Tools - Agent Development Kit

Source URL: https://google.github.io/adk-docs/tools/

Tools¶

What is a Tool?¶

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/ Java 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.

Agent tool call

Key Characteristics¶

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

Execute predefined logic: Crucially, 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 describes 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.

Think of the tools as a specialized toolkit that the agent's intelligent core (the LLM) can access and utilize as needed to accomplish complex tasks.

Tool Types in ADK¶

ADK offers flexibility by supporting several types of tools:

1. **Function Tools:** Tools created by you, tailored to your specific application's needs.
2. **Functions/Methods:** Define standard synchronous functions or methods in your code (e.g., Python def).
3. **Agents-as-Tools:** Use another, potentially specialized, agent as a tool for a parent agent.
4. **Long Running Function Tools:** Support for tools that perform asynchronous operations or take significant time to complete.
5. **Built-in Tools:** Ready-to-use tools provided by the framework for common tasks. Examples: Google Search, Code Execution, Retrieval-Augmented Generation (RAG).

6. **Third-Party Tools:** Integrate tools seamlessly from popular external libraries. Examples: LangChain Tools, CrewAI Tools.

Navigate to the respective documentation pages linked above for detailed information and examples for each tool type.

Referencing Tool 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 Large Language Model (LLM) should utilize the tool**. This promotes clarity and helps the model understand the intended use of each 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.

Example¶

The following example showcases how an agent can use tools by **referencing their function names in its instructions**. It also demonstrates how to guide the agent to **handle different return values from tools**, such as success or error messages, and how to orchestrate the **sequential use of multiple tools** to accomplish a task.

PythonJava

```
from google.adk.agents import Agent
from google.adk.tools import FunctionTool
from google.adk.runners import Runner
from google.adk.sessions import InMemorySessionService
from google.genai import types
```

```
APP_NAME="weather_sentiment_agent"
USER_ID="user1234"
SESSION_ID="1234"
MODEL_ID="gemini-2.0-flash"

# Tool 1
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 '
    """
    if city.lower() == "london":
        return {"status": "success", "report": "The current weather in
    elif city.lower() == "paris":
        return {"status": "success", "report": "The weather in Paris i
    else:
        return {"status": "error", "error_message": f"Weather informat

weather_tool = FunctionTool(func=get_weather_report)

# Tool 2
def analyze_sentiment(text: str) -> dict:
    """Analyzes the sentiment of the given text.

    Returns:
        dict: A dictionary with 'sentiment' ('positive', 'negative', o
    """
    if "good" in text.lower() or "sunny" in text.lower():
        return {"sentiment": "positive", "confidence": 0.8}
    elif "rain" in text.lower() or "bad" in text.lower():
        return {"sentiment": "negative", "confidence": 0.7}
    else:
        return {"sentiment": "neutral", "confidence": 0.6}
```

```
sentiment_tool = FunctionTool(func=analyze_sentiment)

# Agent
weather_sentiment_agent = Agent(
    model=MODEL_ID,
    name='weather_sentiment_agent',
    instruction="""You are a helpful assistant that provides weather i
**If the user asks about the weather in a specific city, use the 'get_
**If the 'get_weather_report' tool returns a 'success' status, provide
**If the 'get_weather_report' tool returns an 'error' status, inform t
**After providing a weather report, if the user gives feedback on the
You can handle these tasks sequentially if needed.""",
    tools=[weather_tool, sentiment_tool]
)

# Session and Runner
session_service = InMemorySessionService()
session = session_service.create_session(app_name=APP_NAME, user_id=US
runner = Runner(agent=weather_sentiment_agent, app_name=APP_NAME, sess

# Agent Interaction
def call_agent(query):
    content = types.Content(role='user', parts=[types.Part(text=query)
    events = runner.run(user_id=USER_ID, session_id=SESSION_ID, new_me

    for event in events:
        if event.is_final_response():
            final_response = event.content.parts[0].text
            print("Agent Response: ", final_response)

call_agent("weather in london?")
```

```
import com.google.adk.agents.BaseAgent;
import com.google.adk.agents.LlmAgent;
```

```
import com.google.adk.runner.Runner;
import com.google.adk.sessions.InMemorySessionService;
import com.google.adk.sessions.Session;
import com.google.adk.tools.Annotations.Schema;
import com.google.adk.tools.FunctionTool;
import com.google.adk.tools.ToolContext; // Ensure this import is corr
import com.google.common.collect.ImmutableList;
import com.google.genai.types.Content;
import com.google.genai.types.Part;
import java.util.HashMap;
import java.util.Locale;
import java.util.Map;

public class WeatherSentimentAgentApp {

  private static final String APP_NAME = "weather_sentiment_agent";
  private static final String USER_ID = "user1234";
  private static final String SESSION_ID = "1234";
  private static final String MODEL_ID = "gemini-2.0-flash";

  /**
   * Retrieves the current weather report for a specified city.
   *
   * @param city The city for which to retrieve the weather report.
   * @param toolContext The context for the tool.
   * @return A dictionary containing the weather information.
   */
  public static Map<String, Object> getWeatherReport(
      @Schema(name = "city")
      String city,
      @Schema(name = "toolContext")
      ToolContext toolContext) {
    Map<String, Object> response = new HashMap<>();

    if (city.toLowerCase(Locale.ROOT).equals("london")) {
      response.put("status", "success");
```

```
      response.put(
          "report",
          "The current weather in London is cloudy with a temperature
              + " chance of rain.");
    } else if (city.toLowerCase(Locale.ROOT).equals("paris")) {
      response.put("status", "success");
      response.put(
          "report", "The weather in Paris is sunny with a temperature
    } else {
      response.put("status", "error");
      response.put(
          "error_message", String.format("Weather information for '%s'
    }
    return response;
  }

  /**
   * Analyzes the sentiment of the given text.
   *
   * @param text The text to analyze.
   * @param toolContext The context for the tool.
   * @return A dictionary with sentiment and confidence score.
   */
  public static Map<String, Object> analyzeSentiment(
      @Schema(name = "text")
      String text,
      @Schema(name = "toolContext")
      ToolContext toolContext) {
    Map<String, Object> response = new HashMap<>();
    String lowerText = text.toLowerCase(Locale.ROOT);
    if (lowerText.contains("good") || lowerText.contains("sunny")) {
      response.put("sentiment", "positive");
      response.put("confidence", 0.8);
    } else if (lowerText.contains("rain") || lowerText.contains("bad")
      response.put("sentiment", "negative");
      response.put("confidence", 0.7);
```

```
    } else {
      response.put("sentiment", "neutral");
      response.put("confidence", 0.6);
    }
    return response;
  }

  /**
   * Calls the agent with the given query and prints the final respons
   *
   * @param runner The runner to use.
   * @param query The query to send to the agent.
   */
  public static void callAgent(Runner runner, String query) {
    Content content = Content.fromParts(Part.fromText(query));

    InMemorySessionService sessionService = (InMemorySessionService) r
    Session session =
        sessionService
            .createSession(APP_NAME, USER_ID, /* state= */ null, SESSI
            .blockingGet();

    runner
        .runAsync(session.userId(), session.id(), content)
        .forEach(
            event -> {
              if (event.finalResponse()
                  && event.content().isPresent()
                  && event.content().get().parts().isPresent()
                  && !event.content().get().parts().get().isEmpty()
                  && event.content().get().parts().get().get(0).text()
                String finalResponse = event.content().get().parts().g
                System.out.println("Agent Response: " + finalResponse)
              }
            });
  }
```

```
public static void main(String[] args) throws NoSuchMethodException
  FunctionTool weatherTool =
      FunctionTool.create(
          WeatherSentimentAgentApp.class.getMethod(
              "getWeatherReport", String.class, ToolContext.class));
  FunctionTool sentimentTool =
      FunctionTool.create(
          WeatherSentimentAgentApp.class.getMethod(
              "analyzeSentiment", String.class, ToolContext.class));

  BaseAgent weatherSentimentAgent =
      LlmAgent.builder()
          .model(MODEL_ID)
          .name("weather_sentiment_agent")
          .description("Weather Sentiment Agent")
          .instruction("""
                  You are a helpful assistant that provides weather
                  sentiment of user feedback
                  **If the user asks about the weather in a specific
                  'get_weather_report' tool to retrieve the weather
                  **If the 'get_weather_report' tool returns a 'succ
                  weather report to the user.**
                  **If the 'get_weather_report' tool returns an 'err
                  user that the weather information for the specifie
                  and ask if they have another city in mind.**
                  **After providing a weather report, if the user gi
                  weather (e.g., 'That's good' or 'I don't like rain
                  'analyze_sentiment' tool to understand their senti
                  acknowledge their sentiment.
                  You can handle these tasks sequentially if needed.
                  """)
          .tools(ImmutableList.of(weatherTool, sentimentTool))
          .build();

  InMemorySessionService sessionService = new InMemorySessionService
```

```
    Runner runner = new Runner(weatherSentimentAgent, APP_NAME, null,

    // Change the query to ensure the tool is called with a valid city
    // response from the tool, like "london" (without the question mar
    callAgent(runner, "weather in paris");
  }
}
```

Tool Context¶

For more advanced scenarios, ADK allows you to access additional contextual information within your tool function by including the special parameter `tool_context: ToolContext`. By including this in the function signature, ADK will **automatically** provide an **instance of the ToolContext** class when your tool is called during agent execution.

The **ToolContext** provides access to several key pieces of information and control levers:

- `state: State`: Read and modify the current session's state. Changes made here are tracked and persisted.
- `actions: EventActions`: Influence the agent's subsequent actions after the tool runs (e.g., skip summarization, transfer to another agent).
- `function_call_id: str`: The unique identifier assigned by the framework to this specific invocation of the tool. Useful for tracking and correlating with authentication responses. This can also be helpful when multiple tools are called within a single model response.
- `function_call_event_id: str`: This attribute provides the unique identifier of the **event** that triggered the current tool call. This can be useful for tracking and logging purposes.
- `auth_response: Any`: Contains the authentication response/ credentials if an authentication flow was completed before this tool call.
- Access to Services: Methods to interact with configured services like Artifacts and Memory.

Note that you shouldn't include the `tool_context` parameter in the tool function docstring. Since `ToolContext` is automatically injected by the ADK

framework *after* the LLM decides to call the tool function, it is not relevant for the LLM's decision-making and including it can confuse the LLM.

State Management¶

The `tool_context.state` attribute provides direct read and write access to the state associated with the current session. It behaves like a dictionary but ensures that any modifications are tracked as deltas and persisted by the session service. This enables tools to maintain and share information across different interactions and agent steps.

- **Reading State**: Use standard dictionary access (`tool_context.state['my_key']`) or the `.get()` method (`tool_context.state.get('my_key', default_value)`).
- **Writing State**: Assign values directly (`tool_context.state['new_key'] = 'new_value'`). These changes are recorded in the state_delta of the resulting event.
- **State Prefixes**: Remember the standard state prefixes:
- `app:*`: Shared across all users of the application.
- `user:*`: Specific to the current user across all their sessions.
- (No prefix): Specific to the current session.
- `temp:*`: Temporary, not persisted across invocations (useful for passing data within a single run call but generally less useful inside a tool context which operates between LLM calls).

PythonJava

```
from google.adk.tools import ToolContext, FunctionTool

def update_user_preference(preference: str, value: str, tool_context:
    """Updates a user-specific preference."""
    user_prefs_key = "user:preferences"
    # Get current preferences or initialize if none exist
    preferences = tool_context.state.get(user_prefs_key, {})
    preferences[preference] = value
    # Write the updated dictionary back to the state
```

```
    tool_context.state[user_prefs_key] = preferences
    print(f"Tool: Updated user preference '{preference}' to '{value}'"
    return {"status": "success", "updated_preference": preference}

pref_tool = FunctionTool(func=update_user_preference)

# In an Agent:
# my_agent = Agent(..., tools=[pref_tool])

# When the LLM calls update_user_preference(preference='theme', value=
# The tool_context.state will be updated, and the change will be part
# resulting tool response event's actions.state_delta.
```

```
import com.google.adk.tools.FunctionTool;
import com.google.adk.tools.ToolContext;

// Updates a user-specific preference.
public Map<String, String> updateUserThemePreference(String value, Too
  String userPrefsKey = "user:preferences:theme";

  // Get current preferences or initialize if none exist
  String preference = toolContext.state().getOrDefault(userPrefsKey, "
  if (preference.isEmpty()) {
    preference = value;
  }

  // Write the updated dictionary back to the state
  toolContext.state().put("user:preferences", preference);
  System.out.printf("Tool: Updated user preference %s to %s", userPref

  return Map.of("status", "success", "updated_preference", toolContext
  // When the LLM calls updateUserThemePreference("dark"):
  // The toolContext.state will be updated, and the change will be par
  // resulting tool response event's actions.stateDelta.
```

```
}
```

Controlling Agent Flow¶

The `tool_context.actions` attribute (`ToolContext.actions()` in Java) holds an **EventActions** object. Modifying attributes on this object allows your tool to influence what the agent or framework does after the tool finishes execution.

- **`skip_summarization: bool`**: (Default: False) If set to True, instructs the ADK to bypass the LLM call that typically summarizes the tool's output. This is useful if your tool's return value is already a user-ready message.
- **`transfer_to_agent: str`**: Set this to the name of another agent. The framework will halt the current agent's execution and **transfer control of the conversation to the specified agent**. This allows tools to dynamically hand off tasks to more specialized agents.
- **`escalate: bool`**: (Default: False) Setting this to True signals that the current agent cannot handle the request and should pass control up to its parent agent (if in a hierarchy). In a LoopAgent, setting **escalate=True** in a sub-agent's tool will terminate the loop.

Example¶

PythonJava

```
 from google.adk.agents import Agent
from google.adk.tools import FunctionTool
from google.adk.runners import Runner
from google.adk.sessions import InMemorySessionService
from google.adk.tools import ToolContext
from google.genai import types

APP_NAME="customer_support_agent"
USER_ID="user1234"
SESSION_ID="1234"
```

```
def check_and_transfer(query: str, tool_context: ToolContext) -> str:
    """Checks if the query requires escalation and transfers to anothe
    if "urgent" in query.lower():
        print("Tool: Detected urgency, transferring to the support age
        tool_context.actions.transfer_to_agent = "support_agent"
        return "Transferring to the support agent..."
    else:
        return f"Processed query: '{query}'. No further action needed.

escalation_tool = FunctionTool(func=check_and_transfer)

main_agent = Agent(
    model='gemini-2.0-flash',
    name='main_agent',
    instruction="""You are the first point of contact for customer sup
    tools=[check_and_transfer]
)

support_agent = Agent(
    model='gemini-2.0-flash',
    name='support_agent',
    instruction="""You are the dedicated support agent. Mentioned you
)

main_agent.sub_agents = [support_agent]

# Session and Runner
session_service = InMemorySessionService()
session = session_service.create_session(app_name=APP_NAME, user_id=US
runner = Runner(agent=main_agent, app_name=APP_NAME, session_service=s

# Agent Interaction
def call_agent(query):
    content = types.Content(role='user', parts=[types.Part(text=query)
    events = runner.run(user_id=USER_ID, session_id=SESSION_ID, new_me
```

```
    for event in events:
        if event.is_final_response():
            final_response = event.content.parts[0].text
            print("Agent Response: ", final_response)

call_agent("this is urgent, i cant login")
```

```
import com.google.adk.agents.LlmAgent;
import com.google.adk.runner.Runner;
import com.google.adk.sessions.InMemorySessionService;
import com.google.adk.sessions.Session;
import com.google.adk.tools.Annotations.Schema;
import com.google.adk.tools.FunctionTool;
import com.google.adk.tools.ToolContext;
import com.google.common.collect.ImmutableList;
import com.google.genai.types.Content;
import com.google.genai.types.Part;
import java.util.HashMap;
import java.util.Locale;
import java.util.Map;

public class CustomerSupportAgentApp {

  private static final String APP_NAME = "customer_support_agent";
  private static final String USER_ID = "user1234";
  private static final String SESSION_ID = "1234";
  private static final String MODEL_ID = "gemini-2.0-flash";

  /**
   * Checks if the query requires escalation and transfers to another
   *
   * @param query The user's query.
   * @param toolContext The context for the tool.
   * @return A map indicating the result of the check and transfer.
```

```
 */
public static Map<String, Object> checkAndTransfer(
    @Schema(name = "query", description = "the user query")
    String query,
    @Schema(name = "toolContext", description = "the tool context")
    ToolContext toolContext) {
  Map<String, Object> response = new HashMap<>();
  if (query.toLowerCase(Locale.ROOT).contains("urgent")) {
    System.out.println("Tool: Detected urgency, transferring to the
    toolContext.actions().setTransferToAgent("support_agent");
    response.put("status", "transferring");
    response.put("message", "Transferring to the support agent...");
  } else {
    response.put("status", "processed");
    response.put(
        "message", String.format("Processed query: '%s'. No further
  }
  return response;
}

/**
 * Calls the agent with the given query and prints the final respons
 *
 * @param runner The runner to use.
 * @param query The query to send to the agent.
 */
public static void callAgent(Runner runner, String query) {
  Content content =
      Content.fromParts(Part.fromText(query));

  InMemorySessionService sessionService = (InMemorySessionService) r
  // Fixed: session ID does not need to be an optional.
  Session session =
      sessionService
          .createSession(APP_NAME, USER_ID, /* state= */ null, SESSI
          .blockingGet();
```

```
  runner
      .runAsync(session.userId(), session.id(), content)
      .forEach(
          event -> {
            if (event.finalResponse()
                && event.content().isPresent()
                && event.content().get().parts().isPresent()
                && !event.content().get().parts().get().isEmpty()
                && event.content().get().parts().get().get(0).text()
              String finalResponse = event.content().get().parts().g
              System.out.println("Agent Response: " + finalResponse)
            }
          });
}

public static void main(String[] args) throws NoSuchMethodException
  FunctionTool escalationTool =
      FunctionTool.create(
          CustomerSupportAgentApp.class.getMethod(
              "checkAndTransfer", String.class, ToolContext.class));

  LlmAgent supportAgent =
      LlmAgent.builder()
          .model(MODEL_ID)
          .name("support_agent")
          .description("""
              The dedicated support agent.
              Mentions it is a support handler and helps the user wi
          """)
          .instruction("""
              You are the dedicated support agent.
              Mentioned you are a support handler and please help th
          """)
          .build();
```

```
    LlmAgent mainAgent =
        LlmAgent.builder()
            .model(MODEL_ID)
            .name("main_agent")
            .description("""
                The first point of contact for customer support of an
                Answers general queries.
                If the user indicates urgency, uses the 'check_and_tra
                """)
            .instruction("""
                You are the first point of contact for customer suppor
                Answer general queries.
                If the user indicates urgency, use the 'check_and_tran
                """)
            .tools(ImmutableList.of(escalationTool))
            .subAgents(supportAgent)
            .build();
    // Fixed: LlmAgent.subAgents() expects 0 arguments.
    // Sub-agents are now added to the main agent via its builder,
    // as `subAgents` is a property that should be set during agent co
    // if it's not dynamically managed.

    InMemorySessionService sessionService = new InMemorySessionService
    Runner runner = new Runner(mainAgent, APP_NAME, null, sessionServi

    // Agent Interaction
    callAgent(runner, "this is urgent, i cant login");
  }
}
```

Explanation¶

- We define two agents: `main_agent` and `support_agent`. The `main_agent` is designed to be the initial point of contact.

- The `check_and_transfer` tool, when called by `main_agent`, examines the user's query.
- If the query contains the word "urgent", the tool accesses the `tool_context`, specifically **`tool_context.actions`**, and sets the transfer_to_agent attribute to `support_agent`.
- This action signals to the framework to **transfer the control of the conversation to the agent named `support_agent`**.
- When the `main_agent` processes the urgent query, the `check_and_transfer` tool triggers the transfer. The subsequent response would ideally come from the `support_agent`.
- For a normal query without urgency, the tool simply processes it without triggering a transfer.

This example illustrates how a tool, through EventActions in its ToolContext, can dynamically influence the flow of the conversation by transferring control to another specialized agent.

Authentication¶

Currently supported in Python

ToolContext provides mechanisms for tools interacting with authenticated APIs. If your tool needs to handle authentication, you might use the following:

- **`auth_response`**: Contains credentials (e.g., a token) if authentication was already handled by the framework before your tool was called (common with RestApiTool and OpenAPI security schemes).
- **`request_credential(auth_config: dict)`**: Call this method if your tool determines authentication is needed but credentials aren't available. This signals the framework to start an authentication flow based on the provided auth_config.
- **`get_auth_response()`**: Call this in a subsequent invocation (after request_credential was successfully handled) to retrieve the credentials the user provided.

For detailed explanations of authentication flows, configuration, and examples, please refer to the dedicated Tool Authentication documentation page.

Context-Aware Data Access Methods¶

These methods provide convenient ways for your tool to interact with persistent data associated with the session or user, managed by configured services.

- **`list_artifacts()`** (or **`listArtifacts()`** in Java): Returns a list of filenames (or keys) for all artifacts currently stored for the session via the artifact_service. Artifacts are typically files (images, documents, etc.) uploaded by the user or generated by tools/agents.
- **`load_artifact(filename: str)`**: Retrieves a specific artifact by its filename from the **artifact_service**. You can optionally specify a version; if omitted, the latest version is returned. Returns a `google.genai.types.Part` object containing the artifact data and mime type, or None if not found.
- **`save_artifact(filename: str, artifact: types.Part)`**: Saves a new version of an artifact to the artifact_service. Returns the new version number (starting from 0).
- **`search_memory(query: str)`** Currently supported in Python

Queries the user's long-term memory using the configured `memory_service`. This is useful for retrieving relevant information from past interactions or stored knowledge. The structure of the **SearchMemoryResponse** depends on the specific memory service implementation but typically contains relevant text snippets or conversation excerpts.

Example¶

PythonJava

```
# Copyright 2025 Google LLC
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
```

```
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or impl
# See the License for the specific language governing permissions and
# limitations under the License.

from google.adk.tools import ToolContext, FunctionTool
from google.genai import types

def process_document(
    document_name: str, analysis_query: str, tool_context: ToolContext
) -> dict:
    """Analyzes a document using context from memory."""

    # 1. Load the artifact
    print(f"Tool: Attempting to load artifact: {document_name}")
    document_part = tool_context.load_artifact(document_name)

    if not document_part:
        return {"status": "error", "message": f"Document '{document_na

    document_text = document_part.text  # Assuming it's text for simpl
    print(f"Tool: Loaded document '{document_name}' ({len(document_tex

    # 2. Search memory for related context
    print(f"Tool: Searching memory for context related to: '{analysis_
    memory_response = tool_context.search_memory(
        f"Context for analyzing document about {analysis_query}"
    )
    memory_context = "\n".join(
        [
            m.events[0].content.parts[0].text
            for m in memory_response.memories
            if m.events and m.events[0].content
        ]
    )  # Simplified extraction
    print(f"Tool: Found memory context: {memory_context[:100]}...")
```

```
    # 3. Perform analysis (placeholder)
    analysis_result = f"Analysis of '{document_name}' regarding '{anal
    print("Tool: Performed analysis.")

    # 4. Save the analysis result as a new artifact
    analysis_part = types.Part.from_text(text=analysis_result)
    new_artifact_name = f"analysis_{document_name}"
    version = await tool_context.save_artifact(new_artifact_name, anal
    print(f"Tool: Saved analysis result as '{new_artifact_name}' versi

    return {
        "status": "success",
        "analysis_artifact": new_artifact_name,
        "version": version,
    }

doc_analysis_tool = FunctionTool(func=process_document)

# In an Agent:
# Assume artifact 'report.txt' was previously saved.
# Assume memory service is configured and has relevant past data.
# my_agent = Agent(..., tools=[doc_analysis_tool], artifact_service=..
```

```
// Analyzes a document using context from memory.
// You can also list, load and save artifacts using Callback Context o
public static @NonNull Maybe<ImmutableMap<String, Object>> processDocu
    @Annotations.Schema(description = "The name of the document to ana
    @Annotations.Schema(description = "The query for the analysis.") S
    ToolContext toolContext) {

  // 1. List all available artifacts
  System.out.printf(
      "Listing all available artifacts %s:", toolContext.listArtifacts
```

```
  // 2. Load an artifact to memory
  System.out.println("Tool: Attempting to load artifact: " + documentN
  Part documentPart = toolContext.loadArtifact(documentName, Optional.
  if (documentPart == null) {
    System.out.println("Tool: Document '" + documentName + "' not foun
    return Maybe.just(
        ImmutableMap.<String, Object>of(
            "status", "error", "message", "Document '" + documentName
  }
  String documentText = documentPart.text().orElse("");
  System.out.println(
      "Tool: Loaded document '" + documentName + "' (" + documentText.

  // 3. Perform analysis (placeholder)
  String analysisResult =
      "Analysis of '"
          + documentName
          + "' regarding '"
          + analysisQuery
          + " [Placeholder Analysis Result]";
  System.out.println("Tool: Performed analysis.");

  // 4. Save the analysis result as a new artifact
  Part analysisPart = Part.fromText(analysisResult);
  String newArtifactName = "analysis_" + documentName;

  toolContext.saveArtifact(newArtifactName, analysisPart);

  return Maybe.just(
      ImmutableMap.<String, Object>builder()
          .put("status", "success")
          .put("analysis_artifact", newArtifactName)
          .build());
}
// FunctionTool processDocumentTool =
```

```
//      FunctionTool.create(ToolContextArtifactExample.class, "process
// In the Agent, include this function tool.
// LlmAgent agent = LlmAgent().builder().tools(processDocumentTool).bu
```

By leveraging the **ToolContext**, developers can create more sophisticated and context-aware custom tools that seamlessly integrate with ADK's architecture and enhance the overall capabilities of their agents.

Defining Effective Tool Functions¶

When using a method or function as an ADK Tool, how you define it significantly impacts the agent's ability to use it correctly. The agent's Large Language Model (LLM) relies heavily on the function's **name**, **parameters (arguments)**, **type hints**, and **docstring / source code comments** to understand its purpose and generate the correct call.

Here are key guidelines for defining effective tool functions:

- **Function Name:**
- Use descriptive, verb-noun based names that clearly indicate the action (e.g., `get_weather`, `searchDocuments`, `schedule_meeting`).
- Avoid generic names like `run`, `process`, `handle_data`, or overly ambiguous names like `doStuff`. Even with a good description, a name like `do_stuff` might confuse the model about when to use the tool versus, for example, `cancelFlight`.
- The LLM uses the function name as a primary identifier during tool selection.
- **Parameters (Arguments):**
- Your function can have any number of parameters.
- Use clear and descriptive names (e.g., `city` instead of `c`, `search_query` instead of `q`).
- **Provide type hints in Python** for all parameters (e.g., `city: str`, `user_id: int`, `items: list[str]`). This is essential for ADK to generate the correct schema for the LLM.

- Ensure all parameter types are **JSON serializable**. All java primitives as well as standard Python types like `str`, `int`, `float`, `bool`, `list`, `dict`, and their combinations are generally safe. Avoid complex custom class instances as direct parameters unless they have a clear JSON representation.
- **Do not set default values** for parameters. E.g., `def my_func(param1: str = "default")`. Default values are not reliably supported or used by the underlying models during function call generation. All necessary information should be derived by the LLM from the context or explicitly requested if missing.
- **`self` / `cls` Handled Automatically:** Implicit parameters like `self` (for instance methods) or `cls` (for class methods) are automatically handled by ADK and excluded from the schema shown to the LLM. You only need to define type hints and descriptions for the logical parameters your tool requires the LLM to provide.

- **Return Type:**

- The function's return value **must be a dictionary (`dict`)** in Python or a **Map** in Java.

- If your function returns a non-dictionary type (e.g., a string, number, list), the ADK framework will automatically wrap it into a dictionary/Map like `{'result': your_original_return_value}` before passing the result back to the model.
- Design the dictionary/Map keys and values to be **descriptive and easily understood *by the LLM***. Remember, the model reads this output to decide its next step.
- Include meaningful keys. For example, instead of returning just an error code like `500`, return `{'status': 'error', 'error_message': 'Database connection failed'}`.
- It's a **highly recommended practice** to include a `status` key (e.g., `'success'`, `'error'`, `'pending'`, `'ambiguous'`) to clearly indicate the outcome of the tool execution for the model.

- **Docstring / Source Code Comments:**

- **This is critical.** The docstring is the primary source of descriptive information for the LLM.

- **Clearly state what the tool *does*.** Be specific about its purpose and limitations.
- **Explain *when* the tool should be used.** Provide context or example scenarios to guide the LLM's decision-making.
- **Describe *each parameter* clearly.** Explain what information the LLM needs to provide for that argument.
- Describe the **structure and meaning of the expected `dict` return value**, especially the different `status` values and associated data keys.
- **Do not describe the injected ToolContext parameter**. Avoid mentioning the optional `tool_context: ToolContext` parameter within the docstring description since it is not a parameter the LLM needs to know about. ToolContext is injected by ADK, *after* the LLM decides to call it.

Example of a good definition:

PythonJava

```
def lookup_order_status(order_id: str) -> dict:
  """Fetches the current status of a customer's order using its ID.

  Use this tool ONLY when a user explicitly asks for the status of
  a specific order and provides the order ID. Do not use it for
  general inquiries.

  Args:
      order_id: The unique identifier of the order to look up.

  Returns:
      A dictionary containing the order status.
      Possible statuses: 'shipped', 'processing', 'pending', 'error'.
      Example success: {'status': 'shipped', 'tracking_number': '1Z9..
      Example error: {'status': 'error', 'error_message': 'Order ID no
  """
  # ... function implementation to fetch status ...
  if status := fetch_status_from_backend(order_id):
       return {"status": status.state, "tracking_number": status.track
  else:
```

```
        return {"status": "error", "error_message": f"Order ID {order_i
```

```
/**
 * Retrieves the current weather report for a specified city.
 *
 * @param city The city for which to retrieve the weather report.
 * @param toolContext The context for the tool.
 * @return A dictionary containing the weather information.
 */
public static Map<String, Object> getWeatherReport(String city, ToolCo
    Map<String, Object> response = new HashMap<>();
    if (city.toLowerCase(Locale.ROOT).equals("london")) {
        response.put("status", "success");
        response.put(
                "report",
                "The current weather in London is cloudy with a temper
                        + " chance of rain.");
    } else if (city.toLowerCase(Locale.ROOT).equals("paris")) {
        response.put("status", "success");
        response.put("report", "The weather in Paris is sunny with a t
    } else {
        response.put("status", "error");
        response.put("error_message", String.format("Weather informati
    }
    return response;
}
```

- **Simplicity and Focus:**
- **Keep Tools Focused:** Each tool should ideally perform one well-defined task.
- **Fewer Parameters are Better:** Models generally handle tools with fewer, clearly defined parameters more reliably than those with many optional or complex ones.

- **Use Simple Data Types:** Prefer basic types (`str`, `int`, `bool`, `float`, `List[str]`, in **Python**, or `int`, `byte`, `short`, `long`, `float`, `double`, `boolean` and `char` in **Java**) over complex custom classes or deeply nested structures as parameters when possible.
- **Decompose Complex Tasks:** Break down functions that perform multiple distinct logical steps into smaller, more focused tools. For instance, instead of a single `update_user_profile(profile: ProfileObject)` tool, consider separate tools like `update_user_name(name: str)`, `update_user_address(address: str)`, `update_user_preferences(preferences: list[str])`, etc. This makes it easier for the LLM to select and use the correct capability.

By adhering to these guidelines, you provide the LLM with the clarity and structure it needs to effectively utilize your custom function tools, leading to more capable and reliable agent behavior.

Toolsets: Grouping and Dynamically Providing Tools python_only¶

Beyond individual tools, ADK introduces the concept of a **Toolset** via the `BaseToolset` interface (defined in `google.adk.tools.base_toolset`). A toolset allows you to manage and provide a collection of `BaseTool` instances, often dynamically, to an agent.

This approach is beneficial for:

- **Organizing Related Tools:** Grouping tools that serve a common purpose (e.g., all tools for mathematical operations, or all tools interacting with a specific API).
- **Dynamic Tool Availability:** Enabling an agent to have different tools available based on the current context (e.g., user permissions, session state, or other runtime conditions). The `get_tools` method of a toolset can decide which tools to expose.
- **Integrating External Tool Providers:** Toolsets can act as adapters for tools coming from external systems, like an OpenAPI specification or an MCP server, converting them into ADK-compatible `BaseTool` objects.

The `BaseToolset` Interface¶

Any class acting as a toolset in ADK should implement the `BaseToolset` abstract base class. This interface primarily defines two methods:

- **`async def get_tools(...) -> list[BaseTool]:`** This is the core method of a toolset. When an ADK agent needs to know its available tools, it will call `get_tools()` on each `BaseToolset` instance provided in its `tools` list.
- It receives an optional `readonly_context` (an instance of `ReadonlyContext`). This context provides read-only access to information like the current session state (`readonly_context.state`), agent name, and invocation ID. The toolset can use this context to dynamically decide which tools to return.
- It **must** return a `list` of `BaseTool` instances (e.g., `FunctionTool`, `RestApiTool`).
- **`async def close(self) -> None:`** This asynchronous method is called by the ADK framework when the toolset is no longer needed, for example, when an agent server is shutting down or the `Runner` is being closed. Implement this method to perform any necessary cleanup, such as closing network connections, releasing file handles, or cleaning up other resources managed by the toolset.

Using Toolsets with Agents¶

You can include instances of your `BaseToolset` implementations directly in an `LlmAgent`'s `tools` list, alongside individual `BaseTool` instances.

When the agent initializes or needs to determine its available capabilities, the ADK framework will iterate through the `tools` list:

- If an item is a `BaseTool` instance, it's used directly.
- If an item is a `BaseToolset` instance, its `get_tools()` method is called (with the current `ReadonlyContext`), and the returned list of `BaseTool`s is added to the agent's available tools.

Example: A Simple Math Toolset¶

Let's create a basic example of a toolset that provides simple arithmetic operations.

```
 # 1. Define the individual tool functions
def add_numbers(a: int, b: int, tool_context: ToolContext) -> Dict[str
    """Adds two integer numbers.
    Args:
        a: The first number.
        b: The second number.
    Returns:
        A dictionary with the sum, e.g., {'status': 'success', 'result
    """
    print(f"Tool: add_numbers called with a={a}, b={b}")
    result = a + b
    # Example: Storing something in tool_context state
    tool_context.state["last_math_operation"] = "addition"
    return {"status": "success", "result": result}

def subtract_numbers(a: int, b: int) -> Dict[str, Any]:
    """Subtracts the second number from the first.
    Args:
        a: The first number.
        b: The second number.
    Returns:
        A dictionary with the difference, e.g., {'status': 'success',
    """
    print(f"Tool: subtract_numbers called with a={a}, b={b}")
    return {"status": "success", "result": a - b}

# 2. Create the Toolset by implementing BaseToolset
class SimpleMathToolset(BaseToolset):
    def __init__(self, prefix: str = "math_"):
        self.prefix = prefix
        # Create FunctionTool instances once
```

```
        self._add_tool = FunctionTool(
            func=add_numbers,
            name=f"{self.prefix}add_numbers",  # Toolset can customize
        )
        self._subtract_tool = FunctionTool(
            func=subtract_numbers, name=f"{self.prefix}subtract_number
        )
        print(f"SimpleMathToolset initialized with prefix '{self.prefi

    async def get_tools(
        self, readonly_context: Optional[ReadonlyContext] = None
    ) -> List[BaseTool]:
        print(f"SimpleMathToolset.get_tools() called.")
        # Example of dynamic behavior:
        # Could use readonly_context.state to decide which tools to re
        # For instance, if readonly_context.state.get("enable_advanced
        #    return [self._add_tool, self._subtract_tool, self._multip

        # For this simple example, always return both tools
        tools_to_return = [self._add_tool, self._subtract_tool]
        print(f"SimpleMathToolset providing tools: {[t.name for t in t
        return tools_to_return

    async def close(self) -> None:
        # No resources to clean up in this simple example
        print(f"SimpleMathToolset.close() called for prefix '{self.pre
        await asyncio.sleep(0)  # Placeholder for async cleanup if nee

# 3. Define an individual tool (not part of the toolset)
def greet_user(name: str = "User") -> Dict[str, str]:
    """Greets the user."""
    print(f"Tool: greet_user called with name={name}")
    return {"greeting": f"Hello, {name}!"}

greet_tool = FunctionTool(func=greet_user)
```

```
# 4. Instantiate the toolset
math_toolset_instance = SimpleMathToolset(prefix="calculator_")

# 5. Define an agent that uses both the individual tool and the toolse
calculator_agent = LlmAgent(
    name="CalculatorAgent",
    model="gemini-2.0-flash",  # Replace with your desired model
    instruction="You are a helpful calculator and greeter. "
    "Use 'greet_user' for greetings. "
    "Use 'calculator_add_numbers' to add and 'calculator_subtract_numb
    "Announce the state of 'last_math_operation' if it's set.",
    tools=[greet_tool, math_toolset_instance],  # Individual tool  # T
)
```

In this example:

- `SimpleMathToolset` implements `BaseToolset` and its `get_tools()` method returns `FunctionTool` instances for `add_numbers` and `subtract_numbers`. It also customizes their names using a prefix.
- The `calculator_agent` is configured with both an individual `greet_tool` and an instance of `SimpleMathToolset`.
- When `calculator_agent` is run, ADK will call `math_toolset_instance.get_tools()`. The agent's LLM will then have access to `greet_user`, `calculator_add_numbers`, and `calculator_subtract_numbers` to handle user requests.
- The `add_numbers` tool demonstrates writing to `tool_context.state`, and the agent's instruction mentions reading this state.
- The `close()` method is called to ensure any resources held by the toolset are released.

Toolsets offer a powerful way to organize, manage, and dynamically provide collections of tools to your ADK agents, leading to more modular, maintainable, and adaptable agentic applications.