**Chapter 5\_ Tool Use**

Chapter 5: Tool Use (Function Calling)

**Tool Use Pattern Overview**

So far, we've discussed agentic patterns that primarily involve orchestrating interactions between language models and managing the flow of information within the agent's internal workflow (Chaining, Routing, Parallelization, Reflection). However, for agents to be truly useful and interact with the real world or external systems, they need the ability to use Tools.

The Tool Use pattern, often implemented through a mechanism called Function Calling, enables an agent to interact with external APIs, databases, services, or even execute code. It allows the LLM at the core of the agent to decide when and how to use a specific external function based on the user's request or the current state of the task.

The process typically involves:

1. **Tool Definition:** External functions or capabilities are defined and described to the LLM. This description includes the function's purpose, its name, and the parameters it accepts, along with their types and descriptions.
2. **LLM Decision:** The LLM receives the user's request and the available tool definitions. Based on its understanding of the request and the tools, the LLM decides if calling one or more tools is necessary to fulfill the request.
3. **Function Call Generation:** If the LLM decides to use a tool, it generates a structured output (often a JSON object) that specifies the name of the tool to call and the arguments (parameters) to pass to it, extracted from the user's request.
4. **Tool Execution:** The agentic framework or orchestration layer intercepts this structured output. It identifies the requested tool and executes the actual external function with the provided arguments.
5. **Observation/Result:** The output or result from the tool execution is returned to the agent.
6. **LLM Processing (Optional but common):** The LLM receives the tool's output as context and uses it to formulate a final response to the user or decide on the next step in the workflow (which might involve calling another tool, reflecting, or providing a final answer).

This pattern is fundamental because it breaks the limitations of the LLM's training data and allows it to access up-to-date information, perform calculations it can't do internally, interact with user-specific data, or trigger real-world actions. Function calling is the technical mechanism that bridges the gap between the LLM's reasoning capabilities and the vast array of external functionalities available.

While "function calling" aptly describes invoking specific, predefined code functions, it's useful to consider the more expansive concept of "tool calling." This broader term acknowledges that an agent's capabilities can extend far beyond simple function execution. A "tool" can be a traditional function, but it can also be a complex API endpoint, a request to a database, or even an instruction directed at another specialized agent. This perspective allows us to envision more sophisticated systems where, for instance, a primary agent might delegate a complex data analysis task to a dedicated "analyst agent" or query an external knowledge base through its API. Thinking in terms of "tool calling" better captures the full potential of agents to act as orchestrators across a diverse ecosystem of digital resources and other intelligent entities.

Frameworks like LangChain, LangGraph, and Google Agent Developer Kit (ADK) provide robust support for defining tools and integrating them into agent workflows, often leveraging the native function calling capabilities of modern LLMs like those in the Gemini or OpenAI series. On the "canvas" of these frameworks, you define the tools and then configure agents (typically LLM Agents) to be aware of and capable of using these tools.

Tool Use is a cornerstone pattern for building powerful, interactive, and externally aware agents.

**Practical Applications & Use Cases**

The Tool Use pattern is applicable in virtually any scenario where an agent needs to go beyond generating text to perform an action or retrieve specific, dynamic information:

1. Information Retrieval from External Sources:

Accessing real-time data or information that is not present in the LLM's training data.

* **Use Case:** A weather agent.
* **Tool:** A weather API that takes a location and returns the current weather conditions.
* **Agent Flow:** User asks, "What's the weather in London?", LLM identifies the need for the weather tool, calls the tool with "London", tool returns data, LLM formats the data into a user-friendly response.

2. Interacting with Databases and APIs:

Performing queries, updates, or other operations on structured data.

* **Use Case:** An e-commerce agent.
* **Tools:** API calls to check product inventory, get order status, or process payments.
* **Agent Flow:** User asks "Is product X in stock?", LLM calls the inventory API, tool returns stock count, LLM tells the user the stock status.

3. Performing Calculations and Data Analysis:

Using external calculators, data analysis libraries, or statistical tools.

* **Use Case:** A financial agent.
* **Tools:** A calculator function, a stock market data API, a spreadsheet tool.
* **Agent Flow:** User asks "What's the current price of AAPL and calculate the potential profit if I bought 100 shares at $150?", LLM calls stock API, gets current price, then calls calculator tool, gets result, formats response.

4. Sending Communications:

Sending emails, messages, or making API calls to external communication services.

* **Use Case:** A personal assistant agent.
* **Tool:** An email sending API.
* **Agent Flow:** User says, "Send an email to John about the meeting tomorrow.", LLM calls an email tool with the recipient, subject, and body extracted from the request.

5. Executing Code:

Running code snippets in a safe environment to perform specific tasks.

* **Use Case:** A coding assistant agent.
* **Tool:** A code interpreter.
* **Agent Flow:** User provides a Python snippet and asks, "What does this code do?", LLM uses the interpreter tool to run the code and analyze its output.

6. Controlling Other Systems or Devices:

Interacting with smart home devices, IoT platforms, or other connected systems.

* **Use Case:** A smart home agent.
* **Tool:** An API to control smart lights.
* **Agent Flow:** User says, "Turn off the living room lights." LLM calls the smart home tool with the command and target device.

Tool Use is what transforms a language model from a text generator into an agent capable of sensing, reasoning, and acting in the digital or physical world (see Fig. 1)

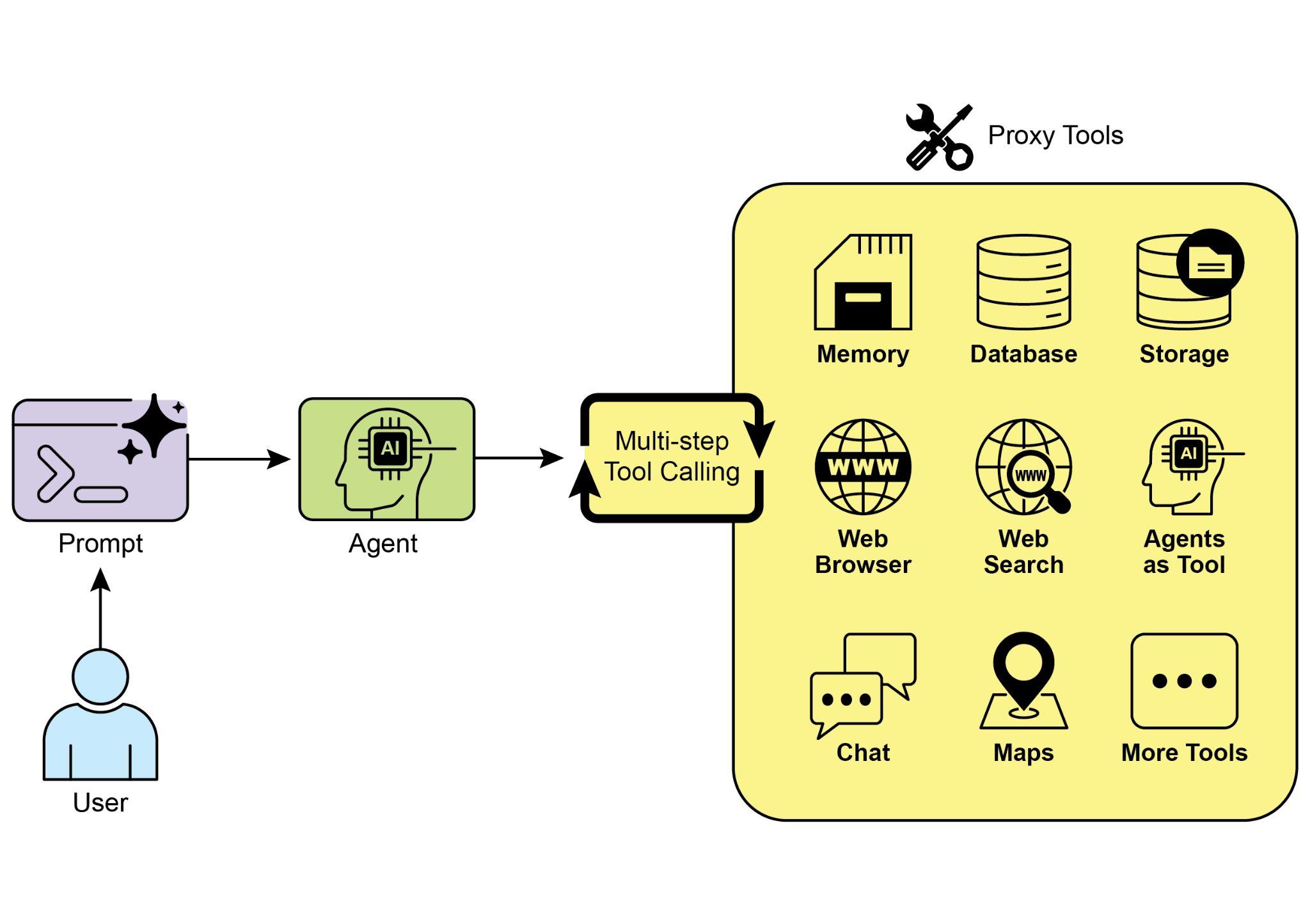


Fig.1: Some examples of an Agent using Tools

**Hands-On Code Example (LangChain)**

The implementation of tool use within the LangChain framework is a two-stage process. Initially, one or more tools are defined, typically by encapsulating existing Python functions or other runnable components. Subsequently, these tools are bound to a language model, thereby granting the model the capability to generate a structured tool-use request when it determines that an external function call is required to fulfill a user's query.

The following implementation will demonstrate this principle by first defining a simple function to simulate an information retrieval tool. Following this, an agent will be constructed and configured to leverage this tool in response to user input. The execution of this example requires the installation of the core LangChain libraries and a model-specific provider package. Furthermore, proper authentication with the selected language model service, typically via an API key configured in the local environment, is a necessary prerequisite.

|  |
| --- |
| import os, getpass  import asyncio  import nest\_asyncio  from typing import List  from dotenv import load\_dotenv  import logging  from langchain\_google\_genai import ChatGoogleGenerativeAI  from langchain\_core.prompts import ChatPromptTemplate  from langchain\_core.tools import tool as langchain\_tool  from langchain.agents import create\_tool\_calling\_agent, AgentExecutor  # UNCOMMENT  # Prompt the user securely and set API keys as an environment variables  os.environ["GOOGLE\_API\_KEY"] = getpass.getpass("Enter your Google API key: ")  os.environ["OPENAI\_API\_KEY"] = getpass.getpass("Enter your OpenAI API key: ")  try:  # A model with function/tool calling capabilities is required.  llm = ChatGoogleGenerativeAI(model="gemini-2.0-flash", temperature=0)  print(f"✅ Language model initialized: {llm.model}")  except Exception as e:  print(f"🛑 Error initializing language model: {e}")  llm = None  # --- Define a Tool ---  @langchain\_tool  def search\_information(query: str) -> str:  """  Provides factual information on a given topic. Use this tool to find answers to phrases  like 'capital of France' or 'weather in London?'.  """  print(f"\n--- 🛠️ Tool Called: search\_information with query: '{query}' ---")  # Simulate a search tool with a dictionary of predefined results.  simulated\_results = {  "weather in london": "The weather in London is currently cloudy with a temperature of 15°C.",  "capital of france": "The capital of France is Paris.",  "population of earth": "The estimated population of Earth is around 8 billion people.",  "tallest mountain": "Mount Everest is the tallest mountain above sea level.",  "default": f"Simulated search result for '{query}': No specific information found, but the topic seems interesting."  }  result = simulated\_results.get(query.lower(), simulated\_results["default"])  print(f"--- TOOL RESULT: {result} ---")  return result  tools = [search\_information]  # --- Create a Tool-Calling Agent ---  if llm:  # This prompt template requires an `agent\_scratchpad` placeholder for the agent's internal steps.  agent\_prompt = ChatPromptTemplate.from\_messages([  ("system", "You are a helpful assistant."),  ("human", "{input}"),  ("placeholder", "{agent\_scratchpad}"),  ])  # Create the agent, binding the LLM, tools, and prompt together.  agent = create\_tool\_calling\_agent(llm, tools, agent\_prompt)  # AgentExecutor is the runtime that invokes the agent and executes the chosen tools.  # The 'tools' argument is not needed here as they are already bound to the agent.  agent\_executor = AgentExecutor(agent=agent, verbose=True, tools=tools)  async def run\_agent\_with\_tool(query: str):  """Invokes the agent executor with a query and prints the final response."""  print(f"\n--- 🏃 Running Agent with Query: '{query}' ---")  try:  response = await agent\_executor.ainvoke({"input": query})  print("\n--- ✅ Final Agent Response ---")  print(response["output"])  except Exception as e:  print(f"\n🛑 An error occurred during agent execution: {e}")  async def main():  """Runs all agent queries concurrently."""  tasks = [  run\_agent\_with\_tool("What is the capital of France?"),  run\_agent\_with\_tool("What's the weather like in London?"),  run\_agent\_with\_tool("Tell me something about dogs.") # Should trigger the default tool response  ]  await asyncio.gather(\*tasks)  nest\_asyncio.apply()  asyncio.run(main()) |

The code sets up a tool-calling agent using the LangChain library and the Google Gemini model. It defines a search\_information tool that simulates providing factual answers to specific queries. The tool has predefined responses for "weather in london," "capital of france," and "population of earth," and a default response for other queries. A ChatGoogleGenerativeAI model is initialized, ensuring it has tool-calling capabilities. A ChatPromptTemplate is created to guide the agent's interaction. The create\_tool\_calling\_agent function is used to combine the language model, tools, and prompt into an agent. An AgentExecutor is then set up to manage the agent's execution and tool invocation. The run\_agent\_with\_tool asynchronous function is defined to invoke the agent with a given query and print the result. The main asynchronous function prepares multiple queries to be run concurrently. These queries are designed to test both the specific and default responses of the search\_information tool. Finally, the asyncio.run(main()) call executes all the agent tasks. The code includes checks for successful LLM initialization before proceeding with agent setup and execution.

**Hands-On Code Example (CrewAI)**

This code provides a practical example of how to implement function calling (Tools) within the CrewAI framework. It sets up a simple scenario where an agent is equipped with a tool to look up information. The example specifically demonstrates fetching a simulated stock price using this agent and tool.

|  |
| --- |
| # pip install crewai langchain-openai  import os  from crewai import Agent, Task, Crew  from crewai.tools import tool  import logging  # --- Best Practice: Configure Logging ---  # A basic logging setup helps in debugging and tracking the crew's execution.  logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(levelname)s - %(message)s')  # --- Set up your API Key ---  # For production, it's recommended to use a more secure method for key management  # like environment variables loaded at runtime or a secret manager.  #  # Set the environment variable for your chosen LLM provider (e.g., OPENAI\_API\_KEY)  # os.environ["OPENAI\_API\_KEY"] = "YOUR\_API\_KEY"  # os.environ["OPENAI\_MODEL\_NAME"] = "gpt-4o"  # --- 1. Refactored Tool: Returns Clean Data ---  # The tool now returns raw data (a float) or raises a standard Python error.  # This makes it more reusable and forces the agent to handle outcomes properly.  @tool("Stock Price Lookup Tool")  def get\_stock\_price(ticker: str) -> float:  """  Fetches the latest simulated stock price for a given stock ticker symbol.  Returns the price as a float. Raises a ValueError if the ticker is not found.  """  logging.info(f"Tool Call: get\_stock\_price for ticker '{ticker}'")  simulated\_prices = {  "AAPL": 178.15,  "GOOGL": 1750.30,  "MSFT": 425.50,  }  price = simulated\_prices.get(ticker.upper())  if price is not None:  return price  else:  # Raising a specific error is better than returning a string.  # The agent is equipped to handle exceptions and can decide on the next action.  raise ValueError(f"Simulated price for ticker '{ticker.upper()}' not found.")  # --- 2. Define the Agent ---  # The agent definition remains the same, but it will now leverage the improved tool.  financial\_analyst\_agent = Agent(  role='Senior Financial Analyst',  goal='Analyze stock data using provided tools and report key prices.',  backstory="You are an experienced financial analyst adept at using data sources to find stock information. You provide clear, direct answers.",  verbose=True,  tools=[get\_stock\_price],  # Allowing delegation can be useful, but is not necessary for this simple task.  allow\_delegation=False,  )  # --- 3. Refined Task: Clearer Instructions and Error Handling ---  # The task description is more specific and guides the agent on how to react  # to both successful data retrieval and potential errors.  analyze\_aapl\_task = Task(  description=(  "What is the current simulated stock price for Apple (ticker: AAPL)? "  "Use the 'Stock Price Lookup Tool' to find it. "  "If the ticker is not found, you must report that you were unable to retrieve the price."  ),  expected\_output=(  "A single, clear sentence stating the simulated stock price for AAPL. "  "For example: 'The simulated stock price for AAPL is $178.15.' "  "If the price cannot be found, state that clearly."  ),  agent=financial\_analyst\_agent,  )  # --- 4. Formulate the Crew ---  # The crew orchestrates how the agent and task work together.  financial\_crew = Crew(  agents=[financial\_analyst\_agent],  tasks=[analyze\_aapl\_task],  verbose=True # Set to False for less detailed logs in production  )  # --- 5. Run the Crew within a Main Execution Block ---  # Using a \_\_name\_\_ == "\_\_main\_\_": block is a standard Python best practice.  def main():  """Main function to run the crew."""  # Check for API key before starting to avoid runtime errors.  if not os.environ.get("OPENAI\_API\_KEY"):  print("ERROR: The OPENAI\_API\_KEY environment variable is not set.")  print("Please set it before running the script.")  return  print("\n## Starting the Financial Crew...")  print("---------------------------------")    # The kickoff method starts the execution.  result = financial\_crew.kickoff()  print("\n---------------------------------")  print("## Crew execution finished.")  print("\nFinal Result:\n", result)  if \_\_name\_\_ == "\_\_main\_\_":  main() |

This code demonstrates a simple application using the Crew.ai library to simulate a financial analysis task. It defines a custom tool, get\_stock\_price, that simulates looking up stock prices for predefined tickers. The tool is designed to return a floating-point number for valid tickers or raise a ValueError for invalid ones. A Crew.ai Agent named financial\_analyst\_agent is created with the role of a Senior Financial Analyst. This agent is given the get\_stock\_price tool to interact with. A Task is defined, analyze\_aapl\_task, specifically instructing the agent to find the simulated stock price for AAPL using the tool. The task description includes clear instructions on how to handle both success and failure cases when using the tool. A Crew is assembled, comprising the financial\_analyst\_agent and the analyze\_aapl\_task. The verbose setting is enabled for both the agent and the crew to provide detailed logging during execution. The main part of the script runs the crew's task using the kickoff() method within a standard if \_\_name\_\_ == "\_\_main\_\_": block. Before starting the crew, it checks if the OPENAI\_API\_KEY environment variable is set, which is required for the agent to function. The result of the crew's execution, which is the output of the task, is then printed to the console. The code also includes basic logging configuration for better tracking of the crew's actions and tool calls. It uses environment variables for API key management, though it notes that more secure methods are recommended for production environments. In short, the core logic showcases how to define tools, agents, and tasks to create a collaborative workflow in Crew.ai.

**Hands-on code (Google ADK)**

**The Google Agent Developer Kit (ADK) includes a library of natively integrated tools that can be directly incorporated into an agent's capabilities.**

**Google search: A primary example of such a component is the Google Search tool. This tool serves as a direct interface to the Google Search engine, equipping the agent with the functionality to perform web searches and retrieve external information.**

|  |
| --- |
| from google.adk.agents import Agent  from google.adk.runners import Runner  from google.adk.sessions import InMemorySessionService  from google.adk.tools import google\_search  from google.genai import types  import nest\_asyncio  import asyncio  # Define variables required for Session setup and Agent execution  APP\_NAME="Google Search\_agent"  USER\_ID="user1234"  SESSION\_ID="1234"  # Define Agent with access to search tool  root\_agent = ADKAgent(  name="basic\_search\_agent",  model="gemini-2.0-flash-exp",  description="Agent to answer questions using Google Search.",  instruction="I can answer your questions by searching the internet. Just ask me anything!",  tools=[google\_search] # Google Search is a pre-built tool to perform Google searches.  )  # Agent Interaction  async def call\_agent(query):  """  Helper function to call the agent with a query.  """  # Session and Runner  session\_service = InMemorySessionService()  session = await session\_service.create\_session(app\_name=APP\_NAME, user\_id=USER\_ID, session\_id=SESSION\_ID)  runner = Runner(agent=root\_agent, app\_name=APP\_NAME, session\_service=session\_service)  content = types.Content(role='user', parts=[types.Part(text=query)])  events = runner.run(user\_id=USER\_ID, session\_id=SESSION\_ID, new\_message=content)  for event in events:  if event.is\_final\_response():  final\_response = event.content.parts[0].text  print("Agent Response: ", final\_response)  nest\_asyncio.apply()  asyncio.run(call\_agent("what's the latest ai news?")) |

This code demonstrates how to create and use a basic agent powered by the Google ADK for Python. The agent is designed to answer questions by utilizing Google Search as a tool. First, necessary libraries from IPython, google.adk, and google.genai are imported. Constants for the application name, user ID, and session ID are defined. An Agent instance named "basic\_search\_agent" is created with a description and instructions indicating its purpose. It's configured to use the Google Search tool, which is a pre-built tool provided by the ADK. An InMemorySessionService (see Chapter 8) is initialized to manage sessions for the agent. A new session is created for the specified application, user, and session IDs. A Runner is instantiated, linking the created agent with the session service. This runner is responsible for executing the agent's interactions within a session. A helper function call\_agent is defined to simplify the process of sending a query to the agent and processing the response. Inside call\_agent, the user's query is formatted as a types.Content object with the role 'user'. The runner.run method is called with the user ID, session ID, and the new message content. The runner.run method returns a list of events representing the agent's actions and responses. The code iterates through these events to find the final response. If an event is identified as the final response, the text content of that response is extracted. The extracted agent response is then printed to the console. Finally, the call\_agent function is called with the query "what's the latest ai news?" to demonstrate the agent in action.

**Code execution:** The Google ADK features integrated components for specialized tasks, including an environment for dynamic code execution. The built\_in\_code\_execution tool provides an agent with a sandboxed Python interpreter. This allows the model to write and run code to perform computational tasks, manipulate data structures, and execute procedural scripts. Such functionality is critical for addressing problems that require deterministic logic and precise calculations, which are outside the scope of probabilistic language generation alone.

|  |
| --- |
| import os, getpass  import asyncio  import nest\_asyncio  from typing import List  from dotenv import load\_dotenv  import logging  from google.adk.agents import Agent as ADKAgent, LlmAgent  from google.adk.runners import Runner  from google.adk.sessions import InMemorySessionService  from google.adk.tools import google\_search  from google.adk.code\_executors import BuiltInCodeExecutor  from google.genai import types  # Define variables required for Session setup and Agent execution  APP\_NAME="calculator"  USER\_ID="user1234"  SESSION\_ID="session\_code\_exec\_async"  # Agent Definition  code\_agent = LlmAgent(  name="calculator\_agent",  model="gemini-2.0-flash",  code\_executor=BuiltInCodeExecutor(),  instruction="""You are a calculator agent.  When given a mathematical expression, write and execute Python code to calculate the result.  Return only the final numerical result as plain text, without markdown or code blocks.  """,  description="Executes Python code to perform calculations.",  )  # Agent Interaction (Async)  async def call\_agent\_async(query):  # Session and Runner  session\_service = InMemorySessionService()  session = await session\_service.create\_session(app\_name=APP\_NAME, user\_id=USER\_ID, session\_id=SESSION\_ID)  runner = Runner(agent=code\_agent, app\_name=APP\_NAME, session\_service=session\_service)  content = types.Content(role='user', parts=[types.Part(text=query)])  print(f"\n--- Running Query: {query} ---")  final\_response\_text = "No final text response captured."  try:  # Use run\_async  async for event in runner.run\_async(user\_id=USER\_ID, session\_id=SESSION\_ID, new\_message=content):  print(f"Event ID: {event.id}, Author: {event.author}")  # --- Check for specific parts FIRST ---  # has\_specific\_part = False  if event.content and event.content.parts and event.is\_final\_response():  for part in event.content.parts: # Iterate through all parts  if part.executable\_code:  # Access the actual code string via .code  print(f" Debug: Agent generated code:\n```python\n{part.executable\_code.code}\n```")  has\_specific\_part = True  elif part.code\_execution\_result:  # Access outcome and output correctly  print(f" Debug: Code Execution Result: {part.code\_execution\_result.outcome} - Output:\n{part.code\_execution\_result.output}")  has\_specific\_part = True  # Also print any text parts found in any event for debugging  elif part.text and not part.text.isspace():  print(f" Text: '{part.text.strip()}'")  # Do not set has\_specific\_part=True here, as we want the final response logic below  # --- Check for final response AFTER specific parts ---  text\_parts = [part.text for part in event.content.parts if part.text]  final\_result = "".join(text\_parts)  print(f"==> Final Agent Response: {final\_result}")  except Exception as e:  print(f"ERROR during agent run: {e}")  print("-" \* 30)  # Main async function to run the examples  async def main():  await call\_agent\_async("Calculate the value of (5 + 7) \* 3")  await call\_agent\_async("What is 10 factorial?")  # Execute the main async function  try:  nest\_asyncio.apply()  asyncio.run(main())  except RuntimeError as e:  # Handle specific error when running asyncio.run in an already running loop (like Jupyter/Colab)  if "cannot be called from a running event loop" in str(e):  print("\nRunning in an existing event loop (like Colab/Jupyter).")  print("Please run `await main()` in a notebook cell instead.")  # If in an interactive environment like a notebook, you might need to run:  # await main()  else:  raise e # Re-raise other runtime errors |

This script uses Google's Agent Development Kit (ADK) to create an agent that solves mathematical problems by writing and executing Python code. It defines an LlmAgent specifically instructed to act as a calculator, equipping it with the built\_in\_code\_execution tool. The primary logic resides in the call\_agent\_async function, which sends a user's query to the agent's runner and processes the resulting events. Inside this function, an asynchronous loop iterates through events, printing the generated Python code and its execution result for debugging. The code carefully distinguishes between these intermediate steps and the final event containing the numerical answer. Finally, a main function runs the agent with two different mathematical expressions to demonstrate its ability to perform calculations.

**Enterprise search:** This code defines a Google ADK application using the google.adk library in Python. It specifically uses a VSearchAgent, which is designed to answer questions by searching a specified Vertex AI Search datastore. The code initializes a VSearchAgent named "q2\_strategy\_vsearch\_agent", providing a description, the model to use ("gemini-2.0-flash-exp"), and the ID of the Vertex AI Search datastore. The DATASTORE\_ID is expected to be set as an environment variable. It then sets up a Runner for the agent, using an InMemorySessionService to manage conversation history. An asynchronous function call\_vsearch\_agent\_async is defined to interact with the agent. This function takes a query, constructs a message content object, and calls the runner's run\_async method to send the query to the agent. The function then streams the agent's response back to the console as it arrives. It also prints information about the final response, including any source attributions from the datastore. Error handling is included to catch exceptions during the agent's execution, providing informative messages about potential issues like an incorrect datastore ID or missing permissions. Another asynchronous function run\_vsearch\_example is provided to demonstrate how to call the agent with example queries. The main execution block checks if the DATASTORE\_ID is set and then runs the example using asyncio.run. It includes a check to handle cases where the code is run in an environment that already has a running event loop, like a Jupyter notebook.

|  |
| --- |
| import asyncio  from google.genai import types  from google.adk import agents  from google.adk.runners import Runner  from google.adk.sessions import InMemorySessionService  import os  # --- Configuration ---  # Ensure you have set your GOOGLE\_API\_KEY and DATASTORE\_ID environment variables  # For example:  # os.environ["GOOGLE\_API\_KEY"] = "YOUR\_API\_KEY"  # os.environ["DATASTORE\_ID"] = "YOUR\_DATASTORE\_ID"  DATASTORE\_ID = os.environ.get("DATASTORE\_ID")  # --- Application Constants ---  APP\_NAME = "vsearch\_app"  USER\_ID = "user\_123" # Example User ID  SESSION\_ID = "session\_456" # Example Session ID  # --- Agent Definition (Updated with the newer model from the guide) ---  vsearch\_agent = agents.VSearchAgent(  name="q2\_strategy\_vsearch\_agent",  description="Answers questions about Q2 strategy documents using Vertex AI Search.",  model="gemini-2.0-flash-exp", # Updated model based on the guide's examples  datastore\_id=DATASTORE\_ID,  model\_parameters={"temperature": 0.0}  )  # --- Runner and Session Initialization ---  runner = Runner(  agent=vsearch\_agent,  app\_name=APP\_NAME,  session\_service=InMemorySessionService(),  )  # --- Agent Invocation Logic ---  async def call\_vsearch\_agent\_async(query: str):  """Initializes a session and streams the agent's response."""  print(f"User: {query}")  print("Agent: ", end="", flush=True)  try:  # Construct the message content correctly  content = types.Content(role='user', parts=[types.Part(text=query)])  # Process events as they arrive from the asynchronous runner  async for event in runner.run\_async(  user\_id=USER\_ID,  session\_id=SESSION\_ID,  new\_message=content  ):  # For token-by-token streaming of the response text  if hasattr(event, 'content\_part\_delta') and event.content\_part\_delta:  print(event.content\_part\_delta.text, end="", flush=True)  # Process the final response and its associated metadata  if event.is\_final\_response():  print() # Newline after the streaming response  if event.grounding\_metadata:  print(f" (Source Attributions: {len(event.grounding\_metadata.grounding\_attributions)} sources found)")  else:  print(" (No grounding metadata found)")  print("-" \* 30)  except Exception as e:  print(f"\nAn error occurred: {e}")  print("Please ensure your datastore ID is correct and that the service account has the necessary permissions.")  print("-" \* 30)  # --- Run Example ---  async def run\_vsearch\_example():  # Replace with a question relevant to YOUR datastore content  await call\_vsearch\_agent\_async("Summarize the main points about the Q2 strategy document.")  await call\_vsearch\_agent\_async("What safety procedures are mentioned for lab X?")  # --- Execution ---  if \_\_name\_\_ == "\_\_main\_\_":  if not DATASTORE\_ID:  print("Error: DATASTORE\_ID environment variable is not set.")  else:  try:  asyncio.run(run\_vsearch\_example())  except RuntimeError as e:  # This handles cases where asyncio.run is called in an environment  # that already has a running event loop (like a Jupyter notebook).  if "cannot be called from a running event loop" in str(e):  print("Skipping execution in a running event loop. Please run this script directly.")  else:  raise e |

Overall, this code provides a basic framework for building a conversational AI application that leverages Vertex AI Search to answer questions based on information stored in a datastore. It demonstrates how to define an agent, set up a runner, and interact with the agent asynchronously while streaming the response. The focus is on retrieving and synthesizing information from a specific datastore to answer user queries.

**Vertex Extensions:** A Vertex AI extension is a structured API wrapper that enables a model to connect with external APIs for real-time data processing and action execution. Extensions offer enterprise-grade security, data privacy, and performance guarantees. They can be used for tasks like generating and running code, querying websites, and analyzing information from private datastores. Google provides prebuilt extensions for common use cases like Code Interpreter and Vertex AI Search, with the option to create custom ones. The primary benefit of extensions includes strong enterprise controls and seamless integration with other Google products. The key difference between extensions and function calling lies in their execution: Vertex AI automatically executes extensions, whereas function calls require manual execution by the user or client.

**At a Glance**

**What:** LLMs are powerful text generators, but they are fundamentally disconnected from the outside world. Their knowledge is static, limited to the data they were trained on, and they lack the ability to perform actions or retrieve real-time information. This inherent limitation prevents them from completing tasks that require interaction with external APIs, databases, or services. Without a bridge to these external systems, their utility for solving real-world problems is severely constrained.

**Why:** The Tool Use pattern, often implemented via function calling, provides a standardized solution to this problem. It works by describing available external functions, or "tools," to the LLM in a way it can understand. Based on a user's request, the agentic LLM can then decide if a tool is needed and generate a structured data object (like a JSON) specifying which function to call and with what arguments. An orchestration layer executes this function call, retrieves the result, and feeds it back to the LLM. This allows the LLM to incorporate up-to-date, external information or the result of an action into its final response, effectively giving it the ability to act.

**Rule of thumb:** Use the Tool Use pattern whenever an agent needs to break out of the LLM's internal knowledge and interact with the outside world. This is essential for tasks requiring real-time data (e.g., checking weather, stock prices), accessing private or proprietary information (e.g., querying a company's database), performing precise calculations, executing code, or triggering actions in other systems (e.g., sending an email, controlling smart devices).

**Visual summary:**

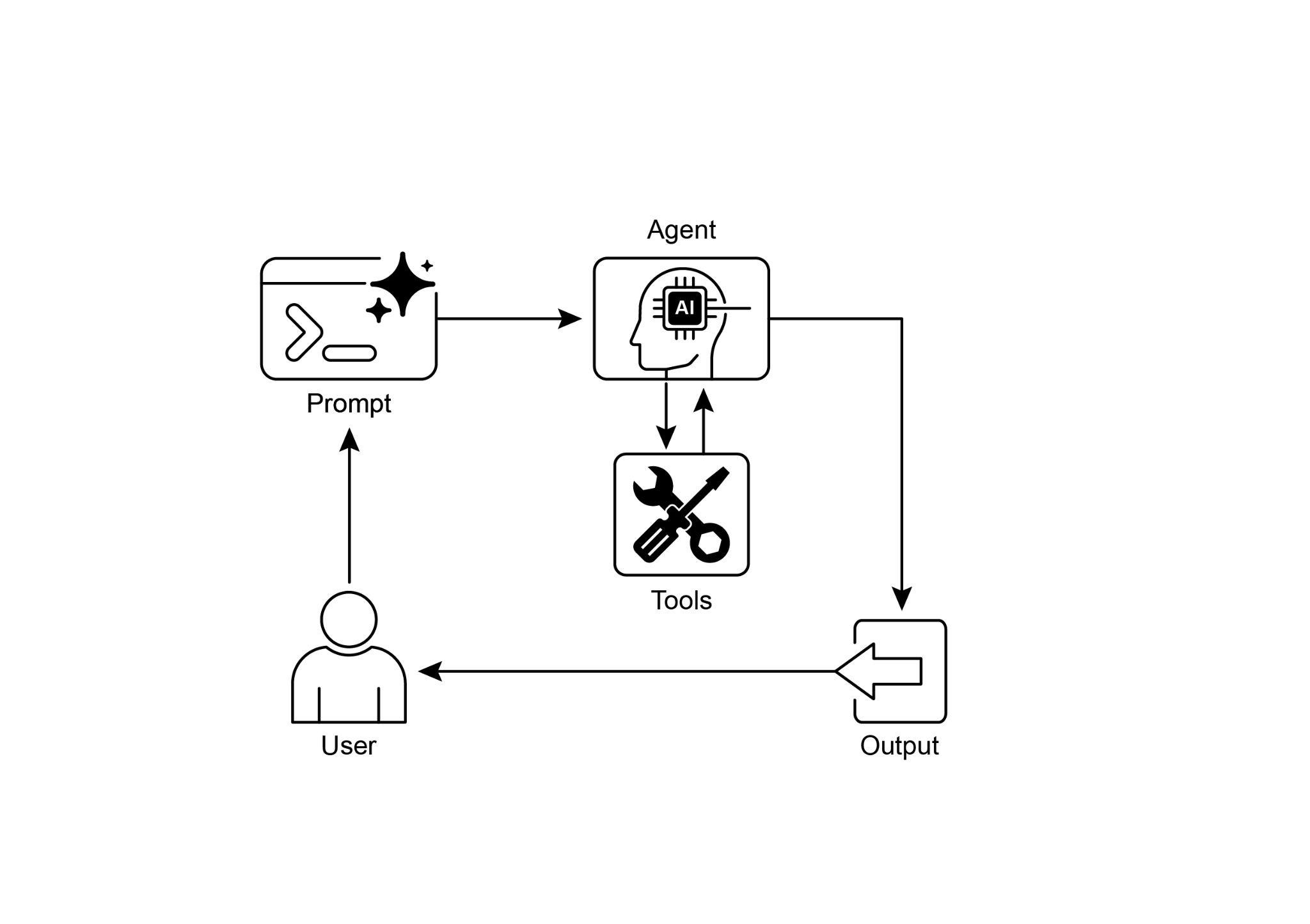


Fig.2: Tool use design pattern

**Key Takeaways**

* Tool Use (Function Calling) allows agents to interact with external systems and access dynamic information.
* It involves defining tools with clear descriptions and parameters that the LLM can understand.
* The LLM decides when to use a tool and generates structured function calls.
* Agentic frameworks execute the actual tool calls and return the results to the LLM.
* Tool Use is essential for building agents that can perform real-world actions and provide up-to-date information.
* LangChain simplifies tool definition using the @tool decorator and provides create\_tool\_calling\_agent and AgentExecutor for building tool-using agents.
* Google ADK has a number of very useful pre-built tools such as Google Search, Code Execution and Vertex AI Search Tool.

**Conclusion**

The Tool Use pattern is a critical architectural principle for extending the functional scope of large language models beyond their intrinsic text generation capabilities. By equipping a model with the ability to interface with external software and data sources, this paradigm allows an agent to perform actions, execute computations, and retrieve information from other systems. This process involves the model generating a structured request to call an external tool when it determines that doing so is necessary to fulfill a user's query. Frameworks such as LangChain, Google ADK, and Crew AI offer structured abstractions and components that facilitate the integration of these external tools. These frameworks manage the process of exposing tool specifications to the model and parsing its subsequent tool-use requests. This simplifies the development of sophisticated agentic systems that can interact with and take action within external digital environments.

**References**

1. LangChain Documentation (Tools): <https://python.langchain.com/docs/integrations/tools/>
2. Google Agent Developer Kit (ADK) Documentation (Tools): <https://google.github.io/adk-docs/tools/>
3. OpenAI Function Calling Documentation: <https://platform.openai.com/docs/guides/function-calling>
4. CrewAI Documentation (Tools): <https://docs.crewai.com/concepts/tools>

**第五章\_工具使用**

第5章：工具使用（函数调用）

**工具使用模式概述**

到目前为止，我们已经讨论了主要涉及编排语言模型之间的交互以及管理智能体内部工作流程中信息流的智能体模式（链式、路由、并行化、反思）。然而，要使智能体真正发挥作用并与现实世界或外部系统进行交互，它们需要具备使用工具的能力。

工具使用模式通常通过一种称为函数调用的机制来实现，它使智能体能够与外部API、数据库、服务进行交互，甚至执行代码。该模式允许智能体核心的大语言模型（LLM）根据用户请求或任务的当前状态来决定何时以及如何使用特定的外部函数。

该过程通常包括：

1. **工具定义：**外部函数或功能被定义并向大语言模型（LLM）进行描述。此描述包括函数的用途、名称、接受的参数，以及参数的类型和描述。
2. **大语言模型决策：**大语言模型接收用户请求和可用的工具定义。基于对请求和工具的理解，大语言模型决定是否需要调用一个或多个工具来满足请求。
3. **函数调用生成：**如果大语言模型（LLM）决定使用某个工具，它会生成一个结构化输出（通常是一个JSON对象），该输出指定要调用的工具名称以及从用户请求中提取的要传递给它的参数。
4. **工具执行：**代理框架或编排层拦截此结构化输出。它识别请求的工具，并使用提供的参数执行实际的外部函数。
5. **观察/结果：**工具执行的输出或结果会返回给智能体。
6. **大语言模型处理（可选但常见）：**大语言模型接收工具的输出作为上下文，并利用它来形成对用户的最终响应，或决定工作流程中的下一步（这可能涉及调用另一个工具、进行反思或提供最终答案）。

这种模式至关重要，因为它打破了大语言模型（LLM）训练数据的局限性，使其能够访问最新信息、执行内部无法完成的计算、与用户特定数据进行交互，或触发现实世界的行动。函数调用是一种技术机制，它弥合了大语言模型的推理能力与可用的大量外部功能之间的差距。

虽然“函数调用”恰当地描述了调用特定的、预定义的代码函数，但考虑更广泛的“工具调用”概念是很有用的。这个更宽泛的术语承认，智能体的能力可以远远超出简单的函数执行。“工具”可以是传统的函数，但也可以是复杂的 API 端点、对数据库的请求，甚至是指向另一个专门智能体的指令。这种视角使我们能够设想更复杂的系统，例如，主要智能体可以将复杂的数据分析任务委托给专门的“分析智能体”，或者通过其 API 查询外部知识库。从“工具调用”的角度思考，能更好地体现智能体作为跨数字资源和其他智能实体的多样化生态系统的协调者的全部潜力。

像LangChain、LangGraph和谷歌代理开发工具包（ADK）这样的框架，为定义工具并将其集成到代理工作流程中提供了强大支持，通常会利用Gemini或OpenAI系列等现代大语言模型（LLM）的原生函数调用能力。在这些框架的“画布”上，你可以定义工具，然后配置代理（通常是大语言模型代理），使其能够感知并使用这些工具。

工具使用是构建强大、交互式且具有外部感知能力的智能体的基石模式。

**实际应用与用例**

工具使用模式几乎适用于任何场景，在这些场景中，智能体需要超越文本生成，以执行操作或检索特定的动态信息：

1. 从外部来源检索信息：

访问不在大语言模型训练数据中的实时数据或信息。

* **用例：**一个天气代理。
* **工具：**一个天气 API，它接收一个地点信息并返回当前的天气状况。
* **代理流程：**用户询问“伦敦的天气如何？”，大语言模型（LLM）识别出需要使用天气工具，调用该工具并传入“伦敦”，工具返回数据，大语言模型将数据整理成用户友好的响应。

2. 与数据库和 API 交互：

对结构化数据执行查询、更新或其他操作。

* **用例：**电子商务代理。
* **工具：**用于检查产品库存、获取订单状态或处理付款的 API 调用。
* **代理流程：**用户询问“产品X是否有库存？”，大语言模型调用库存API，工具返回库存数量，大语言模型告知用户库存状态。

3. 执行计算和数据分析：

使用外部计算器、数据分析库或统计工具。

* **用例：**金融代理。
* **工具：**计算器功能、股票市场数据 API、电子表格工具。
* **代理流程：**用户询问 “AAPL的当前股价是多少，若我以150美元的价格买入100股，潜在利润是多少？”，大语言模型调用股票API获取当前股价，然后调用计算器工具获取结果，最后格式化响应。

4. 发送通信：

发送电子邮件、消息，或向外部通信服务进行 API 调用。

* **用例：**个人助理代理。
* **工具：**一个电子邮件发送 API。
* **代理流程：**用户说：“给约翰发一封关于明天会议的电子邮件。”，大语言模型调用电子邮件工具，其中收件人、主题和正文是从请求中提取的。

5. 执行代码：

在安全环境中运行代码片段以执行特定任务。

* **用例：**一个编码辅助代理。
* **工具：**代码解释器。
* **代理流程：**用户提供一段Python代码片段并询问“这段代码的作用是什么？”，大语言模型（LLM）使用解释器工具运行代码并分析其输出。

6. 控制其他系统或设备：

与智能家居设备、物联网平台或其他互联系统进行交互。

* **用例：**智能家居代理。
* **工具：**用于控制智能灯的 API。
* **智能体流程：**用户说：“关闭客厅的灯。”大语言模型（LLM）调用智能家居工具并附带命令和目标设备。

工具使用是将语言模型从文本生成器转变为能够在数字或物理世界中感知、推理和行动的智能体的关键（见图1）

图1：智能体使用工具的一些示例

**实践代码示例（LangChain）**

在LangChain框架内实现工具使用是一个两阶段的过程。首先，定义一个或多个工具，通常是通过封装现有的Python函数或其他可运行组件来实现。随后，将这些工具绑定到语言模型上，从而使模型在确定需要调用外部函数来满足用户查询时，能够生成结构化的工具使用请求。

以下实现将通过首先定义一个简单的函数来模拟信息检索工具，以此来演示这一原则。随后，将构建并配置一个智能体，使其能够利用该工具来响应用户输入。运行此示例需要安装核心的LangChain库和特定模型的提供程序包。此外，通常通过在本地环境中配置的API密钥对所选语言模型服务进行正确身份验证，是必要的先决条件。

|  |
| --- |
| 导入 os, getpass  导入异步I/O库  导入nest\_asyncio  从 typing 导入 List  from dotenv import load\_dotenv  导入日志记录模块  从 langchain\_google\_genai 导入 ChatGoogleGenerativeAI  from langchain\_core.prompts import ChatPromptTemplate  from langchain\_core.tools import tool as langchain\_tool  从 langchain.agents 导入 create\_tool\_calling\_agent、AgentExecutor  #取消注释  #安全地提示用户，并将API密钥设置为环境变量  os.environ["GOOGLE\_API\_KEY"] = getpass.getpass("请输入你的 Google API 密钥: ")  os.environ["OPENAI\_API\_KEY"] = getpass.getpass("请输入你的OpenAI API密钥: ")  try:  # 需要一个具备函数/工具调用能力的模型。  llm = ChatGoogleGenerativeAI(model="gemini-2.0-flash", temperature=0)  print(f"✅ 语言模型已初始化：{llm.model}")  except Exception as e:  print(f"🛑 初始化语言模型时出错: {e}")  llm = None  # --- 定义一个工具 ---  @langchain\_tool  def search\_information(query: str) -> str:  """  提供有关给定主题的事实信息。使用此工具查找短语的答案  如'法国首都'或'伦敦的天气如何？'。  """  print(f"\n--- 🛠️ 调用工具: search\_information，查询内容: '{query}' ---")  # 使用预定义结果字典模拟搜索工具。  simulated\_results = {  "伦敦的天气": "伦敦目前的天气多云，气温为15°C。",  "法国的首都": "法国的首都是巴黎。",  "地球人口": "地球的估计人口约为80亿人。",  "最高的山峰": "珠穆朗玛峰是海拔最高的山峰。",  "default": f"模拟的 '{query}' 搜索结果：未找到具体信息，但该主题似乎很有趣。"  }  result = simulated\_results.get(query.lower(), simulated\_results["default"])  print(f"--- 工具结果: {result} ---")  返回结果  工具 = [搜索信息]  # --- 创建一个工具调用代理 ---  如果大语言模型存在：  # 此提示模板需要一个 `agent\_scratchpad` 占位符，用于存放智能体的内部步骤。  agent\_prompt = ChatPromptTemplate.from\_messages([  ("system", "你是一个乐于助人的助手。"),  ("人类", "{输入}"),  ("占位符", "{代理暂存区}"),  ])  # 创建代理，将大语言模型、工具和提示绑定在一起。  agent = create\_tool\_calling\_agent(llm, tools, agent\_prompt)  # AgentExecutor是运行时，它调用代理并执行所选工具。  # 此处不需要 'tools' 参数，因为它们已经绑定到代理。  agent\_executor = AgentExecutor(agent=agent, verbose=True, tools=tools)  async def run\_agent\_with\_tool(query: str):  """使用查询调用代理执行器并打印最终响应。"""  print(f"\n--- 🏃 正在运行代理，查询内容为：'{query}' ---")  try:  response = await agent\_executor.ainvoke({"input": query})  print("\n--- ✅ 最终代理响应 ---")  print(response["output"])  except Exception as e:  print(f"\n🛑 代理执行期间发生错误：{e}")  async def main():  """并发运行所有代理查询。"""  tasks = [  run\_agent\_with\_tool("法国的首都是什么？"),  run\_agent\_with\_tool("伦敦的天气如何？"),  run\_agent\_with\_tool("给我讲讲关于狗的事情。") # 应该触发默认工具响应  ]  await asyncio.gather(\*tasks)  nest\_asyncio.apply()  asyncio.run(main()) |

该代码使用LangChain库和Google Gemini模型设置了一个工具调用代理。它定义了一个search\_information工具，该工具模拟为特定查询提供事实性答案。该工具针对“伦敦的天气”、“法国的首都”和“地球的人口”有预定义的响应，对于其他查询则有默认响应。初始化了一个ChatGoogleGenerativeAI模型，确保其具备工具调用能力。创建了一个ChatPromptTemplate来指导代理的交互。使用create\_tool\_calling\_agent函数将语言模型、工具和提示组合成一个代理。然后设置了一个AgentExecutor来管理代理的执行和工具调用。定义了异步函数run\_agent\_with\_tool，用于使用给定查询调用代理并打印结果。主异步函数准备了多个查询以并发运行。这些查询旨在测试search\_information工具的特定响应和默认响应。最后，asyncio.run(main())调用执行所有代理任务。该代码在进行代理设置和执行之前，会检查LLM是否成功初始化。

**实践代码示例（CrewAI）**

此代码提供了一个在CrewAI框架内实现函数调用（工具）的实际示例。它设置了一个简单的场景，其中一个代理配备了一个用于查找信息的工具。该示例具体展示了如何使用这个代理和工具获取模拟的股票价格。

|  |
| --- |
| # pip install crewai langchain-openai  import os  从crewai导入Agent、Task、Crew  从crewai.tools导入工具  导入日志记录模块  # --- 最佳实践：配置日志记录 ---  # 基本的日志设置有助于调试和跟踪机组人员的执行情况。  logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(levelname)s - %(message)s')  # --- 设置你的 API 密钥 ---  # 在生产环境中，建议使用更安全的密钥管理方法  # 例如在运行时加载的环境变量或密钥管理器。  #  # 设置所选大语言模型（LLM）提供商的环境变量（例如，OPENAI\_API\_KEY）  # os.environ["OPENAI\_API\_KEY"] = "YOUR\_API\_KEY"  # os.environ["OPENAI\_MODEL\_NAME"] = "gpt-4o"  # --- 1. 重构工具：返回干净数据 ---  #该工具现在返回原始数据（浮点数）或抛出标准的 Python 错误。  #这使其更具可复用性，并促使代理正确处理结果。  @工具("股票价格查询工具")  def get\_stock\_price(ticker: str) -> float:  """  获取给定股票代码的最新模拟股票价格。  以浮点数形式返回价格。如果未找到股票代码，则引发 ValueError 异常。  """  logging.info(f"工具调用：获取股票代码为 '{ticker}' 的股票价格")  simulated\_prices = {  "AAPL": 178.15,  "GOOGL": 1750.30,  "微软": 425.50,  }  price = simulated\_prices.get(ticker.upper())  if price is not None:  返回价格  否则:  # 抛出特定错误比返回字符串更好。  代理能够处理异常，并决定下一步行动。  raise ValueError(f"未找到股票代码为 '{ticker.upper()}' 的模拟价格。")  # --- 2. 定义智能体 ---  # 代理定义保持不变，但现在将利用改进后的工具。  financial\_analyst\_agent = Agent(  职位='高级财务分析师'  目标='使用提供的工具分析股票数据并报告关键价格'。  背景故事="你是一位经验丰富的金融分析师，擅长利用数据源查找股票信息。你提供清晰、直接的答案。",  verbose=True,  工具=[获取股票价格]  允许委托可能有用，但对于这个简单的任务来说并非必要。  allow\_delegation=False,  )  # --- 3. 细化任务：更清晰的说明和错误处理 ---  #任务描述更加具体，指导代理如何做出反应  # 既适用于成功的数据检索，也适用于潜在的错误。  analyze\_aapl\_task = Task(  描述=(  当前苹果公司（股票代码：AAPL）的模拟股价是多少？  使用“股票价格查询工具”来查找。  如果未找到股票代码，你必须报告无法获取价格。  ),  预期输出=(  一个清晰的句子，说明苹果公司（AAPL）的模拟股价。  例如：“苹果公司（AAPL）的模拟股价为178.15美元。”  如果找不到价格，请明确说明。  ),  代理=金融分析师代理  )  # --- 4. 组建机组人员 ---  # 团队负责协调代理和任务如何协同工作。  financial\_crew = Crew(  agents=[financial\_analyst\_agent],  tasks=[analyze\_aapl\_task],  verbose=True # 在生产环境中设置为 False 以减少详细日志  )  # --- 5. 在主执行块中运行Crew ---  # 使用 \_\_name\_\_ == "\_\_main\_\_": 代码块是 Python 的标准最佳实践。  def main():  """运行机组的主要功能。"""  # 在开始前检查 API 密钥，以避免运行时错误。  if not os.environ.get("OPENAI\_API\_KEY"):  print("错误：OPENAI\_API\_KEY环境变量未设置。")  print("请在运行脚本前设置好。")  返回  print("\n## 启动财务团队...")  print("---------------------------------")    # 启动方法开始执行。  result = financial\_crew.kickoff()  print("\n---------------------------------")  print("## 机组执行完成。")  print("\n最终结果:\n", result)  if \_\_name\_\_ == "\_\_main\_\_":  main() |

此代码展示了一个使用Crew.ai库的简单应用程序，用于模拟财务分析任务。它定义了一个自定义工具get\_stock\_price，用于模拟查询预定义股票代码的股价。该工具设计为对有效股票代码返回一个浮点数，对无效股票代码则抛出ValueError异常。创建了一个名为financial\_analyst\_agent的Crew.ai代理，其角色为高级财务分析师。该代理被赋予get\_stock\_price工具以进行交互。定义了一个任务analyze\_aapl\_task，专门指示代理使用该工具查找AAPL的模拟股价。任务描述中包含了在使用工具时如何处理成功和失败情况的明确说明。组建了一个团队，由financial\_analyst\_agent和analyze\_aapl\_task组成。为代理和团队都启用了详细模式，以便在执行过程中提供详细的日志记录。脚本的主要部分在标准的if \_\_name\_\_ == "\_\_main\_\_":块中使用kickoff()方法运行团队的任务。在启动团队之前，它会检查是否设置了OPENAI\_API\_KEY环境变量，这是代理正常运行所必需的。然后将团队执行的结果（即任务的输出）打印到控制台。代码还包含基本的日志配置，以便更好地跟踪团队的操作和工具调用。它使用环境变量进行API密钥管理，不过它指出在生产环境中建议使用更安全的方法。简而言之，核心逻辑展示了如何定义工具、代理和任务，以在Crew.ai中创建协作工作流。

**实践代码（谷歌ADK）**

**谷歌代理开发者套件（ADK）包含一个原生集成工具库，这些工具可直接融入代理的功能中。**

**谷歌搜索：此类组件的一个主要示例是谷歌搜索工具。该工具作为谷歌搜索引擎的直接接口，使智能体具备执行网络搜索和获取外部信息的功能。**

|  |
| --- |
| from google.adk.agents import Agent  from google.adk.runners import Runner  from google.adk.sessions import InMemorySessionService  from google.adk.tools import google\_search  从google.genai导入types  导入nest\_asyncio  导入异步I/O库  # 定义会话设置和代理执行所需的变量  APP\_NAME="谷歌搜索\_代理"  USER\_ID="user1234"  SESSION\_ID="1234"  # 定义可访问搜索工具的代理  root\_agent = ADKAgent(  name="basic\_search\_agent",  model="gemini-2.0-flash-exp",  描述="使用谷歌搜索回答问题的代理。",  指令="我可以通过搜索互联网来回答你的问题。有任何问题尽管问我！",  tools=[google\_search] # Google Search是一个预建工具，用于执行谷歌搜索。  )  # 座席交互  async def call\_agent(query):  """  使用查询调用代理的辅助函数。  """  #会话和运行器  session\_service = InMemorySessionService()  session = await session\_service.create\_session(app\_name=APP\_NAME, user\_id=USER\_ID, session\_id=SESSION\_ID)  runner = Runner(agent=root\_agent, app\_name=APP\_NAME, session\_service=session\_service)  content = types.Content(role='user', parts=[types.Part(text=query)])  events = runner.run(user\_id=USER\_ID, session\_id=SESSION\_ID, new\_message=content)  for event in events:  if event.is\_final\_response():  final\_response = event.content.parts[0].text  print("代理回复: ", final\_response)  nest\_asyncio.apply()  asyncio.run(call\_agent("最新的人工智能新闻是什么？")) |

此代码展示了如何创建和使用由 Google ADK for Python 驱动的基本代理。该代理旨在通过使用 Google 搜索作为工具来回答问题。首先，导入来自 IPython、google.adk 和 google.genai 的必要库。定义应用程序名称、用户 ID 和会话 ID 的常量。创建一个名为 "basic\_search\_agent" 的代理实例，并带有描述和说明，表明其用途。它被配置为使用 Google 搜索工具，这是 ADK 提供的预构建工具。初始化一个 InMemorySessionService（见第 8 章）来管理代理的会话。为指定的应用程序、用户和会话 ID 创建一个新会话。实例化一个 Runner，将创建的代理与会话服务关联起来。这个 Runner 负责在会话中执行代理的交互。定义一个辅助函数 call\_agent，以简化向代理发送查询并处理响应的过程。在 call\_agent 内部，用户的查询被格式化为角色为 'user' 的 types.Content 对象。调用 runner.run 方法，传入用户 ID、会话 ID 和新的消息内容。runner.run 方法返回一个事件列表，代表代理的操作和响应。代码遍历这些事件以找到最终响应。如果一个事件被识别为最终响应，则提取该响应的文本内容。然后将提取的代理响应打印到控制台。最后，使用查询 "what's the latest ai news?" 调用 call\_agent 函数。来展示代理的实际操作。

**代码执行：**Google ADK具备用于特定任务的集成组件，包括一个动态代码执行环境。内置的代码执行工具为智能体提供了一个沙箱化的Python解释器。这使得模型能够编写和运行代码，以执行计算任务、操作数据结构并执行程序脚本。这种功能对于解决需要确定性逻辑和精确计算的问题至关重要，而这些问题仅靠概率性语言生成是无法解决的。

|  |
| --- |
| 导入 os, getpass  导入异步I/O库  导入nest\_asyncio  从 typing 导入 List  from dotenv import load\_dotenv  导入日志记录模块  从google.adk.agents导入Agent作为ADKAgent，LlmAgent  from google.adk.runners import Runner  from google.adk.sessions import InMemorySessionService  from google.adk.tools import google\_search  from google.adk.code\_executors import BuiltInCodeExecutor  从google.genai导入types  # 定义会话设置和代理执行所需的变量  APP\_NAME="计算器"  USER\_ID="user1234"  SESSION\_ID="session\_code\_exec\_async"  # 代理定义  code\_agent = LlmAgent(  name="calculator\_agent",  模型="gemini-2.0-flash",  code\_executor=BuiltInCodeExecutor(),  指令="你是一个计算器代理。  当给定一个数学表达式时，编写并执行Python代码来计算结果。  仅以纯文本形式返回最终数值结果，不使用Markdown或代码块。  """,  描述="执行Python代码以进行计算。",  )  # 代理交互（异步）  async def call\_agent\_async(query):  #会话和运行器  session\_service = InMemorySessionService()  session = await session\_service.create\_session(app\_name=APP\_NAME, user\_id=USER\_ID, session\_id=SESSION\_ID)  runner = Runner(agent=code\_agent, app\_name=APP\_NAME, session\_service=session\_service)  content = types.Content(role='user', parts=[types.Part(text=query)])  print(f"\n--- 正在运行查询: {query} ---")  final\_response\_text = "未捕获到最终文本响应。"  try:  # 使用 run\_async  async for event in runner.run\_async(user\_id=USER\_ID, session\_id=SESSION\_ID, new\_message=content):  print(f"事件ID: {event.id}, 作者: {event.author}")  # --- 首先检查特定部分 ---  # has\_specific\_part = False  if event.content and event.content.parts and event.is\_final\_response():  for part in event.content.parts: # 遍历所有部分  if part.executable\_code:  # 通过.code访问实际代码字符串  print(f" 调试信息：代理生成的代码：\n```python\n{part.executable\_code.code}\n```")  has\_specific\_part = True  elif part.code\_execution\_result:  # 正确访问结果和输出  print(f" 调试：代码执行结果：{part.code\_execution\_result.outcome} - 输出：\n{part.code\_execution\_result.output}")  has\_specific\_part = True  # 为了调试，还会打印在任何事件中找到的所有文本部分  elif part.text and not part.text.isspace():  print(f" 文本: '{part.text.strip()}'")  # 这里不要设置 has\_specific\_part=True，因为我们需要下面的最终响应逻辑  # --- 在特定部分之后检查最终响应 ---  text\_parts = [part.text for part in event.content.parts if part.text]  final\_result = "".join(text\_parts)  print(f"==> 最终代理响应: {final\_result}")  except Exception as e:  print(f"代理运行期间出错: {e}")  print("-" \* 30)  # 运行示例的主异步函数  async def main():  await call\_agent\_async("计算(5 + 7) \* 3的值")  await call\_agent\_async("10的阶乘是多少？")  # 执行主异步函数  try:  nest\_asyncio.apply()  asyncio.run(main())  except RuntimeError as e:  # 处理在已运行的事件循环（如 Jupyter/Colab）中运行 asyncio.run 时的特定错误  if "cannot be called from a running event loop" in str(e):  print("\n在现有的事件循环中运行（如Colab/Jupyter）。")  print("请在笔记本单元格中运行 `await main()`。")  # 如果在像笔记本这样的交互式环境中，你可能需要运行：  # await main()  else:  raise e # 重新抛出其他运行时错误 |

此脚本使用谷歌的代理开发工具包（ADK）创建一个代理，该代理通过编写和执行Python代码来解决数学问题。它定义了一个专门被指示充当计算器的LlmAgent，并为其配备了内置代码执行工具。主要逻辑位于call\_agent\_async函数中，该函数将用户的查询发送给代理的运行器，并处理产生的事件。在这个函数内部，一个异步循环遍历事件，打印生成的Python代码及其执行结果以进行调试。代码仔细区分这些中间步骤和包含数值答案的最终事件。最后，一个主函数使用两个不同的数学表达式运行代理，以展示其执行计算的能力。

**企业搜索：**此代码使用Python中的google.adk库定义了一个Google ADK应用程序。它专门使用了VSearchAgent，该代理旨在通过搜索指定的Vertex AI搜索数据存储来回答问题。代码初始化了一个名为“q2\_strategy\_vsearch\_agent”的VSearchAgent，提供了描述、要使用的模型（“gemini-2.0-flash-exp”）以及Vertex AI搜索数据存储的ID。DATASTORE\_ID应设置为环境变量。然后，它为代理设置了一个Runner，使用InMemorySessionService来管理对话历史。定义了一个异步函数call\_vsearch\_agent\_async来与代理进行交互。该函数接收一个查询，构造一个消息内容对象，并调用Runner的run\_async方法将查询发送给代理。该函数随后将代理的响应流式传输回控制台。它还会打印有关最终响应的信息，包括数据存储中的任何来源归因。包含错误处理以捕获代理执行期间的异常，提供有关潜在问题（如数据存储ID不正确或缺少权限）的信息性消息。另一个异步函数run\_vsearch\_example用于演示如何使用示例查询调用代理。主执行块检查DATASTORE\_ID是否已设置，然后使用asyncio.run运行示例。它还包含一个检查，以处理代码在已经有运行中的事件循环的环境（如Jupyter笔记本）中运行的情况。

|  |
| --- |
| 导入异步I/O库  从google.genai导入types  from google.adk import agents  from google.adk.runners import Runner  from google.adk.sessions import InMemorySessionService  import os  # --- 配置 ---  # 确保你已经设置了 GOOGLE\_API\_KEY 和 DATASTORE\_ID 环境变量  # 例如：  # os.environ["GOOGLE\_API\_KEY"] = "YOUR\_API\_KEY"  # os.environ["DATASTORE\_ID"] = "YOUR\_DATASTORE\_ID"  DATASTORE\_ID = os.environ.get("DATASTORE\_ID")  # --- 应用常量 ---  APP\_NAME = "vsearch\_app"  USER\_ID = "user\_123" # 示例用户ID  SESSION\_ID = "session\_456" # 示例会话ID  # --- 代理定义（根据指南中的新模型更新） ---  vsearch\_agent = agents.VSearchAgent(  name="Q2策略向量搜索代理",  description="使用Vertex AI Search回答有关Q2战略文件的问题。",  model="gemini-2.0-flash-exp", # 根据指南示例更新的模型  datastore\_id=DATASTORE\_ID,  model\_parameters={"temperature": 0.0}  )  # --- 运行器和会话初始化 ---  runner = Runner(  代理=vsearch\_agent，  app\_name=APP\_NAME,  session\_service=InMemorySessionService(),  )  # --- 代理调用逻辑 ---  async def call\_vsearch\_agent\_async(query: str):  """初始化会话并流式传输代理的响应。"""  print(f"用户: {query}")  print("代理: ", end="", flush=True)  try:  # 正确构建消息内容  content = types.Content(role='user', parts=[types.Part(text=query)])  # 处理从异步运行器到达的事件  async for event in runner.run\_async(  user\_id=用户ID,  session\_id=SESSION\_ID,  new\_message=content  ):  # 用于响应文本的逐令牌流式传输  if hasattr(event, 'content\_part\_delta') and event.content\_part\_delta:  print(event.content\_part\_delta.text, end="", flush=True)  # 处理最终响应及其关联的元数据  if event.is\_final\_response():  print() # 流式响应后的换行符  if event.grounding\_metadata:  print(f" (来源归因：找到 {len(event.grounding\_metadata.grounding\_attributions)} 个来源)")  否则:  print(" (未找到接地元数据)")  print("-" \* 30)  except Exception as e:  print(f"\n发生错误: {e}")  print("请确保您的数据存储ID正确，且服务号拥有必要的权限。")  print("-" \* 30)  # ---运行示例---  async def run\_vsearch\_example():  # 替换为与您的数据存储内容相关的问题  await call\_vsearch\_agent\_async("总结Q2战略文档的要点。")  await call\_vsearch\_agent\_async("实验室X提到了哪些安全程序？")  # --- 执行 ---  if \_\_name\_\_ == "\_\_main\_\_":  if not DATASTORE\_ID:  print("错误：DATASTORE\_ID环境变量未设置。")  否则:  try:  asyncio.run(run\_vsearch\_example())  except RuntimeError as e:  # 这处理了在某个环境中调用 asyncio.run 的情况  （例如 Jupyter 笔记本）已经有一个正在运行的事件循环。  if "cannot be called from a running event loop" in str(e):  print("在正在运行的事件循环中跳过执行。请直接运行此脚本。")  否则:  raise e |

总体而言，此代码为构建一个利用Vertex AI Search根据存储在数据存储中的信息回答问题的对话式AI应用程序提供了一个基本框架。它展示了如何定义一个代理、设置一个运行器，并在流式传输响应的同时与代理进行异步交互。重点在于从特定的数据存储中检索和合成信息，以回答用户查询。

**顶点扩展：**顶点AI扩展是一种结构化的API包装器，它使模型能够与外部API连接，以进行实时数据处理和动作执行。扩展提供企业级的安全性、数据隐私和性能保证。它们可用于生成和运行代码、查询网站以及分析来自私有数据存储的信息等任务。谷歌为代码解释器和顶点AI搜索等常见用例提供预构建的扩展，同时也支持创建自定义扩展。扩展的主要优势包括强大的企业控制能力以及与其他谷歌产品的无缝集成。扩展与函数调用的关键区别在于执行方式：顶点AI会自动执行扩展，而函数调用则需要用户或客户端手动执行。

**概览**

**问题：**大语言模型是强大的文本生成器，但它们从根本上与外部世界脱节。它们的知识是静态的，局限于训练数据，并且缺乏执行操作或检索实时信息的能力。这种内在的局限性使它们无法完成需要与外部应用程序编程接口（API）、数据库或服务进行交互的任务。如果没有与这些外部系统的桥梁，它们解决现实世界问题的效用将受到严重限制。

**原因：**工具使用模式（通常通过函数调用实现）为这个问题提供了标准化的解决方案。它的工作原理是，以大语言模型（LLM）能够理解的方式，向其描述可用的外部函数或“工具”。基于用户的请求，具有自主能力的大语言模型可以决定是否需要使用工具，并生成一个结构化的数据对象（如JSON），指定要调用的函数及其参数。编排层执行这个函数调用，获取结果，并将其反馈给大语言模型。这使得大语言模型能够将最新的外部信息或某个操作的结果纳入其最终响应中，从而有效地赋予其行动能力。

**经验法则：**每当智能体需要突破大语言模型（LLM）的内部知识并与外部世界进行交互时，就使用工具使用模式。这对于需要实时数据的任务（例如，查询天气、股票价格）、访问私有或专有信息（例如，查询公司数据库）、执行精确计算、执行代码或触发其他系统中的操作（例如，发送电子邮件、控制智能设备）至关重要。

**可视化总结：**

图2：工具使用设计模式

**要点总结**

* 工具使用（函数调用）允许智能体与外部系统进行交互并访问动态信息。
* 它涉及定义具有清晰描述和参数的工具，以便大语言模型（LLM）能够理解。
* 大语言模型（LLM）决定何时使用工具并生成结构化的函数调用。
* 能动框架执行实际的工具调用，并将结果返回给大语言模型。
* 工具使用对于构建能够执行现实世界行动并提供最新信息的智能体至关重要。
* LangChain使用@tool装饰器简化工具定义，并提供create\_tool\_calling\_agent和AgentExecutor来构建使用工具的智能体。
* Google ADK拥有许多非常有用的预建工具，如Google搜索、代码执行和Vertex AI搜索工具。

**结论**

工具使用模式是一项关键的架构原则，用于将大语言模型的功能范围扩展到其固有文本生成能力之外。通过赋予模型与外部软件和数据源交互的能力，这种范式使智能体能够执行操作、进行计算并从其他系统中检索信息。这一过程涉及模型在确定有必要时生成结构化请求，以调用外部工具来满足用户的查询。诸如LangChain、Google ADK和Crew AI等框架提供了结构化的抽象和组件，便于集成这些外部工具。这些框架管理向模型公开工具规范并解析其后续工具使用请求的过程。这简化了能够与外部数字环境交互并采取行动的复杂智能体系统的开发。

**参考文献**

1. LangChain文档（工具）：<https://python.langchain.com/docs/integrations/tools/>
2. Google Agent Developer Kit (ADK) 文档 (工具): <https://google.github.io/adk-docs/tools/>
3. OpenAI函数调用文档：<https://platform.openai.com/docs/guides/function-calling>
4. CrewAI文档（工具）：<https://docs.crewai.com/concepts/tools>