Context - Agent Development Kit

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

Context¶

What are Context¶

In the Agent Development Kit (ADK), "context" refers to the crucial bundle of information available to your agent and its tools during specific operations. Think of it as the necessary background knowledge and resources needed to handle a current task or conversation turn effectively.

Agents often need more than just the latest user message to perform well. Context is essential because it enables:

1. **Maintaining State:** Remembering details across multiple steps in a conversation (e.g., user preferences, previous calculations, items in a shopping cart). This is primarily managed through **session state**.
2. **Passing Data:** Sharing information discovered or generated in one step (like an LLM call or a tool execution) with subsequent steps. Session state is key here too.
3. **Accessing Services:** Interacting with framework capabilities like:
4. **Artifact Storage:** Saving or loading files or data blobs (like PDFs, images, configuration files) associated with the session.
5. **Memory:** Searching for relevant information from past interactions or external knowledge sources connected to the user.
6. **Authentication:** Requesting and retrieving credentials needed by tools to access external APIs securely.
7. **Identity and Tracking:** Knowing which agent is currently running (`agent.name`) and uniquely identifying the current request-response cycle (`invocation_id`) for logging and debugging.
8. **Tool-Specific Actions:** Enabling specialized operations within tools, such as requesting authentication or searching memory, which require access to the current interaction's details.

The central piece holding all this information together for a single, complete user-request-to-final-response cycle (an **invocation**) is the `InvocationContext`. However, you typically won't create or manage this object directly. The ADK framework creates it when an invocation starts (e.g., via `runner.run_async`) and passes the relevant contextual information implicitly to your agent code, callbacks, and tools.

PythonJava

```
# Conceptual Pseudocode: How the framework provides context (Internal

# runner = Runner(agent=my_root_agent, session_service=..., artifact_s
# user_message = types.Content(...)
# session = session_service.get_session(...) # Or create new

# --- Inside runner.run_async(...) ---
# 1. Framework creates the main context for this specific run
# invocation_context = InvocationContext(
#     invocation_id="unique-id-for-this-run",
#     session=session,
#     user_content=user_message,
#     agent=my_root_agent, # The starting agent
#     session_service=session_service,
#     artifact_service=artifact_service,
#     memory_service=memory_service,
#     # ... other necessary fields ...
# )
#
# 2. Framework calls the agent's run method, passing the context impli
#    (The agent's method signature will receive it, e.g., runAsyncImpl
# await my_root_agent.run_async(invocation_context)
#   --- End Internal Logic ---
#
# As a developer, you work with the context objects provided in methoc
```

```
/* Conceptual Pseudocode: How the framework provides context (Interna
InMemoryRunner runner = new InMemoryRunner(agent);
Session session = runner
    .sessionService()
    .createSession(runner.appName(), USER_ID, initialState, SESSION_ID
    .blockingGet();

try (Scanner scanner = new Scanner(System.in, StandardCharsets.UTF_8))
  while (true) {
    System.out.print("\nYou > ");
  }
  String userInput = scanner.nextLine();
  if ("quit".equalsIgnoreCase(userInput)) {
    break;
  }
  Content userMsg = Content.fromParts(Part.fromText(userInput));
  Flowable<Event> events = runner.runAsync(session.userId(), session.i
  System.out.print("\nAgent > ");
  events.blockingForEach(event -> System.out.print(event.stringifyCont
}
```

The Different types of Context¶

While `InvocationContext` acts as the comprehensive internal container, ADK provides specialized context objects tailored to specific situations. This ensures you have the right tools and permissions for the task at hand without needing to handle the full complexity of the internal context everywhere. Here are the different "flavors" you'll encounter:

1. **`InvocationContext`**
2. **Where Used:** Received as the `ctx` argument directly within an agent's core implementation methods (`_run_async_impl` , `_run_live_impl`).

3. **Purpose:** Provides access to the *entire* state of the current invocation. This is the most comprehensive context object.
4. **Key Contents:** Direct access to `session` (including `state` and `events`), the current `agent` instance, `invocation_id`, initial `user_content`, references to configured services (`artifact_service`, `memory_service`, `session_service`), and fields related to live/streaming modes.
5. **Use Case:** Primarily used when the agent's core logic needs direct access to the overall session or services, though often state and artifact interactions are delegated to callbacks/tools which use their own contexts. Also used to control the invocation itself (e.g., setting `ctx.end_invocation = True`).

PythonJava

``` # Pseudocode: Agent implementation receiving InvocationContext from google.adk.agents import BaseAgent from google.adk.agents.invocation_context import InvocationContext from google.adk.events import Event from typing import AsyncGenerator

class MyAgent(BaseAgent): async def _run_async_impl(self, ctx: InvocationContext) -> AsyncGenerator[Event, None]: # Direct access example agent_name = ctx.agent.name session_id = ctx.session.id print(f"Agent {agent_name} running in session {session_id} for invocation {ctx.invocation_id}") # ... agent logic using ctx ... yield # ... event ...

```

``` // Pseudocode: Agent implementation receiving InvocationContext import com.google.adk.agents.BaseAgent; import com.google.adk.agents.InvocationContext;

```
LlmAgent root_agent =
    LlmAgent.builder()
        .model("gemini-***")
        .name("sample_agent")
        .description("Answers user questions.")
        .instruction(
            """
```
```

```
            provide instruction for the agent here.
            """
        )
        .tools(sampleTool)
        .outputKey("YOUR_KEY")
        .build();

 ConcurrentMap<String, Object> initialState = new ConcurrentHashMap<
 initialState.put("YOUR_KEY", "");

 InMemoryRunner runner = new InMemoryRunner(agent);
 Session session =
       runner
           .sessionService()
           .createSession(runner.appName(), USER_ID, initialState, S
           .blockingGet();

try (Scanner scanner = new Scanner(System.in, StandardCharsets.UTF_8
     while (true) {
       System.out.print("\nYou > ");
       String userInput = scanner.nextLine();

       if ("quit".equalsIgnoreCase(userInput)) {
         break;
       }

       Content userMsg = Content.fromParts(Part.fromText(userInput))
       Flowable<Event> events =
               runner.runAsync(session.userId(), session.id(), userM

       System.out.print("\nAgent > ");
       events.blockingForEach(event ->
               System.out.print(event.stringifyContent()));
   }

 protected Flowable<Event> runAsyncImpl(InvocationContext invocation
```

```
        // Direct access example
        String agentName = invocationContext.agent.name
        String sessionId = invocationContext.session.id
        String invocationId = invocationContext.invocationId
        System.out.println("Agent " + agent_name + " running in session
        // ... agent logic using ctx ...
    }
```

`` 2. **`ReadonlyContext`**

- **Where Used:** Provided in scenarios where only read access to basic information is needed and mutation is disallowed (e.g., `InstructionProvider` functions). It's also the base class for other contexts.
- **Purpose:** Offers a safe, read-only view of fundamental contextual details.
- **Key Contents:** `invocation_id`, `agent_name`, and a read-only *view* of the current `state`.

PythonJava

``` # Pseudocode: Instruction provider receiving ReadonlyContext from google.adk.agents import ReadonlyContext

def my_instruction_provider(context: ReadonlyContext) -> str: # Read-only access example user_tier = context.state().get("user_tier", "standard") # Can read state # context.state['new_key'] = 'value' # This would typically cause an error or be ineffective return f"Process the request for a {user_tier} user."

```

``` // Pseudocode: Instruction provider receiving ReadonlyContext import com.google.adk.agents.ReadonlyContext;

public String myInstructionProvider(ReadonlyContext context){ // Read-only access example String userTier = context.state().get("user_tier", "standard"); context.state().put('new_key', 'value'); //This would typically cause an error return "Process the request for a " + userTier + " user." }
```

`` 3. **`CallbackContext`**

- **Where Used:** Passed as `callback_context` to agent lifecycle callbacks (`before_agent_callback`, `after_agent_callback`) and model interaction callbacks (`before_model_callback`, `after_model_callback`).
- **Purpose:** Facilitates inspecting and modifying state, interacting with artifacts, and accessing invocation details *specifically within callbacks*.
- **Key Capabilities (Adds to `ReadonlyContext`):**
  - **Mutable `state` Property:** Allows reading *and writing* to session state. Changes made here (`callback_context.state['key'] = value`) are tracked and associated with the event generated by the framework after the callback.
  - **Artifact Methods:** `load_artifact(filename)` and `save_artifact(filename, part)` methods for interacting with the configured `artifact_service`.
  - Direct `user_content` access.

PythonJava

``` # Pseudocode: Callback receiving CallbackContext from google.adk.agents.callback_context import CallbackContext from google.adk.models import LlmRequest from google.genai import types from typing import Optional

def my_before_model_cb(callback_context: CallbackContext, request: LlmRequest) -> Optional[types.Content]: # Read/Write state example call_count = callback_context.state.get("model_calls", 0) callback_context.state["model_calls"] = call_count + 1 # Modify state

```
    # Optionally load an artifact
    # config_part = callback_context.load_artifact("model_config.json")
    print(f"Preparing model call #{call_count + 1} for invocation {call
    return None # Allow model call to proceed
```

```

``` // Pseudocode: Callback receiving CallbackContext import com.google.adk.agents.CallbackContext; import com.google.adk.models.LlmRequest; import com.google.genai.types.Content; import java.util.Optional;

public Maybe myBeforeModelCb(CallbackContext callbackContext, LlmRequest request){ // Read/Write state example callCount = callbackContext.state().get("model_calls", 0) callbackContext.state().put("model_calls") = callCount + 1 # Modify state

```
    // Optionally load an artifact
    // Maybe<Part> configPart = callbackContext.loadArtifact("model_con
    System.out.println("Preparing model call " + callCount + 1);
    return Maybe.empty(); // Allow model call to proceed
```

}

`` 4. **`ToolContext`**

- **Where Used:** Passed as `tool_context` to the functions backing `FunctionTool`s and to tool execution callbacks (`before_tool_callback`, `after_tool_callback`).
- **Purpose:** Provides everything `CallbackContext` does, plus specialized methods essential for tool execution, like handling authentication, searching memory, and listing artifacts.
- **Key Capabilities (Adds to `CallbackContext`):**
  - **Authentication Methods:** `request_credential(auth_config)` to trigger an auth flow, and `get_auth_response(auth_config)` to retrieve credentials provided by the user/system.
  - **Artifact Listing:** `list_artifacts()` to discover available artifacts in the session.
  - **Memory Search:** `search_memory(query)` to query the configured `memory_service`.
  - **`function_call_id` Property:** Identifies the specific function call from the LLM that triggered this tool execution, crucial for linking authentication requests or responses back correctly.
```

- **`actions` Property:** Direct access to the `EventActions` object for this step, allowing the tool to signal state changes, auth requests, etc.

PythonJava

``` # Pseudocode: Tool function receiving ToolContext from google.adk.tools import ToolContext from typing import Dict, Any

# Assume this function is wrapped by a FunctionTool def search_external_api(query: str, tool_context: ToolContext) -> Dict[str, Any]: api_key = tool_context.state.get("api_key") if not api_key: # Define required auth config # auth_config = AuthConfig(...) # tool_context.request_credential(auth_config) # Request credentials # Use the 'actions' property to signal the auth request has been made # tool_context.actions.requested_auth_configs[tool_context.function_call_id] = auth_config return {"status": "Auth Required"}

```
    # Use the API key...
    print(f"Tool executing for query '{query}' using API key. Invocatio

    # Optionally search memory or list artifacts
    # relevant_docs = tool_context.search_memory(f"info related to {que
    # available_files = tool_context.list_artifacts()

    return {"result": f"Data for {query} fetched."}
```

```

``` // Pseudocode: Tool function receiving ToolContext import com.google.adk.tools.ToolContext; import java.util.HashMap; import java.util.Map;

// Assume this function is wrapped by a FunctionTool public Map searchExternalApi(String query, ToolContext toolContext){ String apiKey = toolContext.state.get("api_key"); if(apiKey.isEmpty()){ // Define required auth config // authConfig = AuthConfig(...); // toolContext.requestCredential(authConfig); # Request credentials // Use the
```

'actions' property to signal the auth request has been made ... return Map.of("status", "Auth Required");

```
    // Use the API key...
    System.out.println("Tool executing for query " + query + " using AP

    // Optionally list artifacts
    // Single<List<String>> availableFiles = toolContext.listArtifacts(

    return Map.of("result", "Data for " + query + " fetched");
```

}

```

Understanding these different context objects and when to use them is key to effectively managing state, accessing services, and controlling the flow of your ADK application. The next section will detail common tasks you can perform using these contexts.

## Common Tasks Using Context¶

Now that you understand the different context objects, let's focus on how to use them for common tasks when building your agents and tools.

### Accessing Information¶

You'll frequently need to read information stored within the context.

- **Reading Session State:** Access data saved in previous steps or user/app-level settings. Use dictionary-like access on the `state` property.

PythonJava

``` # Pseudocode: In a Tool function from google.adk.tools import ToolContext

def my_tool(tool_context: ToolContext, **kwargs): user_pref = tool_context.state.get("user_display_preference", "default_mode") api_endpoint = tool_context.state.get("app:api_endpoint") # Read app-level state

```
    if user_pref == "dark_mode":
        # ... apply dark mode logic ...
        pass
    print(f"Using API endpoint: {api_endpoint}")
    # ... rest of tool logic ...
```

Pseudocode: In a Callback function from google.adk.agents.callback_context import CallbackContext

def my_callback(callback_context: CallbackContext, **kwargs): last_tool_result = callback_context.state.get("temp:last_api_result") # Read temporary state if last_tool_result: print(f"Found temporary result from last tool: {last_tool_result}") # ... callback logic ...

```

``` // Pseudocode: In a Tool function import com.google.adk.tools.ToolContext;

public void myTool(ToolContext toolContext){ String userPref = toolContext.state().get("user_display_preference"); String apiEndpoint = toolContext.state().get("app:api_endpoint"); // Read app-level state if(userPref.equals("dark_mode")){ // ... apply dark mode logic ... pass } System.out.println("Using API endpoint: " + api_endpoint); // ... rest of tool logic ... }

// Pseudocode: In a Callback function import com.google.adk.agents.CallbackContext;

```
    public void myCallback(CallbackContext callbackContext){
        String lastToolResult = (String) callbackContext.state().get("te
    }
    if(!(lastToolResult.isEmpty())){
        System.out.println("Found temporary result from last tool: " + l
    }
    // ... callback logic ...
```

``` * **Getting Current Identifiers:** Useful for logging or custom logic based on the current operation.
```

PythonJava

``` # Pseudocode: In any context (ToolContext shown) from google.adk.tools import ToolContext

def log_tool_usage(tool_context: ToolContext, **kwargs): agent_name = tool_context.agent_nameSystem.out.println("Found temporary result from last tool: " + lastToolResult); inv_id = tool_context.invocation_id func_call_id = getattr(tool_context, 'function_call_id', 'N/A') # Specific to ToolContext

```
print(f"Log: Invocation={inv_id}, Agent={agent_name}, FunctionCallI
```

```

``` // Pseudocode: In any context (ToolContext shown) import com.google.adk.tools.ToolContext;

public void logToolUsage(ToolContext toolContext){ String agentName = toolContext.agentName; String invId = toolContext.invocationId; String functionCallId = toolContext.functionCallId().get(); // Specific to ToolContext System.out.println("Log: Invocation= " + invId &+ " Agent= " + agentName); }

``` * **Accessing the Initial User Input:** Refer back to the message that started the current invocation.

PythonJava

``` # Pseudocode: In a Callback from google.adk.agents.callback_context import CallbackContext

def check_initial_intent(callback_context: CallbackContext, **kwargs): initial_text = "N/A" if callback_context.user_content and callback_context.user_content.parts: initial_text = callback_context.user_content.parts[0].text or "Non-text input"

```
print(f"This invocation started with user input: '{initial_text}'")
```

# Pseudocode: In an Agent's _run_async_impl # async def _run_async_impl(self, ctx: InvocationContext) -> AsyncGenerator[Event, None]: # if ctx.user_content and ctx.user_content.parts: # initial_text =
```

ctx.user_content.parts[0].text # print(f"Agent logic remembering initial query: {initial_text}") # ...

```

``` // Pseudocode: In a Callback import com.google.adk.agents.CallbackContext;

public void checkInitialIntent(CallbackContext callbackContext){ String initialText = "N/A"; if((!(callbackContext.userContent().isEmpty())) && (!(callbackContext.userContent().parts.isEmpty()))){ initialText = cbx.userContent().get().parts().get().get(0).text().get(); ... System.out.println("This invocation started with user input: " + initialText) } }

```

### Managing Session State¶

State is crucial for memory and data flow. When you modify state using `CallbackContext` or `ToolContext`, the changes are automatically tracked and persisted by the framework.

- **How it Works:** Writing to `callback_context.state['my_key'] = my_value` or `tool_context.state['my_key'] = my_value` adds this change to the `EventActions.state_delta` associated with the current step's event. The `SessionService` then applies these deltas when persisting the event.
- **Passing Data Between Tools:**

PythonJava

``` # Pseudocode: Tool 1 - Fetches user ID from google.adk.tools import ToolContext import uuid

def get_user_profile(tool_context: ToolContext) -> dict: user_id = str(uuid.uuid4()) # Simulate fetching ID # Save the ID to state for the next tool tool_context.state["temp:current_user_id"] = user_id return {"profile_status": "ID generated"}

Pseudocode: Tool 2 - Uses user ID from state def get_user_orders(tool_context: ToolContext) -> dict: user_id =

tool_context.state.get("temp:current_user_id") if not user_id: return {"error": "User ID not found in state"}

```
    print(f"Fetching orders for user ID: {user_id}")
    # ... logic to fetch orders using user_id ...
    return {"orders": ["order123", "order456"]}
```

```

``` // Pseudocode: Tool 1 - Fetches user ID import com.google.adk.tools.ToolContext; import java.util.UUID;

public Map getUserProfile(ToolContext toolContext){ String userId = UUID.randomUUID().toString(); // Save the ID to state for the next tool toolContext.state().put("temp:current_user_id", user_id); return Map.of("profile_status", "ID generated"); }

// Pseudocode: Tool 2 - Uses user ID from state public Map getUserOrders(ToolContext toolContext){ String userId = toolContext.state().get("temp:current_user_id"); if(userId.isEmpty()){ return Map.of("error", "User ID not found in state"); } System.out.println("Fetching orders for user id: " + userId); // ... logic to fetch orders using user_id ... return Map.of("orders", "order123"); }

``` * **Updating User Preferences:**

PythonJava

``` # Pseudocode: Tool or Callback identifies a preference from google.adk.tools import ToolContext # Or CallbackContext

def set_user_preference(tool_context: ToolContext, preference: str, value: str) -> dict: # Use 'user:' prefix for user-level state (if using a persistent SessionService) state_key = f"user:{preference}" tool_context.state[state_key] = value print(f"Set user preference '{preference}' to '{value}'") return {"status": "Preference updated"}

```

``` // Pseudocode: Tool or Callback identifies a preference import com.google.adk.tools.ToolContext; // Or CallbackContext

public Map setUserPreference(ToolContext toolContext, String preference, String value){ // Use 'user:' prefix for user-level state (if using a persistent SessionService) String stateKey = "user:" + preference; toolContext.state().put(stateKey, value); System.out.println("Set user preference '" + preference + "' to '" + value + "'"); return Map.of("status", "Preference updated"); }

``` * **State Prefixes:** While basic state is session-specific, prefixes like app: and user: can be used with persistent SessionService implementations (like DatabaseSessionService or VertexAiSessionService) to indicate broader scope (app-wide or user-wide across sessions). temp:` can denote data only relevant within the current invocation.

Working with Artifacts¶

Use artifacts to handle files or large data blobs associated with the session. Common use case: processing uploaded documents.

- **Document Summarizer Example Flow:**
- **Ingest Reference (e.g., in a Setup Tool or Callback):** Save the *path or URI* of the document, not the entire content, as an artifact.

  PythonJava

  ``` # Pseudocode: In a callback or initial tool from google.adk.agents import CallbackContext # Or ToolContext from google.genai import types

  def save_document_reference(context: CallbackContext, file_path: str) -> None: # Assume file_path is something like "gs://my-bucket/docs/report.pdf" or "/local/path/to/report.pdf" try: # Create a Part containing the path/URI text artifact_part = types.Part(text=file_path) version = context.save_artifact("document_to_summarize.txt", artifact_part) print(f"Saved document reference '{file_path}' as artifact version {version}") # Store the filename in state if needed by other tools context.state["temp:doc_artifact_name"] = "document_to_summarize.txt" except ValueError as e: print(f"Error saving artifact: {e}") # E.g., Artifact

service not configured except Exception as e: print(f"Unexpected error saving artifact reference: {e}")

# Example usage: # save_document_reference(callback_context, "gs://my-bucket/docs/report.pdf")

```

``` // Pseudocode: In a callback or initial tool import com.google.adk.agents.CallbackContext; import com.google.genai.types.Content; import com.google.genai.types.Part;

pubic void saveDocumentReference(CallbackContext context, String filePath){ // Assume file_path is something like "gs://my-bucket/docs/report.pdf" or "/local/path/to/report.pdf" try{ // Create a Part containing the path/URI text Part artifactPart = types.Part(filePath) Optional version = context.saveArtifact("document_to_summarize.txt", artifactPart) System.out.println("Saved document reference" + filePath + " as artifact version " + version); // Store the filename in state if needed by other tools context.state().put("temp:doc_artifact_name", "document_to_summarize.txt"); } catch(Exception e) { System.out.println("Unexpected error saving artifact reference: " + e); } }

// Example usage: // saveDocumentReference(context, "gs://my-bucket/docs/report.pdf")

``` 2. **Summarizer Tool:** Load the artifact to get the path/URI, read the actual document content using appropriate libraries, summarize, and return the result.

PythonJava

``` # Pseudocode: In the Summarizer tool function from google.adk.tools import ToolContext from google.genai import types # Assume libraries like google.cloud.storage or built-in open are available # Assume a 'summarize_text' function exists # from my_summarizer_lib import summarize_text

def summarize_document_tool(tool_context: ToolContext) -> dict: artifact_name = tool_context.state.get("temp:doc_artifact_name") if not

artifact_name: return {"error": "Document artifact name not found in state."}

```
try:
    # 1. Load the artifact part containing the path/URI
    artifact_part = tool_context.load_artifact(artifact_name)
    if not artifact_part or not artifact_part.text:
        return {"error": f"Could not load artifact or artifact ha

    file_path = artifact_part.text
    print(f"Loaded document reference: {file_path}")

    # 2. Read the actual document content (outside ADK context)
    document_content = ""
    if file_path.startswith("gs://"):
        # Example: Use GCS client library to download/read
        # from google.cloud import storage
        # client = storage.Client()
        # blob = storage.Blob.from_string(file_path, client=clien
        # document_content = blob.download_as_text() # Or bytes d
        pass # Replace with actual GCS reading logic
    elif file_path.startswith("/"):
         # Example: Use local file system
         with open(file_path, 'r', encoding='utf-8') as f:
             document_content = f.read()
    else:
        return {"error": f"Unsupported file path scheme: {file_pa

    # 3. Summarize the content
    if not document_content:
         return {"error": "Failed to read document content."}

    # summary = summarize_text(document_content) # Call your summ
    summary = f"Summary of content from {file_path}" # Placeholde

    return {"summary": summary}
```

```
except ValueError as e:
     return {"error": f"Artifact service error: {e}"}
except FileNotFoundError:
     return {"error": f"Local file not found: {file_path}"}
# except Exception as e: # Catch specific exceptions for GCS etc.
#      return {"error": f"Error reading document {file_path}: {e}
```

```

``` // Pseudocode: In the Summarizer tool function import com.google.adk.tools.ToolContext; import com.google.genai.types.Content; import com.google.genai.types.Part;

public Map summarizeDocumentTool(ToolContext toolContext){ String artifactName = toolContext.state().get("temp:doc_artifact_name"); if(artifactName.isEmpty()){ return Map.of("error", "Document artifact name not found in state."); } try{ // 1. Load the artifact part containing the path/ URI Maybe artifactPart = toolContext.loadArtifact(artifactName); if((artifactPart == null) || (artifactPart.text().isEmpty())){ return Map.of("error", "Could not load artifact or artifact has no text path: " + artifactName); } filePath = artifactPart.text(); System.out.println("Loaded document reference: " + filePath);

```
     // 2. Read the actual document content (outside ADK context)
     String documentContent = "";
     if(filePath.startsWith("gs://")){
         // Example: Use GCS client library to download/read into
         pass; // Replace with actual GCS reading logic
     } else if(){
         // Example: Use local file system to download/read into c
     } else{
         return Map.of("error", "Unsupported file path scheme: " +
     }

     // 3. Summarize the content
     if(documentContent.isEmpty()){
```

```
            return Map.of("error", "Failed to read document content."
        }

        // summary = summarizeText(documentContent) // Call your summ
        summary = "Summary of content from " + filePath; // Placehold

        return Map.of("summary", summary);
    } catch(IllegalArgumentException e){
        return Map.of("error", "Artifact service error " + filePath +
    } catch(FileNotFoundException e){
        return Map.of("error", "Local file not found " + filePath + e
    } catch(Exception e){
        return Map.of("error", "Error reading document " + filePath +
    }
}
```

``` * **Listing Artifacts:** Discover what files are available.

PythonJava

``` # Pseudocode: In a tool function from google.adk.tools import ToolContext

def check_available_docs(tool_context: ToolContext) -> dict: try: artifact_keys = tool_context.list_artifacts() print(f"Available artifacts: {artifact_keys}") return {"available_docs": artifact_keys} except ValueError as e: return {"error": f"Artifact service error: {e}"}

```

``` // Pseudocode: In a tool function import com.google.adk.tools.ToolContext;

public Map checkAvailableDocs(ToolContext toolContext){ try{ Single> artifactKeys = toolContext.listArtifacts(); System.out.println("Available artifacts" + artifactKeys.tostring()); return Map.of("availableDocs", "artifactKeys"); } catch(IllegalArgumentException e){ return Map.of("error", "Artifact service error: " + e); } }

```
```

Handling Tool Authentication¶

Currently supported in Python

Securely manage API keys or other credentials needed by tools.

```
 # Pseudocode: Tool requiring auth
from google.adk.tools import ToolContext
from google.adk.auth import AuthConfig # Assume appropriate AuthConfig

# Define your required auth configuration (e.g., OAuth, API Key)
MY_API_AUTH_CONFIG = AuthConfig(...)
AUTH_STATE_KEY = "user:my_api_credential" # Key to store retrieved cre

def call_secure_api(tool_context: ToolContext, request_data: str) -> d
    # 1. Check if credential already exists in state
    credential = tool_context.state.get(AUTH_STATE_KEY)

    if not credential:
        # 2. If not, request it
        print("Credential not found, requesting...")
        try:
            tool_context.request_credential(MY_API_AUTH_CONFIG)
            # The framework handles yielding the event. The tool execu
            return {"status": "Authentication required. Please provide
        except ValueError as e:
            return {"error": f"Auth error: {e}"} # e.g., function_call
        except Exception as e:
            return {"error": f"Failed to request credential: {e}"}

    # 3. If credential exists (might be from a previous turn after req
    #    or if this is a subsequent call after auth flow completed ext
    try:
        # Optionally, re-validate/retrieve if needed, or use directly
        # This might retrieve the credential if the external flow just
        auth_credential_obj = tool_context.get_auth_response(MY_API_AU
        api_key = auth_credential_obj.api_key # Or access_token, etc.
```

```
        # Store it back in state for future calls within the session
        tool_context.state[AUTH_STATE_KEY] = auth_credential_obj.model

        print(f"Using retrieved credential to call API with data: {req
        # ... Make the actual API call using api_key ...
        api_result = f"API result for {request_data}"

        return {"result": api_result}
    except Exception as e:
        # Handle errors retrieving/using the credential
        print(f"Error using credential: {e}")
        # Maybe clear the state key if credential is invalid?
        # tool_context.state[AUTH_STATE_KEY] = None
        return {"error": "Failed to use credential"}
```

Remember: `request_credential` *pauses the tool and signals the need for authentication. The user/system provides credentials, and on a subsequent call,* `get_auth_response` *(or checking state again) allows the tool to proceed.* The `tool_context.function_call_id` is used implicitly by the framework to link the request and response.

Leveraging Memory¶

Currently supported in Python

Access relevant information from the past or external sources.

```
# Pseudocode: Tool using memory search
from google.adk.tools import ToolContext

def find_related_info(tool_context: ToolContext, topic: str) -> dict:
    try:
        search_results = tool_context.search_memory(f"Information abou
        if search_results.results:
            print(f"Found {len(search_results.results)} memory results
```

```
            # Process search_results.results (which are SearchMemoryRe
            top_result_text = search_results.results[0].text
            return {"memory_snippet": top_result_text}
        else:
            return {"message": "No relevant memories found."}
    except ValueError as e:
        return {"error": f"Memory service error: {e}"} # e.g., Service
    except Exception as e:
        return {"error": f"Unexpected error searching memory: {e}"}
```

Advanced: Direct `InvocationContext` Usage¶

Currently supported in Python

While most interactions happen via `CallbackContext` or `ToolContext`, sometimes the agent's core logic (`_run_async_impl`/`_run_live_impl`) needs direct access.

```
# Pseudocode: Inside agent's _run_async_impl
from google.adk.agents import BaseAgent
from google.adk.agents.invocation_context import InvocationContext
from google.adk.events import Event
from typing import AsyncGenerator

class MyControllingAgent(BaseAgent):
    async def _run_async_impl(self, ctx: InvocationContext) -> AsyncGe
        # Example: Check if a specific service is available
        if not ctx.memory_service:
            print("Memory service is not available for this invocation
            # Potentially change agent behavior

        # Example: Early termination based on some condition
        if ctx.session.state.get("critical_error_flag"):
            print("Critical error detected, ending invocation.")
            ctx.end_invocation = True # Signal framework to stop proce
```

```
            yield Event(author=self.name, invocation_id=ctx.invocation
            return # Stop this agent's execution

        # ... Normal agent processing ...
        yield # ... event ...
```

Setting `ctx.end_invocation = True` is a way to gracefully stop the entire request-response cycle from within the agent or its callbacks/tools (via their respective context objects which also have access to modify the underlying `InvocationContext`'s flag).

Key Takeaways & Best Practices¶

- **Use the Right Context:** Always use the most specific context object provided (`ToolContext` in tools/tool-callbacks, `CallbackContext` in agent/model-callbacks, `ReadonlyContext` where applicable). Use the full `InvocationContext` (`ctx`) directly in `_run_async_impl` / `_run_live_impl` only when necessary.
- **State for Data Flow:** `context.state` is the primary way to share data, remember preferences, and manage conversational memory *within* an invocation. Use prefixes (`app:`, `user:`, `temp:`) thoughtfully when using persistent storage.
- **Artifacts for Files:** Use `context.save_artifact` and `context.load_artifact` for managing file references (like paths or URIs) or larger data blobs. Store references, load content on demand.
- **Tracked Changes:** Modifications to state or artifacts made via context methods are automatically linked to the current step's `EventActions` and handled by the `SessionService`.
- **Start Simple:** Focus on `state` and basic artifact usage first. Explore authentication, memory, and advanced `InvocationContext` fields (like those for live streaming) as your needs become more complex.

By understanding and effectively using these context objects, you can build more sophisticated, stateful, and capable agents with ADK.