Day 4: Agents and Tools

Welcome to day four of our AI Agents Crash Course.

In the first part of the course, we focused on data preparation. Now the data is prepared and indexed so that we can use it for AI agents

So far, we have done:

- Day 1: Downloaded the data from a GitHub repository
- Day 2: Processed it by chunking it where necessary
- Day 3: Indexed the data so it's searchable

Note that it took us quite a lot of time. We're halfway through the course, and only now we started working on agents. Most of the time so far, we have spent on data preparation.

This is not a coincidence. Data preparation is the most time-consuming and critical part of building AI agents. Without properly prepared, cleaned, and indexed data, even the most sophisticated agent will provide poor results.

Now it's time to create an AI agent that will use this data through the search engine that we created yesterday.

This allows us to build context-aware agents. They can provide accurate, relevant answers based on your specific domain knowledge rather than just general training data.

In particular, we will:

- Learn what makes an AI system "agentic" through tool use
- Build an agent that can use the search function
- Use Pydantic AI to make it easier to implement agents

At the end of this lesson, you'll have a working AI Agent that you can answer your questions in a Jupyter notebook.

1. Tools and Agents

You can find many agent definitions online.

But we will use a simple one: an *agent* is an LLM that can not only generate texts, but also invoke tools. Tools are external functions that the LLM can call in order to retrieve information, perform calculations, or take actions.

In our case, the agent needs to answer our questions using the content of the GitHub repository. So, the tool (only one) is a `search(query)`.

But first, let's consider a situation where we have no tools at all. This is not an agent, it's just an LLM that can generate texts. Access to tools is what makes agents "agentic".

Let's see the difference with an example.

We will try asking a question without giving the LLM access to search:

```python
import openai

openai_client = openai.OpenAI()

user_prompt = "I just discovered the course, can I join now?"

chat_messages = [
    {"role": "user", "content": user_prompt}
]

response = openai_client.responses.create(
    model='gpt-4o-mini',
    input=chat_messages,
)

print(response.output_text)
```

The response is generic. In our case, it's this:

> “It depends on the course you're interested in. Many courses allow late enrollment, while others might have specific deadlines. I recommend checking the course's official website or contacting the instructor or administration for more details on joining.”

This answer is not really useful.

But if we let it invoke the `search(query)`, the agent can give us a more useful answer.

Here's how the conversation would flow with our agent using the `search` tool:

- **User**: "I just discovered the course, can I join now?"
- **Agent thinking**: I can't answer this question, so I need to search for information about course enrollment and timing.
- **Tool call**: `search("course enrollment join registration deadline")`
- **Tool response**: (...search results...)
- **Agent response**: "Yes, you can still join the course even after the start date..."

We will now explore how to implement it with OpenAI.

2. Function Calling with OpenAI

Let's create an agent now. In OpenAI's terminology, we'll need to use "function calling".

We will begin with our FAQ example and text search. You can easily extend it to vector or hybrid search or change it to the Evidently docs.

This is the function we implemented yesterday:

```python
def text_search(query):
    return faq_index.search(query, num_results=5)
```

We can't just pass this function to OpenAI. First, we need to describe this function, so the LLM understands how to use it.

This is done using a special description format:

```python
text_search_tool = {
    "type": "function",
    "name": "text_search",
    "description": "Search the FAQ database",
    "parameters": {
        "type": "object",
        "properties": {
            "query": {
                "type": "string",
                "description": "Search query text to look up in the course 
FAQ."
            }
        },
        "required": ["query"],
        "additionalProperties": False
    }
}
```

This description tells OpenAI:

- The function is called `text_search`
- It searches the FAQ database
- It takes one required parameter: `query` (a string)
- The query should be the search text to look up in the course FAQ

Now we can use it:

```Python
system_prompt = """
You are a helpful assistant for a course.
"""

question = "I just discovered the course, can I join now?"

chat_messages = [
    {"role": "system", "content": system_prompt},
    {"role": "user", "content": question}
]

response = openai_client.responses.create(
    model='gpt-4o-mini',
    input=chat_messages,
    tools=[text_search_tool]
)
```

Previously, we had a simple text response; now, the response includes function calls.

Let's look at `response.output`. In my case, it contains the following:

```None
[ResponseFunctionToolCall(arguments='{"query":"join course"}',
call_id='call_O0Wbg2qnqCJqAVPjpr8JSFjg', name='text_search',
type='function_call', id='fc_68d53ecc2d9881948419c14db8ef11d9099f344ef74b51a3',
status='completed')]
```

The agent analyzed the user's question and determined that to answer it, it needs to invoke the `text_search` function with the arguments `{"query":"join course"}`.

Let's invoke the function with these arguments:

```Python
import json

call = response.output[0]

arguments = json.loads(call.arguments)
result = text_search(**arguments)

call_output = {
    "type": "function_call_output",
    "call_id": call.call_id,
    "output": json.dumps(result),
}
```

Here's what's happening:

1. The LLM decided to execute a function and let us know about it
2. We executed the function and saved the results
3. Now we need to pass this information back to the LLM

We do it by extending the `chat_messages` list and sending the entire conversation history back to the LLM:

```Python
chat_messages.append(call)
chat_messages.append(call_output)

response = openai_client.responses.create(
    model='gpt-4o-mini',
    input=chat_messages,
    tools=[text_search_tool]
)
```

```
print(response.output_text)
```

LLMs are *stateless*. When we make one call to the OpenAI API and then shortly afterwards make another, it doesn't know anything about the first call. So if we only send it `call_output`, it would have no idea how to respond to it.

This is why we need to send it the entire conversation history. It needs to know everything that happened so far:

- The system prompt (so it knows what the initial instructions are) - `system_prompt`
- The user prompt (so it knows what task it needs to perform) - `question`
- The decision to invoke the `text_search` tool (so it knows what function was called) - that's our `call`
- The output of the function (so it knows what the function returned) - that's our `call_output`

After we invoke it, we get back the response:

> “Yes, you can still join the course even after the start date. While you won't be able to officially register, you are eligible to submit your homework. Just keep in mind that there are deadlines for submitting assignments and final projects, so it's best not to leave everything to the last minute.”

This is a useful response that we were hoping to get.

3. System Prompt: Instructions

Let's take another look at the code we wrote previously:

```python
chat_messages = [
    {"role": "system", "content": system_prompt},
    {"role": "user", "content": question}
]

response = openai_client.responses.create(
    model='gpt-4o-mini',
    input=chat_messages,
    tools=[text_search_tool]
)
```

We have two things here:

- `system_prompt` contains instructions for the LLM
- `question` ("user prompt") is the actual question or task

The system prompt is very important: it influences how the agent behaves. This is how we can control what the agent does and how it responds to user questions.

Usually, the more complete the instructions in the system prompt are, the better the results.

So we can extend it:

```python
system_prompt = """
You are a helpful assistant for a course.

Use the search tool to find relevant information from the course materials
before answering questions.

If you can find specific information through search, use it to provide accurate
answers.
If the search doesn't return relevant results, let the user know and provide
general guidance.
"""
```

When working with agents, the system prompt becomes one of the most essential variables we can adjust to influence our agent.

For example, if we want the agent to make multiple search queries, we can modify the prompt:

```python
system_prompt = """
You are a helpful assistant for a course.
```

```
Always search for relevant information before answering.
If the first search doesn't give you enough information, try different search
terms.

Make multiple searches if needed to provide comprehensive answers.
"""
```

4. Pydantic AI

Dealing with function calls can be cumbersome. We first need to understand which function we need to invoke. Then we need to pass the results back to the LLM and perform other tasks. It's easy to make a mistake there.

That's why we'll use a library to handle it. There are many agentic libraries: OpenAI Agents SDK, Langchain, Pydantic AI, and many more. For educational purposes, I also implemented an agents library. It's called ToyAIKit. We won't use it here, but I often use it in my lessons.

Today, we will use Pydantic AI. I like its API; it's simpler than other libraries and has good documentation.

Let's install it:

```shell
uv add pydantic-ai
```

For Pydantic AI (and for other agents libraries), we don't need to describe the function in the JSON format like we did with the plain OpenAI API. The libraries take care of it.

But we do need to add docstrings and type hints to our function. I asked ChatGPT to do it:

```python
from typing import List, Any

def text_search(query: str) -> List[Any]:
    """
    Perform a text-based search on the FAQ index.
    Args:
        query (str): The search query string.

    Returns:
        List[Any]: A list of up to 5 search results returned by the FAQ index.
    """
    return faq_index.search(query, num_results=5)
```

We can now define an agent with Pydantic AI and give it the `text_search` tool:

```python
from pydantic_ai import Agent

agent = Agent(
    name="faq_agent",
    instructions=system_prompt,
    tools=[text_search],
    model='gpt-4o-mini'
)
```

We don't need to do anything with our `text_search` function. We just pass it directly to the agent.

Let's run it:

```python
question = "I just discovered the course, can I join now?"

result = await agent.run(user_prompt=question)
```

We use `await` because Pydantic AI is asynchronous. If you're not running in Jupyter, you need to use `asyncio.run()`:

```python
import asyncio

result = asyncio.run(agent.run(user_prompt=question))
```

The output:

> “Yes, you can still join the course even after the start date. Although you may not officially register, you are eligible to submit your homework. Just keep in mind that there are deadlines for turning in homework and final projects, so it's advisable not to delay everything until the last minute.”

We can also look inside the result to get a detailed breakdown of the agent's reasoning and actions:

```python
result.new_messages()
```

It contains four items:

- ModelRequest: Represents a request sent to the model. It includes the user's prompt (`UserPromptPart`) and the agent's instructions.
- ModelResponse: The model's reply. We see a `ToolCallPart` with the decision to invoke `text_search`.
- ModelRequest: Contains `ToolReturnPart` - the results returned by the tool (search results from the FAQ index).
- ModelResponse: The final answer generated by the model in `TextPart`.

Pydantic AI and other frameworks handle all the complexity of function calling for us. We don't need to manually parse responses, handle tool calls, or manage conversation history. This makes our code cleaner and less error-prone.

We implemented an agent. Great! But how good is it? Is the prompt we came up good? What's better for our agent, text search, vector search or hybrid? Tomorrow we will be able to answer these questions: we will learn how to use AI to evaluate our agent.

Homework

- Do the same, but for the documentation we extracted instead of FAQ
- For the project you selected, create an agent with Pydantic AI
- Interact with the agent, improve the system prompt if needed
- Make a post on social media about what you're building

Example post for LinkedIn

🤖 Day 4 of building AI agent: adding function calling!

Today I learned how to make my AI system truly "agentic" by adding tools:

• Provided the agent with the search function
• Implemented function calling with OpenAI's API
• Used Pydantic AI to simplify agent development

My project focuses on [YOUR PROJECT DESCRIPTION]. The agent can now search through specific domain knowledge and provide accurate answers!

Here's my repo: [YOUR_REPO_LINK]

Next up: Evaluating the agent!

Following along with this amazing course - who else is building AI agents?

You can sign up here: https://alexeygrigorev.com/aihero/

Example post for Twitter/X

Day 4: My AI agent can now use tools! 🛠️

✅ Function calling implemented
✅ Search integrated
✅ Pydantic AI for cleaner code

The difference between a chatbot and an agent? Tools!

Here's my repo: [YOUR_REPO_LINK]

Tomorrow: evals

Join me: https://alexeygrigorev.com/aihero/

Community

Do you have questions about this lesson or suggestions for improvement? You can find me and other learners in **DataTalks.Club Slack**:

- Join DataTalks.Club
- Find us in the `#course-ai-bootcamp` channel

In the community channel, you can:

- Ask questions about the course content
- Share your implementation and get feedback
- Show off your GitHub repositories
- Suggest improvements to the course materials
- Connect with other course participants

Don't hesitate to reach out - the community is here to help each other succeed!