Google Agent Development Kit (ADK): Tools and Integrations

Overview of Tools in ADK

In the context of ADK, a Tool represents a specific capability provided to an AI agent, enabling it to perform actions and interact with the world beyond its core text generation and reasoning abilities. What distinguishes capable agents from basic language models is often their effective use of tools.

Technically, a tool is typically a modular code component—like a Python function, a class method, or even another specialized agent—designed to execute a distinct, predefined task. These tasks often involve interacting with external systems or data.

Key Characteristics of Tools

Action-Oriented: Tools perform specific actions, such as: - Querying databases - Making API requests (e.g., fetching weather data, booking systems) - Searching the web - Executing code snippets - Retrieving information from documents (RAG) - Interacting with other software or services

Extends Agent capabilities: Tools empower agents to access real-time information, affect external systems, and overcome the knowledge limitations inherent in their training data.

Execute predefined logic: Tools execute specific, developer-defined logic. They do not possess their own independent reasoning capabilities like the agent's core Large Language Model (LLM). The LLM reasons about which tool to use, when, and with what inputs, but the tool itself just executes its designated function.

How Agents Use Tools

Agents leverage tools dynamically through mechanisms often involving function calling. The process generally follows these steps:

1. **Reasoning:** The agent's LLM analyzes its system instruction, conversation history, and user request.

2. **Selection:** Based on the analysis, the LLM decides on which tool, if any, to execute, based on the tools available to the agent and the docstrings that describe each tool.
3. **Invocation:** The LLM generates the required arguments (inputs) for the selected tool and triggers its execution.
4. **Observation:** The agent receives the output (result) returned by the tool.
5. **Finalization:** The agent incorporates the tool's output into its ongoing reasoning process to formulate the next response, decide the subsequent step, or determine if the goal has been achieved.

Types of Tools in ADK

ADK offers flexibility by supporting several types of tools:

1. Function Tools

Function tools are custom tools created by developers, tailored to specific application needs. They come in three main varieties:

a) Standard Function Tools

These are the most straightforward tools, created by transforming Python functions into tools that agents can use.

Key features: - Parameters should use JSON-serializable types (e.g., string, integer, list, dictionary) - The preferred return type is a dictionary, which allows structured responses with key-value pairs - Docstrings are crucial as they serve as the tool's description for the LLM - Best practice is to include a "status" key in return dictionaries to indicate success or failure

Example:

```
def get_weather_report(city: str) -> dict:
    """Retrieves the current weather report for a specified city.
    Returns:
        dict: A dictionary containing the weather information with a 'status' key ('success' or 'error') and a 'report' key with the weather details if successful, or an 'error_message' if an error occurred.
    """
    if city.lower() == "london":
        return {"status": "success", "report": "The current weather in London is cloudy with a temperature of 18 degrees Celsius and a chance of rain."}
    elif city.lower() == "paris":
        return {"status": "success", "report": "The weather in Paris is sunny with a
```

```
temperature of 25 degrees Celsius."}
    else:
        return {"status": "error", "error_message": f"Weather information for '{city}' is
not available."}

weather_tool = FunctionTool(func=get_weather_report)
```

b) Long Running Function Tools

These tools support asynchronous operations or tasks that take significant time to complete, allowing the agent to continue working on other tasks while waiting for results.

Key features: - Designed for operations that may take time to complete (API calls, complex calculations) - Can provide intermediate updates during execution - Allows agents to handle non-blocking operations efficiently

c) Agent-as-a-Tool

This approach allows using one agent as a tool for another agent, enabling specialized capabilities and modular design.

Key features: - Enables hierarchical agent structures - Allows specialized agents to handle specific tasks - Different from sub-agents in that the parent agent explicitly invokes the agent-tool rather than transferring control

2. Built-in Tools

These are ready-to-use tools provided by the ADK framework for common tasks, requiring minimal setup.

Key built-in tools include:

a) Google Search

The `google_search` tool allows agents to perform web searches using Google Search. It's simply added to the agent's tools list and is compatible with Gemini 2 models.

Usage:

```
 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

agent = Agent(
```

```
    model="gemini-2.0-flash-exp",
    name="search_agent",
    tools=[google_search]
)
```

b) Code Execution

Enables agents to execute code snippets, particularly useful for data analysis, calculations, or generating visualizations.

c) Retrieval Tools

Tools for retrieving information from documents, including Vertex AI Search integration for enterprise search capabilities.

3. Third-Party Tools

ADK is designed to be highly extensible, allowing seamless integration of tools from other AI agent frameworks.

Key integrations include:

a) LangChain Tools

ADK provides the `LangchainTool` wrapper to integrate tools from the LangChain ecosystem into ADK agents.

Example: Using Tavily search tool from LangChain:

```
 pip install langchain_community tavily-python

from google.adk.agents import Agent
from google.adk.tools import LangchainTool
from langchain_community.tools.tavily_search import TavilySearchResults

tavily_tool = LangchainTool(tool=TavilySearchResults(max_results=5))

agent = Agent(
    model="gemini-2.0-flash-exp",
    name="search_agent",
    tools=[tavily_tool]
)
```

b) CrewAI Tools

Similar to LangChain, ADK can integrate tools from the CrewAI framework, such as the Serper API for web search.

4. Google Cloud Tools

These tools make it easier to connect agents to Google Cloud's products and services with minimal code.

Key capabilities include:

a) API Hub Integration

The `ApiHubToolset` lets you turn any documented API from Apigee API Hub into a tool with a few lines of code, providing secure connections to enterprise APIs.

b) Integration Connectors

Provides access to 100+ prebuilt connectors to enterprise systems like Salesforce, Workday, and SAP.

c) Application Integration Workflows

Allows using existing Application Integration workflows as tools for your agent.

d) Database Access via Toolbox

Enables agents to interact with databases like Spanner, AlloyDB, Postgres, and more using the MCP Toolbox.

5. MCP Tools

Model Context Protocol (MCP) is an open standard designed to standardize how Large Language Models (LLMs) like Gemini and Claude communicate with external applications, data sources, and tools.

Key aspects:

a) MCP Client Mode

ADK agents can act as MCP clients, leveraging tools provided by external MCP servers. This allows integration with file systems, Google Maps, and other services that expose MCP interfaces.

b) MCP Server Mode

ADK tools can be exposed as an MCP server, making them accessible to any MCP client, not just ADK agents.

6. OpenAPI Tools

ADK simplifies interacting with external REST APIs by automatically generating callable tools directly from an OpenAPI Specification (v3.x).

Key components:

a) OpenAPIToolset

This is the primary class for OpenAPI integration. You initialize it with your OpenAPI specification, and it handles the parsing and generation of tools.

b) RestApiTool

This class represents a single, callable API operation (like GET /pets/{petId} or POST /pets). OpenAPIToolset creates one RestApiTool instance for each operation defined in your spec.

Example usage:

```
from google.adk.tools import OpenAPIToolset

# Initialize with an OpenAPI spec (from file, URL, or dict)
openapi_tools = OpenAPIToolset(spec_path="path/to/petstore.json")

# Add the generated tools to your agent
agent = Agent(
    model="gemini-2.0-flash-exp",
    name="pet_store_agent",
    tools=openapi_tools.get_tools()
)
```

Referencing Tools in Agent's Instructions

Within an agent's instructions, you can directly reference a tool by using its function name. If the tool's function name and docstring are sufficiently descriptive, your instructions can primarily focus on when the LLM should utilize the tool.

It is crucial to clearly instruct the agent on how to handle different return values that a tool might produce. For example, if a tool returns an error message, your instructions

should specify whether the agent should retry the operation, give up on the task, or request additional information from the user.

Furthermore, ADK supports the sequential use of tools, where the output of one tool can serve as the input for another. When implementing such workflows, it's important to describe the intended sequence of tool usage within the agent's instructions to guide the model through the necessary steps.

Conclusion

The ADK tools ecosystem provides a comprehensive and flexible approach to extending agent capabilities. From simple function tools to complex integrations with Google Cloud services, third-party frameworks, and external APIs, ADK offers developers multiple ways to enhance their agents' abilities to interact with the world. This rich tool ecosystem is a key differentiator that enables the creation of sophisticated, capable AI applications that can perform real-world tasks effectively.