ChatGPT

Healthcare Patient Journey – Deterministic Agent Workflows

In this tutorial, we explore a healthcare **patient journey** optimization use case implemented with the OpenAI Agents SDK. We will walk through a baseline solution and an enhanced solution – **both following deterministic agent workflows** – and explain how they work. Along the way, we clarify key concepts:

- **Simple agent:** a deterministic flow with a single output and no tool use.
- **Function-calling agent:** a deterministic agent that can invoke functions (tools) for enhanced capabilities (e.g. data analysis, chart generation, report synthesis).
- **Deterministic pipeline:** chaining multiple agents in a fixed sequence where each agent's output feeds into the next (no dynamic branching) [1].
- How to evolve the healthcare use case from a simple agent into a function-enhanced deterministic pipeline, explaining each agent, data class, tool, context, and output_type with sample code.
- Finally, we discuss how this deterministic flow could be varied into other **agent orchestration patterns**: a **parallel execution** pattern (running sub-tasks concurrently) [2] and an **agents-as-tools** pattern (using specialized agents as callable tools under a central orchestrator) [3].

Each section below builds on the previous, with beginner-friendly explanations, code snippets, and diagrams illustrating the concepts.

Simple Agent: Deterministic Single-Output (No Tools)

A **simple agent** in the OpenAI Agents SDK is essentially an LLM-powered agent with a fixed prompt (instructions) that produces a single output directly, without calling any external tools or functions. The agent's behavior is deterministic in the sense that its workflow is predefined: it takes an input and produces an output in one step (no intermediate branching or tool use).

Characteristics of a simple agent:

- **Fixed instructions:** You provide a system prompt or instructions that guide the agent's response style or task.
- **No tool usage:** The agent does *not* call any functions or external APIs. It relies solely on the model's internal knowledge and reasoning.
- **Single-turn output:** The agent receives an input (which could be a user query or data) and directly produces a final answer (text or structured output) in one go.

Example: A trivial example of a simple agent might be one that always responds with a greeting. We configure an agent with instructions and a model, but no tools:

```
from agents import Agent
```

```
greeting_agent = Agent(
    name="GreetingAgent",
    instructions="You are a polite assistant. When greeted, respond with a
friendly greeting.",
    model="gpt-3.5-turbo",   # or any appropriate model
    output_type=str          # output will be a simple string
)

# Run the agent synchronously for demonstration:
result = greeting_agent.run_sync("Hello agent!")  # passing a user message as
input
print(result.final_output)
```

In this code, `GreetingAgent` has deterministic behavior: given an input ("Hello agent!"), it produces a single output (e.g. "Hello! How can I assist you today?"). No functions or tools are invoked; the flow is a straightforward single-step LLM response.

Function-Calling Agent: Deterministic Workflow with Tools

A **function-calling agent** extends the simple agent by incorporating tools (functions) that the agent can call to perform specific operations. Even though the agent has the flexibility to call functions, the overall workflow can remain deterministic if the sequence of operations is predetermined by how we set up the agent and prompt. In other words, the agent is still part of a fixed pipeline, but within its step it might invoke tools to fetch data or perform calculations. The LLM's function-calling ability allows it to delegate certain tasks to *deterministic functions*.

Key points of a function-calling deterministic agent:

- It has one or more **tools** (Python functions or API calls) registered. Tools are often decorated with `@function_tool` from the SDK to expose them to the agent.
- The agent's instructions typically guide it to use those tools for specific tasks. (You can even force the agent to use a tool via model settings if needed.)
- The presence of tools does not introduce randomness in the workflow – the agent will use them as instructed, and the overall flow (input -> [optional tool calls] -> output) is still predetermined by design, not by the agent's autonomous decision-making.

For example, suppose we want an agent that can calculate simple math using a tool. We could define a function tool and an agent like so:

```
from agents import function_tool, Agent

@function_tool
def add_two_numbers(a: int, b: int) -> int:
    return a + b

calc_agent = Agent(
```

```
    name="CalculatorAgent",
    instructions="You can add two numbers using the provided tool. Always call
the tool to compute the sum.",
    model="gpt-3.5-turbo",
    tools=[add_two_numbers],
    output_type=str
)

result = calc_agent.run_sync("What is 40 plus 2?")
print(result.final_output)  # Expected output: "42"
```

Here, `CalculatorAgent` is deterministic: given the prompt, it will call the `add_two_numbers` function (as instructed) to get the result and then output "42". The agent's reasoning is not open-ended; it has a specific task (addition) and a fixed way to accomplish it (call the provided tool). The use of a tool is *within* the agent's single step, making it a *function-calling agent*. Importantly, the sequence of actions is predictable (receive question -> call function -> return answer), so this still fits the deterministic workflow pattern [1].

Baseline Deterministic Workflow – Patient Journey Use Case

Now, let's apply these concepts to a healthcare scenario. The goal of our **patient journey** use case is to analyze hospital patient data and produce recommendations to improve efficiency. We'll start with a **baseline deterministic pipeline**: a fixed sequence of three agents, each performing a sub-task, passing their outputs along to ultimately produce a final recommendation. This pipeline uses structured data input and simple function tools for deterministic analysis at each step.

Overview of the baseline workflow:

1. **Structured input:** We define a `PatientJourney` data class that holds patient visit information (admission/discharge times, wait times for various steps, and a flag if the patient was readmitted). This will serve as a structured context for the agents.
2. **Wait Time Analysis agent:** An agent that identifies any excessive wait times using a tool function.
3. **Readmission Check agent:** An agent that checks if the patient was readmitted, also using a function tool.
4. **Recommendation agent:** A final agent that takes the outputs of the first two agents and generates a summary recommendation.

Each of these agents is deterministic and specialized. The control flow (wait analysis -> readmission check -> recommendation) is hard-coded in our pipeline. There is no branching or skipping; we always execute all three in order.

Structured Data Class for Patient Journey

To provide input in a structured way, we use a Python dataclass `PatientJourney`. This encapsulates the patient's journey information:

```
from dataclasses import dataclass

@dataclass
class PatientJourney:
    admission_time: str
    discharge_time: str
    wait_times: dict            # e.g. {"triage": 40, "consultation": 75,
"labs": 30}
    readmitted_within_30_days: bool
```

This dataclass will allow us to pass the patient's data as a single object (context) into our agents. By using a dataclass (or Pydantic model), we can also enforce that the agent's `context` is of type `PatientJourney`. This context object will be available to any tools or agents that need patient info during the run.

Analysis Functions as Tools

Next, we implement two simple analysis functions as tools:

- `analyze_wait_times`: Checks the wait times in the context and returns a note about any steps that have delays over 60 minutes.
- `check_readmission`: Checks the readmission flag in the context and returns a note indicating if the patient was readmitted within 30 days.

We decorate these with `@function_tool` so that agents can call them. They use the `RunContextWrapper[PatientJourney]` to access the `PatientJourney` context passed into the agent run:

```
from agents import function_tool, RunContextWrapper

@function_tool
def analyze_wait_times(wrapper: RunContextWrapper[PatientJourney]) -> str:
    # Access the PatientJourney context
    journey = wrapper.context
    # Identify any wait times over 60 minutes
    high_waits = [step for step, t in journey.wait_times.items() if t > 60]
    if high_waits:
        return f"Bottlenecks found in: {', '.join(high_waits)}"
    return "No major wait time bottlenecks detected."

@function_tool
def check_readmission(wrapper: RunContextWrapper[PatientJourney]) -> str:
    journey = wrapper.context
    if journey.readmitted_within_30_days:
```

```
        return "  Readmitted within 30 days"
    return "  No readmission"
```

Both functions deterministically produce an output based on the given data: - **Wait time tool:** returns a specific message listing which stages (if any) exceed 60 minutes. - **Readmission tool:** returns one of two fixed messages depending on the Boolean flag.

These are straightforward computations with no randomness. By using them as tools, we offload the logic from the LLM – the agent doesn't need to "think hard" or have prior knowledge to detect long waits; it simply calls the tool which performs the check instantly.

Defining Specialized Agents

With our tools ready, we define three agents:

- **WaitTimeAgent:** Uses `analyze_wait_times` tool to get a wait time analysis message.
- **ReadmitAgent:** Uses `check_readmission` tool to get a readmission message.
- **RecommendationAgent:** No tools – it will just take the outputs from the first two agents and synthesize a recommendation.

Each agent has instructions tailored to its sub-task, an appropriate model, and an `output_type` of `str` since each produces text output. We also provide the `PatientJourney` type as the context type for the first two agents (so they can use the context in their tools):

```
from agents import Agent

wait_agent = Agent[PatientJourney](
    name="WaitTimeAgent",
    instructions="Analyze wait times and identify any delays over 60 minutes
using the patient journey information provided.",
    model="gpt-4-0613",          # Using a GPT-4 model for richer analysis
(example model name)
    tools=[analyze_wait_times],
    output_type=str
)

readmit_agent = Agent[PatientJourney](
    name="ReadmitAgent",
    instructions="Check if the patient was readmitted within 30 days using the
provided patient journey data.",
    model="gpt-4-0613",
    tools=[check_readmission],
    output_type=str
)

recommendation_agent = Agent(
```

```
    name="RecommendationAgent",
    instructions="Given the wait time findings and readmission status, generate
a final summary recommendation for improving the patient journey.",
    model="gpt-4-0613",
    output_type=str
)
```

A few things to note in this configuration:

- Both **WaitTimeAgent** and **ReadmitAgent** include the respective function tool in their `tools` list. Their instructions also hint that they should use the patient data (from context) to perform the analysis. Since these agents have the `PatientJourney` context, when they call the tool, the tool can access the context directly.
- **RecommendationAgent** has no tools. It will rely on the text outputs from the first two agents (which we will supply as its input) to craft a final recommendation. We use the same model for simplicity; it could be a smaller model if needed, since it's just summarizing two strings in this case.
- We haven't explicitly set `output_type` for the context-based agents beyond specifying the generic type in `Agent[PatientJourney]`. The outputs of those will default to `str` (as specified). For the recommendation agent, we explicitly set `output_type=str` as well.

Orchestrating the Pipeline

Now we put it all together in a deterministic sequence. We use the `Runner.run` method from the SDK to execute each agent in turn, passing along the necessary inputs and context:

```
from agents import Runner

async def run_pipeline(journey: PatientJourney):
    print("🔍 Step 1: Wait time analysis...")
    wait_result = await Runner.run(wait_agent,
input="Analyze the patient's wait times.", context=journey)
    print("   Output:", wait_result.final_output)

    print("🔍 Step 2: Readmission check...")
    readmit_result = await Runner.run(readmit_agent, input="Check readmission
status.", context=journey)
    print("   Output:", readmit_result.final_output)

    print("  Step 3: Generating final recommendation...")
    # Prepare a combined input for the recommendation agent:
    combined_input = [wait_result.final_output, readmit_result.final_output]
    final_result = await Runner.run(recommendation_agent, input=combined_input)
    print("\n  Recommendation:\n", final_result.final_output)
```

Let's break down what happens here:

- We call `Runner.run(wait_agent, ...)` with the patient journey as context. The WaitTimeAgent will call `analyze_wait_times` (using the context) and produce a message about any delays. We print the result.
- Next, we call `Runner.run(readmit_agent, ...)` similarly. The ReadmitAgent calls `check_readmission` on the context and returns a message about readmission.
- Finally, we call `Runner.run(recommendation_agent, ...)` with a **list** of inputs: `[wait_result.final_output, readmit_result.final_output]`. In the OpenAI Agents SDK, if you pass a list as the `input`, it treats each item as a separate message in the conversation. Essentially, we are providing the recommendation agent with the two previous outputs as if they were prior messages or facts to consider. The agent (with its instruction to generate a summary recommendation) will then produce the final output by considering those two pieces of information.

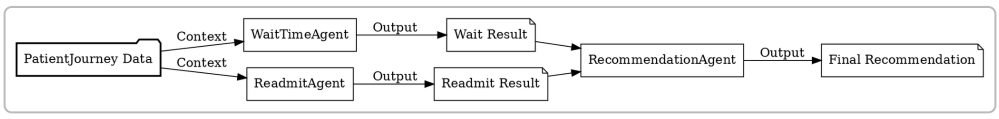
Since this is an asynchronous flow (using `await`), we would run `run_pipeline(example_input)` within an event loop (e.g., in an async context or Jupyter notebook). For simplicity, assume `example_input` is a `PatientJourney` instance we created with some sample data:

```
example_input = PatientJourney(
    admission_time="2025-06-10T10:00:00",
    discharge_time="2025-06-12T14:00:00",
    wait_times={"triage": 45, "consultation": 80, "labs": 25},
    readmitted_within_30_days=True
)
```

Running the pipeline on this sample data might produce output like:

- *Step 1 (Wait time analysis):* **"Bottlenecks found in: consultation"** (since consultation wait was 80 minutes, >60).
- *Step 2 (Readmission check):* **" Readmitted within 30 days"** (because the flag was True).
- *Step 3 (Recommendation):* A detailed multi-point recommendation string that addresses the long consultation wait and the readmission, suggesting improvements (e.g. reviewing triage protocols, enhancing post-discharge support, etc.).

Determinism: Throughout this baseline run, note that the flow did not branch or loop. It strictly executed Step 1, then Step 2, then Step 3. Each agent did its job using deterministic logic (either via the tools or straightforward summarization). The content of the final recommendation will be influenced by the LLM (so it could word things differently each run), but the *workflow* – which tasks are done and in what order – remains fixed.



Baseline deterministic workflow for the patient journey use case. The PatientJourney data is provided as context to two analysis agents (WaitTimeAgent and ReadmitAgent). Each produces a result string ("Wait Result" and "Readmit

Result"), which are then passed to the RecommendationAgent to generate a final recommendation. The sequence of agents is fixed and does not change based on content.

Enhanced Deterministic Workflow – Tools, Structured Output, and Reporting

In the baseline, we dealt with a single patient's data (hardcoded) and produced a text recommendation. Now, we enhance the workflow to handle more data and produce more structured analysis. Even with these enhancements, we will maintain a deterministic pipeline structure. The improvements include:

- **Batch data analysis with a code execution tool:** We use the OpenAI `CodeInterpreterTool` to perform analysis on a batch of patient records (simulating analyzing multiple patients at once). This allows complex computations (like averaging wait times, identifying overall trends) within the agent.
- **Structured output using data classes:** The analysis agent will output a `TrendAnalysisResult` data structure instead of plain text, making it easier to use the results in subsequent steps.
- **Report synthesis function:** We add a function tool to format the structured results into a nicely formatted report (which could then be emailed or presented).
- **External action (email via MCP):** Optionally, demonstrating an external action by sending the report via email (using the SDK's MCP integration). This uses a deterministic call to an external service – while interesting, it's an extra step that doesn't introduce nondeterminism in the agent flow.

The enhanced flow still proceeds linearly: **generate data -> analyze trends -> format report -> (send email)**. Each step is predetermined by our code; the agent doesn't deviate from these steps.

Trend Analysis Agent with Code Interpreter

First, imagine we have many patient journeys and we want to find overall trends (average wait times, which services are most often bottlenecked, readmission rate, etc.). Instead of manually coding these calculations in Python, we can let an agent use the **CodeInterpreterTool** – a tool that allows the agent to execute Python code in a sandbox to analyze data.

Synthetic data: For demonstration, we generate 100 synthetic patient records and save them as a CSV (in practice, this could be real data). The CSV has columns corresponding to the fields of `PatientJourney` (admission_time, discharge_time, wait_times as a JSON string or similar, readmitted_within_30_days). We won't show the data generation code here, but assume we have a file `"synthetic_patient_data.csv"` with 100 records.

TrendAnalysisResult class: We define a data class to structure the output of our trend analysis agent:

```
@dataclass
class TrendAnalysisResult:
    avg_wait_times: Dict[str, float]        # average wait time per service
(e.g., triage, consultation, labs)
    delays_over_60min: Dict[str, int]       # count of records where each
service had >60 min wait
```

```
    readmission_rate_percent: float
# percentage of patients readmitted within 30 days
    summary_text: str                       # a summary of findings or
additional commentary
```

This class will be used as the `output_type` for our trend analysis agent, so the agent will attempt to produce a JSON/object matching this structure. Using a structured output ensures we get parseable results (not just free-form text), which is very useful for subsequent tools (like the report generator).

TrendAnalyzer agent: Now we create an agent to analyze the CSV data. We give it the `CodeInterpreterTool` in its tools, enabling it to run Python code for data analysis and visualization. The instructions for this agent will explicitly tell it what to compute and remind it to output in the `TrendAnalysisResult` format:

```
from agents import Agent, CodeInterpreterTool, AgentOutputSchema

trend_agent = Agent[TrendAnalysisResult](
    name="TrendAnalyzer",
    instructions=(
        "Analyze the patient journeys CSV file and return:\n"
        "1. Average wait times per service (triage, consultation, labs)\n"
        "2. Services most frequently delayed (>60 min wait)\n"
        "3. Readmission rate as a percentage of all patients\n"
        "Also generate a bar chart for average wait times.\n"
        f"Provide the result in the format {TrendAnalysisResult.__name__}"
    ),
    model="gpt-4-0613",
# a capable model to interpret instructions and perform coding
    tools=[ CodeInterpreterTool(tool_config={"type": "code_interpreter"}) ],
    output_type=AgentOutputSchema(TrendAnalysisResult, strict_json_schema=False)
)
```

A few details here:

- We use `AgentOutputSchema(TrendAnalysisResult, strict_json_schema=False)` to tell the agent to produce output matching the `TrendAnalysisResult` structure. Setting `strict_json_schema=False` gives the agent some flexibility in formatting (it doesn't have to be exact JSON, as long as it can be parsed into the dataclass).
- The instructions clearly enumerate what needs to be computed. This helps the agent plan the code it will write in the Code Interpreter.
- We included "generate a bar chart for average wait times" – the Code Interpreter can create plots. When it does, the resulting plot image can be returned (or displayed) as part of the tool execution result. In a Jupyter environment, the chart would display inline. For our purposes, the chart is a side-effect; the main output of interest is the data.

- The agent will be given the CSV content as input when we run it. Typically, we could provide a path or the file content. In the SDK, one common approach is to read the file into a string and pass it as the input, so the agent (via Code Interpreter) can then write it to a file in the sandbox or directly parse it.

When we execute this `trend_agent`, under the hood the LLM will likely produce some Python code to read the CSV (using pandas, for example), compute the required metrics, maybe generate a matplotlib chart for the averages, and output the results as a JSON/dict. The SDK's CodeInterpreterTool will run that code and capture its outputs. The final agent output will then be parsed into a `TrendAnalysisResult` object.

This approach shifts the heavy lifting to a controlled Python execution, which is *deterministic* given the same input data and code. The only non-deterministic element is the LLM coming up with the code, but since it's constrained by the clear instructions and the data, repeated runs should yield the same logical results (the code might vary in style but does the same job).

Report Synthesis with Function Tool

Once we have a `TrendAnalysisResult` from the trend analysis agent, the next step is to turn those findings into a human-friendly report (in text form). We can write a function to do this formatting, which ensures a consistent structure for the report. This function doesn't need any ML – it's purely string formatting based on the data class.

For example, the report might list average wait times per service, the counts of delays, the readmission rate, followed by a brief summary text. Here's our `synthesize_report` function tool:

```
@function_tool(strict_mode=False)
def synthesize_report(wrapper: RunContextWrapper[TrendAnalysisResult]) -> str:
    result = wrapper.context  # TrendAnalysisResult object from context

    # Format the average wait times
    avg_wait_lines = "\n".join(f"- {service.title()}: {minutes:.2f} min"
                               for service, minutes in
result.avg_wait_times.items())
    # Format the delay counts
    delay_lines = "\n".join(f"- {service.title()}: {count} occurrences"
                            for service, count in
result.delays_over_60min.items())
    # Compose the report text
    return f"""📋 **Healthcare System Summary Report**

**Average Wait Times:**
{avg_wait_lines}

**Delays Over 60 Minutes:**
{delay_lines}

**Readmission Rate:**
{result.readmission_rate_percent:.1f}%
```

```
**Summary:**
{result.summary_text}

Regards,
*AI Monitoring System*
"""
```

This tool produces a nicely formatted markdown text (with some basic styling like bold sections and bullet points). We mark `strict_mode=False` because we are fine with the function returning a string that may contain characters the LLM didn't explicitly specify (we're directly generating the string in code, not via the model). The function simply reads the `TrendAnalysisResult` from the context and uses its fields to build the report text.

Integrating Report Generation (and Optional Email Step)

Finally, we want to integrate this report generation into our pipeline. We have a couple of options:

- We could bypass the LLM entirely for this step and just call `synthesize_report` directly in our code, since it's a deterministic function. However, for demonstration (and if we wanted the LLM to maybe add some narrative flair in the summary text), we can involve an agent.
- We create a **ReportAgent** that will call the `synthesize_report` tool. This agent's job is just to invoke the tool (ensuring the data is formatted) and perhaps to wrap it as an email. We can even hook up an **MCP (Multichannel Plugin) server** to actually send the email via an external service like Zapier.

Let's illustrate using an agent approach with MCP for email. (MCP allows the agent to perform actions like web requests or sending emails via configured channels – here we use a Zapier Gmail webhook as an example.)

```
from agents.mcp import MCPServerSse
from agents import ModelSettings

# Assume we have an MCP endpoint URL (for Zapier email) in environment variable
mcp_url = os.environ.get('MCP_SERVER_URL_SSE')
mcp_server = MCPServerSse(name="MCP_Email", params={"url": mcp_url},
client_session_timeout_seconds=30)

report_agent = Agent(
    name="EmailAgent",
    instructions=(
       "Generate a professional email summary by calling the synthesize_report
tool on the analysis results. "
       "The email should be addressed to the hospital administrator (e.g.,
Anjali Jain) and should exclude any raw charts or images."
    ),
```

```
    model="gpt-4-0613",
    tools=[synthesize_report],
    mcp_servers=[mcp_server],
    model_settings=ModelSettings(tool_choice="required"),  # force using the
tool
    output_type=str
)
```

What's happening here:

- We configure an `MCPServerSse` with the Zapier webhook URL (this will handle sending the email when triggered).
- The `report_agent` has the `synthesize_report` tool. We set `tool_choice="required"` in model settings to force the agent to use the tool. Essentially, the agent *must* call `synthesize_report` to get the email content; it won't try to free-form write the whole email by itself.
- The presence of `mcp_server` means the agent can output through that channel. In practice, when the agent produces a final output, it will be sent via the MCP server to Zapier, resulting in an email being sent.

To run this final part of the pipeline:

```
# Trend analysis step - we pass the CSV text as input to the trend_agent
csv_text = open("synthetic_patient_data.csv", "r").read()
analysis_result = await Runner.run(trend_agent, input=csv_text)
print("Trend analysis output:", analysis_result.final_output)  # This is a
TrendAnalysisResult object

# Engage the MCP server context (setup connection)
await mcp_server.__aenter__()

# Report/email step - run the report agent with the TrendAnalysisResult as
context
email_result = await Runner.run(report_agent,
    input="Please draft the summary report email.",
    context=analysis_result.final_output  # pass the TrendAnalysisResult object
as context
)
print("  Email content:\n", email_result.final_output)

# Clean up the MCP server session
await mcp_server.__aexit__(None, None, None)
```

The above code will cause the report agent to call `synthesize_report` with the given context (our trend results). The output (a nicely formatted email text) is both printed and sent via the MCP server to the

configured email address. The final output might look like a well-structured email report containing bullet-point stats and a brief summary of what the data indicates.

Despite involving more steps and even an external email action, this enhanced workflow is still **deterministic**: it always goes **data -> analysis agent -> report agent** in the same order. Each agent's role is predefined and does not change. The usage of tools (CodeInterpreter, function tool) does not introduce non-determinism in the flow; it just provides more capabilities within that flow.

Beyond Deterministic Pipelines: Parallel and Agents-as-Tools Variations

The solutions we described above use a **deterministic workflow design pattern**, meaning a fixed sequence of agent calls orchestrated by our code [1]. This is ideal when each step logically follows the previous and there's no need for the agents to dynamically decide the flow. However, the OpenAI Agents SDK supports other orchestration patterns that can be beneficial in different scenarios [3] [2]. In this section, we briefly outline how one might transform the above deterministic pipeline into two other patterns: **Parallel Execution** and **Agents-as-Tools**. We won't provide a full implementation, but we'll discuss the necessary changes and benefits of each variation.

Variation 1: Parallel Execution Pattern

In the deterministic pipeline, our wait time analysis and readmission check happen sequentially (Step 1 then Step 2). But note that those two tasks are independent – one does not need the result of the other. We could perform them *concurrently* to save time. The **parallel execution pattern** involves running multiple agents simultaneously and then combining their outputs [2]. This can reduce overall latency, especially if using relatively slow LLM calls, by leveraging concurrency.

How to implement parallel execution: Instead of awaiting each `Runner.run` one after the other, we can launch them together and wait for both to finish:

```
import asyncio

async def run_pipeline_parallel(journey: PatientJourney):
    # Kick off both agents concurrently
    wait_task = Runner.run(wait_agent, input="Analyze waits", context=journey)
    readmit_task = Runner.run(readmit_agent, input="Check readmit",
context=journey)
    # Wait for both results
    wait_result, readmit_result = await asyncio.gather(wait_task, readmit_task)
    print("Wait analysis output:", wait_result.final_output)
    print("Readmission output:", readmit_result.final_output)
    # Then proceed to the recommendation agent as before
    final_result = await Runner.run(recommendation_agent,
                                    input=[wait_result.final_output,
```

```
readmit_result.final_output])
    print("Final recommendation:", final_result.final_output)
```

In this parallel version, the two analysis agents run at the same time. The `asyncio.gather` collects both results once ready. The final recommendation step waits until it has both pieces, then runs. This pattern is particularly useful if, for example, each analysis call took a couple of seconds – running them concurrently could almost halve the total time.

Important considerations in parallel pattern: - Agents should be thread-safe or async-safe if they share any global resources (not a problem in our example as each call is independent). - If using the same model for both calls, you are effectively making two LLM calls in parallel – which is fine if your setup and API support that concurrency. - This pattern works best when sub-tasks do not depend on each other's output [4], which is true in our case. If tasks were dependent (e.g., step 2 needs step 1's result), you cannot parallelize them straightforwardly.

Variation 2: Agents-as-Tools Pattern

The **Agents-as-Tools** pattern involves using a **central orchestrator agent** that can invoke other agents as if they were tools or functions [3]. In this pattern, we don't manually sequence the calls in code. Instead, we rely on one agent (often a "manager" or "planner" agent) to decide *at runtime* which specialized agent to call next, based on the situation. The orchestrator never hands off control completely; it uses the other agents' capabilities by calling them like subroutines, and it can iterate or choose different paths dynamically [5].

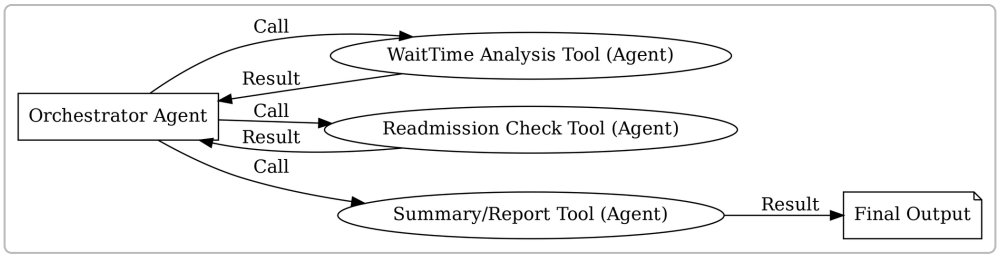
How to implement agents-as-tools: In our use case, we could wrap the functionality of WaitTimeAgent, ReadmitAgent, and RecommendationAgent into tools that an orchestrator agent can call. There are a couple of ways to do this:

- **Expose agent functionality via function tools:** For example, write a function `run_wait_analysis(journey: PatientJourney) -> str` that internally calls `Runner.run_sync(wait_agent, ...)` and returns the result string. Decorate this function with `@function_tool`. Do the same for a `run_readmission_check` function. Now an orchestrator agent can be given these two as tools. It could decide to call them (in any order it deems necessary) and get their outputs to use in composing a final answer.
- **Use agent handoffs or an agent-of-agents approach:** The SDK might allow an agent to directly use another agent as a tool (if supported), or one could incorporate the sub-agents as part of the context that the orchestrator can access (though the simpler method is wrapping them in functions as described above).

Imagine an **OrchestratorAgent** with instructions like: *"You are an expert healthcare analysis agent. You have tools to (1) analyze wait times, (2) check readmission, and (3) draft a summary recommendation. Use these tools as needed to produce a final comprehensive recommendation for a patient's journey."* This agent would have three tools: `analyze_wait_times` (or the wrapper function variant), `check_readmission`, and perhaps a `synthesize_summary` tool (which could combine results – or it might even just use the first two tools and then formulate the summary itself without a third tool).

During runtime, the orchestrator agent might do the following internally: 1. Call the wait analysis tool to get wait issues (if any). 2. Call the readmission tool to get readmission status. 3. Combine those findings (and optionally call a summary tool or just write the summary directly as the final answer).

The sequence and logic can be decided by the agent's prompts and the model's reasoning. For instance, if the wait times had no bottlenecks, the agent might choose to still mention that but focus more on readmission (the script could allow such conditional emphasis). If both tools returned issues, it ensures both are covered in the final output. The crucial point is that *the orchestrator agent controls the flow* instead of the Python code doing so.



Agents-as-Tools pattern illustrated. The OrchestratorAgent (central box) can call specialized agents as tools – in this case, a WaitTime Analysis tool, a Readmission Check tool, and a Summary/Report tool. The orchestrator decides the order of calls and uses the returned results to formulate the final output. Unlike the deterministic pipeline, this allows dynamic decision-making (for example, skipping a step if not needed, or repeating a step for clarification) while still keeping the orchestrator in control of the conversation flow [3] [6] .

When to use agents-as-tools: This pattern shines in more complex scenarios where the exact steps may vary or where the agent might benefit from planning its approach [6] . In our healthcare example, if we expanded the task to a more open-ended analysis (maybe the agent should decide what analyses to run based on patient data), an orchestrator could decide to invoke different tools. For instance, *if* the patient was readmitted, the agent might decide to do a deeper analysis of possible causes (invoking another tool or agent), whereas if not, it might skip that. In a deterministic pipeline, you'd run the same steps regardless. With agents-as-tools, the flow can branch or loop as needed, guided by the LLM's reasoning. This offers flexibility at the cost of some predictability and requires careful prompt design to ensure the orchestrator uses the tools wisely.

Conclusion

In this tutorial, we demonstrated how to build a baseline and an enhanced **deterministic agent workflow** for a healthcare patient journey optimization use case. We started from a simple agent concept and progressed to a multi-agent pipeline utilizing function tools and code execution for data analysis, all while maintaining a predictable linear flow. The baseline workflow handled one patient's data with straightforward analysis, and the enhanced workflow scaled to multiple records and introduced structured outputs and a report generation step – yet both workflows were deterministic, executing a fixed sequence of steps defined by our code (not by the AI's whim).

We also explored how the same problem could be approached with more advanced patterns: - **Parallel execution**, to perform independent analyses concurrently for efficiency. - **Agents-as-tools**, to allow a central agent to flexibly orchestrate sub-tasks at runtime.

Each pattern (Deterministic vs Parallel vs Agents-as-Tools) has its use cases and trade-offs. Deterministic workflows are simpler and more predictable [7], making them a great starting point (as we've done here). Parallel execution can improve performance when applicable [8]. Agents-as-tools offer dynamic problem-solving ability, which is powerful for complex tasks but requires robust prompting and sometimes more experimentation to get right [5].

By understanding these patterns and how to implement them, you can choose the best approach for your AI agent applications. In the context of our healthcare example, the deterministic approach gave us clarity and reliability in how each piece functions, while knowing the alternative patterns suggests ways to iterate and expand the solution in the future (e.g., handling more diverse inputs or optimizing for speed). We encourage you to try these patterns in your own use cases and see how an orchestrated agent can streamline even complex workflows in a maintainable way.

---

[1] [2] [3] [4] [5] [6] [7] [8] OPENAI AGENT SDK - Full Code Assignment.pptx
file://file-NJiM5zfBhUoGA8c6i4mffD