Multi-Agent Systems with OpenAI Agents SDK: Tutorial

Prerequisites

- **OpenAI API Access:** An OpenAI account with API key (for calling language models and using OpenAI-hosted tools).
- **Python Basics:** Familiarity with Python programming and asynchronous code (the SDK uses async calls).
- **LLM and Function-Calling Knowledge:** Understanding of prompt design and how large language models can call functions/tools.
- **Google Account:** Required to use Google Colab for running the code (no local installation needed).

Google Colab Setup

1. **Open a Colab Notebook:** Go to colab.research.google.com and create a new Python 3 notebook. Ensure GPU/TPU acceleration is **not** needed (CPU is sufficient for API calls).

2. **Install the OpenAI Agents SDK:** In a Colab code cell, install the SDK from PyPI. Use the `pip` command:

```
!pip install openai-agents
```

This will install the Agents SDK and its dependencies.

1. **Set Up OpenAI API Key:** You can store your API key securely in Colab. For example, use `python-dotenv` or `os.environ`:

```
import os
os.environ["OPENAI_API_KEY"] = "sk-..."  # replace with your actual key
```

Alternatively, use Colab's *secrets* feature or input form to avoid hardcoding the key. The OpenAI Agents SDK will pick up the `OPENAI_API_KEY` environment variable for authentication.

1. **Import the SDK Modules:** At minimum, you'll need:

```
from agents import Agent, Runner, ModelSettings, trace, gen_trace_id
from agents import WebSearchTool, CodeInterpreterTool  # (optional tools, as
needed)
```

- `Agent` is the core class to define an agent.

- `Runner` executes agents.
- `ModelSettings` allows configuration like parallelism or reasoning mode.
- `trace` and `gen_trace_id` help capture and group execution traces (discussed later).

- Tool classes like `WebSearchTool` (for web queries) or `CodeInterpreterTool` (to run code) can be imported when you plan to use them.

- **Verify the Installation:** Run a quick test to ensure everything is set. For example:

```
print("OpenAI Agents SDK version:", __import__('agents').__version__)
```

This should output the SDK version, confirming that the library is ready.

Understanding Agent Design Patterns

The OpenAI Agents SDK supports multiple **agent orchestration patterns**. Each pattern defines how tasks are broken down and how multiple agents might interact. Here we explain the three official patterns you can choose from, along with when to use each:

1. Deterministic Workflows

Deterministic workflows are straightforward, fixed sequences of steps to accomplish a task. In this pattern, the flow of execution is predetermined by code (not by the LLM's reasoning). One agent’s output feeds directly into the next agent, and so on, in a linear pipeline.

- *Characteristics:* A **fixed sequence** of operations; each step is well-defined and does not vary. No branching logic based on content – it's always "do X, then Y, then Z".
- *When to Use:* Ideal when the problem can be broken into clear sub-tasks and the path to the goal is known in advance. If the end goal and method are certain, a deterministic chain is both **simpler** and **more predictable** in terms of cost, speed, and outcome.
- *Example Use-Case:* **Data Processing Pipeline** – e.g., first an agent extracts data from a document, then a second agent analyzes that data. Another example: first generate a winter shopping list, then have another agent calculate the total cost. The steps and their order do not change.

Code Snippet (Deterministic Pattern): Below, one agent generates a list of winter clothing items, and a second agent deterministically calculates the total cost of those items. The workflow always executes the *Shopping List Agent* first, then passes its output to the *Cost Calculator Agent*:

```
from pydantic import BaseModel

# Define structured output models for clarity
class ShoppingItem(BaseModel):
    name: str
    price: float
    url: str
```

```
class ShoppingList(BaseModel):
    items: list[ShoppingItem]

class TotalCostSummary(BaseModel):
    total: float
    item_count: int
    details: str

# Define agents
winter_agent = Agent(
    name="Winter Shopping List Agent",
    instructions="""
        You are an agent that generates a random shopping list for a complete
winter outfit.
        Include various items like a jacket, scarf, gloves, hat, boots, etc.
        For each item, provide a name, a realistic price, and a placeholder URL.
        Return the list as a ShoppingList.
    """,
    output_type=ShoppingList
)
cost_agent = Agent(
    name="Total Cost Agent",
    instructions="""
        You are a deterministic agent. Given a ShoppingList, calculate the total
cost and item count,
        and return a TotalCostSummary with a details string in the format:
'Total for X items: $Y'.
    """,
    output_type=TotalCostSummary
)

# Execute sequentially (deterministic flow):
shopping_list_result = await Runner.run(winter_agent, input="")
shopping_list = shopping_list_result.final_output

cost_result = await Runner.run(cost_agent,
input=shopping_list.model_dump_json())
summary = cost_result.final_output

print("Summary:", summary.details)
```

In this deterministic flow, the sequence is fixed: generate list → calculate costs. A sample output might be:

```
Winter Shopping List:
- Winter Jacket: $120.99 (https://example.com/winter-jacket)
- Wool Scarf: $25.50 (https://example.com/wool-scarf)
... (more items) ...

Summary:
Total for 6 items: $297.23
```

(Output truncated for brevity. In the actual run, the list agent produced 6 items, and the cost agent summed their prices to produce the final total.)

2. Agents as Tools

The **Agents-as-Tools** pattern uses a **central orchestrator agent** that can invoke other specialist agents as *tools* to complete sub-tasks, without handing over full control. Unlike deterministic pipelines, the orchestrator *decides at runtime* which agent/tool to call next based on intermediate results or the problem's needs.

- *Characteristics:* The orchestrator agent **never relinquishes control**; it uses other agents like function calls. The sequence of calls can vary: which sub-agent is invoked, how many times, and in what order can depend on the content of the task and partial results. This often involves the main agent doing some reasoning/planning step (sometimes called a "planner" agent).
- *When to Use:* Use this when you have a complex, open-ended task requiring multiple skills or tools, and the exact steps aren't a straight line. The LLM can **plan and select** the appropriate tools/agents dynamically. It's great for scenarios where different inputs may require different workflows, or iterative refinement is needed.
- *Example Use-Case:* **Research & Synthesis Agent** – an agent that given a query, decides to use a Web Search tool if needed, then perhaps a Calculator tool, etc., to produce an answer. Or consider an **outfit recommendation agent** that must gather trends, optimize for budget, check quality, and so on, by invoking specialist sub-agents for each aspect.

Code Snippet (Agents-as-Tools Pattern): Below is a simplified illustration: an orchestrator agent uses two sub-agents as tools – one to fetch information and another to compose an answer. This pattern uses the `tools=[...]` parameter, adding each sub-agent via the `.as_tool()` method:

```
# Define a simple information-fetching agent as a tool (e.g., uses web search)
search_tool_agent = Agent(
    name="WebSearchAgent",
    instructions="You are an agent that can search the web for information on a
query and return findings.",
    tools=[WebSearchTool()],  # integrate OpenAI's hosted web search tool
    output_type=str  # assume it returns a text with search results summary
)

# Define an answer composing agent
answer_agent = Agent(
    name="AnswerComposerAgent",
    instructions="You take search findings and craft a concise answer for the
user query, with citations if possible.",
    output_type=str
)

# Orchestrator agent that decides when to search and when to answer
orchestrator = Agent(
    name="OrchestratorAgent",
    instructions="""
        You are a knowledgeable assistant who can use tools to answer questions.
        1. If the user question needs external information, use the Search tool
to find relevant facts.
        2. Once you have enough information, use the AnswerComposer tool to
formulate the final answer.
        Always return the final answer to the user when done.
    """,
    model_settings=ModelSettings(reasoning={"summary": "auto"}),  # enable
reasoning steps (chain-of-thought)
    tools=[
        search_tool_agent.as_tool(tool_name="Search", tool_description="Find
information on the web."),
        answer_agent.as_tool(tool_name="AnswerComposer",
tool_description="Compose the final answer.")
    ]
)

result = await Runner.run(orchestrator, input="What are the latest Mars rover
findings?")
print(result.final_output)
```

In this pattern, the *OrchestratorAgent* will autonomously decide to call the `Search` tool (which triggers the WebSearchAgent) to gather data, then pass those results to the `AnswerComposer` tool (AnswerComposerAgent) to generate the answer. The orchestrator *plans these steps itself* as it's prompted to do so. The intermediate reasoning chain and tool usage are handled by the SDK under the hood.

For instance, given a question about Mars rover findings, the orchestrator might output a final answer like:

```
According to recent mission updates, NASA's Perseverance rover found evidence of
ancient organic compounds in Martian rocks, suggesting Mars had conditions
suitable for life in the past. It also collected several rock samples from the
Jezero Crater that scientists plan to return to Earth. These findings indicate a
watery history and possible habitability of ancient Mars.
```

(In this hypothetical answer, the orchestrator decided the question needed a web search. It used the Search tool to retrieve information (e.g., news on Perseverance's findings) and then employed the AnswerComposer tool to produce a succinct summary. The orchestrator's prompt ensured it cites sources and only provides the final answer once tools have done their part.)

Why "Agents as Tools"? This pattern adds flexibility. The LLM (within the orchestrator agent) can conditionally use any of the provided tools/agents based on the situation – e.g., skip a step if it's not needed, repeat a tool if results are insufficient, or handle errors. It essentially treats sub-agents like function calls, **without handing off full control** as a deterministic handoff would. This is useful for orchestration where the agent might loop through tools until some success criteria is met, making the workflow more **adaptive**.

> **Note:** In the Agents SDK, using `Agent.as_tool()` wraps an agent as a tool that the orchestrator can invoke via function calling. OpenAI's SDK also allows directly using **hosted tools** (like `WebSearchTool`, `CodeInterpreterTool`, etc.) without making a custom agent for them. For example, one could directly put `tools=[WebSearchTool()]` in an agent to let it call the web, as we did inside `search_tool_agent` above. The orchestrator could also have used a built-in Calculator or Code tool similarly if needed.

3. Parallel Agent Execution

The **Parallel Execution** pattern involves running multiple agents *simultaneously* (concurrently) to speed up processing or generate diverse results that can later be merged. Instead of sequential or orchestrated calls, parallel agents work independently on subtasks, and then their outputs are combined by a final step.

- *Characteristics:* **I**ndependent agents address different aspects of a task (or the same task in different ways) at the same time, often using Python concurrency (e.g., `asyncio.gather`) to run them in parallel threads or async tasks. There is typically an **aggregator** agent or code that waits for all results and then composes a final result.
- *When to Use:* Best when sub-tasks do not depend on each other's immediate results (no need to wait for one agent before starting another). This pattern shines for **speeding up multi-part problems** (reducing latency by concurrent execution) or for **exploring variations** (having multiple agents attempt the same task in different ways and then picking the best result).
- *Example Use-Case:* **Document Summarization** – split a long document into sections and have multiple summarizer agents work on different sections in parallel, then have a final agent or function combine the section summaries into one cohesive summary. Another example: gather answers from multiple sources or perspectives concurrently, then unify them. In an e-commerce case, an orchestrator might query several pricing APIs or recommendation models in parallel to get a variety of items, then aggregate them.

Code Snippet (Parallel Pattern): In Python, you can leverage `asyncio` to run agents concurrently. For instance, imagine we want to summarize different aspects of a product review (features, pros/cons, sentiment, etc.) in parallel, then merge them:

```python
import asyncio

# Define multiple specialized agents (for brevity, instructions are abstracted)
features_agent = Agent(name="FeaturesAgent", instructions="Extract key
features...", output_type=str)
pros_cons_agent = Agent(name="ProsConsAgent", instructions="List pros and
cons...", output_type=str)
sentiment_agent = Agent(name="SentimentAgent", instructions="Analyze overall
sentiment...", output_type=str)
recommend_agent = Agent(name="RecommendAgent", instructions="Give a
recommendation with stars...", output_type=str)

# Input text to summarize (e.g., a product review)
review_text = "<LONG REVIEW TEXT>"

# Run all four agents in parallel on the same review text
results = await asyncio.gather(
    Runner.run(features_agent, input=review_text),
    Runner.run(pros_cons_agent, input=review_text),
    Runner.run(sentiment_agent, input=review_text),
    Runner.run(recommend_agent, input=review_text)
)
# Each result is a RunnerResult; get final outputs
features_summary = results[0].final_output
pros_cons_summary = results[1].final_output
sentiment_summary = results[2].final_output
recommendation = results[3].final_output

# Aggregate the summaries (could use another agent or just format them)
combined_summary = f"""**Features:** {features_summary}\n\n**Pros & Cons:**
{pros_cons_summary}\n\n"""
combined_summary += f"""**Sentiment:** {sentiment_summary}
\n\n**Recommendation:** {recommendation}"""
print(combined_summary)
```

Here, four agents operate concurrently, each focusing on a different summary aspect. The `asyncio.gather` call ensures they run at the same time, and we only proceed once *all* have completed. This drastically reduces total latency compared to running them one-by-one. After that, we combine their outputs (either via simple string formatting as above, or by passing all results to a final aggregator agent for a more nuanced integration).

Parallel Variation – Planner with Parallel Tools: The Agents SDK also supports a hybrid approach where an agent can be configured to execute multiple tools in parallel if the model supports it. For example, setting `ModelSettings(parallel_tool_calls=True)` allows an orchestrator agent to call multiple tool-agents simultaneously (the model decides to invoke them together). This is a more advanced scenario and can introduce complexity in collating results, but it's another way to leverage parallelism – letting the LLM plan parallel steps when appropriate.

Step-by-Step Guide: Building Your Multi-Agent Project

Follow these steps to create your own multi-agent system in Colab. Each student should choose *one* of the above patterns for their project, based on the scenario they want to tackle. We'll outline a generic workflow and illustrate with examples:

Step 1: Choose Your Use-Case and Select a Pattern

Begin by deciding **what problem or scenario** your agents will solve, and **which pattern** best fits the solution. Consider the nature of the task:

- If the task breaks down into a **clear, linear sequence of sub-tasks**, a **Deterministic Workflow** is likely best. (E.g., data extraction → analysis → report generation.)
- If the task requires **dynamic decision-making**, multiple tools, or may loop until criteria are met, use **Agents as Tools**. (E.g., an agent that decides to use search and calculation tools in whatever order needed to answer a complex question.)
- If the task can be **parallelized** to save time (or you want results from multiple sources), choose **Parallel Execution**. (E.g., querying multiple databases or sources at once and then merging answers.)

Justify the Pattern: Write down *why* the chosen pattern suits your scenario. For instance: *"I chose the Agents-as-Tools pattern for a travel planning assistant because the assistant needs to decide when to use tools like FlightSearch, WeatherInfo, etc., and the sequence can vary with user requests. A deterministic script would be too rigid for this."* Being clear on this helps guide your implementation.

Step 2: Build a Baseline Agent (or Agents)

Start coding the simplest version of your agent system that can handle the task, using the chosen pattern:

- **For Deterministic Workflow:** Implement each step as a separate agent or function, then chain them. For example, if doing "research then summarize", you might have `research_agent` and `summarize_agent`, and you call them one after the other. At first, keep prompts/basic instructions minimal just to get any output flowing through the chain.
- **For Agents as Tools:** Create the primary orchestrator agent and the specialist sub-agents (tools). Initially, you can set the orchestrator to use one tool at a time in a simple way. Keep the instructions simple (but clear about when to use which tool). Ensure you add the sub-agents via `tools=[agent.as_tool(...), ...]`.
- **For Parallel Execution:** Implement all the sub-agents for each parallel task. Test each one individually first. Then write the code (using `asyncio` or similar) to run them concurrently. Initially, you might simply print each agent's output without aggregating, just to verify they run.

Include at least one prebuilt tool: No matter the pattern, integrate at least one of OpenAI's *prebuilt tools* into your flow. This could be:

- A **hosted tool** like `WebSearchTool`, `FileSearchTool`, `ComputerTool`, `CodeInterpreterTool`, etc., which OpenAI provides for agents. For example, you can give an agent internet search capability by adding `tools=[WebSearchTool()]`.
- Or a custom **function tool** (Python function exposed to the agent). E.g., define a Python function to do a calculation or access a local resource, and use `my_func_tool = Tool.from_function(my_function)` (if using SDK's utility) or via the function calling interface.

For instance, to add a web search ability to an agent, you might do:

```
from agents import WebSearchTool
assistant_agent = Agent(
    name="ResearchAssistant",
    instructions="You can search the web to gather information before
answering.",
    tools=[WebSearchTool()],
    output_type=str
)
```

Now `assistant_agent` can call out to the web when needed. The same can be done for other tools (like adding `CodeInterpreterTool()` to run Python code for calculations, etc.).

> **Tip:** When designing prompts for an agent with tools, explicitly mention the tool names and capabilities in the instructions. For example: *"You have a tool 'Calculator' to do math, and a tool 'WebSearch' to look up facts. Use them whenever appropriate. Use the tools rather than guessing."* This clarity helps the LLM decide to use the functions.

- **Define Output Schema:** It's highly recommended to define `output_type` for each agent (using Pydantic models or basic Python types). This ensures the agent's response is properly structured (the SDK will validate and parse the LLM's output into that type). This makes it easier to pass data between agents. For example, in our baseline shopping list example, we defined `ShoppingList` and `TotalCostSummary` models and set those as output types, so the SDK gave us `result.final_output` already as a Python object with attributes. This avoids fragile string parsing.

Baseline Code Example: Here's a mini baseline example for a **deterministic** pattern use-case: a two-step agent that (1) searches the web for a topic, and (2) summarizes the search results:

```
# Step 1 agent: Web search (using built-in tool)
search_agent = Agent(
    name="WebSearcher",

instructions="Use the web to find information on the user's query. Provide a
summary of findings.",
    tools=[WebSearchTool()],
    output_type=str
)
# Step 2 agent: Summarizer
summarizer_agent = Agent(
    name="Summarizer",
    instructions="Summarize the given text into a concise answer for the user.",
    output_type=str
)

# Baseline execution: Always search, then summarize
```

```
query = "Latest developments in renewable energy storage"
search_result = await Runner.run(search_agent, input=query)
found_info = search_result.final_output

summary_result = await Runner.run(summarizer_agent, input=found_info)
answer = summary_result.final_output

print("Answer:", answer)
```

Even with minimal prompts, this will produce some answer. The quality might not be great initially – that's okay. The goal is to have a **working end-to-end pipeline** or agent loop that we can **iteratively refine**.

Step 3: Test and Observe the Baseline

Run your baseline system on a few sample inputs (or the main task prompt). In Colab, you'll see the printed outputs or any errors/exceptions.

- **Verify basic functionality:** Did each agent produce output of the expected type? (If not, adjust the `output_type` or prompt format. A common hiccup is the LLM not outputting valid JSON for the structured type – you might need to clarify the format in instructions or use a less strict type during development.)
- **Check the content:** Is the final result on topic? Are there obvious mistakes or missing pieces? For example, if your summarizer returns an overly brief summary that misses key points, note that.
- **Logging & Debugging:** Use `print` statements to output intermediate results (as we did above with `found_info`) to inspect what each agent is doing. The Agents SDK may also log tool usage or reasoning steps in the console. Pay attention to those, as they can reveal if the agent got confused or took a wrong turn.

If something is breaking (e.g., an agent's output can't be parsed into the `output_type`), you might need to adjust the prompt to ensure the format, or use `try/except` around `Runner.run` for debugging. At this stage, aim to get a end-to-end run without errors.

Step 4: Iteratively Improve Your Agents

This is where you refine prompts, settings, and even add agents or tools to achieve better performance. Iteration is key in agent development. Here are ways to improve:

- **Prompt Refinement:** Provide more explicit instructions, examples, or constraints in the system prompt (`instructions`). A well-crafted prompt can significantly improve output quality. For instance, if the output was too verbose, instruct the agent to be more concise. If it missed details, emphasize those details in the prompt.
- **Add Reasoning (if using LLM orchestration):** The Agents SDK can let the model "think" before final answer. For an orchestrator agent, enabling `ModelSettings(reasoning={"summary": "auto"})` as we did, causes the LLM to generate a reasoning trace (chain-of-thought) that isn't part of the final answer. This often improves complex decision-making. You might see the agent output thoughts like "*I should use the Search tool first because the query is about current events.*" – which

indicates it's working properly. You generally don't need to manually parse these, but they help the model organize its plan.

- **Incorporate Feedback Loops:** For some tasks, you might implement a simple loop: e.g., an agent tries to solve a task, then another "judge" agent evaluates the output, and if it's not good enough, you loop back and let the first agent try again with some feedback. This “judge pattern” (an evaluator providing feedback) can greatly improve results through self-correction. For example, after summarization, a second agent could check if the summary contains specific keywords or answers the question, and if not, prompt the summarizer to refine.
- **Employ Additional Tools:** If your agent struggled due to lack of information or skills, consider adding another tool. Example: If a single WebSearch wasn’t enough (maybe you need multiple searches or a calculation), you could add a custom `CalculatorTool` or call the search agent multiple times from the orchestrator.
- **Optimize Parallelism:** If you used parallel execution and find that the aggregation is weak (e.g., the final combination is disjointed), you might add a final agent to intelligently merge results rather than a naive concatenation.

After each change, **test again** with the same inputs and compare outputs:

- Did the output get more accurate, more complete, more structured?
- Is it faster now (especially if you added parallelism or better tools)?
- Check that you didn’t introduce new issues (like maybe the prompt is now too restrictive or the agent always uses the tool even when not needed).

Document these iterations: keep notes on what you changed and why, and observe the effect. This will be crucial for your analysis and final justification.

Step 5: Justify Pattern Choice and Iterations

Throughout the process, maintain a rationale for your decisions:

- **Why this Pattern:** Articulate why the chosen agent pattern is appropriate for the scenario. For example, *“Using a deterministic workflow ensured a reliable sequence for data processing, and our task didn’t require dynamic tool use, making it the simplest effective choice.”* Or, *“We chose the agents-as-tools pattern because the agent needed to flexibly decide whether to use the web or not depending on the question; a fixed script would either waste calls or miss needed info.”*
- **What Improved and Why:** After iterative tweaks, summarize what got better. For instance, *“By adding the WebSearchTool and refining the prompt with an example, the agent’s answers included up-to-date facts with sources, whereas initially it was guessing.”* If you introduced a feedback loop, note how it helped (e.g., *“The critic agent caught when the summary lacked a conclusion, prompting an improvement in the next iteration.”*). This reflection not only solidifies your understanding, but also is essential for the learning goals of the project – demonstrating you can analyze and enhance an agent’s performance.

Step 6: Capture and Review Execution Traces

A unique feature of the OpenAI Agents SDK is its **tracing capability**, which can visually show the flow of your multi-agent system. Every time you run an agent or workflow with OpenAI’s API, a trace is recorded on the OpenAI platform (if logged in with your API key). These traces illustrate each step: which agent ran, what tools were called, what inputs/outputs were at each stage, etc.. Reviewing traces can greatly aid debugging and understanding.

How to capture traces: By default, if your OpenAI API key is set, the runs should appear in OpenAI's tracing UI. You can label or group runs using the SDK's trace context manager. For example:

```
from agents import trace, gen_trace_id

trace_id = gen_trace_id()
# Group two runs under one trace ID (useful if your workflow involves multiple
separate Runner.run calls)
with trace(workflow_name="MyMultiAgentWorkflow", trace_id=trace_id):
    result1 = await Runner.run(orchestrator, input="First input")
with trace(workflow_name="MyMultiAgentWorkflow", trace_id=trace_id):
    result2 = await Runner.run(orchestrator, input="Second input")
```

This will log both runs under a single workflow trace named "MyMultiAgentWorkflow" on the OpenAI platform. In most simple cases, you can just do one `Runner.run` inside a `with trace(...):` block, or even omit the explicit `trace` block and rely on default tracing per run.

After running your agents, visit the OpenAI platform's **Monitoring/Trace** section. You should see a timeline of calls. For example, a trace might show:

- **Agent:** OrchestratorAgent – *Prompt sent to model...*
- **Function Call:** `Search` tool called with query "Mars rover Perseverance findings"
- **Agent:** WebSearchAgent – (the query ran, returned some snippets)
- **Agent:** OrchestratorAgent – *LLM reasoning... decides to use AnswerComposer next*
- **Function Call:** `AnswerComposer` tool called with the text from WebSearchAgent
- **Agent:** AnswerComposerAgent – returns a composed answer.
- **Final Output:** (the answer text)

This step-by-step trace is extremely useful. It **"shows the execution flow step by step, indicating which agent was executed and the LLM request/response, including function calls"**. By inspecting the trace, you can verify that:

- The orchestrator chose the intended tools or path.
- Each agent's input and output look as expected (no truncation or format errors).
- The number of iterations or parallel branches matches your design.

If something looks off in the trace, you can further tweak your agents. For example, if you see the orchestrator agent *didn't* call a tool when it should have, you might refine its prompt instructions to more strongly encourage that action.

Step 7: Final Testing and Wrap-Up

Once satisfied, do a final run with a **fresh example** or two (perhaps a slightly different input) to ensure your system generalizes. Save the final Colab notebook with all code, outputs, and written analysis of improvements.

Double-check that you've included:

- **Comments or Markdown** explaining each part of your code.
- Reasoning for each design choice (pattern, tools used, prompt phrasing, etc.).
- Examples of input and output (screenshots or copied text) demonstrating the agent's behavior before and after improvements.

This will make your work clear to others (and to yourself when presenting or writing about it).

Step 8: File/Notebook Organization

- All work is done in a single Colab notebook for this project.
- **Sections in Notebook:**
- Setup (installation, imports, API key),
- Definition of agents (code),
- Execution and testing,
- Iteration logs/notes,
- Conclusion analysis.
- **Code vs Markdown:** Use Markdown cells for explanations (document your thought process), Code cells for agent implementation and execution.
- Optional: If using any external data files (unlikely in this project), note how they are accessed (e.g., mounted Drive or via URLs).