Agents

In the Agent Development Kit (ADK), an **Agent** is a self-contained execution unit designed to act autonomously to achieve specific goals. Agents can perform tasks, interact with users, utilize external tools, and coordinate with other agents.

The foundation for all agents in ADK is the BaseAgent class. It serves as the fundamental blueprint. To create functional agents, you typically extend BaseAgent in one of three main ways, catering to different needs – from intelligent reasoning to structured process control.

Core Agent Categories¶

ADK provides distinct agent categories to build sophisticated applications:

**• LLM Agents (LlmAgent, Agent)**: These agents utilize Large Language Models (LLMs) as their core engine to understand natural language, reason, plan, generate responses, and dynamically decide how to proceed or which tools to use, making them ideal for flexible, language-centric tasks. Learn more about LLM Agents...

**• Workflow Agents (SequentialAgent, ParallelAgent, LoopAgent)**: These specialized agents control the execution flow of other agents in predefined, deterministic patterns (sequence, parallel, or loop) without using an LLM for the flow control itself, perfect for structured processes needing predictable execution. Explore Workflow Agents...

**• Custom Agents**: Created by extending BaseAgent directly, these agents allow you to implement unique operational logic, specific control flows, or specialized integrations not covered by the standard types, catering to highly tailored application requirements. Discover how to build Custom Agents...

Choosing the Right Agent Type¶

The following table provides a high-level comparison to help distinguish between the agent types. As you explore each type in more detail in the subsequent sections, these distinctions will become clearer.

**Feature**

**LLM Agent (LlmAgent)**

**Workflow Agent**

**Custom Agent (BaseAgent subclass)**

**Primary Function**

Reasoning, Generation, Tool Use

Controlling Agent Execution Flow

Implementing Unique Logic/Integrations

**Core Engine**

Large Language Model (LLM)

Predefined Logic (Sequence, Parallel, Loop)

Custom Code

**Determinism**

Non-deterministic (Flexible)

Deterministic (Predictable)

Can be either, based on implementation

**Primary Use**

Language tasks, Dynamic decisions

Structured processes, Orchestration

Tailored requirements, Specific workflows

Agents Working Together: Multi-Agent Systems¶

While each agent type serves a distinct purpose, the true power often comes from combining them. Complex applications frequently employ multi-agent architectures where:

**• LLM Agents** handle intelligent, language-based task execution.

**• Workflow Agents** manage the overall process flow using standard patterns.

**• Custom Agents** provide specialized capabilities or rules needed for unique integrations.

Understanding these core types is the first step toward building sophisticated, capable AI applications with ADK.

What's Next?¶

Now that you have an overview of the different agent types available in ADK, dive deeper into how they work and how to use them effectively:

**• LLM Agents:** Explore how to configure agents powered by large language models, including setting instructions, providing tools, and enabling advanced features like planning and code execution.

**• Workflow Agents:** Learn how to orchestrate tasks using SequentialAgent, ParallelAgent, and LoopAgent for structured and predictable processes.

**• Custom Agents:** Discover the principles of extending BaseAgent to build agents with unique logic and integrations tailored to your specific needs.

**• Multi-Agents:** Understand how to combine different agent types to create sophisticated, collaborative systems capable of tackling complex problems.

**• Models:** Learn about the different LLM integrations available and how to select the right model for your agents.

LLM Agent¶

The LlmAgent (often aliased simply as Agent) is a core component in ADK, acting as the "thinking" part of your application. It leverages the power of a Large Language Model (LLM) for reasoning, understanding natural language, making decisions, generating responses, and interacting with tools.

Unlike deterministic Workflow Agents that follow predefined execution paths, LlmAgent behavior is non-deterministic. It uses the LLM to interpret instructions and context, deciding dynamically how to proceed, which tools to use (if any), or whether to transfer control to another agent.

Building an effective LlmAgent involves defining its identity, clearly guiding its behavior through instructions, and equipping it with the necessary tools and capabilities.

Defining the Agent's Identity and Purpose¶

First, you need to establish what the agent *is* and what it's *for*.

**• name (Required):** Every agent needs a unique string identifier. This name is crucial for internal operations, especially in multi-agent systems where agents need to refer to or delegate tasks to each other. Choose a descriptive name that reflects the agent's function (e.g., customer\_support\_router, billing\_inquiry\_agent). Avoid reserved names like user.

**• description (Optional, Recommended for Multi-Agent):** Provide a concise summary of the agent's capabilities. This description is primarily used by *other* LLM agents to determine if they should route a task to this agent. Make it specific enough to differentiate it from peers (e.g., "Handles inquiries about current billing statements," not just "Billing agent").

**• model (Required):** Specify the underlying LLM that will power this agent's reasoning. This is a string identifier like "gemini-2.0-flash". The choice of model impacts the agent's capabilities, cost, and performance. See the Models page for available options and considerations.

**Python**

**Java**

# Example: Defining the basic identity

capital\_agent = LlmAgent(

model="gemini-2.0-flash",

name="capital\_agent",

description="Answers user questions about the capital city of a given country."

# instruction and tools will be added next

)

Guiding the Agent: Instructions (instruction)¶

The instruction parameter is arguably the most critical for shaping an LlmAgent's behavior. It's a string (or a function returning a string) that tells the agent:

• Its core task or goal.

• Its personality or persona (e.g., "You are a helpful assistant," "You are a witty pirate").

• Constraints on its behavior (e.g., "Only answer questions about X," "Never reveal Y").

• How and when to use its tools. You should explain the purpose of each tool and the circumstances under which it should be called, supplementing any descriptions within the tool itself.

• The desired format for its output (e.g., "Respond in JSON," "Provide a bulleted list").

**Tips for Effective Instructions:**

**• Be Clear and Specific:** Avoid ambiguity. Clearly state the desired actions and outcomes.

**• Use Markdown:** Improve readability for complex instructions using headings, lists, etc.

**• Provide Examples (Few-Shot):** For complex tasks or specific output formats, include examples directly in the instruction.

**• Guide Tool Use:** Don't just list tools; explain *when* and *why* the agent should use them.

**State:**

• The instruction is a string template, you can use the {var} syntax to insert dynamic values into the instruction.

• {var} is used to insert the value of the state variable named var.

• {artifact.var} is used to insert the text content of the artifact named var.

• If the state variable or artifact does not exist, the agent will raise an error. If you want to ignore the error, you can append a ? to the variable name as in {var?}.

**Python**

**Java**

# Example: Adding instructions

capital\_agent = LlmAgent(

model="gemini-2.0-flash",

name="capital\_agent",

description="Answers user questions about the capital city of a given country.",

instruction="""You are an agent that provides the capital city of a country.

When a user asks for the capital of a country:

1. Identify the country name from the user's query.

2. Use the `get\_capital\_city` tool to find the capital.

3. Respond clearly to the user, stating the capital city.

Example Query: "What's the capital of {country}?"

Example Response: "The capital of France is Paris."

""",

# tools will be added next

)

*(Note: For instructions that apply to* all *agents in a system, consider using global\_instruction on the root agent, detailed further in the Multi-Agents section.)*

Equipping the Agent: Tools (tools)¶

Tools give your LlmAgent capabilities beyond the LLM's built-in knowledge or reasoning. They allow the agent to interact with the outside world, perform calculations, fetch real-time data, or execute specific actions.

**• tools (Optional):** Provide a list of tools the agent can use. Each item in the list can be:

• A native function or method (wrapped as a FunctionTool). Python ADK automatically wraps the native function into a FuntionTool whereas, you must explicitly wrap your Java methods using FunctionTool.create(...)

• An instance of a class inheriting from BaseTool.

• An instance of another agent (AgentTool, enabling agent-to-agent delegation - see Multi-Agents).

The LLM uses the function/tool names, descriptions (from docstrings or the description field), and parameter schemas to decide which tool to call based on the conversation and its instructions.

**Python**

**Java**

# Define a tool function

def get\_capital\_city(country: str) -> str:

"""Retrieves the capital city for a given country."""

# Replace with actual logic (e.g., API call, database lookup)

capitals = {"france": "Paris", "japan": "Tokyo", "canada": "Ottawa"}

return capitals.get(country.lower(), f"Sorry, I don't know the capital of {country}.")

# Add the tool to the agent

capital\_agent = LlmAgent(

model="gemini-2.0-flash",

name="capital\_agent",

description="Answers user questions about the capital city of a given country.",

instruction="""You are an agent that provides the capital city of a country... (previous instruction text)""",

tools=[get\_capital\_city] # Provide the function directly

)

Learn more about Tools in the Tools section.

Advanced Configuration & Control¶

Beyond the core parameters, LlmAgent offers several options for finer control:

Fine-Tuning LLM Generation (generate\_content\_config)¶

You can adjust how the underlying LLM generates responses using generate\_content\_config.

**• generate\_content\_config (Optional):** Pass an instance of google.genai.types.GenerateContentConfig to control parameters like temperature (randomness), max\_output\_tokens (response length), top\_p, top\_k, and safety settings.

**Python**

**Java**

from google.genai import types

agent = LlmAgent(

# ... other params

generate\_content\_config=types.GenerateContentConfig(

temperature=0.2, # More deterministic output

max\_output\_tokens=250

)

)

Structuring Data (input\_schema, output\_schema, output\_key)¶

For scenarios requiring structured data exchange with an LLM Agent, the ADK provides mechanisms to define expected input and desired output formats using schema definitions.

**• input\_schema (Optional):** Define a schema representing the expected input structure. If set, the user message content passed to this agent *must* be a JSON string conforming to this schema. Your instructions should guide the user or preceding agent accordingly.

**• output\_schema (Optional):** Define a schema representing the desired output structure. If set, the agent's final response *must* be a JSON string conforming to this schema.

**• Constraint:** Using output\_schema enables controlled generation within the LLM but **disables the agent's ability to use tools or transfer control to other agents**. Your instructions must guide the LLM to produce JSON matching the schema directly.

**• output\_key (Optional):** Provide a string key. If set, the text content of the agent's *final* response will be automatically saved to the session's state dictionary under this key. This is useful for passing results between agents or steps in a workflow.

• In Python, this might look like: session.state[output\_key] = agent\_response\_text

• In Java: session.state().put(outputKey, agentResponseText)

**Python**

**Java**

The input and output schema is typically a Pydantic BaseModel.

from pydantic import BaseModel, Field

class CapitalOutput(BaseModel):

capital: str = Field(description="The capital of the country.")

structured\_capital\_agent = LlmAgent(

# ... name, model, description

instruction="""You are a Capital Information Agent. Given a country, respond ONLY with a JSON object containing the capital. Format: {"capital": "capital\_name"}""",

output\_schema=CapitalOutput, # Enforce JSON output

output\_key="found\_capital" # Store result in state['found\_capital']

# Cannot use tools=[get\_capital\_city] effectively here

)

Managing Context (include\_contents)¶

Control whether the agent receives the prior conversation history.

**• include\_contents (Optional, Default: 'default'):** Determines if the contents (history) are sent to the LLM.

• 'default': The agent receives the relevant conversation history.

• 'none': The agent receives no prior contents. It operates based solely on its current instruction and any input provided in the *current* turn (useful for stateless tasks or enforcing specific contexts).

**Python**

**Java**

stateless\_agent = LlmAgent(

# ... other params

include\_contents='none'

)

Planning & Code Execution¶

For more complex reasoning involving multiple steps or executing code:

**• planner (Optional):** Assign a BasePlanner instance to enable multi-step reasoning and planning before execution. (See Multi-Agents patterns).

**• code\_executor (Optional):** Provide a BaseCodeExecutor instance to allow the agent to execute code blocks (e.g., Python) found in the LLM's response. (See Tools/Built-in tools).

Putting It Together: Example¶

**Code**

Here's the complete basic capital\_agent:

**Python**

**Java**

# --- Full example code demonstrating LlmAgent with Tools vs. Output Schema ---

import json # Needed for pretty printing dicts

from google.adk.agents import LlmAgent

from google.adk.runners import Runner

from google.adk.sessions import InMemorySessionService

from google.genai import types

from pydantic import BaseModel, Field

# --- 1. Define Constants ---

APP\_NAME = "agent\_comparison\_app"

USER\_ID = "test\_user\_456"

SESSION\_ID\_TOOL\_AGENT = "session\_tool\_agent\_xyz"

SESSION\_ID\_SCHEMA\_AGENT = "session\_schema\_agent\_xyz"

MODEL\_NAME = "gemini-2.0-flash"

# --- 2. Define Schemas ---

# Input schema used by both agents

class CountryInput(BaseModel):

country: str = Field(description="The country to get information about.")

# Output schema ONLY for the second agent

class CapitalInfoOutput(BaseModel):

capital: str = Field(description="The capital city of the country.")

# Note: Population is illustrative; the LLM will infer or estimate this

# as it cannot use tools when output\_schema is set.

population\_estimate: str = Field(description="An estimated population of the capital city.")

# --- 3. Define the Tool (Only for the first agent) ---

def get\_capital\_city(country: str) -> str:

"""Retrieves the capital city of a given country."""

print(f"\n-- Tool Call: get\_capital\_city(country='{country}') --")

country\_capitals = {

"united states": "Washington, D.C.",

"canada": "Ottawa",

"france": "Paris",

"japan": "Tokyo",

}

result = country\_capitals.get(country.lower(), f"Sorry, I couldn't find the capital for {country}.")

print(f"-- Tool Result: '{result}' --")

return result

# --- 4. Configure Agents ---

# Agent 1: Uses a tool and output\_key

capital\_agent\_with\_tool = LlmAgent(

model=MODEL\_NAME,

name="capital\_agent\_tool",

description="Retrieves the capital city using a specific tool.",

instruction="""You are a helpful agent that provides the capital city of a country using a tool.

The user will provide the country name in a JSON format like {"country": "country\_name"}.

1. Extract the country name.

2. Use the `get\_capital\_city` tool to find the capital.

3. Respond clearly to the user, stating the capital city found by the tool.

""",

tools=[get\_capital\_city],

input\_schema=CountryInput,

output\_key="capital\_tool\_result", # Store final text response

)

# Agent 2: Uses output\_schema (NO tools possible)

structured\_info\_agent\_schema = LlmAgent(

model=MODEL\_NAME,

name="structured\_info\_agent\_schema",

description="Provides capital and estimated population in a specific JSON format.",

instruction=f"""You are an agent that provides country information.

The user will provide the country name in a JSON format like {{"country": "country\_name"}}.

Respond ONLY with a JSON object matching this exact schema:

{json.dumps(CapitalInfoOutput.model\_json\_schema(), indent=2)}

Use your knowledge to determine the capital and estimate the population. Do not use any tools.

""",

# \*\*\* NO tools parameter here - using output\_schema prevents tool use \*\*\*

input\_schema=CountryInput,

output\_schema=CapitalInfoOutput, # Enforce JSON output structure

output\_key="structured\_info\_result", # Store final JSON response

)

# --- 5. Set up Session Management and Runners ---

session\_service = InMemorySessionService()

# Create separate sessions for clarity, though not strictly necessary if context is managed

session\_service.create\_session(app\_name=APP\_NAME, user\_id=USER\_ID, session\_id=SESSION\_ID\_TOOL\_AGENT)

session\_service.create\_session(app\_name=APP\_NAME, user\_id=USER\_ID, session\_id=SESSION\_ID\_SCHEMA\_AGENT)

# Create a runner for EACH agent

capital\_runner = Runner(

agent=capital\_agent\_with\_tool,

app\_name=APP\_NAME,

session\_service=session\_service

)

structured\_runner = Runner(

agent=structured\_info\_agent\_schema,

app\_name=APP\_NAME,

session\_service=session\_service

)

# --- 6. Define Agent Interaction Logic ---

async def call\_agent\_and\_print(

runner\_instance: Runner,

agent\_instance: LlmAgent,

session\_id: str,

query\_json: str

):

"""Sends a query to the specified agent/runner and prints results."""

print(f"\n>>> Calling Agent: '{agent\_instance.name}' | Query: {query\_json}")

user\_content = types.Content(role='user', parts=[types.Part(text=query\_json)])

final\_response\_content = "No final response received."

async for event in runner\_instance.run\_async(user\_id=USER\_ID, session\_id=session\_id, new\_message=user\_content):

# print(f"Event: {event.type}, Author: {event.author}") # Uncomment for detailed logging

if event.is\_final\_response() and event.content and event.content.parts:

# For output\_schema, the content is the JSON string itself

final\_response\_content = event.content.parts[0].text

print(f"<<< Agent '{agent\_instance.name}' Response: {final\_response\_content}")

current\_session = session\_service.get\_session(app\_name=APP\_NAME,

user\_id=USER\_ID,

session\_id=session\_id)

stored\_output = current\_session.state.get(agent\_instance.output\_key)

# Pretty print if the stored output looks like JSON (likely from output\_schema)

print(f"--- Session State ['{agent\_instance.output\_key}']: ", end="")

try:

# Attempt to parse and pretty print if it's JSON

parsed\_output = json.loads(stored\_output)

print(json.dumps(parsed\_output, indent=2))

except (json.JSONDecodeError, TypeError):

# Otherwise, print as string

print(stored\_output)

print("-" \* 30)

# --- 7. Run Interactions ---

async def main():

print("--- Testing Agent with Tool ---")

await call\_agent\_and\_print(capital\_runner, capital\_agent\_with\_tool, SESSION\_ID\_TOOL\_AGENT, '{"country": "France"}')

await call\_agent\_and\_print(capital\_runner, capital\_agent\_with\_tool, SESSION\_ID\_TOOL\_AGENT, '{"country": "Canada"}')

print("\n\n--- Testing Agent with Output Schema (No Tool Use) ---")

await call\_agent\_and\_print(structured\_runner, structured\_info\_agent\_schema, SESSION\_ID\_SCHEMA\_AGENT, '{"country": "France"}')

await call\_agent\_and\_print(structured\_runner, structured\_info\_agent\_schema, SESSION\_ID\_SCHEMA\_AGENT, '{"country": "Japan"}')

if \_\_name\_\_ == "\_\_main\_\_":

await main()

*(This example demonstrates the core concepts. More complex agents might incorporate schemas, context control, planning, etc.)*

Related Concepts (Deferred Topics)¶

While this page covers the core configuration of LlmAgent, several related concepts provide more advanced control and are detailed elsewhere:

**• Callbacks:** Intercepting execution points (before/after model calls, before/after tool calls) using before\_model\_callback, after\_model\_callback, etc. See Callbacks.

**• Multi-Agent Control:** Advanced strategies for agent interaction, including planning (planner), controlling agent transfer (disallow\_transfer\_to\_parent, disallow\_transfer\_to\_peers), and system-wide instructions (global\_instruction). See Multi-Agents.

Workflow Agents¶

This section introduces "*workflow agents*" - **specialized agents that control the execution flow of its sub-agents**.

Workflow agents are specialized components in ADK designed purely for **orchestrating the execution flow of sub-agents**. Their primary role is to manage *how* and *when* other agents run, defining the control flow of a process.

Unlike LLM Agents, which use Large Language Models for dynamic reasoning and decision-making, Workflow Agents operate based on **predefined logic**. They determine the execution sequence according to their type (e.g., sequential, parallel, loop) without consulting an LLM for the orchestration itself. This results in **deterministic and predictable execution patterns**.

ADK provides three core workflow agent types, each implementing a distinct execution pattern:

• **Sequential Agents**

• Executes sub-agents one after another, in **sequence**.  
 Learn more

• **Loop Agents**

**• Repeatedly** executes its sub-agents until a specific termination condition is met.  
 Learn more

• **Parallel Agents**

• Executes multiple sub-agents in **parallel**.  
 Learn more

Why Use Workflow Agents?¶

Workflow agents are essential when you need explicit control over how a series of tasks or agents are executed. They provide:

**• Predictability:** The flow of execution is guaranteed based on the agent type and configuration.

**• Reliability:** Ensures tasks run in the required order or pattern consistently.

**• Structure:** Allows you to build complex processes by composing agents within clear control structures.

While the workflow agent manages the control flow deterministically, the sub-agents it orchestrates can themselves be any type of agent, including intelligent LLM Agent instances. This allows you to combine structured process control with flexible, LLM-powered task execution.

Sequential agents¶

The SequentialAgent¶

The SequentialAgent is a workflow agent that executes its sub-agents in the order they are specified in the list.

Use the SequentialAgent when you want the execution to occur in a fixed, strict order.

Example¶

• You want to build an agent that can summarize any webpage, using two tools: Get Page Contents and Summarize Page. Because the agent must always call Get Page Contents before calling Summarize Page (you can't summarize from nothing!), you should build your agent using a SequentialAgent.

As with other workflow agents, the SequentialAgent is not powered by an LLM, and is thus deterministic in how it executes. That being said, workflow agents are concerned only with their execution (i.e. in sequence), and not their internal logic; the tools or sub-agents of a workflow agent may or may not utilize LLMs.

How it works¶

When the SequentialAgent's Run Async method is called, it performs the following actions:

**• Iteration:** It iterates through the sub agents list in the order they were provided.

**• Sub-Agent Execution:** For each sub-agent in the list, it calls the sub-agent's Run Async method.

Full Example: Code Development Pipeline¶

Consider a simplified code development pipeline:

**• Code Writer Agent:** An LLM Agent that generates initial code based on a specification.

**• Code Reviewer Agent:** An LLM Agent that reviews the generated code for errors, style issues, and adherence to best practices. It receives the output of the Code Writer Agent.

**• Code Refactorer Agent:** An LLM Agent that takes the reviewed code (and the reviewer's comments) and refactors it to improve quality and address issues.

A SequentialAgent is perfect for this:

SequentialAgent(sub\_agents=[CodeWriterAgent, CodeReviewerAgent, CodeRefactorerAgent])

This ensures the code is written, *then* reviewed, and *finally* refactored, in a strict, dependable order. **The output from each sub-agent is passed to the next by storing them in state via Output Key**.

**Code**

**Python**

**Java**

# Part of agent.py --> Follow https://google.github.io/adk-docs/get-started/quickstart/ to learn the setup

# --- 1. Define Sub-Agents for Each Pipeline Stage ---

# Code Writer Agent

# Takes the initial specification (from user query) and writes code.

code\_writer\_agent = LlmAgent(

name="CodeWriterAgent",

model=GEMINI\_MODEL,

# Change 3: Improved instruction

instruction="""You are a Python Code Generator.

Based \*only\* on the user's request, write Python code that fulfills the requirement.

Output \*only\* the complete Python code block, enclosed in triple backticks (```python ... ```).

Do not add any other text before or after the code block.

""",

description="Writes initial Python code based on a specification.",

output\_key="generated\_code" # Stores output in state['generated\_code']

)

# Code Reviewer Agent

# Takes the code generated by the previous agent (read from state) and provides feedback.

code\_reviewer\_agent = LlmAgent(

name="CodeReviewerAgent",

model=GEMINI\_MODEL,

# Change 3: Improved instruction, correctly using state key injection

instruction="""You are an expert Python Code Reviewer.

Your task is to provide constructive feedback on the provided code.

\*\*Code to Review:\*\*

```python

{generated\_code}

```

\*\*Review Criteria:\*\*

1. \*\*Correctness:\*\* Does the code work as intended? Are there logic errors?

2. \*\*Readability:\*\* Is the code clear and easy to understand? Follows PEP 8 style guidelines?

3. \*\*Efficiency:\*\* Is the code reasonably efficient? Any obvious performance bottlenecks?

4. \*\*Edge Cases:\*\* Does the code handle potential edge cases or invalid inputs gracefully?

5. \*\*Best Practices:\*\* Does the code follow common Python best practices?

\*\*Output:\*\*

Provide your feedback as a concise, bulleted list. Focus on the most important points for improvement.

If the code is excellent and requires no changes, simply state: "No major issues found."

Output \*only\* the review comments or the "No major issues" statement.

""",

description="Reviews code and provides feedback.",

output\_key="review\_comments", # Stores output in state['review\_comments']

)

# Code Refactorer Agent

# Takes the original code and the review comments (read from state) and refactors the code.

code\_refactorer\_agent = LlmAgent(

name="CodeRefactorerAgent",

model=GEMINI\_MODEL,

# Change 3: Improved instruction, correctly using state key injection

instruction="""You are a Python Code Refactoring AI.

Your goal is to improve the given Python code based on the provided review comments.

\*\*Original Code:\*\*

```python

{generated\_code}

```

\*\*Review Comments:\*\*

{review\_comments}

\*\*Task:\*\*

Carefully apply the suggestions from the review comments to refactor the original code.

If the review comments state "No major issues found," return the original code unchanged.

Ensure the final code is complete, functional, and includes necessary imports and docstrings.

\*\*Output:\*\*

Output \*only\* the final, refactored Python code block, enclosed in triple backticks (```python ... ```).

Do not add any other text before or after the code block.

""",

description="Refactors code based on review comments.",

output\_key="refactored\_code", # Stores output in state['refactored\_code']

)

# --- 2. Create the SequentialAgent ---

# This agent orchestrates the pipeline by running the sub\_agents in order.

code\_pipeline\_agent = SequentialAgent(

name="CodePipelineAgent",

sub\_agents=[code\_writer\_agent, code\_reviewer\_agent, code\_refactorer\_agent],

description="Executes a sequence of code writing, reviewing, and refactoring.",

# The agents will run in the order provided: Writer -> Reviewer -> Refactorer

)

# For ADK tools compatibility, the root agent must be named `root\_agent`

root\_agent = code\_pipeline\_agent

Back to top

Previous

Workflow Agents

Next

Loop agents

Copyright Google 2025

Made with Material for MkDocs

Copied to clipboard

Loop agents¶

The LoopAgent¶

The LoopAgent is a workflow agent that executes its sub-agents in a loop (i.e. iteratively). It ***repeatedly runs* a sequence of agents** for a specified number of iterations or until a termination condition is met.

Use the LoopAgent when your workflow involves repetition or iterative refinement, such as like revising code.

Example¶

• You want to build an agent that can generate images of food, but sometimes when you want to generate a specific number of items (e.g. 5 bananas), it generates a different number of those items in the image (e.g. an image of 7 bananas). You have two tools: Generate Image, Count Food Items. Because you want to keep generating images until it either correctly generates the specified number of items, or after a certain number of iterations, you should build your agent using a LoopAgent.

As with other workflow agents, the LoopAgent is not powered by an LLM, and is thus deterministic in how it executes. That being said, workflow agents are only concerned only with their execution (i.e. in a loop), and not their internal logic; the tools or sub-agents of a workflow agent may or may not utilize LLMs.

How it Works¶

When the LoopAgent's Run Async method is called, it performs the following actions:

**• Sub-Agent Execution:** It iterates through the Sub Agents list *in order*. For *each* sub-agent, it calls the agent's Run Async method.

**• Termination Check:***Crucially*, the LoopAgent itself does *not* inherently decide when to stop looping. You *must* implement a termination mechanism to prevent infinite loops. Common strategies include:

**• Max Iterations**: Set a maximum number of iterations in the LoopAgent. **The loop will terminate after that many iterations**.

**• Escalation from sub-agent**: Design one or more sub-agents to evaluate a condition (e.g., "Is the document quality good enough?", "Has a consensus been reached?"). If the condition is met, the sub-agent can signal termination (e.g., by raising a custom event, setting a flag in a shared context, or returning a specific value).

Full Example: Iterative Document Improvement¶

Imagine a scenario where you want to iteratively improve a document:

**• Writer Agent:** An LlmAgent that generates or refines a draft on a topic.

**• Critic Agent:** An LlmAgent that critiques the draft, identifying areas for improvement.

LoopAgent(sub\_agents=[WriterAgent, CriticAgent], max\_iterations=5)

•

In this setup, the LoopAgent would manage the iterative process. The CriticAgent could be **designed to return a "STOP" signal when the document reaches a satisfactory quality level**, preventing further iterations. Alternatively, the max iterations parameter could be used to limit the process to a fixed number of cycles, or external logic could be implemented to make stop decisions. The **loop would run at most five times**, ensuring the iterative refinement doesn't continue indefinitely.

**Full Code**

**Python**

**Java**

# Part of agent.py --> Follow https://google.github.io/adk-docs/get-started/quickstart/ to learn the setup

# --- Constants ---

APP\_NAME = "doc\_writing\_app\_v3" # New App Name

USER\_ID = "dev\_user\_01"

SESSION\_ID\_BASE = "loop\_exit\_tool\_session" # New Base Session ID

GEMINI\_MODEL = "gemini-2.0-flash"

STATE\_INITIAL\_TOPIC = "initial\_topic"

# --- State Keys ---

STATE\_CURRENT\_DOC = "current\_document"

STATE\_CRITICISM = "criticism"

# Define the exact phrase the Critic should use to signal completion

COMPLETION\_PHRASE = "No major issues found."

# --- Tool Definition ---

def exit\_loop(tool\_context: ToolContext):

"""Call this function ONLY when the critique indicates no further changes are needed, signaling the iterative process should end."""

print(f" [Tool Call] exit\_loop triggered by {tool\_context.agent\_name}")

tool\_context.actions.escalate = True

# Return empty dict as tools should typically return JSON-serializable output

return {}

# --- Agent Definitions ---

# STEP 1: Initial Writer Agent (Runs ONCE at the beginning)

initial\_writer\_agent = LlmAgent(

name="InitialWriterAgent",

model=GEMINI\_MODEL,

include\_contents='none',

# MODIFIED Instruction: Ask for a slightly more developed start

instruction=f"""You are a Creative Writing Assistant tasked with starting a story.

Write the \*first draft\* of a short story (aim for 2-4 sentences).

Base the content \*only\* on the topic provided below. Try to introduce a specific element (like a character, a setting detail, or a starting action) to make it engaging.

Topic: {{initial\_topic}}

Output \*only\* the story/document text. Do not add introductions or explanations.

""",

description="Writes the initial document draft based on the topic, aiming for some initial substance.",

output\_key=STATE\_CURRENT\_DOC

)

# STEP 2a: Critic Agent (Inside the Refinement Loop)

critic\_agent\_in\_loop = LlmAgent(

name="CriticAgent",

model=GEMINI\_MODEL,

include\_contents='none',

# MODIFIED Instruction: More nuanced completion criteria, look for clear improvement paths.

instruction=f"""You are a Constructive Critic AI reviewing a short document draft (typically 2-6 sentences). Your goal is balanced feedback.

\*\*Document to Review:\*\*

```

{{current\_document}}

```

\*\*Task:\*\*

Review the document for clarity, engagement, and basic coherence according to the initial topic (if known).

IF you identify 1-2 \*clear and actionable\* ways the document could be improved to better capture the topic or enhance reader engagement (e.g., "Needs a stronger opening sentence", "Clarify the character's goal"):

Provide these specific suggestions concisely. Output \*only\* the critique text.

ELSE IF the document is coherent, addresses the topic adequately for its length, and has no glaring errors or obvious omissions:

Respond \*exactly\* with the phrase "{COMPLETION\_PHRASE}" and nothing else. It doesn't need to be perfect, just functionally complete for this stage. Avoid suggesting purely subjective stylistic preferences if the core is sound.

Do not add explanations. Output only the critique OR the exact completion phrase.

""",

description="Reviews the current draft, providing critique if clear improvements are needed, otherwise signals completion.",

output\_key=STATE\_CRITICISM

)

# STEP 2b: Refiner/Exiter Agent (Inside the Refinement Loop)

refiner\_agent\_in\_loop = LlmAgent(

name="RefinerAgent",

model=GEMINI\_MODEL,

# Relies solely on state via placeholders

include\_contents='none',

instruction=f"""You are a Creative Writing Assistant refining a document based on feedback OR exiting the process.

\*\*Current Document:\*\*

```

{{current\_document}}

```

\*\*Critique/Suggestions:\*\*

{{criticism}}

\*\*Task:\*\*

Analyze the 'Critique/Suggestions'.

IF the critique is \*exactly\* "{COMPLETION\_PHRASE}":

You MUST call the 'exit\_loop' function. Do not output any text.

ELSE (the critique contains actionable feedback):

Carefully apply the suggestions to improve the 'Current Document'. Output \*only\* the refined document text.

Do not add explanations. Either output the refined document OR call the exit\_loop function.

""",

description="Refines the document based on critique, or calls exit\_loop if critique indicates completion.",

tools=[exit\_loop], # Provide the exit\_loop tool

output\_key=STATE\_CURRENT\_DOC # Overwrites state['current\_document'] with the refined version

)

# STEP 2: Refinement Loop Agent

refinement\_loop = LoopAgent(

name="RefinementLoop",

# Agent order is crucial: Critique first, then Refine/Exit

sub\_agents=[

critic\_agent\_in\_loop,

refiner\_agent\_in\_loop,

],

max\_iterations=5 # Limit loops

)

# STEP 3: Overall Sequential Pipeline

# For ADK tools compatibility, the root agent must be named `root\_agent`

root\_agent = SequentialAgent(

name="IterativeWritingPipeline",

sub\_agents=[

initial\_writer\_agent, # Run first to create initial doc

refinement\_loop # Then run the critique/refine loop

],

description="Writes an initial document and then iteratively refines it with critique using an exit tool."

)

Back to top

Previous

Sequential agents

Next

Parallel agents

Copyright Google 2025

Made with Material for MkDocs

Copied to clipboard

Parallel agents¶

The ParallelAgent is a workflow agent that executes its sub-agents *concurrently*. This dramatically speeds up workflows where tasks can be performed independently.

Use ParallelAgent when: For scenarios prioritizing speed and involving independent, resource-intensive tasks, a ParallelAgent facilitates efficient parallel execution. **When sub-agents operate without dependencies, their tasks can be performed concurrently**, significantly reducing overall processing time.

As with other workflow agents, the ParallelAgent is not powered by an LLM, and is thus deterministic in how it executes. That being said, workflow agents are only concerned with their execution (i.e. executing sub-agents in parallel), and not their internal logic; the tools or sub-agents of a workflow agent may or may not utilize LLMs.

Example¶

This approach is particularly beneficial for operations like multi-source data retrieval or heavy computations, where parallelization yields substantial performance gains. Importantly, this strategy assumes no inherent need for shared state or direct information exchange between the concurrently executing agents.

How it works¶

When the ParallelAgent's run\_async() method is called:

**• Concurrent Execution:** It initiates the run\_async() method of *each* sub-agent present in the sub\_agents list *concurrently*. This means all the agents start running at (approximately) the same time.

**• Independent Branches:** Each sub-agent operates in its own execution branch. There is ***no* automatic sharing of conversation history or state between these branches** during execution.

**• Result Collection:** The ParallelAgent manages the parallel execution and, typically, provides a way to access the results from each sub-agent after they have completed (e.g., through a list of results or events). The order of results may not be deterministic.

Independent Execution and State Management¶

It's *crucial* to understand that sub-agents within a ParallelAgent run independently. If you *need* communication or data sharing between these agents, you must implement it explicitly. Possible approaches include:

**• Shared InvocationContext:** You could pass a shared InvocationContext object to each sub-agent. This object could act as a shared data store. However, you'd need to manage concurrent access to this shared context carefully (e.g., using locks) to avoid race conditions.

**• External State Management:** Use an external database, message queue, or other mechanism to manage shared state and facilitate communication between agents.

**• Post-Processing:** Collect results from each branch, and then implement logic to coordinate data afterwards.

Full Example: Parallel Web Research¶

Imagine researching multiple topics simultaneously:

**• Researcher Agent 1:** An LlmAgent that researches "renewable energy sources."

**• Researcher Agent 2:** An LlmAgent that researches "electric vehicle technology."

**• Researcher Agent 3:** An LlmAgent that researches "carbon capture methods."

ParallelAgent(sub\_agents=[ResearcherAgent1, ResearcherAgent2, ResearcherAgent3])

•

These research tasks are independent. Using a ParallelAgent allows them to run concurrently, potentially reducing the total research time significantly compared to running them sequentially. The results from each agent would be collected separately after they finish.

**Full Code**

**Python**

**Java**

# Part of agent.py --> Follow https://google.github.io/adk-docs/get-started/quickstart/ to learn the setup

# --- 1. Define Researcher Sub-Agents (to run in parallel) ---

# Researcher 1: Renewable Energy

researcher\_agent\_1 = LlmAgent(

name="RenewableEnergyResearcher",

model=GEMINI\_MODEL,

instruction="""You are an AI Research Assistant specializing in energy.

Research the latest advancements in 'renewable energy sources'.

Use the Google Search tool provided.

Summarize your key findings concisely (1-2 sentences).

Output \*only\* the summary.

""",

description="Researches renewable energy sources.",

tools=[google\_search],

# Store result in state for the merger agent

output\_key="renewable\_energy\_result"

)

# Researcher 2: Electric Vehicles

researcher\_agent\_2 = LlmAgent(

name="EVResearcher",

model=GEMINI\_MODEL,

instruction="""You are an AI Research Assistant specializing in transportation.

Research the latest developments in 'electric vehicle technology'.

Use the Google Search tool provided.

Summarize your key findings concisely (1-2 sentences).

Output \*only\* the summary.

""",

description="Researches electric vehicle technology.",

tools=[google\_search],

# Store result in state for the merger agent

output\_key="ev\_technology\_result"

)

# Researcher 3: Carbon Capture

researcher\_agent\_3 = LlmAgent(

name="CarbonCaptureResearcher",

model=GEMINI\_MODEL,

instruction="""You are an AI Research Assistant specializing in climate solutions.

Research the current state of 'carbon capture methods'.

Use the Google Search tool provided.

Summarize your key findings concisely (1-2 sentences).

Output \*only\* the summary.

""",

description="Researches carbon capture methods.",

tools=[google\_search],

# Store result in state for the merger agent

output\_key="carbon\_capture\_result"

)

# --- 2. Create the ParallelAgent (Runs researchers concurrently) ---

# This agent orchestrates the concurrent execution of the researchers.

# It finishes once all researchers have completed and stored their results in state.

parallel\_research\_agent = ParallelAgent(

name="ParallelWebResearchAgent",

sub\_agents=[researcher\_agent\_1, researcher\_agent\_2, researcher\_agent\_3],

description="Runs multiple research agents in parallel to gather information."

)

# --- 3. Define the Merger Agent (Runs \*after\* the parallel agents) ---

# This agent takes the results stored in the session state by the parallel agents

# and synthesizes them into a single, structured response with attributions.

merger\_agent = LlmAgent(

name="SynthesisAgent",

model=GEMINI\_MODEL, # Or potentially a more powerful model if needed for synthesis

instruction="""You are an AI Assistant responsible for combining research findings into a structured report.

Your primary task is to synthesize the following research summaries, clearly attributing findings to their source areas. Structure your response using headings for each topic. Ensure the report is coherent and integrates the key points smoothly.

\*\*Crucially: Your entire response MUST be grounded \*exclusively\* on the information provided in the 'Input Summaries' below. Do NOT add any external knowledge, facts, or details not present in these specific summaries.\*\*

\*\*Input Summaries:\*\*

\* \*\*Renewable Energy:\*\*

{renewable\_energy\_result}

\* \*\*Electric Vehicles:\*\*

{ev\_technology\_result}

\* \*\*Carbon Capture:\*\*

{carbon\_capture\_result}

\*\*Output Format:\*\*

## Summary of Recent Sustainable Technology Advancements

### Renewable Energy Findings

(Based on RenewableEnergyResearcher's findings)

[Synthesize and elaborate \*only\* on the renewable energy input summary provided above.]

### Electric Vehicle Findings

(Based on EVResearcher's findings)

[Synthesize and elaborate \*only\* on the EV input summary provided above.]

### Carbon Capture Findings

(Based on CarbonCaptureResearcher's findings)

[Synthesize and elaborate \*only\* on the carbon capture input summary provided above.]

### Overall Conclusion

[Provide a brief (1-2 sentence) concluding statement that connects \*only\* the findings presented above.]

Output \*only\* the structured report following this format. Do not include introductory or concluding phrases outside this structure, and strictly adhere to using only the provided input summary content.

""",

description="Combines research findings from parallel agents into a structured, cited report, strictly grounded on provided inputs.",

# No tools needed for merging

# No output\_key needed here, as its direct response is the final output of the sequence

)

# --- 4. Create the SequentialAgent (Orchestrates the overall flow) ---

# This is the main agent that will be run. It first executes the ParallelAgent

# to populate the state, and then executes the MergerAgent to produce the final output.

sequential\_pipeline\_agent = SequentialAgent(

name="ResearchAndSynthesisPipeline",

# Run parallel research first, then merge

sub\_agents=[parallel\_research\_agent, merger\_agent],

description="Coordinates parallel research and synthesizes the results."

)

root\_agent = sequential\_pipeline\_agent

Back to top

Previous

Loop agents

Next

Custom agents

Copyright Google 2025

Made with Material for MkDocs

Copied to clipboard

**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.

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.

**Python**

**Java**

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.

**Key Capabilities within the Core Asynchronous Method:**

**Python**

**Java**

**• 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

•

**• 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"

•

**• 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.

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 LlmAgents 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¶

**Python**

**Java**

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 story,

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 attributes with type hints

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.g., arbitrary\_types\_allowed

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 criticism.

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], max\_iterations=2

)

sequential\_agent = SequentialAgent(

name="PostProcessing", sub\_agents=[grammar\_check, tone\_check]

)

# 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 annotations.

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 directly

)

Part 2: Defining the Custom Execution Logic¶

**Python**

**Java**

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.story\_generator).

"""

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.model\_dump\_json(indent=2, exclude\_none=True)}")

yield event

# Check if story was generated before proceeding

if "current\_story" not in ctx.session.state or not ctx.session.state["current\_story"]:

logger.error(f"[{self.name}] Failed to generate initial story. Aborting workflow.")

return # Stop processing if initial story failed

logger.info(f"[{self.name}] Story state after generator: {ctx.session.state.get('current\_story')}")

# 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: {event.model\_dump\_json(indent=2, exclude\_none=True)}")

yield event

logger.info(f"[{self.name}] Story state after loop: {ctx.session.state.get('current\_story')}")

# 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.model\_dump\_json(indent=2, exclude\_none=True)}")

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 story...")

async for event in self.story\_generator.run\_async(ctx):

logger.info(f"[{self.name}] Event from StoryGenerator (Regen): {event.model\_dump\_json(indent=2, exclude\_none=True)}")

yield event

else:

logger.info(f"[{self.name}] Tone is not negative. Keeping current story.")

pass

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

**Explanation of Logic:**

• The initial story\_generator runs. Its output is expected to be in ctx.session.state["current\_story"].

• 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.

• The sequential\_agent runs, calling grammar\_check then tone\_check, reading current\_story and writing grammar\_suggestions and tone\_check\_result to the state.

**• 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.

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.

**Python**

**Java**

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 100 words) about a cat,

based on the topic provided in session state with key 'topic'""",

input\_schema=None,

output\_key="current\_story", # Key for storing output in session state

)

critic = LlmAgent(

name="Critic",

model=GEMINI\_2\_FLASH,

instruction="""You are a story critic. Review the story provided in

session state with key 'current\_story'. Provide 1-2 sentences of constructive criticism

on how to improve it. Focus on plot or character.""",

input\_schema=None,

output\_key="criticism", # Key for storing criticism in session state

)

reviser = LlmAgent(

name="Reviser",

model=GEMINI\_2\_FLASH,

instruction="""You are a story reviser. Revise the story provided in

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 story

provided in session state with key 'current\_story'. Output only the suggested

corrections as a list, or output 'Grammar is good!' if there are no errors.""",

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 story

provided in session state with key 'current\_story'. Output only one word: 'positive' if

the tone is generally positive, 'negative' if the tone is generally negative, or 'neutral'

otherwise.""",

input\_schema=None,

output\_key="tone\_check\_result", # This agent's output determines the conditional flow

)

Part 4: Instantiating and Running the custom agent¶

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

**Python**

**Java**

# --- 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 needed)

and runs the workflow.

"""

current\_session = session\_service.get\_session(app\_name=APP\_NAME,

user\_id=USER\_ID,

session\_id=SESSION\_ID)

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"Generate a story about: {user\_input\_topic}")])

events = runner.run(user\_id=USER\_ID, session\_id=SESSION\_ID, new\_message=content)

final\_response = "No final response captured."

for event in events:

if event.is\_final\_response() and event.content and event.content.parts:

logger.info(f"Potential final response from [{event.author}]: {event.content.parts[0].text}")

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")

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

Full Code Example¶

**Storyflow Agent**

**Python**

**Java**

# 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, SequentialAgent

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 story,

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 attributes with type hints

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.g., arbitrary\_types\_allowed

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 criticism.

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], max\_iterations=2

)

sequential\_agent = SequentialAgent(

name="PostProcessing", sub\_agents=[grammar\_check, tone\_check]

)

# 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 annotations.

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 directly

)

@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.story\_generator).

"""

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.model\_dump\_json(indent=2, exclude\_none=True)}")

yield event

# Check if story was generated before proceeding

if "current\_story" not in ctx.session.state or not ctx.session.state["current\_story"]:

logger.error(f"[{self.name}] Failed to generate initial story. Aborting workflow.")

return # Stop processing if initial story failed

logger.info(f"[{self.name}] Story state after generator: {ctx.session.state.get('current\_story')}")

# 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: {event.model\_dump\_json(indent=2, exclude\_none=True)}")

yield event

logger.info(f"[{self.name}] Story state after loop: {ctx.session.state.get('current\_story')}")

# 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.model\_dump\_json(indent=2, exclude\_none=True)}")

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 story...")

async for event in self.story\_generator.run\_async(ctx):

logger.info(f"[{self.name}] Event from StoryGenerator (Regen): {event.model\_dump\_json(indent=2, exclude\_none=True)}")

yield event

else:

logger.info(f"[{self.name}] Tone is not negative. Keeping current story.")

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 (around 100 words) about a cat,

based on the topic provided in session state with key 'topic'""",

input\_schema=None,

output\_key="current\_story", # Key for storing output in session state

)

critic = LlmAgent(

name="Critic",

model=GEMINI\_2\_FLASH,

instruction="""You are a story critic. Review the story provided in

session state with key 'current\_story'. Provide 1-2 sentences of constructive criticism

on how to improve it. Focus on plot or character.""",

input\_schema=None,

output\_key="criticism", # Key for storing criticism in session state

)

reviser = LlmAgent(

name="Reviser",

model=GEMINI\_2\_FLASH,

instruction="""You are a story reviser. Revise the story provided in

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 story

provided in session state with key 'current\_story'. Output only the suggested

corrections as a list, or output 'Grammar is good!' if there are no errors.""",

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 story

provided in session state with key 'current\_story'. Output only one word: 'positive' if

the tone is generally positive, 'negative' if the tone is generally negative, or 'neutral'

otherwise.""",

input\_schema=None,

output\_key="tone\_check\_result", # This agent's output determines the conditional flow

)

# --- 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 needed)

and runs the workflow.

"""

current\_session = session\_service.get\_session(app\_name=APP\_NAME,

user\_id=USER\_ID,

session\_id=SESSION\_ID)

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"Generate a story about: {user\_input\_topic}")])

events = runner.run(user\_id=USER\_ID, session\_id=SESSION\_ID, new\_message=content)

final\_response = "No final response captured."

for event in events:

if event.is\_final\_response() and event.content and event.content.parts:

logger.info(f"Potential final response from [{event.author}]: {event.content.parts[0].text}")

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")

Back to top

Previous

Parallel agents

Next

Multi-agent systems

Copyright Google 2025

Made with Material for MkDocs

Copied to clipboard

Multi-Agent Systems in ADK¶

As agentic applications grow in complexity, structuring them as a single, monolithic agent can become challenging to develop, maintain, and reason about. The Agent Development Kit (ADK) supports building sophisticated applications by composing multiple, distinct BaseAgent instances into a **Multi-Agent System (MAS)**.

In ADK, a multi-agent system is an application where different agents, often forming a hierarchy, collaborate or coordinate to achieve a larger goal. Structuring your application this way offers significant advantages, including enhanced modularity, specialization, reusability, maintainability, and the ability to define structured control flows using dedicated workflow agents.

You can compose various types of agents derived from BaseAgent to build these systems:

**• LLM Agents:** Agents powered by large language models. (See LLM Agents)

**• Workflow Agents:** Specialized agents (SequentialAgent, ParallelAgent, LoopAgent) designed to manage the execution flow of their sub-agents. (See Workflow Agents)

**• Custom agents:** Your own agents inheriting from BaseAgent with specialized, non-LLM logic. (See Custom Agents)

The following sections detail the core ADK primitives—such as agent hierarchy, workflow agents, and interaction mechanisms—that enable you to construct and manage these multi-agent systems effectively.

1. ADK Primitives for Agent Composition¶

ADK provides core building blocks—primitives—that enable you to structure and manage interactions within your multi-agent system.

**Note**

The specific parameters or method names for the primitives may vary slightly by SDK language (e.g., sub\_agents in Python, subAgents in Java). Refer to the language-specific API documentation for details.

1.1. Agent Hierarchy (Parent agent, Sub Agents)¶

The foundation for structuring multi-agent systems is the parent-child relationship defined in BaseAgent.

**• Establishing Hierarchy:** You create a tree structure by passing a list of agent instances to the sub\_agents argument when initializing a parent agent. ADK automatically sets the parent\_agent attribute on each child agent during initialization.

**• Single Parent Rule:** An agent instance can only be added as a sub-agent once. Attempting to assign a second parent will result in a ValueError.

**• Importance:** This hierarchy defines the scope for Workflow Agents and influences the potential targets for LLM-Driven Delegation. You can navigate the hierarchy using agent.parent\_agent or find descendants using agent.find\_agent(name).

**Python**

**Java**

# Conceptual Example: Defining Hierarchy

from google.adk.agents import LlmAgent, BaseAgent

# Define individual agents

greeter = LlmAgent(name="Greeter", model="gemini-2.0-flash")

task\_doer = BaseAgent(name="TaskExecutor") # Custom non-LLM agent

# Create parent agent and assign children via sub\_agents

coordinator = LlmAgent(

name="Coordinator",

model="gemini-2.0-flash",

description="I coordinate greetings and tasks.",

sub\_agents=[ # Assign sub\_agents here

greeter,

task\_doer

]

)

# Framework automatically sets:

# assert greeter.parent\_agent == coordinator

# assert task\_doer.parent\_agent == coordinator

1.2. Workflow Agents as Orchestrators¶

ADK includes specialized agents derived from BaseAgent that don't perform tasks themselves but orchestrate the execution flow of their sub\_agents.

**• SequentialAgent:** Executes its sub\_agents one after another in the order they are listed.

**• Context:** Passes the *same* InvocationContext sequentially, allowing agents to easily pass results via shared state.

**Python**

**Java**

# Conceptual Example: Sequential Pipeline

from google.adk.agents import SequentialAgent, LlmAgent

step1 = LlmAgent(name="Step1\_Fetch", output\_key="data") # Saves output to state['data']

step2 = LlmAgent(name="Step2\_Process", instruction="Process data from state key 'data'.")

pipeline = SequentialAgent(name="MyPipeline", sub\_agents=[step1, step2])

# When pipeline runs, Step2 can access the state['data'] set by Step1.

**• ParallelAgent:** Executes its sub\_agents in parallel. Events from sub-agents may be interleaved.

**• Context:** Modifies the InvocationContext.branch for each child agent (e.g., ParentBranch.ChildName), providing a distinct contextual path which can be useful for isolating history in some memory implementations.

**• State:** Despite different branches, all parallel children access the *same shared* session.state, enabling them to read initial state and write results (use distinct keys to avoid race conditions).

**Python**

**Java**

# Conceptual Example: Parallel Execution

from google.adk.agents import ParallelAgent, LlmAgent

fetch\_weather = LlmAgent(name="WeatherFetcher", output\_key="weather")

fetch\_news = LlmAgent(name="NewsFetcher", output\_key="news")

gatherer = ParallelAgent(name="InfoGatherer", sub\_agents=[fetch\_weather, fetch\_news])

# When gatherer runs, WeatherFetcher and NewsFetcher run concurrently.

# A subsequent agent could read state['weather'] and state['news'].

**• LoopAgent:** Executes its sub\_agents sequentially in a loop.

**• Termination:** The loop stops if the optional max\_iterations is reached, or if any sub-agent returns an Event with escalate=True in it's Event Actions.

**• Context & State:** Passes the *same* InvocationContext in each iteration, allowing state changes (e.g., counters, flags) to persist across loops.

**Python**

**Java**

# Conceptual Example: Loop with Condition

from google.adk.agents import LoopAgent, LlmAgent, BaseAgent

from google.adk.events import Event, EventActions

from google.adk.agents.invocation\_context import InvocationContext

from typing import AsyncGenerator

class CheckCondition(BaseAgent): # Custom agent to check state

async def \_run\_async\_impl(self, ctx: InvocationContext) -> AsyncGenerator[Event, None]:

status = ctx.session.state.get("status", "pending")

is\_done = (status == "completed")

yield Event(author=self.name, actions=EventActions(escalate=is\_done)) # Escalate if done

process\_step = LlmAgent(name="ProcessingStep") # Agent that might update state['status']

poller = LoopAgent(

name="StatusPoller",

max\_iterations=10,

sub\_agents=[process\_step, CheckCondition(name="Checker")]

)

# When poller runs, it executes process\_step then Checker repeatedly

# until Checker escalates (state['status'] == 'completed') or 10 iterations pass.

1.3. Interaction & Communication Mechanisms¶

Agents within a system often need to exchange data or trigger actions in one another. ADK facilitates this through:

**a) Shared Session State (session.state)¶**

The most fundamental way for agents operating within the same invocation (and thus sharing the same Session object via the InvocationContext) to communicate passively.

**• Mechanism:** One agent (or its tool/callback) writes a value (context.state['data\_key'] = processed\_data), and a subsequent agent reads it (data = context.state.get('data\_key')). State changes are tracked via CallbackContext.

**• Convenience:** The output\_key property on LlmAgent automatically saves the agent's final response text (or structured output) to the specified state key.

**• Nature:** Asynchronous, passive communication. Ideal for pipelines orchestrated by SequentialAgent or passing data across LoopAgent iterations.

**• See Also:** State Management

**Python**

**Java**

# Conceptual Example: Using output\_key and reading state

from google.adk.agents import LlmAgent, SequentialAgent

agent\_A = LlmAgent(name="AgentA", instruction="Find the capital of France.", output\_key="capital\_city")

agent\_B = LlmAgent(name="AgentB", instruction="Tell me about the city stored in state key 'capital\_city'.")

pipeline = SequentialAgent(name="CityInfo", sub\_agents=[agent\_A, agent\_B])

# AgentA runs, saves "Paris" to state['capital\_city'].

# AgentB runs, its instruction processor reads state['capital\_city'] to get "Paris".

**b) LLM-Driven Delegation (Agent Transfer)¶**

Leverages an LlmAgent's understanding to dynamically route tasks to other suitable agents within the hierarchy.

**• Mechanism:** The agent's LLM generates a specific function call: transfer\_to\_agent(agent\_name='target\_agent\_name').

**• Handling:** The AutoFlow, used by default when sub-agents are present or transfer isn't disallowed, intercepts this call. It identifies the target agent using root\_agent.find\_agent() and updates the InvocationContext to switch execution focus.

**• Requires:** The calling LlmAgent needs clear instructions on when to transfer, and potential target agents need distinct descriptions for the LLM to make informed decisions. Transfer scope (parent, sub-agent, siblings) can be configured on the LlmAgent.

**• Nature:** Dynamic, flexible routing based on LLM interpretation.

**Python**

**Java**

# Conceptual Setup: LLM Transfer

from google.adk.agents import LlmAgent

booking\_agent = LlmAgent(name="Booker", description="Handles flight and hotel bookings.")

info\_agent = LlmAgent(name="Info", description="Provides general information and answers questions.")

coordinator = LlmAgent(

name="Coordinator",

model="gemini-2.0-flash",

instruction="You are an assistant. Delegate booking tasks to Booker and info requests to Info.",

description="Main coordinator.",

# AutoFlow is typically used implicitly here

sub\_agents=[booking\_agent, info\_agent]

)

# If coordinator receives "Book a flight", its LLM should generate:

# FunctionCall(name='transfer\_to\_agent', args={'agent\_name': 'Booker'})

# ADK framework then routes execution to booking\_agent.

**c) Explicit Invocation (AgentTool)¶**

Allows an LlmAgent to treat another BaseAgent instance as a callable function or Tool.

**• Mechanism:** Wrap the target agent instance in AgentTool and include it in the parent LlmAgent's tools list. AgentTool generates a corresponding function declaration for the LLM.

**• Handling:** When the parent LLM generates a function call targeting the AgentTool, the framework executes AgentTool.run\_async. This method runs the target agent, captures its final response, forwards any state/artifact changes back to the parent's context, and returns the response as the tool's result.

**• Nature:** Synchronous (within the parent's flow), explicit, controlled invocation like any other tool.

**• (Note:** AgentTool needs to be imported and used explicitly).

**Python**

**Java**

# Conceptual Setup: Agent as a Tool

from google.adk.agents import LlmAgent, BaseAgent

from google.adk.tools import agent\_tool

from pydantic import BaseModel

# Define a target agent (could be LlmAgent or custom BaseAgent)

class ImageGeneratorAgent(BaseAgent): # Example custom agent

name: str = "ImageGen"

description: str = "Generates an image based on a prompt."

# ... internal logic ...

async def \_run\_async\_impl(self, ctx): # Simplified run logic

prompt = ctx.session.state.get("image\_prompt", "default prompt")

# ... generate image bytes ...

image\_bytes = b"..."

yield Event(author=self.name, content=types.Content(parts=[types.Part.from\_bytes(image\_bytes, "image/png")]))

image\_agent = ImageGeneratorAgent()

image\_tool = agent\_tool.AgentTool(agent=image\_agent) # Wrap the agent

# Parent agent uses the AgentTool

artist\_agent = LlmAgent(

name="Artist",

model="gemini-2.0-flash",

instruction="Create a prompt and use the ImageGen tool to generate the image.",

tools=[image\_tool] # Include the AgentTool

)

# Artist LLM generates a prompt, then calls:

# FunctionCall(name='ImageGen', args={'image\_prompt': 'a cat wearing a hat'})

# Framework calls image\_tool.run\_async(...), which runs ImageGeneratorAgent.

# The resulting image Part is returned to the Artist agent as the tool result.

These primitives provide the flexibility to design multi-agent interactions ranging from tightly coupled sequential workflows to dynamic, LLM-driven delegation networks.

2. Common Multi-Agent Patterns using ADK Primitives¶

By combining ADK's composition primitives, you can implement various established patterns for multi-agent collaboration.

Coordinator/Dispatcher Pattern¶

**• Structure:** A central LlmAgent (Coordinator) manages several specialized sub\_agents.

**• Goal:** Route incoming requests to the appropriate specialist agent.

**• ADK Primitives Used:**

**• Hierarchy:** Coordinator has specialists listed in sub\_agents.

**• Interaction:** Primarily uses **LLM-Driven Delegation** (requires clear descriptions on sub-agents and appropriate instruction on Coordinator) or **Explicit Invocation (AgentTool)** (Coordinator includes AgentTool-wrapped specialists in its tools).

**Python**

**Java**

# Conceptual Code: Coordinator using LLM Transfer

from google.adk.agents import LlmAgent

billing\_agent = LlmAgent(name="Billing", description="Handles billing inquiries.")

support\_agent = LlmAgent(name="Support", description="Handles technical support requests.")

coordinator = LlmAgent(

name="HelpDeskCoordinator",

model="gemini-2.0-flash",

instruction="Route user requests: Use Billing agent for payment issues, Support agent for technical problems.",

description="Main help desk router.",

# allow\_transfer=True is often implicit with sub\_agents in AutoFlow

sub\_agents=[billing\_agent, support\_agent]

)

# User asks "My payment failed" -> Coordinator's LLM should call transfer\_to\_agent(agent\_name='Billing')

# User asks "I can't log in" -> Coordinator's LLM should call transfer\_to\_agent(agent\_name='Support')

Sequential Pipeline Pattern¶

**• Structure:** A SequentialAgent contains sub\_agents executed in a fixed order.

**• Goal:** Implement a multi-step process where the output of one step feeds into the next.

**• ADK Primitives Used:**

**• Workflow:** SequentialAgent defines the order.

**• Communication:** Primarily uses **Shared Session State**. Earlier agents write results (often via output\_key), later agents read those results from context.state.

**Python**

**Java**

# Conceptual Code: Sequential Data Pipeline

from google.adk.agents import SequentialAgent, LlmAgent

validator = LlmAgent(name="ValidateInput", instruction="Validate the input.", output\_key="validation\_status")

processor = LlmAgent(name="ProcessData", instruction="Process data if state key 'validation\_status' is 'valid'.", output\_key="result")

reporter = LlmAgent(name="ReportResult", instruction="Report the result from state key 'result'.")

data\_pipeline = SequentialAgent(

name="DataPipeline",

sub\_agents=[validator, processor, reporter]

)

# validator runs -> saves to state['validation\_status']

# processor runs -> reads state['validation\_status'], saves to state['result']

# reporter runs -> reads state['result']

Parallel Fan-Out/Gather Pattern¶

**• Structure:** A ParallelAgent runs multiple sub\_agents concurrently, often followed by a later agent (in a SequentialAgent) that aggregates results.

**• Goal:** Execute independent tasks simultaneously to reduce latency, then combine their outputs.

**• ADK Primitives Used:**

**• Workflow:** ParallelAgent for concurrent execution (Fan-Out). Often nested within a SequentialAgent to handle the subsequent aggregation step (Gather).

**• Communication:** Sub-agents write results to distinct keys in **Shared Session State**. The subsequent "Gather" agent reads multiple state keys.

**Python**

**Java**

# Conceptual Code: Parallel Information Gathering

from google.adk.agents import SequentialAgent, ParallelAgent, LlmAgent

fetch\_api1 = LlmAgent(name="API1Fetcher", instruction="Fetch data from API 1.", output\_key="api1\_data")

fetch\_api2 = LlmAgent(name="API2Fetcher", instruction="Fetch data from API 2.", output\_key="api2\_data")

gather\_concurrently = ParallelAgent(

name="ConcurrentFetch",

sub\_agents=[fetch\_api1, fetch\_api2]

)

synthesizer = LlmAgent(

name="Synthesizer",

instruction="Combine results from state keys 'api1\_data' and 'api2\_data'."

)

overall\_workflow = SequentialAgent(

name="FetchAndSynthesize",

sub\_agents=[gather\_concurrently, synthesizer] # Run parallel fetch, then synthesize

)

# fetch\_api1 and fetch\_api2 run concurrently, saving to state.

# synthesizer runs afterwards, reading state['api1\_data'] and state['api2\_data'].

Hierarchical Task Decomposition¶

**• Structure:** A multi-level tree of agents where higher-level agents break down complex goals and delegate sub-tasks to lower-level agents.

**• Goal:** Solve complex problems by recursively breaking them down into simpler, executable steps.

**• ADK Primitives Used:**

**• Hierarchy:** Multi-level parent\_agent/sub\_agents structure.

**• Interaction:** Primarily **LLM-Driven Delegation** or **Explicit Invocation (AgentTool)** used by parent agents to assign tasks to subagents. Results are returned up the hierarchy (via tool responses or state).

**Python**

**Java**

# Conceptual Code: Hierarchical Research Task

from google.adk.agents import LlmAgent

from google.adk.tools import agent\_tool

# Low-level tool-like agents

web\_searcher = LlmAgent(name="WebSearch", description="Performs web searches for facts.")

summarizer = LlmAgent(name="Summarizer", description="Summarizes text.")

# Mid-level agent combining tools

research\_assistant = LlmAgent(

name="ResearchAssistant",

model="gemini-2.0-flash",

description="Finds and summarizes information on a topic.",

tools=[agent\_tool.AgentTool(agent=web\_searcher), agent\_tool.AgentTool(agent=summarizer)]

)

# High-level agent delegating research

report\_writer = LlmAgent(

name="ReportWriter",

model="gemini-2.0-flash",

instruction="Write a report on topic X. Use the ResearchAssistant to gather information.",

tools=[agent\_tool.AgentTool(agent=research\_assistant)]

# Alternatively, could use LLM Transfer if research\_assistant is a sub\_agent

)

# User interacts with ReportWriter.

# ReportWriter calls ResearchAssistant tool.

# ResearchAssistant calls WebSearch and Summarizer tools.

# Results flow back up.

Review/Critique Pattern (Generator-Critic)¶

**• Structure:** Typically involves two agents within a SequentialAgent: a Generator and a Critic/Reviewer.

**• Goal:** Improve the quality or validity of generated output by having a dedicated agent review it.

**• ADK Primitives Used:**

**• Workflow:** SequentialAgent ensures generation happens before review.

**• Communication:** **Shared Session State** (Generator uses output\_key to save output; Reviewer reads that state key). The Reviewer might save its feedback to another state key for subsequent steps.

**Python**

**Java**

# Conceptual Code: Generator-Critic

from google.adk.agents import SequentialAgent, LlmAgent

generator = LlmAgent(

name="DraftWriter",

instruction="Write a short paragraph about subject X.",

output\_key="draft\_text"

)

reviewer = LlmAgent(

name="FactChecker",

instruction="Review the text in state key 'draft\_text' for factual accuracy. Output 'valid' or 'invalid' with reasons.",

output\_key="review\_status"

)

# Optional: Further steps based on review\_status

review\_pipeline = SequentialAgent(

name="WriteAndReview",

sub\_agents=[generator, reviewer]

)

# generator runs -> saves draft to state['draft\_text']

# reviewer runs -> reads state['draft\_text'], saves status to state['review\_status']

Iterative Refinement Pattern¶

**• Structure:** Uses a LoopAgent containing one or more agents that work on a task over multiple iterations.

**• Goal:** Progressively improve a result (e.g., code, text, plan) stored in the session state until a quality threshold is met or a maximum number of iterations is reached.

**• ADK Primitives Used:**

**• Workflow:** LoopAgent manages the repetition.

**• Communication:** **Shared Session State** is essential for agents to read the previous iteration's output and save the refined version.

**• Termination:** The loop typically ends based on max\_iterations or a dedicated checking agent setting escalate=True in the Event Actions when the result is satisfactory.

**Python**

**Java**

# Conceptual Code: Iterative Code Refinement

from google.adk.agents import LoopAgent, LlmAgent, BaseAgent

from google.adk.events import Event, EventActions

from google.adk.agents.invocation\_context import InvocationContext

from typing import AsyncGenerator

# Agent to generate/refine code based on state['current\_code'] and state['requirements']

code\_refiner = LlmAgent(

name="CodeRefiner",

instruction="Read state['current\_code'] (if exists) and state['requirements']. Generate/refine Python code to meet requirements. Save to state['current\_code'].",

output\_key="current\_code" # Overwrites previous code in state

)

# Agent to check if the code meets quality standards

quality\_checker = LlmAgent(

name="QualityChecker",

instruction="Evaluate the code in state['current\_code'] against state['requirements']. Output 'pass' or 'fail'.",

output\_key="quality\_status"

)

# Custom agent to check the status and escalate if 'pass'

class CheckStatusAndEscalate(BaseAgent):

async def \_run\_async\_impl(self, ctx: InvocationContext) -> AsyncGenerator[Event, None]:

status = ctx.session.state.get("quality\_status", "fail")

should\_stop = (status == "pass")

yield Event(author=self.name, actions=EventActions(escalate=should\_stop))

refinement\_loop = LoopAgent(

name="CodeRefinementLoop",

max\_iterations=5,

sub\_agents=[code\_refiner, quality\_checker, CheckStatusAndEscalate(name="StopChecker")]

)

# Loop runs: Refiner -> Checker -> StopChecker

# State['current\_code'] is updated each iteration.

# Loop stops if QualityChecker outputs 'pass' (leading to StopChecker escalating) or after 5 iterations.

Human-in-the-Loop Pattern¶

**• Structure:** Integrates human intervention points within an agent workflow.

**• Goal:** Allow for human oversight, approval, correction, or tasks that AI cannot perform.

**• ADK Primitives Used (Conceptual):**

**• Interaction:** Can be implemented using a custom **Tool** that pauses execution and sends a request to an external system (e.g., a UI, ticketing system) waiting for human input. The tool then returns the human's response to the agent.

**• Workflow:** Could use **LLM-Driven Delegation** (transfer\_to\_agent) targeting a conceptual "Human Agent" that triggers the external workflow, or use the custom tool within an LlmAgent.

**• State/Callbacks:** State can hold task details for the human; callbacks can manage the interaction flow.

**• Note:** ADK doesn't have a built-in "Human Agent" type, so this requires custom integration.

**Python**

**Java**

# Conceptual Code: Using a Tool for Human Approval

from google.adk.agents import LlmAgent, SequentialAgent

from google.adk.tools import FunctionTool

# --- Assume external\_approval\_tool exists ---

# This tool would:

# 1. Take details (e.g., request\_id, amount, reason).

# 2. Send these details to a human review system (e.g., via API).

# 3. Poll or wait for the human response (approved/rejected).

# 4. Return the human's decision.

# async def external\_approval\_tool(amount: float, reason: str) -> str: ...

approval\_tool = FunctionTool(func=external\_approval\_tool)

# Agent that prepares the request

prepare\_request = LlmAgent(

name="PrepareApproval",

instruction="Prepare the approval request details based on user input. Store amount and reason in state.",

# ... likely sets state['approval\_amount'] and state['approval\_reason'] ...

)

# Agent that calls the human approval tool

request\_approval = LlmAgent(

name="RequestHumanApproval",

instruction="Use the external\_approval\_tool with amount from state['approval\_amount'] and reason from state['approval\_reason'].",

tools=[approval\_tool],

output\_key="human\_decision"

)

# Agent that proceeds based on human decision

process\_decision = LlmAgent(

name="ProcessDecision",

instruction="Check state key 'human\_decision'. If 'approved', proceed. If 'rejected', inform user."

)

approval\_workflow = SequentialAgent(

name="HumanApprovalWorkflow",

sub\_agents=[prepare\_request, request\_approval, process\_decision]

)

These patterns provide starting points for structuring your multi-agent systems. You can mix and match them as needed to create the most effective architecture for your specific application.

Using Different Models with ADK¶

**Note**

Java ADK currently supports Gemini and Anthropic models. More model support coming soon.

The Agent Development Kit (ADK) is designed for flexibility, allowing you to integrate various Large Language Models (LLMs) into your agents. While the setup for Google Gemini models is covered in the Setup Foundation Models guide, this page details how to leverage Gemini effectively and integrate other popular models, including those hosted externally or running locally.

ADK primarily uses two mechanisms for model integration:

**• Direct String / Registry:** For models tightly integrated with Google Cloud (like Gemini models accessed via Google AI Studio or Vertex AI) or models hosted on Vertex AI endpoints. You typically provide the model name or endpoint resource string directly to the LlmAgent. ADK's internal registry resolves this string to the appropriate backend client, often utilizing the google-genai library.

**• Wrapper Classes:** For broader compatibility, especially with models outside the Google ecosystem or those requiring specific client configurations (like models accessed via LiteLLM). You instantiate a specific wrapper class (e.g., LiteLlm) and pass this object as the model parameter to your LlmAgent.

The following sections guide you through using these methods based on your needs.

Using Google Gemini Models¶

This is the most direct way to use Google's flagship models within ADK.

**Integration Method:** Pass the model's identifier string directly to the model parameter of LlmAgent (or its alias, Agent).

**Backend Options & Setup:**

The google-genai library, used internally by ADK for Gemini, can connect through either Google AI Studio or Vertex AI.

**Model support for voice/video streaming**

In order to use voice/video streaming in ADK, you will need to use Gemini models that support the Live API. You can find the **model ID(s)** that support the Gemini Live API in the documentation:

• Google AI Studio: Gemini Live API

• Vertex AI: Gemini Live API

Google AI Studio¶

**• Use Case:** Google AI Studio is the easiest way to get started with Gemini. All you need is the API key. Best for rapid prototyping and development.

**• Setup:** Typically requires an API key:

• Set as an environment variable or

• Passed during the model initialization via the Client (see example below)

export GOOGLE\_API\_KEY="YOUR\_GOOGLE\_API\_KEY"

export GOOGLE\_GENAI\_USE\_VERTEXAI=FALSE

**• Models:** Find all available models on the Google AI for Developers site.

Vertex AI¶

**• Use Case:** Recommended for production applications, leveraging Google Cloud infrastructure. Gemini on Vertex AI supports enterprise-grade features, security, and compliance controls.

**• Setup:**

• Authenticate using Application Default Credentials (ADC):

gcloud auth application-default login

•

• Configure these variables either as environment variables or by providing them directly when initializing the Model.  
Set your Google Cloud project and location:

export GOOGLE\_CLOUD\_PROJECT="YOUR\_PROJECT\_ID"

•

export GOOGLE\_CLOUD\_LOCATION="YOUR\_VERTEX\_AI\_LOCATION" # e.g., us-central1

•

• Explicitly tell the library to use Vertex AI:

export GOOGLE\_GENAI\_USE\_VERTEXAI=TRUE

•

**• Models:** Find available model IDs in the Vertex AI documentation.

**Example:**

**Python**

**Java**

from google.adk.agents import LlmAgent

# --- Example using a stable Gemini Flash model ---

agent\_gemini\_flash = LlmAgent(

# Use the latest stable Flash model identifier

model="gemini-2.0-flash",

name="gemini\_flash\_agent",

instruction="You are a fast and helpful Gemini assistant.",

# ... other agent parameters

)

# --- Example using a powerful Gemini Pro model ---

# Note: Always check the official Gemini documentation for the latest model names,

# including specific preview versions if needed. Preview models might have

# different availability or quota limitations.

agent\_gemini\_pro = LlmAgent(

# Use the latest generally available Pro model identifier

model="gemini-2.5-pro-preview-03-25",

name="gemini\_pro\_agent",

instruction="You are a powerful and knowledgeable Gemini assistant.",

# ... other agent parameters

)

Using Anthropic models¶

You can integrate Anthropic's Claude models directly using their API key or from a Vertex AI backend into your Java ADK applications by using the ADK's Claude wrapper class.

For Vertex AI backend, see the Third-Party Models on Vertex AI section.

**Prerequisites:**

**• Dependencies:**

**• Anthropic SDK Classes (Transitive):** The Java ADK's com.google.adk.models.Claude wrapper relies on classes from Anthropic's official Java SDK. These are typically included as **transitive dependencies**.

**• Anthropic API Key:**

• Obtain an API key from Anthropic. Securely manage this key using a secret manager.

**Integration:**

Instantiate com.google.adk.models.Claude, providing the desired Claude model name and an AnthropicOkHttpClient configured with your API key. Then, pass this Claude instance to your LlmAgent.

**Example:**

import com.anthropic.client.AnthropicClient;

import com.google.adk.agents.LlmAgent;

import com.google.adk.models.Claude;

import com.anthropic.client.okhttp.AnthropicOkHttpClient; // From Anthropic's SDK

public class DirectAnthropicAgent {

private static final String CLAUDE\_MODEL\_ID = "claude-3-7-sonnet-latest"; // Or your preferred Claude model

public static LlmAgent createAgent() {

// It's recommended to load sensitive keys from a secure config

AnthropicClient anthropicClient = AnthropicOkHttpClient.builder()

.apiKey("ANTHROPIC\_API\_KEY")

.build();

Claude claudeModel = new Claude(

CLAUDE\_MODEL\_ID,

anthropicClient

);

return LlmAgent.builder()

.name("claude\_direct\_agent")

.model(claudeModel)

.instruction("You are a helpful AI assistant powered by Anthropic Claude.")

// ... other LlmAgent configurations

.build();

}

public static void main(String[] args) {

try {

LlmAgent agent = createAgent();

System.out.println("Successfully created direct Anthropic agent: " + agent.name());

} catch (IllegalStateException e) {

System.err.println("Error creating agent: " + e.getMessage());

}

}

}

Using Cloud & Proprietary Models via LiteLLM¶

To access a vast range of LLMs from providers like OpenAI, Anthropic (non-Vertex AI), Cohere, and many others, ADK offers integration through the LiteLLM library.

**Integration Method:** Instantiate the LiteLlm wrapper class and pass it to the model parameter of LlmAgent.

**LiteLLM Overview:** LiteLLM acts as a translation layer, providing a standardized, OpenAI-compatible interface to over 100+ LLMs.

**Setup:**

**• Install LiteLLM:**

pip install litellm

•

**• Set Provider API Keys:** Configure API keys as environment variables for the specific providers you intend to use.

*• Example for OpenAI:*

export OPENAI\_API\_KEY="YOUR\_OPENAI\_API\_KEY"

•

*• Example for Anthropic (non-Vertex AI):*

export ANTHROPIC\_API\_KEY="YOUR\_ANTHROPIC\_API\_KEY"

•

*• Consult the LiteLLM Providers Documentation for the correct environment variable names for other providers.***Example:**

from google.adk.agents import LlmAgent

•

from google.adk.models.lite\_llm import LiteLlm

•

•

# --- Example Agent using OpenAI's GPT-4o ---

•

# (Requires OPENAI\_API\_KEY)

•

agent\_openai = LlmAgent(

•

model=LiteLlm(model="openai/gpt-4o"), # LiteLLM model string format

•

name="openai\_agent",

•

instruction="You are a helpful assistant powered by GPT-4o.",

•

# ... other agent parameters

•

)

•

•

# --- Example Agent using Anthropic's Claude Haiku (non-Vertex) ---

•

# (Requires ANTHROPIC\_API\_KEY)

•

agent\_claude\_direct = LlmAgent(

•

model=LiteLlm(model="anthropic/claude-3-haiku-20240307"),

•

name="claude\_direct\_agent",

•

instruction="You are an assistant powered by Claude Haiku.",

•

# ... other agent parameters

•

)

•

Using Open & Local Models via LiteLLM¶

For maximum control, cost savings, privacy, or offline use cases, you can run open-source models locally or self-host them and integrate them using LiteLLM.

**Integration Method:** Instantiate the LiteLlm wrapper class, configured to point to your local model server.

Ollama Integration¶

Ollama allows you to easily run open-source models locally.

**Model choice¶**

If your agent is relying on tools, please make sure that you select a model with tool support from Ollama website.

For reliable results, we recommend using a decent-sized model with tool support.

The tool support for the model can be checked with the following command:

ollama show mistral-small3.1

Model

architecture mistral3

parameters 24.0B

context length 131072

embedding length 5120

quantization Q4\_K\_M

Capabilities

completion

vision

tools

You are supposed to see tools listed under capabilities.

You can also look at the template the model is using and tweak it based on your needs.

ollama show --modelfile llama3.2 > model\_file\_to\_modify

For instance, the default template for the above model inherently suggests that the model shall call a function all the time. This may result in an infinite loop of function calls.

Given the following functions, please respond with a JSON for a function call

with its proper arguments that best answers the given prompt.

Respond in the format {"name": function name, "parameters": dictionary of

argument name and its value}. Do not use variables.

You can swap such prompts with a more descriptive one to prevent infinite tool call loops.

For instance:

Review the user's prompt and the available functions listed below.

First, determine if calling one of these functions is the most appropriate way to respond. A function call is likely needed if the prompt asks for a specific action, requires external data lookup, or involves calculations handled by the functions. If the prompt is a general question or can be answered directly, a function call is likely NOT needed.

If you determine a function call IS required: Respond ONLY with a JSON object in the format {"name": "function\_name", "parameters": {"argument\_name": "value"}}. Ensure parameter values are concrete, not variables.

If you determine a function call IS NOT required: Respond directly to the user's prompt in plain text, providing the answer or information requested. Do not output any JSON.

Then you can create a new model with the following command:

ollama create llama3.2-modified -f model\_file\_to\_modify

**Using ollama\_chat provider¶**

Our LiteLLM wrapper can be used to create agents with Ollama models.

root\_agent = Agent(

model=LiteLlm(model="ollama\_chat/mistral-small3.1"),

name="dice\_agent",

description=(

"hello world agent that can roll a dice of 8 sides and check prime"

" numbers."

),

instruction="""

You roll dice and answer questions about the outcome of the dice rolls.

""",

tools=[

roll\_die,

check\_prime,

],

)

**It is important to set the provider ollama\_chat instead of ollama. Using ollama will result in unexpected behaviors such as infinite tool call loops and ignoring previous context.**

While api\_base can be provided inside LiteLLM for generation, LiteLLM library is calling other APIs relying on the env variable instead as of v1.65.5 after completion. So at this time, we recommend setting the env variable OLLAMA\_API\_BASE to point to the ollama server.

export OLLAMA\_API\_BASE="http://localhost:11434"

adk web

**Using openai provider¶**

Alternatively, openai can be used as the provider name. But this will also require setting the OPENAI\_API\_BASE=http://localhost:11434/v1 and OPENAI\_API\_KEY=anything env variables instead of OLLAMA\_API\_BASE. **Please note that api base now has /v1 at the end.**

root\_agent = Agent(

model=LiteLlm(model="openai/mistral-small3.1"),

name="dice\_agent",

description=(

"hello world agent that can roll a dice of 8 sides and check prime"

" numbers."

),

instruction="""

You roll dice and answer questions about the outcome of the dice rolls.

""",

tools=[

roll\_die,

check\_prime,

],

)

export OPENAI\_API\_BASE=http://localhost:11434/v1

export OPENAI\_API\_KEY=anything

adk web

**Debugging¶**

You can see the request sent to the Ollama server by adding the following in your agent code just after imports.

import litellm

litellm.\_turn\_on\_debug()

Look for a line like the following:

Request Sent from LiteLLM:

curl -X POST \

http://localhost:11434/api/chat \

-d '{'model': 'mistral-small3.1', 'messages': [{'role': 'system', 'content': ...

Self-Hosted Endpoint (e.g., vLLM)¶

Tools such as vLLM allow you to host models efficiently and often expose an OpenAI-compatible API endpoint.

**Setup:**

**• Deploy Model:** Deploy your chosen model using vLLM (or a similar tool). Note the API base URL (e.g., https://your-vllm-endpoint.run.app/v1).

*• Important for ADK Tools:* When deploying, ensure the serving tool supports and enables OpenAI-compatible tool/function calling. For vLLM, this might involve flags like --enable-auto-tool-choice and potentially a specific --tool-call-parser, depending on the model. Refer to the vLLM documentation on Tool Use.

**• Authentication:** Determine how your endpoint handles authentication (e.g., API key, bearer token).  
**Integration Example:**

import subprocess

•

from google.adk.agents import LlmAgent

•

from google.adk.models.lite\_llm import LiteLlm

•

•

# --- Example Agent using a model hosted on a vLLM endpoint ---

•

•

# Endpoint URL provided by your vLLM deployment

•

api\_base\_url = "https://your-vllm-endpoint.run.app/v1"

•

•

# Model name as recognized by \*your\* vLLM endpoint configuration

•

model\_name\_at\_endpoint = "hosted\_vllm/google/gemma-3-4b-it" # Example from vllm\_test.py

•

•

# Authentication (Example: using gcloud identity token for a Cloud Run deployment)

•

# Adapt this based on your endpoint's security

•

try:

•

gcloud\_token = subprocess.check\_output(

•

["gcloud", "auth", "print-identity-token", "-q"]

•

).decode().strip()

•

auth\_headers = {"Authorization": f"Bearer {gcloud\_token}"}

•

except Exception as e:

•

print(f"Warning: Could not get gcloud token - {e}. Endpoint might be unsecured or require different auth.")

•

auth\_headers = None # Or handle error appropriately

•

•

agent\_vllm = LlmAgent(

•

model=LiteLlm(

•

model=model\_name\_at\_endpoint,

•

api\_base=api\_base\_url,

•

# Pass authentication headers if needed

•

extra\_headers=auth\_headers

•

# Alternatively, if endpoint uses an API key:

•

# api\_key="YOUR\_ENDPOINT\_API\_KEY"

•

),

•

name="vllm\_agent",

•

instruction="You are a helpful assistant running on a self-hosted vLLM endpoint.",

•

# ... other agent parameters

•

)

•

Using Hosted & Tuned Models on Vertex AI¶

For enterprise-grade scalability, reliability, and integration with Google Cloud's MLOps ecosystem, you can use models deployed to Vertex AI Endpoints. This includes models from Model Garden or your own fine-tuned models.

**Integration Method:** Pass the full Vertex AI Endpoint resource string (projects/PROJECT\_ID/locations/LOCATION/endpoints/ENDPOINT\_ID) directly to the model parameter of LlmAgent.

**Vertex AI Setup (Consolidated):**

Ensure your environment is configured for Vertex AI:

**• Authentication:** Use Application Default Credentials (ADC):

gcloud auth application-default login

•

**• Environment Variables:** Set your project and location:

export GOOGLE\_CLOUD\_PROJECT="YOUR\_PROJECT\_ID"

•

export GOOGLE\_CLOUD\_LOCATION="YOUR\_VERTEX\_AI\_LOCATION" # e.g., us-central1

•

**• Enable Vertex Backend:** Crucially, ensure the google-genai library targets Vertex AI:

export GOOGLE\_GENAI\_USE\_VERTEXAI=TRUE

•

Model Garden Deployments¶

You can deploy various open and proprietary models from the Vertex AI Model Garden to an endpoint.

**Example:**

from google.adk.agents import LlmAgent

from google.genai import types # For config objects

# --- Example Agent using a Llama 3 model deployed from Model Garden ---

# Replace with your actual Vertex AI Endpoint resource name

llama3\_endpoint = "projects/YOUR\_PROJECT\_ID/locations/us-central1/endpoints/YOUR\_LLAMA3\_ENDPOINT\_ID"

agent\_llama3\_vertex = LlmAgent(

model=llama3\_endpoint,

name="llama3\_vertex\_agent",

instruction="You are a helpful assistant based on Llama 3, hosted on Vertex AI.",

generate\_content\_config=types.GenerateContentConfig(max\_output\_tokens=2048),

# ... other agent parameters

)

Fine-tuned Model Endpoints¶

Deploying your fine-tuned models (whether based on Gemini or other architectures supported by Vertex AI) results in an endpoint that can be used directly.

**Example:**

from google.adk.agents import LlmAgent

# --- Example Agent using a fine-tuned Gemini model endpoint ---

# Replace with your fine-tuned model's endpoint resource name

finetuned\_gemini\_endpoint = "projects/YOUR\_PROJECT\_ID/locations/us-central1/endpoints/YOUR\_FINETUNED\_ENDPOINT\_ID"

agent\_finetuned\_gemini = LlmAgent(

model=finetuned\_gemini\_endpoint,

name="finetuned\_gemini\_agent",

instruction="You are a specialized assistant trained on specific data.",

# ... other agent parameters

)

Third-Party Models on Vertex AI (e.g., Anthropic Claude)¶

Some providers, like Anthropic, make their models available directly through Vertex AI.

**Python**

**Java**

**Integration Method:** Uses the direct model string (e.g., "claude-3-sonnet@20240229"), *but requires manual registration* within ADK.

**Why Registration?** ADK's registry automatically recognizes gemini-\* strings and standard Vertex AI endpoint strings (projects/.../endpoints/...) and routes them via the google-genai library. For other model types used directly via Vertex AI (like Claude), you must explicitly tell the ADK registry which specific wrapper class (Claude in this case) knows how to handle that model identifier string with the Vertex AI backend.

**Setup:**

**• Vertex AI Environment:** Ensure the consolidated Vertex AI setup (ADC, Env Vars, GOOGLE\_GENAI\_USE\_VERTEXAI=TRUE) is complete.

**• Install Provider Library:** Install the necessary client library configured for Vertex AI.

pip install "anthropic[vertex]"

•

**• Register Model Class:** Add this code near the start of your application, *before* creating an agent using the Claude model string:

# Required for using Claude model strings directly via Vertex AI with LlmAgent

•

from google.adk.models.anthropic\_llm import Claude

•

from google.adk.models.registry import LLMRegistry

•

•

LLMRegistry.register(Claude)

•

**Example:**

from google.adk.agents import LlmAgent

from google.adk.models.anthropic\_llm import Claude # Import needed for registration

from google.adk.models.registry import LLMRegistry # Import needed for registration

from google.genai import types

# --- Register Claude class (do this once at startup) ---

LLMRegistry.register(Claude)

# --- Example Agent using Claude 3 Sonnet on Vertex AI ---

# Standard model name for Claude 3 Sonnet on Vertex AI

claude\_model\_vertexai = "claude-3-sonnet@20240229"

agent\_claude\_vertexai = LlmAgent(

model=claude\_model\_vertexai, # Pass the direct string after registration

name="claude\_vertexai\_agent",

instruction="You are an assistant powered by Claude 3 Sonnet on Vertex AI.",

generate\_content\_config=types.GenerateContentConfig(max\_output\_tokens=4096),

# ... other agent parameters

)