OPENAI AGENTS SDK

Teaching Nuances & Non-Obvious Behaviors

Instructor Reference Guide | December 2024

1. THE FIVE CORE PRIMITIVES

A	H	G	S	T
Agents	**Handoffs**	**Guardrails**	**Sessions**	**Tracing**

Design Philosophy

Minimal but sufficient — Enough features to be useful, but few enough primitives to learn quickly. This is deliberately simpler than frameworks like LangChain.

2. WHY ASYNC IS DEFAULT (NOT JUST AN OPTION)

The Three Runner Methods

- `Runner.run()` — async, returns RunResult
- `Runner.run_sync()` — thin sync wrapper
- `Runner.run_streamed()` — async, real-time events

Why Async by Default

- LLM calls are **I/O-bound**, not CPU-bound
- Guardrails run in **parallel** by default
- Scales naturally to FastAPI deployments
- `run_sync()` is only for scripts/notebooks

3. THE AGENT LOOP (FOUNDATIONAL MENTAL MODEL)

1	2	3	4
Call LLM	**LLM Responds**	**Process**	**Loop/End**
Send agent + input + tools	Output, tool calls, OR handoff	Execute tools or switch agent	Continue or return final output

Critical Understanding

This is a **single-agent sequential loop**, NOT multi-agent orchestration. Even with handoffs, only ONE agent is active at any moment. Tool results are appended to history—the LLM sees its own work.

4. HANDOFFS VS. AGENTS-AS-TOOLS (MOST

CONFUSING DECISION)

Handoffs

- **Control transfers** to receiving agent
- Each agent "owns" the conversation
- Natural conversation flow
- Each agent has its own tools
- `handoffs=[agent_a, agent_b]`

Agents as Tools

- **Control retained** by calling agent
- "Manager coordinating workers" feel
- Calling agent combines results
- Calling agent must know all tools
- `tools=[agent.as_tool()]`

Teaching Recommendation

Most beginners should start with **handoffs**. The manager pattern is harder to get right and often unnecessary. Use handoffs for support triage → specialist. Use as_tool when you need to aggregate results from multiple agents.

5. GUARDRAILS: BOUNDARY VALIDATION ONLY

Input Guardrails

Only run if agent is the **FIRST** agent in the chain. Validates user input before expensive LLM processing.

Output Guardrails

Only run if agent is the **LAST** agent in the chain. Validates final output before returning to user.

Non-Obvious Behavior

Guardrails on **middle agents are ignored**. If Agent A hands off to Agent B, Agent B's input guardrails WON'T run because B is not the entry point. Also: tripwire = **exception raised**, not silent filter.

6. LOCAL CONTEXT VS. LLM CONTEXT (UNIVERSAL CONFUSION)

Local Context (YOUR Code)

```
@dataclass
class MyContext:
    user_id: str
    db: Database  # LLM NEVER sees this

@function_tool
def fetch(ctx:
RunContextWrapper[MyContext]):
    # ctx.context.db is available here
    return ctx.context.db.query(...)
```

LLM Context (What Model Sees)

```
agent = Agent(
    instructions="You are helpful."
    # THIS is what the LLM sees
)

# LLM only sees:
# - Instructions (system prompt)
# - Messages (conversation history)
# - Tool descriptions
```

Teaching Point

The `context` object is for **dependency injection** (your code). The LLM has no idea your database exists. Students constantly assume the context is sent to the model.

7. OUTPUT_TYPE CHANGES THE API CALL

Regular Agent

`agent = Agent()`

Uses standard chat completions. Faster and cheaper.

Structured Output Agent

`agent = Agent(output_type=MyModel)`

Uses structured outputs. **Slower and more expensive.** Only use when you need typed data.

8. TOOL USE BEHAVIOR MODES

When to Use Each Mode

Mode	Behavior	Use Case
`run_llm_again`	LLM processes tool result, may call more tools	Calculator (interpret result)
`stop_on_first_tool`	Tool result IS the final answer	Weather lookup (API = answer)
`StopAtTools([...])`	Stop only for specific tools	Some tools terminal, others not
Custom function	Your logic decides when to stop	Complex stopping conditions

9. BUILT-IN SAFETY FEATURES

Infinite Loop Prevention

If `tool_choice="required"`, SDK auto-resets to `"auto"` after each tool call. Controlled by `agent.reset_tool_choice`.

Max Turns Limit

Set `max_turns` parameter. Raises `MaxTurnsExceeded` exception if exceeded. Essential for production.

Tracing Always On

Every `Runner.run()` is traced by default. Data goes to OpenAI. Use `trace_include_sensitive_data=False` for secrets.

Handoff History Nesting

Prior conversation collapsed into `<CONVERSATION_HISTORY>` block. Prevents context explosion in multi-handoff chains.

10. SESSIONS: CONVERSATION MEMORY, NOT EXECUTION STATE

What Sessions DO

- Store conversation history between runs
- Auto-prepend history to next run
- Multiple backends: SQLite, SQLAlchemy, OpenAI

What Sessions DON'T Do

- Persist execution state/variables
- Carry over mid-run progress
- Accumulate token counts across runs

11. VOICE/REALTIME: FUNDAMENTALLY DIFFERENT

Text Agents

- Request-response model
- `await Runner.run(agent, input)`
- Supports `output_type`

Realtime Agents

- Persistent WebSocket connection
- Continuous audio event stream
- **No** `output_type` support
- Must handle `audio_interrupted`

12. WHEN TO USE WHAT (QUICK REFERENCE)

Decision Guide

Need	Use	Why
Multi-turn conversation	Session	No `.to_input_list()` boilerplate
Multi-agent (delegated)	handoffs	Natural flow, each agent independent
Multi-agent (coordinated)	as_tool()	Central control, aggregate results
Validate before processing	Input guardrail	Cheap check before expensive LLM
Structured response	output_type=Model	Type safety (but slower/costlier)
Real-time UI updates	run_streamed()	Token-by-token feedback

TEACHING RECOMMENDATIONS

- **Start with single agents + tools** before multi-agent patterns
- **Use handoffs by default** — save manager pattern for specific needs
- **Always show the agent loop diagram** — it's the foundation for everything
- **Distinguish context types early** — local context vs. LLM context causes universal confusion
- **Demo streaming** — it dramatically changes the user experience
- **Explain guardrails as "circuit breakers"** not filters — they raise exceptions