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:

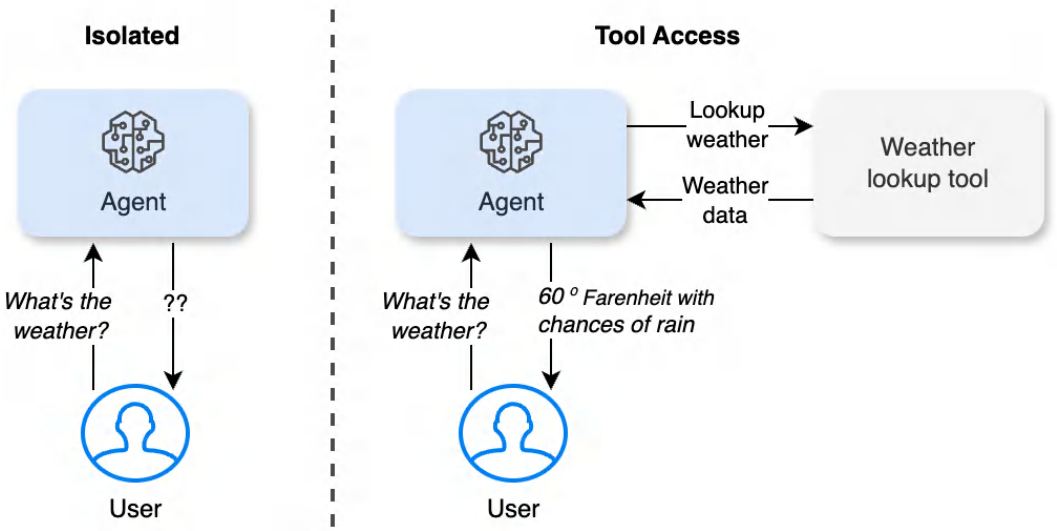

Figure 5.1 – Agent behavior in isolation versus with access to a tool

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:

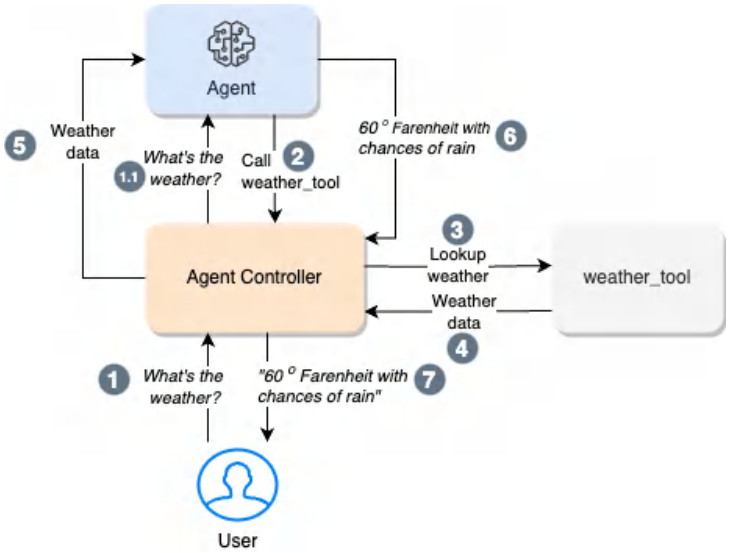

Figure 5.2 – LLM agent tool calling and tool execution by the Agent Controller

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:

```
1 class TravelTools:
2   def search_flights(self, ...) 6 -> dict:
     """Basic flight search simulation"""
3    return {
4     "flights": [ {"airline": "Alitalian airlines",
5                   "price": 800, "duration": "9h"}]
6     }
7
8  def check_hotels(self, ...) -> dict:
9      """Basic hotel search simulation"""
10     return {
11         "hotels": [ {"name": "Roma Inn",
12                      "price": 150, "rating": 4.0}]
13     }
14
15 travel_agent = Agent(
16    role='Travel Agent',
17    goal='Find suitable flight and hotel options within
           budget',
18  tools=[TravelTools().search_flights,
19         TravelTools().check_hotels]
20  )
21
22 search_task = Task(
23    description="Find flights and hotels for 2 adults to
                Rome, June 15-22, budget $3000",
24    agent=travel_agent )
25
26 crew = Crew(agents=[travel_agent], tasks=[search_task])
27 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

removed or changed by an action – in language-based planning. Perhaps most challengingly, the fundamental state representation requires substantial adaptation to work effectively with natural language. In practice, FF could be adapted for LLM agents as follows:

```
1 class LLMFastForward:
2    def create_relaxed_plan(self,
3                            current_state: str,
4                            goal: str) -> list:
5     """Create a simplified plan ignoring complexities"""
6     # Use LLM to generate a high-level plan
7     prompt = f"Given current state: {current_state}\nAnd
               goal: {goal}\n"
8     prompt += "Generate a simplified step-by-stepplan"
9        return self.llm.generate_plan(prompt)
10
11   def select_next_action(self, relaxed_plan: list):
12      """Choose next action based on the relaxed plan"""
13      # Implement action selection logic
14      return relaxed_plan[0]  # Simplified selection
```

This code demonstrates a simplified adaptation of FF planning for LLM agents. Let me explain its key components. The `LLMFastForward` class has two main methods:

- `create_relaxed_plan`: This method takes the current state and goal as text strings and uses an LLM to generate a simplified plan. Think of it as asking the LLM, "*Given where we are now, and where we want to go, what are the main steps we should take?*". It ignores many complexities, similar to how traditional FF planning ignores delete effects.
- `select_next_action`: This method chooses the next action from the relaxed plan. In this simplified version, it just takes the first action from the plan (`relaxed_plan[0]`). In a more sophisticated implementation, it would use additional logic to select the most appropriate next action.

In essence, this code shows how FF planning's core concept of using simplified plans to guide decision-making can be adapted to work with language models, even though it's a significant simplification of both FF planning and LLM capabilities. While this adaptation shows potential, implementing FF for LLM agents requires careful consideration of how to represent states, actions, and relaxed problems in a language-model context. This makes it moderately practical – possible but requiring significant modifications from its original form.

Most practical planning algorithms

When it comes to planning algorithms for LLM agents, two approaches stand out as particularly effective: LLM-based planning and HTN planning. These algorithms have proven especially suitable

for language models because they naturally align with how LLMs process information and handle complex tasks. While traditional planning algorithms often struggle with the ambiguity and complexity of natural language, these approaches embrace the fluid, contextual nature of language-based planning. Let's explore each of these algorithms and understand why they've become the preferred choices for modern AI agent frameworks.

LLM-based planning

Modern approaches leverage LLMs to generate plans in a more flexible and natural way. This approach can handle complex, real-world scenarios and understand context better than traditional planning algorithms. LLM-based planning operates on the principle that language models can understand complex goals, generate appropriate steps to achieve them, and adapt these steps based on changing contexts. Unlike traditional planners that require explicit state representations, LLM planners work with natural language descriptions of states and actions, making them inherently more flexible and expressive. Let's visualize the planning process using *Figure 5.3*:

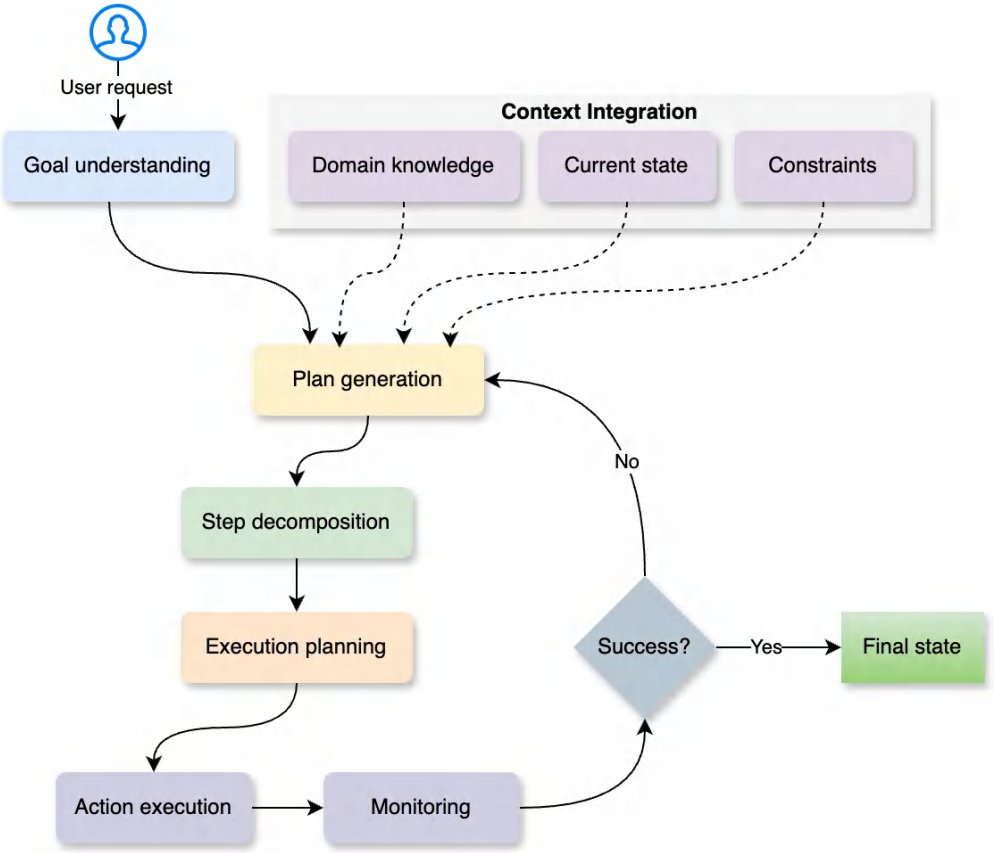

Figure 5.3 – LLM-based planning algorithm flow

Let's examine a practical implementation using CrewAI that demonstrates this planning approach. In this example, we'll create a travel planning system with two specialized agents: a *Travel Planning Strategist* who breaks down travel requests into manageable steps, and a *Travel Researcher* who validates and finds specific options. The system processes natural language travel requests and generates comprehensive travel plans through collaborative agent interaction. Here's the implementation:

```
class TravelPlanner:
    def __init__(self):
        self.planner = Agent(
            role='Travel Planning Strategist',
            goal='Create comprehensive, personalized travel plans',
            ... # Other parameters
        )
        self.researcher = Agent(
            role='Travel Researcher',
            goal='Find and validate travel options and
opportunities',
            ... # Other parameters
        )

    def create_travel_plan(self, request: str) -> Dict:
        planning_task = Task(
            description=f"""
            Analyze the following travel request and
            create a detailed plan:
            {request}
            Break this down into actionable steps by:
            1. Understanding client requirements
            3. Specific booking requirements
            4. Required validations
            """, agent=self.planner )

        research_task = Task(
            description="""
            Based on the initial plan, research and
            validate: Flight availability, hotel options,
            and local transportation
            """, agent=self.researcher)

        crew = Crew(
            agents=[self.planner, self.researcher],
            tasks=[planning_task, research_task],
            process=Process.sequential )
        return crew.kickoff(inputs={"request": request})
```

This implementation demonstrates several key advantages of LLM-based planning. The planner can understand complex natural language requests, dynamically generate appropriate steps, and adapt to different types of travel planning scenarios. The agents can work together, sharing context and building upon each other's outputs. The system's sophistication comes from its ability to handle nuanced requirements. For instance, when a user requests "*a relaxing beach vacation with some cultural activities*," the planner understands these abstract concepts and can translate them into concrete recommendations.

However, developers should be mindful of certain caveats. LLM-based planning systems can sometimes generate overly optimistic or impractical plans if not properly constrained. They may also struggle with highly specific numerical constraints or strict timing requirements unless these are explicitly handled in the implementation. A significant advantage of LLM-based planning over traditional algorithms lies in the system's adaptability. While STRIPS or A* planning would require explicit state representations for every possible travel scenario, LLM-based planning can handle novel situations by leveraging its understanding of language and context. This makes it particularly suitable for domains where requirements are often ambiguous or evolving. This planning approach also excels at handling uncertainty and partial information, something traditional planners struggle with. When information is missing or ambiguous, the system can generate reasonable assumptions and include contingency steps in its plans.

HTN

HTN planning breaks down complex tasks into simpler subtasks, creating a hierarchy of actions. Unlike STRIPS, which works with primitive actions, HTN can work with abstract tasks and decompose them into more concrete steps. This makes it particularly well-suited for real-world planning problems where tasks naturally decompose into subtasks. HTN planning works by breaking down high-level tasks into progressively smaller subtasks. Consider the following example code:

```
def buy_groceries_task():
    return [
        ('go_to_store', []),
        ('select_items', []),
        ('checkout', []),
        ('return_home', [])
    ]

def select_items_task():
    return [
        ('check_list', []),
        ('find_item', []),
        ('add_to_cart', [])
    ]
```

HTN planning operates on the principle of task decomposition, where high-level tasks (compound tasks) are broken down into smaller, more manageable subtasks until reaching primitive tasks that can be directly executed. This hierarchical structure allows for intuitive problem representation and efficient solution finding. In our example, `buy_groceries_task` is a high-level task broken down into four subtasks. One of these subtasks, `select_items`, is further decomposed into three more specific actions, and so on. In the context of our travel agent example, we can use a similar hierarchical breakdown of complex tasks decomposed into smaller tasks. Visually, *Figure 5.4* shows how this may look:

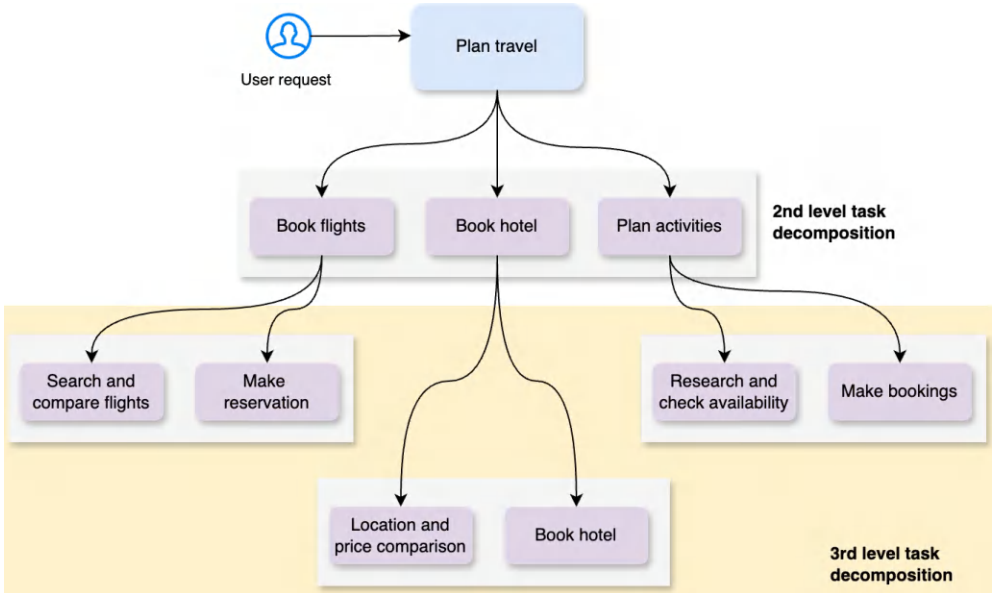

Figure 5.4 – HTN decomposition

To implement this with CrewAI, we can use CrewAI's *hierarchical* processing, where tasks are broken down into a hierarchical manner as explained using the HTN planning algorithm. With the CrewAI framework, the hierarchical method requires a m**anager** unit, which would be responsible for breaking down the tasks and *delegating* individual tasks to the agents. The Manager can either be an agent or it can be the LLM itself. If the Manager is an agent, then you can control how the manager breaks down the tasks to *n*-level tasks as per the workflow's needs. If the Manager is an LLM, then it will use the arbitrary plan generated by the LLM itself based on the user's query. With a Manager LLM, you may be able to control how the task breakdown works and how the delegation works using some prompt engineering; however, it is generally less flexible and is meant for simpler workflows. Here's a sample code for an HTN-like workflow for the travel planner:

```
flight_specialist = Agent(
    role='Flight Planning Specialist',
    goal='Handle all aspects of flight arrangements',
```

```
    backstory='Expert in airline bookings and flight
              logistics.')

accommodation_specialist = Agent(
   role='Accommodation Specialist',
   goal='Manage all accommodation-related planning',
   backstory='Expert in hotel and accommodation booking')

activity_specialist = Agent(
   role='Vacation Activity Specialist',
   goal='Manage all activity-related planning',
   backstory="Expert in recreational activity
              arrangements.",)

manager_llm = ChatOpenAI(model="gpt-4o-mini")
travel_planning_task = Task(
   description=f"""
     Plan a comprehensive flight itinerary based on the
     following request:
     {request}
     The plan should include: Flight arrangements,
     Accommodation bookings, other relevant travel
       components
     """,
     expected_output="A detailed flight itinerary
     covering all requested aspects.",
     agent=None) #No agent; the manager will delegate
                 subtasks

crew = Crew(
          agents=[self.flight_specialist,
                  self.accommodation_specialist,
                  self.activity_specialist],
          tasks=[travel_planning_task],
          process=Process.hierarchical,
          manager_llm=self.manager_llm,)
     return crew.kickoff()
```

The output of this execution may look as shown (output has been trimmed for brevity):

```
Final Travel Plan:
Here's the complete travel itinerary for a 5-day trip to Paris from
New York for two adults:
---
```

```
Travel Itinerary for Paris Trip
From New York (JFK) to Paris (CDG)
Travelers: 2 Adults , Duration: 5 Days
---
1. Flights:
- Departure Flight: ...
- Total Flight Cost: $2,960
---
2. Hotel Accommodations:
- Hotel: ...
- Estimated Total = €800.
---
3. Airport Transfers:
- Option 1: ...
- Option 2: ...
---
4. Day Trip to Versailles:
- Transportation: Round-trip via RER C train from …
   - Cost: Approximately …
   - Departure Time: 9:00 AM from …
   - Return Time: 5:00 PM from Versailles.
   ...
   - Overall Total for Day Trip: Approximately €364.20.
---
Grand Total Estimated Cost:
- Flights: $2,960
- Accommodation: €800 (with Le Fabe Hotel)
- Airport Transfers: €100 (may vary)
- Day Trip to Versailles: Approximately €364.20
- Convert Total Costs as Necessary to USD.
...
```

Note that, in this case, the agentic system has no access to external tools or lookup, so whatever response it generates is going to be completely fictional and non-factual. This underscores the importance of tools, which we will look at in the next section. For now, the previous example shows how you can use the framework for task breakdown and have a Manager manage several agents to perform decomposed simplified tasks from a user's request. You can see the full code in the Python notebook (`Chapter_05.ipynb`) in the GitHub repository.

HTN planning offers several significant advantages that make it particularly effective for complex planning scenarios. Its natural problem representation mirrors human thinking patterns, making it intuitive to understand and maintain. The hierarchical approach enables better scalability by breaking down complex problems into manageable subtasks, effectively reducing the search space. HTN's structure excels at encoding expert knowledge through its task hierarchies, allowing for reusable

patterns across similar problems. Additionally, its flexibility in handling both abstract and primitive tasks makes it adaptable to various planning situations, enabling planners to work at different levels of abstraction as needed.

So far, we've learned about tools and several planning algorithms, but together they can enable LLM agents to perform more complex, multi-step tasks by combining strategic planning with effective tool use. Let's further explore how we can effectively integrate tool use with planning within agentic systems.

Integrating tool use and planning

Most of the earlier work in AI planning and tool usage was done in isolation, focusing on either planning algorithms or tool capabilities separately. However, to achieve truly intelligent agents, there is a need to integrate tool use with planning effectively. As we already saw in the previous section, our travel planner gave us a detailed travel plan but none of the details were factual – that is, it contained information that the LLM simply made up. In order to infuse our system with actual flight, hotel, and activity data so that the travel plan is grounded in facts, we will need to utilize tools along with the planning algorithm. This section will discuss how to combine these two aspects to generate relevant responses and complete tasks accurately.

Reasoning about tools

Agents need the ability to reason about the available tools at their disposal, understanding the functionality, capabilities, and limitations of each tool, as well as the contexts and conditions under which they can be applied effectively. The reasoning process involves assessing the available tools based on the current goals and objectives, and then choosing the most appropriate ones that can be utilized in the given situation or problem domain.

For example, in the case of our travel planner, the agent will have access to various tools such as flight booking APIs, hotel reservation systems, and activity planning software. The agent needs to reason about the capabilities of each tool, such as which tools can be used for booking flights or book hotels, and which ones can provide information about local attractions.

When working with LLM agents, reasoning about tools is largely handled by the language model's inherent capabilities. Modern LLMs are trained to understand tool descriptions, purposes, and appropriate usage contexts. This means we don't need to explicitly program complex reasoning mechanisms – instead, we provide clear tool descriptions and let the LLM determine when and how to use them. For example, let's look at our travel planner agent scenario:

```
1 from crewai import Agent
2
3 travel_agent = Agent(
4   role='Travel Planner',
5   goal='Plan comprehensive travel itineraries',
6   tools=[
```

```
7       flight_search_tool,    # Tool for finding and booking flights
8       hotel_booking_tool,    # Tool for hotel reservations
9       activity_planner_tool  # Tool for local activities and
                                 attractions
10    ])
```

The LLM agent can naturally understand the following:

- Which tool to use for each task (for example, `flight_search_tool` for air travel)
- When to use tools in combination (for example, coordinating flight and hotel dates)
- How to adapt tool usage based on user requirements (for example, budget constraints)

This built-in reasoning capability means we can focus on providing well-defined tools with clear descriptions, rather than implementing complex reasoning mechanisms. The LLM will handle the decision-making process of tool selection and application based on the context and requirements of each situation. However, not all language models are equally capable of effective tool reasoning. This capability typically requires models that have been specifically trained or fine-tuned for tool use and function calling. Smaller models or those without tool-use training may have the following issues:

- Failing to understand when a tool is needed
- Making incorrect assumptions about tool capabilities
- Using tools in the wrong sequences
- Missing opportunities to use available tools
- Ignoring tool constraints or requirements

Even capable models can face limitations such as the following:

- Difficulty with complex tool combinations requiring many steps
- Inconsistency in tool selection across similar scenarios
- Challenges with tools that have subtle differences in functionality
- Struggles with error recovery when tools fail

This is why frameworks such as CrewAI, LangGraph, and AutoGen often work best with more advanced models that have demonstrated strong tool reasoning capabilities, and why it's important to test your agent's tool usage patterns before deployment.

Planning for tool use

The planning process in modern AI agents is fundamentally driven by LLM capabilities, building upon the principles we discussed in LLM-based planning and HTN approaches. Rather than following rigid planning algorithms, agents leverage their language model's understanding to create flexible, context-aware plans for tool usage. *Figure 5.5* depicts this process:

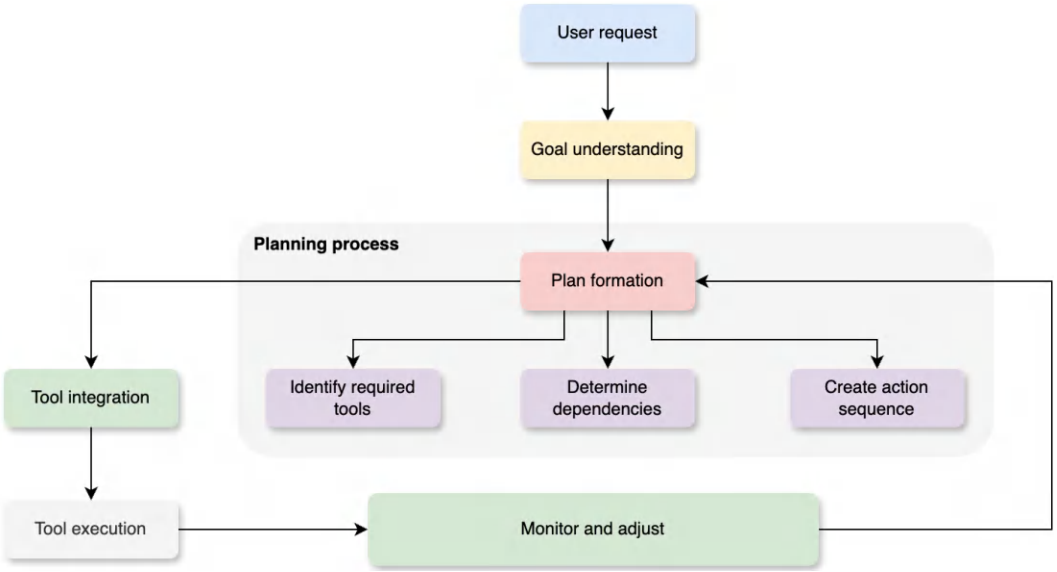

Figure 5.5 – Tool planning flow

When an agent receives a request, it first understands the goals through natural language processing. For a travel agent, this might mean comprehending that a family vacation request requires not just flight bookings but also family-friendly accommodation and activities. This goal-understanding phase draws directly from the LLM's trained comprehension abilities.

The planning process then shifts to identifying which tools are needed and in what sequence they should be used. This mirrors the hierarchical decomposition we saw in HTN planning but with the flexibility of LLM-based decision-making. The agent doesn't just follow predefined decomposition rules; it adapts its planning based on the specific context and requirements of each request.

Tool integration into the plan happens naturally as part of this process. The agent understands tool capabilities through their descriptions and can sequence them appropriately. For instance, when planning a vacation, the agent knows that flight dates need to be confirmed before booking hotels, and that activity planning should consider the location and timing of both.

This planning approach combines the structured nature of traditional planning algorithms with the adaptability of language models. The agent can adjust its plans based on new information or changing

circumstances, much like a human travel agent would modify their approach based on client feedback or availability changes.

The success of this planning process relies heavily on the LLM's ability to understand context and generate appropriate sequences of actions. This is why frameworks such as CrewAI often implement this type of planning, allowing agents to leverage their language understanding capabilities while maintaining the systematic approach needed for complex task completion.

Exploring practical implementations

To demonstrate how various AI/ML frameworks can be used to create intelligent agents capable of executing complex tasks through tool use and planning, let's explore examples using CrewAI, AutoGen, and LangGraph (the agentic framework of LangChain). You can find the full code for each of the framework examples in the `Chapter_05.ipynb` Python notebook in the GitHub repository.

CrewAI example

Let's examine how CrewAI implements tool-based reasoning through a practical travel planning example. The framework's Python library provides a `@tool` decorator that allows us to define tools with clear descriptions and documentation. Here's how we can create a set of travel-related tools:

```
@tool("Search for available flights between cities")
def search_flights(...) -> dict:
    """Search for available flights between cities."""
    # Call flight API and other tool logic

@tool("Find available hotels in a location")
def find_hotels(...) -> dict:
   """Search for available hotels in a location."""
   # Call hotels API and other tool logic

@tool("Find available activities in a location")
def find_activities(...) -> dict:
   """Find available activities in a location."""
   # Call activities API and other tool logic
```

The tools are then assigned to an agent that understands how to use them in context. The agent is created with a specific role, goal, and backstory that helps guide its decision-making:

```
Agent(
    role='An expert travel concierge',
    goal='Handle all aspects of travel planning',
    backstory="Expert in airline bookings and flight
          logistics, hotel bookings, and booking vacation
```

```
            activities.",
    tools=[search_flights, find_hotels, find_activities],
    verbose=False
)
```

When given a task, the agent uses these tools based on the context and requirements:

```
travel_planning_task = Task(
    description=f"""
    Plan a comprehensive travel and leisure itinerary
    based on the following request:
    {request}
    The plan should include:
    - Flight arrangements
    - Accommodation bookings
    - Any other relevant travel components
    """,
    expected_output="A detailed travel itinerary covering
        all requested aspects.",
    agent=self.travel_specialist )
```

When `crew.kickoff()` is called, CrewAI orchestrates the tool usage in the following ways:

- Understanding the task requirements through the task description
- Identifying which tools are needed based on the agent's role and the task goals
- Using the tools in a logical sequence to build the travel plan
- Processing tool outputs and incorporating them into the final response

This implementation demonstrates how CrewAI combines tool definitions, agent capabilities, and task specifications to create a coherent planning system. The framework handles the complexity of tool reasoning while allowing developers to focus on defining clear tool interfaces and agent behaviors.

AutoGen example

AutoGen provides a platform for developing AI agents that can engage in conversations and, through these interactions, arrive at solutions for given tasks. AutoGen approaches multi-agent collaboration through a RoundRobinGroupChat system where specialized agents interact to create a comprehensive travel plan. The implementation defines four key agents: a flight planner, a hotel planner, an activities planner, and a summary agent, each with specific responsibilities and tools.

Each agent is initialized with the following:

- A name and description
- A model client (in this case, OpenAI's GPT-4o-mini)
- Specific tools they can access
- A system message defining their role and responsibilities

The key differentiators from CrewAI lie in the execution model:

- **Agent communication**: While CrewAI uses a hierarchical task-based approach, AutoGen implements a round-robin group chat where agents take turns contributing to the solution. The `RoundRobinGroupChat` class orchestrates this conversation flow, allowing agents to build upon each other's suggestions.
- **Termination handling**: AutoGen uses an explicit termination condition through the `TextMentionTermination` class. The travel summary agent can end the conversation by mentioning `"TERMINATE"` when a complete plan is ready. This differs from CrewAI's task-completion-based termination. Here are the parameters of `TextMentionTermination`:
  - `mention_text(str)`: The keyword or phrase that triggers termination (e.g., `"TERMINATE"`)
  - `case_sensitive (bool, optional)`: Whether the keyword matching should be case-sensitive
  - `strip_whitespace (bool, optional)`: Whether to ignore leading/trailing spaces in the detected text
  - `regex_match (bool, optional)`: Allows for using regular expressions for more flexible termination triggers
- **Tool integration**: Instead of CrewAI's decorator-based tool definition, AutoGen associates tools directly with agents during initialization. Each agent has access to specific tools relevant to their role.
- **Coordination pattern**: While CrewAI often uses a manager-worker pattern, AutoGen's round-robin approach creates a more collaborative environment where agents contribute equally to the solution, with the summary agent responsible for creating the final integrated plan.

This implementation showcases AutoGen's strength in handling complex multi-agent conversations while maintaining clear role separation and specialized tool usage for each agent. The following is how you define agents with AutoGen:

```
1 flight_agent = AssistantAgent(
2    name="flight_planner",
3    model_client=model_client,
4    tools=[travel_tools.search_flights],
```

```
5    description="A helpful assistant that can plan flights
                itinerary for vacation trips.",
6    system_message="You are a helpful assistant that can
           plan flight itinerary for a travel plan for a
           user based on their request." )
7
8 hotel_agent = AssistantAgent(
9    name="hotel_planner",
10    model_client=model_client,
11    tools=[travel_tools.search_flights],
12    description="...", system_message="..." )
```

Once agents are defined, a `RoundRobinGroupChat` class can be defined using the agents and a conversation with the multi-agent system can be invoked:

```
2 group_chat = RoundRobinGroupChat(
3     [flight_agent, hotel_agent],
4     termination_condition=termination)
6  await Console(group_chat.run_stream(task="I need to plan
              a trip to Paris from New York for 5 days."))
```

LangGraph example

LangChain provides a framework for developing applications that can leverage LLMs alongside other tools and data sources. In the context of agentic systems, LangChain provides a sub-framework known as LangGraph that is used to build powerful LLM agent-based workflows. LangGraph approaches agent-based travel planning through a workflow graph system, offering a different paradigm from both CrewAI and AutoGen. Let's examine how this implementation works and its distinguishing characteristics.

LangGraph uses a state machine approach where the workflow is defined as a graph with nodes and edges. The implementation centers around two main nodes:

- An agent node that processes messages and makes decisions
- A tool node that executes the requested tools (flight search, hotel booking, and activity planning)

The workflow follows a cycle where the agent node evaluates the current state and either makes tool calls or provides a final response. This is controlled through a function that interprets the model's next move (that is, call a tool or end the response), which determines whether to route to the tools node or end the conversation. Just like CrewAI, LangGraph also uses the `@tool` decorator (for Python) with which the tool functions can be defined:

```
1 @tool
2 def search_flights(...) -> dict:
3    """Search for available flights between cities."""
```

```
    # Emulate JSON data from an API
    return data
```

Once nodes are defined with or without tools, they can be connected to each other to build a full graph structure of the workflow. For example, in our case, the following code defines a state graph-based workflow using LangGraph, where a task cycles between two nodes: agent and tools. The graph starts at the agent node (defined as the entry point), which calls a function (`call_model`) to process input. After the agent runs, a conditional function (`should_continue`) determines the next node – either looping back to the tools node or ending the workflow. The tools node (`tool_node`) processes intermediate tasks and always transitions back to the `agent` node, creating a repetitive cycle until the conditional function decides to stop. A `MemorySaver` checkpoint is used to persist the state across runs, and the graph is compiled into a LangChain-compatible runnable. Finally, the workflow is invoked with an initial input message about planning a trip, and the final message content is printed after the graph execution concludes:

```
workflow = StateGraph(MessagesState)
workflow.add_node("agent", call_model)
workflow.add_node("tools", tool_node)
workflow.add_edge(START, "agent")
workflow.add_conditional_edges("agent",should_continue)
workflow.add_edge("tools", 'agent')
checkpointer = MemorySaver()
app = workflow.compile(checkpointer=checkpointer)
final_state = app.invoke(
    {"messages": [HumanMessage(content="I need to plan a
                trip to Paris from New York for 5 days")]},
     config={"configurable": {"thread_id": 42}})
```

LangGraph's approach offers several notable advantages. For example, its graph structure provides explicit flow control, making workflows easy to visualize and understand, while built-in state management with checkpointing capabilities ensures robust handling of the application state. However, these benefits come with certain trade-offs. The framework requires a solid understanding of graph-based programming concepts, and its initial setup involves more overhead compared to CrewAI's more straightforward agent definition. The full code implementation can be found in the `Chapter_05.ipynb` Python notebook in the GitHub repository.

Table 5.1 illustrates some key differences between LangGraph, CrewAI, and AutoGen:

	LangGraph	**CrewAI**	**AutoGen**
State management	Uses explicit state management	Manages state through agent instances and their task context	Handles state through group chat message history
Tool integration	Tools are managed through a dedicated tool node	Uses a decorator-based tool definition with direct agent association	Associates tools directly with specific agents
Flow control	Uses a graph-based workflow	Uses hierarchical task decomposition or sequential flow	Implements round-robin turn-taking between agents

Table 5.1 – Comparison of LangGraph, CrewAI, and AutoGen implementation approaches

The preceding table shows the differences between LangGraph, CrewAI, and AutoGen based on our implementation.

Summary

In this chapter, we learned about the crucial role of tools and planning in AI agent systems. We discussed what tool/function calling is and how LLM agents exhibit this property. We also learned about various tool types and saw examples of how to use tools with frameworks or natively with an LLM. Subsequently, we explored various planning algorithms, from traditional approaches such as STRIPS and HTN to modern LLM-based planning methods, understanding their relative practicality in the context of language models. Through a practical travel planning example, we saw how tools can be defined, integrated, and utilized within each framework to create sophisticated planning systems.

We learned how integrating tool calling with planning can supercharge agentic systems by making them more capable of handling complex tasks. We also reviewed the implementation patterns across three frameworks (CrewAI, AutoGen, and LangGraph), which revealed distinct approaches to agent coordination and tool usage.

In the next chapter, we will dive into the concepts of the coordinator, worker, and delegator approach in agentic systems, and learn how they can help with completing complex real-world tasks.

Questions

1. What is the purpose of tools in AI agents, and how do docstrings help in tool definition?
2. Explain the difference between traditional planning algorithms (such as STRIPS) and modern LLM-based planning. Why are traditional algorithms less practical for LLM agents?
3. How does HTN planning work, and why is it considered one of the more practical approaches for LLM agents?
4. What role does reasoning play in tool selection for LLM agents, and what are its limitations?
5. When comparing frameworks (CrewAI, AutoGen, and LangGraph), what are the key factors to consider for an AI agent implementation?

Answers

1. Tools in AI agents are functions that enable agents to perform specific tasks or access external services. Docstrings provide crucial information about the tool's purpose, expected parameters, and return values, helping the LLM understand when and how to use each tool effectively. This documentation serves as the context that guides the model's decision-making process.
2. Traditional planning algorithms such as STRIPS rely on explicit state representations and predefined action sets, working with binary conditions and clear state transitions. LLM-based planning, however, operates with natural language understanding and can handle ambiguous states and actions. Traditional algorithms struggle with LLMs because they can't effectively represent the nuanced, contextual nature of language-based tasks.
3. HTN planning works by breaking down complex tasks into progressively simpler subtasks in a hierarchical structure. It's practical for LLM agents because this hierarchical decomposition mirrors how language models naturally process and understand tasks. The approach allows for both structured planning and the flexibility needed for language-based interactions.
4. Reasoning in LLM agents is largely handled by the model's built-in capabilities to understand context and make decisions. While this makes tool selection more natural, not all models are equally capable. Limitations include potential inconsistencies in tool selection, difficulties with complex tool combinations, and challenges in error recovery when tools fail.
5. Key factors for framework selection include the complexity of the workflow (structured versus conversational), the need for state management, multi-agent collaboration requirements, and development complexity. CrewAI offers straightforward implementation, AutoGen excels at multi-agent interaction, and LangGraph provides robust workflow control but requires more setup.