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

```
# 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
```

Java

```
// Conceptual Example: Defining Hierarchy
import com.google.adk.agents.SequentialAgent;
import com.google.adk.agents.LlmAgent;

// Define individual agents
LlmAgent greeter = LlmAgent.builder().name("Greeter").model("gemini-2.0-
flash").build();
SequentialAgent taskDoer =
SequentialAgent.builder().name("TaskExecutor").subAgents(...).build(); //
Sequential Agent

// Create parent agent and assign sub_agents
LlmAgent coordinator = LlmAgent.builder()
    .name("Coordinator")
    .model("gemini-2.0-flash")
    .description("I coordinate greetings and tasks")
    .subAgents(greeter, taskDoer) // Assign sub_agents here
    .build();

// Framework automatically sets:
```

```
// assert greeter.parentAgent().equals(coordinator);
// assert taskDoer.parentAgent().equals(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

```
# 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.
```

Java

```
// Conceptual Example: Sequential Pipeline
import com.google.adk.agents.SequentialAgent;
import com.google.adk.agents.LlmAgent;

LlmAgent step1 =
LlmAgent.builder().name("Step1_Fetch").outputKey("data").build(); // Saves
output to state.get("data")
LlmAgent step2 =
LlmAgent.builder().name("Step2_Process").instruction("Process data from
state key 'data'.").build();

SequentialAgent pipeline =
SequentialAgent.builder().name("MyPipeline").subAgents(step1,
step2).build();
// When pipeline runs, Step2 can access the state.get("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

```python
# 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'].
```

Java

```java
// Conceptual Example: Parallel Execution
import com.google.adk.agents.LlmAgent;
import com.google.adk.agents.ParallelAgent;

LlmAgent fetchWeather = LlmAgent.builder()
    .name("WeatherFetcher")
    .outputKey("weather")
    .build();

LlmAgent fetchNews = LlmAgent.builder()
    .name("NewsFetcher")
    .instruction("news")
    .build();

ParallelAgent gatherer = ParallelAgent.builder()
    .name("InfoGatherer")
    .subAgents(fetchWeather, fetchNews)
    .build();

// 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

```python
# 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.
```

Java

```
// Conceptual Example: Loop with Condition
// Custom agent to check state and potentially escalate
public static class CheckConditionAgent extends BaseAgent {
  public CheckConditionAgent(String name, String description) {
    super(name, description, List.of(), null, null);
  }

  @Override
  protected Flowable<Event> runAsyncImpl(InvocationContext ctx) {
    String status = (String) ctx.session().state().getOrDefault("status", "pending");
    boolean isDone = "completed".equalsIgnoreCase(status);

    // Emit an event that signals to escalate (exit the loop) if the condition is met.
    // If not done, the escalate flag will be false or absent, and the loop continues.
    Event checkEvent = Event.builder()
            .author(name())
            .id(Event.generateEventId()) // Important to give events unique IDs
            .actions(EventActions.builder().escalate(isDone).build()) // Escalate if done
            .build();
    return Flowable.just(checkEvent);
  }
}

// Agent that might update state.put("status")
LlmAgent processingStepAgent = LlmAgent.builder().name("ProcessingStep").build();
// Custom agent instance for checking the condition
CheckConditionAgent conditionCheckerAgent = new CheckConditionAgent(
    "ConditionChecker",
    "Checks if the status is 'completed'."
);
LoopAgent poller =
LoopAgent.builder().name("StatusPoller").maxIterations(10).subAgents(processingStepAgent,
conditionCheckerAgent).build();
// When poller runs, it executes processingStepAgent then conditionCheckerAgent
repeatedly
// until Checker escalates (state.get("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

```python
# 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".
```

Java

```java
// Conceptual Example: Using outputKey and reading state
import com.google.adk.agents.LlmAgent;
import com.google.adk.agents.SequentialAgent;

LlmAgent agentA = LlmAgent.builder()
    .name("AgentA")
    .instruction("Find the capital of France.")
    .outputKey("capital_city")
    .build();

LlmAgent agentB = LlmAgent.builder()
    .name("AgentB")
    .instruction("Tell me about the city stored in state key
'capital_city'.")
    .outputKey("capital_city")
    .build();

SequentialAgent pipeline =
SequentialAgent.builder().name("CityInfo").subAgents(agentA,
```

```
agentB).build();
// AgentA runs, saves "Paris" to state('capital_city').
// AgentB runs, its instruction processor reads state.get("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 `description`s 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

```python
# 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.
```

Java

```java
// Conceptual Setup: LLM Transfer
import com.google.adk.agents.LlmAgent;

LlmAgent bookingAgent = LlmAgent.builder()
    .name("Booker")
    .description("Handles flight and hotel bookings.")
    .build();
```

```
LlmAgent infoAgent = LlmAgent.builder()
    .name("Info")
    .description("Provides general information and answers questions.")
    .build();

// Define the coordinator agent
LlmAgent coordinator = LlmAgent.builder()
    .name("Coordinator")
    .model("gemini-2.0-flash") // Or your desired model
    .instruction("You are an assistant. Delegate booking tasks to Booker and
info requests to Info.")
    .description("Main coordinator.")
    // AutoFlow will be used by default (implicitly) because subAgents are
present
    // and transfer is not disallowed.
    .subAgents(bookingAgent, infoAgent)
    .build();

// If coordinator receives "Book a flight", its LLM should generate:
//
FunctionCall.builder.name("transferToAgent").args(ImmutableMap.of("agent_name",
"Booker")).build()
// ADK framework then routes execution to bookingAgent.
```

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

```
# 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.
```

Java

```
// Conceptual Setup: Agent as a Tool
import com.google.adk.agents.BaseAgent;
import com.google.adk.agents.LlmAgent;
import com.google.adk.tools.AgentTool;

// Example custom agent (could be LlmAgent or custom BaseAgent)
public class ImageGeneratorAgent extends BaseAgent  {

  public ImageGeneratorAgent(String name, String description) {
    super(name, description, List.of(), null, null);
  }

  // ... internal logic ...
  @Override
  protected Flowable<Event> runAsyncImpl(InvocationContext
invocationContext) { // Simplified run logic
    invocationContext.session().state().get("image_prompt");
    // Generate image bytes
    // ...

    Event responseEvent = Event.builder()
        .author(this.name())
        .content(Content.fromParts(Part.fromText("\b...")))
        .build();

    return Flowable.just(responseEvent);
  }

  @Override
  protected Flowable<Event> runLiveImpl(InvocationContext
invocationContext) {
    return null;
  }
}

// Wrap the agent using AgentTool
```

```
ImageGeneratorAgent imageAgent = new ImageGeneratorAgent("image_agent",
"generates images");
AgentTool imageTool = AgentTool.create(imageAgent);

// Parent agent uses the AgentTool
LlmAgent artistAgent = LlmAgent.builder()
        .name("Artist")
        .model("gemini-2.0-flash")
        .instruction(
                "You are an artist. Create a detailed prompt for an image
and then " +
                        "use the 'ImageGen' tool to generate the image. "
+
                        "The 'ImageGen' tool expects a single string
argument named 'request' " +
                        "containing the image prompt. The tool will return
a JSON string in its " +
                        "'result' field, containing 'image_base64',
'mime_type', and 'status'."
        )
        .description("An agent that can create images using a generation
tool.")
        .tools(imageTool) // Include the AgentTool
        .build();

// Artist LLM generates a prompt, then calls:
// FunctionCall(name='ImageGen', args={'imagePrompt': 'a cat wearing a
hat'})
// Framework calls imageTool.runAsync(...), 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 `description`s on sub-agents and appropriate `instruction` on Coordinator) or **Explicit Invocation (`AgentTool`)** (Coordinator includes `AgentTool`-wrapped specialists in its `tools`).

Python

```python
# 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')
```

Java

```java
// Conceptual Code: Coordinator using LLM Transfer
import com.google.adk.agents.LlmAgent;

LlmAgent billingAgent = LlmAgent.builder()
    .name("Billing")
    .description("Handles billing inquiries and payment issues.")
    .build();

LlmAgent supportAgent = LlmAgent.builder()
    .name("Support")
    .description("Handles technical support requests and login problems.")
    .build();

LlmAgent coordinator = LlmAgent.builder()
    .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.")
    .subAgents(billingAgent, supportAgent)
    // Agent transfer is implicit with sub agents in the Autoflow, unless
specified
    // using .disallowTransferToParent or disallowTransferToPeers
    .build();

// User asks "My payment failed" -> Coordinator's LLM should call
// transferToAgent(agentName='Billing')
// User asks "I can't log in" -> Coordinator's LLM should call
// transferToAgent(agentName='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

```python
# 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']
```

Java

```java
// Conceptual Code: Sequential Data Pipeline
import com.google.adk.agents.SequentialAgent;

LlmAgent validator = LlmAgent.builder()
    .name("ValidateInput")
    .instruction("Validate the input")
    .outputKey("validation_status") // Saves its main text output to
session.state["validation_status"]
    .build();

LlmAgent processor = LlmAgent.builder()
    .name("ProcessData")
    .instruction("Process data if state key 'validation_status' is
'valid'")
    .outputKey("result") // Saves its main text output to
session.state["result"]
    .build();

LlmAgent reporter = LlmAgent.builder()
    .name("ReportResult")
    .instruction("Report the result from state key 'result'")
    .build();
```

```
SequentialAgent dataPipeline = SequentialAgent.builder()
    .name("DataPipeline")
    .subAgents(validator, processor, reporter)
    .build();

// 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

```
# 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'].
```

Java

```
// Conceptual Code: Parallel Information Gathering
import com.google.adk.agents.LlmAgent;
import com.google.adk.agents.ParallelAgent;
import com.google.adk.agents.SequentialAgent;

LlmAgent fetchApi1 = LlmAgent.builder()
    .name("API1Fetcher")
    .instruction("Fetch data from API 1.")
    .outputKey("api1_data")
    .build();

LlmAgent fetchApi2 = LlmAgent.builder()
    .name("API2Fetcher")
    .instruction("Fetch data from API 2.")
    .outputKey("api2_data")
    .build();

ParallelAgent gatherConcurrently = ParallelAgent.builder()
    .name("ConcurrentFetcher")
    .subAgents(fetchApi2, fetchApi1)
    .build();

LlmAgent synthesizer = LlmAgent.builder()
    .name("Synthesizer")
    .instruction("Combine results from state keys 'api1_data' and
'api2_data'.")
    .build();

SequentialAgent overallWorfklow = SequentialAgent.builder()
    .name("FetchAndSynthesize") // Run parallel fetch, then synthesize
    .subAgents(gatherConcurrently, synthesizer)
    .build();

// 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

```python
# 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.
```

Java

```java
// Conceptual Code: Hierarchical Research Task
import com.google.adk.agents.LlmAgent;
import com.google.adk.tools.AgentTool;

// Low-level tool-like agents
LlmAgent webSearcher = LlmAgent.builder()
    .name("WebSearch")
    .description("Performs web searches for facts.")
    .build();

LlmAgent summarizer = LlmAgent.builder()
    .name("Summarizer")
    .description("Summarizes text.")
    .build();

// Mid-level agent combining tools
LlmAgent researchAssistant = LlmAgent.builder()
    .name("ResearchAssistant")
    .model("gemini-2.0-flash")
    .description("Finds and summarizes information on a topic.")
    .tools(AgentTool.create(webSearcher), AgentTool.create(summarizer))
    .build();

// High-level agent delegating research
LlmAgent reportWriter = LlmAgent.builder()
    .name("ReportWriter")
```

```
    .model("gemini-2.0-flash")
    .instruction("Write a report on topic X. Use the ResearchAssistant to
gather information.")
    .tools(AgentTool.create(researchAssistant))
    // Alternatively, could use LLM Transfer if research_assistant is a
subAgent
    .build();

// 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

```
# 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']
```

Java

```
// Conceptual Code: Generator-Critic
import com.google.adk.agents.LlmAgent;
import com.google.adk.agents.SequentialAgent;

LlmAgent generator = LlmAgent.builder()
    .name("DraftWriter")
    .instruction("Write a short paragraph about subject X.")
    .outputKey("draft_text")
    .build();

LlmAgent reviewer = LlmAgent.builder()
    .name("FactChecker")
    .instruction("Review the text in state key 'draft_text' for factual
accuracy. Output 'valid' or 'invalid' with reasons.")
    .outputKey("review_status")
    .build();

// Optional: Further steps based on review_status

SequentialAgent reviewPipeline = SequentialAgent.builder()
    .name("WriteAndReview")
    .subAgents(generator, reviewer)
    .build();

// 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

```
# 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.
```

Java

```
// Conceptual Code: Iterative Code Refinement
import com.google.adk.agents.BaseAgent;
import com.google.adk.agents.LlmAgent;
import com.google.adk.agents.LoopAgent;
import com.google.adk.events.Event;
import com.google.adk.events.EventActions;
import com.google.adk.agents.InvocationContext;
import io.reactivex.rxjava3.core.Flowable;
import java.util.List;

// Agent to generate/refine code based on state['current_code'] and
state['requirements']
LlmAgent codeRefiner = LlmAgent.builder()
    .name("CodeRefiner")
    .instruction("Read state['current_code'] (if exists) and
state['requirements']. Generate/refine Java code to meet requirements.
Save to state['current_code'].")
    .outputKey("current_code") // Overwrites previous code in state
    .build();
```

```
// Agent to check if the code meets quality standards
LlmAgent qualityChecker = LlmAgent.builder()
    .name("QualityChecker")
    .instruction("Evaluate the code in state['current_code'] against
state['requirements']. Output 'pass' or 'fail'.")
    .outputKey("quality_status")
    .build();

BaseAgent checkStatusAndEscalate = new BaseAgent(
    "StopChecker","Checks quality_status and escalates if 'pass'.",
List.of(), null, null) {

  @Override
  protected Flowable<Event> runAsyncImpl(InvocationContext
invocationContext) {
    String status = (String)
invocationContext.session().state().getOrDefault("quality_status",
"fail");
    boolean shouldStop = "pass".equals(status);

    EventActions actions =
EventActions.builder().escalate(shouldStop).build();
    Event event = Event.builder()
        .author(this.name())
        .actions(actions)
        .build();
    return Flowable.just(event);
  }
};

LoopAgent refinementLoop = LoopAgent.builder()
    .name("CodeRefinementLoop")
    .maxIterations(5)
    .subAgents(codeRefiner, qualityChecker, checkStatusAndEscalate)
    .build();

// 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

```python
# 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]
)
```

Java

```java
// Conceptual Code: Using a Tool for Human Approval
import com.google.adk.agents.LlmAgent;
import com.google.adk.agents.SequentialAgent;
import com.google.adk.tools.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.
// public boolean externalApprovalTool(float amount, String reason) { ...
}
FunctionTool approvalTool = FunctionTool.create(externalApprovalTool);

// Agent that prepares the request
LlmAgent prepareRequest = LlmAgent.builder()
    .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'] ...
    .build();

// Agent that calls the human approval tool
LlmAgent requestApproval = LlmAgent.builder()
    .name("RequestHumanApproval")
    .instruction("Use the external_approval_tool with amount from
state['approval_amount'] and reason from state['approval_reason'].")
    .tools(approvalTool)
    .outputKey("human_decision")
    .build();

// Agent that proceeds based on human decision
LlmAgent processDecision = LlmAgent.builder()
    .name("ProcessDecision")
    .instruction("Check state key 'human_decision'. If 'approved',
proceed. If 'rejected', inform user.")
    .build();

SequentialAgent approvalWorkflow = SequentialAgent.builder()
    .name("HumanApprovalWorkflow")
    .subAgents(prepareRequest, requestApproval, processDecision)
    .build();
```

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.