Callbacks: Observe, Customize, and Control Agent Behavior - Agent Development Kit

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

Callbacks: Observe, Customize, and Control Agent Behavior¶

Introduction: What are Callbacks and Why Use Them?¶

Callbacks are a cornerstone feature of ADK, providing a powerful mechanism to hook into an agent's execution process. They allow you to observe, customize, and even control the agent's behavior at specific, predefined points without modifying the core ADK framework code.

What are they? In essence, callbacks are standard functions that you define. You then associate these functions with an agent when you create it. The ADK framework automatically calls your functions at key stages, letting you observe or intervene. Think of it like checkpoints during the agent's process:

- **Before the agent starts its main work on a request, and after it finishes:** When you ask an agent to do something (e.g., answer a question), it runs its internal logic to figure out the response.
- The `Before Agent` callback executes *right before* this main work begins for that specific request.
- The `After Agent` callback executes *right after* the agent has finished all its steps for that request and has prepared the final result, but just before the result is returned.
- This "main work" encompasses the agent's *entire* process for handling that single request. This might involve deciding to call an LLM, actually calling the LLM, deciding to use a tool, using the tool, processing the results, and finally putting together the answer. These callbacks essentially wrap the whole sequence from receiving the input to producing the final output for that one interaction.

- **Before sending a request to, or after receiving a response from, the Large Language Model (LLM):** These callbacks (`Before Model`, `After Model`) allow you to inspect or modify the data going to and coming from the LLM specifically.
- **Before executing a tool (like a Python function or another agent) or after it finishes:** Similarly, `Before Tool` and `After Tool` callbacks give you control points specifically around the execution of tools invoked by the agent.

intro_components.png

Why use them? Callbacks unlock significant flexibility and enable advanced agent capabilities:

- **Observe & Debug:** Log detailed information at critical steps for monitoring and troubleshooting.
- **Customize & Control:** Modify data flowing through the agent (like LLM requests or tool results) or even bypass certain steps entirely based on your logic.
- **Implement Guardrails:** Enforce safety rules, validate inputs/outputs, or prevent disallowed operations.
- **Manage State:** Read or dynamically update the agent's session state during execution.
- **Integrate & Enhance:** Trigger external actions (API calls, notifications) or add features like caching.

How are they added:

Code

PythonJava

```
 from google.adk.agents import LlmAgent
from google.adk.agents.callback_context import CallbackContext
from google.adk.models import LlmResponse, LlmRequest
from typing import Optional

# --- Define your callback function ---
def my_before_model_logic(
```

```python
    callback_context: CallbackContext, llm_request: LlmRequest
) -> Optional[LlmResponse]:
    print(f"Callback running before model call for agent: {callback_co
    # ... your custom logic here ...
    return None # Allow the model call to proceed

# --- Register it during Agent creation ---
my_agent = LlmAgent(
    name="MyCallbackAgent",
    model="gemini-2.0-flash", # Or your desired model
    instruction="Be helpful.",
    # Other agent parameters...
    before_model_callback=my_before_model_logic # Pass the function he
)
```

```java
import com.google.adk.agents.CallbackContext;
import com.google.adk.agents.Callbacks;
import com.google.adk.agents.LlmAgent;
import com.google.adk.models.LlmRequest;
import java.util.Optional;

public class AgentWithBeforeModelCallback {

  public static void main(String[] args) {
    // --- Define your callback logic ---
    Callbacks.BeforeModelCallbackSync myBeforeModelLogic =
        (CallbackContext callbackContext, LlmRequest llmRequest) -> {
          System.out.println(
              "Callback running before model call for agent: " + callb
          // ... your custom logic here ...

          // Return Optional.empty() to allow the model call to procee
          // similar to returning None in the Python example.
          // If you wanted to return a response and skip the model cal
          // you would return Optional.of(yourLlmResponse).
          return Optional.empty();
        };

    // --- Register it during Agent creation ---
    LlmAgent myAgent =
        LlmAgent.builder()
            .name("MyCallbackAgent")
            .model("gemini-2.0-flash") // Or your desired model
            .instruction("Be helpful.")
            // Other agent parameters...
            .beforeModelCallbackSync(myBeforeModelLogic) // Pass the c
            .build();
  }
}
```

The Callback Mechanism: Interception and Control¶

When the ADK framework encounters a point where a callback can run (e.g., just before calling the LLM), it checks if you provided a corresponding callback function for that agent. If you did, the framework executes your function.

Context is Key: Your callback function isn't called in isolation. The framework provides special **context objects** (`CallbackContext` or `ToolContext`) as arguments. These objects contain vital information about the current state of the agent's execution, including the invocation details, session state, and potentially references to services like artifacts or memory. You use these context objects to understand the situation and interact with the framework. (See the dedicated "Context Objects" section for full details).

Controlling the Flow (The Core Mechanism): The most powerful aspect of callbacks lies in how their **return value** influences the agent's subsequent actions. This is how you intercept and control the execution flow:

1. **`return None` (Allow Default Behavior):**

2. The specific return type can vary depending on the language. In Java, the equivalent return type is `Optional.empty()`. Refer to the API documentation for language specific guidance.

3. This is the standard way to signal that your callback has finished its work (e.g., logging, inspection, minor modifications to *mutable* input arguments like `llm_request`) and that the ADK agent should **proceed with its normal operation**.
4. For `before_*` callbacks (`before_agent`, `before_model`, `before_tool`), returning `None` means the next step in the sequence (running the agent logic, calling the LLM, executing the tool) will occur.
5. For `after_*` callbacks (`after_agent`, `after_model`, `after_tool`), returning `None` means the result just produced by the preceding step (the agent's output, the LLM's response, the tool's result) will be used as is.

6. **`return <Specific Object>` (Override Default Behavior):**

7. Returning a *specific type of object* (instead of `None`) is how you **override** the ADK agent's default behavior. The framework will use the object you return and *skip* the step that would normally follow or *replace* the result that was just generated.

8. **`before_agent_callback`** → **`types.Content`**: Skips the agent's main execution logic (`_run_async_impl` / `_run_live_impl`). The returned `Content` object is immediately treated as the agent's final output for this turn. Useful for handling simple requests directly or enforcing access control.
9. **`before_model_callback`** → **`LlmResponse`**: Skips the call to the external Large Language Model. The returned `LlmResponse` object is processed as if it were the actual response from the LLM. Ideal for implementing input guardrails, prompt validation, or serving cached responses.
10. **`before_tool_callback`** → **`dict`** **or** **`Map`**: Skips the execution of the actual tool function (or sub-agent). The returned `dict` is used as the result of the tool call, which is then typically passed back to the LLM. Perfect for validating tool arguments, applying policy restrictions, or returning mocked/cached tool results.

11. **`after_agent_callback`** → **`types.Content`**: *Replaces* the `Content` that the agent's run logic just produced.
12. **`after_model_callback`** → **`LlmResponse`**: *Replaces* the `LlmResponse` received from the LLM. Useful for sanitizing outputs, adding standard disclaimers, or modifying the LLM's response structure.
13. **`after_tool_callback`** → **`dict` or `Map`**: *Replaces* the `dict` result returned by the tool. Allows for post-processing or standardization of tool outputs before they are sent back to the LLM.

Conceptual Code Example (Guardrail):

This example demonstrates the common pattern for a guardrail using `before_model_callback`.

Code

PythonJava

```
from google.adk.agents import LlmAgent
from google.adk.agents.callback_context import CallbackContext
from google.adk.models import LlmResponse, LlmRequest
from google.adk.runners import Runner
from typing import Optional
from google.genai import types
from google.adk.sessions import InMemorySessionService

GEMINI_2_FLASH="gemini-2.0-flash"

# --- Define the Callback Function ---
def simple_before_model_modifier(
    callback_context: CallbackContext, llm_request: LlmRequest
) -> Optional[LlmResponse]:
    """Inspects/modifies the LLM request or skips the call."""
    agent_name = callback_context.agent_name
    print(f"[Callback] Before model call for agent: {agent_name}")

    # Inspect the last user message in the request contents
    last_user_message = ""
    if llm_request.contents and llm_request.contents[-1].role == 'user
         if llm_request.contents[-1].parts:
            last_user_message = llm_request.contents[-1].parts[0].text
    print(f"[Callback] Inspecting last user message: '{last_user_messa

    # --- Modification Example ---
    # Add a prefix to the system instruction
    original_instruction = llm_request.config.system_instruction or ty
    prefix = "[Modified by Callback] "
    # Ensure system_instruction is Content and parts list exists
    if not isinstance(original_instruction, types.Content):
         # Handle case where it might be a string (though config expec
         original_instruction = types.Content(role="system", parts=[ty
    if not original_instruction.parts:
        original_instruction.parts.append(types.Part(text="")) # Add a

    # Modify the text of the first part
    modified_text = prefix + (original_instruction.parts[0].text or ""
    original_instruction.parts[0].text = modified_text
    llm_request.config.system_instruction = original_instruction
    print(f"[Callback] Modified system instruction to: '{modified_text

    # --- Skip Example ---
    # Check if the last user message contains "BLOCK"
    if "BLOCK" in last_user_message.upper():
        print("[Callback] 'BLOCK' keyword found. Skipping LLM call.")
        # Return an LlmResponse to skip the actual LLM call
        return LlmResponse(
            content=types.Content(
                role="model",
                parts=[types.Part(text="LLM call was blocked by before
            )
        )
    else:
        print("[Callback] Proceeding with LLM call.")
        # Return None to allow the (modified) request to go to the LLM
        return None

# Create LlmAgent and Assign Callback
my_llm_agent = LlmAgent(
        name="ModelCallbackAgent",
        model=GEMINI_2_FLASH,
        instruction="You are a helpful assistant.", # Base instructio
        description="An LLM agent demonstrating before_model_callback"
        before_model_callback=simple_before_model_modifier # Assign th
)

APP_NAME = "guardrail_app"
USER_ID = "user_1"
SESSION_ID = "session_001"

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

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

call_agent("callback example")
```

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

```java
import com.google.adk.events.Event;
import com.google.adk.models.LlmRequest;
import com.google.adk.models.LlmResponse;
import com.google.adk.runner.InMemoryRunner;
import com.google.adk.sessions.Session;
import com.google.genai.types.Content;
import com.google.genai.types.GenerateContentConfig;
import com.google.genai.types.Part;
import io.reactivex.rxjava3.core.Flowable;
import java.util.ArrayList;
import java.util.List;
import java.util.Optional;
import java.util.stream.Collectors;

public class BeforeModelGuardrailExample {

  private static final String MODEL_ID = "gemini-2.0-flash";
  private static final String APP_NAME = "guardrail_app";
  private static final String USER_ID = "user_1";

  public static void main(String[] args) {
    BeforeModelGuardrailExample example = new BeforeModelGuardrailExam
    example.defineAgentAndRun("Tell me about quantum computing. This i
  }

  // --- Define your callback logic ---
  // Looks for the word "BLOCK" in the user prompt and blocks the call
  // Otherwise the LLM call proceeds as usual.
  public Optional<LlmResponse> simpleBeforeModelModifier(
      CallbackContext callbackContext, LlmRequest llmRequest) {
    System.out.println("[Callback] Before model call for agent: " + ca

    // Inspect the last user message in the request contents
    String lastUserMessageText = "";
    List<Content> requestContents = llmRequest.contents();
    if (requestContents != null && !requestContents.isEmpty()) {
      Content lastContent = requestContents.get(requestContents.size()
      if (lastContent.role().isPresent() && "user".equals(lastContent.
        lastUserMessageText =
            lastContent.parts().orElse(List.of()).stream()
                .flatMap(part -> part.text().stream())
                .collect(Collectors.joining(" ")); // Concatenate text
      }
    }
    System.out.println("[Callback] Inspecting last user message: '" +

    String prefix = "[Modified by Callback] ";
    GenerateContentConfig currentConfig =
        llmRequest.config().orElse(GenerateContentConfig.builder().bui
    Optional<Content> optOriginalSystemInstruction = currentConfig.sys

    Content conceptualModifiedSystemInstruction;
    if (optOriginalSystemInstruction.isPresent()) {
      Content originalSystemInstruction = optOriginalSystemInstructio
      List<Part> originalParts =
          new ArrayList<>(originalSystemInstruction.parts().orElse(Lis
      String originalText = "";

      if (!originalParts.isEmpty()) {
        Part firstPart = originalParts.get(0);
        if (firstPart.text().isPresent()) {
          originalText = firstPart.text().get();
        }
        originalParts.set(0, Part.fromText(prefix + originalText));
      } else {
        originalParts.add(Part.fromText(prefix));
      }
      conceptualModifiedSystemInstruction =
          originalSystemInstruction.toBuilder().parts(originalParts).b
    } else {
      conceptualModifiedSystemInstruction =
          Content.builder()
```

```
            .role("system")
            .parts(List.of(Part.fromText(prefix)))
            .build();
  }

  // This demonstrates building a new LlmRequest with the modified c
  llmRequest =
      llmRequest.toBuilder()
          .config(
              currentConfig.toBuilder()
                  .systemInstruction(conceptualModifiedSystemInstruc
                  .build())
          .build();

  System.out.println(
      "[Callback] Conceptually modified system instruction is: '"
          + llmRequest.config().get().systemInstruction().get().part

  // --- Skip Example ---
  // Check if the last user message contains "BLOCK"
  if (lastUserMessageText.toUpperCase().contains("BLOCK")) {
    System.out.println("[Callback] 'BLOCK' keyword found. Skipping L
    LlmResponse skipResponse =
        LlmResponse.builder()
            .content(
                Content.builder()
                    .role("model")
                    .parts(
                        List.of(
                            Part.builder()
                                .text("LLM call was blocked by befor
                                .build()))
                    .build())
            .build();
    return Optional.of(skipResponse);
  }
```

```
    System.out.println("[Callback] Proceeding with LLM call.");
    // Return Optional.empty() to allow the (modified) request to go t
    return Optional.empty();
  }

  public void defineAgentAndRun(String prompt) {
    // --- Create LlmAgent and Assign Callback ---
    LlmAgent myLlmAgent =
        LlmAgent.builder()
            .name("ModelCallbackAgent")
            .model(MODEL_ID)
            .instruction("You are a helpful assistant.") // Base instr
            .description("An LLM agent demonstrating before_model_call
            .beforeModelCallbackSync(this::simpleBeforeModelModifier)
            .build();

    // Session and Runner
    InMemoryRunner runner = new InMemoryRunner(myLlmAgent, APP_NAME);
    // InMemoryRunner automatically creates a session service. Create
    Session session = runner.sessionService().createSession(APP_NAME,
    Content userMessage =
        Content.fromParts(Part.fromText(prompt));

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

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

By understanding this mechanism of returning `None` versus returning specific objects, you can precisely control the agent's execution path, making callbacks an essential tool for building sophisticated and reliable agents with ADK.