

# AI for App Development

---

## Building AI Apps that leverage agents, MCP, and RAG

---

### Session labs

---

### Revision 2.6 - 10/09/25

---

### (c) 2025 Tech Skills Transformations

---

**Follow the startup instructions in the README.md file IF NOT ALREADY DONE!**

**NOTES** - To copy and paste in the codespace, you may need to use keyboard commands - CTRL-C and CTRL-V. Chrome may work best for this. - If your codespace has to be restarted, run these commands again! `ollama serve & python warmup_models.py`

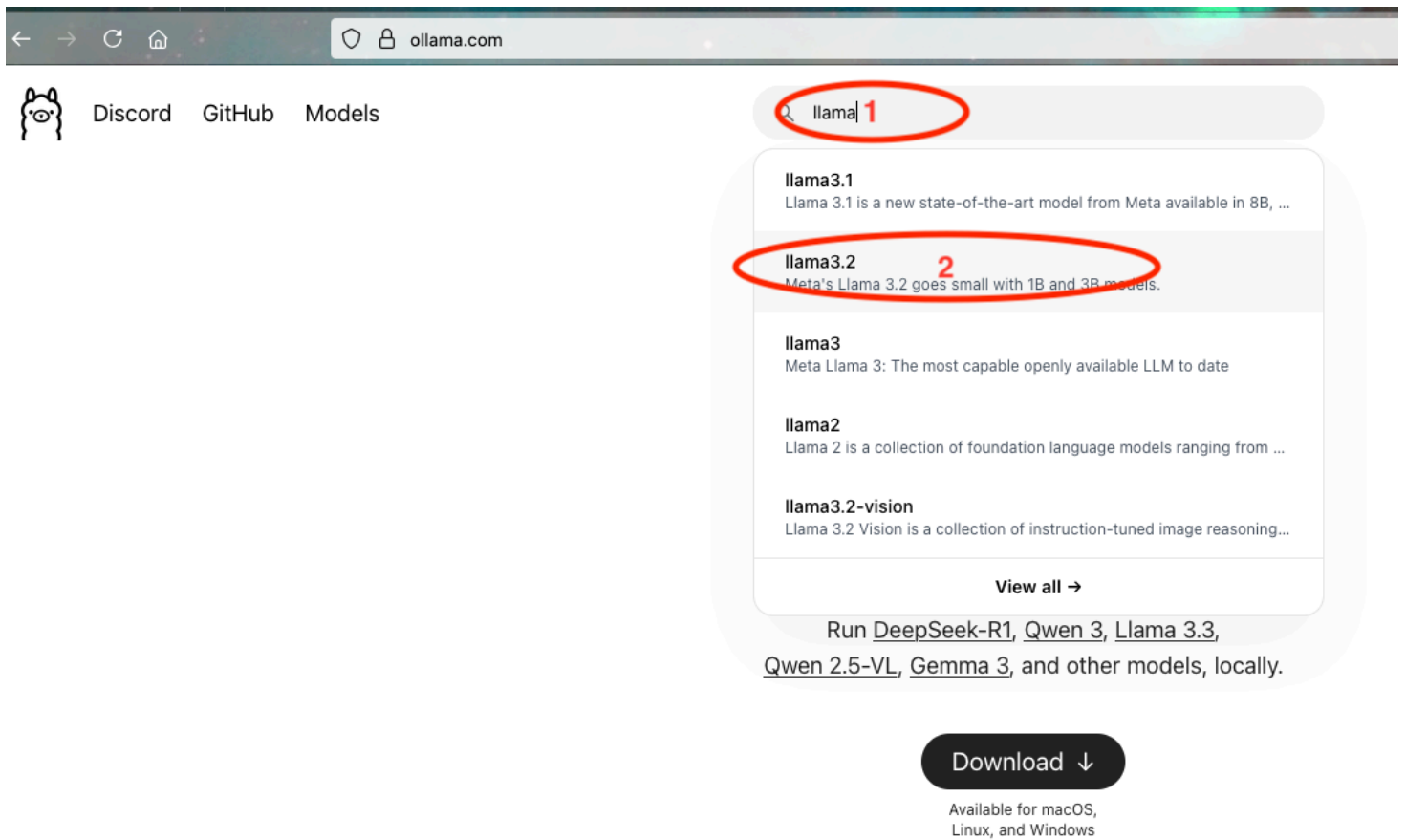
#### Lab 1 - Using Ollama to run models locally

**Purpose:** In this lab, we'll start getting familiar with Ollama, a way to run models locally.

1. The Ollama app is already installed as part of the codespace setup via [scripts/startOllama.sh](#). Start it running with the first command below. (If you need to restart it at some point, you can use the same command. To see the different options Ollama makes available for working with models, you can run the second command below in the *TERMINAL*.

```
ollama serve &  
<Hit Enter>  
ollama
```

1. Now let's find a model to use. Go to <https://ollama.com> and in the *Search models* box at the top, enter *llama*. In the list that pops up, choose the entry for "llama3.2".



1. This will put you on the specific page about that model. Scroll down and scan the various information available about this model.



## llama3.2

```
ollama run llama3.2
```



23.4M Downloads Updated 9 months ago

Meta's Llama 3.2 goes small with 1B and 3B models.

tools

1b

3b

## Models

[View all -](#)

Name	Size	Context	Input
llama3.2:latest	2.0GB	128K	Text
llama3.2:1b	1.3GB	128K	Text
llama3.2:3b <span>latest</span>	2.0GB	128K	Text

## Readme



The Meta Llama 3.2 collection of multilingual large language models (LLMs) is a collection of pretrained and instruction-tuned generative models in 1B and 3B sizes (text in/text out). The Llama 3.2 instruction-tuned text only models are optimized for multilingual dialogue use cases, including agentic retrieval and summarization tasks. They outperform many of the available open source and closed chat models on common industry benchmarks.

1. Switch back to a terminal in your codespace. Run the first command to see what models are loaded (none currently). Then pull the model down with the second command. (This will take a few minutes.)

```
ollama list
ollama pull llama3.2
```

```
(py_env) @techupskills + /workspaces/ai-3in1 (main) $ ollama pull llama3.2
[GIN] 2025/07/05 - 20:21:00 | 200 | 51.717µs | 127.0.0.1 | HEAD | "/"
pulling manifest .. time=2025-07-05T20:21:01.350Z level=INFO source=download.go:177 msg="downloading dde5aa3fc5ff in 16 126 MB part(s)"
pulling manifest
pulling dde5aa3fc5ff: 100% ██████████ 2.0 GB
pulling manifest
pulling dde5aa3fc5ff: 100% ██████████ 2.0 GB
pulling manifest
pulling dde5aa3fc5ff: 100% ██████████ 2.0 GB
pulling manifest
pulling dde5aa3fc5ff: 100% ██████████ 2.0 GB
pulling manifest
```

1. Once the model is downloaded, you can see it with the first command below. Then run the model with the second command below. This will load it and make it available to query/prompt.

```
ollama list
ollama run llama3.2
```

1. Now you can query the model by inputting text at the >>>*Send a message (/? for help)* prompt. Let's ask it about what the weather is in Paris. What you'll see is it telling you that it doesn't have access to current weather data and suggesting some ways to gather it yourself.

What's the current weather in Paris?

```
[GIN] 2025/07/05 - 20:24:46 | 200 | 2.407674148s | 127.0.0.1 | POST "/api/generate"
>>> What's the current weather in Paris?
I'm a large language model, I don't have real-time access to current weather conditions. However, I can suggest some ways for you to find out the current weather in Paris:

1. Check online weather websites: You can check websites like AccuWeather, Weather.com, or the French Meteo website (Météo-France) for the latest weather forecast and conditions in Paris.
2. Use a mobile app: There are several mobile apps available that provide real-time weather information, such as Dark Sky or Weather Underground.
3. Check social media: Many meteorologists and weather services share updates on their social media accounts, so you can try searching for "Paris weather" or "Météo-France" to get the latest information.

Please note that I'm a text-based model trained until December 2023, so I won't have real-time access to current weather conditions.[GIN]
5/07/05 - 20:25:50 | 200 | 17.283091173s | 127.0.0.1 | POST "/api/chat"

>>> Send a message (/? for help)
```

1. Now, let's try a call with the API. You can stop the current run with a Ctrl-D or switch to another terminal. Then put in the command below (or whatever simple prompt you want).

```
curl http://localhost:11434/api/generate -d '{
  "model": "llama3.2",
  "prompt": "What causes weather changes?",
  "stream": false
}' | jq -r '.response'
```

1. This will take a minute or so to run. You should see a long text response . You can try out some other prompts/queries if you want.

```

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS 3
(py_env) @brentlaster2 → /workspaces/ai-apps (main) $ curl http://localhost:11434/api/generate -d '{
  "model": "llama3.2",
  "prompt": "What causes weather changes?",
  "stream": false
}' | jq -r '.response'
5. ocean currents: ocean currents can also influence local weather patterns by transferring heat, moisture, and other atmospheric factors.
6. Topography: Mountain ranges, hills, and valleys can force air to rise, cool, or sink, leading to changes in the weather.
7. Global circulation patterns: Large-scale circulation patterns, such as El Niño and La Niña events, affect global climate and weather.

Some of the specific mechanisms that drive weather changes include:

1. Evaporation and condensation: When water evaporates from oceans, lakes, or rivers, it cools the air, leading to cloud formation.
2. Heat transfer: When warm air rises and cool air sinks, it can lead to changes in atmospheric circulation patterns and temperature.
3. Atmospheric instability: Changes in atmospheric stability, such as when a layer of cool air is trapped under a layer of warm air, can lead to the formation of thunderstorms or other severe weather events.

These factors interact with each other in complex ways, resulting in a wide range of weather phenomena and changes over time.
(py_env) @brentlaster2 → /workspaces/ai-apps (main) $

```

1. Now let's try a simple Python script that uses Ollama programmatically. We have a basic example script called `simple_ollama.py`. Take a look at it either via [simple\\_ollama.py](#) or via the command below.

```
code simple_ollama.py
```

You should see a simple script that:

- Imports the ChatOllama class from langchain\_ollama
- Initializes the Ollama client with the llama3.2 model
- Takes user input
- Sends it to Ollama
- Displays the response

1. Now you can run the script with the command below.

```
python simple_ollama.py
```

1. When prompted, enter a question like "What is the capital of France?" and press Enter. You should see the model's response printed to the terminal. This demonstrates how easy it is to integrate Ollama into a Python application. Feel free to try other prompts.
2. In preparation for the remaining labs, let's get the model access approaches "warmed up". Start the command below and just leave it running while we continue (if it doesn't finish quickly).

```
python warmup_models.py
```

**\*\*[END OF LAB]\*\***

## Lab 2 - Creating a simple agent

**Purpose:** In this lab, we'll learn about the basics of agents and see how tools are called. We'll also see how Chain of Thought prompting works with LLMs and how we can have ReAct agents reason and act.

1. For this lab, we have the outline of an agent in a file called `agent.py` in that directory. You can take a look at the code either by clicking on [agent.py](#) or by entering the command below in the codespace's terminal.

code agent.py

```

56
57 #
58 # 2. "Tool" functions (simple Python, no server needed)
59 #
60 def get_weather(lat: float, lon: float) -> dict:
61
62
63 def convert_c_to_f(c: float) -> float:
64
65 #
66 # 3. Local LLM wrapper (LangChain + Ollama)
67 #
68
69 #
70 # 4. "System" prompt that defines the tools and the TAO protocol
71 #
72
73 #
74 # 5. Helper that runs a single TAO episode and prints the trace
75 #
76 def run(question: str) -> str:

```

1. As you can see, this outlines the steps the agent will go through without all the code. When you are done looking at it, close the file by clicking on the "X" in the tab at the top of the file.

1. Now, let's fill in the code. To keep things simple and avoid formatting/typing frustration, we already have the code in another file that we can merge into this one. Run the command below in the terminal.

```
code -d labs/common/lab2_agent_solution.txt agent.py
```

1. Once you have run the command, you'll have a side-by-side in your editor of the completed code and the agent1.py file. You can merge each section of code into the agent1.py file by hovering over the middle bar and clicking on the arrows pointing right. Go through each section, look at the code, and then click to merge the changes in, one at a time.

```

agent.py > ...
60 def get_weather(lat: float, lon: float) -> dict:
61     url = (
62         f"?latitude={lat}&longitude={lon}"
63         "&daily=weathercode,temperature_2m_max,temperature_2m_min"
64         "&forecast_days=1&timezone=auto"
65     )
66     r = requests.get(url, timeout=15)
67     r.raise_for_status() # raise for any 4xx / 5xx
68     daily = r.json()["daily"]
69
70     return {
71         "high": daily["temperature_2m_max"][0],
72         "low": daily["temperature_2m_min"][0],
73         "conditions": WEATHER_CODES.get(daily["weathercode"][0], "Unknown")
74     }
75
76
77 def convert_c_to_f(c: float) -> float:
78     """Classic °C → °F conversion."""
79     return c * 9 / 5 + 32
80
81
82 # 3. Local LLM wrapper (LangChain + Ollama)
83 # The model is fetched from your local Ollama instance; set temperature
84 # to 0.0 for deterministic planning.
85 llm = ChatOllama(model="llama3.2", temperature=0.0)
86
87 # 4. "System" prompt that defines the tools and the TA0 protocol

```

1. When you have finished merging all the sections in, the files should show no differences. Save the changes simply by clicking on the "X" in the tab name.

```

agent.py > ...
1 #!/usr/bin/env python3
2 """
3
4 A **tiny demonstration** of the classic Thought-Action-Observation (TA0)
5 loop using:
6
7 * **Open-Meteo** ☐ free REST weather API
8 * **LangChain + Ollama** ☐ local Llama-3.2 language model
9 * **Two "tools" defined in pure Python

```

1. Now you can run your agent with the following command:

```
python agent.py
```

```

(py_env) @techupskills → /workspaces/ai-3in1 (main) $ python agent.py

Ask your question: What's the current weather in Paris?
time=2025-07-05T20:42:28.311Z level=INFO source=server.go:135 msg="system memory" total="15.6 GiB" free="12.4 GiB" free_sw
time=2025-07-05T20:42:28.311Z level=INFO source=server.go:175 msg="offload library=cpu layers.requested=-1 layers.model=29
yers.split="" memory.available="[12.4 GiB]" memory.gpu_overhead="0 B" memory.required.full="3.3 GiB" memory.required.parti
uired.kv="896.0 MiB" memory.required.allocations="[3.3 GiB]" memory.weights.total="1.9 GiB" memory.weights.repeating="1.6
.nonrepeating="308.2 MiB" memory.graph.full="424.0 MiB" memory.graph.partial="570.7 MiB"
llama_model_loader: loaded meta data with 30 key-value pairs and 255 tensors from /home/vscode/.ollama/models/blobs/sha256
b5e8bdc82f587b24b2678c6c66101bf7da77af9f7ccdff (version GGUF V3 (latest))
llama_model_loader: Dumping metadata keys/values. Note: KV overrides do not apply in this output.
llama_model_loader: - kv 0:          general.architecture str   = llama
llama_model_loader: - kv 1:          general.type str          = model
llama_model_loader: - kv 2:          general.name str           = Llama 3.2 3B Instruct

```

1. The agent will start running and will prompt for a location (or "exit" to finish). At the prompt, you can type in a location like "Paris, France" or "London" or "Raleigh" and hit *Enter*. You may see activity while the model is loaded. After that you'll be able to see the Thought -> Action -> Observation loop in practice as each one is listed out. You'll also see the arguments being passed to the tools as they are called. Finally you should see a human-friendly message summarizing the weather forecast.

```

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS 2
llama_context:      CPU compute buffer size = 424.01 MiB
llama_context: graph nodes = 958
llama_context: graph splits = 1
time=2025-07-06T16:31:03.638Z level=INFO source=server.go:637 msg="llama runner started in 2.01 seconds"
[GIN] 2025/07/06 - 16:31:13 | 200 | 12.046030574s | 127.0.0.1 | POST "/api/chat"
Thought: Get the current weather for Paris, France.
Action: get_weather
Args: {"lat":48.8566,"lon":2.3522}

Observation: {'high': 20.9, 'low': 14.8, 'conditions': 'Thunderstorm'}

[GIN] 2025/07/06 - 16:31:18 | 200 | 4.408840681s | 127.0.0.1 | POST "/api/chat"
Thought: Convert the high and low temperatures from Celsius to Fahrenheit.
Action: convert_c_to_f
Args: {"c": 20.9}

Observation: {'high_f': 69.62, 'low_f': 58.64}

Final: Today will be **Thunderstorm** with a high of **69.6 °F** and a low of **58.6 °F**.

Location (or 'exit'): 

```

1. You can then input another location and run the agent again or exit. Note that if you get a timeout error, the API may be limiting the number of accesses in a short period of time. You can usually just try again and it will work.

**\*\*[END OF LAB]\*\***

### Lab 3 - Exploring MCP

**Purpose:** In this lab, we'll see how MCP can be used to standardize an agent's interaction with tools.

1. We have partial implementations of an MCP server and an agent that uses an MCP client to connect to tools on the server. So that you can get acquainted with the main parts of each, we'll build them out as we did the agent in the second lab - by viewing differences and merging. Let's start with the server. Run the command below to see the differences.

```
code -d labs/common/lab3_server_solution.txt mcp_server.py
```



```

77 def get_weather(lat: float, lon: float) -> dict:
88     Parameters
89     -----
90     lat, lon : float
91     |     Geographic coordinates in decimal degrees.
92
93     Returns
94     -----
95     dict
96     {
97         "temperature": <float °C>,
98         "code":       <int WMO weathercode>,
99         "conditions": <friendly description>,
100        "error":       <error message if request failed>
101    }
102    """
103    url = (
104        "https://api.open-meteo.com/v1/forecast"
105        f"?latitude={lat}&longitude={lon}&current_weather=true"
106    )
107
108    last_error = None
109
110    # Retry loop with fresh connections
111    for attempt in range(MAX_RETRIES):
112        try:
113            # Fresh session per attempt avoids connection pool reuse i
114            session = requests.Session()
115            resp = session.get(url, timeout=15)

```

```

82
83     Parameters
84     -----
85     lat, lon : float
86     |     Geographic coordinates in decimal degrees.
87
88     Returns
89     -----
90+
91+
92
93     last_error = None
94
95     # Retry loop with fresh connections
96     for attempt in range(MAX_RETRIES):
97         try:
98             # Fresh session per attempt avoids connection pool
99             session = requests.Session()
100             resp = session.get(url, timeout=15)

```

1. As you look at the differences, note that we are using FastMCP to more easily set up a server, with its `@mcp.tool` decorators to designate our functions as MCP tools. Also, we run this using the `streamable-http` transport protocol. Review each difference to see what is being done, then use the arrows to merge. When finished, click the "x" in the tab at the top to close and save the files.

1. Now that we've built out the server code, run it using the command below. You should see some startup messages similar to the ones in the screenshot.

```
python mcp_server.py
```

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS 2
(py_env) @techupskills → /workspaces/ai-3in1 (main) $ python mcp_server.py

FastMCP 2.0

Server name: WeatherServer
Transport: Streamable-HTTP
Server URL: http://127.0.0.1:8000/mcp/

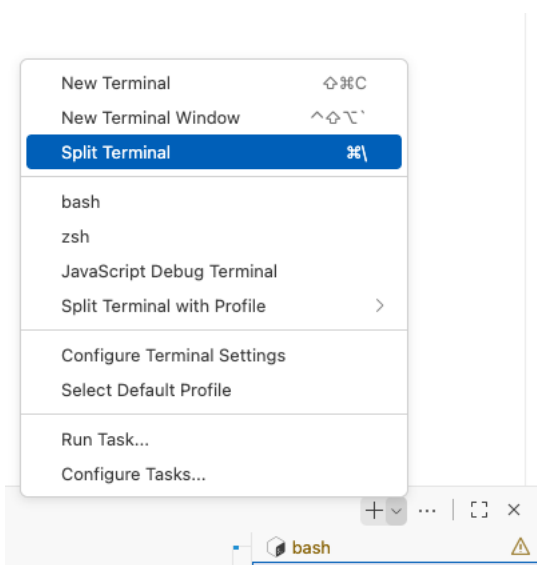
Docs: https://gofastmcp.com
Deploy: https://fastmcp.cloud

FastMCP version: 2.10.2
MCP version: 1.10.1

[07/06/25 16:54:10] INFO Starting MCP server 'WeatherServer' with transport 'http' on http://127.0.0.1:8000/mcp/
/workspaces/ai-3in1/py_env/lib/python3.11/site-packages/websockets/legacy/__init__.py:6: DeprecationWarning: websockets.legacy is deprecated. See https://websockets.readthedocs.io/en/stable/howto/upgrade.html for upgrade instructions
warnings.warn( # deprecated in 14.0 - 2024-11-09
/workspaces/ai-3in1/py_env/lib/python3.11/site-packages/uvicorn/protocols/websockets/websockets_impl.py:17: DeprecationWarning: websockets.legacy is deprecated
from websockets.server import WebSocketServerProtocol
INFO: Started server process [29199]
INFO: Waiting for application startup.
INFO: Application startup complete.
INFO: Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to quit)

```

1. Since this terminal is now tied up with the running server, we need to have a second terminal to use to work with the client. So that we can see the server responses, let's just open another terminal side-by-side with this one. To do that, over in the upper right section of the *TERMINAL* panel, find the plus sign and click on the downward arrow next to it. (See screenshot below.) Then select "Split Terminal" from the popup menu. Then click into that terminal to do the steps for the rest of the lab. (FYI: If you want to open another full terminal at some point, you can just click on the "+" itself and not the down arrow.)



1. We also have a small tool that can call the MCP *discover* method to find the list of tools from our server. This is just for demo purposes. You can take a look at the code either by clicking on [tools/discover\\_tools.py](#) or by entering the first

command below in the codespace's terminal. The actual code here is minimal. It connects to our server and invokes the `list_tools` method. Run it with the second command below and you should see the list of tools like in the screenshot.

```
code tools/discover_tools.py
python tools/discover_tools.py
```

```
(py_env) @brentlaster → /workspaces/ai-apps (main) $ python tools/discover_tools.py

Discovered 3 tool(s):

=====
Tool 1: get_weather
=====

Description
=====
Fetch current weather from Open-Meteo and return a concise dict.

Retry policy
=====
* Up to MAX_RETRIES total attempts with fresh connections.
* Retries on network errors or HTTP 429/5xx.
* Exponential back-off (1.5 s, 2.25 s, ...).
* Each retry uses a new session to avoid connection pool issues.
```

1. Now, let's turn our attention to the agent that will use the MCP server through an MCP client interface. First, in the second terminal, run a diff command so we can build out the new agent.

```
code -d labs/common/lab3_agent_solution_dynamic.txt mcp_agent.py
```

1. Review and merge the changes as before. What we're highlighting in this step are the *System Prompt* that drives the LLM used by the agent, the connection with the MCP client at the `/mcp/` endpoint, and the mpc calls to the tools on the server. When finished, close the tab to save the changes as before.

[Preview] README.md

mcp\_server.py M

lab3\_agent\_solution\_dynamic.txt ↔ mcp\_agent.py 1, M X

mcp\_agent.py > ...

```

4
5 A TRUE agentic implementation where the LLM dynamically selects which
6 tools to call and when to stop. This demonstrates:
7
8-**LLM-Driven Control Flow**: Agent loop runs until LLM says "DONE"
9-**Dynamic Tool Selection**: LLM chooses which MCP tool to invoke each step
10-**Flexible Reasoning**: Can handle queries requiring different tool sequences
11-**TAO Protocol**: Full thought/action/observation trace with real agent behavior
12-
13-Example Flows:
14-1. Standard: geocode → get_weather → convert_c_to_f → DONE
15-2. With coords: get_weather → convert_c_to_f → DONE (skip geocode)
16-3. Celsius OK: geocode → get_weather → DONE (skip conversion)
17
18 Prerequisites: FastMCP weather server must be running on localhost:8000
19 """
20
21 import asyncio
22 import json
23 import re
24 import textwrap
25 from typing import Optional, Dict, Any
26
27 from fastmcp import Client
28 from fastmcp.exceptions import ToolError
29 from langchain_ollama import ChatOllama
30
31-#

```

```

4
5 A TRUE agentic implementation where the L
6 tools to call and when to stop. This dem
7
8
9 Prerequisites: FastMCP weather server mu
10 """
11
12 import asyncio
13 import json
14 import re
15 import textwrap
16 from typing import Optional, Dict, Any
17
18 from fastmcp import Client
19 from fastmcp.exceptions import ToolError
20 from langchain_ollama import ChatOllama
21

```

1. After you've made and saved the changes, you can run the client in the terminal with the command below. **Note that there may be a long pause initially while the model is loaded and processed before you get the final answer. This could be on the order of minutes.**

```
python mcp_agent.py
```

1. The agent should start up, and wait for you to prompt it about weather in a location. You'll be able to see similar TAO output. And you'll also be able to see the server INFO messages in the other terminal as the MCP connections and events happen. A suggested prompt is below.

What is the weather in New York?

```

TERMINAL  PORTS 3
Ask about the weather: What is the weather in New York?
🔍 Detected city: New York

=====
Dynamic TAO Agent - LLM Controls Tool Selection
=====

[Step 1]
Thought: I have all the information needed
Action: geocode_location
Args: {"name": "New York"}

→ Calling MCP tool: geocode_location({"name": "New York"})
Observation: {"latitude": 40.71427, "longitude": -74.00597, "name": "New York"}

[Step 2]
Thought: Now that we have the coordinates for New York, I can get the current weather
Action: get_weather
Args: {"lat": 40.71427, "lon": -74.00597}

```

1. When you're done, you can use 'exit' to stop the client and CTRL-C to stop the server.

**\*\*[END OF LAB]\*\***

## Lab 4 - Working with Vector Databases

**Purpose:** In this lab, we'll learn about how to use vector databases for storing supporting data and doing similarity searches.

1. For this lab and the next one, we have a data file that we'll be using that contains a list of office information and details for a fictitious company. The file is in [data/offices.pdf](#). You can use the link to open it and take a look at it.

labs.md

index\_code.py

index\_pdf.py

offices.pdf

discover\_tools.py

requirements.txt

data > offices.pdf

/workspaces/ai-3in1/data/offices.pdf (preview)

1 of 2

Automatic Zoom

Office Name	Address	Number of Employees	Revenue (USD)	Services Offered
HQ	123 Main St, New York, NY	200	15M	Corporate Operations, Finance
West Coast Hub	456 Market St, San Francisco, CA	150	12M	Tech Development, Customer Support
Midwest Office	789 Elm St, Chicago, IL	100	8M	Sales, Marketing
Southern Office	321 Pine St, Austin, TX	80	5M	Customer Support, Sales
Northeast Office	654 Maple St, Boston, MA	120	10M	Tech Development, HR
London Office	1 High St, London, UK	140	11M	Corporate Strategy, Marketing
Toronto Office	468 Palm St, Toronto, Canada	130	9M	Corporate Operations, Customer Support
Tokyo Office	5-2 Ginza St, Tokyo, Japan	110	10M	Product Development, Tech Support
Sydney Office	77 George St, Sydney, Australia	90	7M	Marketing, Customer Support
Berlin Office	22 Friedrichstrasse, Berlin, Germany	100	8M	Sales, Product Design
Paris Office	88 Champs-Élysées, Paris, France	95	9M	Marketing, Sales
Dubai Office	101 Sheikh Zayed Rd, Dubai, UAE	85	7M	Corporate Operations, Finance
Mumbai Office	55 Marine Drive, Mumbai, India	150	6M	Tech Development, Customer Support
Sao Paulo Office	33 Paulista Ave, Sao Paulo, Brazil	120	8M	Sales, Marketing
Cape Town Office	12 Table Mountain St, Cape Town, South Africa	70	5M	Customer Support, Sales
Amsterdam Office	9 Damrak St, Amsterdam, Netherlands	90	7M	Tech Support, HR

1. In our repository, we have some simple tools built around a popular vector database called Chroma. There are two files which will create a vector db (index) for the \*.py files in our repo and another to do the same for the office pdf. You can look at the files either via the usual "code " method or clicking on [tools/index\\_code.py](#) or [tools/index\\_pdf.py](#).

```
code tools/index_code.py
code tools/index_pdf.py
```

1. Let's create a vector database of our local python files. Run the program to index those as below. You'll see the program loading the embedding model that will turn the code chunks into numeric representations in the vector database and then it will read and index our .py files. *When you run the command below, there will be a very long pause while things get loaded.\**

```
python tools/index_code.py
```

```
(py_env) @techupskills + /workspaces/ai-3in1 (main) $ python tools/index_code.py  
Embedding model: all-MiniLM-L6-v2  
modules.json: 100%|██████████████████████████████████████| 349/349 [00:00<00:00, 3.55MB/s]  
config_sentence_transformers.json: 100%|██████████████████████████████████| 116/116 [00:00<00:00, 1.09MB/s]  
README.md: 10.5kB [00:00, 41.4MB/s]  
sentence_bert_config.json: 100%|██████████████████████████████████████| 53.0/53.0 [00:00<00:00, 494kB/s]  
config.json: 100%|██████████████████████████████████████| 612/612 [00:00<00:00, 6.39MB/s]  
model.safetensors: 100%|██████████████████████████████████████| 90.9M/90.9M [00:00<00:00, 128MB/s]  
tokenizer_config.json: 100%|██████████████████████████████████████| 350/350 [00:00<00:00, 3.34MB/s]  
vocab.txt: 232kB [00:00, 25.7MB/s]  
tokenizer.json: 466kB [00:00, 44.9MB/s]  
special_tokens_map.json: 100%|██████████████████████████████████████| 112/112 [00:00<00:00, 1.69MB/s]  
config.json: 100%|██████████████████████████████████████| 190/190 [00:00<00:00, 2.43MB/s]  
  
Indexed mcp_server.py  
Indexed mcp_agent.py  
Indexed agent.py  
Indexed tools/index_pdf.py  
Indexed tools/index_code.py  
Indexed tools/search.py  
Indexed tools/discover_tools.py  
Indexing complete: 7 Python files processed.  
New vector DB saved to ./chromadb  
  
(py_env) @techupskills + /workspaces/ai-3in1 (main) $
```

1. To help us do easy/simple searches against our vector databases, we have another tool at [tools/search.py](#). This tool connects to the ChromaDB vector database we create, and, using cosine similarity metrics, finds the top "hits" (matching chunks) and prints them out. You can open it and look at the code in the usual way if you want. No changes are needed to the code.

code tools/search.py

1. Now, let's run the search tool against the vector database we built in step 3. You can prompt it with phrases related to our coding like any of the ones shown below. When done, just type "exit". Notice the top hits and their respective cosine similarity values. Are they close? Farther apart?

```
python tools/search.py
```

```
convert celsius to fahrenheit
fastmcp tools
embed model sentence-transformers
async with Client mcp
```

T

DEBUG CONSOLE

TERMINAL

PORTS 1

🔍 Search: convert celcius to farenheit fastmcp tool

Collection contains 159 chunks.

Result 1/5

@mcp.tool  
def convert\_c\_to\_f(c: float) -> float:  
 """Simple Celsius → Fahrenheit conversion."""  
 return c \* 9 / 5 + 32

Cosine similarity: 0.6578  
Source: mcp\_server.py (chunk 21)

Result 2/5

#  
# 3. Instantiate FastMCP and define tool functions  
#  
mcp = FastMCP("WeatherServer")

Cosine similarity: 0.5640

1. Now, let's recreate our vector database based off of the PDF file. Type "exit" to end the current search. Then run the indexer for the pdf file.

```
python tools/index_pdf.py
```

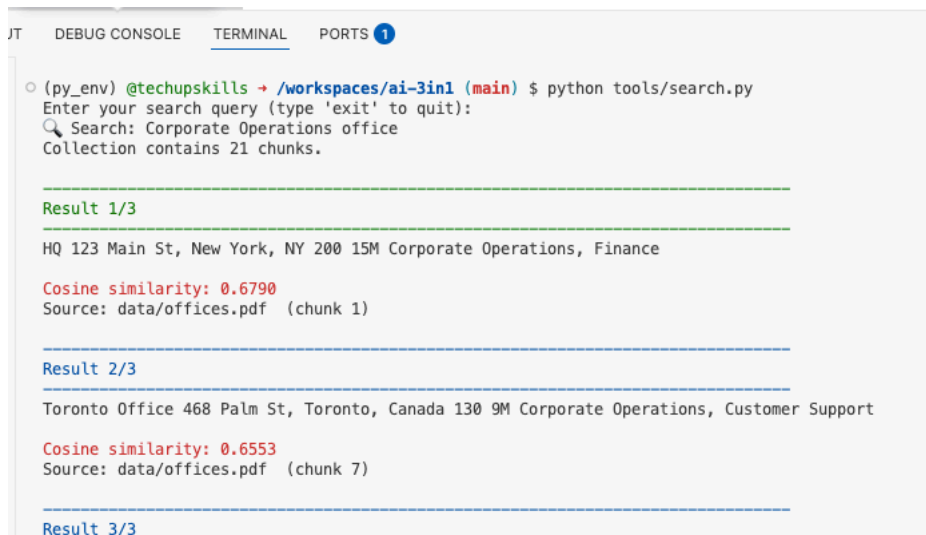
```
(py_env) @techupskills → /workspaces/ai-3in1 (main) $ python tools/index_pdf.py
Embedding model: all-MiniLM-L6-v2
→ Indexing offices.pdf
Indexing complete - new DB stored in ./chroma_db
(py_env) @techupskills → /workspaces/ai-3in1 (main) $
```

1. Now, we can run the same search tool to find the top hits for information about offices. Below are some prompts you can try here. Note that in some of them, we're using keywords only found in the PDF document. Notice the cosine similarity values on each - are they close? Farther apart? When done, just type "exit".

```
python tools/search.py
```

Queries:

Corporate Operations office  
Seaside cities  
Tech Development sites  
High revenue branch



```

JT  DEBUG CONSOLE  TERMINAL  PORTS 1
(py_env) @techupskills → /workspaces/ai-3in1 (main) $ python tools/search.py
Enter your search query (type 'exit' to quit):
Search: Corporate Operations office
Collection contains 21 chunks.

Result 1/3
HQ 123 Main St, New York, NY 200 15M Corporate Operations, Finance
Cosine similarity: 0.6790
Source: data/offices.pdf (chunk 1)

Result 2/3
Toronto Office 468 Palm St, Toronto, Canada 130 9M Corporate Operations, Customer Support
Cosine similarity: 0.6553
Source: data/offices.pdf (chunk 7)

Result 3/3

```

1. Keep in mind that this is not trying to intelligently answer your prompts at this point. This is a simple semantic search to find related chunks. In lab 5, we'll add in the LLM to give us better responses. In preparation for that lab, make sure that indexing for the PDF is the last one you ran and not the indexing for the Python files.

**\*\*[END OF LAB]\*\***

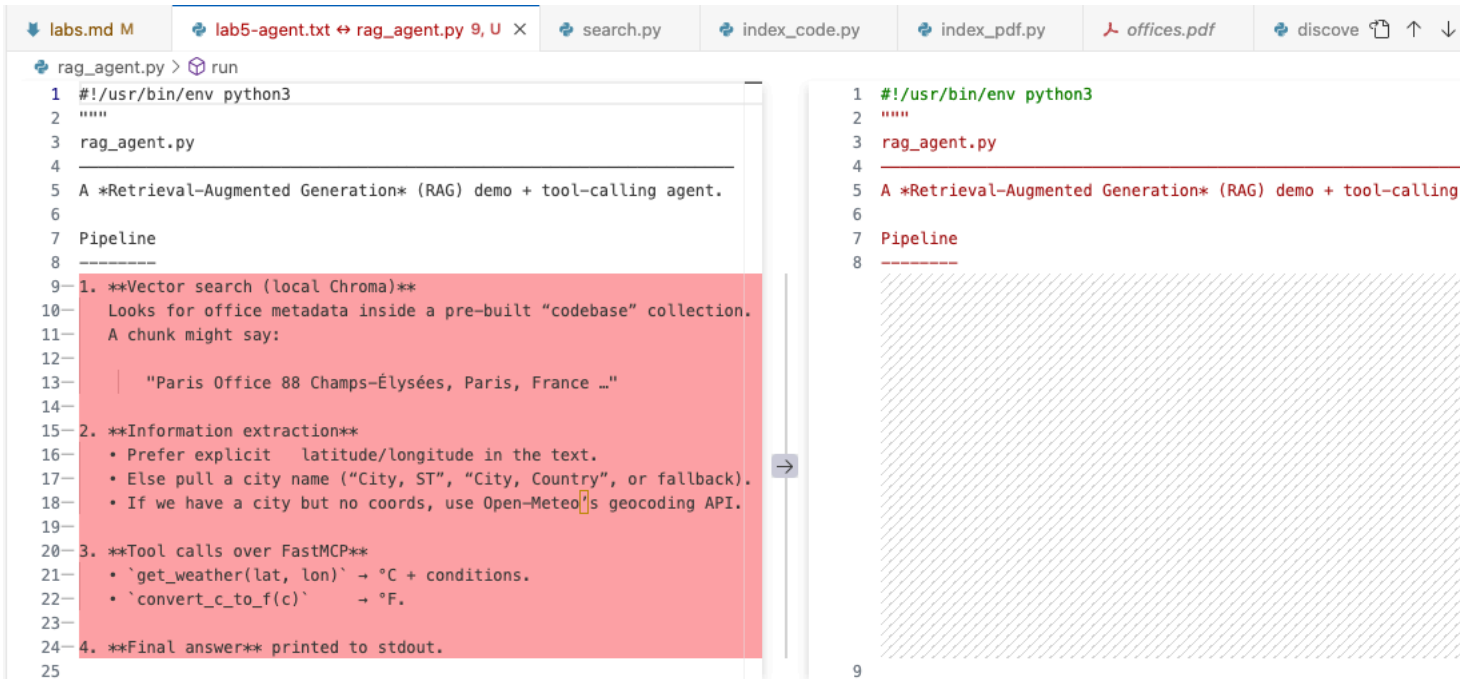
## Lab 5 - Using RAG with Agents

**Purpose:** In this lab, we'll explore how agents can leverage external data stores via RAG and tie in our previous tool use.

1. For this lab, we're going to combine our previous agent that looks up weather with RAG to get information about offices based on a prompt and tell us what the weather is like for that location.

1. We have a starter file for the new agent with rag in [rag\\_agent.py](#). As before, we'll use the "view differences and merge" technique to learn about the code we'll be working with. The command to run this time is below. There are a number of helper functions in this code that are useful to understand. Take some time to look at each section as you merge them in.

```
code -d labs/common/lab5_agent_solution.txt rag_agent.py
```



1. When you're done merging, close the tab as usual to save your changes. Now, if the MCP server is not still running from lab3, in a terminal, start it running again:

```
python mcp_server.py
```

1. In a separate terminal, start the new agent running.

```
python rag_agent.py
```

1. You'll see a `User:` prompt when it is ready for input from you. The agent is geared around you entering a prompt about an office. Try a prompt like one of the ones below about office "names" that are only in the PDF. **NOTE: After the first run, subsequent queries may take longer due to retries required for the open-meteo API that the MCP server is running.**

Tell me about HQ

Tell me about the Southern office



1. What you should see after that are some messages that show internal processing, such as the retrieved items from the RAG datastore. Then the agent will run through the necessary steps like parsing the query to find a location, getting the coordinates for the location, getting the weather etc. At the end it will print out an answer to your prompt and the weather determined from the tool.

```
(py_env) @techupskills → /workspaces/ai-3in1 (main) $ python rag_agent.py
Office-aware weather agent. Type 'exit' to quit.

Prompt: Tell me about HQ

Top RAG hit:
HQ 123 Main St, New York, NY 200 15M Corporate Operations, Finance

No coords found; geocoding 'New York, NY'.
Using coordinates: 40.7143, -74.0060

Weather: Clear sky, 89.6 °F

Prompt: █
```

1. After the initial run, you can try prompts about other offices or cities mentioned in the PDF. Type *exit* when done.

1. While this works, it could be more informative and user-friendly. Let's change the prompt and directions to the LLM to have it add an additional fact about the city where the office is located and include that and the weather in a more user-friendly response. To see and make the changes you can do the usual diff and merge using the command below.

```
code -d labs/common/lab5_agent_solution_v2.txt rag_agent.py
```

```

rag_agent.py > ...
1  #!/usr/bin/env python3
2-# rag_agent.py - now ends with an LLM-generated, human-friendly summary
3-#
4-# NEW FEATURES
5-# -----
6-# • After tool calls succeed we invoke Llama-3 (via LangChain-Ollama) s
7-#   model writes a concise paragraph that includes:
8-#       • Office name (if available) and city / country
9-#       • Current weather (conditions + °F)
10-#       • ONE interesting fact about the city
11-# • Works even if the indexed PDF line is messy; we hand the *raw* top-
12-#   line plus structured weather data to the LLM and let it compose.
13-#
14-# Prerequisites (additions)
15-#   pip install langchain-ollama
16-#
17-# Example
18-# -----
19-#   Prompt: paris marketing office
20-#
21-#   Top RAG hit:
22-#       Paris Office 88 Champs-Élysées, Paris, France 95 9M Marketing,
23-#
24-#   Final summary:
25-#       **Paris Office - Paris, France**

rag_agent.py M
requirements.txt

1  #!/usr/bin/env python3
2+ """
3+ rag_agent.py
4+
5+ A *Retrieval-Augmented Generation* (RAG) demo + tool-calling
6+
7+ Pipeline
8+ -----
9+ 1. **Vector search (local Chroma)**
10+    Looks for office metadata inside a pre-built "codebase" c
11+    A chunk might say:
12+
13+    "Paris Office 88 Champs-Élysées, Paris, France ..."
14+
15+ 2. **Information extraction**
16+    • Prefer explicit latitude/longitude in the text.
17+    • Else pull a city name ("City, ST", "City, Country", or
18+    • If we have a city but no coords, use Open-Meteo's geoco
19+
20+ 3. **Tool calls over FastMCP**
21+    • `get_weather(lat, lon)` → °C + conditions.
22+    • `convert_c_to_f(c)` → °F.
23+
24+ 4. **Final answer** printed to stdout.
25+

```

1. Once you've finished the merge, you can run the new agent code the same way again.

```
python rag_agent.py
```

1. Now, you can try the same queries as before and you should get more user-friendly answers.

Tell me about HQ

Tell me about the Southern office

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS 2
bash - requirements +

cepted
INFO: 127.0.0.1:49052 - "GET /mcp/ HTTP/1.1" 200 OK
INFO: 127.0.0.1:49062 - "POST /mcp/ HTTP/1.1" 200 OK
INFO: 127.0.0.1:39956 - "POST /mcp/ HTTP/1.1" 200 OK
INFO: 127.0.0.1:39966 - "POST /mcp/ HTTP/1.1" 200 OK
INFO: 127.0.0.1:39972 - "DELETE /mcp/ HTTP/1.1" 200 OK
[GIN] 2025/07/06 - 19:24:28 | 200 | 11.354703366s |
127.0.0.1 | POST "/api/chat"
INFO: 127.0.0.1:45318 - "POST /mcp/ HTTP/1.1" 200 OK
INFO: 127.0.0.1:45324 - "POST /mcp/ HTTP/1.1" 202 Accepted
cepted
INFO: 127.0.0.1:45336 - "GET /mcp/ HTTP/1.1" 200 OK
INFO: 127.0.0.1:45352 - "POST /mcp/ HTTP/1.1" 200 OK
INFO: 127.0.0.1:45364 - "POST /mcp/ HTTP/1.1" 200 OK
INFO: 127.0.0.1:45370 - "POST /mcp/ HTTP/1.1" 200 OK
INFO: 127.0.0.1:45376 - "DELETE /mcp/ HTTP/1.1" 200 OK

• (py_env) @techupskills → /workspaces/ai-3in1 (main) $ python rag_agent.py
Office-aware weather agent. Type 'exit' to quit.

Prompt: Tell me about HQ

Top RAG hit:
HQ 123 Main St, New York, NY 200 15M Corporate Operations, Finance

No coords found; geocoding 'New York, NY'.
Using coordinates: 40.7143, -74.0060

Here are three short sentences:

The HQ office is located in New York, NY. The current weather is a clear sky with temperatures at 89.8 °F. New York City was originally inhabited by the Lenape Native American tribe before becoming a major hub for finance and commerce.

```

1. When done, you can stop the MCP server via Ctrl-C and "exit" out of the agent.

\*\*[END OF LAB]\*\*

## Lab 6 - Building a Classification MCP Server

**Purpose:** In this lab, we'll transform our simple MCP server to use classifications and prompt templates. This creates a scalable architecture where the server manages query interpretation and templates, while the client focuses on LLM execution.

1. First, let's understand what we're building. The classification approach separates concerns:
2. **Server:** Query catalog, classification logic, prompt templates, data access
3. **Client:** LLM execution, workflow orchestration, result formatting

This makes adding new analysis capabilities much easier - you just update server configuration instead of modifying agent code.

1. We have a skeleton file for our new classification server that shows the structure. Let's examine it and then build it out using our familiar diff-and-merge approach.

```
code -d labs/common/lab6_mcp_server_solution.txt mcp_server_classification.py
```

mcp\_server\_classification.py &gt; ...

```

52
53 #
54 # 2. Canonical Query Definitions and Templates
55 #
56 CANONICAL_QUERIES = {
57     "revenue_stats": {
58         "description": "Calculate revenue statistics across all offices",
59         "parameters": [],
60         "data_requirements": ["revenue_million", "city"],
61         "prompt_template": """You are a financial analyst. Calculate revenue statistics across all offices.
62
63 Office Revenue Data:
64 {data}
65
66 Please calculate and report:
67 1. Which office has the highest revenue (city name and amount)
68 2. Which office has the lowest revenue (city name and amount)
69 3. Average revenue across all offices (in millions, to 1 decimal)
70 4. Total revenue across all offices
71
72 Be precise and only use the data provided above.""",
73         "example_queries": [
74             "What's the average revenue?",
75             "Show me revenue statistics",
76             "How much revenue do we make?",
77             "Which office has the highest revenue?",
78             "What office has the most revenue?"
79         ]
80     },
81 }

```

```

52
53 #
54 # 2. Canonical Query Definitions and Templates
55 #
56 CANONICAL_QUERIES = {
57     "revenue_stats": {
58         "description": "Calculate revenue statistics across all offices",
59         "parameters": [],
60         "data_requirements": ["revenue_million", "city"],
61         "prompt_template": """You are a financial analyst. Calculate revenue statistics across all offices.
62
63 Office Revenue Data:
64 {data}
65
66 Please calculate and report:
67 1. Which office has the highest revenue (city name and amount)
68 2. Which office has the lowest revenue (city name and amount)
69 3. Average revenue across all offices (in millions, to 1 decimal)
70 4. Total revenue across all offices
71
72 Be precise and only use the data provided above.""",
73         "example_queries": [
74             "What's the average revenue?",
75             "Show me revenue statistics",
76             "How much revenue do we make?",
77             "Which office has the highest revenue?",
78             "What office has the most revenue?"
79         ]
80     },
81 }

```

1. As you review the differences, note the key components:
2. **CANONICAL\_QUERIES**: A catalog of supported analysis types with templates and examples
3. **Classification tools**: `classify_canonical_query()` matches user intent to canonical queries
4. **Template tools**: `get_query_template()` returns structured prompts for the LLM
5. **Data tools**: `get_filtered_office_data()` provides specific data columns needed
6. **Validation tools**: `validate_query_parameters()` ensures proper inputs

1. Merge each section by clicking the arrows. Pay attention to:
2. How canonical queries are defined with descriptions, parameters, templates, and examples
3. The keyword matching logic in `classify_canonical_query()`
4. How templates use `{data}` placeholder for client-side data substitution
5. The filtering capabilities in `get_filtered_office_data()`

1. When finished merging, save the file by closing the tab. Now start the new server:

```
python mcp_server_classification.py
```

```

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS 8
ication.py

[10/07/25 01:00:57] INFO      Starting MCP server                                server.py:1429
                          'CanonicalQueryServer' with transport
                          'http' on http://127.0.0.1:8000/mcp/
/workspaces/ai-apps/py_env/lib/python3.11/site-packages/websockets/legacy/__init_
_.py:6: DeprecationWarning: websockets.legacy is deprecated; see https://websocke
ts.readthedocs.io/en/stable/howto/upgrade.html for upgrade instructions
  warnings.warn( # deprecated in 14.0 - 2024-11-09
/workspaces/ai-apps/py_env/lib/python3.11/site-packages/uvicorn/protocols/websock
ets/websockets_impl.py:17: DeprecationWarning: websockets.server.WebSocketServerP
rotocol is deprecated
  from websockets.server import WebSocketServerProtocol
INFO:      Started server process [87881]
INFO:      Waiting for application startup.
INFO:      Application startup complete.
INFO:      Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to quit)

```

1. The server should start and initialize its vector database. **This will take awhile to load and be ready.** You'll see:
2. Loading of the embedding model (all-MiniLM-L6-v2)
3. Initialization of ChromaDB at ./mcp\_chroma\_db
4. Population of two vector collections:
  - office\_locations (from PDF data)
  - office\_analytics (from CSV data)
5. List of available tool categories

The server now provides several categories of tools: - **Vector Search** : Semantic search for locations and analytics - **Weather** : Weather and geocoding tools (from previous labs) - **Classification** : Query intent classification - **Templates** : Prompt template management - **Structured Data** : Raw CSV data access - **Legacy Tools** : Alternative keyword-based search

```

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS 8

(py_env) @techupskills + /workspaces/ai-apps (update) $ code -d labs/common/lab6_
mcp_server_solution.txt mcp_server_classification.py
ication.py
Starting Canonical Query Classification MCP Server...

Available tool categories:
[Weather]: get_weather, convert_c_to_f
[Classification]: list_canonical_queries, classify_canonical_query
[Templates]: get_query_template, validate_query_parameters
[Data]: get_office_dataset, get_filtered_office_data

Workflow:
1. Client sends natural language query
2. Use classify_canonical_query to determine intent
3. Use get_query_template to get LLM prompt
4. Use data tools to get required data
5. Client executes LLM with template + data

```

1. Understanding the vector database architecture:
  - The MCP server now owns and manages the vector database
  - Both PDF (locations) and CSV (analytics) data are embedded
  - This centralized approach allows multiple agents to share the same data
  - The server creates semantic embeddings for intelligent query matching

Let's see the list of tools the MCP server makes available. Run the discovery tool again.

```
python tools/discover_tools.py
```

1. You should see several tool categories:
2. **New vector search tools:** `vector_search_locations` , `vector_search_analytics`
3. **Classification tools:** `classify_canonical_query` , `get_query_template` , `list_canonical_queries`
4. **Data access tools:** `get_office_dataset` , `get_filtered_office_data`
5. **Validation tools:** `validate_query_parameters`
6. **Weather tools:** `get_weather` , `geocode_location` , `convert_c_to_f`

```
(py_env) @brentlaster → /workspaces/ai-apps (main) $ python tools/discover_tools.py
Discovered 9 tool(s):

=====
Tool 1: get_weather
=====

Description
=====
Fetch current weather from Open-Meteo API with retry logic.
Returns dict with temperature, code, and conditions, or error key on failure.

=====
Tool 2: convert_c_to_f
=====

Description
=====
Convert Celsius to Fahrenheit.

=====
Tool 3: geocode_location
=====

Description
=====
```

1. The server is now ready to handle sophisticated query interpretation and provide structured templates for analysis. In the next lab, we'll build an agent that leverages these capabilities, so you can leave it running.

**\*\*[END OF LAB]\*\***

## Lab 7 - Building an Classification-Based RAG Agent

**Purpose:** In this lab, we'll build an agent that uses the classification server from Lab 6. This agent demonstrates the **4-step classification workflow: classify → template → data → execute**.

1. Let's build out the classification-based agent using our skeleton file. This agent demonstrates a **centralized data architecture** where:
2. The MCP server owns all data access (vector DB, raw files, embeddings)
3. The agent is pure orchestration (no local files, no local vector DB)
4. Queries are routed to either weather workflow (vector search + weather API) or classification workflow (structured analytics)

This represents best practices for production RAG systems with clear separation of concerns.

```
code -d labs/common/lab7_rag_agent_solution.txt rag_agent_classification.py
```

```

[Preview] README.md  labs.md  lab7_rag_agent_solution.txt ↔ rag_agent_classification.py 9+  discover_tools.py
rag_agent_classification.py > ...
47 #
48 def open_collection() -> chromadb.Collection:
49     """Return (or create) the Chroma collection."""
50     client = chromadb.PersistentClient(
51         path=str(CHROMA_PATH),
52         settings=Settings(),
53         tenant=DEFAULT_TENANT,
54         database=DEFAULT_DATABASE,
55     )
56     return client.get_or_create_collection(COLLECTION_NAME)
57
58 def rag_search(query: str, model: SentenceTransformer, coll: chromadb.Co
59     """Embed query and search vector DB."""
60     q_emb = model.encode(query).tolist()
61     res = coll.query(query_embeddings=[q_emb], n_results=TOP_K, include=
62     return res["documents"][0] if res["documents"] else []
63
64 #
65 # 3. Location extraction helpers (from previous labs)
66 #
67 def find_coords(texts: List[str]) -> Optional[Tuple[float, float]]:
68     for txt in texts:
69         for m in COORD_RE.finditer(txt):
70             lat, lon = map(float, m.groups())
71             if -90 <= lat <= 90 and -180 <= lon <= 180:
72                 return lat, lon
73     return None
74
75 def find_city_state(texts: List[str]) -> Optional[str]:
47 #
48 def open_collection() -> chromadb.Collection:
49     """Return (or create) the Chroma collection."""
50     client = chromadb.PersistentClient(
51         path=str(CHROMA_PATH),
52         settings=Settings(),
53         tenant=DEFAULT_TENANT,
54         database=DEFAULT_DATABASE,
55     )
56     return client.get_or_create_collection(COLLECTION_NAME)
57
58 def rag_search(query: str, model: SentenceTransformer, coll
59+
60 #
61 # 3. Location extraction helpers (from previous labs)
62 #
63 def find_coords(texts: List[str]) -> Optional[Tuple[float,
64     for txt in texts:
65         for m in COORD_RE.finditer(txt):
66             lat, lon = map(float, m.groups())
67             if -90 <= lat <= 90 and -180 <= lon <= 180:
68                 return lat, lon
69     return None
70
71 def find_city_state(texts: List[str]) -> Optional[str]:

```

1. As you review and merge the differences, observe the key architectural patterns:
2. **Simplified imports:** No chromadb, pdfplumber, or sentence-transformers needed
3. **MCP-centric data access:** All data comes from MCP server tools
4. **Query routing:** Keyword matching determines weather vs. analytics workflow
5. **Weather workflow:** Uses MCP's `vector_search_locations()` for semantic location matching
6. **Classification workflow:** 4-step process (classify → template → data → execute)
7. **Local LLM execution:** Agent only runs the LLM, data comes from server

1. Merge each section carefully. Notice two key functions:
2. `handle_weather_query()` : Now uses MCP's `vector_search_locations()` instead of local ChromaDB
3. `handle_canonical_query_with_classification()` : Orchestrates the 4-step classification workflow

The agent has no local file reading, no embeddings, no vector database - everything comes from MCP.

1. When finished merging, save the file. Make sure your classification server from Lab 6 is still running (if not, restart it).

1. Now start the classification agent in a second terminal:

```
python rag_agent_classification.py
```

1. The agent will start and explain that it uses classification and prompt templates. You can try out some of the queries shown below. Note that some may take multiple minutes to process and respond.

What's the weather like at our Chicago office?  
 Which office has the highest revenue?  
 What's the average revenue across our offices?  
 Which office has the most employees?  
 Tell me about the Austin office  
 What offices opened after 2014?

#### Example Queries:

🌤️ **Weather:** 'What is the weather at HQ?'  
 📊 **Analytics:** 'Which office has the most employees?'  
 🔍 **Semantic:** 'Show me offices with high revenue'

Query: What's the weather like at our Chicago office?

🌤️ Detected weather query, using RAG workflow...

Searching for office location: 'What's the weather like at our Chicago office?'

📍 Top RAG hit: Midwest Office 789 Elm St, Chicago, IL 100 8M Sales, Marketing...

Geocoding 'Chicago, IL'...

Geocoding error: No location found for 'Chicago, IL'. Try a different search term.

Using coordinates: 41.8500, -87.6500

Here is the weather summary:

Midwest Office, Chicago, IL - Clear Sky, 60.4°F

Interesting Fact: Did you know that Chicago is often referred to as the "Windy City," but it's actually named after the Chicago River, which was nicknamed for its tendency to have strong winds blowing from Lake Michigan?

Query: Which office has the highest revenue?

[INFO] Detected data analysis query, using classification workflow...

[1/4] Classifying canonical query...

[Result] Suggested query: revenue\_stats (confidence: 1.00)

[2/4] Getting prompt template...

[3/4] Fetching data: ['revenue\_million', 'city']

[4/4] Executing LLM with template...

#### 1. Observe the workflow differences:

**Weather queries** ("What's the weather at HQ?"): - Agent calls MCP's `vector_search_locations`("What's the weather at HQ?") - MCP performs semantic search in its vector database - Returns: "HQ 123 Main St, New York, NY..." - Agent extracts location and calls `geocode_location` and `get_weather`

**Analytics queries** ("Which office has the most employees?"): - Agent calls MCP's `classify_canonical_query`(...) - MCP returns: "employee\_analysis" - Agent calls `get_query_template`("employee\_analysis") - Agent calls `get_filtered_office_data`(columns=["employees", "city"]) - Agent executes LLM locally with template + data

1. Notice the architectural benefits:
2. **Weather queries:** Use MCP's semantic vector search (RAG via server)
3. **Analytics queries:** Use MCP's classification system (structured via server)
4. **No duplication:** MCP server owns all data, agent is pure orchestration
5. **Scalability:** Multiple agents could use the same MCP server
6. **Fallback:** If LLM times out, agent provides calculated results directly

This centralized architecture follows best practices.

The power of this architecture is that you can add new canonical queries just by updating the server configuration - no agent code changes needed. Type 'exit' when done to stop the agent.

\*\*[END OF LAB]\*\*

## Lab 8 - Creating a Streamlit Web Application



**Purpose:** In this lab, we'll create a modern web interface for our classification-based RAG agent using Streamlit. This provides a user-friendly way to interact with our canonical query system.

1. We've already created the start of a Streamlit app that leverages our rag\_classification agent. As usual, diff and merge so you can see key parts and complete it.

```
code -d labs/common/lab8_streamlit_solution.txt streamlit_app.py
```

```

[Preview] README.md  streamlit_app.py 9+  lab8_streamlit_solution.txt ↔ streamlit_app.py 9+ x  mcp_server_classification.py 9+  ▸ ▢ ↑ ↓
streamlit_app.py > ...
1  #!/usr/bin/env python3
2  """
3  Streamlit Web Application for Classification-Based RAG Agent
4  =====
5
6  This web app provides a user-friendly interface for our classification
7  Features:
8  - Chat-like interface for natural language queries
9  - Real-time display of classification process
10 - Confidence scores and alternative query suggestions
11 - Support for both weather and office data queries
12 - Visual indicators for processing steps
13 """
14
15 import asyncio
16 import streamlit as st
17 import time
18 import json
19 from pathlib import Path
20 import sys
21 from datetime import datetime
22
23 # Add current directory to path
24 sys.path.insert(0, str(Path(__file__).parent))
25
26 # Import our agent
27 from rag_agent_classification import process_query
28
29 # Page configuration
30 st.set_page_config(
31     page_title="AI Office Assistant",
32     page_icon="🏢",
33 )
  
```

```

1  #!/usr/bin/env python3
2  """
3  Streamlit Web Application for Classification-Based RAG Agent
4  =====
5
6  This web app provides a user-friendly interface for our class
7  Features:
8  - Chat-like interface for natural language queries
9  - Real-time display of classification process
10 - Confidence scores and alternative query suggestions
11 - Support for both weather and office data queries
12 - Visual indicators for processing steps
13 """
14
15 import asyncio
16
17 import time
18 import json
19 from pathlib import Path
20 import sys
21 from datetime import datetime
22
23 # Add current directory to path
24 sys.path.insert(0, str(Path(__file__).parent))
25
26 # Page configuration
27 st.set_page_config(
28     page_title="AI Office Assistant",
29     page_icon="🏢",
30 )
  
```

1. When done reviewing and merging, close the diff tab as usual.

1. Before we use the Streamlit app, make sure your MCP classification server from Lab 6 is running:

```
python mcp_server_classification.py
```



```

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS 3
(py_env) @brentlaster → /workspaces/ai-apps (main) $ python mcp_server_classification.py

=====
MCP Server with Vector Search & Canonical Queries
=====

[Initialization] Setting up vector database...
Loading embedding model: all-MiniLM-L6-v2...
✓ Embedding model loaded
Initializing ChromaDB at mcp_chroma_db...
✓ ChromaDB initialized
📍 Locations collection: 21 documents
📊 Analytics collection: 10 documents

=====

Available tool categories:
[Vector Search] 🔍
  • vector_search_locations - Semantic search for office locations
  • vector_search_analytics - Semantic search for office analytics
[Weather] 🌤️
  • get_weather, convert_c_to_f, geocode_location
[Classification] 🗂️
  • list_canonical_queries, classify_canonical_query

```

1. In an unused terminal, start the Streamlit application:

```
streamlit run streamlit_app.py
```

```

(py_env) @brentlaster → /workspaces/ai-apps (main) $ streamlit run streamlit_app.py

Collecting usage statistics. To deactivate, set browser.gatherUsageStats to false.

You can now view your Streamlit app in your browser.

Local URL: http://localhost:8501
Network URL: http://10.0.4.170:8501
External URL: http://74.249.85.201:8501

```

1. Your codespace will start the Streamlit app running at <http://localhost:8501>. You will probably see a dialog pop up to open Open a browser tab to that location. That does not always work. The recommend approach is to go to the **PORTS** tab (next to **TERMINAL**), find the row for **8501** and then hover over the *Forwarded Address* column and click on the "globe icon". See screenshot below.

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS 4 1			
Port	Forwarded Address	Running Process	Visibil
5301	<a href="https://curly-barnacle-gx...">https://curly-barnacle-gx...</a>	/vscode/bin/linux-x64/385651c938df8a906869babee...	Priv
8000	<a href="https://curly-barnacle-gx...">https://curly-barnacle-gx...</a>	python mcp_server_classification.py (156431)	Priv
8501 2	<a href="https://curly-ba...">https://curly-ba...</a> 3	/workspaces/ai-apps/py_env/bin/python3 /workspaces...	Priv
11434	<a href="https://curly-barnacle-gx...">https://curly-barnacle-gx...</a>	ollama serve (28501)	Priv
Add Port			

1. Explore the web interface:
2. **Sidebar**: Shows system status, query examples, and available data information
3. **Main area**: Enter queries and see real-time processing steps
4. **Processing indicators**: Watch as the system analyzes your query intent and routes it appropriately

MCP Server Connected

### Query Examples

**Revenue Analysis:**

- Which office has the highest revenue?
- What's the average revenue?
- Show me revenue statistics

**Employee Analysis:**

- Which office has the most employees?
- How are employees distributed?

**Office Profiles:**

- Tell me about the Chicago office
- What's the profile of New York?

**Weather Queries:**

- What's the weather at our Paris office?
- Temperature in New York office

**Available Data**

- Offices:** 10 locations
- Cities:** New York, Chicago, San Francisco, etc.
- Metrics:** Revenue, employees, opening year
- Weather:** Real-time data via API

**Conversation Memory**

Total Exchanges

# AI Office Assistant

Ask questions about office data, weather, and more.

## Ask Your Question

Enter your question:

e.g., Which office has the highest revenue?

Ask Assistant

**Your Question:** How are employees distributed?

## Processing Steps

- [1/4] Analyzing query intent...
- [2/4] Data analysis query detected - using classification workflow
- [3/4] Processing with AI agent...
- [4/4] Analysis complete!

## Assistant Response

Based on the provided data, here is a factual summary:

- The office with the most employees is New York, with 380 employees.
- There are 5 offices in total, each with 250 employees.
- To calculate the average number of employees per office, we divide the total number of employees by the number of offices.

- Here are some example queries you can try in the web interface: (the ones that go through the canonical flow may take a while initially)
- "Tell me about the Chicago office"
- "How are employees distributed?"
- "Which office has the highest revenue?"
- "What's the weather at our New York office?"
- "Which office has the most employees?"

- Notice how the web interface shows:
- Query analysis and intent detection
- Workflow routing (weather vs. data analysis)
- AI processing with the classification system
- Completion with structured results

1. The web interface provides several advantages:
2. **User-friendly:** Non-technical users can easily interact with the AI
3. **Visual feedback:** Clear indication of processing steps
4. **System monitoring:** Shows MCP server connection status
5. **Example guidance:** Built-in query examples and help

1. The Streamlit app includes a conversation memory dashboard in the sidebar (a feature of Streamlit). Look for:

- **Total Exchanges:** Counter showing how many conversations have been stored
- **Estimated Tokens:** Approximate token usage with progress bar
- **Offices Discussed:** Entity tracking showing which offices were mentioned
- **Clear Memory Button:** Reset conversation history
- **View History Button:** Toggle to see recent conversation excerpts

1. Try using the memory features:

- Ask about multiple offices and watch the "Offices Discussed" list grow
- Make several queries and see the token counter increase
- Click "View History" to see recent exchanges
- Click "Clear Memory" to reset and start fresh

### Conversation Memory

Total Exchanges

# 1

Estimated Tokens

# 548

### Offices Discussed

- Austin
- Chicago
- New York
- Seattle

Clear MemoryView History

Conversation History

00:33:29

Q: Tell me about the Chicago office...

A: **\*\*Chicago Office Profile\*\***

**\*\*Location and Basic Details:\*\***

...

1. When finished, stop the Streamlit app and the server with Ctrl-C in their respective terminals.

**\*\*[END OF LAB]\*\***

## Lab 9 - Deploying to Hugging Face Spaces

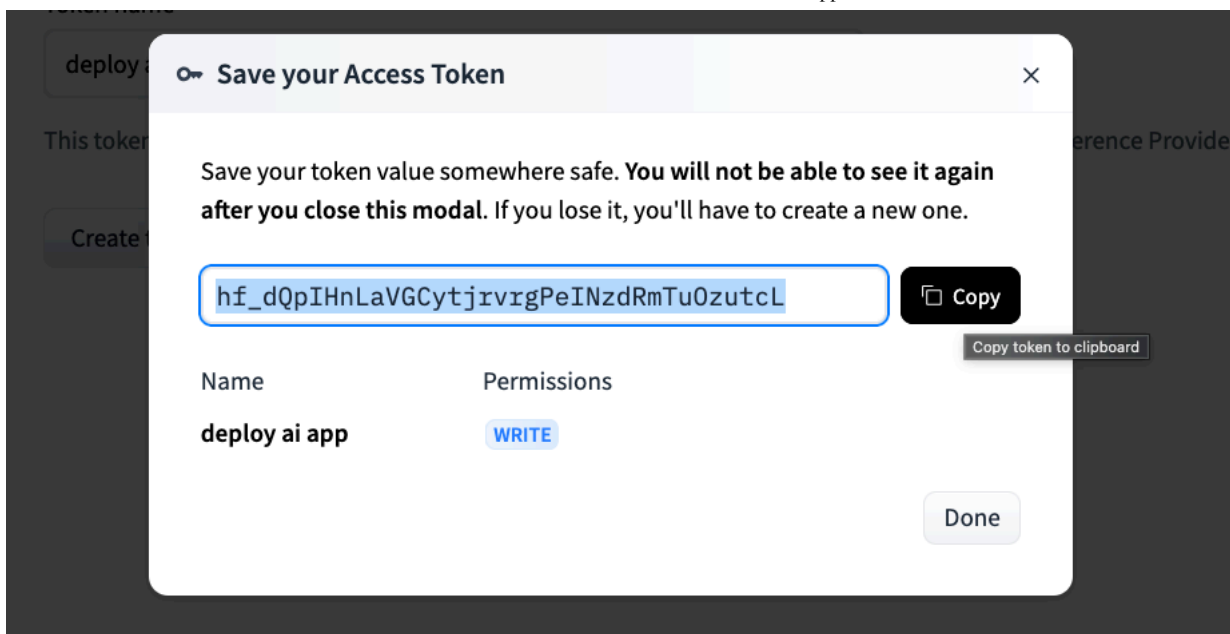
**Purpose:** In this lab, we'll deploy our classification-based RAG agent to Hugging Face Spaces, creating a publicly accessible web application.

1. **Prerequisites:** You'll need a free Hugging Face account. Go to <https://huggingface.co> and sign up if you haven't already.

1. Make sure you are logged in to your Hugging Face account. We need to have an access token to work with. Go to this URL: <https://huggingface.co/settings/tokens/new?tokenType=write> to create a new token. (Alternatively, you can go to your user Settings, then select Access Tokens, then Create new token, and select Write for the token type.) Select a name for the token and then Create token.

The screenshot displays the Hugging Face interface for creating a new access token. On the left, a sidebar shows the user's profile 'Brent Laster' with the username 'techupskills' and a list of settings: Profile, Account, Authentication, Organizations, Billing, Access Tokens, SSH and GPG Keys, and Inference Providers. The main content area is titled 'Create new Access Token'. Under 'Token type', 'Write' is selected, with a warning: 'This cannot be changed after token creation.' The 'Token name' field is filled with 'deploy ai app'. Below this, a message states: 'This token has read and write access to all your and your orgs resources'. A 'Create token' button is located at the bottom of the form.

1. After you click the Create button, your new token will be displayed on the screen. **Make sure to copy it and save it somewhere you can get to it for the next steps. You will not be able to see it again.**



1. Run the following command to login with your Hugging Face account credentials. Replace "" with the actual value of the token you created in the previous steps.

```
hf auth login --token <YOUR_SAVED_TOKEN>
```

```

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS 8
NAME      ID          SIZE      MODIFIED
(py_env) @techupskills → /workspaces/ai-apps (update) $ open rag_agent_classification.py
bash: open: command not found
(py_env) @techupskills → /workspaces/ai-apps (update) $ code rag_agent_classification.py
(py_env) @techupskills → /workspaces/ai-apps (update) $ hf auth login --token hf_IGKWlKZgx...
The token has not been saved to the git credentials helper. Pass `add_to_git_credential=True`
if you want to set the git credential as well.
Token is valid (permission: write).
The token `deploy ai app` has been saved to /home/vscode/.cache/huggingface/stored_tokens
Your token has been saved to /home/vscode/.cache/huggingface/token
Login successful.
The current active token is: `deploy ai app`

```

1. Now let's create a new Hugging Face Space. This can be done via the browser interface. But it can be quicker just to use this command line. Run the command below. ("aiapp" is the name of the space we're creating.

```
hf repo create --repo-type space --space_sdk docker aiapp
```

```

(py_env) @techupskills → /workspaces/ai-apps (update) $ hf repo create --repo-type space --space_sdk docker aiapp
Successfully created techupskills/aiapp on the Hub.
Your repo is now available at https://huggingface.co/spaces/techupskills/aiapp
(py_env) @techupskills → /workspaces/ai-apps (update) $

```

1. Next, clone your space repository. Make sure you are in the root of your project first.

```
cd /workspaces/ai-apps git clone https://huggingface.co/spaces/YOUR_USERNAME/aiapp
```

```

(py_env) @techupskills → /workspaces/ai-apps (update) $ git clone https://huggingface.co/spaces/techupskills/aiapp
Cloning into 'aiapp'...
remote: Enumerating objects: 4, done.
remote: Total 4 (delta 0), reused 0 (delta 0), pack-reused 4 (from 1)
Unpacking objects: 100% (4/4), 1.24 KiB | 1.24 MiB/s, done.

```

- Now, we need to copy the deployment files we need into the new repo. Run the following copy commands. To make this simple, there's already a script that will copy and locate the needed files in [scripts/copyfiles.sh](#). You can look at that with the usual methods if you want. When ready, go ahead and change into the *aiapp* directory and run it. **This will run very quickly.** `cd aiapp ../scripts/copyfiles.sh`

- Push the changes back to the Hugging Face repo. **Notice there is a required "." at the end of the command.** `hf upload --repo-type space aiapp .`

```
(py_env) @brentlaster → /workspaces/ai-apps/aiapp (main) $ hf upload --repo-type space aiapp .
Start hashing 12 files.
Finished hashing 12 files.
Removing 1 file(s) from commit that have not changed.
https://huggingface.co/spaces/techupskills/aiapp/tree/main/.
(py_env) @brentlaster → /workspaces/ai-apps/aiapp (main) $
```

- Monitor your deployment by going to your space page on Hugging Face. Go to [https://huggingface.co/spaces/YOUR\\_USERNAME/aiapp](https://huggingface.co/spaces/YOUR_USERNAME/aiapp).
- You'll see the build process and container execution in the logs
- The Space will automatically start once the build completes (takes several minutes)

The screenshot shows the Hugging Face Spaces interface for the space `techupskills/aiapp`. The status is `Building`. The `Logs` tab is active, displaying the following build logs:

```
-----
4.0/4.0 MB 398.1 MB/s 0:00:00
Downloading watchfiles-1.1.0-cp311-cp311-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (453 kB)
Downloading websocket_client-1.9.0-py3-none-any.whl (82 kB)
Downloading websockets-15.0.1-cp311-cp311-manylinux_2_5_x86_64.manylinux1_x86_64.manylinux_2_17_x86_64.manylinux2014_x86_64.whl (182 kB)
Downloading zipp-3.23.0-py3-none-any.whl (10 kB)
Downloading zstandard-0.25.0-cp311-cp311-manylinux2014_x86_64.manylinux_2_17_x86_64.whl (5.6 MB)
5.6/5.6 MB 467.3 MB/s 0:00:00
Downloading coloredlogs-15.0.1-py2.py3-none-any.whl (46 kB)
Downloading humanfriendly-10.0-py2.py3-none-any.whl (86 kB)
Downloading filelock-3.20.0-py3-none-any.whl (16 kB)
Downloading flatbuffers-25.9.23-py2.py3-none-any.whl (30 kB)
Downloading importlib_resources-6.5.2-py3-none-any.whl (37 kB)
Downloading networkx-3.5-py3-none-any.whl (2.0 MB)
2.0/2.0 MB 745.9 MB/s 0:00:00
Downloading pycparser-2.23-py3-none-any.whl (118 kB)
Downloading pyproject_hooks-1.2.0-py3-none-any.whl (10 kB)
Downloading scikit_learn-1.7.2-cp311-cp311-manylinux2014_x86_64.manylinux_2_17_x86_64.whl (9.7 MB)
9.7/9.7 MB 368.5 MB/s 0:00:00
Downloading joblib-1.5.2-py3-none-any.whl (308 kB)
Downloading scipy-1.16.2-cp311-cp311-manylinux2014_x86_64.manylinux_2_17_x86_64.whl (35.9 MB)
```

- Test your deployed application.
  - Once the Space is running, you'll see your Streamlit app

- Try the same queries you tested locally

The screenshot shows a web browser at [huggingface.co/spaces/techupskills/aiapp2](https://huggingface.co/spaces/techupskills/aiapp2). The interface includes a sidebar on the left with the following sections:

- System Status:** MCP Server Connected
- Query Examples:**
  - Revenue Analysis:**
    - Which office has the highest revenue?
    - What's the average revenue?
    - Show me revenue statistics
  - Employee Analysis:**
    - Which office has the most employees?
    - How are employees distributed?
  - Office Profiles:**
    - Tell me about the Chicago office
    - What's the profile of New York?
  - Weather Queries:**
    - What's the weather at our Paris office?
    - Temperature in New York office

The main content area features a large heading "AI Office Assistant" and a subheading "Ask questions about office data, weather, and more". Below this is a section titled "Ask Your Question" with a text input field containing "e.g., Which office has the highest revenue?" and a red "Ask" button. A "Your Question:" section displays "How are employees distributed?". At the bottom, a "Processing Steps" section shows "[1/4] Analyzing query intent..." in a yellow bar.

1. You can share your URL to share your app with others.

- Your Space is now publicly accessible at: [https://huggingface.co/spaces/YOUR\\_USERNAME/aiapp](https://huggingface.co/spaces/YOUR_USERNAME/aiapp)
- You can share this URL with others to demonstrate your AI application

**Congratulations!** You've successfully deployed an AI application with sophisticated classification capabilities on the web. Your application demonstrates advanced concepts like canonical query classification, use of RAG and MCP protocols and user-friendly AI interfaces.

**\*\*[END OF LAB]\*\***

**\*\*For educational use only by the attendees of our workshops.\*\***

**\*\* (c) 2025 Tech Skills Transformations and Brent C. Laster. All rights reserved. \*\***