Agentic AI
Design Patterns

A Hands-On Workshop with Pure Python & OpenAI SDK

No Frameworks | Pure Python | OpenAI SDK | Pydantic | Multi-Provider

What You'll Learn

01

LLM Foundations

Chat Completions API, system prompts, structured outputs

02

Tools & Function Calling

Give LLMs the ability to take actions in the real world

03

The Agent Loop

Reason → Act → Observe — the core of every agent

04

5 Orchestration Patterns

Sequential, Concurrent, Group Chat, Handoff, Magentic

05

Production Considerations

Context management, reliability, human-in-the-loop

Workshop Structure

Day 1

Foundations

- LLM Basics & Chat API
- System Prompts
- Structured Outputs
- Tools & Function Calling
- Single Agent Pattern

`00_setup → 03_single_agent`

Day 2

Multi-Agent

- Sequential Pipeline
- Concurrent Fan-out/Fan-in
- Group Chat & Maker-Checker

`04_sequential → 06_group_chat`

Day 3

Advanced

- Handoff & Routing
- Magentic Planning
- Reliability & HITL

`07_handoff → 08_magentic`

Philosophy: Build from Scratch

By building each pattern yourself, you understand exactly what frameworks do under the hood.

OpenAI SDK

client.chat.completions
.create()
The only LLM call you need

Pydantic

Structured outputs &
tool parameter schemas

Python stdlib

logging, dataclasses,
concurrent.futures

No LangChain · No LangGraph · No AutoGen · No CrewAI · No Semantic Kernel

Works with OpenAI · Azure OpenAI · GitHub Models

The Commons Module: exercises/commons/

Three files that power every exercise — understand these first.

llm_client.py

`get_client()`
Returns OpenAI or AzureOpenAI

`get_model()`
Provider-specific default model

`get_provider()`
openai | azure | github

agent.py

`Agent dataclass`
name, system_prompt, tools, tool_functions, max_iterations

`run(agent, messages, client)`
Reason → Act → Observe loop

utils.py

`setup_logging()`
Colored console output

`log_tool_call()`

`log_handoff() / log_stage()`
Narrates agent behavior

Logging IS the teaching tool — every tool call, handoff, and context pass is logged so you can see the agent's reasoning.

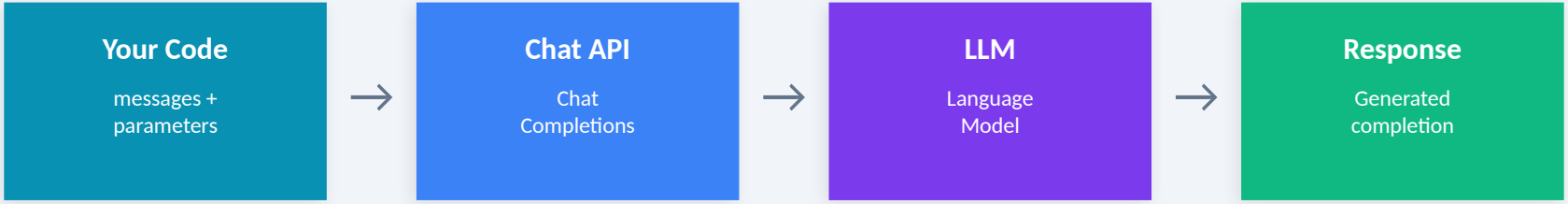
01

LLM Foundations

Chat Completions API · System Prompts · Structured Outputs

The Chat Completions API

A stateless, request-response interface. You send messages, the model returns a completion.



Message Roles

`system`	`user`	`assistant`	`tool`
Sets behavior, persona, constraints	Human input — questions, data	Model's previous responses	Results from function calls

Exercise: Chat Completion & Multi-Turn

`exercises/01_llm_basics/01_chat_completion.py`

Single-Turn Request

```
messages = [
  {"role": "system", "content": SYSTEM_PROMPT},
  {"role": "user",   "content": question},
]
response = client.chat.completions.create(
  model=model, messages=messages,
  temperature=0.7, max_tokens=300
)
```

Multi-Turn: Append & Resend

```
# Append assistant reply to history
messages.append(
  {"role": "assistant", "content": reply}
)
# Add follow-up question
messages.append(
  {"role": "user", "content": follow_up}
)
```

Key Insights

The API is stateless

You must send the full message history every time — the model has no memory between calls.

temperature controls randomness

0 = deterministic, 0.7 = creative, 2.0 = very random. Usage stats track token consumption.

Exercise: System Prompts Shape Identity

`exercises/01_llm_basics/02_system_prompts.py`

Persona 1: Formal Travel Advisor

```
FORMAL_PROMPT = (
  "You are a professional travel "
  "consultant with 20 years of "
  "experience. You speak formally, "
  "use precise language..."
```

Result: Structured, formal recommendations with budget/visa details

Persona 2: Casual Travel Buddy

```
CASUAL_PROMPT = (
  "You are a fun-loving travel "
  "buddy who has backpacked across "
  "50+ countries. You speak "
  "casually, use informal language,"
```

Result: Informal tips, hidden gems, personal stories

The Pattern: Same query + same model + different system prompt = different agent behavior

```
for persona_name, system_prompt in PERSONAS:
    messages = [{"role": "system", "content": system_prompt}, {"role": "user", "content": USER_QUERY}]
```

Structured Outputs

Guarantee the model returns data in a specific format — not just free text.

When to Use

- ✓ Classification & routing
- ✓ Entity extraction from text
- ✓ Decision making with reasoning
- ✓ Handoff context between agents
- ✓ Data transformation pipelines

parse() + Pydantic

```
class ReviewAnalysis(BaseModel):
    sentiment: str
    confidence: float
    keywords: list[str]

response = client.chat.completions
    .parse(response_format=ReviewAnalysis)

analysis = response.choices[0]
    .message.parsed  # Pydantic obj!
```

Use the stable API — client.chat.completions.parse() — never client.beta.*

Exercise: Structured Outputs with parse()

`exercises/01_llm_basics/03_structured_outputs.py`

1. Define the Schema

```
class ReviewAnalysis(BaseModel):
    sentiment: str    # positive/negative/mixed
    rating: int       # 1-5 stars
    keywords: list[str]
    summary: str      # One-sentence
    recommended: bool
```

2. Call parse() with the Model

```
response = client.chat.completions.parse(
    model=model,
    messages=[
      {"role": "system", "content": PROMPT},
      {"role": "user",   "content": review},
    ],
    response_format=ReviewAnalysis,
)
```

3. Use the Typed Result

```
# analysis is a ReviewAnalysis instance — not a dict, not a string
logger.info("Sentiment:   %s", analysis.sentiment)      # "positive"
logger.info("Rating:      %s/5", analysis.rating)        # 4
logger.info("Keywords:    %s", analysis.keywords)        # ["laptop", "backpack", "hip belt"]
logger.info("Recommended: %s", analysis.recommended)     # True
```

No JSON parsing, no schema validation, no try/except — Pydantic handles it all.

Chapter 1 Recap: LLM Foundations

1. Chat Completions API is stateless — you manage conversation history as a messages list
2. System prompts define agent identity — same model, different prompt = different behavior
3. Structured outputs with parse() give you typed Pydantic objects, not raw text
4. The commons module (llm_client, agent, utils) is reused in every exercise
5. Multi-turn = append assistant reply + new user message, then resend the full list

Next: Chapter 2 — Tools & Function Calling → Give your LLM the ability to take actions