ChatGPT

OpenAI Agents SDK – Hands-On Examples and Experiments

This repository contains a collection of practical examples, implementations, and educational code snippets using the OpenAI Agents SDK. The projects here demonstrate how to build autonomous AI agents with OpenAI's framework – including how to configure agent behavior, use tools, delegate tasks between agents, and incorporate structured outputs. It's a hands-on learning playground for key concepts like tools, handoffs, tracing, and agent workflows.

Installation

To set up this project, we recommend using the **uv** package manager (a fast Python package manager by Astral) instead of pip. Follow these steps:

1. Clone the repository:

   ```
   git clone https://github.com/DoniaBatool/openai_agent_sdk.git
   ```

2. Navigate into the project directory:

   ```
   cd openai_agent_sdk
   ```

3. Install **uv** (if not already installed). You can install uv globally using pip (this is the only time we'll use pip):

   ```
   pip install uv
   ```

4. Create the environment and install dependencies using uv. This will create a virtual environment (as specified by `.python-version`) and install all required packages (like the OpenAI Agents SDK, etc.) as locked in `uv.lock`:

   ```
   uv sync
   ```

   Alternatively, you can add packages manually (not usually needed if you ran the sync). For example:

   ```
   uv add openai-agents python-dotenv
   ```

   (The command above would add the OpenAI Agents SDK and Dotenv to the project.)

5. Configure API keys: create a `.env` file in the project root (this file is already listed in `.gitignore`) and add your OpenAI API key:

   ```
   OPENAI_API_KEY=<your_openai_api_key>
   ```

   Ensure you **do not commit** your API key or any secrets to version control. The code will automatically read this key from the environment when running.

Now you're ready to run the examples.

Repository Contents

Below is a detailed overview of every file and folder in this repository, along with what each contains or demonstrates, and key concepts/keywords associated with them:

File/Folder	Description	Key Concepts
`.chainlit/`	Configuration folder for Chainlit UI (if using Chainlit to run agents in a chat interface). It may contain UI settings or cached data for Chainlit.	Chainlit integration, UI config
`.gitignore`	Lists files & directories to be ignored by Git (e.g., `.env`, virtual environments, `__pycache__`, etc.), helping to keep API keys and build files out of version control.	Version control ignore rules
`.python-version`	Specifies the Python version (e.g. 3.x) for the project. This ensures consistency in the development environment (used by tools like uv or pyenv).	Environment management
`README.md`	(You are here!) Repository documentation and guide, describing the project, setup, file contents, usage examples, and best practices.	Documentation
`chainlit.md`	Markdown content for Chainlit, typically displayed as an app description or welcome message when running a Chainlit UI. It can provide instructions or context in the chat interface.	Chainlit app info
`requirements.txt`	An alternative list of Python dependencies (for pip users). Contains packages required (e.g., `openai-agents`, etc.). This is mostly for reference – uv users rely on `pyproject.toml` and `uv.lock`.	Dependencies, pip support

File/Folder	Description	Key Concepts
`pyproject.toml`	Project configuration file defining project metadata and dependencies. Notably, it lists required packages like OpenAI's Agents SDK. uv uses this to manage the environment.	Project config, dependencies
`uv.lock`	Lockfile generated by uv, pinning exact versions of dependencies for reproducible installs. Ensure this is updated when dependencies change.	Dependency lockfile
`ComputerTool.py`	Example demonstrating use of the built-in **Computer Tool** – which allows an agent to perform computer operations in a sandbox. Shows how an agent can execute local/sandboxed commands or calculations.	Tools, `ComputerTool`, automation
`WebSearchTool.py`	Example of using the built-in **Web Search Tool**. This agent can search the web and return results. Demonstrates enabling internet access for an agent through OpenAI's web search integration (no external API key needed for search).	Tools, `WebSearchTool`, web access
`fileSearch.py`	Demonstrates the built-in **File Search Tool**. The agent can search within local files or contents. Useful for retrieval from documents. Shows how to incorporate file searching capability in an agent.	Tools, `FileSearchTool`, local file retrieval
`hello.py`	The classic "Hello World" of the Agents SDK. Creates a simple agent (with a basic instruction like "You are a helpful assistant") and asks it a sample question (e.g., to write a haiku or greet the user). Useful as a quick smoke test to verify the SDK is working and to illustrate minimal agent setup.	Basic agent, simple prompt, quick start

File/Folder	Description	Key Concepts
`include_usage.py`	Shows how to retrieve and use token usage information from an agent run. After running an agent, it prints out usage stats (tokens used, etc.) or includes usage info in the output. This is helpful for monitoring costs and optimizing prompts. (It might also demonstrate setting usage limits to prevent too many tokens from being consumed.)	Token usage, usage stats, cost monitoring
`metadata.py`	Explores how to attach or utilize metadata in agent runs. This could involve adding custom metadata to prompts or tools (for logging or context), or retrieving metadata from the agent's result (e.g., model identifiers or timestamps). It's an educational snippet to show how metadata can be handled or logged in the SDK workflow.	Metadata handling, context info, logging
`mycode.py`	A personal scratchpad for experimenting with the SDK. This file contains miscellaneous tests or a mix of features as the author was learning (it might not correspond to a single concept). Use it to see various ideas or drafts, such as trying out multiple tools or prompts in one place.	Miscellaneous experiments, (multiple concepts)
`agent_flow.py`	Explores the control flow of an agent solving a task. It runs an agent through a multi-turn interaction (including tool use or handoffs) and prints/logs each step. Use this to understand how the Agents SDK's runner loop processes prompts, tool calls, and handoffs in sequence.	Agent loop, flow control, multi-step reasoning
`agent_flow.ipynb`	Notebook counterpart to `agent_flow.py`. Allows you to run the agent step-by-step and inspect the intermediate states in a Jupyter environment. Helpful for learning and debugging the agent's decision process.	Agent workflow, interactive tracing

File/Folder	Description	Key Concepts
`code_gemini.py`	An example using Google Gemini (or another non-OpenAI model) with the Agents SDK. This shows how to configure an agent to use an alternative LLM provider (via the SDK's provider-agnostic interface). It illustrates the SDK's flexibility with models like Gemini (e.g., through OpenAI's APIs or OpenRouter).	Custom model provider, Gemini, model flexibility
`default_model.py`	Simple test of the SDK's default model behavior. Creates an agent without explicitly specifying a model to see which model is used by default. This helps clarify what the Agents SDK chooses as a default (e.g., GPT-3.5 Turbo) and how to override it.	Agent default model, model selection
`handoff1.py`	Basic handoff example: implements a simple triage scenario with multiple agents. For example, an agent that hands off Spanish queries to a "Spanish Agent" and English queries to an "English Agent." Illustrates how `Agent.handoffs` can be used to delegate a user query to another agent based on some condition (in this case, language).	Handoff, triage agent, multi-agent workflow
`handoffs2.py`	Custom handoff configuration: builds on the handoff concept with more customization. Uses the `handoff()` function to create a handoff tool explicitly – allowing custom tool names or descriptions. (For instance, an agent might have two handoff targets like a "BillingAgent" and "SupportAgent," and we override the tool name/description for clarity.) Shows multiple handoff targets in one agent.	Multiple handoffs, `handoff()` utility, custom tool naming

File/Folder	Description	Key Concepts
`handoff3.py`	Handoff with input data (escalation): demonstrates an advanced handoff where the agent passes structured data to the next agent. For example, a customer service agent might escalate an issue to a supervisor agent with an **EscalationData** payload (containing a reason for escalation). Utilizes `RunContextWrapper` and an `on_handoff` callback to handle additional input (like the escalation reason) during the handoff.	Handoff with input, escalation, `RunContextWrapper`, callbacks
`handoff4.py`	Handoff with filters/advanced logic: an extended handoff example that might include input filtering or conditional delegation. For instance, it could demonstrate using an input filter to decide if a handoff should occur (maybe only escalate if confidence is low, etc.). This example showcases fine-grained control over when and how handoffs happen, and how to filter or preprocess input before handing off.	Conditional handoff, input filters, advanced delegation
`parallel_tool_call1.py`	Experiment on parallel tool calls (Part 1). This example tests how the agent or model might handle invoking multiple tools in parallel. For instance, if an LLM's response suggests using two tools at once or in rapid succession, how does the SDK handle it? The script may simulate or force concurrent tool usage to see the behavior.	Parallel tool execution, concurrency, agent planning
`parallel_tool_call2.py`	Continuation of the parallel tools experiment (Part 2). Perhaps a variation or more controlled scenario of parallel tool usage. It could compare different approaches to achieve parallel calls or use the SDK's features (if any) to handle simultaneous actions. Together with part 1, it helps understand the limitations and possibilities of parallelizing agent tool calls.	Parallel calls, concurrent tools, advanced usage

File/Folder	Description	Key Concepts
`reasoning.py`	Focuses on the agent's reasoning process. It likely prompts the agent to explicitly outline its thought process or uses the SDK's trace events to display the chain-of-thought. This can show how the agent arrives at an answer (for instance, by printing intermediate reasoning or decision steps). It's useful for understanding and debugging the decision logic of the LLM.	Chain-of-thought, reasoning steps, intermediate output
`structured.py`	Demonstrates structured output from an agent. Uses Pydantic `BaseModel` classes to define the expected format of the answer. The agent is given an `output_type` so that its final answer conforms to a JSON/object structure (e.g., a dictionary with specific fields). This example shows how to get the model to return well-formatted data (for example, returning a weather report with specific fields filled in).	Structured output, Pydantic models, `output_type`
`tool_choice.py`	An agent with multiple tools available, illustrating how it chooses the right tool for a given query. For example, the agent might have a calculator tool and a search tool, and depending on the question it will decide which to invoke. This teaches how the prompt/instructions and the model's reasoning lead to tool selection. (It's essentially about giving the agent options and seeing how it decides.)	Multiple tools, dynamic tool selection, decision-making
`tracing1.py`	Introduction to tracing agent runs (Part 1). Runs an agent with tracing enabled and outputs the trace of steps. You'll see events like model requests, tool calls, and responses logged to the console or to the SDK's tracing UI. This helps visualize what the agent is doing under the hood at each step of the prompt-tool-handoff cycle.	Tracing, debug logging, run visualization

File/Folder	Description	Key Concepts
`tracing2.py`	Advanced tracing example (Part 2). This example demonstrates more fine-grained tracing or custom trace handling – such as manually creating spans or sending trace data to an external dashboard. It could also show how to trace asynchronous calls or multiple agents. Use this to learn how to instrument your agent with custom trace points or how to integrate the SDK’s tracing with other tools.	Advanced tracing, custom spans, debugging
`with_model.py`	Example of explicitly specifying a model when creating or running an agent. It shows usage of the `Agent(model="<provider:model-name>")` pattern or providing a `ModelSettings` to an agent/run. For instance, it might run the same prompt with GPT-4 versus GPT-3.5 to compare outputs. This highlights how to change models in the SDK.	Custom model selection, `ModelSettings` usage, model override

How to Run

After installing the dependencies and setting your API key, you can run each example individually using **uv**. The `uv run <script_name.py>` command will automatically activate the project environment and execute the script. Here are some usage tips and examples:

- **Basic usage:** Use `uv run <script_name.py>` to run any example. For instance:

  ```
  uv run hello.py
  ```

  This will execute the **hello.py** agent, and you should see the agent’s output (e.g. a haiku or greeting) in your terminal.
- **Running handoff examples:** Try running:

  ```
  uv run handoff1.py
  ```

  This will demonstrate the triage agent deciding which sub-agent responds (based on the query language). Feel free to modify the prompt inside the script to test different inputs (e.g., an English vs. Spanish query) and observe the handoff behavior.
- **Tool example scripts:** You can run specialized tool-use examples:
- `uv run WebSearchTool.py` – The agent will perform a web search query (this uses OpenAI's built-in web search tool, so ensure the agent has internet access via the API).

- `uv run ComputerTool.py` – The agent will attempt a simple computation or file operation using a sandboxed computer tool.
- `uv run structured.py` – The agent will output a structured JSON result following a Pydantic model (demonstrating structured output enforcement).
- **Using Chainlit (optional):** For an interactive chat UI, you can use Chainlit. First, install Chainlit in the env (`uv add chainlit`). Then run an agent script with Chainlit. For example:

  ```
  chainlit run hello.py -w
  ```

  Open the local host link that Chainlit provides in your browser. You'll see any content from `chainlit.md` as the welcome message. This allows you to chat with the agent via a web interface. (This is optional; all examples can be run via the console as shown above.)

  Note: Ensure your environment has the `OPENAI_API_KEY` set before running any agent. The Agents SDK uses this key to authorize API calls. If you created the `.env` file, uv will auto-load it. Otherwise, export the key in your shell (e.g., `export OPENAI_API_KEY=...`).

Highlights

This project is rich in learning outcomes. By exploring the code in this repo, you'll pick up key concepts and skills, including:

- **Agent Basics** – Setting up an agent with instructions and running it with a `Runner`. You'll see how an agent processes a simple prompt end-to-end.
- **Tool Usage** – Registering function tools (with the `@function_tool` decorator) and using built-in tools like web search or file search. Learn how agents can extend their capabilities beyond the base LLM by calling functions.
- **Multi-Agent Workflows (Handoffs)** – Coordinating multiple agents by handing off tasks. The examples show a "triage" pattern where one agent decides which specialist agent should handle a query, as well as how to escalate to a different agent with additional context. This models real-world scenarios where complex tasks are delegated.
- **Model Configuration** – Using the `ModelSettings` class and related parameters to fine-tune model behavior (e.g., adjusting `temperature` for creativity, or `max_tokens` for response length). Also understanding how to plug in different model providers (OpenAI vs others like Anthropic or Google Gemini) through the SDK's flexible model naming convention.
- **Structured Outputs** – Enforcing a response format by defining output schemas (with Pydantic) so that the agent's answer can be parsed as JSON or an object. This is crucial for building applications where the agent's output needs to feed reliably into other systems.
- **Tracing and Debugging** – The tracing examples illustrate how to monitor the internal decision-making of agents. You'll learn to trace each step, which is invaluable for debugging agent logic, ensuring the agent uses tools as expected, and improving prompt design by observing where the AI might get confused.
- **Parallel and Advanced Tool Use** – Although LLMs usually call one function at a time, the parallel tool-call experiments encourage thinking about concurrency: how might an agent handle or suggest multiple actions at once, and how can the SDK or developer manage such cases.

- **Agent as a Tool** – A clever pattern where entire agents are treated as callable tools, enabling composition of capabilities. This teaches how to build larger systems from modular agent components (e.g., having a main agent call a sub-agent for a specific skill).
- **Usage Monitoring** – Keeping an eye on token usage to manage cost and performance. By extracting usage stats after runs or setting limits, you learn to build agents that are mindful of API usage (important in production scenarios).

Best Practices

When building and experimenting with agent-based applications (as shown in this repo), consider the following best practices:

- **Secure your API keys:** Never hard-code API keys in your code or share them publicly. This repo uses a `.env` file (listed in `.gitignore`) to load the OpenAI API key safely. Always store secrets in environment variables or secure vaults. Commit only non-sensitive code to the repository.
- **Use virtual environments:** Maintain an isolated environment for your project (the provided `pyproject.toml` and uv workflow help with this). This ensures dependency versions are consistent. The included `.python-version` file pins a specific Python interpreter version to avoid compatibility issues.
- **Structure agents into specialized units:** Rather than one monolithic agent that does everything, follow a modular approach. For example, use a triage agent to route queries to specialized sub-agents (as shown in the handoff examples). This makes the system more transparent and easier to maintain or upgrade (you can tweak one agent’s instructions without affecting others).
- **Leverage handoffs and tools thoughtfully:** Handoffs (delegating to other agents) and tools (functions the agent can call) are powerful features. Use handoffs when a completely different skillset or role is needed for a subtask (e.g., escalate to a “Manager” agent), and use tools when the task is mechanical or external (e.g., perform a calculation, fetch web data). Always define clear instructions for when to use each tool or handoff so the AI knows how to choose appropriately.
- **Keep prompts and instructions clear and constrained:** Each agent’s instructions should state its role and what it can or cannot do (especially important if it has tools or handoff options). Clear instructions and guardrails (like specifying an output format or using an `output_type` for structured output) lead to more reliable behavior.
- **Monitor and limit usage:** As shown in `include_usage.py`, keep track of how many tokens your agent is using. In production, you might set usage limits (if available) or implement checks to prevent runaway costs or infinite loops. For example, use the `max_turns` parameter on `Runner.run()` or design your prompts to encourage the agent to finish in a reasonable number of turns.

License

This project is released under the MIT License. See LICENSE for full text.
