O'REILLY®

AI Agents with MCP

Model Context Protocol for Building Clients, Services, and End-to-End Agents

Kyle Stratis

AI Agents with MCP

Model Context Protocol for Building Clients, Services, and End-to-End Agents

Kyle Stratis

AI Agents with MCP

by Kyle Stratis

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (http://oreilly.com). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editors: Jill Leonard and Nicole Butterfield

Interior Designer: David Futato

Cover Designer: Karen Montgomery

Illustrator: Kate Dullea

May 2026: First Edition

Revision History for the Early Release

- 2025-07-07: First Release

See http://oreilly.com/catalog/errata.csp?isbn=9798341639553 for release details.

979-8-341-63950-8

Brief Table of Contents (*Not Yet Final*)

Chapter 1: Agentic AI and MCP (available)

Chapter 2: An Introduction to the Model Context Protocol (unavailable)

Chapter 3: Hosting Clients (available)

Chapter 3: Identify a Clear and Present Problem (unavailable)

Chapter 4: The Server (unavailable)

Chapter 5: Transports (unavailable)

Chapter 6: Host Application (unavailable)

Chapter 7: MCP Clients (unavailable)

Chapter 8: MCP Servers (unavailable)

Chapter 9: Putting It All Together (unavailable)

Chapter 1. Agentic AI and MCP

Towards the end of 2024, Anthropic released the Model Context Protocol (MCP) to great fanfare. Not only did Anthropic standardize a way to transform chatbots from simple talkers into agents that could act, but they released MCP concurrently with support from popular AI-powered IDEs as well as Anthropic's Claude Desktop, driving rapid adoption. One of the major pieces of the MCP architecture, the MCP servers, was made to be incredibly simple to spin up, but powerfully complex when advanced workflows were needed. This lead to a rapid flood of user-created MCP servers, augmenting generative AI workflows and cementing the adoption of MCP as *the* standard for building AI agents.

This book aims to go deep into MCP, taking you beyond the available documentation and giving you a deep understanding of the MCP architecture, the features the protocol and its SDKs provide, and most importantly, how to use that understanding to build your own MCP servers and clients and augment your own generative AI workflows with them. By doing this, you can transform standard generative AI chatbots into fully fledged agents, which can act, plan, and use tools on their own, and even work together to accomplish complex tasks. In this chapter, you will learn more about AI agents, what they are, how they are used, and how MCP fits into the agentic AI landscape. The next chapter will cover MCP itself: its history, the problems it solves, its architecture, and real-world examples of MCP in action. The next three chapters will cover the major components of MCP: clients and their host applications, servers, and transports. With this knowledge in hand, the final three chapters will use a project in order to give you practical experience building not just your own servers, cliens, and transports, but also how to leverage them in an actual agent.

What is an Agent, Anyway?

But what exactly *is* an agent? If you Google search this, you will find more definitions than you will find links. If you ask Claude or ChatGPT a few times in a row, you might even get different answers as well. Everyone seems to have an opinion on what is or isn't an agent, but it is difficult to talk about agents without settling on a common definition. For the purposes of this book, we will use the definition used by Anthropic in their article Building Effective Agents: an *agent* is a "system…where large language models (LLMs) dynamically direct their own processes and tool usage, maintaining control over how they accomplish tasks." They're usually implemented as LLMs that receive a task and then use tools (here, a *tool* is any code "outside" of the agent that is called by the agent to accomplish some task) to execute an action and gather feedback before proceeding. The action and feedback gathering occurs in a loop, where the LLM does some action, feedback is gathered and sent back to the LLM, and then the LLM decides whether to present the result as a final result to the user, use the feedback it gathered to do another action, or gather more feedback from the user.

Let's take a look at a common use for agents, and what this loop would look like for each outcome. For this example, let's imagine a coding task: you are in an AI-powered code editor like Cursor or Windsurf and you ask the chatbot to write unit tests for your code. The action/feedback loops may look something like this:

(Figure coming soon)

This is distinct from an *agentic workflow*, a system in which code, rather than the LLM, determines the path the logic will take, but still calls an LLM to do some part of the work. This is a common pattern in agentic systems, and can be very useful for complex tasks that require some level of deterministic logic as enforced by the code. For example, imagine an application that takes code as an input and turns it into its equivalent in another language. A workflow approach to this could be to take the input code, call an LLM to detect the language the code is in, and if it doesn't detect a language, return a failure message to the user. If it does detect a language and it is one that you have a deterministic parser for, then the deterministic parser is called on the code, and the translated output is returned to the user. If a language is detected that you don't have a parser for, then you would call an LLM again and prompt it to do the translation for you, returning the output to the user. This is *prompt chaining*, a common agentic workflow pattern for tasks that require one or more calls to an LLM and a code path for deciding which next steps to take to complete that task. Some other common workflow patterns include *paralellization*, where several LLM calls are made at once and the results are combined, *routing*, where an LLM classifies an input then makes one of several possible LLM calls with the classified input, *orchestrator-worker*, where one LLM breaks down a task, calls worker LLMs to work on the smaller tasks, and finally an LLM is called to put the results together, and *evaluator-optimizer*, where an LLM generates a response to the user prompt and a second LLM call evaluates that response and either presents it to the user if it's acceptable or returns it with feeback to the first LLM conversation for refinement.

The figure below shows the prompt chaining workflow in action: the input is sent to the LLM to detect the language of the input. The output is the guessed language, which is checked in code to see if there was a detection or not. It branches to the "fail" state if a language was not detected, and to the "success" state if one was. The next state is also deterministic, and checks if we have a deterministic parser for the detected language. If so, the next state is simply calling the parser and reporting the output to the user. If not, the next state is to call an LLM again and prompt it to do the translation for you, returning the output to the user.

(Figure coming soon)

By contrast, an agentic approach, however, would provide language detection and translation tools to the LLM, and simply ask it to detect the language of the input and use that to pick the correct translation tool to do the translation. The next figure shows this process: The user provides code input, and the LLM is called. It loops through the action-feedback loop, detecting the language of the input, then checking the environment for any tools to do the translation. If there is an appropriate tool for the agent to use, it calls it, if not, it will autonomously attempt the translation on its own. Whatever result it gets back, it will return to the user.

(Figure coming soon)

The chief difference to remember is that agents are able to decide which tools to use and actions to take over 1 or more action-feedback loops, while workflows use code paths to determine which tools to use and actions to take, and how to take those actions.

How are Agents Used in Generative AI?

As you saw above, agents are deployed for complex tasks where planning, decision-making, and tool use are necessary to complete the given task. Typically, to the user, there isn't any obvious difference in how they interact with an agent than how they interact with non-agentic LLMs: there's a chat box with a chatbot at the "other end" of it, and they ask it questions, and it answers. One key difference with agents, though, is that users can ask them to *do* something, and they can take action and interact with their environment given tools. A user might experience this as nothing more than a more powerful chatbot, but agents can provide customized experiences to the user via their usage of tools, specialized prompts, data resources, memory, and the ability to coordinate and collaborate with other agents.

These uncanny abilities have lead to great interest in investigating how to make agents work together.

(More coming soon)

What Agents Enable in Generative AI

Agents and the design patterns that are being invented and explored for them are enabling some of the most exciting new advances in generative AI. A single agent transforms an LLM with a static source of information (from its training data) into something that can access new information, take action on its environment via tools, act semi-autonomously, and communicate with other agents. All of this behavior comes from both the architecture of the agent (specifically, the action-feedback loop discussed earlier in this chapter), and the abilities that are added on to an LLM in an agentic system: data retrieval, tools, and memory.

(More coming soon)

Examples and Use Cases for Agents

To really explore how agents are used, what they allow us to do, and what kinds of tasks they excel at, it will be helpful to look at some use cases of agents, real-life examples of those use cases, and, where possible, their architectures. Agentic systems are being deployed seemingly everywhere, but a few use cases have really shined recently:

- Coding agents
- Research agents
- Customer service agents
- Business-specific copilots/assistants

Coding agents.

Research agents.

Customer service agents.

Business-specific copilots/assistants.

The Model Context Protocol's Role in Agentic AI

Before MCP, even though agentic AI was still fairly new, organizations that were building internal generative AI platforms and public frameworks like LangChain had to come up with different ways for users to implement tools and then pass them to an LLM to use, with different LLMs requiring their own connector. This lead to what is known as the "MxN problem": for each of M LLMs that needs to be supported, you need N connectors for each tool, data source, etc. that you want to connect. If you support 3 models and 10 tools, that means you have to write 30 connectors that largely repeat themselves, increasing the opportunities for hard-to-diagnose bugs to pop up on.

(Figure coming soon)

The Model Context Protocol, inspired by the Language Server Protocol (LSP), provides a common interface for building tools, prompts, and data resources and connecting them to LLMs. In this way, for any number M of LLMs, you just write N connectors, transforming the M x N problem into M + N.

(Figure coming soon)

(More coming soon)

Other Protocols in Agentic AI

(More coming soon)

What You’ll Learn

The aim of this book is twofold. The primary purpose is to get you comfortable with every feature and component of the Model Context Protocol, enabling you to build host applications (i.e. agents, given our definition from the beginning of the chapter), clients, servers, and even contribute to the protocol and its various SDKs. The first part of this book, chapters 2 through 5, accomplishes this via explanation and example. We’ll examine every component of MCP and the official Python SDK, discuss what they’re used for, what they can do, and how the implementation relates back to the protocol. As you progress through these components, you will encounter code examples designed to show a practical implementation with minimal cruft, allowing you to focus on the topic at hand. The second part, chapters 6-9, guides you through building a project that uses every part of the MCP architecture. This will give you real, hands-on experience in building applications that can interface with MCP servers via MCP clients, MCP servers, and even transport layers.

The secondary purpose of this book is to serve as a reference guide. The sections of the following chapters dealing with the architecture and Python implementation of the Model Context Protocol include links to code, the protocol specification, and articles and projects that address or otherwise use the protocol. These will serve as starting points whenever you need to learn more about a concept addressed in the book or where that concept fits into the current (and fast-moving) state of the protocol. In addition, I hope both the code examples and project can serve as reference points that can get you started quickly on your own projects.

Chapter 2. Hosting Clients

When working with the Model Context Protocol, you will typically be working with (or building!) clients or servers. On some projects, you may be able to build both. To really understand the protocol and use it to its full potential on your projects, you will need to be conversant with all components of the MCP architecture. In this chapter, you will learn about the the consumer side of this architecture: host applications and clients.

First, we will dig in to host applications: what are they, what do they do, and what kinds of applications can benefit from incorporating MCP? Then, we will examine the client itself. This is what makes the host application a host: it hosts the client(s) which in turn enable the host application to communicate with MCP servers.

There is one catch, though: a single client can only talk to a single server. If you want to access multiple servers, you will have to launch multiple client instances or have a single client that only makes short, temporary connections to servers. We will examine the pros and cons of each approach, and which use cases each approach best serves.

Then, you will see a very simple host application and the client that it hosts. We will go line-by-line to help you understand the structure of a client and how it provides functionality to the host application. Finally, you will learn about best practices when it comes to building a client into a host application, ensuring that your application remains secure, reliable, and responsive.

- ☐ FIGURE TBC

The Host Application

Your application can be anything that hosts the client(s) that will manage connections to MCP servers, from a simple chatbot script that takes some user input and sends it to an LLM for a response to a fully-featured IDE like Cursor or Windsurf.

Thanks to the modularity of MCP, there are very few restrictions on what your host application does in order to support MCP. It only needs to communicate with an LLM and host one or more MCP clients. If you plan to use tools, then the LLM your host application uses should also support tool calling.

In the example below, you will see a barebones host application. All it does in its current state is take user input in a constant loop and pass it on to an LLM. You can find this script and all other code for this chapter in `ch3` in the book's Github repo.

Example: A Simple Host Application

```
import os

from anthropic import Anthropic
from dotenv import load_dotenv

load_dotenv()

LLM_API_KEY = os.environ["LLM_API_KEY"]
anthropic_client = Anthropic(api_key=LLM_API_KEY)
```

```python
print("Welcome to your AI Assistant. Type 'goodbye' to quit.")

def main():
    while True:
        prompt = input("You: ")
        if prompt.lower() == "goodbye":
            print("AI Assistant: Goodbye!")
            break
        message = anthropic_client.messages.create(
            max_tokens=4096,
            messages=[
                {
                    "role": "user",
                    "content": prompt,
                }
            ],
            model="claude-sonnet-4-0",
        )
        for response in message.content:
            print(f"Assistant: {response.text}")

if __name__ == "__main__":
    main()
```

- ☒ Copy working host application code without clients ✅ 2025-04-26
- ☒ Step through line-by-line what is happening ✅ 2025-05-07

Here we have the beginnings of an MCP host: a very simple script that takes some user input and sends it to an LLM (in this case, Anthropic's Claude 3.5) and prints a response. Let's go through it line by line.

In lines 1-4, we import a few packages: `os` for loading environment variables, `anthropic` for the `Anthropic` client class, and `dotenv` for temporarily loading key-value pairs from a file into environment variables. This allows you to store sensitive information like API keys outside of the code in a file conventionally called `.env` in a `KEY=VALUE` format.

WARNING

Do not commit your .env files to version control, as this can expose your API keys to the public, leading to consequences like unauthorized use and high costs for pay-as-you-go API access. If you accidentally do this, immediately log in to your key provider and de-authorize the compromised key or keys.

In lines 6-9, the application calls `load_dotenv()` which does the work of loading your keys into environment variables, then gets the loaded key by accessing the `os.environ` dictionary. That key is then used to instantiate the Anthropic client.

After providing a welcome message to the user in line 11, the code enters an infinite loop. Line 14 requests a prompt from the user, while lines 15-17 check the prompt for the exit phrase. If it is found, the application prints a goodbye message to the user and exists.

Lines 18-28 handles communication with the model. This is a pattern you'll see over and over again when making LLM calls from code. From the Anthropic client, you access the `.messages.create()` function, and provide it with parameters like `max_tokens` to control the upper limit of tokens to generate in response to the user's query, `system` to set the system prompt, shaping how the model responds to your user, `messages` which is a list of dictionaries of prompts with a specified role (notice that we set the `role` to "user" and the `content` to the prompt from the user), and, finally the `model` specifies which model should be used.

The `messages` interface within the larger Anthropic API is very powerful, and the `create()` function has a number of other parameters that you can experiment with to tune the kinds of responses that you get from the model. You can read more at the official Anthropic documentation.

The final lines, 29-30, loop through the responses generated by the model, printing the text content of any messages received by the Anthropic client.

Next, let's shift from the Anthropic model client to the MCP client that our application will host.

The Client

When you’re building support for MCP into your applications, the client is one of the most important components that you will be building and working with. Clients allow for communication between MCP servers and your application, providing the interface between the server, your application, and the large language model you’re using.

Because clients act as the interface between your application and MCP servers, it is up to them to decide which server features they will support. As of this writing, MCP servers can provide to a host application:

- resources, which represent data such as text files, log files, and more
- prompts
- tools, which are executable code that an agent or model can run
- sampling, a way for a server to request chat completions from the host application’s model,
- images
- context, which provides built-in MCP capabilities to tools

You can learn more about each of these in Chapter 4: The Server.

In addition to this, clients can also support **roots**, which define boundaries (such as a specific filesystem location) within which connecting servers should operate. Roots are not strictly enforced, so it is up to the server to respect them and up to the client user to review potential servers before using them.

- ☐ Figure detailing the client’s location and responsibilities caption

As of this writing, there are two main transport mechanisms supported by MCP: Standard Input/Output (stdio) and Streamable HTTP. We will cover how to support each official mechanism in this chapter, and in Chapter 5 you will learn more about the transport layer itself, how official transport implementations work, and how to implement your own transport.

[!Note] Anthropic has recently released a beta version of MCP connector, which promises to allow users to access remote MCP servers directly via the Anthropic SDK's Messages API. While it seems like this could reduce the need for custom clients, as of this writing it only supports tools and remote MCP servers which, as you will see in the chapter on MCP servers, could represent a security risk. It also ties the user to Anthropic models like Claude, which may not fit your particular use case.

Basic Client Design

In MCP, a client handles all communication and connections to a server. Because client-server connections are one-to-one, it is typically best to build a client class. A client needs to at least do the following things:

- Connect to a server
- Discover the server's resources
- Make those resources available to an LLM

Beyond these basic operations, it is often helpful to implement the following features:

- Authentication
- Resource filtering
- Model agnosticism

For the rest of this chapter, you will learn how to build each of these features and what they enable for your applications. Let's start by sketching out an interface for a client class:

```
...
class MCPClient:
    def __init__(self) -> None:
        pass
```

```python
    async def connect(self) -> None:
        """
        Connect to the server set in the constructor.
        """
        pass

    async def list_tools(self) -> list[Any]:
        """
        Retrieve tools that the server has made available.
        """
        pass

    async def call_tool(self, tool_name: str, tool_args=None: list | None):
        """
        Given a tool name and optionally a list of argumnents, execute the
        tool
        """
        pass

    async def disconnect(self) -> None:
        """
        Clean up any resources
        """
        pass
```

In this class, you created the signatures for 3 methods, excluding the constructor:

- `connect()` - this will initialize your connection to the server. The implementation will look different based on what you choose for your transport layer
- `list_tools()` - this will use the client's server connection to retrieve the tools that the server makes available
- `call_tool()` - this function will call a tool given its name with any arguments provided by the caller

Another good name for your client could be `MCPServer`, as is done in the official Python examples, since from the perspective of the host application the instantiated object would represent an MCP server. This can be confusing, so for this example I decided to call it `MCPClient`. In your projects, choose what makes the most sense for you and your users.

Notice that this skeleton started with support for tools. Tools are one of the major MCP primitives, and arguably the most popular use case for MCP. Tools can help give agents their agency, augment the knowledge of LLMs, and much more, so we're starting with implementing tool support and will move on to show how to support all of the different resources an MCP server can provide.

[!NOTE] The three MCP primitives are **tools**, **prompts**, and **resources**.

Initializing the Client and Connecting to a Server

Before writing any code, let's think about *how* we want to connect to a server. This means thinking about which transport we want to use. In MCP, a **transport** is an implementation of the protocol's transport layer and manages how messages are sent between the client and server.

As of this writing, MCP has two major built-in transport implementations: stdio and streamable HTTP. The stdio transport is suitable for use cases where you expect servers will be run alongside the host application, because it uses standard input and output streams to carry communication between the client and server. This is the most common transport, as it is simple to write for and because most commercially available applications that host MCP clients expect the user to install and run servers themselves.

NOTE

The Python MCP SDK also includes a websocket transport and an HTTP Server-Sent Events (SSE) transport, which is being superseded by Streamable HTTP. Because of this, we will just cover connecting with Streamable HTTP, but the patterns established here along with Anthropic's backwards compatibility guide should give you enough to support both remote transports as a client developer. All of these transports will be explored in more detail in chapter 5.

The built-in Streamable HTTP transport, on the other hand, is best suited for applications where the MCP servers they consume can't be expected to be run alongside their host applications. This is most apparent in platform development and in hosted tools.

When developing a platform or framework for building and deploying downstream products, you can't guarantee that MCP servers will be deployed to the same machine as the application. If they're not, then a stdio client will never be able to talk to a server. Some organizations are experimenting with hosting MCP servers on their own network, exposing them to the public. If you wish to use those, you'll want to build Streamable HTTP support into your client.

WARNING

Warning: Any remote transports, including Streamable HTTP, need to be properly secured to prevent security problems. This includes authenticating the connection and validating origin headers.

First, let's look at the simplest case: connecting via stdio. In this section, you will build out the constructor and `connect()` methods for the `MCPClient` class. After that, you will do the same to build a Streamable HTTP client, which can support connecting to remote MCP servers.

After that, we will look at an example of a client supporting all three methods and how to choose the appropriate one.

Connecting with stdio

When you connect to an MCP server with stdio, the client will launch the server as a subprocess and then read messages from its standard input and send messages via its standard output. Luckily for us, the MCP Python SDK includes all the structures we will need to construct our client and connect via stdio.

First, let's set up the Client's constructor:

```
from contextlib import AsyncExitStack

from mcp import ClientSession

class MCPClient:
    def __init__(self, name: str, command: str, server_args: list[str], env_vars:
dict[str, str]=None) -> None:
        self.name = name
        self.command = command
        self.server_args = server_args
        self.env_vars = env_vars
        self._session: ClientSession = None
        self._exit_stack: AsyncExitStack = AsyncExitStack()
        self._connected: bool = False
```

In this constructor, you set up several important properties that will enable you to connect to and interact with an MCP server. The first, `name` just gives a human-readable name to the instantiated client. This is useful for logging, especially if you are maintaining connections to multiple servers, because it allows you to easily identify the specific origin of any emitted logs. It is not, however, required for an MCP client.

The next 3 parameters are used for setting up the server. `command` identifies the executable to be run to initiate the server. Most commonly, servers will run with either `python` or `node` (or `npx` depending on your local installation). It is common enough that for most use cases, your client could define an enum to ensure that a supported command is being used. `server_args` is a list of all the command-line arguments that are to be passed to the `command` executable. This will include, at the very least, the path to the server file. If the file takes command-line arguments, let's say `--port 8000 --verbose`, your `server_args` list will look like `["server.py", "--port", "8080"]`. Next is a dictionary of environment variables in `env_vars`. These work like typical environment variables, and you can even unpack your system's environment variables into this dictionary with `{**os.env}`. This isn't recommended, however, as it could expose sensitive information to your server.

We also include some "private" connection management properties, such as `_session`, which will hold the client session, `_exit_stack` which manages the asynchronous connection contexts (more on that later), and `_connected` which will prevent us from connecting to a server when we're already connected.

The next step is to implement the `connect()` and `disconnect()` methods. Here, you'll use the Python MCP SDK's built-in connection management objects to make and hold the server connection.

```
from contextlib import AsyncExitStack

from mcp import ClientSession
from mcp.client.stdio import stdio_client

class MCPClient:
    ...
    async def connect(self) -> None:
        """
        Connect to the server set in the constructor.
        """
        if self._connected:
            raise RuntimeError("Client is already connected")

        server_parameters = StdioServerParameters(
            command=self.command,
            args=self.server_args,
            env=self.env_vars if self.env_vars else None
        )

        # Connect to stdio server, starting subprocess
        stdio_connection = await
self._exit_stack.enter_async_context(stdio_client(server_params))
        self.read, self.write = stdio_connection

        # Start MCP client session
        self._session = await
self._exit_stack.enter_async_context(ClientSession(read_stream=self.read,
write_stream=self.write))

        # Initialize session
        await self._session.initialize()
        self._connected = True

    async def disconnect(self) -> None:
        """
        Clean up any resources
        """
        if self._exit_stack:
            await self._exit_stack.aclose()
            self._connected = False
            self._session = None
```

This may look complex, especially if you don't have any experience with writing asynchronous Python. Let's break down what we're doing here to make it more digestible, starting with the `connect()` method. The very first thing this method does is check the `_connected()` property. This will prevent the user from trying to open an already-open connection. Then, we instantiate a `StdioServerParameters` object, passing in the properties we set in the constructor. The rest of the code is creating the necessary processes and connections to run and use the MCP server. We do this via `_exit_stack`, which holds an instance of `AsyncExitStack`.

[!NOTE] `AsyncExitStack` allows you to manually nest asynchronous context managers without using the more common `async with` context manager syntax. This is an ideal choice because you can dynamically add more asynchronous context managers to the stack and can safely release all the resources on the stack in order with just one call. The latter feature is extremely useful because we can determine when to unwind the stack, rather than having to rely on the code leaving the scope of each context manager. This is done by calling the `enter_async_context()` function.

First, we open a connection to the stdio server, which starts the subprocess on which the MCP session will run. We pass in an instance of the `stdio_client` that is instantiated with the `server_params` built just before it. Then, we unpack the results of that call into `self.read` and `self.write` which represent the session's read and write streams, respectively. The next step is to start the MCP client session, again by calling `enter_async_context()`, but this time with an instance of `ClientSession` with the read and write streams set to `self.read` and `self.write`. This is stored in `self._session`, which we then initialize with `initialize()`. This method handles initiating the connection to the server, advertising the client's capabilities to the server, checking the protocol version supported by the server, and telling the server that the client has been successfully initialized. Finally, we set `self._connected` to `True`.

In sum, when writing the connection code for a stdio server, you:

1. create an instance of `StdioServerParams`
2. start the stdio server subprocess in an asynchronous context
3. start the MCP client session in another (nested) asynchronous context and store it in `self._session`
4. initialize the session itself

In this example, we also created the `disconnect()` method, which just calls `self._exit_stack.aclose()` to close all server connections in order, then sets `self._connected` to `False` and `self._session` to `None`. With this, you can cleanly connect and disconnect to MCP servers using the stdio transport at will. While you can't do anything with the servers just yet, you've completed the most complex part. Give yourself a pat on the back. Now we will do the same thing, but in support of the Streamable HTTP transport.

Connecting with Streamable HTTP

Third-party example Streamable HTTP is the main transport designed for use with remote MCP servers. Originally, HTTP Server-Sent Events (SSE) were the standard for connecting to remote servers, but a number of production issues with them made their continued support untenable. Those issues, and the architecture of both transports, will be covered in Chapter 5.

When you connect to a remote server with Streamable HTTP, you will be connected via a single endpoint defined by the server, and then responses can be either immediate standard HTTP responses (triggered by sending a POST to the server and not requiring an always-open connection) or streaming SSE responses (triggered by sending an empty GET to the server), the latter being optional and dependent on the server's implementation.

[!Note] "Wait," you might be saying, "I thought Streamable HTTP replaces HTTP+SSE, how is this different?" We will go into the upgrades in Chapter 5, but the major difference with this is that getting an SSE response is optional - both by the client that has to request a streaming response, and by the server that can choose whether or not to support it.

To support Streamable HTTP, we will again focus on building the `MCPClient`'s constructor, `connect()`, and `disconnect()` methods. Let's start with the constructor:

```python
from contextlib import AsyncExitStack
from typing import Callable

from mcp import ClientSession

class MCPClient:
    def __init__(self, name: str, server_url: str) -> None:
        self.name = name
        self.server_url = server_url
        self._session: ClientSession = None
        self.exit_stack = AsyncExitStack()
        self._connected: bool = False
        self._get_session_id: Callable[[], str] = None
```

This is eerily similar to the constructor for the stdio client. Because we're not starting a process locally, we don't need `command` or `server_args`, but we do need a location for the server, which `server_url` provides. The rest of the properties are the same in the stdio client.

Connecting and disconnecting from the server is also pretty similar to how you do it for the stdio client:

```python
from mcp import ClientSession
from mcp.client.streamable_http import streamablehttp_client

class MCPClient:
    ...
    async def connect(self, headers: dict | None = None) -> None:
        if self._connected:
            raise RuntimeError("Client is already connected")

        # Connect to Streamable HTTP server
        streamable_connection = await
self._exit_stack.enter_async_context(streamablehttp_client(url=self.server_url,
headers=headers)))
        self.read, self.write, self._get_session_id = streamable_connection

        # Start MCP client session
        self._session = await
self._exit_stack.enter_async_context(ClientSession(read_stream=self.read,
write_stream=self.write))

        # Initialize session
        await self._session.initialize()
        self._connected = True

    async def disconnect(self) -> None:
        """
        Clean up any resources
        """
        if self._exit_stack:
            await self._exit_stack.aclose()
            self._connected = False
            self._session = None
```

Hopefully this code looks very familiar - it's almost exactly the same as the `connect()` and `disconnect()` code for the stdio client. What we do differently here is take an optional `headers` parameter and pass that with the `server_url` property to `streamablehttp_client()`, which along with the `read_stream` and `write_stream` returns a callback for retrieving a session ID from the server if it provides one. If the server supports it, this can be used for resuming broken sessions. We also don't instantiate a `StdioServerParams` object, because the `streamablehttp_client` function doesn't take a special parameters object, instead taking the URL and headers directly as parameters. Other parameters that would be of interest for a client developer include:

- `timeout` - how long in seconds until HTTP operations time out, of type `datetime.timedelta`. Defaults to 30 seconds.
- `sse_read_timeout` - how long in seconds to wait for an additional event before timing out, of type `datetime.timedelta`. Defaults to 5 minutes.

- `auth` - this handles authentication and is of type `httpx.auth`. You will learn how to authenticate sessions with servers that support it in a later section.

While you are building out your client class, let’s take a look at one strategy for integrating it into a host chatbot application. We are going to use the stripped-down chatbot introduced at the beginning of this chapter to demonstrate how the MCP client fits in with a wider LLM-powered application. The only additions here will be to instantiate an stdio MCP client and open a connection to a made-up MCP server that provides calculator tools like `add_two_numbers`, `subtract_two_numbers`, `multiply_two_numbers`, and `divide_two_numbers`.

```
from client import MCPClient

mcp_client = MCPClient(name="calculator_server_connection", command="python",
server_args=["calculator_server.py", "--debug"])
mcp_client.connect()

print("Welcome to your AI Assistant. Type 'goodbye' to quit.")

def main():
    while True:
        prompt = input("You: ")
        if prompt.lower() == "goodbye":
            print("AI Assistant: Goodbye!")
            break
        message = anthropic_client.messages.create(
            max_tokens=4096,
            messages=[
                {
                    "role": "user",
                    "content": prompt,
                }
            ],
            model="claude-sonnet-4-0",
        )
        for response in message.content:
            print(f"Assistant: {response.text}")

if __name__ == "__main__":
    main()
```

All that we did here was instantiate the client and then call `connect()` to open a connection to the server. How would you change this for a Streamable HTTP server?

Now let's look at how to make use of everything an MCP server can provide to the client.

Interacting with MCP Server Capabilities

MCP servers can provide a wide variety of capabilities, including the MCP primitives *tools*, *resources*, and *prompts*. As a client developer, you will interact with them via the instantiated `ClientSession` object, and generally use the same two-step workflow for each of these: discovery and use. For each of these capabilities, discovery happens via a `<primitive>/list` request, which in the Python SDK is wrapped by the `list_<primitive>s()` methods. These calls, if the server provides the respective capability, will return a list of the available tools, resources, or prompts. Using the resulting capabilities requires a `<primitive>/<verb>` call, where the verb depends on the primitive being called. In the Python SDK, these calls are wrapped with methods of the form `<verb>_<primitive>()` such as `call_tool()` for `tool/call`.

NOTE

The protocol also allows notification messages to be sent by the server (as well as by the client), then it is up to the receiver to handle the notifications as it wishes. For Streamable HTTP connections, these notifications will be sent like any other message as defined by the transport, but will be sent as a `JSONRPCNotification`, and implementation of the `JSONRPCMessage` class.

While the available MCP SDKs wrap the various request calls in methods that your client can call directly, it's often a good idea to wrap them in your own object methods so that you can add logging, retries, method parameters and any other customizations to your calls to the server, like passing any of the `list_<primitive>()` methods the optional `cursor` parameter for pagination.

Tools

Tools are the most common resource MCP servers provide, and with good reason: agentic workflows often require tools to augment the abilities of a base LLM. Tools are simply a deterministic function that an LLM can decide to call if needed. Naturally, the function can do anything you can think of doing in code, from calculating numbers to finding the weather at a given location. MCP servers allow clients to discover and use these tools.

To discover tools using the Python SDK, your client would just call `self.session.list_tools()` within the client, which returns a `response` object with a `tools` property. This holds a list of any tools made available by the server connected to the client, where tools have the properties `name`, `description`, `inputSchema`, `annotations`, and `model_config`. You can see the current structure of the Tool class in the MCP Python SDK's types module.

Calling `list_tools()` directly can be fine for basic use cases, but if you're building a client that will be used for different servers with different capabilities and possibly in different applications, it can be advantageous to write your own function that has some guardrails, logging, and anything else your users might need abstracted away. For example, you could implement the following function:

```python
import logging
from mcp.types import Tool

logger = logging.getLogger(__name__)

class MCPClient:
    ...
    async def get_available_tools(self) -> list[Tool]:
        if not self._connected:
            raise RuntimeError("Client not connected to a server")

        tools_result = await self._session.list_tools()
        if not tools_result.tools:
            logger.warning("No tools found on server")
        return tools_result.tools
```

This method is a very simple wrapper around the `Session` object's `list_tools()` method which itself wraps the `tools/list` request. While simple, the example shows how you can implement some simple guardrails like the server connection status check, as well as more detailed logging like the empty tool result check. You can extend this pattern to add retries, translate MCP's tool format to another format compatible with another LLM, and more. Another helpful pattern that you will see later in this chapter is converting MCP's `Tool` objects into your own objects where you can customize their structure, implement translation functions to support other LLMs, and more.

Now that your client has tools, what do you do with them? Use them, of course! Like in the `get_available_tools()` example, we will wrap the `Session` object's `call_tool()` function and add just a bit of logging to it and handle the different available response types. `call_tool()` returns a `CallToolResult`, which is a list of any of the SDK's content types:

- `TextContent` - The result of the tool is any kind of text that you would return to user, and it's stored in the `text` property.
- `ImageContent` - If the result of the tool is an image, it will be returned as an `ImageContent` object, with the image data itself stored as a base64-encoded string in the `data` property
- `AudioContent`- Some tools return audio content. Like `ImageContent`, the data is base64-encoded and lives in the `data` property
- `EmbeddedResource` - Tools can return embedded resources as part of their response, typically for extra context or data caching. The resource data lives in the `resource` property, which can be either `TextResourceContents` (representing the text content of, say, a configuration file resource) or `BlobResourceContents`

(representing the binary data of the resource as a base64-encoded string)

One key thing to keep in mind is that tool use requests return lists of content, and in some cases, such as with an `EmbeddedResource` response being used as additional context, that should be expected and handled as such. In the example below, you will wrap `call_tool()` and return string representations of each of the valid tool content response types.

WARNING

For some use cases, handling the response types in the client itself may not be desirable. When building a client, consider your user: will they want or need control over how to handle the different content types?

```
from typing import Any
from mcp.types import TextResourceContents, BlobResourceContents

class MCPClient:
    ...
    async def use_tool(self, tool_name: str, arguments: dict[str, Any] | None=None)
-> list[str]:
        if not self._connected:
            raise RuntimeError("Client not connected to a server")

        tool_call_result = await self.session.call_tool(name=tool_name,
arguments=arguments)
        logger.debug(f"Calling tool {tool_name} with arguments {arguments}")

        results= []
        if tool_call_result.content:
            for content in tool_call_result.content:
                match content.type:
                    case "text":
                        results.append(content.text)
                    case "image" | "audio":
                        results.append(content.data)
                    case "resource":
                        if is_instance(content.resource, TextResourceContents):
                            results.append(content.resource.text)
                        else:
                            results.append(content.resource.blob)
        else:
            logger.warning(f"No content in tool call result for tool {tool_name}")
        return results
```

Here, we built a `use_tool()` method that, like the `list_tools()` method, first checks that the client is connected to a server session and raises an exception if it isn't. Then we make the `call_tool()` call, passing in the tool's name and arguments. On the next line, we added a debug statement so we know what tool is being called. If you want the user to decide how to handle the different content types, you could return the results immediately at this point. But for this example, we check that the `content` list is not empty, then iterate through each item in it. There, we check the item's type and add the proper string representation to our results list. If `content` is empty, we show the user a warning, and then finally return the results list. Just like with `list_tools()`, wrapping `call_tool()` gives us much more control over what happens during the tool call.

WARNING

When building agents that support MCP, keep a close eye on tool-calling and overall performance as you add tools to the agent. Performance tends to drop dramatically as more tools are added, typically due to an increased difficulty in selecting the correct tool for a prompt. This can be due to ambiguity or overlap in tool descriptions as well as descriptions and interfaces having too much information in aggregate for the LLM to process accurately.

If you're building your client to serve a specific application, like the simplified chat application introduced in the beginning of this chapter, you will also have to use these calls. Many LLM APIs that support tool-calling have an optional `tools` parameter, making sending the tools available to a model simple. Other models require building a list of tool names, descriptions, and schemas into your system prompt, but for this example we are focusing on Anthropic's Claude API, which does accept a `tools` parameter. This will be added to any user messages to the model, and then the resulting responses will be analyzed by the host application for any tool usage. If it's found, the application will formulate a `tool_result` message and add it to a messages list that gets sent back to the model to generate the final result for the user.

- ☐ Todo - sequence diagram for this with App, Model, Server explaining the flow above.

```
import os

from anthropic import Anthropic
from dotenv import load_dotenv

load_dotenv()

LLM_API_KEY = os.environ["LLM_API_KEY"]

anthropic_client = Anthropic(api_key=LLM_API_KEY)
mcp_client = MCPClient(name="calculator_server_connection", command="python",
server_args=["calculator_server.py", "--debug"])
mcp_client.connect()
available_tools = mcp_client.get_available_tools()
available_tools_dict = [tool.model_dump() for tool in available_tools]

print("Welcome to your AI Assistant. Type 'goodbye' to quit.")

def main():
    while True:
        prompt = input("You: ")
        if prompt.lower() == "goodbye":
            print("AI Assistant: Goodbye!")
            break
        message = anthropic_client.messages.create(
            max_tokens=4096,
            messages=[
                {
                    "role": "user",
                    "content": prompt,
                }
            ],
            model="claude-sonnet-4-0",
            tools=available_tools_dict,
            tool_choice="auto"
        )

        tool_use_message_block = {
            "role": "user",
            "content": list[dict[str, Any]]: []
        }
        if message.stop_reason == "tool_use":
            tool_use_messages = [message for message in message.content if
message.type == "tool_use"]
            for tool_use in tool_use_messages:
                tool_result = await mcp_client.use_tool(tool_name=tool_use.name,
arguments=tool_use.input)
                tool_use_message_block["content"].append(
                    {
                        "type": "tool_result",
                        "tool_use_id": tool_use.id,
                        "content": tool_result
                    }
                )

        if tool_use_message_block["content"]:
            response = client.messages.create(
                max_tokens=4096,
                messages=[
                    {"role": "user", "content": prompt},
                    {"role": "assistant", "content": message.content},
                    tool_use_message_block
                ],
                model="claude-sonnet-4-0",
                tools=available_tools_dict,
                tool_choice="auto"
            )
        else:
            response = message

        display_response = next(message.text for message in response.content if
hasattr(message, "text"))
        print(f"Assistant: {display_response}")

if __name__ == "__main__":
    main()
```

This code has a number of changes to handle getting and using any required tools. First, during the initialization phase, we get all available tools from the server we're connected to with `get_available_tools()`. Then, we transform the list of `Tool` objects in to a list of plain dictionaries to properly feed it back to the model ([ ] TODO see if this is necessary)). This is done at the top of the module in order to only pull the list of tools once. If you expect the server or servers you connect to to change the tools they provide while connected, you will want to either get a fresh list of available tools for every user input to the model or have your client listen to and handle a `list_changed` notification from the server.

Next, we've updated the code where we create the user message to the model to pass in the list of tools to the `tools` parameter, as well as setting `tool_choice` to `"auto"`. You don't have to do this, since auto is the default setting and allows the model to decide which tool or tools to use to answer the user's query. You can also set it to `"any"`, where it will use any (but at least one) tool that is available to it, a tool name and type `"tool"` to force the model to use a specific tool, and `"none"` to prevent the model from using any available tools.

After this, we inspect the result and initialize a potential tool use message that will be sent back to the model. If the message response has a `stop_reason` of `tool_use`, then we know the model is waiting for a response with the results of the tool that it is requesting. Because there is a chance that the model could request more than one tool call, we crate a filtered list of only tool use messages, and then iterate through them, calling the tool specified by the model with the arguments specified by the model. Then we add a dictionary to the tool use message's `content` property with the key `type` set to `"tool_use"`, `tool_use_id` set to the ID in the model's tool use message response, and `content` to the result of the tool call.

After the code exits that loop's scope, it checks for a truthy value in the tool use message, which will be set if there was a successful tool call. If it is set, we send a list of messages back to the model: the user message, an assistant message with the original tool use message content, and the whole tool use response list we built earlier. If we don't have any tool call responses in the tool use message, we just set `response` to the original message. Finally, we use the `next()` manual iterator to grab the first message's content block that has a `text` attribute and display it to the user.

In this section, you learned about MCP server-provided tools, how to discover them on a server, how to use them, and how to use them in the context of a chat application. In the next section, you will learn about resources and how you can use those in your application like you just did with tools.

Resources

As you learned in Chapter 2, MCP resources can play many roles, some of which have yet to be fully explored by the community members themselves, like as a cache to optimize prompt size. Basically, resources give a host application and the LLM it interacts with access to data served by the MCP server. This can be database records, images, text files, and more. Resources also have several actions you can take on them:

- `resources/list` or `list_resources()` - requests a list of available resources from the server
- `resources/templates/list` or `list_resource_templates()` - requests a list of resource URI templates from the server. These allow you to construct a resource's URI dynamically based on other information you might have. In the Python SDK, these strings look like f-strings, with variable sections enclosed in curly braces.
- `resources/read` or `read_resource()` - accesses the file provided by the server, sending the data in text or blob format to the LLM.
- `resources/subscribe` or `subscribe_resource()` - subscribes the client to updates to to the resource URI you specify in the call.
- `resources/unsubscribe` or `unsubscribe_resource()` - unsubscribes the client from updates to the resource specified by resource URI in the parameters.

NOTE

See the tupac project by Tim Kellogg for a minimalist client implementation that utilizes MCP resources as a cache for efficient prompting.

First, we will implement wrappers for the two list methods. We will continue the pattern that we established in the last section for `get_available_tools()`: a light wrapper with a few basic guardrails and logging that can be used as a base for more complex workflows.

```python
from mcp.types import Resource, ResourceTemplate

class MCPClient:
    ...
    async def get_available_resources(self) -> list[Resource]:
        if not self._connected:
            raise RuntimeError("Client not connected to a server")

        resources_result = await self._session.list_resources()
        if not resources_result.resources:
            logger.warning("No resources found on server")
        return resources_result.resources

    async def get_available_resource_templates(self) -> list[ResourceTemplate]:
        if not self._connected:
            raise RuntimeError("Client not connected to a server")

        resource_templates_result = await self._session.list_resource_templates()
        if not resource_templates_result.resources:
            logger.warning("No resource templates found on server")
        return resource_templates_result.resources
```

These two methods are almost identical to each other and to the `get_available_tools()` method. Again, each function checks for a connection before making any server requests, then calls the appropriate list method, checks for a result, and then returns it, including a warning if the result is empty. Like the `list_tools()` method, both of these list methods take an optional cursor parameter for paginating results. We will see this pattern again for prompts in the next section, so think about how you could expand the functionality of the list method wrappers and how you could also make the code more concise.

NOTE

Resources and resource templates have almost the exact same set of properties: `uri`, `name`, `description`, `mimeType`, `size`, `annotations`, and `model_config`. For resource templates, instead of a `uri`, they have a `uriTemplate` property.

Next, we'll implement a wrapper of the `read_resource()` session method, and following the pattern the previous example established, we'll call it `use_resource()`. This code will take a resource URI (or a filled in URI from a resource template), read the resource and return it to the caller either as a text string or a base64-encoded data string.

```python
from mcp.types import BlobResourceContents, TextResourceContents

class MCPClient:
    ...
    async def get_resource(self, uri: str) -> list[BlobResourceContents |
TextResourceContents]:
        if not self._connected:
            raise RuntimeError("Client not connected to a server")
        resource_read_result = await self._session.read_resource(uri=uri)

        if not resource_read_result.contents:
            logger.warning(f"No content read for resource URI {uri}")
        return resource_read_result.contents
```

This method requires little extra: just our now-standard connection guardrail and empty result warning. Similarly to `get_available_tools()`, instead of transforming the MCP `BlobResourceContents` and `TextResourceContents` classes, we return them as-is to the user, since they provide valuable information about the type of string being returned and this keeps with the minimal processing philosophy of this set of examples.

The MCP protocol also allows clients to subscribe to resource change notifications, however as of this writing, notification handling is only partially implemented in the Python SDK and there is no provided way to route notifications to handler functions or register handlers. This does seem to be available in the Typescript SDK, however.

So far, you learned how to get a list of available resources and resource templates and, when armed with a resource URI, how to read that resource. But how would you use them in a host application? Remember that resources primarily exist to provide additional context to user queries, allowing your application to do things like adding a Bash script to a user query about their script. To see this in action, let's use our minimal host chatbot application example from the beginning of the chapter again, but this time we will look for the key "context:" in the user prompt and use that as context to the model request. When the application starts and any time the user issues the command "refresh," the application will get the current list of resources from the server and then display them. When the application finds "context:" in the user prompt, it will use what comes after to select a specific resource and add it as context to the original user question.

```python
import os
import re

from anthropic import Anthropic
from dotenv import load_dotenv

from mcp_client import MCPClient

load_dotenv()

LLM_API_KEY = os.environ["LLM_API_KEY"]

anthropic_client = Anthropic(api_key=LLM_API_KEY)
mcp_client = MCPClient(name="calculator_server_connection", command="python",
server_args=["calculator_server.py", "--debug"])
mcp_client.connect()


def extract_resource_name(prompt: str) -> str | None:
    match = re.search(r'context:\s+(\S+)', prompt)
    return match.group(1) if match else None

def display_resources(available_resources: dict[str, dict[str, str]) -> None:
    print(f"Available resource contexts are: ")
    for resource in available_resources:
        print(f"{resource.name} -- {resource.description}")
    print(f"Add a context to your prompt with 'context: <name>' entering the name of
one of the resources above.")
```

```
def refresh_resources() -> dict[str, dict[str, str]]:
    available_resources = await mcp_client.get_available_resources()
    return {resource.name: resource.model_dump() for resource in
available_resources}

def main():
    print("Welcome to your AI Assistant. Type 'goodbye' to quit or 'refresh' to
reload and redisplay available resources.")
    available_resources = refresh_resources()
    display_resources(available_resources=available_resources)

    while True:
        prompt = input("You: ")
        lc_prompt = prompt.lower()
        context_messages = []

        if lc_prompt == "goodbye":
            print("AI Assistant: Goodbye!")
            break

        if lc_prompt == "refresh":
            available_resources = refresh_resources()
            display_resources(available_resources=available_resources)
            break

        if "context:" in lc_prompt:
            resource_name = extract_resource_name(prompt=lc_prompt)
            if not resource_name:
                print("WARNING: Context resource name not found in user input")
            elif resource_name in available_resources.keys():
                resource_contents = await
mcp_client.get_resource(uri=available_resources[resource_name]["uri"])
                if isinstance(resource_contents, TextResourceContents):
                    context_message = {
                        "type": "text",
                        "text": resource_contents.text
                    }
                elif resource_contents.mimeType in ["image/jpeg", "image/png",
"image/gif", "image/webp"]:  # b64-encoded image
                    context_message = {
                        "type": image
                        "source": {
                            "type": "base64",
                            "media_type": resource_contents.mimeType,
                            "data": resource_contents.blob
                        }
                    }
                else:
                    print(f"WARNING: Unable to process mimeType
{resource_contents.mimeType} for resource {resource_name}")

        messages = [
            {
                "role": "user",
                "content": [
                    {
                        "type": "text",
                        "text": prompt
                    }
                ]
            }
        ]
        messages[0]["content"].extend(context_messages)

        message = anthropic_client.messages.create(
            messages=messages,
            model="claude-sonnet-4-0",
        )

        display_response = next(message.text for message in response.content if
hasattr(message, "text"))
        print(f"Assistant: {display_response}")

if __name__ == "__main__":
    main()
```

This integration started just like the integration you did in the Tools section: connect to the MCP server via the client, discover available primitives, like resources in this case, and load those primitives into memory. We added two new utility functions: `extract_resource_name()` which does some basic regular expression parsing to get a resource name from the user's input and `display_resources` which displays the list of available resources to the user, creates a dictionary for fast access by resource name, and optionally refreshes them. It also provides information to the user on how to add context to their prompts.

In the `main()` function, we updated the welcome message to let the user know that a message with only the word "refresh" will refresh and redisplay the resource list. We added a check for the "refresh" message, and then a check for the “context:” keyword. If that is found anywhere in the user input, `extract_resource_name()` is called to try and pull the resource name from the prompt. If the code can’t extract a resource name, it prints a warning and then leaves the scope of the “context:” check, extends the `content` field of the first message in the `messages` list. This is still an empty list, and so nothing actually changes in this case, and the plain user prompt is sent as-is to the LLM.

If the resource name is is found in the prompt and also exists in the list of available resources, we read the resource into memory and check whether it is plain text or a blob. If it is text, we create a content block with the text. If it is a blob, we check that the `mimeType` matches any of the Anthropic model API-supported image types and if so construct a content block with that data and then extend the content list in the user message with the constructed dictionary. If neither are true, we warn the user and continue sending the user prompt to the LLM.

NOTE

When doing things like adding context to a user prompt, you should not create a separate user message like you do with the user prompt. Instead, you convert the user message to its full form: a nested dictionary with a “type” key and a “content” key, which holds a list of dictionaries and add any context messages to that list.

There are several improvements you can make to this example. To stretch your knowledge, try to implement support for listing and using resource templates along with normal resources. Then, try to support multiple context files instead of just one, either with regular text parsing or even another LLM call. You can do some more cleanup: whether or not the “context:” keyword is found, that part of the prompt will still be sent to the LLM. This isn’t ideal. How would you improve that?

NOTE

As you might have picked up on, many of these improvements are typical software engineering best practices and not really specific to generative AI. While AI engineering is becoming a discipline unto itself, it is still software engineering at its core and thus the practices and tools you have as a software engineer will also serve you well building for AI.

Prompts

Prompts are the last of the three main primitives provided by MCP servers. These are designed to be controlled by the application user, who should be able to pick out which prompt or prompts they want to use, similar to the interface you saw in the host application example in the previous section. Each prompt definition has a unique identifier `name`, an optional human-readable `description`, and optional `arguments`, which is a list of dictionaries with a `name`, optional human-readable `description` and an optional `required` boolean indicating if the argument is required or not. It is then up to the client implementation or host application using the prompt to inject values for each of the arguments. These are called **dynamic prompts**.

The Python SDK simplifies handling prompts for us, with the `Session` class's `list_prompts()` method returning a list of `Prompt` objects, which have the prompt definition properties listed in the previous paragraph. When you get a specific prompt with `get_prompt()`, you provide the prompt's name and arguments and get back a list of `PromptMessage` objects, which are ready-to-use messages that can be converted to a plain dictionary and sent directly to the LLM. Take a look at the host integration example in the last section, paying close attention to the structure of the messages that are sent to the LLM. They are dictionaries with `role` and `content` keys, and those are the two required properties of a `PromptMessage` object. The content itself can be `TextContent`, `ImageContent`, `AudioContent`, or an`EmbeddedResource`. Embedded resources are just what you think they are: resources that are embedded in a message content block, just like our example host application built in the previous section.

Let’s continue the pattern from the previous two sections and wrap these two methods in our own client:

```
from mcp.types import Prompt

class MCPClient:
    ...
    async def get_available_prompts(self) -> list[Prompt]:
        if not self._connected:
            raise RuntimeError("Client not connected to a server")

        prompt_result = await self._session.list_prompts()
        if not prompt_result.prompts:
            logger.warning("No prompts found on server")
        return prompt_result.prompts
```

There isn’t too much to explain about this pattern that you haven’t seen before. We set up a connection check, get the list of prompts, and ensure that the result list isn’t empty before returning it to the user. The `get_prompt()` wrapper will be very familiar as well. We will get the prompt name and arguments from the user, load the prompt from the server with the provided arguments, and return the MCP `PromptMessage` object back to the application.

```
from mcp.types import PromptMessage

class MCPClient:
    ...
    async def load_prompt(self, name: str, arguments: dict[str, str]) ->
list[PromptMessage]:
        if not self._connected:
            raise RuntimeError("Client not connected to a server")
        prompt_load_result = await self._session.get_prompt(name=name,
arguments=arguments)

        if not prompt_load_result.messages:
            logger.warning(f"No prompt found for prompt {name}")
        else:
            logger.warning(f"Loaded prompt {name} with description
{prompt_load_result.description}")
        return prompt_load_result.messages
```

This is again similar to how we retrieved resources in the last section. We have our standard guardrails, we pass the `name` and `arguments` unchanged to the session's `get_prompt()` method, and add some logging.

NOTE

As with resources and tools, in the protocol there is support for notifications when the list of available prompts has changed. Unfortunately, there is also no user-facing API for handling and listening for these notifications from the server, although clients can subscribe to them. If you want to use them, you will have to handle routing to a callback on your own, but there are a few examples in the SDK's source code you can follow, such as how sampling callbacks are routed in `Session._received_request()`. Source code. You could also create a new function that handles all messages and pass that to the client constructor in the `message_handler` parameter. This would have to handle all messages, but would allow you to build notification handling logic as well.

Like the other primitives, there are as many ways to integrate prompts into your application as you can imagine, and then some. To show off how you can use the available prompt functionality in your application, we will use the same base application as the last two examples so that the usage of the server and prompt primitives is clear and not obscured by too much application logic. We will also use a similar user interaction pattern to the one we used in the Resources section. In this pattern, the application gets a list of available primitives (in this case, prompts), displays them to the user, and allows the user to select which one to use within their input. That prompt will be sent as-is by itself to the LLM and the application will handle and display the response.

```
import os
import re

from anthropic import Anthropic
from dotenv import load_dotenv

from mcp_client import MCPClient

load_dotenv()

LLM_API_KEY = os.environ["LLM_API_KEY"]
```

```python
anthropic_client = Anthropic(api_key=LLM_API_KEY)
mcp_client = MCPClient(name="calculator_server_connection", command="python",
server_args=["calculator_server.py", "--debug"])
mcp_client.connect()


def parse_prompt(prompt: str) -> dict[str, str]:
    pattern = r'prompt:\s*(\w+)(?:\s+(\w+)\s+(\w+))*'
    full_match = re.search(r'prompt:\s*(\w+)((?:\s+\w+\s+\w+)*)', text)

    if not full_match:
        return None

    prompt_name = full_match.group(1)
    args_string = full_match.group(2).strip()
    arg_pairs = re.findall(r'(\w+)\s+(\w+)', args_string)

    return {
        'prompt_name': prompt_name,
        'arg_pairs': {arg[0]: arg[1] for arg in arg_pairs if arg_paris}
    }

def display_prompts(available_prompts: dict[str, dict[str, str]) -> None:
    print(f"Available prompts are: ")
    print(f"Prompt Name -- Prompt Description -- Prompt Arguments")
    for prompt in available_prompts:
        if arguments := prompt.get("arguments"):
            prompt_arguments = "".join([f"{argument['name']},
{argument['description']}, {argument['required']}" for argument in arguments])
        print(f"{prompt['name']} -- {prompt['description']} -- {prompt_arguments}")
    print(f"Use a prompt with 'prompt: <name> <argument_name1> <argument_value1>
...' using the name and argumetns of one of the prompts above.")

def refresh_prompts() -> dict[str, dict[str, str]] | None:
    available_prompts = await mcp_client.get_available_prompts()
    return {prompt.name: prompt.model_dump() for prompt in available_prompts} if
available_prompts else None

def main():
    print("Welcome to your AI Assistant. Type 'goodbye' to quit or 'refresh' to
reload and redisplay available resources.")
    available_prompts = refresh_prompts()
    if available_prompts:
        display_prompts(available_prompts=available_prompts)

    while True:
        prompt = input("You: ")
        lc_prompt = prompt.lower()
        messages = []

        if lc_prompt == "goodbye":
            print("AI Assistant: Goodbye!")
            break

        if lc_prompt == "refresh":
            available_prompts = refresh_prompts()
            display_prompts(available_prompts=available_prompts)
            break

        if "prompt:" in lc_prompt:
            prompt_input = parse_prompt(prompt=lc_prompt)
            if not prompt_input:
                print("WARNING: Prompt input malformed. Use a prompt with 'prompt:
<name> <argument_name1> <argument_value1> ...'")
            elif prompt_input["name"] in available_prompts.keys():
                prompt_contents = await
mcp_client.load_prompt(name=prompt_input["name"],
arguments=prompt_input['arguments'])
                messages = [{"role": prompt_message.role, "content":
prompt_message.content} for prompt_message in prompt_contents]
            else:
                print(f"Prompt name {prompt_input['name']} not found in available
prompts.")

        if not messages:
            messages = [
                {
                    "role": "user",
                    "content": [
                        {
                            "type": "text",
                            "text": prompt
                        }
                    ]
                }
            ]

        message = anthropic_client.messages.create(
            messages=messages,
            model="claude-sonnet-4-0",
        )

        display_response = next(message.text for message in response.content if
hasattr(message, "text"))
        print(f"Assistant: {display_response}")
```

```
if __name__ == "__main__":
    main()
```

This version of the code does all the things we set up in the introduction to this example. Starting with the main loops, we first call a `refresh_prompts()` function that calls the `get_available_prompts()` method that we built into the client, and returns a list of dictionaries similar to the Resources example, where the keys are the prompt names and the values are the dictionary forms of the prompt objects. If prompts are found, we then call `display_prompts()` which presents a list of the available prompts to the user as well as instructions for using them.

In this example, we allow the user to manually refresh and redisplay the prompts with the "refresh" command, and to use a prompt by providing the "prompt:" command, the prompt name, and any pertinent arguments. We unpack the list of Prompt objects into Python primitives to use with the Anthropic Python client, then send the messages to, and display messages from, the model. Being able to handle these primitives from MCP servers will cover nearly all MCP server use cases that are currently available (and some that aren't, like the notification system). But there are a few more things you can do with your client-server connections if you want to work at a lower level.

Going Deeper

At the base of MCP, as well as the various SDK implementations of the protocol, is a system that sends and receives messages between clients and servers. In the Python SDK, all of the session methods we wrapped themselves wrap a `send_request()` method. This method can be found in `mcp.shared.session.BaseSession`, which both the client- and server-specific `Session` classes inherit from. Because of this, you could theoretically also send your own custom requests to a server that supports them, but will have to create your own Request, RequestParams, and Result classes to do so.

There are a few other undocumented methods you can make use of in the client session object. `send_ping()` sends a ping request to the connected server. The server will then return an `EmptyResult` within a `ServerResult`, which carries no data but signifies the presence of a working connection to the server. `set_logging_level()`, given a `LoggingLevel` object from MCP's `types` library, can change the server's logging level for the connection if the server allows it. Server-initiated logs are sent to the client via notifications, however unlike subscription notifications, logging notifications have their own parameter in the `ClientSession` constructor called `logging_callback`. You would implement your own logging callback function that adheres to the `LoggingFnT` protocol (source code) and pass it to the constructor via the `logging_callback` parameter:

```
from mcp.types import LoggingMessageNotificationsParams

class MCPClient:
    ...
    async def _handle_logs(self, params: LoggingMessageNotificationsParams) -> None:
        if params.level in ("error", "critical", "alert", "emergency"):
            print(f"[{params.level}] - {params.data}")

    async def connect(self) -> None:
        """
        Connect to the server set in the constructor.
        """
        ...  # Removing setup code for brevity
        # Start MCP client session
        self._session = await
self._exit_stack.enter_async_context(ClientSession(read_stream=self.read,
write_stream=self.write, logging_callback=_handle_logs))

        # Initialize session
        await self._session.initialize()
        self._connected = True
```

Here, `handle_logs()` is a very simple function that checks the reported log level in the server-provided params, and if the level is within the set we provided, prints the level and data. This list of levels is not exhaustive, but MCP adheres to the syslog severity levels defined in RFC 5424 . In addition to `level` and `data`, `LoggingMessageNotificationsParams` also provides an optional `logger` name and `model_config` dictionary.

`send_progress_notification()` sends a notification to the server with some `progress_token` and the amount of `progress` as a float. Optionally, you can also send a `total` float and `message` string to the server. How these values end up looking will depend on the server you're connecting to and what the server requires for progress updates.

Providing MCP Client Capabilities

In the previous section, you saw an example of implementing a callback to handle log messages from a connected MCP server. In this one, you'll see this pattern repeated, but in service of providing client capabilities to the connected MCP server. There are two major capabilities that a client can provide to a server: **sampling**, where the server is able to request chat completions from the LLM the client is connected to via the client, and **roots**, which indicate to servers what areas in the host application's filesystem the server has access to.

Servers are not required to use or respect these capabilities, but if your client provides them, you are required to advertise that your client does provide them, which happens during the initialization phase of the client-server connection. For providing client capabilities, the general workflow is as follows:

1. In your client class, implement a callback function that will be called when the server requests it.
2. Ensure that the callback follows the established protocol for the capability, such as `SamplingFnT`.
3. Pass the callback function to the `ClientSession` constructor via the appropriate parameter, such as `sampling_callback` for sampling.

And that's it. Of course, the majority of the effort comes in step 1, and will be unique depending on your use case, interaction flow.

Sampling

Sampling is a client capability that allows MCP server to make use of whichever LLM the client and its host application is connected to. This opens up agentic workflows from the MCP server, such as allowing nested tool choice and use, or for the server to ask the LLM a question and use that to choose or fill information into a prompt that it provides, or whatever else your imagination can come up with. Because this is a client-provided capability, there is slightly less work to do here than there is for using server-provided capabilities. Like you saw with logging, the `ClientSession` class provides a `SamplingFnT` protocol which defines the signature of the sampling callback function you will implement, and the constructor has a `sampling_callback` parameter that will take your callback to be used if and when a connected server sends a sampling request.

WARNING

Because of the risk of giving an external server access to the LLM your application is using and that you are paying for, both myself and Anthropic strongly recommend that you implement some form of human-in-the-loop (HITL) approval process before a server is allowed to send a request to an LLM via your client and application.

In this example, we will follow the general workflow above, by creating a simple sampling callback that implements the `SamplingFnT` interface and passing it into the `ClientSession` constructor:

```python
from mcp.shared.context import RequestContext
from mcp.types import CreateMessageRequestParams, CreateMessageResult, ErrorData,
TextContent

class MCPClient:
    def __init__(self, name: str, server_url: str, llm_client: Anthropic) -> None:
        self.name = name
        self.server_url = server_url
        self._session: ClientSession = None
        self.exit_stack = AsyncExitStack()
        self._connected: bool = False
        self._get_session_id: Callable[[], str] = None
        self._llm_client = llm_client

    async def _handle_sampling(self, context: RequestContext, params:
CreateMessageRequestParams) -> CreateMessageResult | ErrorData:
        messages = []
        for message in params.messages:
            if isinstance(message, TextContent):
                messages.append({"role": message.role, "content": message.text})
            else:
                messages.append({"role": message.role, "content": message.data})
        response = self._llm_client.messages.create(
            max_tokens=params.maxTokens,
            messages=messages
            model="claude-sonnet-4-0",
        )
        return CreateMessageResult(
            role=response.role,
            content=response.content
            model="claude-sonnet-4-0"
        )

    async def connect(self) -> None:
        """
        Connect to the server set in the constructor.
        """
        ...  # Removing setup code for brevity
        # Start MCP client session
        self._session = await
self._exit_stack.enter_async_context(ClientSession(read_stream=self.read,
write_stream=self.write, sampling_callback=_handle_sampling))

        # Initialize session
        await self._session.initialize()
        self._connected = True
```

In this example, we added an `llm_client` parameter to the `MCPClient` constructor, since the client will need access to the LLM client to route messages between the LLM and the server. The handler, `_handle_sampling()` constructs a message from the MCP types and sends it on to the LLM, assuming an Anthropic model. Do not do this in practice. Your handler should request permission from the user before sending a client's prompt to the application's model. Then, in the `connect()` method, we pass the sampling handler method into the `ClientSession` constructor via `sampling_callback`. To see all of the parameters that a server is able to send you, you can view the CreateMessageParams class definition.

Supporting Roots

- ☐ TODO
- Roots in MCP are typically exposed through workspace or project configuration interfaces. (from protocol)
- In the protocol, servers SHOULD respect boundaries, but they aren't required to.

Supporting Multiple Models

One of the major benefits of using MCP is that it gives you the freedom, as the application developer, to allow your users to use many models, or the model of your choice. With MCP, you aren't tied to a single model, despite the examples in this chapter working with an Anthropic model. The simplest way to support this would be to write a translation function in your client for each primitive-model family pair that turns MCP objects into the format expected by the model family in question. You can improve on this basic design by creating a class for each primitive that can be constructed from the MCP object and has a translation method as a member of that class. The code below does just that for tools:

```python
from typing import Any

class InternalTool:
    def __init__(self, name: str, input_schema: dict[str, Any], description: str |
None = None) -> None:
        self.name = name,
        self.input_schema = input_schema,
        self.description = description

    def translate_to_openai(self) -> dict[str, Any]:
        return {
            "type": "function",
            "name": self.name,
            "description": self.description,
                "parameters": self.input_schema
            }

example_tool = InternalTool(name=mcp_tool.name, input_schema=mcp_tool.inputSchema,
description=mcp_tool.description)
openai_tool = example_tool.translate_to_openai()
```

Because the Anthropic and OpenAI tool formats are so similar, this example is fairly simple: we write a constructor that sets the instance's `name`, `input_schema`, and `description`, then we write `translate_to_openai()` which returns a dictionary with the keys `type`, `name`, `description` and `parameters`. The expected structure for OpenAI's `parameters` and Anthropic's `inputSchema` is the same, as well.

[!Note] In the previous example, we translated a Tool object from our internal representation to OpenAI's function calling format. OpenAI's API also supports directly calling remote MCP servers.

Using Multiple Servers

All of the examples so far have shown your client connecting to a single server. And while a single client can only connect to a single server, the Python SDK provides a `ClientSessionGroup` class to help manage multiple sessions concurrently within a client. It does this by connecting either to an existing `ClientSession` with `connect_with_session()` or to a server by creating a new `ClientSession` with `conenct_to_server()` and then loading all primitives/components (tools, resources, and prompts) offered by the connected server into object properties. Sessions are also stored in the `ClientSessionGroup` object, along with a mapping of each tool to the appropriate session. All of these can be accessed via the properties `sessions`, `prompts`, `resources`, and `tools`, and tools can be called just like they would be in a single `ClientSession`: with `call_tool()`. You can also disconnect from an individual server with the `disconnect_from_server()` method, which will also remove any loaded server components. In the example below, we will create a simple client that connects to servers via a `ClientSessionGroup`.

```
from mcp.client.session_group import ClientSessionGroup, ServerParameters

class MCPClient:
    def __init__(self, name: str) -> None:
        self.name = name
        self.server_url = server_url
        self._session_group = ClientSessionGroup()

    @property
    def connected() -> bool:
        return True if self._session_group.sessions else False

    async def connect(self, server_params: ServerParameters) -> None:
        """
        Connect to a server via ClientSessionGroup.
        """
        # Start MCP client session
        await self._session_group.connect_to_server(server_params=server_params)
        self._connected = True
```

This is simpler from the client perspective, since the `ClientSessionGroup` handles all of the connection management that the single-server client was in previous examples. Instead of a `_session` property that is `None` until a connection is made, we instantiate a `ClientSessionGroup` object and store it in `_session_group`. We changed the `connected` object member to a dynamic property that checks if there are any session objects in the session group's `sessions` list. Then, we have the `connect` method, which just takes a `ServerParameters` object. This is actually a Python type alias for `StdioServerParameters`, `SseServerParameters`, or `StreamableHttpParameters`. We use that to connect to a single server, and all connection management is handled by the session group. The flipside to this is that there isn't support yet for including callbacks when establishing the single `ClientSession` objects. This can be worked around by accessing the "private" callback properties of the `ClientSession` object, but this should be used sparingly.

The `ClientSessionGroup` constructor also takes an optional hook in the parameter `component_name_hook`, which is for renaming a component so as to prevent naming collisions. This is just any function that takes a string (the component name) and an Implementation `object`, and returns another string (the renamed component).

Best Practices

Some best practices aren't yet fully established as yet, some are established by the community, some are general engineering best practices, but an overview of all these as they apply to clients will help you quickly build resilient, production-ready MCP clients for your own applications.

Security

- Auth
- Human in the loop

Resuming Connections

- resuming (remote clients) - from the protocol spec, then how to do with python sdk

    a. If the client wishes to resume after a broken connection, it **SHOULD** issue an HTTP GET to the MCP endpoint, and include the `Last-Event-ID` header to indicate the last event ID it received.

    – The server **MAY** use this header to replay messages that would have been sent after the last event ID, *on the stream that was disconnected*, and to resume the stream from that point.

    – The server **MUST NOT** replay messages that would have been delivered on a different stream.

Paginating Results

- Supporting pagination for paginated results
    - which results are paginated, how to get the cursor (a PaginatedResult has a nextCursor property), how to use it (list methods get a cursor parameter)

Client Info

- Setting client_info in the Session constructor (not an established best practice but worth using)

About the Author

Kyle Stratis is the founder and chief consultant of Stratis Data Labs, where he provides startups and large firms alike with his expertise in AI engineering, computer vision, data engineering, and machine learning platform engineering garnered over a decade as a software engineer. In addition to designing data architecture, building AI platforms and agents, and migrating analytics workflows to the cloud for his clients, he writes about Python, LLMs, and personal knowledge management.