[ ]:

```
# @title Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
# https://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under 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)

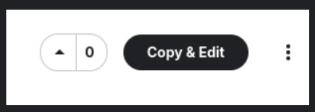
To use the Kaggle Notebooks in this course, you'll need to verify your account with a phone number.

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.

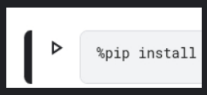


This creates a private copy of the notebook just for you.

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.

[1]:
```
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}"
    )
```

```
✅ Gemini API key setup complete.
```

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.

[2]:
```
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.")
```

```
✅ 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.

[5]:
```
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

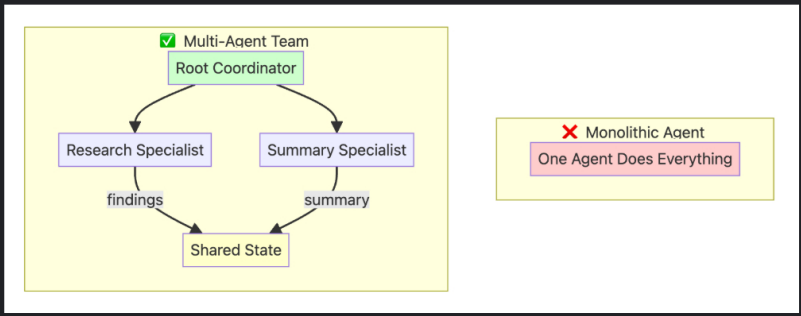
Single agents can do a lot. But what happens when the task gets complex? A single "monolithic" agent that tries to do research, writing, editing, and fact-checking all at once becomes a problem. Its instruction prompt gets long and confusing. It's hard to debug (which part failed?), difficult to maintain,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

[6]:
```
# 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.")
```

```
✅ research_agent created.
```

[7]:
```
# 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.")
```

```
✅ 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:

[8]:
```
# 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.")
```

```
✅ 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:

[10]:
```
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?"
)
```

```
### Created new session: debug_session_id

User > What are the latest advancements in quantum computing and what do they mean for AI?
WARNING:google_genai.types:Warning: there are non-text parts in the response: ['function_call'], returning concatenated text result from text parts. Check the full candidates.content.parts accessor to get the full model response.
WARNING:google_genai.types:Warning: there are non-text parts in the response: ['function_call'], returning concatenated text result from text parts. Check the full candidates.content.parts accessor to get the full model response.
ResearchCoordinator > Recent advancements in quantum computing are poised to revolutionize AI by overcoming classical computing limitations and enabling \"Quantum AI.\" This integration promises significantly faster AI training and performance, and the capacity to solve complex, previously intractable problems in fields like drug discovery, logistics, and materials science. Quantum AI also offers more efficient data processing and can lead to more accurate AI models, improving areas such as natural language processing, cybersecurity, and autonomous systems. The integration is expected to drive major innovations across diverse sectors including healthcare, mobility, finance, and energy. While widespread adoption is still some years off, progress is rapid, with increasing accessibility through cloud services, positioning quantum computing as a strategic accelerator 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

[11]:
```
# 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.")
```

```
✅ outline_agent created.
```

[12]:
```
# 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.")
```

✅ writer_agent created.

[13]:

```
# 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 clarity.""",
    output_key="final_blog",  # This is the final output of the entire pipeline.
)

print("✅ editor_agent created.")
```

✅ editor_agent created.

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

[14]:

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

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

✅ 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"
)
```

```
 ### Created new session: debug_session_id

User > Write a blog post about the benefits of multi-agent systems for software developers
OutlineAgent > ## Blog Outline:

**Headline:** Unleash Your Development Superpowers: How Multi-Agent Systems Are Revolutionizing Software

**Introduction Hook:** Tired of monolithic codebases that buckle under complexity? Imagine a software architecture where independent, intelligent agents collaborate to ac
hieve a common goal, making your development process more agile, scalable, and robust. Welcome to the world of Multi-Agent Systems (MAS), a paradigm shift poised to trans
form how we build software.

---

### Main Sections:

**1. Enhanced Modularity and Maintainability:**

*   **Breaking Down the Monolith:** MAS allows developers to decompose complex problems into smaller, more manageable units (agents). This promotes clear separation of co
ncerns, making individual components easier to understand, debug, and modify.
*   **Independent Evolution:** Each agent can be developed, tested, and updated independently. This drastically reduces the risk of introducing unintended side effects an
d accelerates the development lifecycle.
*   **Simplified Troubleshooting:** When issues arise, developers can pinpoint problems to specific agents, rather than sifting through vast amounts of interconnected cod
e.

**2. Improved Scalability and Resilience:**

*   **Dynamic Scaling:** MAS naturally supports scalability. As demand increases, new agents can be instantiated and deployed to handle the load without disrupting existi
ng functionality.
*   **Fault Tolerance:** If one agent fails, others can often compensate or take over its responsibilities, leading to more resilient systems that can withstand partial f
ailures.
*   **Distributed Processing:** Agents can operate on different machines or even different geographical locations, enabling true distributed computing and leveraging avai
lable resources more effectively.

**3. Fostering Collaboration and Intelligence:**

*   **Complex Problem Solving:** MAS enables the tackling of highly complex problems that might be intractable for a single, monolithic system. Agents can share informati
on, negotiate, and coordinate their actions to find optimal solutions.
*   **Emergent Behavior:** The interaction of simple agents can lead to sophisticated, emergent behaviors that were not explicitly programmed, unlocking new levels of int
elligence and adaptability in your software.
*   **Adaptability to Changing Environments:** Agents can be designed to sense their environment and adapt their behavior accordingly, making MAS ideal for dynamic and un
predictable situations.

---

**Concluding Thought:** Embracing Multi-Agent Systems isn't just about adopting a new architectural pattern; it's about unlocking a more powerful, flexible, and intellige
nt way to develop software. For developers, it means wielding tools that amplify their ability to build the complex, scalable, and resilient applications of the future. A
re you ready to join the agent revolution?
```

```
---------------------------------------------------------------------------
ClientError                               Traceback (most recent call last)
/tmp/ipykernel_48/1463792725.py in <cell line: 1>()
      1 runner = InMemoryRunner(agent=root_agent)
----> 2 response = await runner.run_debug(
      3     "Write a blog post about the benefits of multi-agent systems for software developers"
      4 )

/usr/local/lib/python3.11/dist-packages/google/adk/runners.py in run_debug(self, user_messages, user_id, session_id, run_config, quiet, verbose)
   1021         print(f'\nUser > {message}')
   1022
-> 1023       async for event in self.run_async(
   1024           user_id=user_id,
   1025           session_id=session.id,

/usr/local/lib/python3.11/dist-packages/google/adk/runners.py in run_async(self, user_id, session_id, invocation_id, new_message, state_delta, run_config)
    441
    442     async with Aclosing(_run_with_trace(new_message, invocation_id)) as agen:
--> 443       async for event in agen:
    444         yield event
    445

/usr/local/lib/python3.11/dist-packages/google/adk/runners.py in _run_with_trace(new_message, invocation_id)
    425             )
    426         ) as agen:
--> 427           async for event in agen:
    428             yield event
    429         # Run compaction after all events are yielded from the agent.

/usr/local/lib/python3.11/dist-packages/google/adk/runners.py in _exec_with_plugin(self, invocation_context, session, execute_fn, is_live_call)
    651       # Step 2: Otherwise continue with normal execution
    652       async with Aclosing(execute_fn(invocation_context)) as agen:
--> 653         async for event in agen:
    654           if not event.partial:
    655             if self._should_append_event(event, is_live_call):

/usr/local/lib/python3.11/dist-packages/google/adk/runners.py in execute(ctx)
    414         async def execute(ctx: InvocationContext) -> AsyncGenerator[Event]:
    415           async with Aclosing(ctx.agent.run_async(ctx)) as agen:
--> 416             async for event in agen:
    417               yield event
    418

/usr/local/lib/python3.11/dist-packages/google/adk/agents/base_agent.py in run_async(self, parent_context)
    292
    293       async with Aclosing(self._run_async_impl(ctx)) as agen:
--> 294         async for event in agen:
    295           yield event
    296

/usr/local/lib/python3.11/dist-packages/google/adk/agents/sequential_agent.py in _run_async_impl(self, ctx)
     75
     76       async with Aclosing(sub_agent.run_async(ctx)) as agen:
---> 77         async for event in agen:
     78           yield event
     79           if ctx.should_pause_invocation(event):

/usr/local/lib/python3.11/dist-packages/google/adk/agents/base_agent.py in run_async(self, parent_context)
    292
    293       async with Aclosing(self._run_async_impl(ctx)) as agen:
--> 294         async for event in agen:
    295           yield event
    296

/usr/local/lib/python3.11/dist-packages/google/adk/agents/llm_agent.py in _run_async_impl(self, ctx)
    433     should_pause = False
    434     async with Aclosing(self._llm_flow.run_async(ctx)) as agen:
--> 435       async for event in agen:
    436         self.__maybe_save_output_to_state(event)
    437         yield event

/usr/local/lib/python3.11/dist-packages/google/adk/flows/llm_flows/base_llm_flow.py in run_async(self, invocation_context)
    354       last_event = None
    355       async with Aclosing(self._run_one_step_async(invocation_context)) as agen:
--> 356         async for event in agen:
    357           last_event = event
    358           yield event

/usr/local/lib/python3.11/dist-packages/google/adk/flows/llm_flows/base_llm_flow.py in _run_one_step_async(self, invocation_context)
    431         )
    432     ) as agen:
--> 433       async for llm_response in agen:
    434         # Postprocess after calling the LLM.
    435         async with Aclosing(

/usr/local/lib/python3.11/dist-packages/google/adk/flows/llm_flows/base_llm_flow.py in _call_llm_async(self, invocation_context, llm_request, model_response_event)
    802
    803     async with Aclosing(_call_llm_with_tracing()) as agen:
--> 804       async for event in agen:
    805         yield event
    806

/usr/local/lib/python3.11/dist-packages/google/adk/flows/llm_flows/base_llm_flow.py in _call_llm_with_tracing()
    786                 )
    787           ) as agen:
--> 788             async for llm_response in agen:
    789               trace_call_llm(
    790                   invocation_context,

/usr/local/lib/python3.11/dist-packages/google/adk/flows/llm_flows/base_llm_flow.py in _run_and_handle_error(self, response_generator, invocation_context, llm_request, mo
del_response_event)
    996         yield error_response
    997       else:
--> 998         raise model_error
    999
   1000   def __get_llm(self, invocation_context: InvocationContext) -> BaseLlm:

/usr/local/lib/python3.11/dist-packages/google/adk/flows/llm_flows/base_llm_flow.py in _run_and_handle_error(self, response_generator, invocation_context, llm_request, mo
del_response_event)
    980     try:
    981       async with Aclosing(response_generator) as agen:
```

```
--> 982         async for response in agen:
    983           yield response
    984     except Exception as model_error:

/usr/local/lib/python3.11/dist-packages/google/adk/models/google_llm.py in generate_content_async(self, llm_request, stream)
    179
    180     else:
--> 181       response = await self.api_client.aio.models.generate_content(
    182           model=llm_request.model,
    183           contents=llm_request.contents,

/usr/local/lib/python3.11/dist-packages/google/genai/models.py in generate_content(self, model, contents, config)
   6854     response = types.GenerateContentResponse()
   6855     while remaining_remote_calls_afc > 0:
-> 6856       response = await self._generate_content(
   6857           model=model, contents=contents, config=parsed_config
   6858       )

/usr/local/lib/python3.11/dist-packages/google/genai/models.py in _generate_content(self, model, contents, config)
   5665     request_dict = _common.encode_unserializable_types(request_dict)
   5666
-> 5667     response = await self._api_client.async_request(
   5668         'post', path, request_dict, http_options
   5669     )

/usr/local/lib/python3.11/dist-packages/google/genai/_api_client.py in async_request(self, http_method, path, request_dict, http_options)
   1363     )
   1364
-> 1365     result = await self._async_request(
   1366         http_request=http_request, http_options=http_options, stream=False
   1367     )

/usr/local/lib/python3.11/dist-packages/google/genai/_api_client.py in _async_request(self, http_request, http_options, stream)
   1308         retry = tenacity.AsyncRetrying(**retry_kwargs)
   1309         return await retry(self._async_request_once, http_request, stream)  # type: ignore[no-any-return]
-> 1310     return await self._async_retry(  # type: ignore[no-any-return]
   1311         self._async_request_once, http_request, stream
   1312     )

/usr/local/lib/python3.11/dist-packages/tenacity/asyncio/__init__.py in __call__(self, fn, *args, **kwargs)
    109         retry_state = RetryCallState(retry_object=self, fn=fn, args=args, kwargs=kwargs)
    110         while True:
--> 111             do = await self.iter(retry_state=retry_state)
    112             if isinstance(do, DoAttempt):
    113                 try:

/usr/local/lib/python3.11/dist-packages/tenacity/asyncio/__init__.py in iter(self, retry_state)
    151         result = None
    152         for action in self.iter_state.actions:
--> 153             result = await action(retry_state)
    154         return result
    155

/usr/local/lib/python3.11/dist-packages/tenacity/_utils.py in inner(*args, **kwargs)
     97
     98     async def inner(*args: typing.Any, **kwargs: typing.Any) -> typing.Any:
---> 99         return call(*args, **kwargs)
    100
    101     return inner

/usr/local/lib/python3.11/dist-packages/tenacity/__init__.py in <lambda>(rs)
    398     def _post_retry_check_actions(self, retry_state: "RetryCallState") -> None:
    399         if not (self.iter_state.is_explicit_retry or self.iter_state.retry_run_result):
--> 400             self._add_action_func(lambda rs: rs.outcome.result())
    401             return
    402

/usr/lib/python3.11/concurrent/futures/_base.py in result(self, timeout)
    447                     raise CancelledError()
    448                 elif self._state == FINISHED:
--> 449                     return self.__get_result()
    450
    451                 self._condition.wait(timeout)

/usr/lib/python3.11/concurrent/futures/_base.py in __get_result(self)
    399         if self._exception:
    400             try:
--> 401                 raise self._exception
    402             finally:
    403                 # Break a reference cycle with the exception in self._exception

/usr/local/lib/python3.11/dist-packages/tenacity/asyncio/__init__.py in __call__(self, fn, *args, **kwargs)
    112             if isinstance(do, DoAttempt):
    113                 try:
--> 114                     result = await fn(*args, **kwargs)
    115                 except BaseException:  # noqa: B902
    116                     retry_state.set_exception(sys.exc_info())  # type: ignore[arg-type]

/usr/local/lib/python3.11/dist-packages/google/genai/_api_client.py in _async_request_once(self, http_request, stream)
   1253             **self._async_client_session_request_args,
   1254         )
-> 1255         await errors.APIError.raise_for_async_response(response)
   1256         return HttpResponse(response.headers, [await response.text()])
   1257       except (

/usr/local/lib/python3.11/dist-packages/google/genai/errors.py in raise_for_async_response(cls, response)
    164
    165     if 400 <= status_code < 500:
--> 166       raise ClientError(status_code, response_json, response)
    167     elif 500 <= status_code < 600:
    168       raise ServerError(status_code, response_json, response)

ClientError: 400 INVALID_ARGUMENT. {'error': {'code': 400, 'message': 'API key expired. Please renew the API key.', 'status': 'INVALID_ARGUMENT', 'details': [{'@type': 't
ype.googleapis.com/google.rpc.ErrorInfo', 'reason': 'API_KEY_INVALID', 'domain': 'googleapis.com', 'metadata': {'service': 'generativelanguage.googleapis.com'}}, {'@typ
e': 'type.googleapis.com/google.rpc.LocalizedMessage', 'locale': 'en-US', 'message': 'API key expired. Please renew the API key.'}]}}
```

+ Code + Markdown

👏 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.

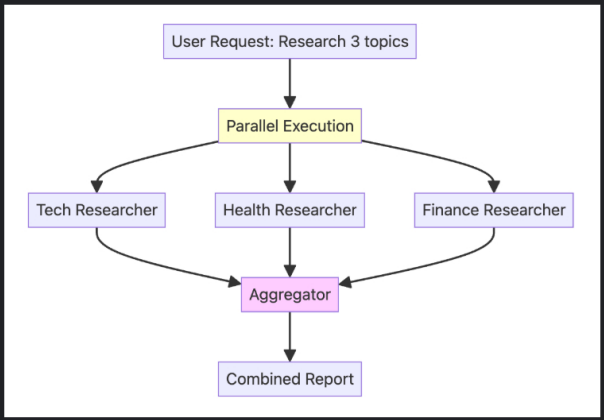
The Solution: Concurrent Execution

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

[16]:
```
# 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.")
```

```
✅ tech_researcher created.
```

[17]:
```
# 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.")
```

```
✅ health_researcher created.
```

[18]:
```
# 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.")
```

```
✅ finance_researcher created.
```

[19]:

```
# 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. The final summary sho
    output_key="executive_summary",  # This will be the final output of the entire system.
)

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

```
✅ 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:

[20]:

```
# 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.")
```

```
✅ Parallel and Sequential Agents created.
```

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

[21]:

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

```
 ### Created new session: debug_session_id

User > Run the daily executive briefing on Tech, Health, and Finance
TechResearcher > **Key AI/ML Trends and Developments:**

1.  **Generative AI for Complex Content Creation:** AI models are increasingly capable of generating diverse content, including graphics, video, and music, beyond just te
xt. Companies like Google (Imagen, Muse) and OpenAI (GPT models) are at the forefront. This trend enhances creativity and practical applications across industries.

2.  **Shift from LLMs to SLMs and Specialized Models:** While Large Language Models (LLMs) have dominated, there's a growing interest in Smaller Language Models (SLMs) an
d highly domain-specific models. This offers efficiency and tailored capabilities.

3.  **AI Agents and Autonomous Systems:** AI agents capable of independent action are emerging for tasks like workflow management and data analysis. Companies like Salesf
orce are developing such tools. This trend promises increased efficiency and automation.

**Main Companies Involved:**

*   **Tech Giants:** Google (Alphabet), OpenAI, Microsoft, Meta, Amazon, NVIDIA.
*   **Specialized AI Companies:** Anthropic, Synthesia, Runway ML, Dataiku, Mistral AI.

**Potential Impact:**

These AI/ML trends are poised to revolutionize industries by increasing productivity, enhancing decision-making, and fostering innovation. They enable personalized custom
```

These AI/ML trends are poised to revolutionize industries by increasing productivity, enhancing decision-making, and fostering innovation. They enable personalized customer experiences, automate complex tasks, and drive new avenues for content creation. However, they also bring challenges related to ethical considerations, data privacy, and workforce adaptation, necessitating a balance between technological advancement and responsible implementation.
HealthResearcher > Here's a brief executive briefing on recent breakthroughs in Technology, Health, and Finance:

Technology: AI and machine learning continue to advance rapidly, with applications in automation, natural language processing, and personalized experiences. Extended Reality (XR), encompassing VR and AR, is transforming gaming, education, and remote collaboration. Quantum computing is showing significant progress, with potential applications in drug discovery, materials science, and cybersecurity, though widespread practical use is still debated.

Health: Next-generation mRNA vaccines offer rapid development and broad applications beyond infectious diseases. Innovative surgical techniques, like the first living mitral valve replacement and total eye/face transplants, are improving patient outcomes. Advancements in AI are also enhancing medical diagnostics and drug discovery.

Finance: Embedded finance is integrating financial services into non-financial platforms, projected for substantial growth. Decentralized finance (DeFi) is maturing with increased security and scalability. AI-powered personalization is becoming central, with AI in financial services market rapidly expanding.
FinanceResearcher > Here's a daily executive briefing for November 11, 2025:

Technology:

*   **AI Integration in Enterprises:** Companies like Spark are focusing on making AI accessible to all employees, emphasizing organizational change management as a key challenge. The trend involves balancing short-term gains with long-term AI investments and adapting to evolving AI models.
*   **Digital Transformation & Data Sovereignty:** Tech conferences are highlighting cloud technologies that respect data sovereignty, particularly in sectors like healthcare, promoting hybrid-cloud models for secure data management.
*   **Cybersecurity & Spyware Concerns:** Tech giants are stepping up defenses against spyware, with ongoing litigation and scrutiny of companies involved in surveillance technology.

Health:

*   **Global Health Financing Emergency:** A significant drop in external health aid threatens health systems, especially in low-income countries. There's a push for increased domestic investment and self-reliance in health financing.
*   **Measles Transmission Resurfaces:** The Americas region has lost its measles-free verification due to reestablished endemic transmission in Canada, highlighting the critical need for high vaccination coverage.
*   **AI in Digital Mental Health:** The use of AI in digital mental health devices presents both opportunities for improved patient care and risks concerning effectiveness and accessibility, prompting careful regulatory consideration.

Finance:

*   **Climate Finance Mobilization:** Efforts are underway to mobilize private climate finance for adaptation, with a focus on developing new financial models and "resilience asset classes" to bridge the gap between philanthropy and markets.
*   **Financial Regulation & Open Banking:** Regulatory bodies are actively involved in rulemaking for open banking and stablecoin frameworks, signaling a dynamic regulatory landscape.
*   **Corporate Earnings & Market Trends:** Companies are reporting on financial performance, with trends including revenue growth from acquisitions, focus on cost savings, and strategic capital management.
AggregatorAgent > ## Executive Summary: Tech, Health, and Finance Briefing

Key Takeaway: Artificial Intelligence is the dominant transformative force across technology, health, and finance, driving innovation while presenting significant ethical and logistical challenges.

Technology is rapidly advancing with sophisticated AI, from generative content creation (graphics, video, music) to emerging AI agents for autonomous tasks. The field is shifting towards more efficient Smaller Language Models (SLMs) and specialized AI, alongside continued progress in Extended Reality (XR) and the potential of quantum computing. However, cybersecurity concerns, particularly around spyware, and the need for organizational change management to integrate AI effectively are critical considerations.

In **Health**, AI is revolutionizing diagnostics and drug discovery, complementing breakthroughs in next-generation mRNA vaccines and innovative surgical techniques. Simultaneously, a global health financing emergency, particularly impacting low-income countries, and resurgent infectious diseases like measles underscore the need for robust public health infrastructure and increased domestic investment. The integration of AI in digital mental health also presents a duality of opportunity and risk.

Finance is being reshaped by embedded finance, maturing DeFi, and pervasive AI-powered personalization. A significant focus is on mobilizing climate finance through new models and addressing regulatory landscapes for open banking and stablecoins. Corporate earnings reflect strategic capital management and cost-saving initiatives.

Connecting Themes: The consistent thread is the pervasive influence of AI, enhancing capabilities across all sectors while necessitating careful consideration of ethical implications, data sovereignty, accessibility, and the balance between innovation and responsible implementation.

+ Code + Markdown

🎉 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.

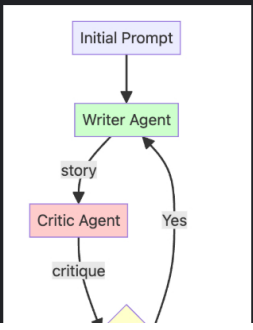
The Solution: Iterative Refinement

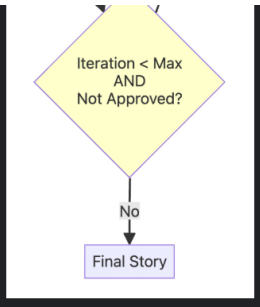
When a task needs to be improved through cycles of feedback and revision, you can use a `LoopAgent`. A `LoopAgent` runs a set of sub-agents repeatedly *until a specific condition is met or a maximum number of iterations is reached.* This creates a refinement cycle, allowing the agent system to improve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

[22]:
```
# 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.")
```

```
✅ initial_writer_agent created.
```

[23]:
```
# 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.")
```

```
✅ 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:

[24]:
```
# 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.")
```

```
✅ 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.

[25]:
```
# 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.")
```

```
✅ 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`:

[26]:
```
# 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.")
```

```
✅ Loop and Sequential Agents created.
```

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

[28]:
```
runner = InMemoryRunner(agent=root_agent)
response = await runner.run_debug(
    "Write a short story about a lighthouse keeper who discovers a mysterious, glowing map"
)
```

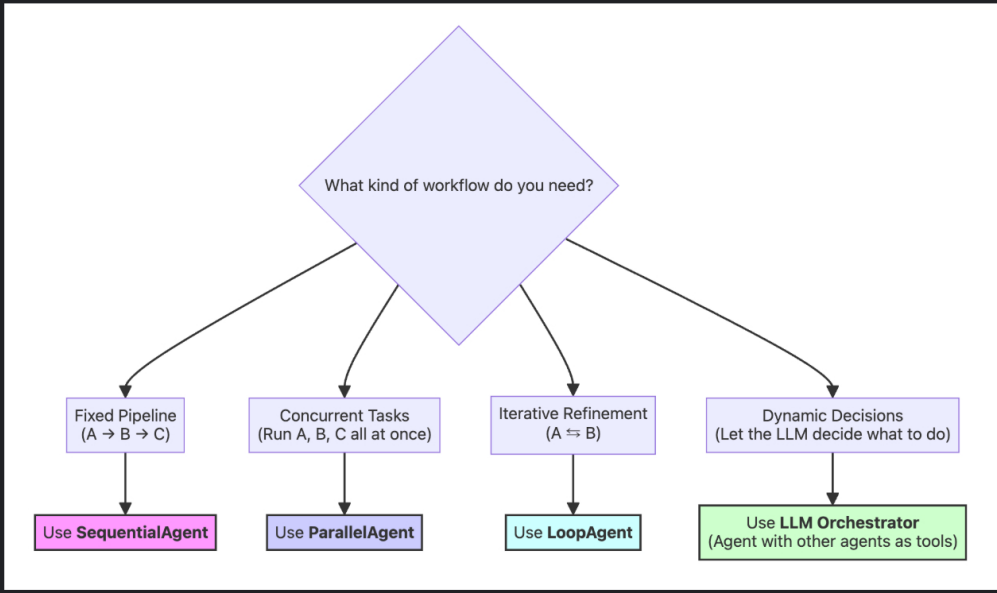
```
 ### Created new session: debug_session_id

User > Write a short story about a lighthouse keeper who discovers a mysterious, glowing map
InitialWriterAgent > Elias traced the salt-worn lines of his map, a familiar comfort against the roaring storm outside. But tonight, something was different. A faint lumi
nescence pulsed from beneath the ancient vellum. He held it closer. The ink, usually faded to near invisibility, now glowed with an ethereal, sea-green light, tracing a p
ath he'd never seen before. It led not to the mainland, but out, into the inky blackness of the churning ocean. A single, glittering X marked a spot far beyond the known
 shipping lanes. Elias's heart hammered. He'd tended this light for thirty years, seen all manner of things the sea could conjure, but this… this was a secret the ocean ha
d kept for itself, and it was now offering it to him.
CriticAgent > The plot has a strong hook and a clear inciting incident: the discovery of the glowing map. The pacing is effective in building immediate intrigue and myste
ry.

Here are a few suggestions for improvement:

1.  **Develop Elias's character beyond his profession:** While his thirty years as a lighthouse keeper provide context, giving Elias a brief internal thought or a small a
ction that reveals a personal desire or a past longing would make his reaction to this extraordinary discovery more impactful. What has he always *hoped* for, or what has
 he *lost* that this map might represent?
2.  **Hint at the nature of the secret:** The phrase "a secret the ocean had kept for itself" is intriguing, but a slightly more concrete, even if still vague, suggestion
could deepen the mystery. Is it treasure? A lost civilization? A mythical creature's lair? A hint, however subtle, would give the reader something more to latch onto.
3.  **Show, don't just tell, Elias's emotional response:** You mention his "heart hammered." Consider adding a physical reaction or a sensory detail to further illustrate
his astonishment and apprehension. Does his breath catch? Does he drop the map momentarily? Does the light reflect strangely in his wide eyes?
RefinerAgent > Elias traced the salt-worn lines of his map, a familiar comfort against the roaring storm outside. Thirty years he'd guarded this lonely light, his life a
quiet rhythm of sea and sky. But tonight, something was different. A faint luminescence pulsed from beneath the ancient vellum. He held it closer, his weathered hand trem
bling. The ink, usually faded to near invisibility, now glowed with an ethereal, sea-green light, tracing a path he'd never seen before. It led not to the mainland, but o
ut, into the inky blackness of the churning ocean. A single, glittering X marked a spot far beyond the known shipping lanes. Elias's breath hitched, the map nearly slippi
ng from his grasp as the light reflected in his wide, astonished eyes. He'd always dreamed of more than just the endless horizon, of a purpose beyond the beam he swept ea
ch night. Could this be it? This glowing mark, hinting at forgotten depths, perhaps sunken cities or the fabled kraken's hoard – a secret the ocean had guarded for millen
nia, and was now offering to him. His heart hammered not with fear, but with a wild, unexpected hope.
CriticAgent > APPROVED
WARNING:google_genai.types:Warning: there are non-text parts in the response: ['function_call'], returning concatenated text result from text parts. Check the full candid
ates.content.parts accessor to get the full model response.
```

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

Refer to the following documentation to learn more:

- Agents in ADK
- Sequential Agents in ADK
- Parallel Agents in ADK
- Loop Agents in ADK
- Custom Agents in ADK

🎯 Next Steps

Ready for the next challenge? Stay tuned for Day 2 notebooks where we'll learn how to create **Custom Functions, use MCP Tools** and manage **Long-Running operations!**

Authors

Kristopher Overholt