Custom agents - Agent Development Kit

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

Advanced Concept

Building custom agents by directly implementing `_run_async_impl` (or its equivalent in other languages) provides powerful control but is more complex than using the predefined `LlmAgent` or standard `WorkflowAgent` types. We recommend understanding those foundational agent types first before tackling custom orchestration logic.

Custom agents¶

Custom agents provide the ultimate flexibility in ADK, allowing you to define **arbitrary orchestration logic** by inheriting directly from `BaseAgent` and implementing your own control flow. This goes beyond the predefined patterns of `SequentialAgent`, `LoopAgent`, and `ParallelAgent`, enabling you to build highly specific and complex agentic workflows.

Introduction: Beyond Predefined Workflows¶

What is a Custom Agent?¶

A Custom Agent is essentially any class you create that inherits from `google.adk.agents.BaseAgent` and implements its core execution logic within the `_run_async_impl` asynchronous method. You have complete control over how this method calls other agents (sub-agents), manages state, and handles events.

Note

The specific method name for implementing an agent's core asynchronous logic may vary slightly by SDK language (e.g., `runAsyncImpl` in Java, `_run_async_impl` in Python). Refer to the language-specific API documentation for details.

Why Use Them?¶

While the standard Workflow Agents (`SequentialAgent`, `LoopAgent`, `ParallelAgent`) cover common orchestration patterns, you'll need a Custom agent when your requirements include:

- **Conditional Logic:** Executing different sub-agents or taking different paths based on runtime conditions or the results of previous steps.
- **Complex State Management:** Implementing intricate logic for maintaining and updating state throughout the workflow beyond simple sequential passing.
- **External Integrations:** Incorporating calls to external APIs, databases, or custom libraries directly within the orchestration flow control.
- **Dynamic Agent Selection:** Choosing which sub-agent(s) to run next based on dynamic evaluation of the situation or input.
- **Unique Workflow Patterns:** Implementing orchestration logic that doesn't fit the standard sequential, parallel, or loop structures.

intro_components.png

Implementing Custom Logic:¶

The core of any custom agent is the method where you define its unique asynchronous behavior. This method allows you to orchestrate sub-agents and manage the flow of execution.

PythonJava

The heart of any custom agent is the `_run_async_impl` method. This is where you define its unique behavior.

- **Signature:** `async def _run_async_impl(self, ctx: InvocationContext) -> AsyncGenerator[Event, None]:`
- **Asynchronous Generator:** It must be an `async def` function and return an `AsyncGenerator`. This allows it to `yield` events produced by sub-agents or its own logic back to the runner.
- **`ctx` (InvocationContext):** Provides access to crucial runtime information, most importantly `ctx.session.state`, which is the primary way to share data between steps orchestrated by your custom agent.

The heart of any custom agent is the `runAsyncImpl` method, which you override from `BaseAgent`.

- **Signature:** `protected Flowable<Event> runAsyncImpl(InvocationContext ctx)`
- **Reactive Stream (`Flowable`):** It must return an `io.reactivex.rxjava3.core.Flowable<Event>`. This `Flowable` represents a stream of events that will be produced by the custom agent's logic, often by combining or transforming multiple `Flowable` from sub-agents.
- **`ctx` (InvocationContext):** Provides access to crucial runtime information, most importantly `ctx.session().state()`, which is a `java.util.concurrent.ConcurrentMap<String, Object>`. This is the primary way to share data between steps orchestrated by your custom agent.

Key Capabilities within the Core Asynchronous Method:

PythonJava

1. **Calling Sub-Agents:** You invoke sub-agents (which are typically stored as instance attributes like `self.my_llm_agent`) using their `run_async` method and yield their events:

``` async for event in self.some_sub_agent.run_async(ctx): # Optionally inspect or log the event yield event # Pass the event up

`` 2. **Managing State:** Read from and write to the session state dictionary (ctx.session.state`) to pass data between sub-agent calls or make decisions:

``` # Read data set by a previous agent previous_result = ctx.session.state.get("some_key")

# Make a decision based on state if previous_result == "some_value": # ... call a specific sub-agent ... else: # ... call another sub-agent ...

# Store a result for a later step (often done via a sub-agent's output_key) # ctx.session.state["my_custom_result"] = "calculated_value"

`` 3. **Implementing Control Flow:** Use standard Python constructs (if / elif / else , for / while loops, try / except`) to create sophisticated, conditional, or iterative workflows involving your sub-agents.

1. **Calling Sub-Agents:** You invoke sub-agents (which are typically stored as instance attributes or objects) using their asynchronous run method and return their event streams:

You typically chain `Flowable`s from sub-agents using RxJava operators like `concatWith`, `flatMapPublisher`, or `concatArray`.

``` // Example: Running one sub-agent // return someSubAgent.runAsync(ctx);

// Example: Running sub-agents sequentially Flowable firstAgentEvents = someSubAgent1.runAsync(ctx) .doOnNext(event -> System.out.println("Event from agent 1: " + event.id()));

Flowable secondAgentEvents = Flowable.defer(() -> someSubAgent2.runAsync(ctx) .doOnNext(event -> System.out.println("Event from agent 2: " + event.id())) );

return firstAgentEvents.concatWith(secondAgentEvents);

```

The `Flowable.defer()` is often used for subsequent stages if their execution depends on the completion or state after prior stages. 2. **Managing State:** Read from and write to the session state to pass data between sub-agent calls or make decisions. The session state is a `java.util.concurrent.ConcurrentMap<String, Object>` obtained via `ctx.session().state()`.

``` // Read data set by a previous agent Object previousResult = ctx.session().state().get("some_key");

// Make a decision based on state if ("some_value".equals(previousResult)) { // ... logic to include a specific sub-agent's Flowable ... } else { // ... logic to include another sub-agent's Flowable ... }

// Store a result for a later step (often done via a sub-agent's output_key) // ctx.session().state().put("my_custom_result", "calculated_value");
```

``` `` 3. **Implementing Control Flow:** Use standard language constructs (``` if `/` else `, loops,` try `/` catch`) combined with reactive operators (RxJava) to create sophisticated workflows.

- **Conditional:** `Flowable.defer()` to choose which `Flowable` to subscribe to based on a condition, or `filter()` if you're filtering events within a stream.
- **Iterative:** Operators like `repeat()`, `retry()`, or by structuring your `Flowable` chain to recursively call parts of itself based on conditions (often managed with `flatMapPublisher` or `concatMap`).

Managing Sub-Agents and State¶

Typically, a custom agent orchestrates other agents (like `LlmAgent`, `LoopAgent`, etc.).

- **Initialization:** You usually pass instances of these sub-agents into your custom agent's constructor and store them as instance fields/attributes (e.g., `this.story_generator = story_generator_instance` or `self.story_generator = story_generator_instance`). This makes them accessible within the custom agent's core asynchronous execution logic (such as: `_run_async_impl` method).
- **Sub Agents List:** When initializing the `BaseAgent` using it's `super()` constructor, you should pass a `sub_agents` list. This list tells the ADK framework about the agents that are part of this custom agent's immediate hierarchy. It's important for framework features like lifecycle management, introspection, and potentially future routing capabilities, even if your core execution logic (`_run_async_impl`) calls the agents directly via `self.xxx_agent`. Include the agents that your custom logic directly invokes at the top level.
- **State:** As mentioned, `ctx.session.state` is the standard way sub-agents (especially `LlmAgent`s using `output_key`) communicate results back to the orchestrator and how the orchestrator passes necessary inputs down.
```

Design Pattern Example: `StoryFlowAgent`¶

Let's illustrate the power of custom agents with an example pattern: a multi-stage content generation workflow with conditional logic.

Goal: Create a system that generates a story, iteratively refines it through critique and revision, performs final checks, and crucially, *regenerates the story if the final tone check fails*.

Why Custom? The core requirement driving the need for a custom agent here is the **conditional regeneration based on the tone check**. Standard workflow agents don't have built-in conditional branching based on the outcome of a sub-agent's task. We need custom logic (`if tone == "negative": ...`) within the orchestrator.

Part 1: Simplified custom agent Initialization¶

PythonJava

We define the `StoryFlowAgent` inheriting from `BaseAgent`. In `__init__`, we store the necessary sub-agents (passed in) as instance attributes and tell the `BaseAgent` framework about the top-level agents this custom agent will directly orchestrate.

```
class StoryFlowAgent(BaseAgent):
    """
    Custom agent for a story generation and refinement workflow.

    This agent orchestrates a sequence of LLM agents to generate a sto
    critique it, revise it, check grammar and tone, and potentially
    regenerate the story if the tone is negative.
    """

    # --- Field Declarations for Pydantic ---
    # Declare the agents passed during initialization as class attribu
    story_generator: LlmAgent
    critic: LlmAgent
    reviser: LlmAgent
    grammar_check: LlmAgent
    tone_check: LlmAgent

    loop_agent: LoopAgent
    sequential_agent: SequentialAgent

    # model_config allows setting Pydantic configurations if needed, e
    model_config = {"arbitrary_types_allowed": True}

    def __init__(
        self,
        name: str,
        story_generator: LlmAgent,
        critic: LlmAgent,
        reviser: LlmAgent,
        grammar_check: LlmAgent,
        tone_check: LlmAgent,
    ):
        """
        Initializes the StoryFlowAgent.

        Args:
            name: The name of the agent.
            story_generator: An LlmAgent to generate the initial story
            critic: An LlmAgent to critique the story.
            reviser: An LlmAgent to revise the story based on criticis
            grammar_check: An LlmAgent to check the grammar.
            tone_check: An LlmAgent to analyze the tone.
        """
        # Create internal agents *before* calling super().__init__
        loop_agent = LoopAgent(
            name="CriticReviserLoop", sub_agents=[critic, reviser], ma
        )
        sequential_agent = SequentialAgent(
            name="PostProcessing", sub_agents=[grammar_check, tone_che
        )
        # Define the sub_agents list for the framework
        sub_agents_list = [
            story_generator,
            loop_agent,
            sequential_agent,
        ]

        # Pydantic will validate and assign them based on the class ar
        super().__init__(
            name=name,
            story_generator=story_generator,
            critic=critic,
            reviser=reviser,
            grammar_check=grammar_check,
            tone_check=tone_check,
            loop_agent=loop_agent,
            sequential_agent=sequential_agent,
            sub_agents=sub_agents_list, # Pass the sub_agents list dir
        )
```

We define the `StoryFlowAgentExample` by extending `BaseAgent`. In its **constructor**, we store the necessary sub-agent instances (passed as parameters) as instance fields. These top-level sub-agents, which this custom agent will directly orchestrate, are also passed to the `super` constructor of `BaseAgent` as a list.

```
 private final LlmAgent storyGenerator;
private final LoopAgent loopAgent;
private final SequentialAgent sequentialAgent;

public StoryFlowAgentExample(
    String name, LlmAgent storyGenerator, LoopAgent loopAgent, Sequent
  super(
      name,
      "Orchestrates story generation, critique, revision, and checks."
      List.of(storyGenerator, loopAgent, sequentialAgent),
      null,
      null);

  this.storyGenerator = storyGenerator;
  this.loopAgent = loopAgent;
  this.sequentialAgent = sequentialAgent;
}
```

Part 2: Defining the Custom Execution Logic¶

PythonJava

This method orchestrates the sub-agents using standard Python async/await and control flow.

```
@override
async def _run_async_impl(
    self, ctx: InvocationContext
) -> AsyncGenerator[Event, None]:
    """
    Implements the custom orchestration logic for the story workflow.
    Uses the instance attributes assigned by Pydantic (e.g., self.stor
    """
    logger.info(f"[{self.name}] Starting story generation workflow.")

    # 1. Initial Story Generation
    logger.info(f"[{self.name}] Running StoryGenerator...")
    async for event in self.story_generator.run_async(ctx):
        logger.info(f"[{self.name}] Event from StoryGenerator: {event.
        yield event

    # Check if story was generated before proceeding
    if "current_story" not in ctx.session.state or not ctx.session.sta
```

```
     logger.error(f"[{self.name}] Failed to generate initial story
     return # Stop processing if initial story failed

logger.info(f"[{self.name}] Story state after generator: {ctx.sess

# 2. Critic-Reviser Loop
logger.info(f"[{self.name}] Running CriticReviserLoop...")
# Use the loop_agent instance attribute assigned during init
async for event in self.loop_agent.run_async(ctx):
    logger.info(f"[{self.name}] Event from CriticReviserLoop: {eve
    yield event

logger.info(f"[{self.name}] Story state after loop: {ctx.session.s

# 3. Sequential Post-Processing (Grammar and Tone Check)
logger.info(f"[{self.name}] Running PostProcessing...")
# Use the sequential_agent instance attribute assigned during init
async for event in self.sequential_agent.run_async(ctx):
    logger.info(f"[{self.name}] Event from PostProcessing: {event.
    yield event

# 4. Tone-Based Conditional Logic
tone_check_result = ctx.session.state.get("tone_check_result")
logger.info(f"[{self.name}] Tone check result: {tone_check_result}

if tone_check_result == "negative":
    logger.info(f"[{self.name}] Tone is negative. Regenerating sto
    async for event in self.story_generator.run_async(ctx):
        logger.info(f"[{self.name}] Event from StoryGenerator (Reg
        yield event
else:
    logger.info(f"[{self.name}] Tone is not negative. Keeping curr
    pass

logger.info(f"[{self.name}] Workflow finished.")
```

Explanation of Logic:

1. The initial `story_generator` runs. Its output is expected to be in `ctx.session.state["current_story"]`.
2. The `loop_agent` runs, which internally calls the `critic` and `reviser` sequentially for `max_iterations` times. They read/write `current_story` and `criticism` from/to the state.
3. The `sequential_agent` runs, calling `grammar_check` then `tone_check`, reading `current_story` and writing `grammar_suggestions` and `tone_check_result` to the state.
4. **Custom Part:** The `if` statement checks the `tone_check_result` from the state. If it's "negative", the `story_generator` is called *again*, overwriting the `current_story` in the state. Otherwise, the flow ends.

The `runAsyncImpl` method orchestrates the sub-agents using RxJava's Flowable streams and operators for asynchronous control flow.

```
@Override
protected Flowable<Event> runAsyncImpl(InvocationContext invocationCor
  // Implements the custom orchestration logic for the story workflow.
  // Uses the instance attributes assigned by Pydantic (e.g., self.sto
  logger.log(Level.INFO, () -> String.format("[%s] Starting story gene

  // Stage 1. Initial Story Generation
  Flowable<Event> storyGenFlow = runStage(storyGenerator, invocationCo

  // Stage 2: Critic-Reviser Loop (runs after story generation complet
  Flowable<Event> criticReviserFlow = Flowable.defer(() -> {
    if (!isStoryGenerated(invocationContext)) {
      logger.log(Level.SEVERE,() ->
          String.format("[%s] Failed to generate initial story. Aborti
              name()));
      return Flowable.empty(); // Stop further processing if no story
    }
      logger.log(Level.INFO, () ->
          String.format("[%s] Story state after generator: %s",
              name(), invocationContext.session().state().get("current
      return runStage(loopAgent, invocationContext, "CriticReviserLoop
  });

  // Stage 3: Post-Processing (runs after critic-reviser loop complete
  Flowable<Event> postProcessingFlow = Flowable.defer(() -> {
    logger.log(Level.INFO, () ->
        String.format("[%s] Story state after loop: %s",
            name(), invocationContext.session().state().get("current_s
    return runStage(sequentialAgent, invocationContext, "PostProcessin
  });

  // Stage 4: Conditional Regeneration (runs after post-processing com
  Flowable<Event> conditionalRegenFlow = Flowable.defer(() -> {
    String toneCheckResult = (String) invocationContext.session().stat
    logger.log(Level.INFO, () -> String.format("[%s] Tone check result

    if ("negative".equalsIgnoreCase(toneCheckResult)) {
      logger.log(Level.INFO, () ->
          String.format("[%s] Tone is negative. Regenerating story..."
      return runStage(storyGenerator, invocationContext, "StoryGenerat
    } else {
      logger.log(Level.INFO, () ->
          String.format("[%s] Tone is not negative. Keeping current st
      return Flowable.empty(); // No regeneration needed
    }
  });

  return Flowable.concatArray(storyGenFlow, criticReviserFlow, postPro
      .doOnComplete(() -> logger.log(Level.INFO, () -> String.format("
}

// Helper method for a single agent run stage with logging
private Flowable<Event> runStage(BaseAgent agentToRun, InvocationConte
  logger.log(Level.INFO, () -> String.format("[%s] Running %s...", nam
  return agentToRun
      .runAsync(ctx)
      .doOnNext(event ->
          logger.log(Level.INFO,() ->
              String.format("[%s] Event from %s: %s", name(), stageNam
      .doOnError(err ->
          logger.log(Level.SEVERE,
              String.format("[%s] Error in %s", name(), stageName), er
      .doOnComplete(() ->
          logger.log(Level.INFO, () ->
              String.format("[%s] %s finished.", name(), stageName)));
}
```

Explanation of Logic:

1. The initial `storyGenerator.runAsync(invocationContext)` Flowable is executed. Its output is expected to be in `invocationContext.session().state().get("current_story")`.
2. The `loopAgent's` Flowable runs next (due to `Flowable.concatArray` and `Flowable.defer`). The LoopAgent internally calls the `critic` and `reviser` sub-agents sequentially for up to `maxIterations`. They read/write `current_story` and `criticism` from/to the state.
3. Then, the `sequentialAgent's` Flowable executes. It calls the `grammar_check` then `tone_check`, reading `current_story` and writing `grammar_suggestions` and `tone_check_result` to the state.
4. **Custom Part:** After the sequentialAgent completes, logic within a `Flowable.defer` checks the "tone_check_result" from `invocationContext.session().state()`. If it's "negative", the `storyGenerator` Flowable is *conditionally concatenated* and executed again, overwriting "current_story". Otherwise, an empty Flowable is used, and the overall workflow proceeds to completion.

Part 3: Defining the LLM Sub-Agents¶

These are standard `LlmAgent` definitions, responsible for specific tasks. Their `output key` parameter is crucial for placing results into the `session.state` where other agents or the custom orchestrator can access them.

PythonJava

```
GEMINI_2_FLASH = "gemini-2.0-flash" # Define model constant
# --- Define the individual LLM agents ---
story_generator = LlmAgent(
    name="StoryGenerator",
    model=GEMINI_2_FLASH,
    instruction="""You are a story writer. Write a short story (around
based on the topic provided in session state with key 'topic'""",
    input_schema=None,
    output_key="current_story",  # Key for storing output in session s
)

critic = LlmAgent(
    name="Critic",
    model=GEMINI_2_FLASH,
    instruction="""You are a story critic. Review the story provided i
session state with key 'current_story'. Provide 1-2 sentences of const
on how to improve it. Focus on plot or character.""",
    input_schema=None,
    output_key="criticism",  # Key for storing criticism in session st
)

reviser = LlmAgent(
    name="Reviser",
    model=GEMINI_2_FLASH,
    instruction="""You are a story reviser. Revise the story provided
session state with key 'current_story', based on the criticism in
session state with key 'criticism'. Output only the revised story.""",
    input_schema=None,
    output_key="current_story",  # Overwrites the original story
)

grammar_check = LlmAgent(
    name="GrammarCheck",
    model=GEMINI_2_FLASH,
    instruction="""You are a grammar checker. Check the grammar of the
provided in session state with key 'current_story'. Output only the su
corrections as a list, or output 'Grammar is good!' if there are no er
    input_schema=None,
    output_key="grammar_suggestions",
)

tone_check = LlmAgent(
    name="ToneCheck",
    model=GEMINI_2_FLASH,
    instruction="""You are a tone analyzer. Analyze the tone of the st
provided in session state with key 'current_story'. Output only one wo
the tone is generally positive, 'negative' if the tone is generally ne
otherwise.""",
    input_schema=None,
    output_key="tone_check_result", # This agent's output determines t
)
```

```
// --- Define the individual LLM agents ---
LlmAgent storyGenerator =
    LlmAgent.builder()
        .name("StoryGenerator")
        .model(MODEL_NAME)
        .description("Generates the initial story.")
        .instruction(
            """
          You are a story writer. Write a short story (around 100 word
          based on the topic provided in session state with key 'topic
          """)
        .inputSchema(null)
        .outputKey("current_story") // Key for storing output in sessi
        .build();

LlmAgent critic =
    LlmAgent.builder()
        .name("Critic")
        .model(MODEL_NAME)
        .description("Critiques the story.")
        .instruction(
            """
          You are a story critic. Review the story provided in
          session state with key 'current_story'. Provide 1-2 sentence
          on how to improve it. Focus on plot or character.
          """)
        .inputSchema(null)
        .outputKey("criticism") // Key for storing criticism in sessio
        .build();

LlmAgent reviser =
    LlmAgent.builder()
        .name("Reviser")
        .model(MODEL_NAME)
        .description("Revises the story based on criticism.")
        .instruction(
            """
          You are a story reviser. Revise the story provided in
          session state with key 'current_story', based on the critici
          session state with key 'criticism'. Output only the revised
          """)
        .inputSchema(null)
        .outputKey("current_story") // Overwrites the original story
        .build();

LlmAgent grammarCheck =
    LlmAgent.builder()
        .name("GrammarCheck")
        .model(MODEL_NAME)
        .description("Checks grammar and suggests corrections.")
        .instruction(
            """
           You are a grammar checker. Check the grammar of the story
           provided in session state with key 'current_story'. Output
           corrections as a list, or output 'Grammar is good!' if ther
           """)
        .outputKey("grammar_suggestions")
        .build();

LlmAgent toneCheck =
    LlmAgent.builder()
        .name("ToneCheck")
        .model(MODEL_NAME)
        .description("Analyzes the tone of the story.")
        .instruction(
            """
          You are a tone analyzer. Analyze the tone of the story
          provided in session state with key 'current_story'. Output c
          the tone is generally positive, 'negative' if the tone is ge
          otherwise.
          """)
        .outputKey("tone_check_result") // This agent's output determi
        .build();

LoopAgent loopAgent =
    LoopAgent.builder()
        .name("CriticReviserLoop")
        .description("Iteratively critiques and revises the story.")
        .subAgents(critic, reviser)
        .maxIterations(2)
        .build();
```

```
SequentialAgent sequentialAgent =
    SequentialAgent.builder()
        .name("PostProcessing")
        .description("Performs grammar and tone checks sequentially.")
        .subAgents(grammarCheck, toneCheck)
        .build();
```

Part 4: Instantiating and Running the custom agent¶

Finally, you instantiate your `StoryFlowAgent` and use the `Runner` as usual.

PythonJava

```
# --- Create the custom agent instance ---
story_flow_agent = StoryFlowAgent(
    name="StoryFlowAgent",
    story_generator=story_generator,
    critic=critic,
    reviser=reviser,
    grammar_check=grammar_check,
    tone_check=tone_check,
)

# --- Setup Runner and Session ---
session_service = InMemorySessionService()
initial_state = {"topic": "a brave kitten exploring a haunted house"}
session = session_service.create_session(
    app_name=APP_NAME,
    user_id=USER_ID,
    session_id=SESSION_ID,
    state=initial_state # Pass initial state here
)
logger.info(f"Initial session state: {session.state}")
```

```
runner = Runner(
    agent=story_flow_agent, # Pass the custom orchestrator agent
    app_name=APP_NAME,
    session_service=session_service
)

# --- Function to Interact with the Agent ---
def call_agent(user_input_topic: str):
    """
    Sends a new topic to the agent (overwriting the initial one if nee
    and runs the workflow.
    """
    current_session = session_service.get_session(app_name=APP_NAME,
                                                  user_id=USER_ID,
                                                  session_id=SESSION_I
    if not current_session:
        logger.error("Session not found!")
        return

    current_session.state["topic"] = user_input_topic
    logger.info(f"Updated session state topic to: {user_input_topic}")

    content = types.Content(role='user', parts=[types.Part(text=f"Gene
    events = runner.run(user_id=USER_ID, session_id=SESSION_ID, new_me

    final_response = "No final response captured."
    for event in events:
        if event.is_final_response() and event.content and event.conte
            logger.info(f"Potential final response from [{event.author
            final_response = event.content.parts[0].text

    print("\n--- Agent Interaction Result ---")
    print("Agent Final Response: ", final_response)

    final_session = session_service.get_session(app_name=APP_NAME,
                                                user_id=USER_ID,
                                                session_id=SESSION_ID)
    print("Final Session State:")
    import json
    print(json.dumps(final_session.state, indent=2))
    print("-------------------------------\n")

# --- Run the Agent ---
call_agent("a lonely robot finding a friend in a junkyard")
```

```
// --- Function to Interact with the Agent ---
// Sends a new topic to the agent (overwriting the initial one if need
// and runs the workflow.
public static void runAgent(StoryFlowAgentExample agent, String userTo
  // --- Setup Runner and Session ---
  InMemoryRunner runner = new InMemoryRunner(agent);

  Map<String, Object> initialState = new HashMap<>();
  initialState.put("topic", "a brave kitten exploring a haunted house"

  Session session =
      runner
          .sessionService()
          .createSession(APP_NAME, USER_ID, new ConcurrentHashMap<>(in
          .blockingGet();
  logger.log(Level.INFO, () -> String.format("Initial session state: %

  session.state().put("topic", userTopic); // Update the state in the
  logger.log(Level.INFO, () -> String.format("Updated session state to

  Content userMessage = Content.fromParts(Part.fromText("Generate a st
  // Use the modified session object for the run
  Flowable<Event> eventStream = runner.runAsync(USER_ID, session.id(),

  final String[] finalResponse = {"No final response captured."};
  eventStream.blockingForEach(
      event -> {
        if (event.finalResponse() && event.content().isPresent()) {
          String author = event.author() != null ? event.author() : "U
          Optional<String> textOpt =
              event
                  .content()
                  .flatMap(Content::parts)
                  .filter(parts -> !parts.isEmpty())
                  .map(parts -> parts.get(0).text().orElse(""));

          logger.log(Level.INFO, () ->
              String.format("Potential final response from [%s]: %s",
          textOpt.ifPresent(text -> finalResponse[0] = text);
        }
      });

  System.out.println("\n--- Agent Interaction Result ---");
  System.out.println("Agent Final Response: " + finalResponse[0]);

  // Retrieve session again to see the final state after the run
  Session finalSession =
      runner
          .sessionService()
          .getSession(APP_NAME, USER_ID, SESSION_ID, Optional.empty())
          .blockingGet();

  assert finalSession != null;
  System.out.println("Final Session State:" + finalSession.state());
  System.out.println("-------------------------------\n");
}
```

(Note: The full runnable code, including imports and execution logic, can be found linked below.)

Full Code Example¶

Storyflow Agent

PythonJava

```
 # Full runnable code for the StoryFlowAgent example
import logging
from typing import AsyncGenerator
from typing_extensions import override

from google.adk.agents import LlmAgent, BaseAgent, LoopAgent, Sequenti
from google.adk.agents.invocation_context import InvocationContext
from google.genai import types
from google.adk.sessions import InMemorySessionService
from google.adk.runners import Runner
from google.adk.events import Event
from pydantic import BaseModel, Field

# --- Constants ---
APP_NAME = "story_app"
USER_ID = "12345"
SESSION_ID = "123344"
GEMINI_2_FLASH = "gemini-2.0-flash"

# --- Configure Logging ---
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

# --- Custom Orchestrator Agent ---
class StoryFlowAgent(BaseAgent):
    """
    Custom agent for a story generation and refinement workflow.

    This agent orchestrates a sequence of LLM agents to generate a sto
    critique it, revise it, check grammar and tone, and potentially
    regenerate the story if the tone is negative.
    """

    # --- Field Declarations for Pydantic ---
    # Declare the agents passed during initialization as class attribu
    story_generator: LlmAgent
    critic: LlmAgent
    reviser: LlmAgent
    grammar_check: LlmAgent
    tone_check: LlmAgent

    loop_agent: LoopAgent
    sequential_agent: SequentialAgent

    # model_config allows setting Pydantic configurations if needed, e
    model_config = {"arbitrary_types_allowed": True}

    def __init__(
        self,
        name: str,
        story_generator: LlmAgent,
        critic: LlmAgent,
        reviser: LlmAgent,
        grammar_check: LlmAgent,
        tone_check: LlmAgent,
    ):
        """
        Initializes the StoryFlowAgent.

        Args:
            name: The name of the agent.
            story_generator: An LlmAgent to generate the initial story
            critic: An LlmAgent to critique the story.
            reviser: An LlmAgent to revise the story based on criticis
            grammar_check: An LlmAgent to check the grammar.
            tone_check: An LlmAgent to analyze the tone.
        """
        # Create internal agents *before* calling super().__init__
        loop_agent = LoopAgent(
            name="CriticReviserLoop", sub_agents=[critic, reviser], ma
        )
        sequential_agent = SequentialAgent(
            name="PostProcessing", sub_agents=[grammar_check, tone_che
        )

        # Define the sub_agents list for the framework
        sub_agents_list = [
            story_generator,
            loop_agent,
            sequential_agent,
        ]

        # Pydantic will validate and assign them based on the class an
        super().__init__(
            name=name,
            story_generator=story_generator,
            critic=critic,
            reviser=reviser,
            grammar_check=grammar_check,
            tone_check=tone_check,
            loop_agent=loop_agent,
            sequential_agent=sequential_agent,
            sub_agents=sub_agents_list, # Pass the sub_agents list dir
        )

    @override
    async def _run_async_impl(
        self, ctx: InvocationContext
    ) -> AsyncGenerator[Event, None]:
        """
        Implements the custom orchestration logic for the story workfl
        Uses the instance attributes assigned by Pydantic (e.g., self.
        """
        logger.info(f"[{self.name}] Starting story generation workflow

        # 1. Initial Story Generation
        logger.info(f"[{self.name}] Running StoryGenerator...")
        async for event in self.story_generator.run_async(ctx):
            logger.info(f"[{self.name}] Event from StoryGenerator: {ev
            yield event

        # Check if story was generated before proceeding
        if "current_story" not in ctx.session.state or not ctx.session
             logger.error(f"[{self.name}] Failed to generate initial s
             return # Stop processing if initial story failed

        logger.info(f"[{self.name}] Story state after generator: {ctx.

        # 2. Critic-Reviser Loop
        logger.info(f"[{self.name}] Running CriticReviserLoop...")
        # Use the loop_agent instance attribute assigned during init
        async for event in self.loop_agent.run_async(ctx):
            logger.info(f"[{self.name}] Event from CriticReviserLoop:
            yield event

        logger.info(f"[{self.name}] Story state after loop: {ctx.sessi

        # 3. Sequential Post-Processing (Grammar and Tone Check)
        logger.info(f"[{self.name}] Running PostProcessing...")
        # Use the sequential_agent instance attribute assigned during
        async for event in self.sequential_agent.run_async(ctx):
            logger.info(f"[{self.name}] Event from PostProcessing: {ev
            yield event

        # 4. Tone-Based Conditional Logic
        tone_check_result = ctx.session.state.get("tone_check_result")
        logger.info(f"[{self.name}] Tone check result: {tone_check_res
        if tone_check_result == "negative":
            logger.info(f"[{self.name}] Tone is negative. Regenerating
            async for event in self.story_generator.run_async(ctx):
                logger.info(f"[{self.name}] Event from StoryGenerator
                yield event
        else:
            logger.info(f"[{self.name}] Tone is not negative. Keeping
            pass

        logger.info(f"[{self.name}] Workflow finished.")

# --- Define the individual LLM agents ---
story_generator = LlmAgent(
    name="StoryGenerator",
    model=GEMINI_2_FLASH,
    instruction="""You are a story writer. Write a short story (arounc
based on the topic provided in session state with key 'topic'""",
    input_schema=None,
    output_key="current_story",  # Key for storing output in session s
)

critic = LlmAgent(
    name="Critic",
    model=GEMINI_2_FLASH,
    instruction="""You are a story critic. Review the story provided i
session state with key 'current_story'. Provide 1-2 sentences of const
on how to improve it. Focus on plot or character.""",
    input_schema=None,
    output_key="criticism",  # Key for storing criticism in session st
)

reviser = LlmAgent(
    name="Reviser",
    model=GEMINI_2_FLASH,
    instruction="""You are a story reviser. Revise the story provided
session state with key 'current_story', based on the criticism in
session state with key 'criticism'. Output only the revised story.""",
    input_schema=None,
    output_key="current_story",  # Overwrites the original story
)

grammar_check = LlmAgent(
    name="GrammarCheck",
    model=GEMINI_2_FLASH,
    instruction="""You are a grammar checker. Check the grammar of the
provided in session state with key 'current_story'. Output only the su
corrections as a list, or output 'Grammar is good!' if there are no er
    input_schema=None,
    output_key="grammar_suggestions",
)

tone_check = LlmAgent(
    name="ToneCheck",
    model=GEMINI_2_FLASH,
    instruction="""You are a tone analyzer. Analyze the tone of the st
provided in session state with key 'current_story'. Output only one wo
the tone is generally positive, 'negative' if the tone is generally ne
otherwise.""",
    input_schema=None,
    output_key="tone_check_result", # This agent's output determines t
)

# --- Create the custom agent instance ---
story_flow_agent = StoryFlowAgent(
    name="StoryFlowAgent",
    story_generator=story_generator,
    critic=critic,
    reviser=reviser,
    grammar_check=grammar_check,
    tone_check=tone_check,
)
```

```
# --- Setup Runner and Session ---
session_service = InMemorySessionService()
initial_state = {"topic": "a brave kitten exploring a haunted house"}
session = session_service.create_session(
    app_name=APP_NAME,
    user_id=USER_ID,
    session_id=SESSION_ID,
    state=initial_state # Pass initial state here
)
logger.info(f"Initial session state: {session.state}")

runner = Runner(
    agent=story_flow_agent, # Pass the custom orchestrator agent
    app_name=APP_NAME,
    session_service=session_service
)

# --- Function to Interact with the Agent ---
def call_agent(user_input_topic: str):
    """
    Sends a new topic to the agent (overwriting the initial one if nee
    and runs the workflow.
    """
    current_session = session_service.get_session(app_name=APP_NAME,
                                                  user_id=USER_ID,
                                                  session_id=SESSION_I
    if not current_session:
        logger.error("Session not found!")
        return

    current_session.state["topic"] = user_input_topic
    logger.info(f"Updated session state topic to: {user_input_topic}")

    content = types.Content(role='user', parts=[types.Part(text=f"Gene
    events = runner.run(user_id=USER_ID, session_id=SESSION_ID, new_me
    final_response = "No final response captured."
    for event in events:
        if event.is_final_response() and event.content and event.conte
            logger.info(f"Potential final response from [{event.author
            final_response = event.content.parts[0].text

    print("\n--- Agent Interaction Result ---")
    print("Agent Final Response: ", final_response)

    final_session = session_service.get_session(app_name=APP_NAME,
                                                user_id=USER_ID,
                                                session_id=SESSION_ID)
    print("Final Session State:")
    import json
    print(json.dumps(final_session.state, indent=2))
    print("-------------------------------\n")

# --- Run the Agent ---
call_agent("a lonely robot finding a friend in a junkyard")
```

```
# Full runnable code for the StoryFlowAgent example

import com.google.adk.agents.LlmAgent;
import com.google.adk.agents.BaseAgent;
import com.google.adk.agents.InvocationContext;
import com.google.adk.agents.LoopAgent;
import com.google.adk.agents.SequentialAgent;
import com.google.adk.events.Event;
import com.google.adk.runner.InMemoryRunner;
import com.google.adk.sessions.Session;
import com.google.genai.types.Content;
import com.google.genai.types.Part;
import io.reactivex.rxjava3.core.Flowable;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.Optional;
import java.util.concurrent.ConcurrentHashMap;
import java.util.logging.Level;
import java.util.logging.Logger;

public class StoryFlowAgentExample extends BaseAgent {

  // --- Constants ---
  private static final String APP_NAME = "story_app";
  private static final String USER_ID = "user_12345";
  private static final String SESSION_ID = "session_123344";
  private static final String MODEL_NAME = "gemini-2.0-flash"; // Ensu

  private static final Logger logger = Logger.getLogger(StoryFlowAgent

  private final LlmAgent storyGenerator;
  private final LoopAgent loopAgent;
  private final SequentialAgent sequentialAgent;

  public StoryFlowAgentExample(
      String name, LlmAgent storyGenerator, LoopAgent loopAgent, Seque
    super(
        name,
        "Orchestrates story generation, critique, revision, and checks
        List.of(storyGenerator, loopAgent, sequentialAgent),
        null,
        null);

    this.storyGenerator = storyGenerator;
    this.loopAgent = loopAgent;
    this.sequentialAgent = sequentialAgent;
  }

  public static void main(String[] args) {
```

```
// --- Define the individual LLM agents ---
LlmAgent storyGenerator =
    LlmAgent.builder()
        .name("StoryGenerator")
        .model(MODEL_NAME)
        .description("Generates the initial story.")
        .instruction(
            """
          You are a story writer. Write a short story (around 100
          based on the topic provided in session state with key 't
          """)
        .inputSchema(null)
        .outputKey("current_story") // Key for storing output in s
        .build();

LlmAgent critic =
    LlmAgent.builder()
        .name("Critic")
        .model(MODEL_NAME)
        .description("Critiques the story.")
        .instruction(
            """
          You are a story critic. Review the story provided in
          session state with key 'current_story'. Provide 1-2 sent
          on how to improve it. Focus on plot or character.
          """)
        .inputSchema(null)
        .outputKey("criticism") // Key for storing criticism in se
        .build();

LlmAgent reviser =
    LlmAgent.builder()
        .name("Reviser")
        .model(MODEL_NAME)
        .description("Revises the story based on criticism.")
```

```
            .instruction(
                """
              You are a story reviser. Revise the story provided in
              session state with key 'current_story', based on the cri
              session state with key 'criticism'. Output only the revi
              """)
            .inputSchema(null)
            .outputKey("current_story") // Overwrites the original sto
            .build();

    LlmAgent grammarCheck =
        LlmAgent.builder()
            .name("GrammarCheck")
            .model(MODEL_NAME)
            .description("Checks grammar and suggests corrections.")
            .instruction(
                """
               You are a grammar checker. Check the grammar of the sto
               provided in session state with key 'current_story'. Out
               corrections as a list, or output 'Grammar is good!' if
               """)
            .outputKey("grammar_suggestions")
            .build();

    LlmAgent toneCheck =
        LlmAgent.builder()
            .name("ToneCheck")
            .model(MODEL_NAME)
            .description("Analyzes the tone of the story.")
            .instruction(
                """
              You are a tone analyzer. Analyze the tone of the story
              provided in session state with key 'current_story'. Outp
              the tone is generally positive, 'negative' if the tone i
              otherwise.
              """)
```

```
          .outputKey("tone_check_result") // This agent's output det
          .build();

  LoopAgent loopAgent =
      LoopAgent.builder()
          .name("CriticReviserLoop")
          .description("Iteratively critiques and revises the story.
          .subAgents(critic, reviser)
          .maxIterations(2)
          .build();

  SequentialAgent sequentialAgent =
      SequentialAgent.builder()
          .name("PostProcessing")
          .description("Performs grammar and tone checks sequentiall
          .subAgents(grammarCheck, toneCheck)
          .build();

  StoryFlowAgentExample storyFlowAgentExample =
      new StoryFlowAgentExample(APP_NAME, storyGenerator, loopAgent,

  // --- Run the Agent ---
  runAgent(storyFlowAgentExample, "a lonely robot finding a friend i
}

// --- Function to Interact with the Agent ---
// Sends a new topic to the agent (overwriting the initial one if ne
// and runs the workflow.
public static void runAgent(StoryFlowAgentExample agent, String user
  // --- Setup Runner and Session ---
  InMemoryRunner runner = new InMemoryRunner(agent);

  Map<String, Object> initialState = new HashMap<>();
  initialState.put("topic", "a brave kitten exploring a haunted hous

  Session session =
      runner
          .sessionService()
          .createSession(APP_NAME, USER_ID, new ConcurrentHashMap<>(
          .blockingGet();
  logger.log(Level.INFO, () -> String.format("Initial session state:

  session.state().put("topic", userTopic); // Update the state in th
  logger.log(Level.INFO, () -> String.format("Updated session state

  Content userMessage = Content.fromParts(Part.fromText("Generate a
  // Use the modified session object for the run
  Flowable<Event> eventStream = runner.runAsync(USER_ID, session.id(

  final String[] finalResponse = {"No final response captured."};
  eventStream.blockingForEach(
      event -> {
        if (event.finalResponse() && event.content().isPresent()) {
          String author = event.author() != null ? event.author() :
          Optional<String> textOpt =
              event
                  .content()
                  .flatMap(Content::parts)
                  .filter(parts -> !parts.isEmpty())
                  .map(parts -> parts.get(0).text().orElse(""));

          logger.log(Level.INFO, () ->
              String.format("Potential final response from [%s]: %s"
          textOpt.ifPresent(text -> finalResponse[0] = text);
        }
      });

  System.out.println("\n--- Agent Interaction Result ---");
  System.out.println("Agent Final Response: " + finalResponse[0]);

  // Retrieve session again to see the final state after the run
  Session finalSession =
      runner
          .sessionService()
          .getSession(APP_NAME, USER_ID, SESSION_ID, Optional.empty(
          .blockingGet();

  assert finalSession != null;
  System.out.println("Final Session State:" + finalSession.state());
  System.out.println("-------------------------------\n");
}

private boolean isStoryGenerated(InvocationContext ctx) {
  Object currentStoryObj = ctx.session().state().get("current_story"
  return currentStoryObj != null && !String.valueOf(currentStoryObj)
}

@Override
protected Flowable<Event> runAsyncImpl(InvocationContext invocationC
  // Implements the custom orchestration logic for the story workflo
  // Uses the instance attributes assigned by Pydantic (e.g., self.s
  logger.log(Level.INFO, () -> String.format("[%s] Starting story ge

  // Stage 1. Initial Story Generation
  Flowable<Event> storyGenFlow = runStage(storyGenerator, invocatior

  // Stage 2: Critic-Reviser Loop (runs after story generation compl
  Flowable<Event> criticReviserFlow = Flowable.defer(() -> {
    if (!isStoryGenerated(invocationContext)) {
      logger.log(Level.SEVERE,() ->
          String.format("[%s] Failed to generate initial story. Abor
              name()));
      return Flowable.empty(); // Stop further processing if no stor
    }
      logger.log(Level.INFO, () ->
          String.format("[%s] Story state after generator: %s",
              name(), invocationContext.session().state().get("curre
      return runStage(loopAgent, invocationContext, "CriticReviserLo
  });

  // Stage 3: Post-Processing (runs after critic-reviser loop comple
  Flowable<Event> postProcessingFlow = Flowable.defer(() -> {
    logger.log(Level.INFO, () ->
        String.format("[%s] Story state after loop: %s",
            name(), invocationContext.session().state().get("current
    return runStage(sequentialAgent, invocationContext, "PostProcess
  });

  // Stage 4: Conditional Regeneration (runs after post-processing c
  Flowable<Event> conditionalRegenFlow = Flowable.defer(() -> {
    String toneCheckResult = (String) invocationContext.session().st
    logger.log(Level.INFO, () -> String.format("[%s] Tone check resu

    if ("negative".equalsIgnoreCase(toneCheckResult)) {
      logger.log(Level.INFO, () ->
          String.format("[%s] Tone is negative. Regenerating story..
      return runStage(storyGenerator, invocationContext, "StoryGener
    } else {
      logger.log(Level.INFO, () ->
          String.format("[%s] Tone is not negative. Keeping current
      return Flowable.empty(); // No regeneration needed
    }
  });

  return Flowable.concatArray(storyGenFlow, criticReviserFlow, postP
      .doOnComplete(() -> logger.log(Level.INFO, () -> String.format
}

// Helper method for a single agent run stage with logging
private Flowable<Event> runStage(BaseAgent agentToRun, InvocationCon
  logger.log(Level.INFO, () -> String.format("[%s] Running %s...", n
  return agentToRun
      .runAsync(ctx)
      .doOnNext(event ->
            logger.log(Level.INFO,() ->
                String.format("[%s] Event from %s: %s", name(), stageN
        .doOnError(err ->
            logger.log(Level.SEVERE,
                String.format("[%s] Error in %s", name(), stageName),
        .doOnComplete(() ->
            logger.log(Level.INFO, () ->
                String.format("[%s] %s finished.", name(), stageName))
  }

  @Override
  protected Flowable<Event> runLiveImpl(InvocationContext invocationCo
    return Flowable.error(new UnsupportedOperationException("runLive r
  }
}
```