Function tools

What are function tools?

When out-of-the-box tools don't fully meet specific requirements, developers can create custom function tools. This allows for **tailored functionality**, such as connecting to proprietary databases or implementing unique algorithms.

For example, a function tool, "myfinancetool", might be a function that calculates a specific financial metric. ADK also supports long running functions, so if that calculation takes a while, the agent can continue working on other tasks.

ADK offers several ways to create functions tools, each suited to different levels of complexity and control:

1. Function Tool
2. Long Running Function Tool
3. Agents-as-a-Tool

1. Function Tool

Transforming a function into a tool is a straightforward way to integrate custom logic into your agents. In fact, when you assign a function to an agent's tools list, the framework will automatically wrap it as a Function Tool for you. This approach offers flexibility and quick integration.

Parameters

Define your function parameters using standard **JSON-serializable types** (e.g., string, integer, list, dictionary). It's important to avoid setting default values for parameters, as the language model (LLM) does not currently support interpreting them.

Return Type

The preferred return type for a Function Tool is a **dictionary** in Python or **Map** in Java. This allows you to structure the response with key-value pairs, providing context and clarity to the LLM. If your function returns a type other than a dictionary, the framework automatically wraps it into a dictionary with a single key named **"result"**.

Strive to make your return values as descriptive as possible. *For example,* instead of returning a numeric error code, return a dictionary with an "error_message" key containing a human-readable explanation. **Remember that the LLM**, not a piece of code, needs to understand the result. As a best practice, include a "status" key in your return dictionary to indicate the overall outcome (e.g., "success", "error", "pending"), providing the LLM with a clear signal about the operation's state.

Docstring / Source code comments

The docstring (or comments above) your function serve as the tool's description and is sent to the LLM. Therefore, a well-written and comprehensive docstring is crucial for the LLM to understand how to use the tool effectively. Clearly explain the purpose of the function, the meaning of its parameters, and the expected return values.

Example

Python

This tool is a python function which obtains the Stock price of a given Stock ticker/ symbol.

Note: You need to `pip install yfinance` library before using this tool.

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

import yfinance as yf


APP_NAME = "stock_app"
USER_ID = "1234"
SESSION_ID = "session1234"

def get_stock_price(symbol: str):
    """
    Retrieves the current stock price for a given symbol.

    Args:
        symbol (str): The stock symbol (e.g., "AAPL", "GOOG").

    Returns:
        float: The current stock price, or None if an error occurs.
    """
    try:
        stock = yf.Ticker(symbol)
        historical_data = stock.history(period="1d")
        if not historical_data.empty:
            current_price = historical_data['Close'].iloc[-1]
            return current_price
        else:
            return None
    except Exception as e:
        print(f"Error retrieving stock price for {symbol}: {e}")
        return None


stock_price_agent = Agent(
    model='gemini-2.0-flash',
    name='stock_agent',
    instruction= 'You are an agent who retrieves stock prices. If a
ticker symbol is provided, fetch the current price. If only a company
name is given, first perform a Google search to find the correct ticker
symbol before retrieving the stock price. If the provided ticker symbol
is invalid or data cannot be retrieved, inform the user that the stock
price could not be found.',
    description='This agent specializes in retrieving real-time stock
prices. Given a stock ticker symbol (e.g., AAPL, GOOG, MSFT) or the
stock name, use the tools and reliable data sources to provide the most
up-to-date price.',
    tools=[get_stock_price], # You can add Python functions directly to
the tools list; they will be automatically wrapped as FunctionTools.
)


# Session and Runner
session_service = InMemorySessionService()
session = session_service.create_session(app_name=APP_NAME,
user_id=USER_ID, session_id=SESSION_ID)
runner = Runner(agent=stock_price_agent, app_name=APP_NAME,
session_service=session_service)


# 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_message=content)

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

call_agent("stock price of GOOG")
```

The return value from this tool will be wrapped into a dictionary.

```
{"result": "$123"}
```

Java

This tool retrieves the mocked value of a stock price.

```
import com.google.adk.agents.LlmAgent;
import com.google.adk.events.Event;
import com.google.adk.runner.InMemoryRunner;
import com.google.adk.sessions.Session;
import com.google.adk.tools.Annotations.Schema;
import com.google.adk.tools.FunctionTool;
import com.google.genai.types.Content;
import com.google.genai.types.Part;
import io.reactivex.rxjava3.core.Flowable;
import java.util.HashMap;
import java.util.Map;

public class StockPriceAgent {

  private static final String APP_NAME = "stock_agent";
  private static final String USER_ID = "user1234";

  // Mock data for various stocks functionality
  // NOTE: This is a MOCK implementation. In a real Java application,
  // you would use a financial data API or library.
  private static final Map<String, Double> mockStockPrices = new
HashMap<>();
```

```
  static {
    mockStockPrices.put("GOOG", 1.0);
    mockStockPrices.put("AAPL", 1.0);
    mockStockPrices.put("MSFT", 1.0);
  }

  @Schema(description = "Retrieves the current stock price for a given
symbol.")
  public static Map<String, Object> getStockPrice(
      @Schema(description = "The stock symbol (e.g., \"AAPL\",
\"GOOG\")",
        name = "symbol")
      String symbol) {

    try {
      if (mockStockPrices.containsKey(symbol.toUpperCase())) {
        double currentPrice =
mockStockPrices.get(symbol.toUpperCase());
        System.out.println("Tool: Found price for " + symbol + ": " +
currentPrice);
        return Map.of("symbol", symbol, "price", currentPrice);
      } else {
        return Map.of("symbol", symbol, "error", "No data found for
symbol");
      }
    } catch (Exception e) {
      return Map.of("symbol", symbol, "error", e.getMessage());
    }
  }

  public static void callAgent(String prompt) {
    // Create the FunctionTool from the Java method
    FunctionTool getStockPriceTool =
FunctionTool.create(StockPriceAgent.class, "getStockPrice");

    LlmAgent stockPriceAgent =
        LlmAgent.builder()
            .model("gemini-2.0-flash")
            .name("stock_agent")
            .instruction(
                "You are an agent who retrieves stock prices. If a
ticker symbol is provided, fetch the current price. If only a company
name is given, first perform a Google search to find the correct ticker
symbol before retrieving the stock price. If the provided ticker symbol
is invalid or data cannot be retrieved, inform the user that the stock
price could not be found.")
            .description(
                "This agent specializes in retrieving real-time stock
prices. Given a stock ticker symbol (e.g., AAPL, GOOG, MSFT) or the
stock name, use the tools and reliable data sources to provide the most
up-to-date price.")
            .tools(getStockPriceTool) // Add the Java FunctionTool
            .build();

    // Create an InMemoryRunner
    InMemoryRunner runner = new InMemoryRunner(stockPriceAgent,
APP_NAME);
    // InMemoryRunner automatically creates a session service. Create a
session using the service
    Session session = runner.sessionService().createSession(APP_NAME,
USER_ID).blockingGet();
    Content userMessage = Content.fromParts(Part.fromText(prompt));
    // Run the agent
    Flowable<Event> eventStream = runner.runAsync(USER_ID,
session.id(), userMessage);

    // Stream event response
    eventStream.blockingForEach(
        event -> {
          if (event.finalResponse()) {
            System.out.println(event.stringifyContent());
          }
        });
  }

  public static void main(String[] args) {
    callAgent("stock price of GOOG");
    callAgent("What's the price of MSFT?");
    callAgent("Can you find the stock price for an unknown company
XYZ?");
  }
}
```

The return value from this tool will be wrapped into a Map.

```
For input `GOOG`: {"symbol": "GOOG", "price": "1.0"}
```

Best Practices

While you have considerable flexibility in defining your function, remember that simplicity enhances usability for the LLM. Consider these guidelines:

- **Fewer Parameters are Better:** Minimize the number of parameters to reduce complexity.
- **Simple Data Types:** Favor primitive data types like `str` and `int` over custom classes whenever possible.
- **Meaningful Names:** The function's name and parameter names significantly influence how the LLM interprets and utilizes the tool. Choose names that clearly reflect the function's purpose and the meaning of its inputs. Avoid generic names like `do_stuff()` or `beAgent()`.

2. Long Running Function Tool

Designed for tasks that require a significant amount of processing time without blocking the agent's execution. This tool is a subclass of `FunctionTool`.

When using a `LongRunningFunctionTool`, your function can initiate the long-running operation and optionally return an **initial result**** (e.g. the long-running operation id). Once a long running function tool is invoked the agent runner will pause the agent run and let the agent client to decide whether to continue or wait until the long-running operation finishes. The agent client can query the progress of the long-running operation and send back an intermediate or final response. The agent can then continue with other tasks. An example is the human-in-the-loop scenario where the agent needs human approval before proceeding with a task.

How it Works

In Python, you wrap a function with `LongRunningFunctionTool`. In Java, you pass a Method name to `LongRunningFunctionTool.create()`.

1. **Initiation:** When the LLM calls the tool, your function starts the long-running operation.
2. **Initial Updates:** Your function should optionally return an initial result (e.g. the long-running operaiton id). The ADK framework takes the result and sends it back to the LLM packaged within a `FunctionResponse`. This allows the LLM to inform the user (e.g., status, percentage complete, messages). And then the agent run is ended / paused.
3. **Continue or Wait:** After each agent run is completed. Agent client can query the progress of the long-running operation and decide whether to continue the agent run with an intermediate response (to update the progress) or wait until a final response is retrieved. Agent client should send the intermediate or final response back to the agent for the next run.
4. **Framework Handling:** The ADK framework manages the execution. It sends the intermediate or final `FunctionResponse` sent by agent client to the LLM to generate a user friendly message.

Creating the Tool

Define your tool function and wrap it using the `LongRunningFunctionTool` class:

Python

```python
# 1. Define the long running function
def ask_for_approval(
    purpose: str, amount: float
) -> dict[str, Any]:
    """Ask for approval for the reimbursement."""
    # create a ticket for the approval
    # Send a notification to the approver with the link of the ticket
    return {'status': 'pending', 'approver': 'Sean Zhou', 'purpose' :
purpose, 'amount': amount, 'ticket-id': 'approval-ticket-1'}

def reimburse(purpose: str, amount: float) -> str:
    """Reimburse the amount of money to the employee."""
    # send the reimbrusement request to payment vendor
    return {'status': 'ok'}

# 2. Wrap the function with LongRunningFunctionTool
long_running_tool = LongRunningFunctionTool(func=ask_for_approval)
```

Java

```
import com.google.adk.agents.LlmAgent;
import com.google.adk.tools.LongRunningFunctionTool;
import java.util.HashMap;
import java.util.Map;

public class ExampleLongRunningFunction {

  // Define your Long Running function.
  // Ask for approval for the reimbursement.
  public static Map<String, Object> askForApproval(String purpose,
double amount) {
    // Simulate creating a ticket and sending a notification
    System.out.println(
        "Simulating ticket creation for purpose: " + purpose + ",
amount: " + amount);

    // Send a notification to the approver with the link of the ticket
    Map<String, Object> result = new HashMap<>();
    result.put("status", "pending");
    result.put("approver", "Sean Zhou");
    result.put("purpose", purpose);
    result.put("amount", amount);
    result.put("ticket-id", "approval-ticket-1");
    return result;
  }

  public static void main(String[] args) throws NoSuchMethodException
{
    // Pass the method to LongRunningFunctionTool.create
    LongRunningFunctionTool approveTool =

LongRunningFunctionTool.create(ExampleLongRunningFunction.class,
"askForApproval");

    // Include the tool in the agent
    LlmAgent approverAgent =
        LlmAgent.builder()
            // ...
            .tools(approveTool)
            .build();
  }
}
```

Intermediate / Final result Updates

Agent client received an event with long running function calls and check the status of the ticket. Then Agent client can send the intermediate or final response back to update the progress. The framework packages this value (even if it's None) into the content of the `FunctionResponse` sent back to the LLM.

Applies to only Java ADK

When passing `ToolContext` with Function Tools, ensure that one of the following is true:

- The Schema is passed with the ToolContext parameter in the function signature, like:

  ```
  @com.google.adk.tools.Annotations.Schema(name = "toolContext")
  ToolContext toolContext
  ```

  OR

- The following `-parameters` flag is set to the mvn compiler plugin

```
<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <version>3.14.0</version> <!-- or newer -->
            <configuration>
                <compilerArgs>
                    <arg>-parameters</arg>
                </compilerArgs>
            </configuration>
        </plugin>
    </plugins>
</build>
```

This constraint is temporary and will be removed.

Python

```python
# Agent Interaction
async def call_agent(query):

    def get_long_running_function_call(event: Event) -> types.FunctionCall:
        # Get the long running function call from the event
        if not event.long_running_tool_ids or not event.content or not
event.content.parts:
            return
        for part in event.content.parts:
            if (
                part
                and part.function_call
                and event.long_running_tool_ids
                and part.function_call.id in event.long_running_tool_ids
            ):
```

```
                return part.function_call

    def get_function_response(event: Event, function_call_id: str) ->
types.FunctionResponse:
        # Get the function response for the fuction call with specified id.
        if not event.content or not event.content.parts:
            return
        for part in event.content.parts:
            if (
                part
                and part.function_response
                and part.function_response.id == function_call_id
            ):
                return part.function_response

    content = types.Content(role='user', parts=[types.Part(text=query)])
    events = runner.run_async(user_id=USER_ID, session_id=SESSION_ID,
new_message=content)

    print("\nRunning agent...")
    events_async = runner.run_async(
        session_id=session.id, user_id=USER_ID, new_message=content
    )


    long_running_function_call, long_running_function_response, ticket_id = None,
None, None
    async for event in events_async:
        # Use helper to check for the specific auth request event
        if not long_running_function_call:
            long_running_function_call = get_long_running_function_call(event)
        else:
            long_running_function_response = get_function_response(event,
long_running_function_call.id)
            if long_running_function_response:
                ticket_id = long_running_function_response.response['ticket-id']
        if event.content and event.content.parts:
            if text := ''.join(part.text or '' for part in event.content.parts):
                print(f'[{event.author}]: {text}')


    if long_running_function_response:
        # query the status of the correpsonding ticket via tciket_id
        # send back an intermediate / final response
        updated_response = long_running_function_response.model_copy(deep=True)
        updated_response.response = {'status': 'approved'}
        async for event in runner.run_async(
          session_id=session.id, user_id=USER_ID, new_message=types.Content(parts=
[types.Part(function_response = updated_response)], role='user')
        ):
            if event.content and event.content.parts:
                if text := ''.join(part.text or '' for part in event.content.parts):
                    print(f'[{event.author}]: {text}')
```

Java

```java
import com.google.adk.agents.LlmAgent;
import com.google.adk.events.Event;
import com.google.adk.runner.InMemoryRunner;
import com.google.adk.runner.Runner;
import com.google.adk.sessions.Session;
import com.google.adk.tools.Annotations.Schema;
import com.google.adk.tools.LongRunningFunctionTool;
import com.google.adk.tools.ToolContext;
import com.google.common.collect.ImmutableList;
import com.google.common.collect.ImmutableMap;
import com.google.genai.types.Content;
import com.google.genai.types.FunctionCall;
import com.google.genai.types.FunctionResponse;
import com.google.genai.types.Part;
import java.util.Optional;
import java.util.UUID;
import java.util.concurrent.atomic.AtomicReference;
import java.util.stream.Collectors;

public class LongRunningFunctionExample {

  private static String USER_ID = "user123";

  @Schema(
      name = "create_ticket_long_running",
      description = """
          Creates a new support ticket with a specified urgency level.
          Examples of urgency are 'high', 'medium', or 'low'.
          The ticket creation is a long-running process, and its ID will be provided
when ready.
      """)
  public static void createTicketAsync(
      @Schema(
              name = "urgency",
              description =
                  "The urgency level for the new ticket, such as 'high', 'medium',
or 'low'.")
          String urgency,
      @Schema(name = "toolContext") // Ensures ADK injection
          ToolContext toolContext) {
    System.out.printf(
        "TOOL_EXEC: 'create_ticket_long_running' called with urgency: %s (Call ID:
%s)%n",
        urgency, toolContext.functionCallId().orElse("N/A"));
  }

  public static void main(String[] args) {
    LlmAgent agent =
        LlmAgent.builder()
            .name("ticket_agent")
            .description("Agent for creating tickets via a long-running task.")
            .model("gemini-2.0-flash")
            .tools(
                ImmutableList.of(
                    LongRunningFunctionTool.create(
                        LongRunningFunctionExample.class, "createTicketAsync")))
            .build();

    Runner runner = new InMemoryRunner(agent);
    Session session =
        runner.sessionService().createSession(agent.name(), USER_ID, null,
null).blockingGet();

    // --- Turn 1: User requests ticket ---
    System.out.println("\n--- Turn 1: User Request ---");
    Content initialUserMessage =
        Content.fromParts(Part.fromText("Create a high urgency ticket for me."));

    AtomicReference<String> funcCallIdRef = new AtomicReference<>();
    runner
        .runAsync(USER_ID, session.id(), initialUserMessage)
        .blockingForEach(
            event -> {
              printEventSummary(event, "T1");
              if (funcCallIdRef.get() == null) { // Capture the first relevant
function call ID

event.content().flatMap(Content::parts).orElse(ImmutableList.of()).stream()
                    .map(Part::functionCall)
                    .flatMap(Optional::stream)
                    .filter(fc ->
"create_ticket_long_running".equals(fc.name().orElse("")))
                    .findFirst()
                    .flatMap(FunctionCall::id)
                    .ifPresent(funcCallIdRef::set);
              }
            });

    if (funcCallIdRef.get() == null) {
      System.out.println("ERROR: Tool 'create_ticket_long_running' not called in
Turn 1.");
      return;
    }
    System.out.println("ACTION: Captured FunctionCall ID: " + funcCallIdRef.get());

    // --- Turn 2: App provides initial ticket_id (simulating async tool completion)
---
    System.out.println("\n--- Turn 2: App provides ticket_id ---");
    String ticketId = "TICKET-" + UUID.randomUUID().toString().substring(0,
8).toUpperCase();
    FunctionResponse ticketCreatedFuncResponse =
        FunctionResponse.builder()
            .name("create_ticket_long_running")
            .id(funcCallIdRef.get())
            .response(ImmutableMap.of("ticket_id", ticketId))
            .build();
    Content appResponseWithTicketId =
        Content.builder()
            .parts(
                ImmutableList.of(

Part.builder().functionResponse(ticketCreatedFuncResponse).build()))
            .role("user")
            .build();

    runner
        .runAsync(USER_ID, session.id(), appResponseWithTicketId)
        .blockingForEach(event -> printEventSummary(event, "T2"));
    System.out.println("ACTION: Sent ticket_id " + ticketId + " to agent.");

    // --- Turn 3: App provides ticket status update ---
    System.out.println("\n--- Turn 3: App provides ticket status ---");
    FunctionResponse ticketStatusFuncResponse =
        FunctionResponse.builder()
            .name("create_ticket_long_running")
            .id(funcCallIdRef.get())
            .response(ImmutableMap.of("status", "approved", "ticket_id", ticketId))
            .build();
    Content appResponseWithStatus =
        Content.builder()
            .parts(

ImmutableList.of(Part.builder().functionResponse(ticketStatusFuncResponse).build()))
            .role("user")
            .build();

    runner
        .runAsync(USER_ID, session.id(), appResponseWithStatus)
        .blockingForEach(event -> printEventSummary(event, "T3_FINAL"));
    System.out.println("Long running function completed successfully.");
  }

  private static void printEventSummary(Event event, String turnLabel) {
    event
        .content()
        .ifPresent(
            content -> {
              String text =
                  content.parts().orElse(ImmutableList.of()).stream()
                      .map(part -> part.text().orElse(""))
                      .filter(s -> !s.isEmpty())
                      .collect(Collectors.joining(" "));
              if (!text.isEmpty()) {
                System.out.printf("[%s][%s_TEXT]: %s%n", turnLabel, event.author(),
text);
              }
              content.parts().orElse(ImmutableList.of()).stream()
                  .map(Part::functionCall)
                  .flatMap(Optional::stream)
                  .findFirst() // Assuming one function call per relevant event for
simplicity
                  .ifPresent(
                      fc ->
                          System.out.printf(
                              "[%s][%s_CALL]: %s(%s) ID: %s%n",
                              turnLabel,
                              event.author(),
                              fc.name().orElse("N/A"),
                              fc.args().orElse(ImmutableMap.of()),
                              fc.id().orElse("N/A")));
            });
  }
}
```

Python complete example: File Processing Simulation

```python
# 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
implied.
# See the License for the specific language governing permissions and
# limitations under the License.

import asyncio
from typing import Any
from google.adk.agents import Agent
from google.adk.events import Event
from google.adk.runners import Runner
from google.adk.tools import LongRunningFunctionTool
from google.adk.sessions import InMemorySessionService
from google.genai import types


# 1. Define the long running function
def ask_for_approval(
    purpose: str, amount: float
) -> dict[str, Any]:
    """Ask for approval for the reimbursement."""
    # create a ticket for the approval
    # Send a notification to the approver with the link of the ticket
    return {'status': 'pending', 'approver': 'Sean Zhou', 'purpose' :
purpose, 'amount': amount, 'ticket-id': 'approval-ticket-1'}

def reimburse(purpose: str, amount: float) -> str:
    """Reimburse the amount of money to the employee."""
    # send the reimbrusement request to payment vendor
    return {'status': 'ok'}

# 2. Wrap the function with LongRunningFunctionTool
long_running_tool = LongRunningFunctionTool(func=ask_for_approval)


# 3. Use the tool in an Agent
file_processor_agent = Agent(
    # Use a model compatible with function calling
    model="gemini-2.0-flash",
    name='reimbursement_agent',
    instruction="""
      You are an agent whose job is to handle the reimbursement process
for
      the employees. If the amount is less than $100, you will
automatically
      approve the reimbursement.

      If the amount is greater than $100, you will
      ask for approval from the manager. If the manager approves, you
will
      call reimburse() to reimburse the amount to the employee. If the
manager
      rejects, you will inform the employee of the rejection.
    """,
    tools=[reimburse, long_running_tool]
)


APP_NAME = "human_in_the_loop"
USER_ID = "1234"
SESSION_ID = "session1234"

# 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=file_processor_agent, app_name=APP_NAME,
session_service=session_service)



# Agent Interaction
async def call_agent(query):

    def get_long_running_function_call(event: Event) ->
types.FunctionCall:
        # Get the long running function call from the event
        if not event.long_running_tool_ids or not event.content or not
event.content.parts:
            return
        for part in event.content.parts:
            if (
                part
                and part.function_call
                and event.long_running_tool_ids
                and part.function_call.id in
event.long_running_tool_ids
            ):
                return part.function_call

    def get_function_response(event: Event, function_call_id: str) ->
types.FunctionResponse:
        # Get the function response for the fuction call with specified
id.
        if not event.content or not event.content.parts:
            return
        for part in event.content.parts:
            if (
                part
                and part.function_response
                and part.function_response.id == function_call_id
            ):
                return part.function_response

    content = types.Content(role='user', parts=
[types.Part(text=query)])
    events = runner.run_async(user_id=USER_ID, session_id=SESSION_ID,
new_message=content)

    print("\nRunning agent...")
    events_async = runner.run_async(
        session_id=session.id, user_id=USER_ID, new_message=content
    )


    long_running_function_call, long_running_function_response,
ticket_id = None, None, None
    async for event in events_async:
        # Use helper to check for the specific auth request event
        if not long_running_function_call:
            long_running_function_call =
get_long_running_function_call(event)
        else:
            long_running_function_response =
get_function_response(event, long_running_function_call.id)
            if long_running_function_response:
                ticket_id =
long_running_function_response.response['ticket-id']
        if event.content and event.content.parts:
            if text := ''.join(part.text or '' for part in
event.content.parts):
                print(f'[{event.author}]: {text}')


    if long_running_function_response:
        # query the status of the correpsonding ticket via tciket_id
        # send back an intermediate / final response
        updated_response =
long_running_function_response.model_copy(deep=True)
        updated_response.response = {'status': 'approved'}
        async for event in runner.run_async(
          session_id=session.id, user_id=USER_ID,
new_message=types.Content(parts=[types.Part(function_response =
updated_response)], role='user')
        ):
            if event.content and event.content.parts:
                if text := ''.join(part.text or '' for part in
event.content.parts):
                    print(f'[{event.author}]: {text}')


# reimbursement that doesn't require approval
asyncio.run(call_agent("Please reimburse 50$ for meals"))
# await call_agent("Please reimburse 50$ for meals") # For Notebooks,
uncomment this line and comment the above line
# reimbursement that requires approval
asyncio.run(call_agent("Please reimburse 200$ for meals"))
# await call_agent("Please reimburse 200$ for meals") # For Notebooks,
uncomment this line and comment the above line
```

Key aspects of this example

- `LongRunningFunctionTool`: Wraps the supplied method/function; the framework handles sending yielded updates and the final return value as sequential FunctionResponses.
- **Agent instruction**: Directs the LLM to use the tool and understand the incoming FunctionResponse stream (progress vs. completion) for user updates.

- **Final return**: The function returns the final result dictionary, which is sent in the concluding FunctionResponse to indicate completion.

3. Agent-as-a-Tool

This powerful feature allows you to leverage the capabilities of other agents within your system by calling them as tools. The Agent-as-a-Tool enables you to invoke another agent to perform a specific task, effectively **delegating responsibility**. This is conceptually similar to creating a Python function that calls another agent and uses the agent's response as the function's return value.

Key difference from sub-agents

It's important to distinguish an Agent-as-a-Tool from a Sub-Agent.

- **Agent-as-a-Tool:** When Agent A calls Agent B as a tool (using Agent-as-a-Tool), Agent B's answer is **passed back** to Agent A, which then summarizes the answer and generates a response to the user. Agent A retains control and continues to handle future user input.
- **Sub-agent:** When Agent A calls Agent B as a sub-agent, the responsibility of answering the user is completely **transferred to Agent B**. Agent A is effectively out of the loop. All subsequent user input will be answered by Agent B.

Usage

To use an agent as a tool, wrap the agent with the AgentTool class.

Python

```
tools=[AgentTool(agent=agent_b)]
```

Java

```
AgentTool.create(agent)
```

Customization

The `AgentTool` class provides the following attributes for customizing its behavior:

- **skip_summarization: bool:** If set to True, the framework will **bypass the LLM-based summarization** of the tool agent's response. This can be useful when the tool's response is already well-formatted and requires no further processing.

Example

Python

```python
from google.adk.agents import Agent
from google.adk.runners import Runner
from google.adk.sessions import InMemorySessionService
from google.adk.tools.agent_tool import AgentTool
from google.genai import types

APP_NAME="summary_agent"
USER_ID="user1234"
SESSION_ID="1234"

summary_agent = Agent(
    model="gemini-2.0-flash",
    name="summary_agent",
    instruction="""You are an expert summarizer. Please read the
following text and provide a concise summary.""",
    description="Agent to summarize text",
)

root_agent = Agent(
    model='gemini-2.0-flash',
    name='root_agent',
    instruction="""You are a helpful assistant. When the user provides
a text, use the 'summarize' tool to generate a summary. Always forward
the user's message exactly as received to the 'summarize' tool, without
modifying or summarizing it yourself. Present the response from the
tool to the user.""",
    tools=[AgentTool(agent=summary_agent)]
)

# Session and Runner
session_service = InMemorySessionService()
session = 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)


# 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_message=content)

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


long_text = """Quantum computing represents a fundamentally different
approach to computation,
leveraging the bizarre principles of quantum mechanics to process
information. Unlike classical computers
```

```
that rely on bits representing either 0 or 1, quantum computers use
qubits which can exist in a state of superposition - effectively
being 0, 1, or a combination of both simultaneously. Furthermore,
qubits can become entangled,
meaning their fates are intertwined regardless of distance, allowing
for complex correlations. This parallelism and
interconnectedness grant quantum computers the potential to solve
specific types of incredibly complex problems - such
as drug discovery, materials science, complex system optimization, and
breaking certain types of cryptography - far
faster than even the most powerful classical supercomputers could ever
achieve, although the technology is still largely in its developmental
stages."""


call_agent(long_text)
```

Java

```java
import com.google.adk.agents.LlmAgent;
import com.google.adk.events.Event;
import com.google.adk.runner.InMemoryRunner;
import com.google.adk.sessions.Session;
import com.google.adk.tools.AgentTool;
import com.google.genai.types.Content;
import com.google.genai.types.Part;
import io.reactivex.rxjava3.core.Flowable;

public class AgentToolCustomization {

  private static final String APP_NAME = "summary_agent";
  private static final String USER_ID = "user1234";

  public static void initAgentAndRun(String prompt) {

    LlmAgent summaryAgent =
        LlmAgent.builder()
            .model("gemini-2.0-flash")
            .name("summaryAgent")
            .instruction(
                "You are an expert summarizer. Please read the
following text and provide a concise summary.")
            .description("Agent to summarize text")
            .build();

    // Define root_agent
    LlmAgent rootAgent =
        LlmAgent.builder()
            .model("gemini-2.0-flash")
            .name("rootAgent")
            .instruction(
                "You are a helpful assistant. When the user provides a
text, always use the 'summaryAgent' tool to generate a summary. Always
forward the user's message exactly as received to the 'summaryAgent'
tool, without modifying or summarizing it yourself. Present the
response from the tool to the user.")
            .description("Assistant agent")
            .tools(AgentTool.create(summaryAgent, true)) // Set
skipSummarization to true
            .build();
    // Create an InMemoryRunner
    InMemoryRunner runner = new InMemoryRunner(rootAgent, APP_NAME);
    // InMemoryRunner automatically creates a session service. Create a
session using the service
    Session session = runner.sessionService().createSession(APP_NAME,
USER_ID).blockingGet();
    Content userMessage = Content.fromParts(Part.fromText(prompt));

    // Run the agent
    Flowable<Event> eventStream = runner.runAsync(USER_ID,
session.id(), userMessage);

    // Stream event response
    eventStream.blockingForEach(
        event -> {
          if (event.finalResponse()) {
            System.out.println(event.stringifyContent());
          }
        });
  }

  public static void main(String[] args) {
    String longText =
        """
            Quantum computing represents a fundamentally different
approach to computation,
            leveraging the bizarre principles of quantum mechanics to
process information. Unlike classical computers
            that rely on bits representing either 0 or 1, quantum
computers use qubits which can exist in a state of superposition -
effectively
            being 0, 1, or a combination of both simultaneously.
Furthermore, qubits can become entangled,
            meaning their fates are intertwined regardless of distance,
allowing for complex correlations. This parallelism and
            interconnectedness grant quantum computers the potential to
solve specific types of incredibly complex problems - such
            as drug discovery, materials science, complex system
optimization, and breaking certain types of cryptography - far
            faster than even the most powerful classical supercomputers
could ever achieve, although the technology is still largely in its
developmental stages.""";

    initAgentAndRun(longText);
  }
}
```

How it works

1. When the `main_agent` receives the long text, its instruction tells it to use the 'summarize' tool for long texts.
2. The framework recognizes 'summarize' as an `AgentTool` that wraps the `summary_agent`.
3. Behind the scenes, the `main_agent` will call the `summary_agent` with the long text as input.

4. The `summary_agent` will process the text according to its instruction and generate a summary.
5. **The response from the `summary_agent` is then passed back to the `main_agent`.**
6. The `main_agent` can then take the summary and formulate its final response to the user (e.g., "Here's a summary of the text: ...")