BUILDING AI AGENTS WITH THE OPENAI AGENTS SDK

A Complete Guide from Fundamentals to Production-Ready Systems

- AGENT FUNDAMENTALS
- TOOLS AND INTEGRATIONS
- MULTI-AGENT ORCHESTRATION
- SAFETY AND GUARDRAILS
- OBSERVABILITY AND EVALUATION
- DEPLOYMENT AND PRODUCTION PATTERNS

YURI ALEKSOV

Building AI Agents with the OpenAI Agents SDK

A Complete Guide from Fundamentals to Production-Ready Systems

Steve T. Publications

This book is available at
https://leanpub.com/buildingaiagentswiththeopenaiagentssdk

This version was published on 2026-07-13

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Contents

A Complete Guide from Fundamentals to Production-Ready Systems

This book teaches you how to build robust, scalable AI agent applications using the OpenAI Agents SDK. Starting from the very first line of code, you will progress through agent configuration, tool integration, multi-agent orchestration, safety guardrails, observability, and production deployment patterns. Every concept is explained with complete, runnable Python examples and detailed walkthroughs. The book assumes basic Python knowledge but no prior experience with AI agents or the OpenAI platform. By the end, you will be able to design and ship production-grade agentic systems with confidence.

Introduction: The Agent Paradigm

What Is an AI Agent?

A chatbot answers your question. An API performs a function when you call it. An agent does something more: it perceives its environment, reasons about what to do next, takes actions using tools, and iterates until a task is complete. This cycle of perception, reasoning, and action is what separates agents from everything that came before them.

Consider the difference between asking a chatbot "What is the weather in San Francisco?" and telling an agent "Plan my weekend trip to San Francisco." The chatbot gives you a weather report. The agent searches for flights, checks hotel availability, looks up restaurant recommendations, cross-references the weather with your activities, and assembles a complete itinerary. Each step requires the agent to decide what information it needs, call the right tool, interpret the results, and determine whether more work is required or whether the task is done.

This decision-making loop is the heart of agentic AI. The model does not just produce text: it produces tool calls that modify state, fetch data, or delegate work. Those tool results feed back into the model as new context, which triggers another round of reasoning and potentially more actions. This loop continues until the agent determines that no further action is needed and produces a final answer.

The OpenAI Agents SDK was built around this exact pattern. It provides the infrastructure to run this loop reliably: managing conversation history, executing tool calls, handling errors, tracking state across multiple agents, and surfacing the results in a structured way. Your job as a developer is to define what the agent should do (instructions), give it the capabilities it needs (tools), and set up the safety boundaries (guardrails). The SDK handles the rest.

Why the OpenAI Agents SDK?

When you look at the landscape of AI frameworks, you will find many options. LangChain offers a broad ecosystem of chains, memory systems, and integrations. AutoGen focuses on multi-agent conversations with configurable speaker patterns. CrewAI provides role-based agent teams with task assignment. Each has its strengths, but they also share a common characteristic: they introduce significant abstraction layers between your code and the underlying model calls.

The OpenAI Agents SDK takes a different approach. It deliberately keeps abstractions to a minimum. The core building blocks are simple: an Agent is an LLM configured with instructions and tools. A Runner executes the agent loop. Handoffs let agents delegate to peers. Guardrails validate inputs and outputs. That is essentially the entire surface area you need to understand before building production systems.

This minimalism is intentional. OpenAI released the Agents SDK as a production-ready upgrade of their earlier experimental framework, Swarm [1]. The design philosophy is that Python itself should be the orchestration language. Instead of learning a proprietary domain-specific language or wrestling with complex configuration objects, you write ordinary Python functions and pass them to agents as tools. You chain agents using standard control flow. You manage state using familiar patterns.

The trade-off is worth understanding. The Agents SDK does not attempt to solve every problem that other frameworks address. There is no built-in retrieval-augmented generation pipeline. There is no vector database integration layer. There is no automated prompt optimization service. The SDK expects you to build those capabilities yourself, using the tools and patterns it provides. This gives you more control and transparency at the cost of having to assemble more pieces yourself.

For developers who want to understand exactly what their agents are doing and maintain full control over the execution flow, this approach is a significant advantage. You can read the SDK source code and understand every line. You can trace every model call in the OpenAI dashboard. You can debug tool executions with standard Python debugging tools.

Core Primitives at a Glance

Before diving into implementation details, it helps to see the complete picture of what you will be working with. The Agents SDK is built on four foundational primitives:

Agents are the central abstraction. An agent wraps an LLM model with system instructions and optional capabilities. You define what the agent should do through natural language instructions, and you give it the ability to act through tools. Agents can also delegate work to other agents through handoffs or by treating other agents as callable tools.

Tools are how agents interact with the world. A tool is any action an agent can take: calling a weather API, searching the web, executing code, running a shell command, or even delegating to another agent. The SDK supports five categories of tools: hosted OpenAI tools (web search, file search, code interpreter, image generation), local runtime tools (shell commands, computer use, file patching), function tools (Python functions decorated with @function_tool), agents as tools (other agents exposed as callable capabilities), and MCP-backed tools (external services connected through the Model Context Protocol).

Handoffs enable decentralized multi-agent coordination. When an agent hands off to another agent, the receiving agent takes over the conversation entirely. This is different from calling a tool: the handoff target becomes the active agent for the remainder of the turn, with full access to the conversation history. Handoffs are ideal for routing conversations to specialized experts.

Guardrails provide safety and validation layers. Input guardrails screen user requests before any model processing begins. Output guardrails validate the agent's final response before delivery. Tool guardrails inspect function arguments before execution and results after completion. Each guardrail type can operate in different modes: input checks can run in parallel with the agent (for low latency) or block until validation completes (for cost optimization). When a guardrail tripwire is triggered, the run halts with a descriptive exception.

Beyond these four primitives, the SDK provides supporting infrastructure: sessions for persistent conversation memory, tracing for observability and debugging, streaming for real-time response delivery, context management for dependency injection, and human-in-the-loop approval workflows for

sensitive operations.

How to Use This Book

This book is organized as a progressive journey from fundamentals to advanced patterns. Each chapter builds on the concepts introduced in earlier chapters, so the recommended reading path is sequential. However, if you are already familiar with basic agent creation and want to jump to specific topics, the chapter titles should guide you to the right starting point.

The early chapters (1 through 3) cover the absolute essentials: installation, your first agent, configuration, and tool integration. By the end of Chapter 3, you will be able to build a single agent that can call external APIs, search the web, and execute code.

The middle chapters (4 through 8) dive into operational concerns: managing conversation state across turns, orchestrating multiple agents, implementing safety controls, and observing agent behavior in production. These chapters contain the patterns you will use most frequently in real applications.

The later chapters (9 through 12) cover advanced topics: human-in-the-loop workflows, sophisticated multi-agent topologies, MCP integration for external services, and production deployment strategies including testing, security, cost optimization, and durable execution.

Every code example in this book is complete and runnable. You can copy the examples directly into Python files and execute them with your own OpenAI API key. Each example includes comments explaining the important lines and design decisions, so you can understand not just what the code does but why it is written that way.

The book focuses on the Python SDK exclusively. While OpenAI also maintains a JavaScript/TypeScript version of the Agents SDK, the concepts and patterns described here transfer directly to the JavaScript implementation. The API surface area is nearly identical between the two versions.

[1] https://github.com/openai/openai-agents-python

Chapter 1: Getting Started with the Agents SDK

Installation and Environment Setup

Before writing any agent code, you need to set up your development environment. The OpenAI Agents SDK requires Python 3.10 or newer and is distributed as a PyPI package. Start by creating an isolated virtual environment for your project:

```
# Create a project directory and virtual environment
mkdir my-agents-project
cd my-agents-project
python -m venv .venv

# Activate the virtual environment
# On macOS/Linux:
source .venv/bin/activate
# On Windows (Command Prompt):
.venv\Scripts\activate

# Install the Agents SDK
pip install openai-agents
```

The `openai-agents` package pulls in several dependencies automatically, including the OpenAI Python client library, Pydantic for schema validation, and griffe for docstring parsing. You can verify your installation by checking the version:

```
import agents
print(agents.__version__)
```

If you encounter a `ModuleNotFoundError: No module named 'agents'` error despite successful installation, you likely have a local folder named `agents/` that is shadowing the installed package. Rename your project folder to something else (for example, `my_agents/`) and retry.

Setting Your API Key

The SDK reads your OpenAI API key from the `OPENAI_API_KEY` environment variable on every `Runner` invocation. Never hardcode API keys in source code. Instead, set them in your shell session or a `.env` file:

```
# Set for the current terminal session
export OPENAI_API_KEY=sk-proj-your-key-here

# Or use python-dotenv for development
pip install python-dotenv
```

```
# In your Python code, load from .env before importing agents
from dotenv import load_dotenv
load_dotenv()  # Reads .env file in the current directory
```

The SDK does not store or cache API keys. Each run reads the key fresh from the environment, which means you can swap keys between runs by changing the variable. This is useful for testing with different OpenAI organizations or projects.

Optional Dependencies

Some features require additional packages. Install them using pip extras:

```
# Redis-backed sessions
pip install openai-agents[redis]

# MongoDB sessions
pip install openai-agents[mongodb]

# Dapr integration
pip install openai-agents[dapr]

# Any-LLM third-party adapter
pip install openai-agents[any-llm]

# LiteLLM third-party adapter
pip install openai-agents[litellm]
```

You do not need these extras for the core agent functionality covered in this chapter. They become relevant when you explore session backends and non-OpenAI model providers in later chapters.

Your First Agent

With your environment ready, let us write a minimal agent. The simplest possible agent requires only two things: a name and some instructions. Here is the complete code:

```
1  import asyncio
2  from agents import Agent, Runner
3
4
5  async def main():
6      # Create an agent with a name and system instructions
7      agent = Agent(
8          name="History Tutor",
9          instructions="You answer history questions clearly and concisely.",
10     )
11
12     # Run the agent with a user prompt
13     result = await Runner.run(agent, "When did the Roman Empire fall?")
14
15     # Print the final output
16     print(result.final_output)
17
18
19 if __name__ == "__main__":
20     asyncio.run(main())
```

Let us walk through every line of this code.

Line 1: We import `asyncio` because the SDK is built around asynchronous execution. Agent runs involve network calls to the OpenAI API, and async/await is the standard Python pattern for handling I/O-bound operations efficiently.

Line 2: The two essential imports are `Agent` and `Runner`. The `Agent` class defines what your agent is: its identity, instructions, tools, and model. The `Runner` class executes agents by running the agent loop: calling the LLM, processing tool calls, feeding results back, and repeating until a final answer is produced.

Lines 6–9: We create an `Agent` instance with two parameters. The `name` parameter is required and serves as a human-readable identifier. It appears in traces, logs, and debugging output. The `instructions` parameter is the system prompt that guides the agent's behavior. While technically optional, you

should almost always provide instructions: they define what the agent knows, how it should behave, and what constraints to follow.

Line 12: We call `Runner.run()` with two arguments: the agent we just created, and a string representing the user's input. This is an asynchronous method, so we use `await` to wait for the result. The runner handles everything internally: it sends the instructions and user input to the OpenAI model, receives the response, and wraps it in a `RunResult` object.

Line 15: We access `result.final_output`, which contains the agent's final text response as a string. This is what you would display to a user in a chat interface.

When you run this script (with `OPENAI_API_KEY` set), you should see a concise answer about the fall of the Roman Empire. The exact wording varies between runs because LLMs are non-deterministic, but the response will be focused and factual thanks to our instruction.

Synchronous Alternative

If you prefer synchronous code or are working in an environment where async is inconvenient, the SDK provides `Runner.run_sync()`:

```
from agents import Agent, Runner

agent = Agent(
    name="History Tutor",
    instructions="You answer history questions clearly and concisely.",
)

# Synchronous run -- works the same way but blocks until complete
result = Runner.run_sync(agent, "When did the Roman Empire fall?")
print(result.final_output)
```

Under the hood, `run_sync()` calls `asyncio.run()` on `Runner.run()`. The behavior is identical; only the execution model differs. Use `run_sync()` for simple scripts and REPL sessions. Use `run()` (async) when building applications that need to handle concurrent requests or integrate with async web frameworks like FastAPI.

Understanding the Runner

The `Runner` class is the engine that drives agent execution. It provides three entry points, each suited to a different use case:

`Runner.run()` – The standard asynchronous method. Returns a `RunResult` object after the agent completes its work. This is what you will use most often in production applications.

```
result = await Runner.run(agent, "What causes lightning?")
print(result.final_output)
```

`Runner.run_sync()` – The synchronous equivalent. Returns a `RunResult` immediately (blocking the current thread). Useful for scripts, notebooks, and simple CLI tools.

```
result = Runner.run_sync(agent, "What causes lightning?")
print(result.final_output)
```

`Runner.run_streamed()` – Returns a `RunResultStreaming` object that provides an async iterator over events as they occur. This enables real-time response delivery: you can show partial text to users as the model generates it, display progress indicators during tool calls, and react to handoffs in real time.

```
result = Runner.run_streamed(agent, "What causes lightning?")

async for event in result.stream_events():
    if event.type == "run_item_stream_event":
        print(f"Event: {event.name}")
    elif event.type == "raw_response_event":
        # Raw token-level streaming from the model
        pass

# After consuming the stream, final_output is available
print(result.final_output)
```

The streaming API requires careful handling. You must fully consume the `stream_events()` iterator before accessing `result.final_output`. The SDK performs post-processing after the last token arrives: persisting session data, tracking approvals, and compacting history. If you break out of the stream early without calling `result.cancel()`, these cleanup operations may not complete properly.

The Agent Loop

Understanding what happens inside a `Runner.run()` call is essential for debugging and designing effective agents. The runner executes a loop with the following steps:

1. Call the LLM for the current agent, passing the instructions and accumulated input (user messages, tool results, conversation history).
2. Process the LLM's output:
    - If the LLM produces a text response with no tool calls, the loop ends and the text becomes `final_output`.
    - If the LLM produces tool calls, execute each tool call, collect the results, append them to the input, and go back to step 1.
    - If the LLM performs a handoff to another agent, switch to the new agent, update the input, and go back to step 1.
3. If the loop exceeds `max_turns` (default is determined by the model), raise a `MaxTurnsExceeded` exception.

This loop is what makes agents fundamentally different from simple chat completions. A single `Runner.run()` call can trigger multiple LLM invocations, each separated by tool executions. The agent "thinks" about what to do, takes action via tools, sees the results, and decides what to do next. This iterative process continues until the agent determines it has enough information to produce a final answer.

You can control the maximum number of turns by passing `max_turns` to any Runner method:

```
# Limit to 5 LLM calls total
result = await Runner.run(agent, "Research quantum computing", max_turns=5)

# Disable the turn limit entirely
result = await Runner.run(agent, "Research quantum computing", max_turns=None)
```

Setting a reasonable `max_turns` is a good practice for production applications. It prevents runaway agents from making excessive API calls if they get stuck in a reasoning loop. The default varies by model but is typically generous enough for most tasks.

The RunResult Object

Every Runner method returns a result object that captures everything that happened during the agent's execution. Understanding the `RunResult` interface is critical because it is your primary way of inspecting and working with agent output.

`final_output` – The final answer produced by the last agent in the run. This is either a string (if no structured output type was defined) or an instance of the agent's `output_type` class (if one was specified). If the run stopped before producing output (for example, due to an approval pause), this property is `None`.

```
result = await Runner.run(agent, "What is 2 + 2?")
print(result.final_output)  # "4" or a similar response
```

`new_items` – A rich list of all items generated during the run. Each item carries metadata about which agent produced it, what type of operation occurred, and any associated tool calls or handoffs. This is the most detailed view of what happened:

```
for item in result.new_items:
    print(f"Type: {type(item).__name__}")
    if hasattr(item, "agent"):
        print(f"  Agent: {item.agent.name}")
```

Common item types include `MessageOutputItem` (assistant text responses), `ToolCallItem` and `ToolCallOutputItem` (tool invocations and their results), `HandoffCallItem` and `HandoffOutputItem` (agent handoffs), and `ToolApprovalItem` (pending approvals).

`to_input_list()` – Converts the run's output into a list of input items suitable for passing to the next turn. This is how you build multi-turn conversations manually:

```python
# First turn
result1 = await Runner.run(agent, "My name is Alice.")
print(result1.final_output)

# Second turn -- carry forward the conversation history
new_input = result1.to_input_list() + [{"role": "user", "content": "What's my
↪  name?"}]
result2 = await Runner.run(agent, new_input)
print(result2.final_output)  # Should reference "Alice"
```

The `to_input_list()` method normalizes all items from the run into a standard input-item format that can be fed back into another `Runner.run()` call. This is the manual approach to conversation management. Later in this book, you will learn about sessions, which automate this process.

`last_agent` – The agent that produced the final output. In multi-agent runs with handoffs, this may differ from the agent you originally passed to `Runner.run()`. Use `last_agent` when continuing a conversation: it is typically the best agent to handle the next user turn.

```python
result = await Runner.run(triage_agent, "I need help with my billing.")
# The triage agent might have handed off to a billing specialist
print(result.last_agent.name)  # "Billing Specialist"
```

`raw_responses` – The raw model responses from every LLM call during the run. Multi-step runs produce multiple responses (one per loop iteration), and this list contains them all. Useful for debugging, auditing, and provider-level diagnostics.

`last_response_id` – The ID of the last model response, used for server-managed conversation continuation via `previous_response_id`. This is relevant when using OpenAI's server-side conversation state feature.

```
result = await Runner.run(agent, "Hello")
# Continue the conversation on the server side
result2 = await Runner.run(
    agent,
    "Tell me more",
    previous_response_id=result.last_response_id,
)
```

`interruptions` – A list of pending approval requests when the run paused for human-in-the-loop decisions. Empty if no approvals were needed. You will explore this in depth in Chapter 9.

```
result = await Runner.run(agent, "Delete all temporary files.")
if result.interruptions:
    print("Run paused for approval")
    state = result.to_state()
    # Approve or reject the pending tool calls
    for interruption in result.interruptions:
        state.approve(interruption)
    result = await Runner.run(agent, state)
```

These result surfaces form the foundation for everything you will build with the SDK. You will use `final_output` to display answers, `new_items` to build rich UIs that show tool calls and reasoning steps, `to_input_list()` to manage conversation history, and `interruptions` to implement approval workflows. Understanding each property now will save significant debugging time later.

Chapter 2: Agent Configuration and Prompt Engineering

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingaiagentswiththeopenaiagentssdk.

Defining an Agent

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingaiagentswiththeopenaiagentssdk.

Dynamic Instructions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingaiagentswiththeopenaiagentssdk.

Model Selection and Configuration

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingaiagentswiththeopenaiagentssdk.

Structured Outputs with Pydantic

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingaiagentswiththeopenaiagentssdk.

Prompt Templates and Platform Prompts

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingaiagentswiththeopenaiagentssdk.

Chapter 3: Building Tools for Your Agents

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingaiagentswiththeopenaiagentssdk.

Function Tools with @function_tool

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingaiagentswiththeopenaiagentssdk.

Advanced Function Tool Patterns

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingaiagentswiththeopenaiagentssdk.

Hosted OpenAI Tools

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingaiagentswiththeopenaiagentssdk.

Tool Namespaces and Deferred Loading

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingaiagentswiththeopenaiagentssdk.

Local Runtime Tools

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingaiagentswiththeopenaiagentssdk.

Chapter 4: Running Agents and Managing State

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingaiagentswiththeopenaiagentssdk.

The Agent Loop Explained

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingaiagentswiththeopenaiagentssdk.

Manual Conversation Management

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingaiagentswiththeopenaiagentssdk.

Session Memory

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingaiagentswiththeopenaiagentssdk.

Server-Managed Conversations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingaiagentswiththeopenaiagentssdk.

Streaming Events and Cancellation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingaiagentswiththeopenaiagentssdk.

Error Handling and Recovery

Chapter 5: Context Management and Dependency Injection

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingaiagentswiththeopenaiagentssdk.

The Context Pattern

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingaiagentswiththeopenaiagentssdk.

Passing Context Through Tools

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingaiagentswiththeopenaiagentssdk.

Dynamic Instructions from Context

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingaiagentswiththeopenaiagentssdk.

Lifecycle Hooks

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingaiagentswiththeopenaiagentssdk.

Usage Tracking and Token Accounting

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingaiagentswiththeopenaiagentssdk.

Cloning and Copying Agents

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingaiagentswiththeopenaiagentssdk.

Tool Input for Nested Agent Runs

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingaiagentswiththeopenaiagentssdk.

Chapter 6: Multi-Agent Systems with Handoffs

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingaiagentswiththeopenaiagentssdk.

The Handoff Pattern Explained

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingaiagentswiththeopenaiagentssdk.

Configuring Handoffs

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingaiagentswiththeopenaiagentssdk.

Structured Handoff Data

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingaiagentswiththeopenaiagentssdk.

Input Filters and History Management

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingaiagentswiththeopenaiagentssdk.

Manager Pattern (Agents as Tools)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingaiagentswiththeopenaiagentssdk.

Handoff Prompt Engineering

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingaiagentswiththeopenaiagentssdk.

Multi-Level Delegation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingaiagentswiththeopenaiagentssdk.

Chapter 7: Guardrails and Safety Controls

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingaiagentswiththeopenaiagentssdk.

Input Guardrails

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingaiagentswiththeopenaiagentssdk.

Output Guardrails

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingaiagentswiththeopenaiagentssdk.

Tool Guardrails

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingaiagentswiththeopenaiagentssdk.

Building Practical Guardrails

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingaiagentswiththeopenaiagentssdk.

Guardrails in Multi-Agent Workflows

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingaiagentswiththeopenaiagentssdk.

Chapter 8: Observability and Tracing

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingaiagentswiththeopenaiagentssdk.

Automatic Tracing

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingaiagentswiththeopenaiagentssdk.

The Trace Dashboard

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingaiagentswiththeopenaiagentssdk.

Custom Spans and Manual Traces

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingaiagentswiththeopenaiagentssdk.

Sensitive Data Controls

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingaiagentswiththeopenaiagentssdk.

Third-Party Integrations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingaiagentswiththeopenaiagentssdk.

Flushing Traces

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingaiagentswiththeopenaiagentssdk.

Non-OpenAI Tracing

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingaiagentswiththeopenaiagentssdk.

Chapter 9: Human-in-the-Loop and Approval Workflows

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingaiagentswiththeopenaiagentssdk.

The Approval Flow

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingaiagentswiththeopenaiagentssdk.

Approving and Rejecting Tool Calls

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingaiagentswiththeopenaiagentssdk.

Streaming with Approvals

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingaiagentswiththeopenaiagentssdk.

Durable State Serialization

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingaiagentswiththeopenaiagentssdk.

Building Interactive Approval Interfaces

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingaiagentswiththeopenaiagentssdk.

Chapter 10: Advanced Multi-Agent Orchestration

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingaiagentswiththeopenaiagentssdk.

Agents as Tools Deep Dive

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingaiagentswiththeopenaiagentssdk.

Structured Input for Tool Agents

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingaiagentswiththeopenaiagentssdk.

Custom Output Extraction

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingaiagentswiththeopenaiagentssdk.

Streaming Nested Agent Runs

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingaiagentswiththeopenaiagentssdk.

Conditional Tool Enabling

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingaiagentswiththeopenaiagentssdk.

Complex Orchestration Topologies

Chapter 11: Model Context Protocol (MCP) Integration

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingaiagentswiththeopenaiagentssdk.

What Is MCP?

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingaiagentswiththeopenaiagentssdk.

Local MCP Servers

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingaiagentswiththeopenaiagentssdk.

Hosted MCP Tools

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingaiagentswiththeopenaiagentssdk.

MCP and Tool Search

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingaiagentswiththeopenaiagentssdk.

Practical MCP Integration Patterns

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingaiagentswiththeopenaiagentssdk.

Agent-Level MCP Configuration

Chapter 12: Production Deployment and Best Practices

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingaiagentswiththeopenaiagentssdk.

Performance Optimization

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingaiagentswiththeopenaiagentssdk.

Cost Management

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingaiagentswiththeopenaiagentssdk.

Security Best Practices

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingaiagentswiththeopenaiagentssdk.

Testing Strategies

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingaiagentswiththeopenaiagentssdk.

Deployment Architectures

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingaiagentswiththeopenaiagentssdk.

Real-World Case Study: Customer Service Bot

Conclusion: The Future of Agentic AI

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingaiagentswiththeopenaiagentssdk.

Key Takeaways

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingaiagentswiththeopenaiagentssdk.

Emerging Capabilities

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingaiagentswiththeopenaiagentssdk.

Your Next Steps

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingaiagentswiththeopenaiagentssdk.

References

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingaiagentswiththeopenaiagentssdk.