

Runtime

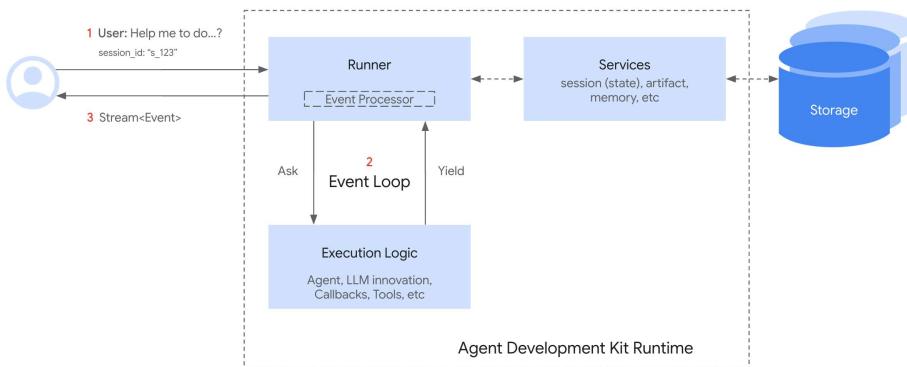
What is runtime?

The ADK Runtime is the underlying engine that powers your agent application during user interactions. It's the system that takes your defined agents, tools, and callbacks and orchestrates their execution in response to user input, managing the flow of information, state changes, and interactions with external services like LLMs or storage.

Think of the Runtime as the **"engine"** of your agentic application. You define the parts (agents, tools), and the Runtime handles how they connect and run together to fulfill a user's request.

Core Idea: The Event Loop

At its heart, the ADK Runtime operates on an **Event Loop**. This loop facilitates a back-and-forth communication between the `Runner` component and your defined "Execution Logic" (which includes your Agents, the LLM calls they make, Callbacks, and Tools).



In simple terms:

1. The `Runner` receives a user query and asks the main `Agent` to start processing.

2. The `Agent` (and its associated logic) runs until it has something to report (like a response, a request to use a tool, or a state change) – it then **yields** an `Event`.
3. The `Runner` receives this `Event`, processes any associated actions (like saving state changes via `Services`), and forwards the event onwards (e.g., to the user interface).
4. Only after the `Runner` has processed the event does the `Agent`'s logic **resume** from where it paused, now potentially seeing the effects of the changes committed by the `Runner`.
5. This cycle repeats until the agent has no more events to yield for the current user query.

This event-driven loop is the fundamental pattern governing how ADK executes your agent code.

The Heartbeat: The Event Loop - Inner workings

The Event Loop is the core operational pattern defining the interaction between the `Runner` and your custom code (Agents, Tools, Callbacks, collectively referred to as "Execution Logic" or "Logic Components" in the design document). It establishes a clear division of responsibilities:

Runner's Role (Orchestrator)

The `Runner` acts as the central coordinator for a single user invocation. Its responsibilities in the loop are:

1. **Initiation:** Receives the end user's query (`new_message`) and typically appends it to the session history via the `SessionService`.
2. **Kick-off:** Starts the event generation process by calling the main agent's execution method (e.g., `agent_to_run.run_async(...)`).
3. **Receive & Process:** Waits for the agent logic to `yield` an `Event`. Upon receiving an event, the `Runner` **promptly processes** it. This involves:
 - Using configured `Services` (`SessionService`, `ArtifactService`, `MemoryService`) to commit changes indicated in `event.actions` (like `state_delta`, `artifact_delta`).
 - Performing other internal bookkeeping.

4. **Yield Upstream:** Forwards the processed event onwards (e.g., to the calling application or UI for rendering).
5. **Iterate:** Signals the agent logic that processing is complete for the yielded event, allowing it to resume and generate the next event.

Conceptual Runner Loop:

```
# Simplified view of Runner's main loop logic
def run(new_query, ...) -> Generator[Event]:
    # 1. Append new_query to session event history (via
    # SessionService)
    session_service.append_event(session, Event(author='user',
content=new_query))

    # 2. Kick off event loop by calling the agent
    agent_event_generator = agent_to_run.run_async(context)

    async for event in agent_event_generator:
        # 3. Process the generated event and commit changes
        session_service.append_event(session, event) # Commits
        state/artifact deltas etc.
        # memory_service.update_memory(...) # If applicable
        # artifact_service might have already been called via
        context during agent run

        # 4. Yield event for upstream processing (e.g., UI
        rendering)
        yield event
        # Runner implicitly signals agent generator can
        continue after yielding
```

Execution Logic's Role (Agent, Tool, Callback)

Your code within agents, tools, and callbacks is responsible for the actual computation and decision-making. Its interaction with the loop involves:

1. **Execute:** Runs its logic based on the current `InvocationContext`, including the session state as *it was when execution resumed*.
2. **Yield:** When the logic needs to communicate (send a message, call a tool, report a state change), it constructs an `Event` containing the relevant content and actions, and then `yield`s this event back to the `Runner`.
3. **Pause:** Crucially, execution of the agent logic **pauses immediately** after the `yield` statement. It waits for the `Runner` to complete step 3 (processing and committing).

4. **Resume:** Only after the `Runner` has processed the yielded event does the agent logic resume execution from the statement immediately following the `yield`.
5. **See Updated State:** Upon resumption, the agent logic can now reliably access the session state (`ctx.session.state`) reflecting the changes that were committed by the `Runner` from the previously yielded event.

Conceptual Execution Logic:

```
# Simplified view of logic inside Agent.run_async, callbacks,
or tools

# ... previous code runs based on current state ...

# 1. Determine a change or output is needed, construct the
event
# Example: Updating state
update_data = {'field_1': 'value_2'}
event_with_state_change = Event(
    author=self.name,
    actions=EventActions(state_delta=update_data),
    content=types.Content(parts=[types.Part(text="State
updated.")])
    # ... other event fields ...
)

# 2. Yield the event to the Runner for processing & commit
yield event_with_state_change
# <<<<<<<< EXECUTION PAUSES HERE >>>>>>>>

# <<<<<<<< RUNNER PROCESSES & COMMITS THE EVENT
>>>>>>>>>

# 3. Resume execution ONLY after Runner is done processing the
above event.
# Now, the state committed by the Runner is reliably reflected.
# Subsequent code can safely assume the change from the yielded
event happened.
val = ctx.session.state['field_1']
# here `val` is guaranteed to be "value_2" (assuming Runner
committed successfully)
print(f"Resumed execution. Value of field_1 is now: {val}")

# ... subsequent code continues ...
# Maybe yield another event later...
```

This cooperative yield/pause/resume cycle between the `Runner` and your Execution Logic, mediated by `Event` objects, forms the core of the ADK Runtime.

Key components of the Runtime

Several components work together within the ADK Runtime to execute an agent invocation. Understanding their roles clarifies how the event loop functions:

1. Runner

- **Role:** The main entry point and orchestrator for a single user query (`run_async`).
- **Function:** Manages the overall Event Loop, receives events yielded by the Execution Logic, coordinates with Services to process and commit event actions (state/artifact changes), and forwards processed events upstream (e.g., to the UI). It essentially drives the conversation turn by turn based on yielded events. (Defined in `google.adk.runners.runner.py`).

2. Execution Logic Components

- **Role:** The parts containing your custom code and the core agent capabilities.
- **Components:**
 - **Agent** (`BaseAgent`, `LlmAgent`, etc.): Your primary logic units that process information and decide on actions. They implement the `_run_async_impl` method which yields events.
 - **Tools** (`BaseTool`, `FunctionTool`, `AgentTool`, etc.): External functions or capabilities used by agents (often `LlmAgent`) to interact with the outside world or perform specific tasks. They execute and return results, which are then wrapped in events.
 - **Callbacks** (Functions): User-defined functions attached to agents (e.g., `before_agent_callback`, `after_model_callback`) that hook into specific points in the execution flow, potentially modifying behavior or state, whose effects are captured in events.
 - **Function:** Perform the actual thinking, calculation, or external interaction. They communicate their results or needs by **yielding** **Event objects** and pausing until the Runner processes them.

3. Event

- **Role:** The message passed back and forth between the `Runner` and the Execution Logic.
- **Function:** Represents an atomic occurrence (user input, agent text, tool call/result, state change request, control signal). It carries both the content of the occurrence and the intended side effects (`actions` like `state_delta`). (Defined in `google.adk.events.event.py`).

4. Services

- **Role:** Backend components responsible for managing persistent or shared resources. Used primarily by the `Runner` during event processing.
- **Components:**
 - `SessionService` (`BaseSessionService`, `InMemorySessionService`, etc.): Manages `Session` objects, including saving/loading them, applying `state_delta` to the session state, and appending events to the `event history`.
 - `ArtifactService` (`BaseArtifactService`, `InMemoryArtifactService`, `GcsArtifactService`, etc.): Manages the storage and retrieval of binary artifact data. Although `save_artifact` is called via context during execution logic, the `artifact_delta` in the event confirms the action for the `Runner/SessionService`.
 - `MemoryService` (`BaseMemoryService`, etc.): (Optional) Manages long-term semantic memory across sessions for a user.
- **Function:** Provide the persistence layer. The `Runner` interacts with them to ensure changes signaled by `event.actions` are reliably stored before the Execution Logic resumes.

5. Session

- **Role:** A data container holding the state and history for one specific conversation between a user and the application.

- **Function:** Stores the current `state` dictionary, the list of all past `events` (`event history`), and references to associated artifacts. It's the primary record of the interaction, managed by the `SessionService`. (Defined in `google.adk.sessions.session.py`).

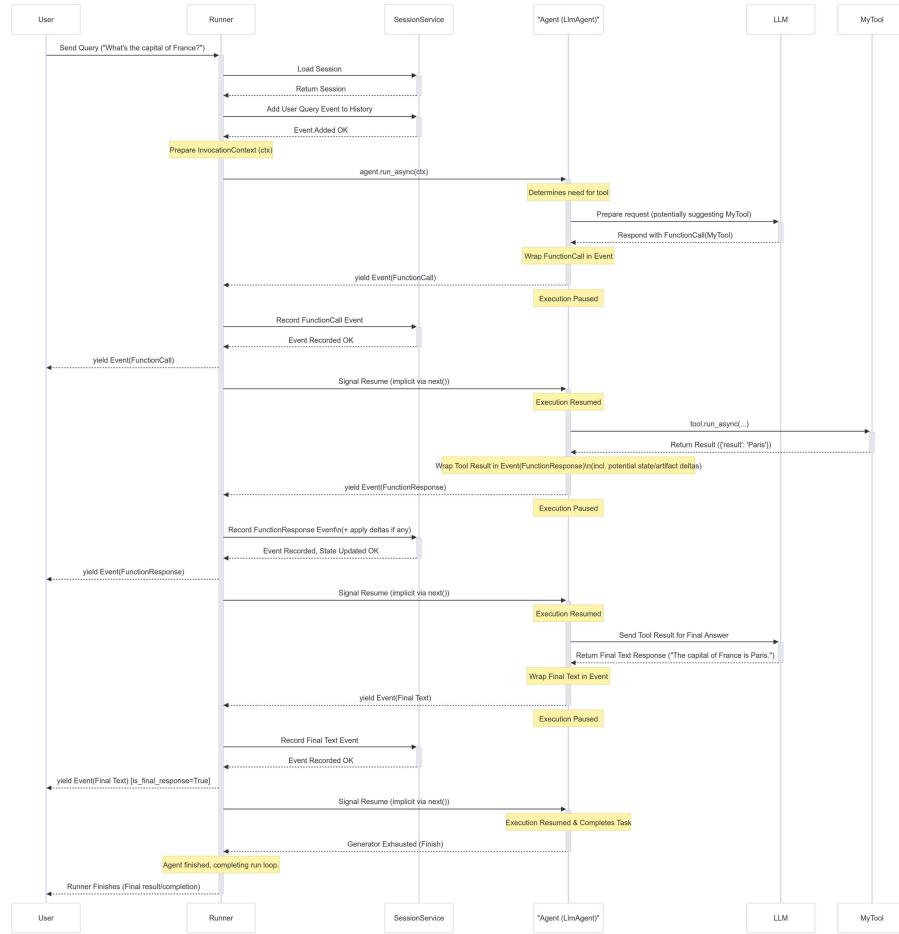
6. Invocation

- **Role:** A conceptual term representing everything that happens in response to a *single* user query, from the moment the `Runner` receives it until the agent logic finishes yielding events for that query.
- **Function:** An invocation might involve multiple agent runs (if using agent transfer or `AgentTool`), multiple LLM calls, tool executions, and callback executions, all tied together by a single `invocation_id` within the `InvocationContext`.

These players interact continuously through the Event Loop to process a user's request.

How It Works: A Simplified Invocation

Let's trace a simplified flow for a typical user query that involves an LLM agent calling a tool:



Step-by-Step Breakdown

- User Input:** The User sends a query (e.g., "What's the capital of France?").
- Runner Starts:** Runner .run_async begins. It interacts with the SessionService to load the relevant Session and adds the user query as the first Event to the session history. An InvocationContext (ctx) is prepared.
- Agent Execution:** The Runner calls agent.run_async(ctx) on the designated root agent (e.g., an LlmAgent).
- LLM Call (Example):** The Agent_Llm determines it needs information, perhaps by calling a tool. It prepares a request for the LLM. Let's assume the LLM decides to call MyTool.
- Yield FunctionCall Event:** The Agent_Llm receives the FunctionCall response from the LLM, wraps it in an Event(author='Agent_Llm',

- content=Content(parts=[Part(function_call=...)])), and yields this event.
6. **Agent Pauses:** The Agent_Llm's execution pauses immediately after the yield.
 7. **Runner Processes:** The Runner receives the FunctionCall event. It passes it to the SessionService to record it in the history. The Runner then yields the event upstream to the User (or application).
 8. **Agent Resumes:** The Runner signals that the event is processed, and Agent_Llm resumes execution.
 9. **Tool Execution:** The Agent_Llm's internal flow now proceeds to execute the requested MyTool. It calls tool.run_async(...).
 10. **Tool Returns Result:** MyTool executes and returns its result (e.g., {'result': 'Paris'}).
 11. **Yield FunctionResponse Event:** The agent (Agent_Llm) wraps the tool result into an Event containing a FunctionResponse part (e.g., Event(author='Agent_Llm', content=Content(role='user', parts=[Part(function_response=...)]))). This event might also contain actions if the tool modified state (state_delta) or saved artifacts (artifact_delta). The agent yields this event.
 12. **Agent Pauses:** Agent_Llm pauses again.
 13. **Runner Processes:** Runner receives the FunctionResponse event. It passes it to SessionService which applies any state_delta / artifact_delta and adds the event to history. Runner yields the event upstream.
 14. **Agent Resumes:** Agent_Llm resumes, now knowing the tool result and any state changes are committed.
 15. **Final LLM Call (Example):** Agent_Llm sends the tool result back to the LLM to generate a natural language response.
 16. **Yield Final Text Event:** Agent_Llm receives the final text from the LLM, wraps it in an Event(author='Agent_Llm', content=Content(parts=[Part(text=...)])), and yields it.
 17. **Agent Pauses:** Agent_Llm pauses.
 18. **Runner Processes:** Runner receives the final text event, passes it to SessionService for history, and yields it upstream to the User. This is likely marked as the is_final_response().

19. **Agent Resumes & Finishes:** Agent_Llm resumes. Having completed its task for this invocation, its `run_async` generator finishes.

20. **Runner Completes:** The Runner sees the agent's generator is exhausted and finishes its loop for this invocation.

This yield/pause/process/resume cycle ensures that state changes are consistently applied and that the execution logic always operates on the most recently committed state after yielding an event.

Important Runtime Behaviors

Understanding a few key aspects of how the ADK Runtime handles state, streaming, and asynchronous operations is crucial for building predictable and efficient agents.

State Updates & Commitment Timing

- **The Rule:** When your code (in an agent, tool, or callback) modifies the session state (e.g., `context.state['my_key'] = 'new_value'`), this change is initially recorded locally within the current `InvocationContext`. The change is only **guaranteed to be persisted** (saved by the `SessionService`) *after* the `Event` carrying the corresponding `state_delta` in its `actions` has been `yield`-ed by your code and subsequently processed by the Runner .
- **Implication:** Code that runs *after* resuming from a `yield` can reliably assume that the state changes signaled in the *yielded* event have been committed.

```
# Inside agent logic (conceptual)

# 1. Modify state
ctx.session.state['status'] = 'processing'
event1 = Event(..., actions=EventActions(state_delta={'status': 'processing'}))

# 2. Yield event with the delta
yield event1
# --- PAUSE --- Runner processes event1, SessionService commits
# 'status' = 'processing' ---

# 3. Resume execution
```

```
# Now it's safe to rely on the committed state
current_status = ctx.session.state['status'] # Guaranteed to be
'processing'
print(f"Status after resuming: {current_status}")
```

"Dirty Reads" of Session State

- **Definition:** While commitment happens *after* the yield, code running *later within the same invocation*, but *before* the state-changing event is actually yielded and processed, **can often see the local, uncommitted changes**. This is sometimes called a "dirty read".
- **Example:**

```
# Code in before_agent_callback
callback_context.state['field_1'] = 'value_1'
# State is locally set to 'value_1', but not yet committed by
Runner

# ... agent runs ...

# Code in a tool called later *within the same invocation*
# Readable (dirty read), but 'value_1' isn't guaranteed
persistent yet.
val = tool_context.state['field_1'] # 'val' will likely be
'value_1' here
print(f"Dirty read value in tool: {val}")

# Assume the event carrying the state_delta={'field_1':
'value_1'}
# is yielded *after* this tool runs and is processed by the
Runner.
```

- **Implications:**
- **Benefit:** Allows different parts of your logic within a single complex step (e.g., multiple callbacks or tool calls before the next LLM turn) to coordinate using state without waiting for a full yield/commit cycle.
- **Caveat:** Relying heavily on dirty reads for critical logic can be risky. If the invocation fails *before* the event carrying the `state_delta` is yielded and processed by the `Runner`, the uncommitted state change will be lost. For critical state transitions, ensure they are associated with an event that gets successfully processed.

Streaming vs. Non-Streaming Output (`partial=True`)

This primarily relates to how responses from the LLM are handled, especially when using streaming generation APIs.

- **Streaming:** The LLM generates its response token-by-token or in small chunks.
- The framework (often within `BaseLlmFlow`) yields multiple `Event` objects for a single conceptual response. Most of these events will have `partial=True`.
- The `Runner`, upon receiving an event with `partial=True`, typically **forwards it immediately** upstream (for UI display) but **skips processing its actions** (like `state_delta`).
- Eventually, the framework yields a final event for that response, marked as non-partial (`partial=False` or implicitly via `turn_complete=True`).
- The `Runner` **fully processes only this final event**, committing any associated `state_delta` or `artifact_delta`.
- **Non-Streaming:** The LLM generates the entire response at once. The framework yields a single event marked as non-partial, which the `Runner` processes fully.
- **Why it Matters:** Ensures that state changes are applied atomically and only once based on the *complete* response from the LLM, while still allowing the UI to display text progressively as it's generated.

Async is Primary (`run_async`)

- **Core Design:** The ADK Runtime is fundamentally built on Python's `asyncio` library to handle concurrent operations (like waiting for LLM responses or tool executions) efficiently without blocking.
- **Main Entry Point:** `Runner.run_async` is the primary method for executing agent invocations. All core runnable components (Agents, specific flows) use `async def` methods internally.
- **Synchronous Convenience (`run`):** A synchronous `Runner.run` method exists mainly for convenience (e.g., in simple scripts or testing environments). However, internally, `Runner.run` typically just calls `Runner.run_async` and manages the async event loop execution for you.

- **Developer Experience:** You should generally design your application logic (e.g., web servers using ADK) using `asyncio`.
- **Sync Callbacks/Tools:** The framework aims to handle both `async def` and regular `def` functions provided as tools or callbacks seamlessly. Long-running synchronous tools or callbacks, especially those performing blocking I/O, can potentially block the main `asyncio` event loop. The framework might use mechanisms like `asyncio.to_thread` to mitigate this by running such blocking synchronous code in a separate thread pool, preventing it from stalling other asynchronous tasks. CPU-bound synchronous code, however, will still block the thread it runs on.

Understanding these behaviors helps you write more robust ADK applications and debug issues related to state consistency, streaming updates, and asynchronous execution.