**Chapter 10\_ Model Context Protocol (MCP)**

Chapter 10: Model Context Protocol

To enable LLMs to function effectively as agents, their capabilities must extend beyond multimodal generation. Interaction with the external environment is necessary, including access to current data, utilization of external software, and execution of specific operational tasks. The Model Context Protocol (MCP) addresses this need by providing a standardized interface for LLMs to interface with external resources. This protocol serves as a key mechanism to facilitate consistent and predictable integration.

**MCP Pattern Overview**

Imagine a universal adapter that allows any LLM to plug into any external system, database, or tool without a custom integration for each one. That's essentially what the Model Context Protocol (MCP) is. It's an open standard designed to standardize how LLMs like Gemini, OpenAI's GPT models, Mixtral, and Claude communicate with external applications, data sources, and tools. Think of it as a universal connection mechanism that simplifies how LLMs obtain context, execute actions, and interact with various systems.

MCP operates on a client-server architecture. It defines how different elements—data (referred to as resources), interactive templates (which are essentially prompts), and actionable functions (known as tools)—are exposed by an MCP server. These are then consumed by an MCP client, which could be an LLM host application or an AI agent itself. This standardized approach dramatically reduces the complexity of integrating LLMs into diverse operational environments.

However, MCP is a contract for an "agentic interface," and its effectiveness depends heavily on the design of the underlying APIs it exposes. There is a risk that developers simply wrap pre-existing, legacy APIs without modification, which can be suboptimal for an agent. For example, if a ticketing system's API only allows retrieving full ticket details one by one, an agent asked to summarize high-priority tickets will be slow and inaccurate at high volumes. To be truly effective, the underlying API should be improved with deterministic features like filtering and sorting to help the non-deterministic agent work efficiently. This highlights that agents do not magically replace deterministic workflows; they often require stronger deterministic support to succeed.

Furthermore, MCP can wrap an API whose input or output is still not inherently understandable by the agent. An API is only useful if its data format is agent-friendly, a guarantee that MCP itself does not enforce. For instance, creating an MCP server for a document store that returns files as PDFs is mostly useless if the consuming agent cannot parse PDF content. The better approach would be to first create an API that returns a textual version of the document, such as Markdown, which the agent can actually read and process. This demonstrates that developers must consider not just the connection, but the nature of the data being exchanged to ensure true compatibility.

**MCP vs. Tool Function Calling**

The Model Context Protocol (MCP) and tool function calling are distinct mechanisms that enable LLMs to interact with external capabilities (including tools) and execute actions. While both serve to extend LLM capabilities beyond text generation, they differ in their approach and level of abstraction.

Tool function calling can be thought of as a direct request from an LLM to a specific, pre-defined tool or function. Note that in this context we use the words "tool" and "function” interchangeably. This interaction is characterized by a one-to-one communication model, where the LLM formats a request based on its understanding of a user's intent requiring external action. The application code then executes this request and returns the result to the LLM. This process is often proprietary and varies across different LLM providers.

In contrast, the Model Context Protocol (MCP) operates as a standardized interface for LLMs to discover, communicate with, and utilize external capabilities. It functions as an open protocol that facilitates interaction with a wide range of tools and systems, aiming to establish an ecosystem where any compliant tool can be accessed by any compliant LLM. This fosters interoperability, composability and reusability across different systems and implementations. By adopting a federated model, we significantly improve interoperability and unlock the value of existing assets. This strategy allows us to bring disparate and legacy services into a modern ecosystem simply by wrapping them in an MCP-compliant interface. These services continue to operate independently, but can now be composed into new applications and workflows, with their collaboration orchestrated by LLMs. This fosters agility and reusability without requiring costly rewrites of foundational systems.

Here's a breakdown of the fundamental distinctions between MCP and tool function calling:

|  |  |  |
| --- | --- | --- |
| **Feature** | **Tool Function Calling** | **Model Context Protocol (MCP)** |
| **Standardization** | Proprietary and vendor-specific. The format and implementation differ across LLM providers. | An open, standardized protocol, promoting interoperability between different LLMs and tools. |
| **Scope** | A direct mechanism for an LLM to request the execution of a specific, predefined function. | A broader framework for how LLMs and external tools discover and communicate with each other. |
| **Architecture** | A one-to-one interaction between the LLM and the application's tool-handling logic. | A client-server architecture where LLM-powered applications (clients) can connect to and utilize various MCP servers (tools). |
| **Discovery** | The LLM is explicitly told which tools are available within the context of a specific conversation. | Enables dynamic discovery of available tools. An MCP client can query a server to see what capabilities it offers. |
| **Reusability** | Tool integrations are often tightly coupled with the specific application and LLM being used. | Promotes the development of reusable, standalone "MCP servers" that can be accessed by any compliant application. |

Think of tool function calling as giving an AI a specific set of custom-built tools, like a particular wrench and screwdriver. This is efficient for a workshop with a fixed set of tasks. MCP (Model Context Protocol), on the other hand, is like creating a universal, standardized power outlet system. It doesn't provide the tools itself, but it allows any compliant tool from any manufacturer to plug in and work, enabling a dynamic and ever-expanding workshop.

In short, function calling provides direct access to a few specific functions, while MCP is the standardized communication framework that lets LLMs discover and use a vast range of external resources. For simple applications, specific tools are enough; for complex, interconnected AI systems that need to adapt, a universal standard like MCP is essential.

**Additional considerations for MCP**

While MCP presents a powerful framework, a thorough evaluation requires considering several crucial aspects that influence its suitability for a given use case. Let's see some aspects in more details:

* **Tool vs. Resource vs. Prompt**: It's important to understand the specific roles of these components. A resource is static data (e.g., a PDF file, a database record). A tool is an executable function that performs an action (e.g., sending an email, querying an API). A prompt is a template that guides the LLM in how to interact with a resource or tool, ensuring the interaction is structured and effective.
* **Discoverability**: A key advantage of MCP is that an MCP client can dynamically query a server to learn what tools and resources it offers. This "just-in-time" discovery mechanism is powerful for agents that need to adapt to new capabilities without being redeployed.
* **Security**: Exposing tools and data via any protocol requires robust security measures. An MCP implementation must include authentication and authorization to control which clients can access which servers and what specific actions they are permitted to perform.
* **Implementation**: While MCP is an open standard, its implementation can be complex. However, providers are beginning to simplify this process. For example, some model providers like Anthropic or FastMCP offer SDKs that abstract away much of the boilerplate code, making it easier for developers to create and connect MCP clients and servers.
* **Error Handling**: A comprehensive error-handling strategy is critical. The protocol must define how errors (e.g., tool execution failure, unavailable server, invalid request) are communicated back to the LLM so it can understand the failure and potentially try an alternative approach.
* **Local vs. Remote Server**: MCP servers can be deployed locally on the same machine as the agent or remotely on a different server. A local server might be chosen for speed and security with sensitive data, while a remote server architecture allows for shared, scalable access to common tools across an organization.
* **On-demand vs. Batch**: MCP can support both on-demand, interactive sessions and larger-scale batch processing. The choice depends on the application, from a real-time conversational agent needing immediate tool access to a data analysis pipeline that processes records in batches.
* **Transportation Mechanism**: The protocol also defines the underlying transport layers for communication. For local interactions, it uses JSON-RPC over STDIO (standard input/output) for efficient inter-process communication. For remote connections, it leverages web-friendly protocols like Streamable HTTP and Server-Sent Events (SSE) to enable persistent and efficient client-server communication.

The Model Context Protocol uses a client-server model to standardize information flow. Understanding component interaction is key to MCP's advanced agentic behavior:

1. **Large Language Model (LLM)**: The core intelligence. It processes user requests, formulates plans, and decides when it needs to access external information or perform an action.
2. **MCP Client**: This is an application or wrapper around the LLM. It acts as the intermediary, translating the LLM's intent into a formal request that conforms to the MCP standard. It is responsible for discovering, connecting to, and communicating with MCP Servers.
3. **MCP Server**: This is the gateway to the external world. It exposes a set of tools, resources, and prompts to any authorized MCP Client. Each server is typically responsible for a specific domain, such as a connection to a company's internal database, an email service, or a public API.
4. ​​**Optional Third-Party (3P) Service:** This represents the actual external tool, application, or data source that the MCP Server manages and exposes. It is the ultimate endpoint that performs the requested action, such as querying a proprietary database, interacting with a SaaS platform, or calling a public weather API.

The interaction flows as follows:

1. **Discovery**: The MCP Client, on behalf of the LLM, queries an MCP Server to ask what capabilities it offers. The server responds with a manifest listing its available tools (e.g., send\_email), resources (e.g., customer\_database), and prompts.
2. **Request Formulation**: The LLM determines that it needs to use one of the discovered tools. For instance, it decides to send an email. It formulates a request, specifying the tool to use (send\_email) and the necessary parameters (recipient, subject, body).
3. **Client Communication**: The MCP Client takes the LLM's formulated request and sends it as a standardized call to the appropriate MCP Server.
4. **Server Execution**: The MCP Server receives the request. It authenticates the client, validates the request, and then executes the specified action by interfacing with the underlying software (e.g., calling the send() function of an email API).
5. **Response and Context Update**: After execution, the MCP Server sends a standardized response back to the MCP Client. This response indicates whether the action was successful and includes any relevant output (e.g., a confirmation ID for the sent email). The client then passes this result back to the LLM, updating its context and enabling it to proceed with the next step of its task.

**Practical Applications & Use Cases**

MCP significantly broadens AI/LLM capabilities, making them more versatile and powerful. Here are nine key use cases:

* **Database Integration:** MCP allows LLMs and agents to seamlessly access and interact with structured data in databases. For instance, using the MCP Toolbox for Databases, an agent can query Google BigQuery datasets to retrieve real-time information, generate reports, or update records, all driven by natural language commands.
* **Generative Media Orchestration:** MCP enables agents to integrate with advanced generative media services. Through MCP Tools for Genmedia Services, an agent can orchestrate workflows involving Google's Imagen for image generation, Google's Veo for video creation, Google's Chirp 3 HD for realistic voices, or Google's Lyria for music composition, allowing for dynamic content creation within AI applications.
* **External API Interaction:** MCP provides a standardized way for LLMs to call and receive responses from any external API. This means an agent can fetch live weather data, pull stock prices, send emails, or interact with CRM systems, extending its capabilities far beyond its core language model.
* **Reasoning-Based Information Extraction:** Leveraging an LLM's strong reasoning skills, MCP facilitates effective, query-dependent information extraction that surpasses conventional search and retrieval systems. Instead of a traditional search tool returning an entire document, an agent can analyze the text and extract the precise clause, figure, or statement that directly answers a user's complex question.
* **Custom Tool Development:** Developers can build custom tools and expose them via an MCP server (e.g., using FastMCP). This allows specialized internal functions or proprietary systems to be made available to LLMs and other agents in a standardized, easily consumable format, without needing to modify the LLM directly.
* **Standardized LLM-to-Application Communication:** MCP ensures a consistent communication layer between LLMs and the applications they interact with. This reduces integration overhead, promotes interoperability between different LLM providers and host applications, and simplifies the development of complex agentic systems.
* **Complex Workflow Orchestration:** By combining various MCP-exposed tools and data sources, agents can orchestrate highly complex, multi-step workflows. An agent could, for example, retrieve customer data from a database, generate a personalized marketing image, draft a tailored email, and then send it, all by interacting with different MCP services.
* **IoT Device Control:** MCP can facilitate LLM interaction with Internet of Things (IoT) devices. An agent could use MCP to send commands to smart home appliances, industrial sensors, or robotics, enabling natural language control and automation of physical systems.
* **Financial Services Automation:** In financial services, MCP could enable LLMs to interact with various financial data sources, trading platforms, or compliance systems. An agent might analyze market data, execute trades, generate personalized financial advice, or automate regulatory reporting, all while maintaining secure and standardized communication.

In short, the Model Context Protocol (MCP) enables agents to access real-time information from databases, APIs, and web resources. It also allows agents to perform actions like sending emails, updating records, controlling devices, and executing complex tasks by integrating and processing data from various sources. Additionally, MCP supports media generation tools for AI applications.

**Hands-On Code Example with ADK**

This section outlines how to connect to a local MCP server that provides file system operations, enabling an ADK agent to interact with the local file system.

**Agent Setup with MCPToolset**

To configure an agent for file system interaction, an `agent.py` file must be created (e.g., at `./adk\_agent\_samples/mcp\_agent/agent.py`). The `MCPToolset` is instantiated within the `tools` list of the `LlmAgent` object. It is crucial to replace `"/path/to/your/folder"` in the `args` list with the absolute path to a directory on the local system that the MCP server can access. This directory will be the root for the file system operations performed by the agent.

|  |
| --- |
| import os  from google.adk.agents import LlmAgent  from google.adk.tools.mcp\_tool.mcp\_toolset import MCPToolset, StdioServerParameters  # Create a reliable absolute path to a folder named 'mcp\_managed\_files'  # within the same directory as this agent script.  # This ensures the agent works out-of-the-box for demonstration.  # For production, you would point this to a more persistent and secure location.  TARGET\_FOLDER\_PATH = os.path.join(os.path.dirname(os.path.abspath(\_\_file\_\_)), "mcp\_managed\_files")  # Ensure the target directory exists before the agent needs it.  os.makedirs(TARGET\_FOLDER\_PATH, exist\_ok=True)  root\_agent = LlmAgent(  model='gemini-2.0-flash',  name='filesystem\_assistant\_agent',  instruction=(  'Help the user manage their files. You can list files, read files, and write files. '  f'You are operating in the following directory: {TARGET\_FOLDER\_PATH}'  ),  tools=[  MCPToolset(  connection\_params=StdioServerParameters(  command='npx',  args=[  "-y", # Argument for npx to auto-confirm install  "@modelcontextprotocol/server-filesystem",  # This MUST be an absolute path to a folder.  TARGET\_FOLDER\_PATH,  ],  ),  # Optional: You can filter which tools from the MCP server are exposed.  # For example, to only allow reading:  # tool\_filter=['list\_directory', 'read\_file']  )  ],  ) |

`npx` (Node Package Execute), bundled with npm (Node Package Manager) versions 5.2.0 and later, is a utility that enables direct execution of Node.js packages from the npm registry. This eliminates the need for global installation. In essence, `npx` serves as an npm package runner, and it is commonly used to run many community MCP servers, which are distributed as Node.js packages.

Creating an \_\_init\_\_.py file is necessary to ensure the agent.py file is recognized as part of a discoverable Python package for the Agent Development Kit (ADK). This file should reside in the same directory as [agent.py](http://agent.py).

|  |
| --- |
| # ./adk\_agent\_samples/mcp\_agent/\_\_init\_\_.py  from . import agent |

Certainly, other supported commands are available for use. For example, connecting to python3 can be achieved as follows:

|  |
| --- |
| connection\_params = StdioConnectionParams(  server\_params={  "command": "python3",  "args": ["./agent/mcp\_server.py"],  "env": {  "SERVICE\_ACCOUNT\_PATH":SERVICE\_ACCOUNT\_PATH,  "DRIVE\_FOLDER\_ID": DRIVE\_FOLDER\_ID  }  }  ) |

UVX, in the context of Python, refers to a command-line tool that utilizes uv to execute commands in a temporary, isolated Python environment. Essentially, it allows you to run Python tools and packages without needing to install them globally or within your project's environment. You can run it via the MCP server.

|  |
| --- |
| connection\_params = StdioConnectionParams(  server\_params={  "command": "uvx",  "args": ["mcp-google-sheets@latest"],  "env": {  "SERVICE\_ACCOUNT\_PATH":SERVICE\_ACCOUNT\_PATH,  "DRIVE\_FOLDER\_ID": DRIVE\_FOLDER\_ID  }  }  ) |

Once the MCP Server is created, the next step is to connect to it.

**Connecting the MCP Server with ADK Web**

To begin, execute 'adk web'. Navigate to the parent directory of mcp\_agent (e.g., adk\_agent\_samples) in your terminal and run:

|  |
| --- |
| cd ./adk\_agent\_samples # Or your equivalent parent directory  adk web |

Once the ADK Web UI has loaded in your browser, select the `filesystem\_assistant\_agent` from the agent menu. Next, experiment with prompts such as:

* "Show me the contents of this folder."
* "Read the `sample.txt` file." (This assumes `sample.txt` is located at `TARGET\_FOLDER\_PATH`.)
* "What's in `another\_file.md`?"

**Creating an MCP Server with FastMCP**

FastMCP is a high-level Python framework designed to streamline the development of MCP servers. It provides an abstraction layer that simplifies protocol complexities, allowing developers to focus on core logic.

The library enables rapid definition of tools, resources, and prompts using simple Python decorators. A significant advantage is its automatic schema generation, which intelligently interprets Python function signatures, type hints, and documentation strings to construct necessary AI model interface specifications. This automation minimizes manual configuration and reduces human error.

Beyond basic tool creation, FastMCP facilitates advanced architectural patterns like server composition and proxying. This enables modular development of complex, multi-component systems and seamless integration of existing services into an AI-accessible framework. Additionally, FastMCP includes optimizations for efficient, distributed, and scalable AI-driven applications.

**Server setup with FastMCP**

**To illustrate, consider a basic "greet" tool provided by the server. ADK agents and other MCP clients can interact with this tool using HTTP once it is active.**

|  |
| --- |
| # fastmcp\_server.py  # This script demonstrates how to create a simple MCP server using FastMCP.  # It exposes a single tool that generates a greeting.  # 1. Make sure you have FastMCP installed:  # pip install fastmcp  from fastmcp import FastMCP, Client  # Initialize the FastMCP server.  mcp\_server = FastMCP()  # Define a simple tool function.  # The `@mcp\_server.tool` decorator registers this Python function as an MCP tool.  # The docstring becomes the tool's description for the LLM.  @mcp\_server.tool  def greet(name: str) -> str:  """  Generates a personalized greeting.  Args:  name: The name of the person to greet.  Returns:  A greeting string.  """  return f"Hello, {name}! Nice to meet you."  # Or if you want to run it from the script:  if \_\_name\_\_ == "\_\_main\_\_":  mcp\_server.run(  transport="http",  host="127.0.0.1",  port=8000  ) |

This Python script defines a single function called greet, which takes a person's name and returns a personalized greeting. The @tool() decorator above this function automatically registers it as a tool that an AI or another program can use. The function's documentation string and type hints are used by FastMCP to tell the Agent how the tool works, what inputs it needs, and what it will return.

When the script is executed, it starts the FastMCP server, which listens for requests on localhost:8000. This makes the greet function available as a network service. An agent could then be configured to connect to this server and use the greet tool to generate greetings as part of a larger task. The server runs continuously until it is manually stopped.

**Consuming the FastMCP Server with an ADK Agent**

An ADK agent can be set up as an MCP client to use a running FastMCP server. This requires configuring HttpServerParameters with the FastMCP server's network address, which is usually http://localhost:8000.

A tool\_filter parameter can be included to restrict the agent's tool usage to specific tools offered by the server, such as 'greet'. When prompted with a request like "Greet John Doe," the agent's embedded LLM identifies the 'greet' tool available via MCP, invokes it with the argument "John Doe," and returns the server's response. This process demonstrates the integration of user-defined tools exposed through MCP with an ADK agent.

To establish this configuration, an agent file (e.g., agent.py located in ./adk\_agent\_samples/fastmcp\_client\_agent/) is required. This file will instantiate an ADK agent and use HttpServerParameters to establish a connection with the operational FastMCP server.

|  |
| --- |
| # ./adk\_agent\_samples/fastmcp\_client\_agent/agent.py  import os  from google.adk.agents import LlmAgent  from google.adk.tools.mcp\_tool.mcp\_toolset import MCPToolset, HttpServerParameters  # Define the FastMCP server's address.  # Make sure your fastmcp\_server.py (defined previously) is running on this port.  FASTMCP\_SERVER\_URL = "http://localhost:8000"  root\_agent = LlmAgent(  model='gemini-2.0-flash', # Or your preferred model  name='fastmcp\_greeter\_agent',  instruction='You are a friendly assistant that can greet people by their name. Use the "greet" tool.',  tools=[  MCPToolset(  connection\_params=HttpServerParameters(  url=FASTMCP\_SERVER\_URL,  ),  # Optional: Filter which tools from the MCP server are exposed  # For this example, we're expecting only 'greet'  tool\_filter=['greet']  )  ],  ) |

The script defines an Agent named fastmcp\_greeter\_agent that uses a Gemini language model. It's given a specific instruction to act as a friendly assistant whose purpose is to greet people. Crucially, the code equips this agent with a tool to perform its task. It configures an MCPToolset to connect to a separate server running on localhost:8000, which is expected to be the FastMCP server from the previous example. The agent is specifically granted access to the greet tool hosted on that server. In essence, this code sets up the client side of the system, creating an intelligent agent that understands its goal is to greet people and knows exactly which external tool to use to accomplish it.

Creating an \_\_init\_\_.py file within the fastmcp\_client\_agent directory is necessary. This ensures the agent is recognized as a discoverable Python package for the ADK.

To begin, open a new terminal and run `python fastmcp\_server.py` to start the FastMCP server. Next, go to the parent directory of `fastmcp\_client\_agent` (for example, `adk\_agent\_samples`) in your terminal and execute `adk web`. Once the ADK Web UI loads in your browser, select the `fastmcp\_greeter\_agent` from the agent menu. You can then test it by entering a prompt like "Greet John Doe." The agent will use the `greet` tool on your FastMCP server to create a response.

**At a Glance**

**What:** To function as effective agents, LLMs must move beyond simple text generation. They require the ability to interact with the external environment to access current data and utilize external software. Without a standardized communication method, each integration between an LLM and an external tool or data source becomes a custom, complex, and non-reusable effort. This ad-hoc approach hinders scalability and makes building complex, interconnected AI systems difficult and inefficient.

**Why:** The Model Context Protocol (MCP) offers a standardized solution by acting as a universal interface between LLMs and external systems. It establishes an open, standardized protocol that defines how external capabilities are discovered and used. Operating on a client-server model, MCP allows servers to expose tools, data resources, and interactive prompts to any compliant client. LLM-powered applications act as these clients, dynamically discovering and interacting with available resources in a predictable manner. This standardized approach fosters an ecosystem of interoperable and reusable components, dramatically simplifying the development of complex agentic workflows.

**Rule of thumb:** Use the Model Context Protocol (MCP) when building complex, scalable, or enterprise-grade agentic systems that need to interact with a diverse and evolving set of external tools, data sources, and APIs. It is ideal when interoperability between different LLMs and tools is a priority, and when agents require the ability to dynamically discover new capabilities without being redeployed. For simpler applications with a fixed and limited number of predefined functions, direct tool function calling may be sufficient.

|  |  |
| --- | --- |
| **Visual summary** |  |

Fig.1: Model Context protocol

**Key Takeaways**

These are the key takeaways:

* The Model Context Protocol (MCP) is an open standard facilitating standardized communication between LLMs and external applications, data sources, and tools.
* It employs a client-server architecture, defining the methods for exposing and consuming resources, prompts, and tools.
* The Agent Development Kit (ADK) supports both utilizing existing MCP servers and exposing ADK tools via an MCP server.
* FastMCP simplifies the development and management of MCP servers, particularly for exposing tools implemented in Python.
* MCP Tools for Genmedia Services allows agents to integrate with Google Cloud's generative media capabilities (Imagen, Veo, Chirp 3 HD, Lyria).
* MCP enables LLMs and agents to interact with real-world systems, access dynamic information, and perform actions beyond text generation.

**Conclusion**

The Model Context Protocol (MCP) is an open standard that facilitates communication between Large Language Models (LLMs) and external systems. It employs a client-server architecture, enabling LLMs to access resources, utilize prompts, and execute actions through standardized tools. MCP allows LLMs to interact with databases, manage generative media workflows, control IoT devices, and automate financial services. Practical examples demonstrate setting up agents to communicate with MCP servers, including filesystem servers and servers built with FastMCP, illustrating its integration with the Agent Development Kit (ADK). MCP is a key component for developing interactive AI agents that extend beyond basic language capabilities.

**References**

1. Model Context Protocol (MCP) Documentation. (Latest). *Model Context Protocol (MCP)*. <https://google.github.io/adk-docs/mcp/>
2. FastMCP Documentation. FastMCP.<https://github.com/jlowin/fastmcp>
3. MCP Tools for Genmedia Services. *MCP Tools for Genmedia Services*. <https://google.github.io/adk-docs/mcp/#mcp-servers-for-google-cloud-genmedia>
4. MCP Toolbox for Databases Documentation. (Latest). *MCP Toolbox for Databases*. <https://google.github.io/adk-docs/mcp/databases/>

**第10章\_模型上下文协议（MCP）**

第10章：模型上下文协议

为使大语言模型（LLMs）能作为智能体有效运行，其能力必须超越多模态生成。与外部环境的交互是必要的，包括访问最新数据、使用外部软件以及执行特定的操作任务。模型上下文协议（MCP）通过为大语言模型提供与外部资源对接的标准化接口来满足这一需求。该协议是促进一致且可预测集成的关键机制。

**MCP模式概述**

想象一下，有一个通用适配器，它能让任何大语言模型（LLM）连接到任何外部系统、数据库或工具，而无需为每个系统单独进行定制集成。这基本上就是模型上下文协议（MCP）的作用。它是一个开放标准，旨在规范像Gemini、OpenAI的GPT模型、Mixtral和Claude等大语言模型如何与外部应用程序、数据源和工具进行通信。可以将其视为一种通用连接机制，简化了大语言模型获取上下文、执行操作以及与各种系统交互的方式。

MCP采用客户端-服务器架构运行。它定义了MCP服务器如何公开不同的元素——数据（称为资源）、交互式模板（本质上是提示）和可操作功能（称为工具）。然后，这些元素由MCP客户端使用，MCP客户端可以是大语言模型（LLM）宿主应用程序或AI代理本身。这种标准化方法极大地降低了将LLM集成到各种操作环境中的复杂性。

然而，MCP是一个关于“智能体接口”的契约，其有效性在很大程度上取决于它所暴露的底层API的设计。存在这样一种风险，即开发者只是简单地封装现有的遗留API而不做修改，这对于智能体来说可能不是最优的。例如，如果票务系统的API只允许逐个检索完整的票务细节，那么被要求总结高优先级票务的智能体在处理大量票务时会变得缓慢且不准确。为了真正有效，底层API应该通过过滤和排序等确定性特性进行改进，以帮助非确定性的智能体高效工作。这凸显了智能体并不能神奇地取代确定流程； 它们往往需要更强有力的确定性支持才能取得成功。

此外，MCP可以封装一个其输入或输出对代理来说仍然不是天生就能理解的API。一个API只有在其数据格式对代理友好时才有用，而MCP本身并不强制保证这一点。例如，如果消费代理无法解析PDF内容，那么为返回PDF格式文件的文档存储创建一个MCP服务器基本上是无用的。更好的做法是首先创建一个返回文档文本版本（如Markdown）的API，这样代理才能真正读取和处理。这表明开发者不仅要考虑连接，还要考虑所交换数据的性质，以确保真正的兼容性。

**MCP与工具函数调用**

模型上下文协议（MCP）和工具函数调用是不同的机制，它们使大语言模型（LLMs）能够与外部功能（包括工具）进行交互并执行操作。虽然两者都旨在将大语言模型的能力扩展到文本生成之外，但它们在方法和抽象程度上有所不同。

工具函数调用可以被视为大语言模型（LLM）向特定的、预定义的工具或函数发出的直接请求。请注意，在本文中，我们会互换使用“工具”和“函数”这两个词。这种交互的特点是一对一的通信模式，即大语言模型根据对用户需要外部操作的意图的理解来格式化请求。然后应用程序代码执行此请求，并将结果返回给大语言模型。这个过程通常是专有的，并且因不同的大语言模型提供商而异。

相比之下，模型上下文协议（MCP）作为大语言模型（LLMs）发现、与外部能力通信并利用这些能力的标准化接口运行。它作为一种开放协议，促进与各种工具和系统的交互，旨在建立一个生态系统，使任何符合标准的工具都能被任何符合标准的大语言模型访问。这促进了不同系统和实现之间的互操作性、可组合性和可重用性。通过采用联合模型，我们显著提高了互操作性，并释放了现有资产的价值。这一策略使我们能够通过将不同的遗留服务封装在符合MCP的接口中，将它们纳入现代生态系统。这些服务继续独立运行，但现在可以组合成新的应用程序和工作流程，其协作由大语言模型进行编排。这促进了灵活性和可重用性，而无需对基础系统进行代价高昂的重写。

以下是MCP和工具函数调用之间基本区别的详细说明：

|  |  |  |
| --- | --- | --- |
| **功能** | **工具函数调用** | **模型上下文协议（MCP）** |
| **标准化** | 专有且特定于供应商。不同大语言模型（LLM）提供商的格式和实现方式各不相同。 | 一种开放的标准化协议，促进不同大语言模型和工具之间的互操作性。 |
| **范围** | 大语言模型请求执行特定预定义函数的直接机制。 | 一个关于大语言模型（LLMs）和外部工具如何相互发现和通信的更广泛框架。 |
| **架构** | 大语言模型（LLM）与应用程序的工具处理逻辑之间的一对一交互。 | 一种客户端-服务器架构，其中由大语言模型（LLM）驱动的应用程序（客户端）可以连接到并使用各种MCP服务器（工具）。 |
| **发现** | 大语言模型（LLM）会被明确告知在特定对话语境中可用的工具。 | 启用对可用工具的动态发现。MCP客户端可以查询服务器，以了解其提供的功能。 |
| **可复用性** | 工具集成通常与正在使用的特定应用程序和大语言模型紧密耦合。 | 促进可重用、独立的“MCP服务器”的开发，任何兼容的应用程序都可以访问这些服务器。 |

Think of tool function calling as giving an AI a specific set of custom-built tools, like a particular wrench and screwdriver. This is efficient for a workshop with a fixed set of tasks. MCP (Model Context Protocol), on the other hand, is like creating a universal, standardized power outlet system. It doesn't provide the tools itself, but it allows any compliant tool from any manufacturer to plug in and work, enabling a dynamic and ever-expanding workshop.

简而言之，函数调用提供对少数特定函数的直接访问，而MCP是标准化通信框架，它使大语言模型能够发现并使用大量外部资源。对于简单应用，特定工具就足够了；对于需要自适应的复杂、相互关联的AI系统，像MCP这样的通用标准是必不可少的。

**MCP的其他注意事项**

虽然MCP提供了一个强大的框架，但全面评估需要考虑几个关键方面，这些方面会影响其对特定用例的适用性。让我们更详细地了解一些方面：

* **工具、资源与提示词**：理解这些组件的具体作用非常重要。资源是静态数据（例如，PDF文件、数据库记录）。工具是执行操作的可执行函数（例如，发送电子邮件、查询API）。提示词是一种模板，用于指导大语言模型（LLM）如何与资源或工具进行交互，确保交互具有结构性和有效性。
* **可发现性**：MCP的一个关键优势在于，MCP客户端可以动态查询服务器，以了解其提供的工具和资源。这种“即时”发现机制对于需要在无需重新部署的情况下适应新功能的代理来说非常强大。
* **安全性**：通过任何协议公开工具和数据都需要强大的安全措施。MCP实现必须包括身份验证和授权，以控制哪些客户端可以访问哪些服务器，以及它们被允许执行哪些特定操作。
* **实施**：虽然MCP是一个开放标准，但其实施过程可能很复杂。不过，供应商们正开始简化这一过程。例如，像Anthropic或FastMCP这样的一些模型供应商提供软件开发工具包（SDK），这些工具包抽象掉了许多样板代码，使开发人员更容易创建和连接MCP客户端和服务器。
* **错误处理**：全面的错误处理策略至关重要。协议必须定义如何将错误（例如，工具执行失败、服务器不可用、无效请求）反馈给大语言模型（LLM），以便它能够理解失败原因并可能尝试替代方法。
* **本地服务器与远程服务器**：MCP服务器可以部署在与代理相同的本地机器上，也可以远程部署在不同的服务器上。出于速度和敏感数据安全的考虑，可能会选择本地服务器，而远程服务器架构则允许整个组织共享、可扩展地访问通用工具。
* **按需处理与批次处理作业**：MCP 既可以支持按需交互式会话，也可以支持大规模批次处理作业。选择哪种方式取决于应用场景，从需要即时工具访问的实时对话代理，到批量处理记录的数据分析管道，不一而足。
* **传输机制**：该协议还定义了用于通信的底层传输层。对于本地交互，它使用基于标准输入输出（STDIO）的JSON-RPC进行高效的进程间通信。对于远程连接，它利用诸如可流式HTTP和服务器发送事件（SSE）等网络友好协议，以实现持久且高效的客户端 - 服务器通信。

模型上下文协议（Model Context Protocol，MCP）采用客户端-服务器模型来规范信息流。理解组件交互是MCP先进代理行为的关键：

1. **大语言模型（LLM）**：核心智能。它处理用户请求、制定计划，并决定何时需要访问外部信息或执行操作。
2. **MCP客户端**：这是一个围绕大语言模型（LLM）的应用程序或包装器。它充当中间媒介，将大语言模型的意图转化为符合MCP标准的正式请求。它负责发现、连接MCP服务器并与之通信。
3. **MCP服务器**：这是通往外部世界的网关。它向任何获得授权的MCP客户端公开一组工具、资源和提示。每个服务器通常负责一个特定的领域，例如连接到公司的内部数据库、电子邮件服务或公共API。
4. **可选第三方（3P）服务：**这代表MCP服务器管理和公开的实际外部工具、应用程序或数据源。它是执行请作的最终端点，例如查询专有数据库、与SaaS平台交互或调用公共天气API。

交互流程如下：

1. **发现**：MCP客户端代表大语言模型（LLM）向MCP服务器查询其提供哪些功能。服务器以清单形式响应，列出其可用工具（如send\_email）、资源（如customer\_database）和提示信息。
2. **请求制定**：大语言模型（LLM）确定需要使用已发现的工具之一。例如，它决定发送一封电子邮件。它会制定一个请求，指定要使用的工具（send\_email）和必要的参数（收件人、主题、正文）。
3. **客户端通信**：MCP客户端接收大语言模型（LLM）生成的请求，并将其作为标准化调用发送到相应的MCP服务器。
4. **服务器执行**：MCP服务器接收请求。它对客户端进行身份验证，验证请求，然后通过与底层软件交互（例如，调用电子邮件API的send()函数）来执行指定的操作。
5. **响应与上下文更新**：执行后，MCP服务器会将标准化响应发送回MCP客户端。此响应表明操作是否成功，并包含任何相关输出（例如，已发送电子邮件的确认ID）。然后，客户端将此结果传递回大语言模型（LLM），更新其上下文并使其能够继续执行任务的下一步。

**实际应用与用例**

MCP显著拓宽了AI/大语言模型的能力，使其更加通用和强大。以下是九个关键用例：

* **数据库集成：** MCP允许大语言模型（LLMs）和智能体无缝访问数据库中的结构化数据并与之交互。例如，使用MCP数据库工具箱，智能体可以查询谷歌BigQuery数据集，以检索实时信息、生成报告或更新记录，所有这些操作均由自然语言命令驱动。
* **生成式媒体编排：** MCP使智能体能够与先进的生成式媒体服务集成。通过MCP生成式媒体服务工具，智能体可以编排涉及谷歌Imagen图像生成、谷歌Veo视频创作、谷歌Chirp 3 HD逼真语音或谷歌Lyria音乐创作的工作流程，从而在AI应用程序中实现动态内容生产。
* **外部 API 交互：** MCP 为大语言模型（LLMs）提供了一种标准化方式，使其能够调用任何外部 API 并接收响应。这意味着智能体可以获取实时天气数据、拉取股票价格、发送电子邮件或与 CRM 系统进行交互，从而将其能力扩展到远远超出其核心语言模型的范围。
* **基于推理的信息提取：**借助大语言模型（LLM）强大的推理能力，MCP能够实现有效的、依赖查询的信息提取，超越传统的搜索和检索系统。与传统搜索工具返回整个文档不同，智能体可以分析文本，提取直接回答用户复杂问题的精确条款、数据或陈述。
* **自定义工具开发：**开发人员可以构建自定义工具，并通过MCP服务器（例如，使用FastMCP）将其公开。这使得专门的内部功能或专有系统能够以标准化、易于使用的格式提供给大语言模型和其他智能体，而无需直接修改大语言模型。
* **标准化大语言模型到应用程序的通信：** MCP确保大语言模型与其交互的应用程序之间有一个一致的通信层。这减少了集成开销，促进了不同大语言模型提供商和宿主应用程序之间的互操作性，并简化了复杂智能体系统的开发。
* **复杂工作流编排：**通过整合各种MCP公开的工具和数据源，智能体可以编排高度复杂的多步骤工作流。例如，智能体可以通过与不同的MCP服务交互，从数据库中检索客户数据、生成个性化营销图像、起草定制化电子邮件，然后发送出去。
* **物联网设备控制：** MCP可以促进大语言模型（LLM）与物联网（IoT）设备的交互。智能体可以使用MCP向智能家居设备、工业传感器或机器人发送命令，从而实现对物理系统的自然语言控制和自动化。
* **金融服务自动化：**在金融服务领域，MCP可以使大语言模型（LLMs）与各种金融数据源、交易平台或合规系统进行交互。智能体可以分析市场数据、执行交易、生成个性化金融建议或自动进行监管报告，同时保持安全和标准化的通信。

简而言之，模型上下文协议（MCP）使智能体能够从数据库、API和网络资源中访问实时信息。它还允许智能体通过集成和处理来自各种来源的数据来执行诸如发送电子邮件、更新记录、控制设备和执行复杂任务等操作。此外，MCP支持用于AI应用的媒体生成工具。

**使用ADK的实践代码示例**

本节概述了如何连接到提供文件系统操作的本地MCP服务器，从而使ADK代理能够与本地文件系统进行交互。

**使用MCP工具集进行代理设置**

若要配置用于文件系统交互的代理，必须创建一个 `agent.py` 文件（例如，位于 `./adk\_agent\_samples/mcp\_agent/agent.py`）。`MCPToolset` 在 `LlmAgent` 对象的 `tools` 列表中实例化。务必将 `args` 列表中的 `"/path/to/your/folder"` 替换为 MCP 服务器可以访问的本地系统目录的绝对路径。该目录将作为代理执行文件系统操作的根目录。

|  |
| --- |
| 导入 os  from google.adk.agents import LlmAgent  从google.adk.tools.mcp\_tool.mcp\_toolset导入MCPToolset、StdioServerParameters  # 创建一个指向名为'mcp\_managed\_files'文件夹的可靠绝对路径  # 与该代理脚本位于同一目录中。  # 这确保了代理可以开箱即用，用于演示。  # 在生产环境中，你需要将其指向一个更持久、更安全的位置。  TARGET\_FOLDER\_PATH = os.path.join(os.path.dirname(os.path.abspath(\_\_file\_\_)), "mcp\_managed\_files")  # 在代理需要目标目录之前，确保该目录存在。  os.makedirs(TARGET\_FOLDER\_PATH, exist\_ok=True)  root\_agent = LlmAgent(  model='gemini-2.0-flash',  name='filesystem\_assistant\_agent',  instruction=(  帮助用户管理他们的文件。你可以列出文件、读取文件和写入文件。  f'您正在以下目录中操作：{TARGET\_FOLDER\_PATH}'  ),  tools=[  MCP工具集(  connection\_params=StdioServerParameters(  command='npx',  args=[  "-y", # npx自动确认安装的参数  "@modelcontextprotocol/server-filesystem",  # 这必须是一个文件夹的绝对路径。  TARGET\_FOLDER\_PATH,  ],  ),  # 可选：您可以筛选出MCP服务器中公开的工具。  # 例如，仅允许读取：  # tool\_filter=['list\_directory', 'read\_file']  )  ],  ) |

`npx`（Node Package Execute，节点包执行）是一个实用工具，它与npm（Node Package Manager，节点包管理器）5.2.0及更高版本捆绑在一起，可直接从npm注册表执行Node.js包。这就无需进行全局安装。本质上，`npx`是一个npm包运行器，常用于运行许多以Node.js包形式分发的社区MCP服务器。

创建一个 \_\_init\_\_.py 文件是必要的，以确保 agent.py 文件被识别为 Agent 开发工具包 (ADK) 可发现的 Python 包的一部分。这个文件应该与[agent.py](http://agent.py)位于同一目录中。

|  |
| --- |
| #./adk\_agent\_samples/mcp\_agent/\_\_init\_\_.py  from. import agent |

当然，还有其他受支持的命令可供使用。例如，连接到 Python3 可以按如下方式实现：

|  |
| --- |
| connection\_params = StdioConnectionParams(  server\_params={  "command": "python3",  "args": ["./agent/mcp\_server.py"],  "env": {  "SERVICE\_ACCOUNT\_PATH":SERVICE\_ACCOUNT\_PATH,  "DRIVE\_FOLDER\_ID": DRIVE\_FOLDER\_ID  }  }  ) |

在Python的语境中，UVX是指一种命令行工具，它利用UV在临时、隔离的Python环境中执行命令。本质上，它允许你运行Python工具和包，而无需在全局或项目环境中安装它们。你可以通过MCP服务器运行它。

|  |
| --- |
| connection\_params = StdioConnectionParams(  server\_params={  "command": "uvx",  "args": ["mcp-google-sheets@latest"],  "env": {  "SERVICE\_ACCOUNT\_PATH":SERVICE\_ACCOUNT\_PATH,  "DRIVE\_FOLDER\_ID": DRIVE\_FOLDER\_ID  }  }  ) |

创建MCP服务器后，下一步是连接到它。

**将MCP服务器与ADK Web连接**

首先，执行 'adk web'。在终端中导航到 mcp\_agent 的父目录（例如 adk\_agent\_samples）并运行：

|  |
| --- |
| cd./adk\_agent\_samples # 或你的等效父目录  adk网络 |

在您的浏览器中加载 ADK Web UI 后，从代理菜单中选择 `filesystem\_assistant\_agent`。接下来，尝试使用以下提示：

* "给我展示这个文件夹的内容。"
* 读取 `sample.txt` 文件。（这假定 `sample.txt` 位于 `TARGET\_FOLDER\_PATH`。）
* "`another\_file.md`里有什么？"

**使用FastMCP创建MCP服务器**

FastMCP是一个高级Python框架，旨在简化MCP服务器的开发。它提供了一个抽象层，简化了协议的复杂性，使开发人员能够专注于核心逻辑。

该库允许使用简单的Python装饰器快速定义工具、资源和提示。一个显著的优势是其自动模式生成功能，它能智能地解释Python函数签名、类型提示和留档字符串，以构建必要的AI模型接口规范。这种自动化最大限度地减少了手动配置，降低了人为错误。

除了基础工具创建之外，FastMCP还支持诸如服务器组合和代理等高级架构模式。这使得复杂的多组件系统能够进行模块化开发，并将现有服务无缝集成到一个AI可访问的框架中。此外，FastMCP还针对高效、分布式和可扩展的AI驱动应用进行了优化。

**使用FastMCP进行服务器设置**

**为了说明这一点，我们来考虑一下服务器提供的一个基本的“问候”工具。一旦该工具处于活动状态，ADK代理和其他MCP客户端就可以使用HTTP与之进行交互。**

|  |
| --- |
| # fastmcp\_server.py  # 本脚本演示了如何使用 FastMCP 创建一个简单的 MCP 服务器。  # 它提供了一个生成问候语的工具。  # 1. 确保你已经安装了 FastMCP：  # pip install fastmcp  从 fastmcp 导入 FastMCP、Client  # 初始化 FastMCP 服务器。  mcp\_server = FastMCP()  # 定义一个简单的工具函数。  # `@mcp\_server.tool` 装饰器将此 Python 函数注册为 MCP 工具。  # 文档字符串成为大语言模型（LLM）的工具描述。  @mcp\_server.tool  def greet(name: str) -> str:  """  生成个性化问候语。  参数：  name: 要问候的人的名字。  返回值：  问候语字符串。  """  return f"你好，{name}！很高兴见到你。"  # 或者如果你想从脚本中运行它：  if \_\_name\_\_ == "\_\_main\_\_":  mcp\_server.run(  transport="http",  host="127.0.0.1",  port=8000  ) |

这个Python脚本定义了一个名为greet的函数，该函数接受一个人的名字并返回个性化问候语。此函数上方的@tool()装饰器会自动将其注册为AI或其他程序可以使用的工具。FastMCP使用该函数的留档字符串和类型提示来告知代理该工具的工作方式、所需输入以及将返回的内容。

当脚本执行时，它会启动FastMCP服务器，该服务器在localhost:8000上监听请求。这使得greet函数可以作为网络服务使用。然后可以配置一个代理连接到该服务器，并使用greet工具生成问候语，作为更大任务的一部分。服务器会持续运行，直到手动停止。

**使用ADK代理使用FastMCP服务器**

ADK代理可以设置为MCP客户端，以使用正在运行的FastMCP服务器。这需要使用FastMCP服务器的网络地址配置HttpServerParameters，该地址通常为http://localhost:8000。

可以包含一个tool\_filter参数，以将代理的工具使用限制在服务器提供的特定工具上，例如'greet'。当收到像“Greet John Doe”这样的请求时，代理中嵌入的大语言模型（LLM）会识别出可通过MCP获取的'greet'工具，使用参数“John Doe”调用该工具，并返回服务器的响应。这个过程展示了通过MCP公开的用户定义工具与ADK代理的集成。

要建立此配置，需要一个代理文件（例如，位于./adk\_agent\_samples/fastmcp\_client\_agent/ 中的 agent.py）。该文件将实例化一个 ADK 代理，并使用 HttpServerParameters 与运行中的 FastMCP 服务器建立连接。

|  |
| --- |
| #./adk\_agent\_samples/fastmcp\_client\_agent/agent.py  导入 os  from google.adk.agents import LlmAgent  从google.adk.tools.mcp\_tool.mcp\_toolset导入MCPToolset、HttpServerParameters  # 定义FastMCP服务器的地址。  # 确保你之前定义的 fastmcp\_server.py 在这个端口上运行。  FASTMCP\_SERVER\_URL = "http://localhost:8000"  root\_agent = LlmAgent(  model='gemini-2.0-flash', # 或者你喜欢的模型  name='fastmcp\_greeter\_agent',  instruction='你是一个友好的助手，可以根据人们的名字向他们打招呼。使用 "greet" 工具。',  tools=[  MCP工具集(  connection\_params=HttpServerParameters(  url=FASTMCP\_SERVER\_URL,  ),  # 可选：筛选MCP服务器中公开的工具  # 对于这个示例，我们只期望 'greet'  tool\_filter=['问候']  )  ],  ) |

该脚本定义了一个名为 fastmcp\_greeter\_agent 的智能体，它使用 Gemini 语言模型。该智能体被赋予了一项特定指令，即充当友好的助手，其目的是向人们打招呼。至关重要的是，代码为该智能体配备了执行任务的工具。它配置了一个 MCPToolset，以连接到在本地主机端口 8000 上运行的独立服务器，预计该服务器就是上一个示例中的 FastMCP 服务器。该智能体被特别授予访问托管在该服务器上的打招呼工具的权限。本质上，这段代码设置了系统的客户端，创建了一个智能体，它明白自己的目标是向人们打招呼，并且确切知道要使用哪个外部工具来实现这一目标。

在 fastmcp\_client\_agent 目录中创建一个 \_\_init\_\_.py 文件是必要的。这可确保该代理被识别为 ADK 可发现的 Python 包。

首先，打开一个新的终端，运行 `python fastmcp\_server.py` 以启动 FastMCP 服务器。接下来，在终端中进入 `fastmcp\_client\_agent` 的父目录（例如 `adk\_agent\_samples`），并执行 `adk web`。一旦 ADK Web UI 在你的浏览器中加载完成，从代理菜单中选择 `fastmcp\_greeter\_agent`。然后，你可以通过输入类似 “向约翰·多伊问好” 这样的提示来测试它。代理将使用你的 FastMCP 服务器上的 `greet` 工具来生成响应。

**概览**

**内容：**要成为有效的智能体，大语言模型（LLMs）必须超越简单的文本生成。它们需要具备与外部环境交互的能力，以便获取最新数据并利用外部软件。如果没有标准化的通信方法，大语言模型与外部工具或数据源之间的每次集成都会成为定制的、复杂且不可复用的工作。这种临时的方法阻碍了可扩展性，使得构建复杂的、相互关联的AI系统变得困难且低效。

**原因：**模型上下文协议（MCP）通过充当大语言模型（LLMs）与外部系统之间的通用接口，提供了一种标准化的解决方案。它建立了一个开放的、标准化的协议，该协议定义了如何发现和使用外部功能。MCP基于客户端 - 服务器模型运行，允许服务器向任何兼容的客户端公开工具、数据资源和交互式提示。由大语言模型驱动的应用程序充当这些客户端，以可预测的方式动态发现并与可用资源进行交互。这种标准化方法促进了一个由可互操作和可重用组件组成的生态系统，极大地简化了复杂智能体工作流的开发。

**经验法则：**在构建需要与各种不断发展的外部工具、数据源和 API 进行交互的复杂、可扩展或企业级智能体系统时，请使用模型上下文协议（MCP）。当不同大语言模型（LLM）和工具之间的互操作性是优先考虑的事项，且智能体需要在不重新部署的情况下动态发现新功能的能力时，MCP 是理想的选择。对于具有固定且有限数量预定义功能的简单应用程序，直接调用工具函数可能就足够了。

|  |  |
| --- | --- |
| **可视化总结** |  |

图1：模型上下文协议

**关键收获**

以下是关键收获：

* 模型上下文协议（MCP）是一项开放标准，旨在促进大语言模型（LLMs）与外部应用程序、数据源和工具之间的标准化通信。
* 它采用客户端-服务器架构，定义了公开和使用资源、提示和工具的方法。
* 代理开发套件（ADK）既支持使用现有的MCP服务器，也支持通过MCP服务器公开ADK工具。
* FastMCP简化了MCP服务器的开发和管理，特别是在暴露用Python实现的工具方面。
* Genmedia Services的MCP工具使代理能够与谷歌云的生成式媒体功能（Imagen、Veo、Chirp 3 HD、Lyria）集成。
* MCP使大语言模型和智能体能够与现实世界系统进行交互，访问动态信息，并执行文本生成之外的操作。

**结论**

模型上下文协议（MCP）是一种开放标准，它促进大语言模型（LLMs）与外部系统之间的通信。它采用客户端 - 服务器架构，使大语言模型能够通过标准化工具访问资源、利用提示并执行操作。MCP允许大语言模型与数据库交互、管理生成式媒体工作流程、控制物联网设备以及自动化金融服务。实际示例展示了如何设置代理与MCP服务器进行通信，包括文件系统服务器和使用FastMCP构建的服务器，说明了其与代理开发工具包（ADK）的集成。MCP是开发超越基本语言能力的交互式AI代理的关键组件。

**参考文献**

1. 模型上下文协议（MCP）文档。（最新）*模型上下文协议（MCP）*。<https://google.github.io/adk-docs/mcp/>
2. FastMCP文档。FastMCP。<https://github.com/jlowin/fastmcp>
3. 适用于Genmedia服务的MCP工具。*适用于Genmedia服务的MCP工具*。<https://google.github.io/adk-docs/mcp/#mcp-servers-for-google-cloud-genmedia>
4. 数据库MCP工具箱文档。（最新）*数据库MCP工具箱*。<https://google.github.io/adk-docs/mcp/databases/>