Licensed under the Apache License, Version 2.0 (the "License");

🚀 Multi-Agent Systems & Workflow Patterns

Welcome to the Kaggle 5-day Agents course!

In the previous notebook, you built a **single agent** that could take action. Now, you'll learn how to scale up by building **agent teams**.

Just like a team of people, you can create specialized agents that collaborate to solve complex problems. This is called a **multi-agent system**, and it's one of the most powerful concepts in AI agent development.

In this notebook, you'll:

- ✅ Learn when to use multi-agent systems in Agent Development Kit (ADK)
- ✅ Build your first system using an LLM as a "manager"
- ✅ Learn three core workflow patterns (Sequential, Parallel, and Loop) to coordinate your agent teams

‼️ Please Read

> ❌ ℹ️ **Note: No submission required!** This notebook is for your hands-on practice and learning only. You **do not** need to submit it anywhere to complete the course.

> ⏸️ **Note:** When you first start the notebook via running a cell you might see a banner in the notebook header that reads **"Waiting for the next available notebook"**. The queue should drop rapidly; however, during peak bursts you might have to wait a few minutes.

> ❌ **Note:** Avoid using the **Run all** cells command as this can trigger a QPM limit resulting in 429 errors when calling the backing model. Suggested flow is to run each cell in order - one at a time. See FAQ on 429 errors for more information.

For help: Ask questions on the Kaggle Discord server.

📖 Get started with Kaggle Notebooks

If this is your first time using Kaggle Notebooks, welcome! You can learn more about using Kaggle Notebooks in the documentation.

Here's how to get started:

1. Verify Your Account (Required)

To use the Kaggle Notebooks in this course, you'll need to verify your account with a phone number.

You can do this in your Kaggle settings.

2. Make Your Own Copy

To run any code in this notebook, you first need your own editable copy.

Click the `Copy and Edit` button in the top-right corner.

This creates a private copy of the notebook just for you.

3. Run Code Cells

Once you have your copy, you can run code.

Click the ▶️ Run button next to any code cell to execute it.

Run the cells in order from top to bottom.

4. If You Get Stuck

To restart: Select `Factory reset` from the `Run` menu.

For help: Ask questions on the Kaggle Discord server.

Section 1: Setup

1.1: Install dependencies

The Kaggle Notebooks environment includes a pre-installed version of the google-adk library for Python and its required dependencies, so you don't need to install additional packages in this notebook.

To install and use ADK in your own Python development environment outside of this course, you can do so by running:

```
pip install google-adk
```

1.2: Configure your Gemini API Key

This notebook uses the Gemini API, which requires authentication.

1. Get your API key

If you don't have one already, create an API key in Google AI Studio.

2. Add the key to Kaggle Secrets

Next, you will need to add your API key to your Kaggle Notebook as a Kaggle User Secret.

1. In the top menu bar of the notebook editor, select `Add-ons` then `Secrets`.
2. Create a new secret with the label `GOOGLE_API_KEY`.
3. Paste your API key into the "Value" field and click "Save".
4. Ensure that the checkbox next to `GOOGLE_API_KEY` is selected so that the secret is attached to the notebook.

3. Authenticate in the notebook

Run the cell below to complete authentication.

```
import os
from kaggle_secrets import UserSecretsClient

try:
    GOOGLE_API_KEY = UserSecretsClient().get_secret("GOOGLE_API_KEY")
    os.environ["GOOGLE_API_KEY"] = GOOGLE_API_KEY
    print("✅ Gemini API key setup complete.")
except Exception as e:
```

```
    print(
        f"    Authentication Error: Please make sure you have added 'GOOGLE_API_KEY' to your Kaggle secrets. Details: {e}"
    )
```

1.3: Import ADK components

Now, import the specific components you'll need from the Agent Development Kit and the Generative AI library. This keeps your code organized and ensures we have access to the necessary building blocks.

```
from google.adk.agents import Agent, SequentialAgent, ParallelAgent, LoopAgent
from google.adk.models.google_llm import Gemini
from google.adk.runners import InMemoryRunner
from google.adk.tools import AgentTool, FunctionTool, google_search
from google.genai import types

print("✅ ADK components imported successfully.")
```

1.4: Configure Retry Options

When working with LLMs, you may encounter transient errors like rate limits or temporary service unavailability. Retry options automatically handle these failures by retrying the request with exponential backoff.

```
retry_config=types.HttpRetryOptions(
    attempts=5,  # Maximum retry attempts
    exp_base=7,  # Delay multiplier
    initial_delay=1,
    http_status_codes=[429, 500, 503, 504], # Retry on these HTTP errors
)
```

🤔 Section 2: Why Multi-Agent Systems? + Your First Multi-Agent

The Problem: The "Do-It-All" Agent

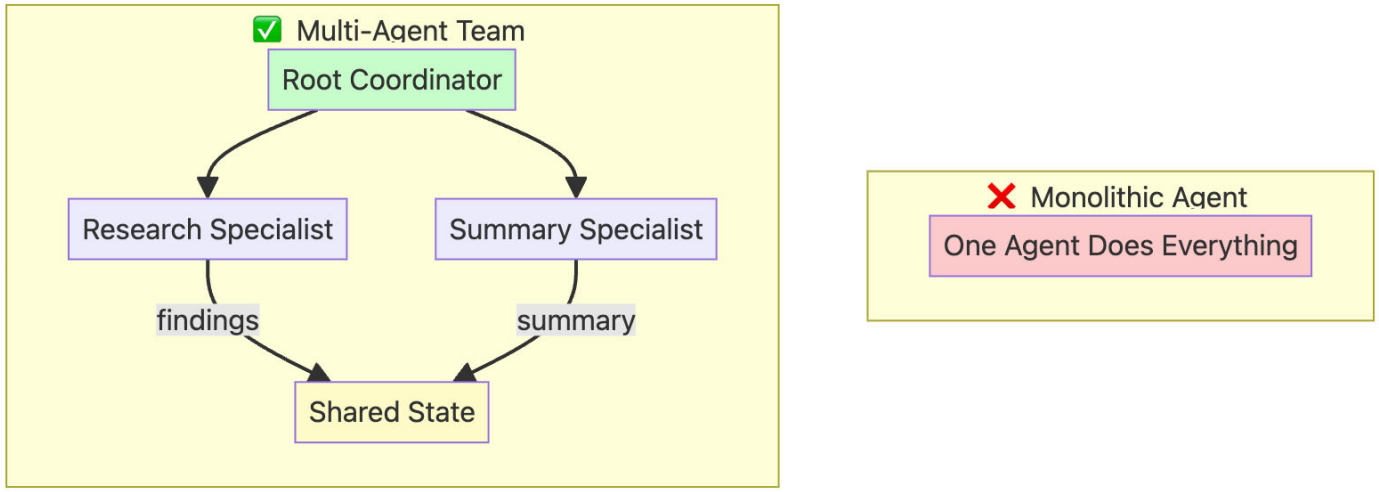
Single agents can do a lot. But what happens when the task gets complex? A single "monolithic" agent that tries to do research, writing, editing, and fact-checking all at once becomes a problem. Its instruction prompt gets long and confusing. It's hard to debug (which part failed?), difficult to maintain, and often produces unreliable results.

The Solution: A Team of Specialists

Instead of one "do-it-all" agent, we can build a **multi-agent system**. This is a team of simple, specialized agents that collaborate, just like a real-world team. Each agent has one clear job (e.g., one agent *only* does research, another *only* writes). This makes them easier to build, easier to test, and much more powerful and reliable when working together.

To learn more, check out the documentation related to LLM agents in ADK.

Architecture: Single Agent vs Multi-Agent Team



2.1 Example: Research & Summarization System

Let's build a system with two specialized agents:

1. **Research Agent** - Searches for information using Google Search
2. **Summarizer Agent** - Creates concise summaries from research findings

```
# Research Agent: Its job is to use the google_search tool and present findings.
research_agent = Agent(
    name="ResearchAgent",
    model=Gemini(
        model="gemini-2.5-flash-lite",
        retry_options=retry_config
    ),
    instruction="""You are a specialized research agent. Your only job is to use the
    google_search tool to find 2-3 pieces of relevant information on the given topic and present the findings with citations.""",
    tools=[google_search],
    output_key="research_findings",  # The result of this agent will be stored in the session state with this key.
)
```

```
print("    research_agent created.")
```

```
# Summarizer Agent: Its job is to summarize the text it receives.
summarizer_agent = Agent(
    name="SummarizerAgent",
    model=Gemini(
        model="gemini-2.5-flash-lite",
        retry_options=retry_config
    ),
    # The instruction is modified to request a bulleted list for a clear output format.
    instruction="""Read the provided research findings: {research_findings}
Create a concise summary as a bulleted list with 3-5 key points.""",
    output_key="final_summary",
)

print("✅ summarizer_agent created.")
```

Refer to the ADK documentation for more information on guiding agents with clear and specific instructions.

Then we bring the agents together under a root agent, or coordinator:

```
# Root Coordinator: Orchestrates the workflow by calling the sub-agents as tools.
root_agent = Agent(
    name="ResearchCoordinator",
    model=Gemini(
        model="gemini-2.5-flash-lite",
        retry_options=retry_config
    ),
    # This instruction tells the root agent HOW to use its tools (which are the other agents).
    instruction="""You are a research coordinator. Your goal is to answer the user's query by orchestrating a workflow.
1. First, you MUST call the `ResearchAgent` tool to find relevant information on the topic provided by the user.
2. Next, after receiving the research findings, you MUST call the `SummarizerAgent` tool to create a concise summary.
3. Finally, present the final summary clearly to the user as your response.""",
    # We wrap the sub-agents in `AgentTool` to make them callable tools for the root agent.
    tools=[AgentTool(research_agent), AgentTool(summarizer_agent)],
)

print("✅ root_agent created.")
```

Here we're using `AgentTool` to wrap the sub-agents to make them callable tools for the root agent. We'll explore `AgentTool` in-detail on Day 2.

Let's run the agent and ask it about a topic:

```
runner = InMemoryRunner(agent=root_agent)
response = await runner.run_debug(
    "What are the latest advancements in quantum computing and what do they mean for AI?"
)
```

You've just built your first multi-agent system! You used a single "coordinator" agent to manage the workflow, which is a powerful and flexible pattern.

‼️ However, **relying on an LLM's instructions to control the order can sometimes be unpredictable.** Next, we'll explore a different pattern that gives you guaranteed, step-by-step execution.

Section 3: Sequential Workflows - The Assembly Line

The Problem: Unpredictable Order

The previous multi-agent system worked, but it relied on a **detailed instruction prompt** to force the LLM to run steps in order. This can be unreliable. A complex LLM might decide to skip a step, run them in the wrong order, or get "stuck," making the process unpredictable.

The Solution: A Fixed Pipeline

When you need tasks to happen in a **guaranteed, specific order**, you can use a `SequentialAgent`. This agent acts like an assembly line, running each sub-agent in the exact order you list them. The output of one agent automatically becomes the input for the next, creating a predictable and reliable workflow.

Use Sequential when: Order matters, you need a linear pipeline, or each step builds on the previous one.

To learn more, check out the documentation related to sequential agents in ADK.

Architecture: Blog Post Creation Pipeline

3.1 Example: Blog Post Creation with Sequential Agents

Let's build a system with three specialized agents:

1. **Outline Agent** - Creates a blog outline for a given topic
2. **Writer Agent** - Writes a blog post

3. **Editor Agent** - Edits a blog post draft for clarity and structure

```
# Outline Agent: Creates the initial blog post outline.
outline_agent = Agent(
    name="OutlineAgent",
    model=Gemini(
        model="gemini-2.5-flash-lite",
        retry_options=retry_config
    ),
    instruction="""Create a blog outline for the given topic with:
    1. A catchy headline
    2. An introduction hook
    3. 3-5 main sections with 2-3 bullet points for each
    4. A concluding thought""",
    output_key="blog_outline",  # The result of this agent will be stored in the session state with this key.
)

print("✅ outline_agent created.")
```

```
# Writer Agent: Writes the full blog post based on the outline from the previous agent.
writer_agent = Agent(
    name="WriterAgent",
    model=Gemini(
        model="gemini-2.5-flash-lite",
        retry_options=retry_config
    ),
    # The `{blog_outline}` placeholder automatically injects the state value from the previous agent's output.
    instruction="""Following this outline strictly: {blog_outline}
    Write a brief, 200 to 300-word blog post with an engaging and informative tone.""",
    output_key="blog_draft",  # The result of this agent will be stored with this key.
)

print("✅ writer_agent created.")
```

```
# Editor Agent: Edits and polishes the draft from the writer agent.
editor_agent = Agent(
    name="EditorAgent",
    model=Gemini(
        model="gemini-2.5-flash-lite",
        retry_options=retry_config
    ),
    # This agent receives the `{blog_draft}` from the writer agent's output.
    instruction="""Edit this draft: {blog_draft}
    Your task is to polish the text by fixing any grammatical errors, improving the flow and sentence structure, and enhancing overall
    output_key="final_blog",  # This is the final output of the entire pipeline.
```

```
)

print("    editor_agent created.")
```

Then we bring the agents together under a sequential agent, which runs the agents in the order that they are listed:

```
root_agent = SequentialAgent(
    name="BlogPipeline",
    sub_agents=[outline_agent, writer_agent, editor_agent],
)

print("✅ Sequential Agent created.")
```

Let's run the agent and give it a topic to write a blog post about:

```
runner = InMemoryRunner(agent=root_agent)
response = await runner.run_debug(
    "Write a blog post about the benefits of multi-agent systems for software developers"
)
```

👏 Great job! You've now created a reliable "assembly line" using a sequential agent, where each step runs in a predictable order.

This is perfect for tasks that build on each other, but it's slow if the tasks are independent. Next, we'll look at how to run multiple agents at the same time to speed up your workflow.

🏞️ Section 4: Parallel Workflows - Independent Researchers

The Problem: The Bottleneck

The previous sequential agent is great, but it's an assembly line. Each step must wait for the previous one to finish. What if you have several tasks that are **not dependent** on each other? For example, researching three *different* topics. Running them in sequence would be slow and inefficient, creating a bottleneck where each task waits unnecessarily.

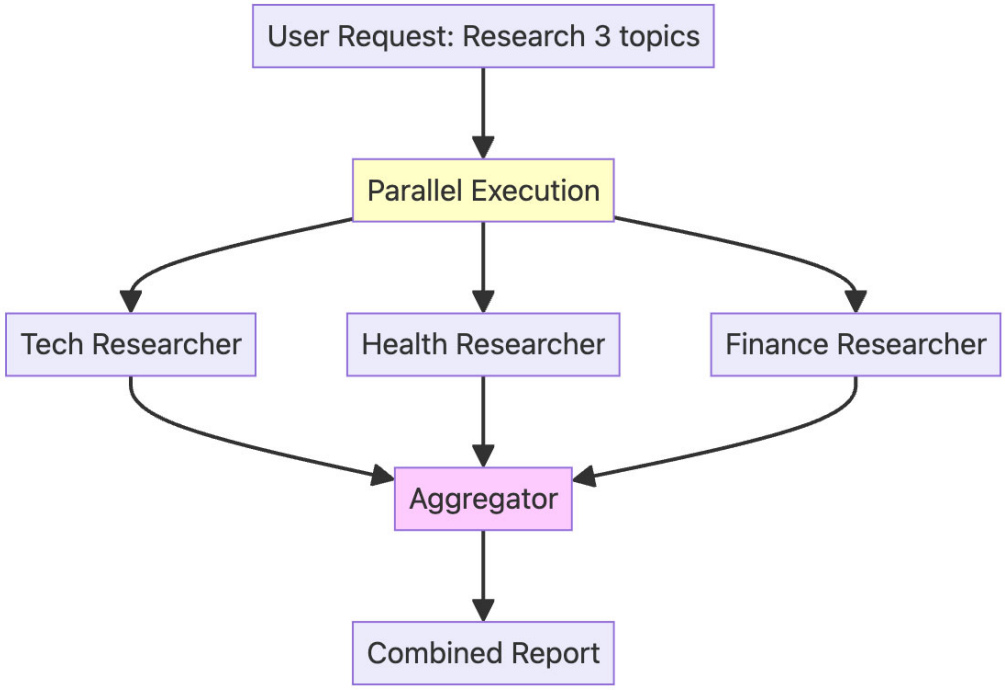
The Solution: Concurrent Execution

When you have independent tasks, you can run them all at the same time using a `ParallelAgent`. This agent executes all of its sub-agents concurrently, dramatically speeding up the workflow. Once all parallel tasks are complete, you can then pass their combined results to a final 'aggregator' step.

Use Parallel when: Tasks are independent, speed matters, and you can execute concurrently.

To learn more, check out the documentation related to parallel agents in ADK.

Architecture: Multi-Topic Research



4.1 Example: Parallel Multi-Topic Research

Let's build a system with four agents:

1. **Tech Researcher** - Researches AI/ML news and trends
2. **Health Researcher** - Researches recent medical news and trends
3. **Finance Researcher** - Researches finance and fintech news and trends
4. **Aggregator Agent** - Combines all research findings into a single summary

```
# Tech Researcher: Focuses on AI and ML trends.
tech_researcher = Agent(
    name="TechResearcher",
    model=Gemini(
        model="gemini-2.5-flash-lite",
        retry_options=retry_config
    ),
    instruction="""Research the latest AI/ML trends. Include 3 key developments,
```

```
the main companies involved, and the potential impact. Keep the report very concise (100 words).""",
    tools=[google_search],
    output_key="tech_research",  # The result of this agent will be stored in the session state with this key.
)

print("   tech_researcher created.")
```

```
# Health Researcher: Focuses on medical breakthroughs.
health_researcher = Agent(
    name="HealthResearcher",
    model=Gemini(
        model="gemini-2.5-flash-lite",
        retry_options=retry_config
    ),
    instruction="""Research recent medical breakthroughs. Include 3 significant advances,
their practical applications, and estimated timelines. Keep the report concise (100 words).""",
    tools=[google_search],
    output_key="health_research",  # The result will be stored with this key.
)

print("✅ health_researcher created.")
```

```
# Finance Researcher: Focuses on fintech trends.
finance_researcher = Agent(
    name="FinanceResearcher",
    model=Gemini(
        model="gemini-2.5-flash-lite",
        retry_options=retry_config
    ),
    instruction="""Research current fintech trends. Include 3 key trends,
their market implications, and the future outlook. Keep the report concise (100 words).""",
    tools=[google_search],
    output_key="finance_research",  # The result will be stored with this key.
)

print("✅ finance_researcher created.")
```

```
# The AggregatorAgent runs *after* the parallel step to synthesize the results.
aggregator_agent = Agent(
    name="AggregatorAgent",
    model=Gemini(
        model="gemini-2.5-flash-lite",
        retry_options=retry_config
    ),
    # It uses placeholders to inject the outputs from the parallel agents, which are now in the session state.
```

```
    instruction="""Combine these three research findings into a single executive summary:

    **Technology Trends:**
    {tech_research}

    **Health Breakthroughs:**
    {health_research}

    **Finance Innovations:**
    {finance_research}

    Your summary should highlight common themes, surprising connections, and the most important key takeaways from all three reports.
    output_key="executive_summary",  # This will be the final output of the entire system.
)

print("    aggregator_agent created.")
```

Then we bring the agents together under a parallel agent, which is itself nested inside of a sequential agent.

This design ensures that the research agents run first in parallel, then once all of their research is complete, the aggregator agent brings together all of the research findings into a single report:

```
# The ParallelAgent runs all its sub-agents simultaneously.
parallel_research_team = ParallelAgent(
    name="ParallelResearchTeam",
    sub_agents=[tech_researcher, health_researcher, finance_researcher],
)

# This SequentialAgent defines the high-level workflow: run the parallel team first, then run the aggregator.
root_agent = SequentialAgent(
    name="ResearchSystem",
    sub_agents=[parallel_research_team, aggregator_agent],
)

print("✅ Parallel and Sequential Agents created.")
```

Let's run the agent and give it a prompt to research the given topics:

```
runner = InMemoryRunner(agent=root_agent)
response = await runner.run_debug(
    "Run the daily executive briefing on Tech, Health, and Finance"
)
```

🎉 Great! You've seen how parallel agents can dramatically speed up workflows by running independent tasks concurrently.

So far, all our workflows run from start to finish and then stop. **But what if you need to review and improve an output multiple times?** Next, we'll build a workflow that can loop and refine its own work.

➰ Section 5: Loop Workflows - The Refinement Cycle

The Problem: One-Shot Quality

All the workflows we've seen so far run from start to finish. The `SequentialAgent` and `ParallelAgent` produce their final output and then stop. This 'one-shot' approach isn't good for tasks that require refinement and quality control. What if the first draft of our story is bad? We have no way to review it and ask for a rewrite.

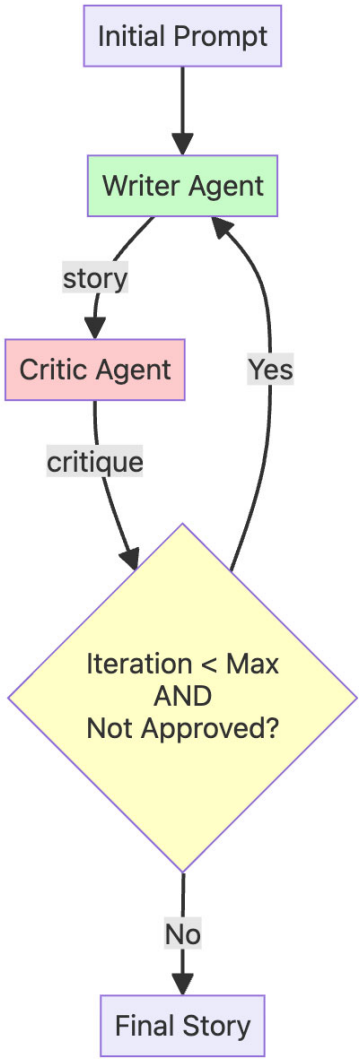
The Solution: Iterative Refinement

When a task needs to be improved through cycles of feedback and revision, you can use a `LoopAgent`. A `LoopAgent` runs a set of sub-agents repeatedly *until a specific condition is met or a maximum number of iterations is reached.* This creates a refinement cycle, allowing the agent system to improve its own work over and over.

Use Loop when: Iterative improvement is needed, quality refinement matters, or you need repeated cycles.

To learn more, check out the documentation related to loop agents in ADK.

Architecture: Story Writing & Critique Loop



5.1 Example: Iterative Story Refinement

Let's build a system with two agents:

1. **Writer Agent** - Writes a draft of a short story
2. **Critic Agent** - Reviews and critiques the short story to suggest improvements

```
# This agent runs ONCE at the beginning to create the first draft.
initial_writer_agent = Agent(
    name="InitialWriterAgent",
    model=Gemini(
```

```
        model="gemini-2.5-flash-lite",
        retry_options=retry_config
    ),
    instruction="""Based on the user's prompt, write the first draft of a short story (around 100-150 words).
    Output only the story text, with no introduction or explanation.""",
    output_key="current_story",  # Stores the first draft in the state.
)

print("    initial_writer_agent created.")
```

```
# This agent's only job is to provide feedback or the approval signal. It has no tools.
critic_agent = Agent(
    name="CriticAgent",
    model=Gemini(
        model="gemini-2.5-flash-lite",
        retry_options=retry_config
    ),
    instruction="""You are a constructive story critic. Review the story provided below.
    Story: {current_story}

    Evaluate the story's plot, characters, and pacing.
    - If the story is well-written and complete, you MUST respond with the exact phrase: "APPROVED"
    - Otherwise, provide 2-3 specific, actionable suggestions for improvement.""",
    output_key="critique",  # Stores the feedback in the state.
)

print("✅ critic_agent created.")
```

Now, we need a way for the loop to actually stop based on the critic's feedback. The `LoopAgent` itself doesn't automatically know that "APPROVED" means "stop."

We need an agent to give it an explicit signal to terminate the loop.

We do this in two parts:

1. A simple Python function that the `LoopAgent` understands as an "exit" signal.
2. An agent that can call that function when the right condition is met.

First, you'll define the `exit_loop` function:

```
# This is the function that the RefinerAgent will call to exit the loop.
def exit_loop():
    """Call this function ONLY when the critique is 'APPROVED', indicating the story is finished and no more changes are needed."""
    return {"status": "approved", "message": "Story approved. Exiting refinement loop."}
```

```
print("    exit_loop function created.")
```

To let an agent call this Python function, we wrap it in a `FunctionTool`. Then, we create a `RefinerAgent` that has this tool.

👉 **Notice its instructions:** this agent is the "brain" of the loop. It reads the `{critique}` from the `CriticAgent` and decides whether to (1) call the `exit_loop` tool or (2) rewrite the story.

```
# This agent refines the story based on critique OR calls the exit_loop function.
refiner_agent = Agent(
    name="RefinerAgent",
    model=Gemini(
        model="gemini-2.5-flash-lite",
        retry_options=retry_config
    ),
    instruction="""You are a story refiner. You have a story draft and critique.

    Story Draft: {current_story}
    Critique: {critique}

    Your task is to analyze the critique.
    - IF the critique is EXACTLY "APPROVED", you MUST call the `exit_loop` function and nothing else.
    - OTHERWISE, rewrite the story draft to fully incorporate the feedback from the critique.""",
    output_key="current_story",  # It overwrites the story with the new, refined version.
    tools=[
        FunctionTool(exit_loop)
    ],  # The tool is now correctly initialized with the function reference.
)

print("✅ refiner_agent created.")
```

Then we bring the agents together under a loop agent, which is itself nested inside of a sequential agent.

This design ensures that the system first produces an initial story draft, then the refinement loop runs up to the specified number of `max_iterations`:

```
# The LoopAgent contains the agents that will run repeatedly: Critic -> Refiner.
story_refinement_loop = LoopAgent(
    name="StoryRefinementLoop",
    sub_agents=[critic_agent, refiner_agent],
    max_iterations=2,  # Prevents infinite loops
)
```

```
# The root agent is a SequentialAgent that defines the overall workflow: Initial Write -> Refinement Loop.
root_agent = SequentialAgent(
    name="StoryPipeline",
    sub_agents=[initial_writer_agent, story_refinement_loop],
)

print("    Loop and Sequential Agents created.")
```

Let's run the agent and give it a topic to write a short story about:

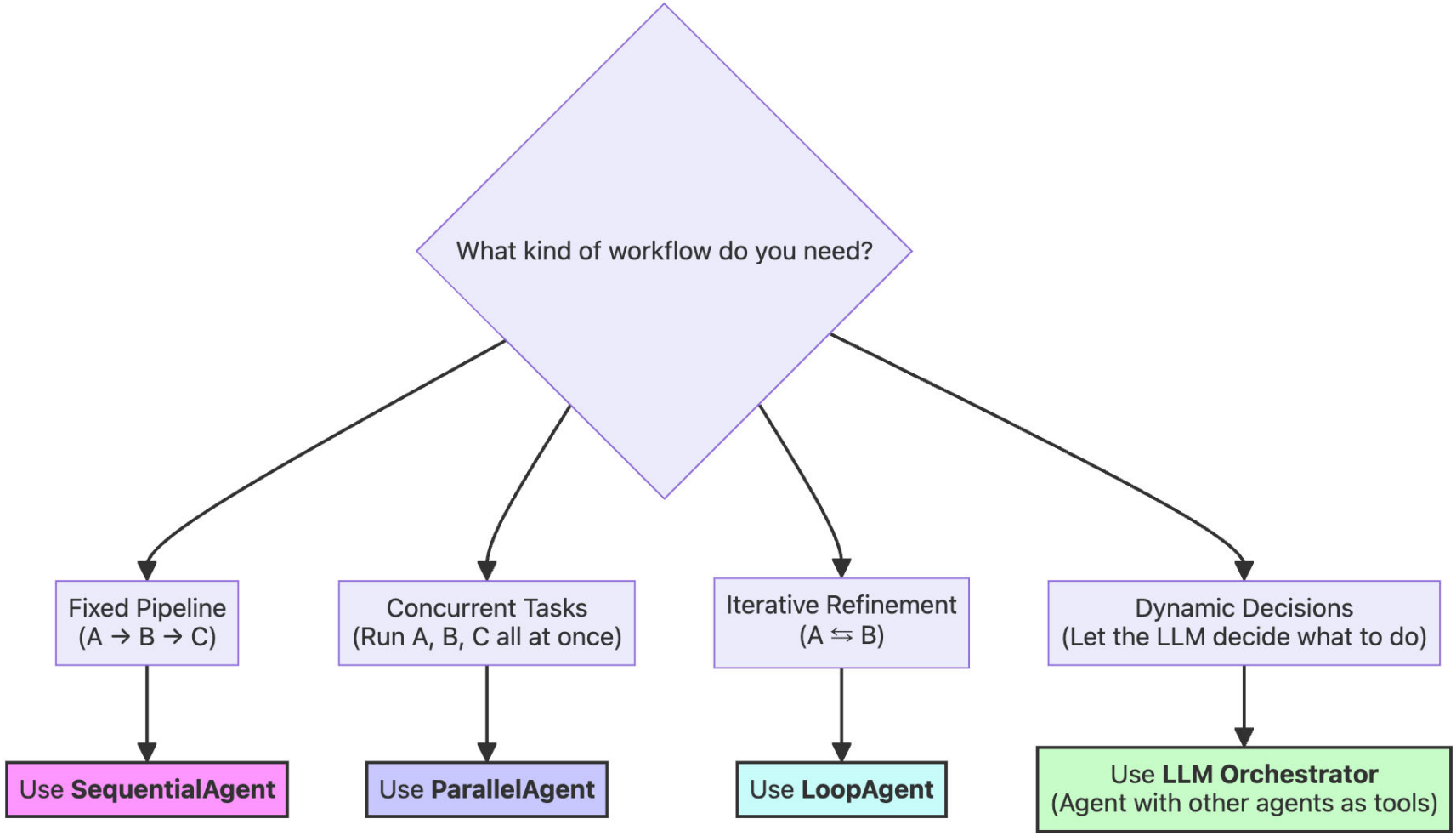
```
runner = InMemoryRunner(agent=root_agent)
response = await runner.run_debug(
    "Write a short story about a lighthouse keeper who discovers a mysterious, glowing map"
)
```

You've now implemented a loop agent, creating a sophisticated system that can iteratively review and improve its own output. This is a key pattern for ensuring high-quality results.

You now have a complete toolkit of workflow patterns. Let's put it all together and review how to choose the right one for your use case.

Section 6: Summary - Choosing the Right Pattern

Decision Tree: Which Workflow Pattern?



Quick Reference Table

Pattern	When to Use	Example	Key Feature
LLM-based (sub_agents)	Dynamic orchestration needed	Research + Summarize	LLM decides what to call
Sequential	Order matters, linear pipeline	Outline ⟶ Write ⟶ Edit	Deterministic order
Parallel	Independent tasks, speed matters	Multi-topic research	Concurrent execution
Loop	Iterative improvement needed	Writer + Critic refinement	Repeated cycles

---

✅ Congratulations! You're Now an Agent Orchestrator

In this notebook, you made the leap from a single agent to a **multi-agent system**.

You saw **why** a team of specialists is easier to build and debug than one "do-it-all" agent. Most importantly, you learned how to be the **director** of that team.

You used `SequentialAgent`, `ParallelAgent`, and `LoopAgent` to create deterministic workflows, and you even used an LLM as a 'manager' to make dynamic decisions. You also mastered the "plumbing" by using `output_key` to pass state between agents and make them collaborative.

ℹ️ **Note: No submission required!**

This notebook is for your hands-on practice and learning only. You **do not** need to submit it anywhere to complete the course.

Learn More