config.json-style configuration file to identify and configure MCP servers.
* Managing MCP server subprocesses using Python's subprocess module, including launch, monitoring, and termination.
* Implementing the critical schema translation layer to convert MCP tool definitions into the format required by the Ollama API for function calling.
* Orchestrating the complex interaction flow between the user interface (conceptual), the Ollama LLM, and the MCP clients/servers.

Key Achievements

The analysis provides a technical blueprint demonstrating the feasibility of combining the flexibility of local LLMs via Ollama with the standardized connectivity of the Model Context Protocol. It details the specific Python libraries (ollama, mcp, subprocess, json) and architectural patterns needed to achieve this integration, enabling developers to build powerful, customizable, and privacy-preserving AI tools that leverage the growing MCP ecosystem.

Limitations

While comprehensive in its approach to the core backend integration, this report has certain limitations:

* **UI Implementation:** Specific UI code for frameworks like Streamlit or OpenWebUI was not provided, focusing instead on the conceptual requirements and backend integration points.
* **Advanced Error Handling:** While basic error handling was discussed, comprehensive strategies for network resilience (for SSE), complex subprocess recovery, or sophisticated LLM error mitigation (e.g., retries on failed tool calls) require further design.
* **SSE Transport:** Detailed implementation for connecting to remote MCP servers via HTTP/SSE was outside the primary scope, which focused on stdio.
* **Security:** Security considerations were mentioned (e.g., running subprocesses, user consent for sampling/tools 18), but a full security hardening guide was not included.

Future Enhancements

Developers building upon this blueprint can consider several enhancements:

* **Polished User Interface:** Implement a feature-rich UI using Streamlit, Gradio, or another suitable framework, incorporating status indicators, tool visualization, and potentially configuration management.
* **SSE Server Support:** Extend the MCP client manager to handle connections to remote servers using the SDK's SSE transport capabilities.27
* **Robustness:** Implement more sophisticated error handling, retry logic for API calls and tool executions, and potentially automated restarting of crashed MCP server subprocesses.
* **Persistence:** Add functionality to save and load conversation histories.
* **Advanced Agentic Logic:** Explore more complex reasoning patterns within the orchestrator, potentially allowing for multi-step tool use sequences or autonomous planning based on user goals.
* **Ecosystem Contribution:** Develop and share new MCP servers for tools or data sources not yet covered by the community.5

Final Thoughts

The convergence of capable local LLMs, made accessible through tools like Ollama, and standardized protocols like MCP presents a significant opportunity. It empowers developers to build sophisticated AI applications that are not only powerful and customizable but also operate within the user's own environment, offering greater control and privacy. While challenges remain, particularly around the reliability of tool use with local models and the complexities of orchestration, the framework outlined in this report provides a solid foundation for creating next-generation local AI assistants integrated with the expanding world of MCP tools and resources.

#### Works cited

1. 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>
2. For Claude Desktop Users - Model Context Protocol, accessed April 8, 2025, <https://modelcontextprotocol.io/quickstart/user>
3. Using Ollama with Python: Step-by-Step Guide - Cohorte Projects, accessed April 8, 2025, <https://www.cohorte.co/blog/using-ollama-with-python-step-by-step-guide>
4. ollama/ollama-python: Ollama Python library - GitHub, accessed April 8, 2025, <https://github.com/ollama/ollama-python>
5. Model Context Protocol - GitHub, accessed April 8, 2025, <https://github.com/modelcontextprotocol>
6. modelcontextprotocol/python-sdk: The official Python SDK ... - GitHub, accessed April 8, 2025, <https://github.com/modelcontextprotocol/python-sdk>
7. 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>
8. 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>
9. An Introduction to Model Context Protocol - MCP 101 - DigitalOcean, accessed April 8, 2025, <https://www.digitalocean.com/community/tutorials/model-context-protocol>
10. Anthropic's Model Context Protocol (MCP) is way bigger than most people think - Reddit, accessed April 8, 2025, <https://www.reddit.com/r/ClaudeAI/comments/1gzv8b9/anthropics_model_context_protocol_mcp_is_way/>
11. Introducing the Model Context Protocol - Anthropic, accessed April 8, 2025, <https://www.anthropic.com/news/model-context-protocol>
12. Claude MCP - Model Context Protocol, accessed April 8, 2025, <https://www.claudemcp.com/>
13. Model Context Protocol (MCP) - Anthropic API, accessed April 8, 2025, <https://docs.anthropic.com/en/docs/agents-and-tools/mcp>
14. I gave Claude root access to my server... Model Context Protocol explained - YouTube, accessed April 8, 2025, <https://www.youtube.com/watch?v=HyzlYwjoXOQ>
15. Model Context Protocol (MCP) and OpenAI's Stance | by Frank Goortani - Medium, accessed April 8, 2025, <https://medium.com/@FrankGoortani/model-context-protocol-mcp-and-openais-stance-496ead1da740>
16. 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/>
17. 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>
18. Model Context Protocol specification, accessed April 8, 2025, <https://spec.modelcontextprotocol.io/specification/2025-03-26/>
19. Model Context Protocol (MCP) an overview - Philschmid, accessed April 8, 2025, <https://www.philschmid.de/mcp-introduction>
20. Model Context Protocol: Introduction, accessed April 8, 2025, <https://modelcontextprotocol.io/introduction>
21. The Model Context Protocol: Simplifying Building AI apps with Anthropic Claude Desktop and Docker, accessed April 8, 2025, <https://www.docker.com/blog/the-model-context-protocol-simplifying-building-ai-apps-with-anthropic-claude-desktop-and-docker/>
22. What Is the Model Context Protocol (MCP) and How It Works - Descope, accessed April 8, 2025, <https://www.descope.com/learn/post/mcp>
23. What is Model Context Protocol (MCP): Explained - Composio, accessed April 8, 2025, <https://composio.dev/blog/what-is-model-context-protocol-mcp-explained/>
24. 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>
25. To MCP or Not to MCP Part 1: A Critical Analysis of Anthropic's Model Context Protocol - Sanjeev Mohan, accessed April 8, 2025, <https://sanjmo.medium.com/to-mcp-or-not-to-mcp-part-1-a-critical-analysis-of-anthropics-model-context-protocol-571a51cb9f05>
26. A beginners Guide on Model Context Protocol (MCP) - OpenCV, accessed April 8, 2025, <https://opencv.org/blog/model-context-protocol/>
27. Model Context Protocol (MCP) :: Spring AI Reference, accessed April 8, 2025, <https://docs.spring.io/spring-ai/reference/api/mcp/mcp-overview.html>
28. Model Context Protocol - Cursor, accessed April 8, 2025, <https://docs.cursor.com/context/model-context-protocol>
29. 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>
30. 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>
31. What is MCP in AI? Model Context Protocol Simply Explained [No BS] - YouTube, accessed April 8, 2025, <https://www.youtube.com/watch?v=Xs9AwE2lyHg>
32. Introducing the Model Context Protocol : r/ClaudeAI - Reddit, accessed April 8, 2025, <https://www.reddit.com/r/ClaudeAI/comments/1gzpf81/introducing_the_model_context_protocol/>
33. modelcontextprotocol/servers: Model Context Protocol Servers - GitHub, accessed April 8, 2025, <https://github.com/modelcontextprotocol/servers>
34. 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>
35. Model Context Protocol (MCP): A comprehensive introduction for developers - Stytch, accessed April 8, 2025, <https://stytch.com/blog/model-context-protocol-introduction/>
36. Convert Ollama's Structured Output from JSON to Python DF - Stack Overflow, accessed April 8, 2025, <https://stackoverflow.com/questions/79295149/convert-ollamas-structured-output-from-json-to-python-df>
37. Implementing MCP server easily. Through the last article, I briefly… | by Terry Cho | Apr, 2025 | Medium, accessed April 8, 2025, <https://medium.com/@terrycho/implementing-mcp-server-easily-977299497448>
38. Connecting Letta to MCP Servers, accessed April 8, 2025, <https://docs.letta.com/guides/mcp/setup>
39. python-sdk/examples/servers/simple-tool/mcp\_simple\_tool/server.py at main - GitHub, accessed April 8, 2025, <https://github.com/modelcontextprotocol/python-sdk/blob/main/examples/servers/simple-tool/mcp_simple_tool/server.py>
40. Build Simple Local MCP Server - Medium, accessed April 8, 2025, <https://medium.com/@heshanhfernando/build-simple-local-mcp-server-5434d19572a4>
41. For Server Developers - Model Context Protocol, accessed April 8, 2025, <https://modelcontextprotocol.io/quickstart/server>
42. Model Context Protocol(MCP) with Google Gemini 2.5 Pro — A Deep Dive (Full Code), accessed April 8, 2025, <https://medium.com/google-cloud/model-context-protocol-mcp-with-google-gemini-llm-a-deep-dive-full-code-ea16e3fac9a3>
43. A Developer's Guide to the MCP - Zep, accessed April 8, 2025, <https://www.getzep.com/ai-agents/developer-guide-to-mcp>
44. MCP Servers - OpenAI Agents SDK, accessed April 8, 2025, <https://openai.github.io/openai-agents-python/ref/mcp/server/>
45. jlowin/fastmcp: The fast, Pythonic way to build Model Context Protocol servers - GitHub, accessed April 8, 2025, <https://github.com/jlowin/fastmcp>
46. 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>
47. The official Typescript SDK for Model Context Protocol servers and clients - GitHub, accessed April 8, 2025, <https://github.com/modelcontextprotocol/typescript-sdk>
48. Tools - Model Context Protocol, accessed April 8, 2025, <https://modelcontextprotocol.io/docs/concepts/tools>
49. Model Context Protocol vs Function Calling: What's the Big Difference? : r/ClaudeAI - Reddit, accessed April 8, 2025, <https://www.reddit.com/r/ClaudeAI/comments/1h0w1z6/model_context_protocol_vs_function_calling_whats/>
50. LLM Function-Calling vs. Model Context Protocol (MCP) - Gentoro, accessed April 8, 2025, <https://www.gentoro.com/blog/function-calling-vs-model-context-protocol-mcp>
51. Function Calling vs. Model Context Protocol (MCP): What You Need to Know, accessed April 8, 2025, <https://dev.to/fotiecodes/function-calling-vs-model-context-protocol-mcp-what-you-need-to-know-4nbo>
52. A quick look at tool use/function calling with Node.js and Ollama | Red Hat Developer, accessed April 8, 2025, <https://developers.redhat.com/blog/2024/09/10/quick-look-tool-usefunction-calling-nodejs-and-ollama>
53. An introduction to function calling and tool use - Apideck, accessed April 8, 2025, <https://www.apideck.com/blog/llm-tool-use-and-function-calling>
54. Building a Model Context Protocol (MCP) server for Discord - Speakeasy, accessed April 8, 2025, <https://www.speakeasy.com/post/build-a-mcp-server-tutorial>
55. Model context protocol (MCP) - OpenAI Agents SDK, accessed April 8, 2025, <https://openai.github.io/openai-agents-python/mcp/>
56. For Client Developers - Model Context Protocol, accessed April 8, 2025, <https://modelcontextprotocol.io/quickstart/client>
57. The Easiest Way to Understand Model Context Protocol (MCP) with code | by Dhananjay Kumar | Mar, 2025, accessed April 8, 2025, <https://dhnanjay.medium.com/the-easiest-way-to-understand-model-context-protocol-mcp-with-code-15fe3b302205?source=rss------ai-5>
58. Ollama API Usage Examples - GPU Mart, accessed April 8, 2025, <https://www.gpu-mart.com/blog/ollama-api-usage-examples>
59. Python & JavaScript Libraries · Ollama Blog, accessed April 8, 2025, <https://ollama.com/blog/python-javascript-libraries>
60. Function Calling with Ollama, Llama 3.1 and Milvus | by Zilliz - Medium, accessed April 8, 2025, <https://medium.com/@zilliz_learn/function-calling-with-ollama-llama-3-1-and-milvus-3fd268405ad7>
61. "Tool Calling" and Ollama - Philippe Charrière's Blog, accessed April 8, 2025, <https://k33g.hashnode.dev/tool-calling-and-ollama>
62. ollama/docs/api.md at main · ollama/ollama · GitHub, accessed April 8, 2025, <https://github.com/ollama/ollama/blob/main/docs/api.md>
63. Ollama now supports tool calling with popular models in local LLM | Hacker News, accessed April 8, 2025, <https://news.ycombinator.com/item?id=41291425>
64. How to do proper function calling on Ollama models - Reddit, accessed April 8, 2025, <https://www.reddit.com/r/ollama/comments/1ioyxkm/how_to_do_proper_function_calling_on_ollama_models/>
65. Getting json from an llama model : r/ollama - Reddit, accessed April 8, 2025, <https://www.reddit.com/r/ollama/comments/1gr8jj4/getting_json_from_an_llama_model/>
66. How To Use Ollama's API With Python - YouTube, accessed April 8, 2025, <https://www.youtube.com/watch?v=pLNqaTxvx3M>
67. Fully local tool calling with Ollama - YouTube, accessed April 8, 2025, <https://www.youtube.com/watch?v=Nfk99Fz8H9k&pp=0gcJCfcAhR29_xXO>
68. ollama-api - PyPI, accessed April 8, 2025, <https://pypi.org/project/ollama-api/>
69. Introducing computer use, a new Claude 3.5 Sonnet, and Claude 3.5 Haiku - Anthropic, accessed April 8, 2025, <https://www.anthropic.com/news/3-5-models-and-computer-use>
70. Easiest Local Function Calling using Ollama and Llama 3.1 [A-Z] - YouTube, accessed April 8, 2025, <https://www.youtube.com/watch?v=i0tsVzRbsNU>
71. Structured outputs · Ollama Blog, accessed April 8, 2025, <https://ollama.com/blog/structured-outputs>
72. How to use MCP tools with a PydanticAI Agent | by Finn Andersen | Mar, 2025 - Medium, accessed April 8, 2025, <https://medium.com/@finndersen/how-to-use-mcp-tools-with-a-pydanticai-agent-0d3a09c93a51>
73. MCP - Model Context Protocol - SDK - Python - YouTube, accessed April 8, 2025, <https://www.youtube.com/watch?v=oq3dkNm51qc>
74. MCP server: A step-by-step guide to building from scratch - Composio, accessed April 8, 2025, <https://composio.dev/blog/mcp-server-step-by-step-guide-to-building-from-scrtch/>
75. Zero to MCP Hero: Building Multi-Tool AI Agents in Python | by Timothy | Apr, 2025 | Medium, accessed April 8, 2025, <https://timtech4u.medium.com/zero-to-mcp-hero-building-multi-tool-ai-agents-in-python-gemini-c181fbb047b7>
76. Build a GPT-4o Powered MCP Client: Seamless Tool Integration - YouTube, accessed April 8, 2025, <https://www.youtube.com/watch?v=eZqzSeVBBDw>
77. mcp-python-sdk - User's blog, accessed April 8, 2025, <https://streetsdigital.com/2025/04/04/mcp-python-sdk/>
78. MCP Run Python - PydanticAI, accessed April 8, 2025, <https://ai.pydantic.dev/mcp/run-python/>
79. Releases · modelcontextprotocol/python-sdk - GitHub, accessed April 8, 2025, <https://github.com/modelcontextprotocol/python-sdk/releases>
80. An Introduction to Python Subprocess: Basics and Examples - DataCamp, accessed April 8, 2025, <https://www.datacamp.com/tutorial/python-subprocess>
81. How to use subprocess for system commands - LabEx, accessed April 8, 2025, <https://labex.io/tutorials/python-how-to-use-subprocess-for-system-commands-464366>
82. Python Subprocess: Master External Command Execution [2025] - Simplilearn.com, accessed April 8, 2025, <https://www.simplilearn.com/tutorials/python-tutorial/subprocess-in-python>
83. subprocess — Subprocess management — Python 3.13.2 documentation, accessed April 8, 2025, <https://docs.python.org/3/library/subprocess.html>
84. The subprocess Module: Wrapping Programs With Python, accessed April 8, 2025, <https://realpython.com/python-subprocess/>
85. Python Popen sending to process on stdin, receiving on stdout - Stack Overflow, accessed April 8, 2025, <https://stackoverflow.com/questions/15785148/python-popen-sending-to-process-on-stdin-receiving-on-stdout>
86. Understanding stdin, stdout, stderr in Python - YouTube, accessed April 8, 2025, <https://m.youtube.com/watch?v=4HY0VBBY7ok&pp=ygULI25ldXJhbG5pbmU%3D>
87. Claude, accessed April 8, 2025, <https://claude.ai/>
88. Unlocking the Power of Ollama's Structured JSON Output - YouTube, accessed April 8, 2025, <https://m.youtube.com/watch?v=nz5c9d1fJZ0>
89. 12 - Streamlit as an MCP Host + Ollama DeepSeek MCP Server - YouTube, accessed April 8, 2025, <https://www.youtube.com/watch?v=Q4KxVRDm3uo>
90. Building LLM Agents from Scratch (Part 2) : A Conversational Search Agent with Ollama, accessed April 8, 2025, <https://blog.gopenai.com/building-llm-agents-from-scratch-part-2-a-conversational-search-agent-with-ollama-a4544b2291cf>