Events - Agent Development Kit

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

Events¶

Events are the fundamental units of information flow within the Agent Development Kit (ADK). They represent every significant occurrence during an agent's interaction lifecycle, from initial user input to the final response and all the steps in between. Understanding events is crucial because they are the primary way components communicate, state is managed, and control flow is directed.

What Events Are and Why They Matter¶

An `Event` in ADK is an immutable record representing a specific point in the agent's execution. It captures user messages, agent replies, requests to use tools (function calls), tool results, state changes, control signals, and errors.

PythonJava

Technically, it's an instance of the `google.adk.events.Event` class, which builds upon the basic `LlmResponse` structure by adding essential ADK-specific metadata and an `actions` payload.

```
 # Conceptual Structure of an Event (Python)
# from google.adk.events import Event, EventActions
# from google.genai import types

# class Event(LlmResponse): # Simplified view
#     # --- LlmResponse fields ---
#     content: Optional[types.Content]
#     partial: Optional[bool]
#     # ... other response fields ...

#     # --- ADK specific additions ---
```

```
#     author: str           # 'user' or agent name
#     invocation_id: str    # ID for the whole interaction run
#     id: str               # Unique ID for this specific event
#     timestamp: float      # Creation time
#     actions: EventActions # Important for side-effects & control
#     branch: Optional[str] # Hierarchy path
#     # ...
```

In Java, this is an instance of the `com.google.adk.events.Event` class. It also builds upon a basic response structure by adding essential ADK-specific metadata and an `actions` payload.

```
// Conceptual Structure of an Event (Java - See com.google.adk.events
// Simplified view based on the provided com.google.adk.events.Event.j
// public class Event extends JsonBaseModel {
//     // --- Fields analogous to LlmResponse ---
//     private Optional<Content> content;
//     private Optional<Boolean> partial;
//     // ... other response fields like errorCode, errorMessage ...

//     // --- ADK specific additions ---
//     private String author;         // 'user' or agent name
//     private String invocationId;   // ID for the whole interaction
//     private String id;             // Unique ID for this specific e
//     private long timestamp;        // Creation time (epoch millisec
//     private EventActions actions;  // Important for side-effects &
//     private Optional<String> branch; // Hierarchy path
//     // ... other fields like turnComplete, longRunningToolIds etc.
// }
```

Events are central to ADK's operation for several key reasons:

1. **Communication:** They serve as the standard message format between the user interface, the `Runner`, agents, the LLM, and tools. Everything flows as an `Event`.
2. **Signaling State & Artifact Changes:** Events carry instructions for state modifications and track artifact updates. The `SessionService` uses these signals to ensure persistence. In Python changes are signaled via `event.actions.state_delta` and `event.actions.artifact_delta`.
3. **Control Flow:** Specific fields like `event.actions.transfer_to_agent` or `event.actions.escalate` act as signals that direct the framework, determining which agent runs next or if a loop should terminate.
4. **History & Observability:** The sequence of events recorded in `session.events` provides a complete, chronological history of an interaction, invaluable for debugging, auditing, and understanding agent behavior step-by-step.

In essence, the entire process, from a user's query to the agent's final answer, is orchestrated through the generation, interpretation, and processing of `Event` objects.

Understanding and Using Events¶

As a developer, you'll primarily interact with the stream of events yielded by the `Runner`. Here's how to understand and extract information from them:

Note

The specific parameters or method names for the primitives may vary slightly by SDK language (e.g., `event.content()` in Python, `event.content().get().parts()` in Java). Refer to the language-specific API documentation for details.

Identifying Event Origin and Type¶

Quickly determine what an event represents by checking:

- **Who sent it? (`event.author`)**

- `'user'`: Indicates input directly from the end-user.
- `'AgentName'`: Indicates output or action from a specific agent (e.g., `'WeatherAgent'`, `'SummarizerAgent'`).

- **What's the main payload? (`event.content` and `event.content.parts`)**

- **Text:** Indicates a conversational message. For Python, check if `event.content.parts[0].text` exists. For Java, check if `event.content()` is present, its `parts()` are present and not empty, and the first part's `text()` is present.

- **Tool Call Request:** Check `event.get_function_calls()`. If not empty, the LLM is asking to execute one or more tools. Each item in the list has `.name` and `.args`.
- **Tool Result:** Check `event.get_function_responses()`. If not empty, this event carries the result(s) from tool execution(s). Each item has `.name` and `.response` (the dictionary returned by the tool). *Note:* For history structuring, the `role` inside the `content` is often `'user'`, but the event `author` is typically the agent that requested the tool call.

- **Is it streaming output? (`event.partial`)** Indicates whether this is an incomplete chunk of text from the LLM.

- `True`: More text will follow.

- `False` or `None`/`Optional.empty()`: This part of the content is complete (though the overall turn might not be finished if `turn_complete` is also false).

PythonJava

```
# Pseudocode: Basic event identification (Python)
# async for event in runner.run_async(...):
#     print(f"Event from: {event.author}")
#
#     if event.content and event.content.parts:
#         if event.get_function_calls():
#             print("  Type: Tool Call Request")
```

```
#         elif event.get_function_responses():
#             print("  Type: Tool Result")
#         elif event.content.parts[0].text:
#             if event.partial:
#                 print("  Type: Streaming Text Chunk")
#             else:
#                 print("  Type: Complete Text Message")
#         else:
#             print("  Type: Other Content (e.g., code result)")
#     elif event.actions and (event.actions.state_delta or event.actio
#         print("  Type: State/Artifact Update")
#     else:
#         print("  Type: Control Signal or Other")
```

```
// Pseudocode: Basic event identification (Java)
// import com.google.genai.types.Content;
// import com.google.adk.events.Event;
// import com.google.adk.events.EventActions;

// runner.runAsync(...).forEach(event -> { // Assuming a synchronous s
//     System.out.println("Event from: " + event.author());
//
//     if (event.content().isPresent()) {
//         Content content = event.content().get();
//         if (!event.functionCalls().isEmpty()) {
//             System.out.println("  Type: Tool Call Request");
//         } else if (!event.functionResponses().isEmpty()) {
//             System.out.println("  Type: Tool Result");
//         } else if (content.parts().isPresent() && !content.parts().
//                     content.parts().get().get(0).text().isPresent())
//             if (event.partial().orElse(false)) {
//                 System.out.println("  Type: Streaming Text Chunk");
//             } else {
//                 System.out.println("  Type: Complete Text Message")
```

```
//             }
//         } else {
//             System.out.println("  Type: Other Content (e.g., code r
//         }
//     } else if (event.actions() != null &&
//                 ((event.actions().stateDelta() != null && !event.act
//                  (event.actions().artifactDelta() != null && !event.
//         System.out.println("  Type: State/Artifact Update");
//     } else {
//         System.out.println("  Type: Control Signal or Other");
//     }
// });
```

Extracting Key Information¶

Once you know the event type, access the relevant data:

- **Text Content:** Always check for the presence of content and parts before accessing text. In Python its `text = event.content.parts[0].text`.
- **Function Call Details:**

PythonJava

``` calls = event.get_function_calls() if calls: for call in calls: tool_name = call.name arguments = call.args # This is usually a dictionary print(f" Tool: {tool_name}, Args: {arguments}") # Application might dispatch execution based on this

```

``` import com.google.genai.types.FunctionCall; import com.google.common.collect.ImmutableList; import java.util.Map;

ImmutableList calls = event.functionCalls(); // from Event.java if (! calls.isEmpty()) { for (FunctionCall call : calls) { String toolName = call.name().get(); // args is Optional> Map arguments = call.args().get();
```

System.out.println(" Tool: " + toolName + ", Args: " + arguments); // Application might dispatch execution based on this } }

``` * **Function Response Details:**

PythonJava

``` responses = event.get_function_responses() if responses: for response in responses: tool_name = response.name result_dict = response.response # The dictionary returned by the tool print(f" Tool Result: {tool_name} -> {result_dict}")

```

``` import com.google.genai.types.FunctionResponse; import com.google.common.collect.ImmutableList; import java.util.Map;

ImmutableList responses = event.functionResponses(); // from Event.java if (! responses.isEmpty()) { for (FunctionResponse response : responses) { String toolName = response.name().get(); Map result= response.response().get(); // Check before getting the response System.out.println(" Tool Result: " + toolName + " -> " + result); } }

``` * **Identifiers:**

- `event.id`: Unique ID for this specific event instance.
- `event.invocation_id`: ID for the entire user-request-to-final-response cycle this event belongs to. Useful for logging and tracing.

Detecting Actions and Side Effects¶

The `event.actions` object signals changes that occurred or should occur. Always check if `event.actions` and it's fields/ methods exists before accessing them.

- **State Changes:** Gives you a collection of key-value pairs that were modified in the session state during the step that produced this event.

PythonJava

`delta = event.actions.state_delta` (a dictionary of `{key: value}` pairs).
```

``` if event.actions and event.actions.state_delta: print(f" State changes: {event.actions.state_delta}") # Update local UI or application state if necessary

```

`ConcurrentMap<String, Object> delta = event.actions().stateDelta();`

``` import java.util.concurrent.ConcurrentMap; import com.google.adk.events.EventActions;

EventActions actions = event.actions(); // Assuming event.actions() is not null if (actions != null && actions.stateDelta() != null && ! actions.stateDelta().isEmpty()) { ConcurrentMap stateChanges = actions.stateDelta(); System.out.println(" State changes: " + stateChanges); // Update local UI or application state if necessary }

`` * **Artifact Saves:** Gives you a collection indicating which artifacts were saved and their new version number (or relevant `Part` information).

PythonJava

`artifact_changes = event.actions.artifact_delta` (a dictionary of `{filename: version}`).

``` if event.actions and event.actions.artifact_delta: print(f" Artifacts saved: {event.actions.artifact_delta}") # UI might refresh an artifact list

```

`ConcurrentMap<String, Part> artifactChanges = event.actions().artifactDelta();`

``` import java.util.concurrent.ConcurrentMap; import com.google.genai.types.Part; import com.google.adk.events.EventActions;

EventActions actions = event.actions(); // Assuming event.actions() is not null if (actions != null && actions.artifactDelta() != null && ! actions.artifactDelta().isEmpty()) { ConcurrentMap artifactChanges = actions.artifactDelta(); System.out.println(" Artifacts saved: " +

artifactChanges); // UI might refresh an artifact list // Iterate through artifactChanges.entrySet() to get filename and Part details }

``` * **Control Flow Signals:** Check boolean flags or string values:

PythonJava

- `event.actions.transfer_to_agent` (string): Control should pass to the named agent.
- `event.actions.escalate` (bool): A loop should terminate.
- `event.actions.skip_summarization` (bool): A tool result should not be summarized by the LLM.

  ``` if event.actions: if event.actions.transfer_to_agent: print(f" Signal: Transfer to {event.actions.transfer_to_agent}") if event.actions.escalate: print(" Signal: Escalate (terminate loop)") if event.actions.skip_summarization: print(" Signal: Skip summarization for tool result")

  ```

- `event.actions().transferToAgent()` (returns `Optional<String>`): Control should pass to the named agent.
- `event.actions().escalate()` (returns `Optional<Boolean>`): A loop should terminate.
- `event.actions().skipSummarization()` (returns `Optional<Boolean>`): A tool result should not be summarized by the LLM.

``` import com.google.adk.events.EventActions; import java.util.Optional;

EventActions actions = event.actions(); // Assuming event.actions() is not null if (actions != null) { Optional transferAgent = actions.transferToAgent(); if (transferAgent.isPresent()) { System.out.println(" Signal: Transfer to " + transferAgent.get()); }

```
Optional<Boolean> escalate = actions.escalate();
if (escalate.orElse(false)) { // or escalate.isPresent() && escalate
    System.out.println(" Signal: Escalate (terminate loop)");
```

```
    }

    Optional<Boolean> skipSummarization = actions.skipSummarization();
    if (skipSummarization.orElse(false)) { // or skipSummarization.isPre
        System.out.println("  Signal: Skip summarization for tool result
    }
```

}

```

### Determining if an Event is a "Final" Response¶

Use the built-in helper method `event.is_final_response()` to identify events suitable for display as the agent's complete output for a turn.

- **Purpose:** Filters out intermediate steps (like tool calls, partial streaming text, internal state updates) from the final user-facing message(s).
- **When `True`?**
- The event contains a tool result (`function_response`) and `skip_summarization` is `True`.
- The event contains a tool call (`function_call`) for a tool marked as `is_long_running=True`. In Java, check if the `longRunningToolIds` list is empty:
  - `event.longRunningToolIds().isPresent() && !event.longRunningToolIds().get().isEmpty()` is `true`.
- OR, **all** of the following are met:
  - No function calls (`get_function_calls()` is empty).
  - No function responses (`get_function_responses()` is empty).
  - Not a partial stream chunk (`partial` is not `True`).
  - Doesn't end with a code execution result that might need further processing/display.
- **Usage:** Filter the event stream in your application logic.

PythonJava

``` # Pseudocode: Handling final responses in application (Python) # full_response_text = "" # async for event in runner.run_async(...): # # Accumulate streaming text if needed... # if event.partial and event.content and

event.content.parts and event.content.parts[0].text: # full_response_text += event.content.parts[0].text # # # Check if it's a final, displayable event # if event.is_final_response(): # print("\n--- Final Output Detected ---") # if event.content and event.content.parts and event.content.parts[0].text: # # If it's the final part of a stream, use accumulated text # final_text = full_response_text + (event.content.parts[0].text if not event.partial else "") # print(f"Display to user: {final_text.strip()}") # full_response_text = "" # Reset accumulator # elif event.actions and event.actions.skip_summarization and event.get_function_responses(): # # Handle displaying the raw tool result if needed # response_data = event.get_function_responses()[0].response # print(f"Display raw tool result: {response_data}") # elif hasattr(event, 'long_running_tool_ids') and event.long_running_tool_ids: # print("Display message: Tool is running in background...") # else: # # Handle other types of final responses if applicable # print("Display: Final non-textual response or signal.")

```

``` // Pseudocode: Handling final responses in application (Java) import com.google.adk.events.Event; import com.google.genai.types.Content; import com.google.genai.types.FunctionResponse; import java.util.Map;

StringBuilder fullResponseText = new StringBuilder(); runner.run(...).forEach(event -> { // Assuming a stream of events // Accumulate streaming text if needed... if (event.partial().orElse(false) && event.content().isPresent()) { event.content().flatMap(Content::parts).ifPresent(parts -> { if (!parts.isEmpty() && parts.get(0).text().isPresent()) { fullResponseText.append(parts.get(0).text().get()); } }); }

```
    // Check if it's a final, displayable event
    if (event.finalResponse()) { // Using the method from Event.java
        System.out.println("\n--- Final Output Detected ---");
        if (event.content().isPresent() &&
            event.content().flatMap(Content::parts).map(parts -> !parts
            // If it's the final part of a stream, use accumulated text
            String eventText = event.content().get().parts().get().get(
            String finalText = fullResponseText.toString() + (event.par
```

```
            System.out.println("Display to user: " + finalText.trim());
            fullResponseText.setLength(0); // Reset accumulator
        } else if (event.actions() != null && event.actions().skipSumma
                && !event.functionResponses().isEmpty()) {
            // Handle displaying the raw tool result if needed,
            // especially if finalResponse() was true due to other conc
            // or if you want to display skipped summarization results
            Map<String, Object> responseData = event.functionResponses(
            System.out.println("Display raw tool result: " + responseDa
        } else if (event.longRunningToolIds().isPresent() && !event.lon
            // This case is covered by event.finalResponse()
            System.out.println("Display message: Tool is running in bac
        } else {
            // Handle other types of final responses if applicable
            System.out.println("Display: Final non-textual response or
        }
    }
```

});

```

By carefully examining these aspects of an event, you can build robust applications that react appropriately to the rich information flowing through the ADK system.

## How Events Flow: Generation and Processing¶

Events are created at different points and processed systematically by the framework. Understanding this flow helps clarify how actions and history are managed.

- **Generation Sources:**
- **User Input:** The `Runner` typically wraps initial user messages or mid-conversation inputs into an `Event` with `author='user'`.
```

- **Agent Logic:** Agents (`BaseAgent`, `LlmAgent`) explicitly `yield Event(...)` objects (setting `author=self.name`) to communicate responses or signal actions.
- **LLM Responses:** The ADK model integration layer translates raw LLM output (text, function calls, errors) into `Event` objects, authored by the calling agent.
- **Tool Results:** After a tool executes, the framework generates an `Event` containing the `function_response`. The `author` is typically the agent that requested the tool, while the `role` inside the `content` is set to `'user'` for the LLM history.

- **Processing Flow:**

- **Yield/Return:** An event is generated and yielded (Python) or returned/emitted (Java) by its source.

- **Runner Receives:** The main `Runner` executing the agent receives the event.
- **SessionService Processing:** The `Runner` sends the event to the configured `SessionService`. This is a critical step:
    - **Applies Deltas:** The service merges `event.actions.state_delta` into `session.state` and updates internal records based on `event.actions.artifact_delta`. (Note: The actual artifact *saving* usually happened earlier when `context.save_artifact` was called).
    - **Finalizes Metadata:** Assigns a unique `event.id` if not present, may update `event.timestamp`.
    - **Persists to History:** Appends the processed event to the `session.events` list.
- **External Yield:** The `Runner` yields (Python) or returns/emits (Java) the processed event outwards to the calling application (e.g., the code that invoked `runner.run_async`).

This flow ensures that state changes and history are consistently recorded alongside the communication content of each event.

Common Event Examples (Illustrative Patterns)¶

Here are concise examples of typical events you might see in the stream:

- **User Input:**

``` { "author": "user", "invocation_id": "e-xyz...", "content": {"parts": [{"text": "Book a flight to London for next Tuesday"}]} // actions usually empty }

`` * **Agent Final Text Response:** (is_final_response() == True`)

``` { "author": "TravelAgent", "invocation_id": "e-xyz...", "content": {"parts": [{"text": "Okay, I can help with that. Could you confirm the departure city?"}]}, "partial": false, "turn_complete": true // actions might have state delta, etc. }

`` * **Agent Streaming Text Response:** (is_final_response() == False`)

``` { "author": "SummaryAgent", "invocation_id": "e-abc...", "content": {"parts": [{"text": "The document discusses three main points:"}]}, "partial": true, "turn_complete": false } // ... more partial=True events follow ...

`` * **Tool Call Request (by LLM):** (is_final_response() == False`)

``` { "author": "TravelAgent", "invocation_id": "e-xyz...", "content": {"parts": [{"function_call": {"name": "find_airports", "args": {"city": "London"}}}]} // actions usually empty }

`` * **Tool Result Provided (to LLM):** (is_final_response() depends on skip_summarization`)

``` { "author": "TravelAgent", // Author is agent that requested the call "invocation_id": "e-xyz...", "content": { "role": "user", // Role for LLM history "parts": [{"function_response": {"name": "find_airports", "response": {"result": ["LHR", "LGW", "STN"]}}}] } // actions might have skip_summarization=True }

`` * **State/Artifact Update Only:** (is_final_response() == False`)
```

``` { "author": "InternalUpdater", "invocation_id": "e-def...", "content": null, "actions": { "state_delta": {"user_status": "verified"}, "artifact_delta": {"verification_doc.pdf": 2} } }

`` * **Agent Transfer Signal:** (is_final_response() == False`)

``` { "author": "OrchestratorAgent", "invocation_id": "e-789...", "content": {"parts": [{"function_call": {"name": "transfer_to_agent", "args": {"agent_name": "BillingAgent"}}}]}, "actions": {"transfer_to_agent": "BillingAgent"} // Added by framework }

`` * **Loop Escalation Signal:** (is_final_response() == False`)

``` { "author": "CheckerAgent", "invocation_id": "e-loop...", "content": {"parts": [{"text": "Maximum retries reached."}]}, // Optional content "actions": {"escalate": true} }

```

Additional Context and Event Details¶

Beyond the core concepts, here are a few specific details about context and events that are important for certain use cases:

1. **`ToolContext.function_call_id` (Linking Tool Actions):**
2. When an LLM requests a tool (FunctionCall), that request has an ID. The `ToolContext` provided to your tool function includes this `function_call_id`.
3. **Importance:** This ID is crucial for linking actions like authentication back to the specific tool request that initiated them, especially if multiple tools are called in one turn. The framework uses this ID internally.
4. **How State/Artifact Changes are Recorded:**
5. When you modify state or save an artifact using `CallbackContext` or `ToolContext`, these changes aren't immediately written to persistent storage.
6. Instead, they populate the `state_delta` and `artifact_delta` fields within the `EventActions` object.

7. This `EventActions` object is attached to the *next event* generated after the change (e.g., the agent's response or a tool result event).
8. The `SessionService.append_event` method reads these deltas from the incoming event and applies them to the session's persistent state and artifact records. This ensures changes are tied chronologically to the event stream.

9. **State Scope Prefixes (`app:`, `user:`, `temp:`):**

10. When managing state via `context.state`, you can optionally use prefixes:

    - `app:my_setting`: Suggests state relevant to the entire application (requires a persistent `SessionService`).
    - `user:user_preference`: Suggests state relevant to the specific user across sessions (requires a persistent `SessionService`).
    - `temp:intermediate_result` or no prefix: Typically session-specific or temporary state for the current invocation.
11. The underlying `SessionService` determines how these prefixes are handled for persistence.

12. **Error Events:**

13. An `Event` can represent an error. Check the `event.error_code` and `event.error_message` fields (inherited from `LlmResponse`).

14. Errors might originate from the LLM (e.g., safety filters, resource limits) or potentially be packaged by the framework if a tool fails critically. Check tool `FunctionResponse` content for typical tool-specific errors.

    ``` // Example Error Event (conceptual) { "author": "LLMAgent", "invocation_id": "e-err...", "content": null, "error_code": "SAFETY_FILTER_TRIGGERED", "error_message": "Response blocked due to safety settings.", "actions": {} }

    ```

These details provide a more complete picture for advanced use cases involving tool authentication, state persistence scope, and error handling within the event stream.

Best Practices for Working with Events¶

To use events effectively in your ADK applications:

- **Clear Authorship:** When building custom agents, ensure correct attribution for agent actions in the history. The framework generally handles authorship correctly for LLM/tool events.

PythonJava

Use `yield Event(author=self.name, ...)` in `BaseAgent` subclasses.

When constructing an `Event` in your custom agent logic, set the author, for example: `Event.builder().author(this.getAgentName()) // ... .build();` * **Semantic Content & Actions:** Use `event.content` for the core message/data (text, function call/response). Use `event.actions` specifically for signaling side effects (state/artifact deltas) or control flow (`transfer`, `escalate`, `skip_summarization`). * **Idempotency Awareness:** Understand that the `SessionService` is responsible for applying the state/artifact changes signaled in `event.actions`. While ADK services aim for consistency, consider potential downstream effects if your application logic re-processes events. * **Use `is_final_response()`:** Rely on this helper method in your application/UI layer to identify complete, user-facing text responses. Avoid manually replicating its logic. * **Leverage History:** The session's event list is your primary debugging tool. Examine the sequence of authors, content, and actions to trace execution and diagnose issues. * **Use Metadata:** Use `invocation_id` to correlate all events within a single user interaction. Use `event.id` to reference specific, unique occurrences.

Treating events as structured messages with clear purposes for their content and actions is key to building, debugging, and managing complex agent behaviors in ADK.