Chapter 4. Tool Use

While foundation models are great at chatting for hours, tools are the building blocks that empower AI agents to retrieve additional information and context, perform tasks, and interact with the environment in meaningful ways. In the context of AI, a tool can be defined as a specific capability or a set of actions that an agent can perform to achieve a desired outcome. These tools range from simple, single-step tasks to complex, multistep operations that require advanced reasoning and problem-solving abilities. Especially if you want your agent to make actual changes, instead of just searching for and providing information, tools will be how those changes are executed.

The significance of tools in AI agents parallels the importance of competencies in human professionals. Just as a doctor needs a diverse set of tools to diagnose and treat patients, an AI agent requires a repertoire of tools to handle various tasks effectively. This chapter aims to provide a comprehensive understanding of tools in AI agents, exploring their design, development, and deployment.

AI agents, at their core, are sophisticated systems designed to interact with their environment, process information, and execute tasks autonomously. To do this efficiently, they rely on a structured set of tools. These tools are modular components that can be developed, tested, and optimized independently, then integrated to form a cohesive system capable of complex behavior.

In practical terms, a tool could be as simple as recognizing an object in an image or as complex as managing a customer support ticket from initial contact to resolution. The design and implementation of these tools are critical to the overall functionality and effectiveness of the AI agent. We'll start with some fundamentals of LangChain, and then cover the different types of tools that can be provided to an autonomous agent, which we will cover in sequence: local tools, API-based tools, and MCP tools.

LangChain Fundamentals

Before diving deeper into tool selection and orchestration, it is helpful to understand some core LangChain concepts. At the heart of LangChain are foundation models and chat models, which process prompts and generate responses. For example, `ChatOpenAI` is a wrapper class that provides a simple interface to interact with OpenAI's chat-based models like GPT-5. You initialize it with parameters such as the model name to specify which model to use:

```
from langchain_openai import ChatOpenAI
llm = ChatOpenAI(model_name="gpt-4o")
```

LangChain structures interactions as messages to maintain conversational context. The two main message types are `HumanMessage`, which represents user inputs, and `AIMessage`, which represents the model's responses:

```
from langchain_core.messages import
HumanMessage messages = [HumanMessage("What is the weat
```

Tools, meanwhile, are external functions that your model can call to extend its capabilities beyond text generation—for instance, calling APIs, retrieving database entries, or performing calculations. You define a tool in LangChain using the `@tool` decorator, which registers the function and automatically generates the schema describing its inputs and outputs:

```
from langchain_core.tools import tool

@tool
def add_numbers(x: int, y: int) -> int:
    """Adds two numbers and returns the sum."""
    return x + y
```

Once you have defined your tools, you bind them to the model using `.bind_tools()`, which enables the model to select and invoke these tools in response to user inputs. To interact with the model, you use the `.invoke()` method, providing it with a list of messages representing the current conversation. If the model decides to call a tool, it will output a tool

call, which you then execute by invoking the corresponding function and appending its result back into the conversation before generating the final response:

```
llm_with_tools = llm.bind_tools([add_numbers])
ai_msg = llm_with_tools.invoke(messages)
for tool_call in ai_msg.tool_calls:
    tool_response = add_numbers.invoke(tool_call)
```

These building blocks—chat models, messages, tools, and tool invocation—form the foundation of LangChain-based systems. Understanding how they fit together will help you follow the examples in this chapter and build your own agents that can seamlessly integrate language understanding with real-world actions.

Local Tools

These tools are designed to run locally. They are often based on predefined rules and logic, tailored to specific tasks. These local tools can be easily built and modified, and are co-deployed with the agent. They can especially augment weaknesses in language models that traditional programming techniques perform better at, such as arithmetic, time-zone conversions, calendar operations, or interactions with maps. These local tools offer precision, predictability, and simplicity. As the logic is explicitly defined, local tools tend to be predictable and reliable.

The metadata—the tool's name, description, and schema—is just as critical as its logic. The model uses that metadata to decide which tool to invoke. Therefore, the following is important:

- Choose precise, narrowly scoped names. If your name is too general, the LLM may call it when it's not needed.
- Write clear, distinctive descriptions. Overly broad or overlapping descriptions across multiple tools guarantee confusion and poor performance.
- Define strict input/output schemas. Explicit schemas help the foundation model understand exactly when and how to use the tool, reducing misfires.

Despite these benefits, local tools have some important drawbacks:

Scalability

Designing, building, and deploying local tools can be cumbersome, time-consuming, and challenging, and local tools are harder to share across use cases. While tools can be exposed as libraries and shared across multiple agent use cases, this can be challenging in practice and at scale.

Duplication

Every team or agent deployment that wants to use local tools will need to deploy the same library along with their agent service, and pushing changes to these tools will require coordinating deployments to each agent service that uses these tools. In practice, many teams simply reimplement the same tools independently to avoid the coordination overhead.

Maintenance

As the environment or requirements change, handcrafted tools may need frequent updates and adjustments. This ongoing maintenance can be resource-intensive and typically requires a redeployment of your agent service.

Despite these drawbacks, manually crafted tools are especially useful in addressing areas of traditional weakness for foundation models. Simple mathematical operations are a great example of this. Unit conversions, calculator operations, calendar changes, operations on dates and times, and operations over maps and graphs, for example, are all areas where handcrafted tools can substantially improve the efficacy of agentic systems.

Let's look at an example of registering a calculator tool. First, we define our simple calculator function:

```
from langchain_core.runnables import ConfigurableField
from langchain_core.tools import tool
from langchain_openai import ChatOpenAI

# Define tools using concise function definitions
@tool
def multiply(x: float, y: float) -> float:
   """Multiply 'x' times 'y'."""
   return x * y
```

```
@tool
def exponentiate(x: float, y: float) -> float:
    """Raise 'x' to the 'y'."""
    return x**y

@tool
def add(x: float, y: float) -> float:
    """Add 'x' and 'y'."""
    return x + y
```

Then, we bind the tool with the foundation model in LangChain:

```
tools = [multiply, exponentiate, add]

# Initialize the LLM with GPT-4o and bind the tools
llm = ChatOpenAI(model_name="gpt-4o", temperature=0)
llm_with_tools = llm.bind_tools(tools)
```

This “binding” operation registers the tool. Under the hood, LangChain will now check if the foundation model response includes any requests to call a tool. Now that we’ve bound the tool, we can ask the foundation model questions, and if the tool is helpful for answering the question, the foundation model will choose the tools, select the parameters for those tools, and invoke those functions:

```
query = "What is 393 * 12.25? Also, what is 11 + 49?"
messages = [HumanMessage(query)]

ai_msg = llm_with_tools.invoke(messages)
messages.append(ai_msg)
for tool_call in ai_msg.tool_calls:
    selected_tool = {"add": add, "multiply": multiply,
        "exponentiate": exponentiate}[tool_call["name"].
    tool_msg = selected_tool.invoke(tool_call)

    print(f’{tool_msg.name} {tool_call['args']} {tool_ms
    messages.append(tool_msg)
    final_response = llm_with_tools.invoke(messages)
```

```
print(final_response.content)
```

With those added print statements for visibility, we can see that the foundation model invokes two function calls—one each for `multiply` and `add`:

```
multiply {'x': 393, 'y': 12.25} Result: 4814.25
add {'x': 11, 'y': 49} 60.0
```

The model will then include this result from the tool call in the generated final response, producing a result such as:

```
393 times 12.25 is 4814.25, and 11 + 49 is 60.
```

While the effect of this is simple, the implications are profound. The foundation model is now able to execute the computer programs that we bind with it. This is a simple example, but we can bind arbitrarily useful and consequential programs to the foundation model, and we now rely on the foundation model to choose which programs to execute with which parameters. Doing so responsibly, and only binding tools that the foundation model will execute in ways that produce more good than harm, is among the paramount responsibilities of developers building agents and agentic systems.

API-Based Tools

API-based tools enable autonomous agents to interact with external services, enhancing their capabilities by accessing additional information, processing data, and executing actions that are not feasible to perform locally. These tools leverage application programming interfaces (APIs) to communicate with public or private services, providing a dynamic and scalable way to extend the functionality of an agent.

API-based tools are particularly valuable in scenarios where the agent needs to integrate with various external systems, retrieve real-time data, or perform complex computations that would be too resource-intensive to handle internally. By connecting to APIs, agents can access a vast array of services, such as weather information, stock market data, translation services, and more, enabling them to provide richer and more accurate responses to user queries. These API-based tools have multiple benefits.

By leveraging external services, these tools can dramatically expand the range of tasks an agent can perform. For instance, an agent can use a weather API to provide current weather conditions and forecasts, a financial API to fetch stock prices, or a translation API to offer multilingual support. This ability to integrate diverse external services greatly broadens the agent's functionality, all without having to retrain a model.

Real-time data access is another major benefit of API-based tools. APIs enable agents to access the most current information from external sources, ensuring that their responses and actions are based on up-to-date data. This is particularly crucial for applications that depend on timely and accurate information, such as financial trading or emergency response systems, where decisions must be made quickly based on the latest available data.

To illustrate the implementation of API-based tools, let's begin with enabling your agent to browse the open web for additional information. In this code snippet, we register a tool to retrieve information from Wikipedia, a step toward a full web browsing agent:

```
from langchain_openai import ChatOpenAI
from langchain_community.tools import WikipediaQueryRur
from langchain_community.utilities import WikipediaAPIW
from langchain_core.messages import HumanMessage

api_wrapper = WikipediaAPIWrapper(top_k_results=1, doc_
tool = WikipediaQueryRun(api_wrapper=api_wrapper)

# Initialize the LLM with GPT-4o and bind the tools
llm = ChatOpenAI(model_name="gpt-4o", temperature=0)
llm_with_tools = llm.bind_tools([tool])

messages = [HumanMessage("What was the most impressive
                         "about Buzz Aldrin?")]

ai_msg = llm_with_tools.invoke(messages)
messages.append(ai_msg)

for tool_call in ai_msg.tool_calls:
    tool_msg = tool.invoke(tool_call)

    print(tool_msg.name)
    print(tool_call['args'])
    print(tool_msg.content)
```

```
        messages.append(tool_msg)
        print()

    final_response = llm_with_tools.invoke(messages)
    print(final_response.content)
```

The foundation model identifies the object of interest in the query and searches Wikipedia for the term. It then uses this additional information to generate its final answer when addressing the question:

```
{'query': 'Buzz Aldrin'}
Page: Buzz Aldrin
Summary: Buzz Aldrin (born Edwin Eugene Aldrin Jr. Janu
American former astronaut, engineer and fighter pilot.
as pilot of the 1966 Gemini 12 mission, and was the Lur
the 1969 Apollo 11 mission.

One of the most impressive things about Buzz Aldrin is
Module Eagle pilot on the 1969 Apollo 11 mission, makir
two humans to land on the Moon. This historic event mar
achievement in space exploration and human history. Adc
three spacewalks as pilot of the 1966 Gemini 12 missior
and contributions to advancing space travel.
```

Let's now look at a second example, for an agent that is designed to fetch and display stock market data. This process involves defining the API interaction, handling the response, and integrating the tool into the agent's workflow. By following this approach, agents can integrate external data sources seamlessly, enhancing their overall functionality and effectiveness.

First, we define the function that interacts with the stock market API. Then, we register this function as a tool for our agent, and we can then invoke it just like the previous tools:

```
from langchain_core.tools import tool
from langchain_openai import ChatOpenAI
from langchain_community.tools import WikipediaQueryRur
from langchain_community.utilities import WikipediaAPIW
from langchain_core.messages import HumanMessage
import requests

@tool
```

```
def get_stock_price(ticker: str) -> float:
    """Get the stock price for the stock exchange ticker
    api_url = f"https://api.example.com/stocks/{ticker}'
    response = requests.get(api_url)
    if response.status_code == 200:
        data = response.json()
        return data["price"]
    else:
        raise ValueError(f"Failed to fetch stock price f


# Initialize the LLM with GPT-4o and bind the tools
llm = ChatOpenAI(model_name="gpt-4o", temperature=0)
llm_with_tools = llm.bind_tools([get_stock_price])

messages = [HumanMessage("What is the stock price of Ap

ai_msg = llm_with_tools.invoke(messages)
messages.append(ai_msg)

for tool_call in ai_msg.tool_calls:
    tool_msg = get_stock_price.invoke(tool_call)

    print(tool_msg.name)
    print(tool_call['args'])
    print(tool_msg.content)
    messages.append(tool_msg)
    print()

final_response = llm_with_tools.invoke(messages)
print(final_response.content)
```

Similar tools can be created to search across team- or company-specific information. By providing your agent with the tools necessary to access the information it needs to handle a task, and the specific tools to operate over that information, you can significantly expand the scope and complexity of tasks that can be automated.

When designing API tools for agents, focus on reliability, security, and graceful failure. External services can go down, so agents need fallbacks or clear error messages. Secure all communications with HTTPS and strong authentication, especially for sensitive data.

Watch out for API rate limits to avoid disruptions, and ensure compliance with data privacy laws—anonymize or obfuscate user data when needed. Handle errors robustly so the agent can recover from network issues or invalid responses without breaking the user experience. When possible, consider alternatives and multiple providers for greater reliability if any given provider is degraded.

APIs empower agents with real-time data, heavy computation, and external actions they couldn't perform alone, making them far more capable and effective.

Plug-In Tools

These tools are modular and can be integrated into the AI agent's framework with minimal customization. They leverage existing libraries, APIs, and third-party services to extend the agent's capabilities without extensive development effort. Plug-in tools enable rapid deployment and scaling of the agent's functionalities. These tools are predesigned modules that can be integrated into an AI system with minimal effort, leveraging existing libraries, APIs, and third-party services. The integration of plug-in tools has become a standard offering from leading platforms such as OpenAI, Anthropic's Claude, Google's Gemini, and Microsoft's Phi as well as a growing open source community. Plug-in tools provide powerful tools to expand the capabilities of AI agents without extensive custom development.

OpenAI's plug-ins ecosystem offers powerful extensions—everything from real-time web search to specialized code generators—but they're only available inside the ChatGPT product, not the public API. You cannot invoke Expedia, Zapier, or any first-party ChatGPT plug-in through the standard OpenAI Completions or Chat endpoints. To replicate similar behavior in your own applications, you must build custom function-calling layers (for example, via LangChain) that approximate plug-in functionality.

Anthropic's Claude, by contrast, exposes its full "tool use" capability directly through the Anthropic Messages API (and on platforms like Amazon Bedrock or Google Cloud's Vertex AI). You simply register your custom tools (or use Anthropic-provided ones), and Claude can call them at inference time—no separate UI required. This API-first approach makes it straightforward to integrate content moderation, bias detection, or domain-specific services into any Claude-powered workflow.

Google's Gemini models support function calling via the Vertex AI API, letting you declare tools in a `FunctionCallingConfig` and have Gemini invoke them as structured calls. Whether you need natural language understanding, image recognition, or database lookups, you define the functions up front and process the returned arguments in your code—no proprietary UI layer stands between your app and the model.

Microsoft's Phi models are offered through Azure AI Foundry, where they integrate seamlessly with other Azure services—such as cognitive search, document processing, and data visualization APIs—via the same public endpoints you use for other Azure AI models. Though not branded as "plug-ins," Phi's tight coupling with Azure's productivity and analytics tools delivers a similarly smooth experience: you call the model, receive structured outputs, and feed them directly into your existing Azure workflows without switching contexts.

One of the significant advantages of plug-in tools is their integration at the model execution layer. This means these tools can be added to AI models with minimal disruption to existing workflows. Developers can simply plug these modules into their AI systems, instantly enhancing their capabilities without extensive customization or development effort. This ease of integration makes plug-in tools an attractive option for rapidly deploying new functionalities in AI applications. However, this ease of use comes with certain limitations. Plug-in tools, while powerful, do not offer the same level of customizability and adaptability as custom-developed tools that can be served either locally or remotely. They are designed to be general-purpose tools that can address a broad range of tasks, but they may not be tailored to the specific needs and nuances of every application. This trade-off between ease of integration and customizability is an important consideration for developers when choosing between plug-in tools and bespoke development.

Despite the current limitations, the catalogs of plug-in tools offered by leading platforms are rapidly growing. As these catalogs expand, the breadth of capabilities available through plug-in tools will increase, providing developers with even more tools to enhance their AI agents. This growth is driven by continuous advancements in AI research and the development of new techniques and technologies. In the near future, we can expect these plug-in tool catalogs to include more specialized and advanced functionalities, catering to a wider range of applications and industries. This expansion will facilitate agent development by providing developers with readily available

tools to address complex and diverse tasks. The growing ecosystem of plug-in tools will enable AI agents to perform increasingly sophisticated functions, making them more versatile and effective in various domains.

In addition to the offerings from major platforms, there is a rapidly growing ecosystem of tools that can be incorporated into open source foundation models. This ecosystem provides a wealth of resources for developers looking to enhance their AI agents with advanced capabilities. Open source communities are actively contributing to the development of plug-in tools, creating a collaborative environment that fosters innovation and knowledge sharing. One notable example is the Hugging Face Transformers library, which offers a wide range of pretrained models and plug-in tools for natural language processing tasks. These tools can be easily integrated into open source foundation models, enabling functionalities such as text generation, sentiment analysis, and language translation. The open source nature of this library enables developers to customize and extend these tools to suit their specific needs. The flexibility of these frameworks means that developers can combine plug-in tools with custom development, creating powerful and adaptable AI systems. The open source AI community is continuously contributing new plug-in tools and enhancements, driven by the collective efforts of researchers, developers, and enthusiasts. Platforms like Glama.ai, and mcp.so aggregate large numbers of MCP servers, making them searchable and discoverable, ranging from simple utilities to complex, stateful services. These contributions enrich the ecosystem and provide valuable resources for developers looking to leverage the latest advancements in AI.

The practical applications of plug-in tools are vast and varied, spanning multiple industries and use cases. By integrating plug-in tools, developers can create AI agents that perform a wide range of tasks efficiently and effectively. In customer support, plug-in tools can enable AI agents to handle queries, provide solutions, and manage support tickets. Tools like natural language understanding and sentiment analysis can help AI agents understand customer issues and respond appropriately, improving customer satisfaction and reducing response times. In healthcare, plug-in tools can assist AI agents in tasks such as medical image analysis, patient triage, and data management. Tools that leverage computer vision can help identify abnormalities in medical images, while natural language processing tools can assist in managing patient records and extracting relevant information from medical literature, and vector search tools can offer grounding in relevant documents to address the current query. In the finance industry, plug-in tools can enhance AI agents' abilities to

analyze market trends, detect fraudulent activities, and manage financial portfolios. Tools like anomaly detection and predictive analytics can provide valuable insights and improve decision-making processes. In education, plug-in tools can support AI agents in personalized learning, automated grading, and content recommendation.

The future of plug-in tools in AI development looks promising, with continuous advancements and growing adoption across various industries. As the capabilities of plug-in tools expand, we can expect AI agents to become even more capable and versatile. The ongoing research and development efforts by leading platforms and the open source community will drive innovation, resulting in more powerful and sophisticated tools for AI development. One important area of focus for the future is the interoperability and standardization of plug-in tools. Establishing common standards and protocols for plug-in tools will facilitate seamless integration and interoperability across different AI platforms and systems. This will enable developers to leverage plug-in tools from various sources, creating more flexible and adaptable AI solutions. Efforts are also being made to enhance the customization and adaptability of plug-in tools. Future plug-in tools may offer more configurable options, enabling developers to tailor them to specific use cases and requirements. This will bridge the gap between the ease of integration and the need for customized solutions, providing the best of both worlds.

Model Context Protocol

As the AI ecosystem matures, agents no longer live in isolated silos. They need to read documents from cloud storage, push data to business applications, call internal APIs, and coordinate with other agents. Custom integrations—where you write bespoke adapters for each data source or service—are brittle and scale poorly. Enter the Model Context Protocol (MCP): an open standard introduced by Anthropic (and since adopted by major players like OpenAI, Google DeepMind, and Microsoft) that provides a uniform, model-agnostic way to connect LLMs to external systems. Think of MCP as a “USB-C port for AI”—a single, well-defined interface that any data source or tool can expose, and any agent can consume, without specialized glue code. At its core, MCP defines two roles:

MCP server

This is a web server that exposes data or services via a standardized JSON-RPC 2.0 interface. A server can wrap anything—cloud object storage, SQL databases, enterprise customer relationship management, proprietary business logic—so long as it implements the MCP specification.

MCP client

This is any agent or LLM application that "speaks" MCP. The client sends JSON-RPC requests (e.g., "List all files in this Salesforce folder," or "Execute function 'getCustomerBalance' with customerId=1234") and receives structured JSON responses. Because the protocol is uniform, an agent developer doesn't need to know the internals of the server—only its exposed methods.

Under the hood, MCP uses JSON-RPC 2.0 over HTTPS or WebSocket. Servers advertise their available methods (e.g., `listFiles`, `getRecord`, `runAnalysis`) and their input/output schemas. Clients fetch the server's "method catalog," allowing an LLM to reason about which method to call and with what parameters. Once the tool call is chosen, the MCP client wraps that call into a JSON-RPC payload, sends it over to the appropriate server, and awaits a response. Because both ends speak the same language, building cross-platform interoperability becomes straightforward.

Before MCP, developers wrote custom adapters for each target system—hard-coding REST calls or SDK usage directly inside their agent code. As the number of data sources grew, these bespoke integrations multiplied, resulting in brittle, error-prone code that was difficult to maintain or extend.

Despite these advantages, several security issues have been raised and are not yet fully addressed—particularly around authentication, access controls, and potential attack vectors when multiple agents share MCP endpoints. Ensuring that only authorized agents invoke specific methods, maintaining role-based access control to sensitive data, preventing malicious payload injection, and maintaining audit logs remain active areas of research and engineering. Some organizations still rely on additional network policies or proxy layers to mitigate these risks, but the core MCP specification does not yet mandate a single, standardized security solution. Nevertheless, MCP solves a critical challenge of tool reuse across multiple agents: once a service is exposed via MCP, any number of agents can discover and invoke its methods without

rewriting custom adapters for each agent. This dramatically reduces development effort and encourages modular, reusable architectures.

To see MCP in action, we'll walk through a self-contained Python example that does the following:

1. Launches a local "math" MCP server (via a subprocess)
2. Connects to a remote "weather" MCP server running on *localhost:8000/mcp*
3. Implements an asynchronous agent loop that inspects the user's last message and decides whether to call the "math" tool (for arithmetic expressions) or the "weather" tool (for weather queries)
4. Demonstrates how the agent parses the tool's output and returns a final assistant response

Here's the complete Python implementation demonstrating these steps:

```
class AgentState(TypedDict):
    messages: Sequence[Any]  # A list of BaseMessage/Hu

mcp_client = MultiServerMCPClient(
    {
        "math": {
            "command": "python3",
            "args": ["src/common/mcp/MCP_weather_server
            "transport": "stdio",  # Subprocess → STDIC
        },
        "weather": {
            # Assumes a separate MCP server is already
            "url": "http://localhost:8000/mcp",
            "transport": "streamable_http",
            # HTTP→JSON-RPC over WebSocket/stream
        },
    }
)

async def get_mcp_tools() -> list[Tool]:
    return await mcp_client.get_tools()

async def call_mcp_tools(state: AgentState) -> dict[str
    messages = state["messages"]
    last_msg = messages[-1].content.lower()
```

```
        # Fetch and cache MCP tools on the first call
        global MCP_TOOLS
        if "MCP_TOOLS" not in globals():
            MCP_TOOLS = await mcp_client.get_tools()

        # Simple heuristic: if any digit-operator token app
        if any(token in last_msg for token in ["+", "-", "*
            tool_name = "math"
        elif "weather" in last_msg:
            tool_name = "weather"
        else:
            # No match → respond directly
            return {
                "messages": [
                    {
                        "role": "assistant",
                        "Sorry, I can only answer math" +
                            " or weather queries."
                    }
                ]
            }

        tool_obj = next(t for t in MCP_TOOLS if t.name == t

        user_input = messages[-1].content
        mcp_result: str = await tool_obj.arun(user_input)

        return {
            "messages": [
                {"role": "assistant", "content": mcp_result
            ]
        }
```

The `"math"` entry uses `command` + `args` to spawn a subprocess that runs *MCP_weather_server.py*. Under the hood, this script must conform to MCP (i.e., serve JSON-RPC over STDIO).

The `"weather"` entry points to an already running HTTP MCP server at *http://localhost:8000/mcp*. The `streamable_http` transport allows duplex JSON-RPC communication over HTTP/WebSocket.

MCP represents a significant step forward in how we design, deploy, and maintain AI agents at scale. By defining a single, standardized JSON-RPC interface for exposing and consuming methods, MCP decouples service

implementation from agent logic, enabling any number of agents to reuse the same tools without bespoke integrations. In practice, this means that as new data sources, microservices, or legacy systems emerge, developers need only implement an MCP-compliant server once—and any MCP-capable agent can discover and invoke its methods immediately.

Although security concerns like robust authentication, fine-grained access control, and payload validation remain active areas of development, the core promise of MCP—seamless interoperability and modular tool reuse—has already been realized in production systems across leading organizations. Looking ahead, we expect continued refinement of MCP's security best practices, broader adoption of standardized method catalogs, and the growth of an ecosystem of public and private MCP endpoints. In sum, MCP solves one of the most persistent challenges in agentic system design—how to integrate diverse services quickly and reliably—while laying a foundation for ever more flexible, maintainable, and distributed AI architectures.

Stateful Tools

Stateful tools span local scripts, external APIs, and MCP-deployed services, yet they all share a common risk: when you hand a foundation model direct power over persistent state, you also empower it to make destructive mistakes or to be exploited by bad actors. In one real-world case, an AI agent "optimized" database performance by dropping half the rows from a production table, erasing critical records in the process. Even without malice, foundation models can misinterpret a user's intent, turning what should be a harmless query into a destructive command. This risk is especially acute for stateful tools because they interact with live data stores whose contents change over time.

To mitigate these dangers, register only narrowly scoped operations as tools instead of exposing an "execute arbitrary SQL" endpoint. For example, define a `get_user_profile(user_id)` tool or an `add_new_customer(record)` tool, each encapsulating a single, well-tested query or procedure. Agents needing only read access should never receive rights to delete or modify data. By constraining tool capabilities at the registration layer, you sharply reduce the attack surface and limit the scope of potential errors.

If your use case absolutely demands free-form queries, you must implement rigorous sanitization and access controls. OWASP's GenAI Security Project warns that prompt injections can slip dangerous clauses like `DROP` or `ALTER` into otherwise benign requests, so input validation must reject any statement containing these patterns. Always bind parameters or use prepared statements to prevent SQL injection, and ensure the database account used by the agent holds only the minimum privileges needed to execute the allowed queries.

Beyond sanitization, logging every tool invocation to detect anomalous behavior and support forensic analysis is highly recommended. Coupled with real-time alerts for suspicious patterns—such as unusually large deletions or schema-altering commands—you can intervene quickly before small errors cascade into major incidents.

Ultimately, the principle of least power should guide your design: give the model only the tools it strictly requires, and guard every operation with precise boundaries and oversight. Whether your tool runs locally, calls an external API, or executes on an MCP server, the same safeguards apply—restrict capabilities, sanitize inputs, enforce least privilege, and maintain full observability. By treating stateful tools with this level of discipline, you ensure that your AI agents remain powerful collaborators rather than uncontrolled database administrators.

Automated Tool Development

Code generation is a technique where AI agents write code autonomously, significantly reducing the time and effort required to create and maintain software applications. This process involves training models on vast amounts of code data, enabling them to understand programming languages, coding patterns, and best practices.

Code generation represents a transformative leap in AI capabilities, particularly when an agent writes its own tools in real time to solve tasks or interact with new APIs. This dynamic approach enables AI agents to adapt and expand their functionality, significantly enhancing their versatility and problem-solving capacity.

Foundation Models as Tool Makers

Foundation models no longer just consume tools—they build them. By feeding an LLM your API specifications or sample inputs, you can have it generate initial wrappers, helper functions, or higher-level "atomic" operations. Let the model draft code stubs, execute them in a safe sandbox, and then critique its own output: "That endpoint returned a 400—adjust the query parameters." Over a few rapid iterations, you end up with a suite of well-tested, narrowly scoped tools that agents can call directly, without crafting every wrapper by hand.

This approach shines when you're wrestling with a sprawling API landscape. Instead of manually writing dozens of microservice clients, you point the model at your OpenAPI spec (or code samples) and let it spin up a first draft of each function. Human reviewers then validate and tighten the generated code before it enters your continuous integration/continuous deployment (CI/CD) pipeline, ensuring security and correctness. As your APIs evolve, you simply rerun the same generate-and-refine loop to keep your tools in sync—saving weeks of boilerplate work and avoiding brittle, handwritten glue code.

While foundation-driven tool creation slashes development time and scales effortlessly, it still demands clear validation criteria (tests, response checks, schema enforcement) and developer oversight. The model's natural language critiques make it easy to understand any recommended fixes, but you're ultimately responsible for catching edge cases, guarding against security gaps, and confirming business logic alignment. When done right, this hybrid of AI creativity and human review transforms a tangled API ecosystem into a lean, agent-ready toolkit—unlocking rapid, reliable automation across your organization.

Real-Time Code Generation

Real-time code generation involves an AI agent writing and executing code as needed during its operation. This capability enables the agent to create new tools or modify existing ones to address specific tasks, making it highly adaptable. For instance, if an AI agent encounters a novel API or an unfamiliar problem, it can generate code to interface with the API or develop a solution to the problem in real time.

The process begins with the agent analyzing the task at hand and determining the necessary steps to accomplish it. Based on its understanding, the agent writes code snippets, which it then attempts to execute. If the code does not perform as expected, the agent iteratively revises it, learning from each attempt until it achieves the desired outcome. This iterative process of trial and error enables the agent to refine its tools continuously, improving its performance and expanding its capabilities autonomously.

Real-time code generation offers several compelling advantages, particularly in terms of adaptability and efficiency. The ability to generate code on-the-fly enables AI agents to quickly adapt to new tasks and environments. This adaptability is crucial for applications requiring dynamic problem-solving and flexibility, such as real-time data analysis and complex software integration tasks. By generating code in real time, AI agents can address immediate needs without waiting for human intervention, significantly speeding up processes, reducing downtime, and enhancing overall efficiency.

However, real-time code generation also presents several challenges and risks. Quality control is a major concern, as ensuring the quality and security of autonomously generated code is critical. Poor-quality code can lead to system failures, security breaches, and other significant issues. Security risks are another major challenge, as allowing AI agents to execute self-generated code introduces the potential for malicious actors to exploit this capability to inject harmful code, leading to data breaches, unauthorized access, or system damage. Implementing robust security measures and oversight is essential to mitigate these risks.

A less obvious but critical drawback is repeatability. When your agent recreates tools from scratch each time, you lose predictability—success for one invocation doesn’t guarantee success for the next. Performance can fluctuate wildly, and subtle changes in prompts or model updates can lead to entirely different code paths. This instability complicates debugging, testing, and compliance, making it hard to certify that your agent will always behave as expected.

Resource consumption is also a critical consideration, as real-time code generation and execution can be resource-intensive, requiring substantial computational power and memory, especially when naive or inefficient solutions are drafted and executed. Placing guardrails on multiple aspects of system performance can help to mitigate these risks.

Tool Use Configuration

Foundation model APIs from OpenAI, Anthropic, Gemini, and more let you explicitly control the model's use of tools via a tool-choice parameter—shifting from flexible foundation model–driven invocation to deterministic behavior. In "auto" mode, the model decides whether to call tools based on context; this is good for general use. In contrast, "any"/"required" forces the model to invoke at least one tool, ideal when tool output is essential. Setting these parameters to "none" blocks all tool calls—useful for controlled outputs or testing environments. Some interfaces even let you pin a specific tool, ensuring predictable, repeatable flows. By choosing the appropriate mode, you decide whether to let the foundation model manage tasks flexibly or impose structure—balancing flexibility, reliability, and predictability.

Even the best agents can misstep—skipping necessary tool calls, outputting invalid JSON, or running tools that error out—so you need reliable fallback and postprocessing mechanisms in place. After every model response, inspect whether it invoked the right tools, produced valid JSON, and succeeded without runtime errors. If anything breaks, respond with a corrective flow:

- Validate first using your schema (e.g., via jsonschema or Pydantic). This catches missing fields or malformed structures. If a tool was skipped, trigger it automatically; if the JSON is invalid, prompt the model to correct it.
- Retry intelligently, using structured logic such as exponential backoff for transient failures, or regenerate only the problematic portion instead of restarting the whole exchange.
- Fall back gracefully when retries fail. Options include switching to a backup model or service, asking the user for clarification, using cached data, or returning a safe default.
- Log everything—prompts, tool calls, validation errors, retries, fallbacks—for observability, debugging, and continuous improvement.

By validating outputs, retrying strategically, and falling back gracefully—all while logging every step—you transform random failures into manageable, predictable behavior. This shift is essential for delivering robust, production-grade agents.

Conclusion

Tools enable AI agents to perform tasks, make decisions, and interact with their environment effectively. These range from simple to complex tasks requiring advanced reasoning. Handcrafted tools, manually designed by developers, offer precision but can be time-consuming to maintain. Plug-in tools, provided by platforms like OpenAI and Google's Gemini, enable rapid integration and scalability but lack customizability.

Automated tool development, including real-time code generation, imitation learning, and reinforcement learning, allows AI agents to dynamically adapt and refine their abilities. This enhances their versatility and problem-solving capabilities, enabling continuous improvement and autonomous expansion of tools. Building and maintaining the toolkit for your agent is one of the most critical ways to give your agent the capabilities to succeed in the task at hand.

Now that we know how to build and curate a set of tools that we provide to our agent, we'll move on to consider how we'll enable the agent to make plans, select and parameterize tools, and put these pieces together to perform useful work. In the next chapter, we'll discuss how we can organize a sequence of tools to perform complex tasks in a process we call orchestration.