5
Enabling Tool Use and Planning in Agents

In the previous chapter, we looked into the intricate concepts of reflection and introspection in intelligent agents. These capabilities empower agents to reason about their own cognitive processes, learn from experiences, and dynamically modify their behaviors.

A significant step forward in AI agents comes from combining how agents plan and use tools. This chapter looks at how tools work, different planning algorithms, how they fit together, and real examples showing how useful they are in practice. We will explore the concept of tool use by intelligent agents that extend their capabilities beyond decision-making and problem-solving. We will look at different types of tools that agents can utilize, such as APIs, databases, and software functions. We will then delve into planning algorithms essential for agents, including state-space search, reinforcement learning, and hierarchical task network planning. We will discuss integrating tool use and planning by reasoning about available tools, assessing their suitability based on goals, selecting appropriate tools, and generating efficient action sequences that leverage those tools.

This chapter is divided into the following main sections:

- Understanding the concept of tool use in agents
- Planning algorithms for agents
- Integrating tool use and planning
- Exploring practical implementations

By the end of this chapter, you will know what tools are, how they can be used to power your agentic systems, and how they work in conjunction with planning algorithms.

Technical requirements

You can find the code file for this chapter on GitHub at `https://github.com/PacktPublishing/Building-Agentic-AI-Systems`. In this chapter, we will also use agentic Python frameworks such as CrewAI, AutoGen, and LangChain to demonstrate the various aspects of AI agents.

Understanding the concept of tool use in agents

At its core, tool usage by an intelligent agent refers to the LLM agent's capability of leveraging external resources or instrumentation to augment the agent's inherent functionality and decision-making processes. This concept extends beyond the traditional notion of an agent as a self-contained (isolated) entity, relying solely on its internal knowledge (training data) and algorithms. Instead, it acknowledges the potential for agents to transcend their intrinsic limitations by strategically harnessing the power of external tools and systems.

For example, when you send a query ("*What's the weather?*") to an agent in isolation, the model is free to either respond with any made-up answer or it may respond that it doesn't know how to find the weather. In this case, the agent will rely on the LLM's training data, which will not have up-to-date information about real-time weather data. On the other hand, if the LLM agent has access to a real-time weather lookup tool, it may be able to answer the question accurately. Tool usage enables agents to access real-time data, execute specialized tasks, and manage complex workflows that go beyond their built-in knowledge and algorithms. *Figure 5.1* shows this isolated versus tool-powered behavior:

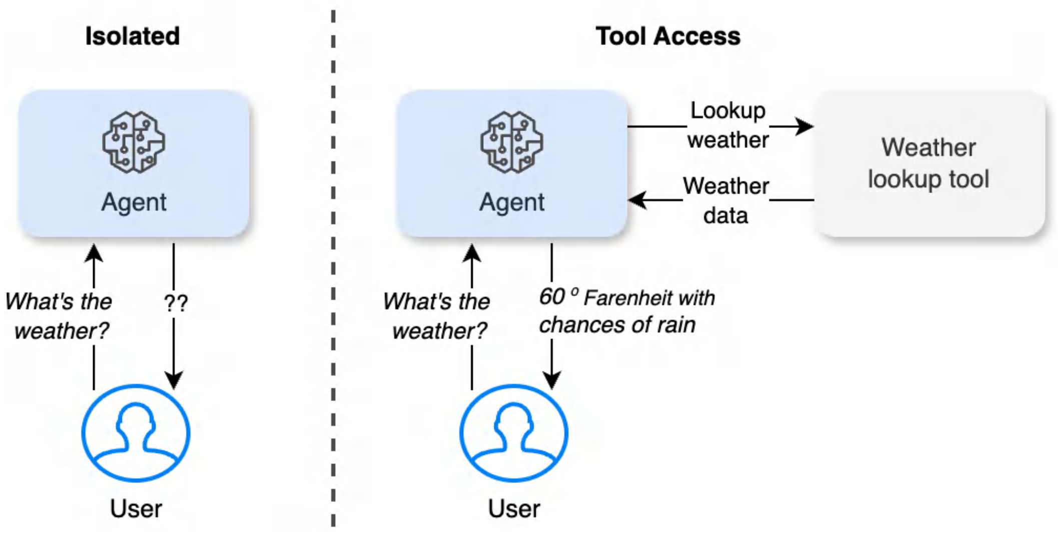

Figure 5.1 – Agent behavior in isolation versus with access to a tool

The significance of tool use lies in its ability to broaden the scope of an agent's (and, in turn, the LLM that powers the agent's) competencies, enabling it to tackle complex, real-world challenges that may be beyond the reach of its native problem-solving capabilities. By integrating and orchestrating the use of various tools, an agent can effectively offload specific tasks or access supplementary data and functionalities, thereby enhancing its overall performance and expanding its scope of achievable objectives. Before we go into the details of tools, let's first understand how LLM tool calling works.

Tool and function calling

While *tool calling* and *function calling* are often used interchangeably in the context of LLMs, they have distinct technical differences. **Function calling** refers to an LLM generating structured calls to predefined functions within the same runtime, typically executing internal tasks such as database lookups or calculations. **Tool calling**, on the other hand, enables LLMs to interact with external APIs, services, or systems, allowing them to access real-time data and perform specialized tasks beyond their intrinsic capabilities. For example, an LLM using function calling might retrieve a user's profile from a local database, while tool calling would involve querying a weather API for live updates. Understanding this distinction is crucial for designing AI agents that seamlessly integrate internal logic with external systems to enhance functionality.

When an LLM invokes a tool or function, it doesn't actually execute any code. Instead, it generates a structured response indicating the following:

- Which tool/function it wants to use
- What parameters should be passed to that tool/function
- How those parameters should be formatted

Think of it like writing a detailed instruction rather than performing the action itself. The LLM acts as a sophisticated dispatcher, determining what needs to be done and how, but the actual execution of the tool or function must be handled by an external runtime environment or an *Agent Controller*. For example, when asked about the weather in Boston, an LLM might recognize the need for the weather lookup function and respond with a structured call such as the following:

```
{
    "function": "weather_lookup",
    "parameters": {
        "location": "Boston",
        "date": "10/01/2024"
    }
}
```

This structured response is then interpreted and executed by the Agent Controller that actually has the capability to run the specified function with the provided parameters. The `weather_lookup` tool (or function) may look something like this:

```
import requests

def weather_lookup(location: str, date: str) -> dict:
   """A function to lookup weather data that takes location and date
   as input"""
   API_KEY = "api_key"
   base_url = "<api URL>"

   params = {
       "q": location,
        "appid": API_KEY,
       "units": "imperial"  # For Fahrenheit
  }
  response = requests.get(base_url, params=params)
  if response.status_code == 200:
      data = response.json()
      return data
```

At the minimum, the LLM agent requires the tool's description of what the tool does and what input it expects. You can also specify which parameters (in this case, `location` and `date`) are mandatory and which ones are optional. *Figure 5.2* demonstrates the flow between an LLM agent, tool, and the Agent Controller:

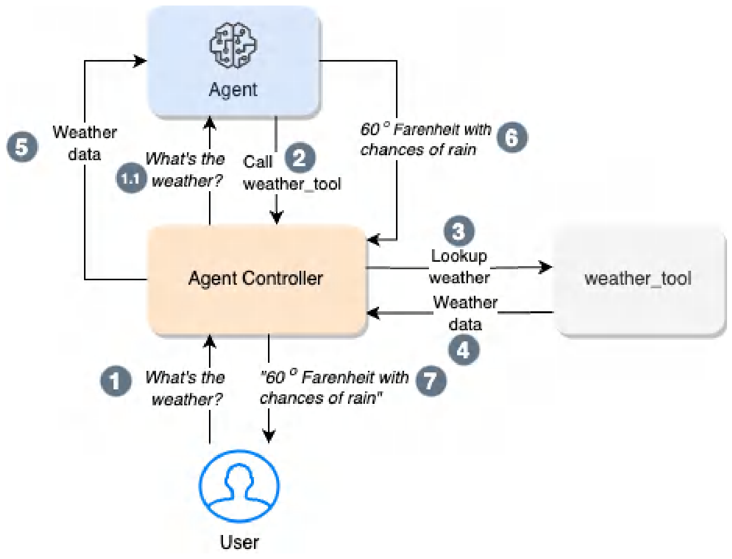

Figure 5.2 – LLM agent tool calling and tool execution by the Agent Controller

It is worth noting that not all LLMs are capable or efficient (or rather accurate) in tool/function calling. While larger models are more capable of tool calling, some larger models (such as OpenAI's GPT-4 and -4o, Anthropic's Claude Sonnet, Haiku, Opus, and Meta's Llama 3 models) are explicitly trained for tool calling behavior. While other models are not explicitly trained on tool calling, they may still be able to achieve similar functionality with aggressive prompt engineering, but with varying degrees of success.

Defining tools for agents

Tools are defined with clear descriptions, typically using docstrings or a JSON schema, to communicate their purpose, required inputs, and expected outputs to the agent. There are two main approaches to defining tools, depending on whether you're using a framework or working directly with LLM APIs.

Framework approach – using docstrings

In frameworks such as CrewAI or LangGraph, tools are defined using docstrings – descriptive text that appears at the beginning of a function. Here's an example of a weather lookup tool:

```
def weather_lookup(location: str, date: str = None):
    """
    A tool that can lookup real-time weather data.
    Arguments:
      location (str): The location to lookup weather for
      date (str) Optional: The date in MM/DD/YYYY format
    """
   # function code and logic
```

The docstring, enclosed within triple quotes (`"""`), provides crucial information about the following:

- The tool's purpose
- Required and optional arguments
- Expected return values

This approach makes tool creation intuitive for developers, as it uses standard programming practices. While Python uses triple quotes for docstrings, other programming languages may have different conventions for defining such documentation.

Direct LLM integration

When working directly with LLM APIs (such as Anthropic's Claude or OpenAI's GPT) without a framework, tools must be defined using a specific JSON schema format:

```
{
  "name": "weather_lookup",
  "description": "A tool that can lookup real-time weather data",
  "input_schema": {
     "type": "object",
     "properties": {
        "location": {
          "type": "string",
          "description": "The city and state, e.g. San Francisco, CA"
         }
       },
       "required": ["location"]
   }
 }
```

Multiple tools can be used as a list (or array) of JSON schema objects with the tool definition when invoking the model, such as the following:

```
tools = [
 { "name": "weather_lookup",
   "description": "A tool that can check weather data",
   … },
 {
  "name": "flight_booking",
  "description": "A tool that can book flights",
   … },
 …
]
```

Note that this is model-dependent, so you must refer to the model's documentation to learn more about how its APIs require you to specify tools. If your project uses multiple models that have different ways of defining tools, then it can quickly become cumbersome to define, manage, and maintain tool definitions. This is one of the reasons why there is an increase in affinity toward using libraries or frameworks such as CrewAI, LangGraph, and AutoGen, which provide a simplified way of defining tools regardless of the LLM being used for the agents.

Types of tools

LLM agents can leverage various types of toolkits to enhance their capabilities and perform complex tasks. Here are the main categories:

- **Application programming interfaces** (**APIs**): APIs serve as the primary gateway for agents to access external services and data in real time. They provide standardized methods for interacting with third-party systems, enabling agents to seamlessly integrate with various services. For instance, in a travel planning context, APIs allow agents to access weather services, payment processing systems, navigation and mapping services, and flight and hotel booking systems. This real-time connectivity ensures agents can provide up-to-date information and services to users.
- **Database tools**: Database tools enable agents to store, retrieve, and manage structured (or semi-structured) data efficiently. These tools support both reading and writing operations, allowing agents to maintain persistent information across sessions. Agents commonly use databases to store customer profiles and preferences, maintain historical transaction records, manage product catalogs, and access domain-specific knowledge bases. This persistent storage capability enables agents to learn from past interactions and provide personalized services.
- **Utility functions**: Utility functions are custom software components designed for specialized tasks that run locally within the agent's environment. These functions handle essential operations such as data processing and analysis, format conversion, mathematical calculations, and natural language processing tasks. They serve as the building blocks for more complex operations and help agents process information efficiently. Utility functions are particularly valuable for tasks that require consistent, repeatable operations.
- **Integration tools**: Integration tools specialize in connecting different systems and services, enabling seamless workflow automation. These tools handle crucial tasks such as calendar synchronization, document processing, file management, and communication systems integration. They act as bridges between different platforms and services, allowing agents to orchestrate complex workflows that span multiple systems and data sources.
- **Hardware interface tools**: Hardware interface tools enable agents to interact with physical devices and systems, bridging the gap between digital and physical worlds. These tools are essential for controlling IoT devices, integrating with robotics systems, processing sensor data, and managing physical automation systems. Through hardware interface tools, agents can extend their influence beyond digital interactions to affect real-world changes and monitor physical environments.

Each tool type serves specific purposes and can be combined to create powerful agent capabilities. The choice of tools depends on the agent's role, requirements, and the complexity of tasks it needs to perform.

Understanding how agents work with these tools involves the following several key considerations that affect their effectiveness and reliability. These aspects are crucial for developing robust agent systems that can handle complex real-world tasks while maintaining security, handling errors gracefully, and adapting to changing requirements:

- **Tool composition and chaining**: Agents often need to combine multiple tools to accomplish complex tasks. Tool composition allows agents to create sophisticated workflows by chaining tools together. For example, a travel planning agent might first use an API to check flight availability, then a database tool to retrieve user preferences, and, finally, a utility function to calculate optimal itineraries. This chaining capability significantly extends what agents can accomplish beyond using tools in isolation.
- **Tool selection and decision-making**: One of the most critical aspects of tool usage is the agent's ability to select the appropriate tool for a given task. Agents must evaluate the context, understand the requirements, and choose the most suitable tool or combination of tools. This involves considering factors such as tool capabilities, reliability, performance, and cost. The agent must also handle cases where multiple tools could solve the same problem, selecting the most efficient option.
- **Error handling and fallbacks**: When working with tools, agents must be prepared for potential failures and have strategies to handle them. This includes detecting failed API calls, managing database connection issues, or handling incorrect function outputs. Robust error handling often involves implementing fallback mechanisms, where agents can switch to alternative tools or approaches if the primary method fails.
- **Tool state management**: Many tools maintain state or require specific initialization and cleanup procedures. Agents need to manage these tool states effectively, ensuring proper resource allocation and release. This includes managing database connections, maintaining API authentication tokens, and handling session states for various services.
- **Tool updates and versioning**: Tools evolve over time with new versions and capabilities. Agents need strategies to handle tool updates, version compatibility, and deprecated features. This might involve maintaining compatibility with multiple versions of a tool, gracefully handling deprecated features, and adapting to new tool interfaces.
- **Tool security and access control**: Security considerations are crucial when agents interact with tools, especially those accessing sensitive data or critical systems. This includes managing authentication credentials, implementing proper authorization checks, and ensuring secure communication channels. Agents must also respect rate limits and usage quotas imposed by various tools.

Consider a practical example of interaction between a user and our AI travel agent using tools effectively.

User: "I need flight and hotel options for Rome for 2 adults, June 15–22, 2024, with a total budget of $3,000."

Using the CrewAI framework in the following code snippet, we will demonstrate how agents use tools in this focused travel planning scenario:

```
class TravelTools:
  def search_flights(self, ...) 6 -> dict:
   """Basic flight search simulation"""
   return {
    "flights": [ {"airline": "Alitalian airlines",
                  "price": 800, "duration": "9h"}]
     }

 def check_hotels(self, ...) -> dict:
     """Basic hotel search simulation"""
      return {
          "hotels": [ {"name": "Roma Inn",
                       "price": 150, "rating": 4.0}]
      }

 travel_agent = Agent(
    role='Travel Agent',
    goal='Find suitable flight and hotel options within
          budget',
  tools=[TravelTools().search_flights,
         TravelTools().check_hotels]
  )

 search_task = Task(
    description="Find flights and hotels for 2 adults to
                Rome, June 15-22, budget $3000",
    agent=travel_agent )

 crew = Crew(agents=[travel_agent], tasks=[search_task])
 result = crew.kickoff()
```

In this example, we can see several key concepts in action:

- **Tool definition**: The `TravelTools` class implements focused tools for specific travel-related tasks
- **Agent configuration**: The travel agent is configured with appropriate tools and a clear goal
- **Task specification**: The task is defined with precise parameters for the agent to work with
- **Tool integration**: The agent seamlessly integrates multiple tools (flight and hotel search) to accomplish its task
- **Execution flow**: The CrewAI framework manages the overall execution and coordination of the agent and its tools

This streamlined implementation demonstrates how agents can effectively use tools while maintaining clarity and purpose in their operations. In our example, the `TravelTools` class uses simplified JSON responses for clarity. However, in a real-world implementation, these tools would interact with actual external services and handle much more complex data.

Note that this is a rather simple implementation, and the actual implementation would involve integrating with various APIs, databases, and software tools specific to the travel domain. Additionally, advanced AI planning algorithms could be employed to optimize the itinerary construction and activity planning steps. This comprehensive tool usage allows the AI travel agent to provide a seamless, end-to-end, trip-planning experience far beyond just searching flights and hotels. You can find the full code in the Python notebook (`Chapter_05.ipynb`) in the GitHub repository.

The significance of tools in agentic systems

The paradigm shift toward tool use is driven by the recognition that many complex problems demand a diverse array of specialized tools and resources, each contributing a unique set of capabilities. Rather than attempting to encapsulate all requisite knowledge and functionalities within the agent itself, a more efficient and scalable approach involves intelligently leveraging the appropriate tools as needed.

For instance, an agent tasked with providing personalized healthcare recommendations could exploit tools such as medical databases, clinical decision support systems, and advanced diagnostic algorithms. By judiciously combining these external resources with its own reasoning capabilities, the agent can deliver more accurate and comprehensive guidance, tailored to individual patient profiles and conditions.

The concept of tool use in intelligent agents is not limited to software-based tools alone. In certain domains, such as robotics and automation, agents may interact with physical tools, machinery, or specialized equipment to extend their capabilities into the physical realm. For example, a robotic agent in a manufacturing plant could leverage various tools and machinery to perform intricate assembly tasks, quality inspections, or material handling operations.

Ultimately, the ability to effectively utilize external tools and resources is a hallmark of truly intelligent agents, capable of adapting and thriving in dynamic, complex environments. By going beyond the limitations of their native capabilities, these agents can continually evolve, leveraging the collective power of diverse tools and systems to achieve ambitious objectives.

Another good example is that of a virtual travel agent that has the capability to access multiple APIs, databases, and software tools to plan and book complete travel itineraries for users. Such a travel agent could leverage APIs from airlines, hotels, rental car companies, and travel review sites to gather real-time data on flight schedules, availability, pricing, and customer ratings. It could also tap into databases of travel advisories, travel document requirements, and destination information. By integrating and reasoning over all this data from various tools, the agent can provide personalized recommendations, make intelligent trade-offs, and seamlessly book and coordinate all aspects of a trip tailored to the user's preferences and constraints. Naturally, the set of tools used in such a case is diverse and they all operate in their unique ways.

We've looked at what tools are and how they work. Next, we will explore another critical aspect of agentic systems – planning – and some of the planning algorithms.

Planning algorithms for agents

Planning is a fundamental capability of intelligent agents, enabling them to reason about their actions and devise strategies to achieve their objectives effectively. Planning algorithms form the backbone of how LLM agents determine and sequence their actions. An algorithm is a step-by-step set of instructions or rules designed to solve a specific problem or complete a task. It is a sequence of unambiguous and finite steps that takes inputs and produces an expected output in a finite amount of time.

There are several planning algorithms in AI, each with its own strengths and approaches. However, when working with LLM agents, we need to consider their practicality in handling natural language, uncertainty, and large state spaces (all possible situations or configurations that an agent might encounter during its task). For example, in a simple robot navigation task, state spaces might include all possible positions and orientations, but in LLM agents, state spaces become vastly more complex as they include all possible conversation states, knowledge contexts, and potential responses.

Among the known planning algorithms – **Stanford Research Institute Problem Solver** (**STRIPS**), **hierarchical task network** (**HTN**), **A* planning**, **Monte Carlo Tree Search** (**MCTS**), **GraphPlan**, **Fast Forward** (**FF**), and **LLM-based planning** – they can be categorized by their practicality for LLM agents.

STRIPS, A* planning, GraphPlan, and MCTS, while powerful in traditional AI, are less practical for LLM agents due to their rigid structure and difficulty handling natural language. FF shows moderate potential but requires significant adaptation. The most practical approaches are LLM-based planning and HTN, as they naturally align with how language models process and decompose tasks. Let's discuss them in detail.

Less practical planning algorithms

As mentioned earlier, less practical planning algorithms include STRIPS, A* planning, GraphPlan, and MCTS. Here's a detailed overview.

STRIPS

STRIPS works with states and actions defined by logical predicates, making it effective for clear, binary conditions. However, it's unsuitable for LLM agents because natural language interactions can't be effectively reduced to simple `true/false` conditions. For example, while STRIPS can easily model `true/false` states, it struggles with nuanced language states such as *partially understanding a concept* or *somewhat satisfied with a response*, making it too rigid for language-based planning.

A* planning

A* planning, while powerful for pathfinding problems, faces fundamental challenges with LLM agents. The algorithm requires a clear way to calculate both the cost of actions taken and a heuristic estimate of the remaining cost to reach a goal. In language-based interactions, defining these costs becomes highly problematic – how do you quantify the "distance" between different conversation states or estimate the "cost" of reaching a particular understanding? These mathematical requirements make A* impractical for natural language planning.

GraphPlan

GraphPlan builds a layered graph structure representing possible actions and their effects at each time step. When applied to LLM agents, this approach breaks down because language interactions don't fit neatly into discrete layers with clear cause-and-effect relationships. The combinatorial explosion of possible language states and the difficulty in determining mutual exclusion relationships between different conversational actions make GraphPlan computationally intractable for language-based planning.

MCTS

For LLM agents, **MCTS** becomes impractical for two main reasons. First, each "simulation" would require actual LLM calls, making it prohibitively expensive in terms of computation and cost; second, the vast space of possible language interactions makes random sampling inefficient for finding meaningful patterns or strategies. The algorithm's strength in game-like scenarios becomes a weakness in open-ended language interactions.

Moderately practical planning algorithm – FF

FF planning is considered to be a moderately practical planning algorithm that can be used in LLM agents. It uses a heuristic search with a simplified version of the planning problem to guide its search. Its focus on goal-oriented planning could be adapted for LLM agents, though it would require modifications to handle natural language effectively. FF planning uses heuristic search with a simplified version of the planning problem to guide its search.

For LLM agents, FF planning offers several compelling advantages that make it worth considering. Its goal-oriented approach naturally aligns with how LLMs handle task completion, while its relaxed planning mechanism provides useful approximations for complex language tasks. The heuristic guidance helps manage the vast search space inherent in language-based planning, and its flexibility allows modification to work with partial state descriptions, which is particularly valuable in natural language contexts.

However, FF planning also faces significant challenges when applied to LLM agents. The original numeric heuristics that make FF effective in traditional planning don't translate smoothly to language states, and relaxed plans risk oversimplifying the rich context present in language interactions. There's also considerable difficulty in defining clear delete effects – what aspects of a conversation state are