# OpenAI Agent Example (Tool Calling)

Minimal agent that uses the OpenAI Chat Completions API with tool-calling to answer questions by invoking small helper functions. Two variants are included: a Python script and a Jupyter notebook.

## What's Included

- `agent_example.py`: CLI + REPL agent that can call tools and produce a final answer.
- `agent_example.ipynb`: Notebook version with the same tools and a couple of demo cells.

## Requirements

- Python 3.9+
- Package: `openai`

Install:

```
pip install openai
```

Environment variable:

- `OPENAI_API_KEY`: your OpenAI API key
- Optional `OPENAI_MODEL` (defaults to `gpt-4o-mini`)

Example:

```
export OPENAI_API_KEY=sk-...
export OPENAI_MODEL=gpt-4o-mini
```

## Quick Start

Script mode:

```
python agent_example.py "What is 3*(7+5), and what time is it?"
```

Interactive REPL:

```
python agent_example.py
```

Notebook:

1) Open `agent_example.ipynb`.
2) Ensure `OPENAI_API_KEY` is set in your environment.
3) Run cells top-to-bottom. Try the "Quick test" cell.

## How It Works

The agent follows a simple two-call pattern using Chat Completions with tools:

1. Build the initial `messages` with a short system prompt and the user's question.
2. Call `client.chat.completions.create(..., tools=TOOL_SPECS, tool_choice="auto")`.
   - The model can choose to call one or more tools (function calls) by returning `tool_calls`.
3. If tool calls are present, the script executes each mapped Python function locally and appends a `tool` role message with the output for each call.
4. Make a second `chat.completions.create` call with the updated `messages` so the model can incorporate tool results and produce the final answer.

This pattern is implemented by `run_agent(...)` in both the script and the notebook.

#### System Prompt

A brief instruction that encourages the assistant to use tools when helpful and to cite results succinctly.

#### Tool Specs and Registry

- `TOOL_SPECS`: JSON schemas that describe each tool's name, description, and parameters; sent to the model.
- `TOOL_REGISTRY`: Python mapping from tool name to the actual function to run when the model requests it.

The model never executes code directly; it emits a structured request describing which tool to call and with what arguments. The script performs the call and returns the output back to the model.

## Provided Tools

- `get_current_time`: returns the current UTC time in ISO 8601 format.
- `calculator`: evaluates basic arithmetic expressions (numbers, +, -, *, /, **, parentheses). It uses a restricted AST-based evaluator to avoid executing arbitrary code.
- `search`: stub tool that returns a canned list of results. Replace with a real search integration if needed.

## Script Walkthrough (`agent_example.py`)

- Reads `OPENAI_API_KEY` from the environment and initializes `OpenAI()` client.
- Defines the three tools and registers them in `TOOL_REGISTRY`.
- `run_agent(user_input: str, model: str = "gpt-4o-mini")`:
  - Makes a first chat completion with `tools` enabled.

    - Executes requested tools, appends their outputs as `tool` messages.
    - Makes a second chat completion to produce the final answer.
- CLI usage accepts a one-off prompt or starts a REPL loop when no arguments are provided.

## Notebook Walkthrough (`agent_example.ipynb`)

- Setup cell: imports, client, and environment checks.
- Tool cells: define the same tools and `TOOL_REGISTRY`.
- Spec cell: defines `TOOL_SPECS` and the system prompt.
- Runner cell: defines `run_agent(...)` (same logic as the script).
- Demo cells: quick test and a customizable prompt.

## Extending the Agent

To add a new tool:

1. Write a Python function that accepts a dict of arguments and returns a string.
2. Add it to `TOOL_REGISTRY` with a unique name.
3. Add a matching entry in `TOOL_SPECS` (name, description, JSON schema for parameters).
4. The model can now discover and call your tool.

Other tweaks:

- Model: set `OPENAI_MODEL` or pass `model=` to `run_agent`.
- Style: adjust `SYSTEM_PROMPT` and `temperature`.
- Search: replace the stub `search` with a real API; ensure outputs are concise, ideally JSON.
- Error handling: wrap tool code with try/except and return user-friendly messages.

## Troubleshooting

- Missing key: ensure `OPENAI_API_KEY` is exported in your shell or environment.
- Tool errors: the agent will return the exception message from the tool; validate input schemas.
- Empty/odd outputs: reduce `temperature` and check your system prompt clarity.
- Rate limits/network: verify account limits and connectivity; consider retries/backoff for production.

## Security Notes

- The calculator uses a restricted AST evaluator to avoid executing arbitrary code. Do not replace it with `eval`.

- Treat model-provided tool arguments as untrusted input; validate and sanitize.
- Be mindful of sending sensitive data to remote services if you add external tools.

---

If you want a Responses API version, streaming, or JSON mode, those can be added with small changes to the runner—happy to help extend this further.

## Streaming Output (Chat Completions)

To stream the final answer (second call) using Chat Completions, set `stream=True` and print deltas as they arrive. Example pattern for the final step after tool outputs are appended:

```
stream = client.chat.completions.create(
    model=chosen_model,
    messages=messages,  # includes tool role messages
    temperature=0.2,
    stream=True,
)
for chunk in stream:
    delta = chunk.choices[0].delta
    if delta and getattr(delta, "content", None):
        print(delta.content, end="", flush=True)
print()  # newline after stream
```

You can keep the first call (where the model decides on tool calls) non-streaming, then stream only the final completion once tool results are available.

## Responses API Variant (Optional)

If you prefer the newer Responses API, the flow is similar:

- Replace `client.chat.completions.create(...)` with `client.responses.create(...)`.
- Provide the conversation as `input` rather than `messages` (the structure is a list of role/content blocks). Tools are still passed via `tools=[...]` with the same schemas.
- Execute any returned tool calls the same way and send results back before the final response call.

For streaming with the Responses API, use its streaming interface and print text deltas as they arrive. Refer to the OpenAI Python SDK documentation for the exact event fields your installed version exposes.

Notes when switching:

- Keep `TOOL_SPECS` unchanged; only the API surface and message vs. input shape differ.

- Some SDK versions provide convenience fields like `response.output_text` when not streaming.
- Validate your installed `openai` version against the examples (run `python -c "import openai, inspect; print(openai.__version__)"`).