

2025 Weaviate

AI and Machine Learning

Tony Shen
Data Communications Lab
7/22/2025

Contents

Introduction 2

Test Environment..... 2

 Hardware Components..... 2

 Software Components 4

Available Models..... 4

Weaviate Test Run 5

 Creating Weaviate database..... 6

 Installing Weaviate client library 8

 Creating Client Connection Wrapper 10

 Injecting Data Using Client Connection Wrapper 12

 Run Semantic Query Using Client Connection Wrapper 14

 Run RAG Using Client Connection Wrapper 20

Summary 22

Introduction

Weaviate is a popular open-source vector database. Vector databases play a central role in many LLM-powered applications, especially those requiring semantic search, retrieval augmented generation (RAG), or context-aware responses. In this short article, we will walk you through lab work by showing you how to download, install Weaviate, and run a few performance tests locally on your PC. This short article is intended for AI beginners. By following along the lab, you may gain hands-on experience to strengthen your understanding of what vector databases, semantic search, and RAG are and how they work.

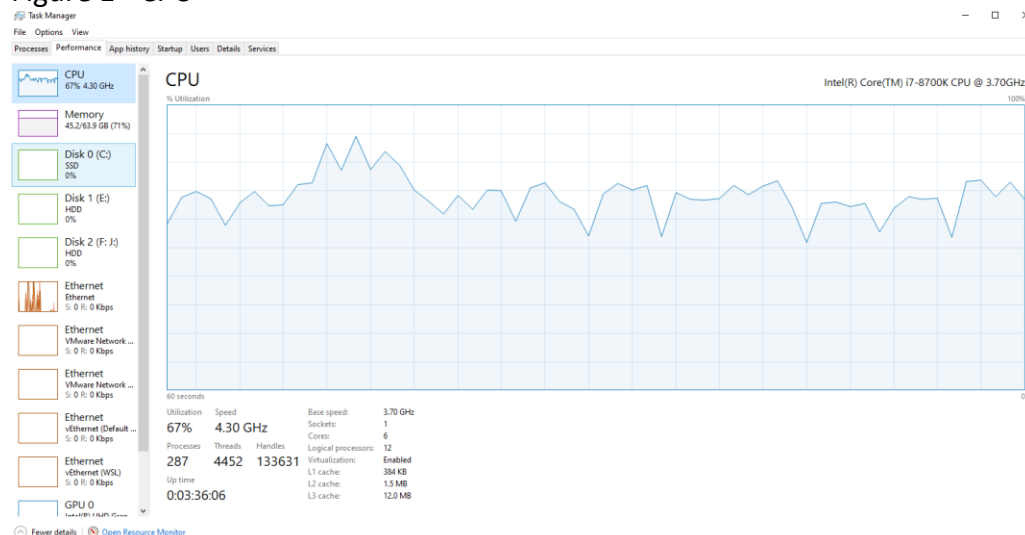
Test Environment

A Windows 10 PC provides the test environment. The PC is equipped with 64GB of memory, an RTX-4070 (a lower-end Nvidia GPU card), and sufficient disk space to accommodate a few selected Opensource LLM models. The test run was conducted locally on the PC. The models were accessed via Ollama, and Open Web UI provided the user interface.

Hardware Components

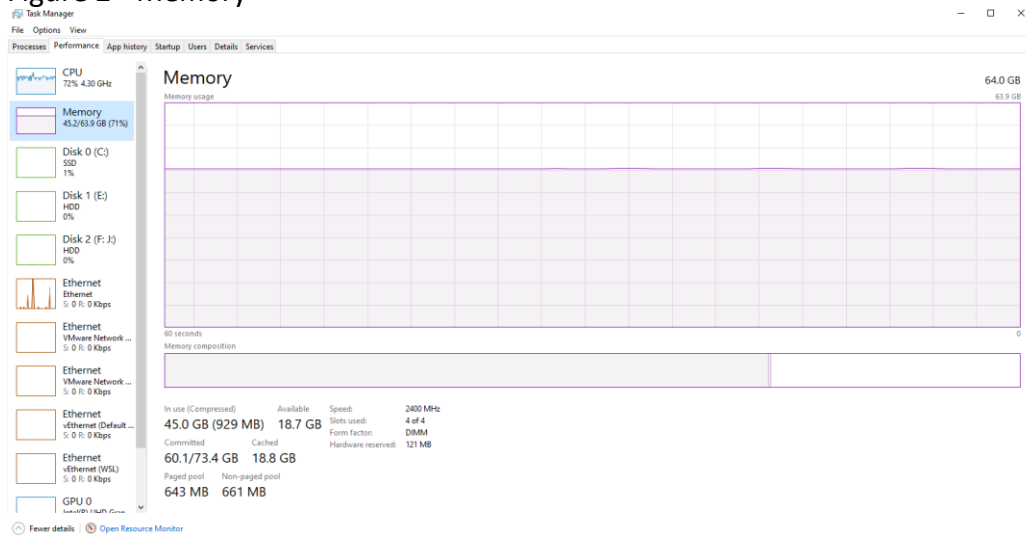
CPU – Intel Core i7-8700K CPU @ 3.70Ghz

Figure 1 - CPU



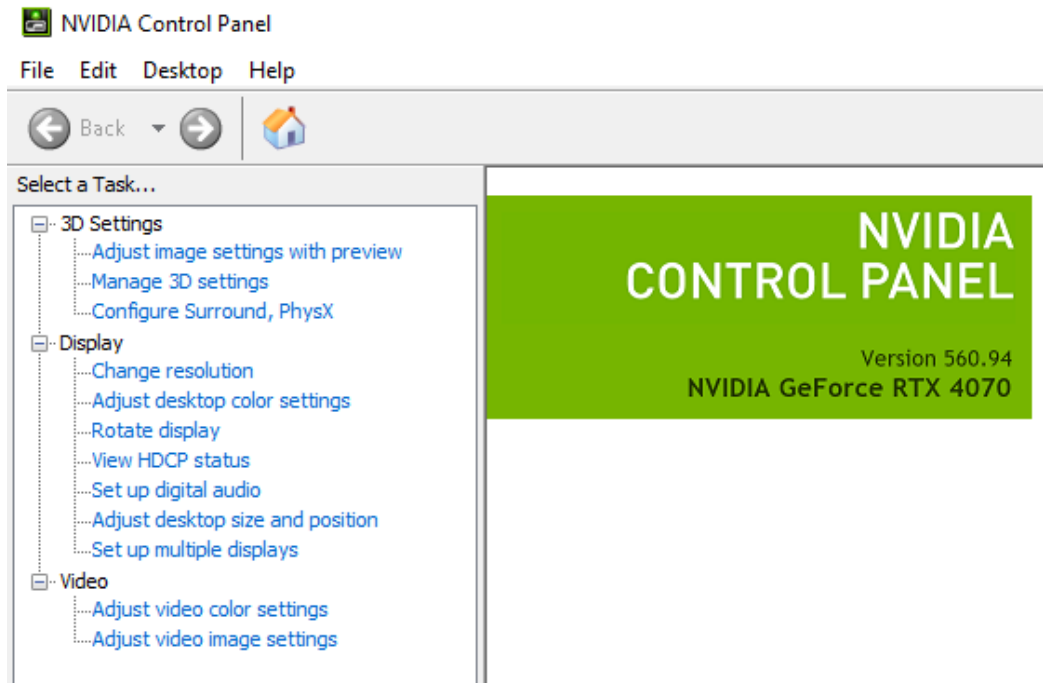
Memory – 64GB

Figure 2 - Memory



GPU – Nvidia GeForce RTX 4070

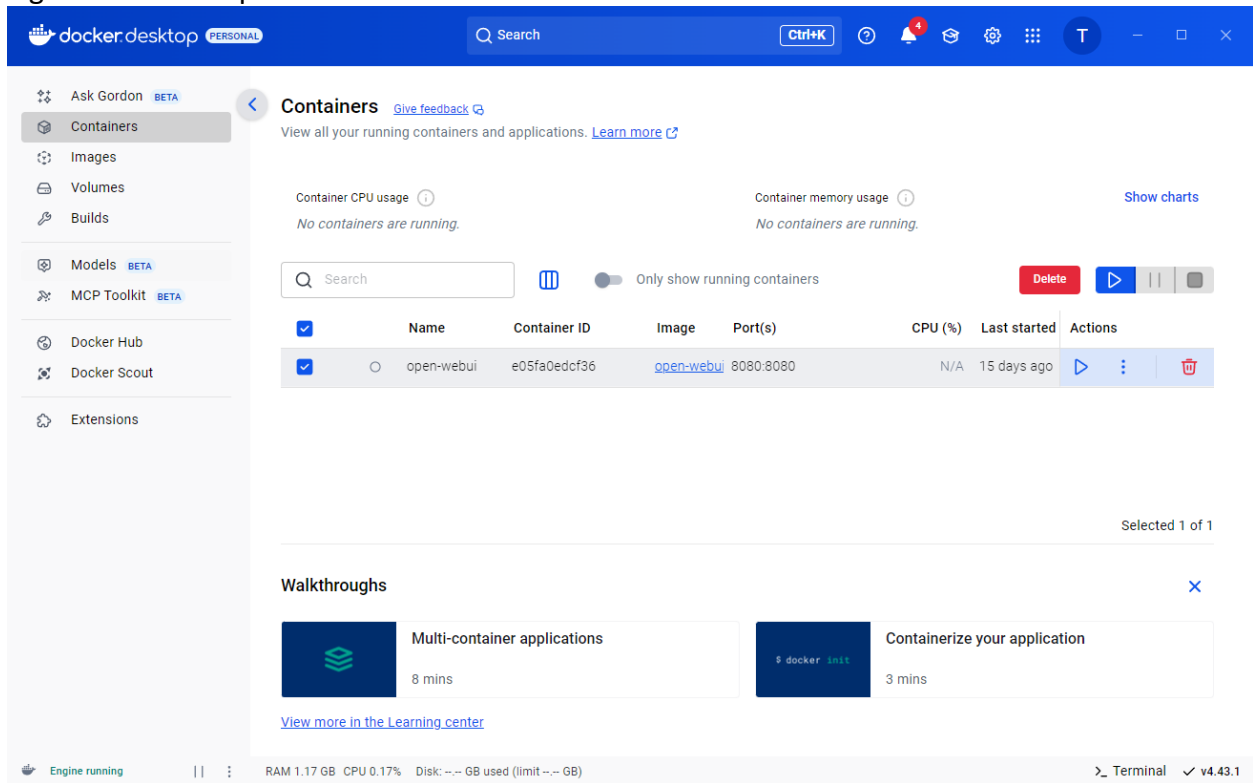
Figure 3 - GPU



Software Components

1. Ollama for Windows
2. Docker Desktop for Windows
3. Open Web UI that runs as a Docker container web server (See Figure 4 below)

Figure 4 Docker Open Web UI container



Available Models

Figure 5 – Available models to run

```
C:\Users\Tony>ollama list
NAME                                ID                                SIZE    MODIFIED
llama3.2:latest                     a80c4f17acd5                     2.0 GB  2 days ago
nomic-embed-text:latest             0a109f422b47                     274 MB  2 days ago
gemma3:27b                          a418f5838ea+                     17 GB   11 days ago
gemma3n:e4b                         15cb39fd9394                     7.5 GB  3 weeks ago
gemma3n:latest                      e8975a94482c                     7.5 GB  3 weeks ago
gemma3:latest                       a2af6cc3eb7f                     3.3 GB  3 weeks ago
deepseek-r1:671b                    739e1b229ad7                     404 GB  5 months ago
llama3.3:latest                     a6eb4748fd29                     42 GB   5 months ago
deepseek-r1:70b                     0c1615a8ca32                     42 GB   5 months ago
deepseek-r1:latest                  0a8c26691023                     4.7 GB  5 months ago
deepseek-r1:32b                     38056bbcbb2d                     19 GB   5 months ago

C:\Users\Tony>
```

2025 Weaviate

For us to run Weaviate, we will use the first two models, namely, **llama3.2** and **nomic-embed-text** as highlighted in Figure 5 above.

If you don't have those models yet, open a command window (CMD), and run the following commands to install the models

```
ollama pull nomic-embed-text  
ollama pull llama3.2
```

The test run requires Python 3.8 or above to run. The author uses Python 3.13. To check your Python version, please run the following command, and you should see a return of your Python version like this in Figure 6 below. If you don't have Python installed yet or you need to upgrade your Python, you may go to Python website at <https://www.python.org/downloads/> to download and install a recent version of your choice.

Figure 6 – Python version



```
Administrator: Command Prompt  
C:\Users\Tony>python --version  
Python 3.13.5  
C:\Users\Tony>
```

Now that you have met all prerequisites for the test run, you are ready to proceed with Weaviate test run procedure

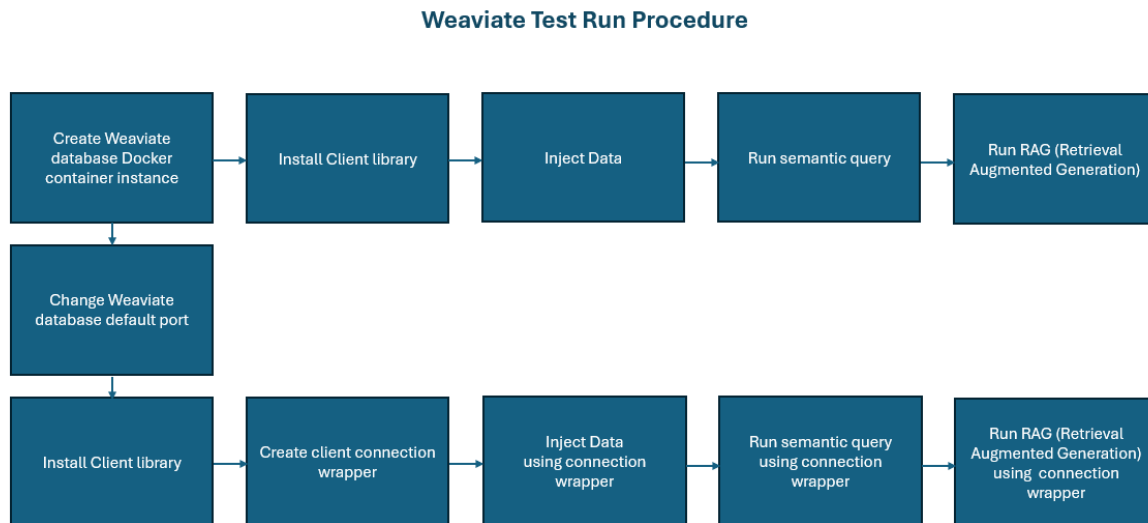
Weaviate Test Run

Weaviate uses port 8080 for the Weaviate client to communicate with the database. Open Web UI also uses port 8080. To avoid the conflict, you may change Weaviate to use a different port. Figure 7 below shows two paths of running the test. The top one uses the default port 8080, straightforward, no changes required in any test scripts and the environment. The bottom path uses port **8082**, a custom port. To for Weaviate to use the custom port in this lab, you need to revise the Weaviate database setup script, build a connection wrapper for the Weaviate client to use the custom port instead of the default port, and revise all Weaviate client scripts to be used in this lab accordingly. You can find the Weaviate provided scripts at <https://docs.weaviate.io/weaviate/quickstart/local>

2025 Weaviate

Figure 7 below shows two paths to do this lab. The top path uses the Weaviate provided scripts without changes. The bottom path involves revising the Weaviate provided scripts and creating a couple of new ones.

Figure 7 – Weaviate Test Run Procedure



In the following sections, we show you how to prepare and run Weaviate locally by taking the bottom path.

Creating Weaviate database

Before creating the database, let's create a project folder. See the example in Figure 8 below. You may make a fold of your choice in your system.

Figure 8 – Project folder



At the folder's root, create an environment file with three lines in it as Figure 9 below shows. Name the file `.env`.

2025 Weaviate

Figure 9 – Environment file .env

```
F:\app\weaviate>more .env
WEAVIATE_HOST=127.0.0.1
WEAVIATE_PORT=8082
WEAVIATE_GRPC_PORT=50051
F:\app\weaviate>
```

Create Weaviate database Docker container setup script as shown in Figure 10 below. In the script, make sure the highlighted lines specify your desired custom port, in this example, port **8082**. Name the script **docker-compose.yml**. Place the file at your project folder root.

Figure 10 – Weaviate database Docker container setup script

```
Administrator: Command Prompt
F:\app\weaviate>more docker-compose.yml
---
services:
  weaviate:
    command:
      - --host
      - 0.0.0.0
      - --port
      - '8082'
      - --scheme
      - http
    image: cr.weaviate.io/semitechnologies/weaviate:1.32.0
    ports:
      - 8082:8082
      - 50051:50051
    volumes:
      - weaviate_data:/var/lib/weaviate
    restart: on-failure:0
    environment:
      QUERY_DEFAULTS_LIMIT: 25
      AUTHENTICATION_ANONYMOUS_ACCESS_ENABLED: 'true'
      PERSISTENCE_DATA_PATH: '/var/lib/weaviate'
      ENABLE_API_BASED_MODULES: 'true'
      ENABLE_MODULES: 'text2vec-ollama,generative-ollama'
      CLUSTER_HOSTNAME: 'node1'
volumes:
  weaviate_data:
  ...
F:\app\weaviate>
```

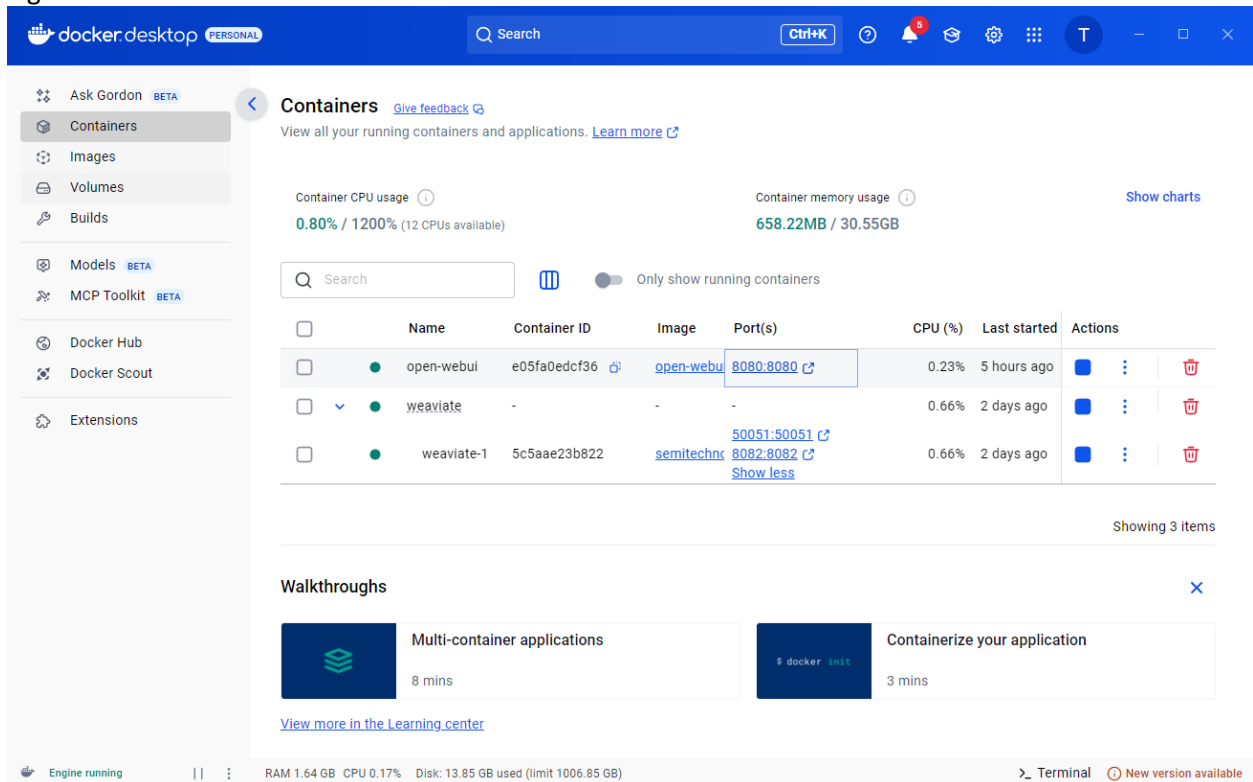
At the project folder root, run the following command to set up and start Weaviate database Docker instance

docker-compose up -d

Your Docker Containers Pane shall display Weaviate database container instance up and running as Figure 11 shows below.

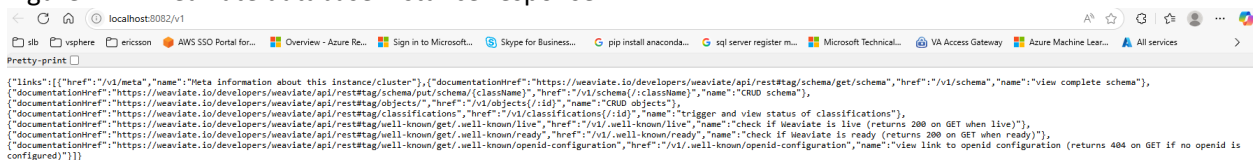
2025 Weaviate

Figure 11 – Docker containers



Click on the Weaviate database ports 8082:8082, it opens a tab in your browser as shown below in Figure 12, indicating the database instance is working and responsive to requests

Figure 12 – Weaviate database instance response



Thus, you have created Weaviate database and changed its port from default 8080 to a custom port 8082.

Installing Weaviate client library

Run this command.

pip install -U weaviate-client

Make sure the messages displayed thereafter indicated downloading, collecting, installing, and finally successfully installed weaviate client and its required modules as shown below:

F:\app\weaviate>**pip install -U weaviate-client**

Collecting weaviate-client

2025 Weaviate

Downloading weaviate_client-4.16.1-py3-none-any.whl.metadata (3.7 kB)
Collecting httpx<0.29.0,>=0.26.0 (from weaviate-client)
Downloading httpx-0.28.1-py3-none-any.whl.metadata (7.1 kB)
Collecting validators<1.0.0,>=0.34.0 (from weaviate-client)
Downloading validators-0.35.0-py3-none-any.whl.metadata (3.9 kB)
Collecting authlib<2.0.0,>=1.2.1 (from weaviate-client)
Downloading authlib-1.6.1-py2.py3-none-any.whl.metadata (1.6 kB)
Collecting pydantic<3.0.0,>=2.8.0 (from weaviate-client)
Downloading pydantic-2.11.7-py3-none-any.whl.metadata (67 kB)
Collecting grpcio<2.0.0,>=1.66.2 (from weaviate-client)
Downloading grpcio-1.73.1-cp313-cp313-win_amd64.whl.metadata (4.0 kB)
Collecting grpcio-health-checking<2.0.0,>=1.66.2 (from weaviate-client)
Downloading grpcio_health_checking-1.73.1-py3-none-any.whl.metadata (1.0 kB)
Collecting deprecation<3.0.0,>=2.1.0 (from weaviate-client)
Downloading deprecation-2.1.0-py2.py3-none-any.whl.metadata (4.6 kB)
Collecting cryptography (from authlib<2.0.0,>=1.2.1->weaviate-client)
Downloading cryptography-45.0.5-cp311-abi3-win_amd64.whl.metadata (5.7 kB)
Collecting packaging (from deprecation<3.0.0,>=2.1.0->weaviate-client)
Downloading packaging-25.0-py3-none-any.whl.metadata (3.3 kB)
Collecting protobuf<7.0.0,>=6.30.0 (from grpcio-health-checking<2.0.0,>=1.66.2->weaviate-client)
Downloading protobuf-6.31.1-cp310-abi3-win_amd64.whl.metadata (593 bytes)
Collecting anyio (from httpx<0.29.0,>=0.26.0->weaviate-client)
Downloading anyio-4.9.0-py3-none-any.whl.metadata (4.7 kB)
Collecting certifi (from httpx<0.29.0,>=0.26.0->weaviate-client)
Downloading certifi-2025.7.14-py3-none-any.whl.metadata (2.4 kB)
Collecting httpcore==1.* (from httpx<0.29.0,>=0.26.0->weaviate-client)
Downloading httpcore-1.0.9-py3-none-any.whl.metadata (21 kB)
Collecting idna (from httpx<0.29.0,>=0.26.0->weaviate-client)
Downloading idna-3.10-py3-none-any.whl.metadata (10 kB)
Collecting h11>=0.16 (from httpcore==1.*->httpx<0.29.0,>=0.26.0->weaviate-client)
Downloading h11-0.16.0-py3-none-any.whl.metadata (8.3 kB)
Collecting annotated-types>=0.6.0 (from pydantic<3.0.0,>=2.8.0->weaviate-client)
Downloading annotated_types-0.7.0-py3-none-any.whl.metadata (15 kB)
Collecting pydantic-core==2.33.2 (from pydantic<3.0.0,>=2.8.0->weaviate-client)
Downloading pydantic_core-2.33.2-cp313-cp313-win_amd64.whl.metadata (6.9 kB)
Collecting typing-extensions>=4.12.2 (from pydantic<3.0.0,>=2.8.0->weaviate-client)
Downloading typing_extensions-4.14.1-py3-none-any.whl.metadata (3.0 kB)
Collecting typing-inspection>=0.4.0 (from pydantic<3.0.0,>=2.8.0->weaviate-client)
Downloading typing_inspection-0.4.1-py3-none-any.whl.metadata (2.6 kB)
Collecting sniffio>=1.1 (from anyio->httpx<0.29.0,>=0.26.0->weaviate-client)
Downloading sniffio-1.3.1-py3-none-any.whl.metadata (3.9 kB)
Collecting cffi>=1.14 (from cryptography->authlib<2.0.0,>=1.2.1->weaviate-client)
Downloading cffi-1.17.1-cp313-cp313-win_amd64.whl.metadata (1.6 kB)
Collecting pycparser (from cffi>=1.14->cryptography->authlib<2.0.0,>=1.2.1->weaviate-client)
Downloading pycparser-2.22-py3-none-any.whl.metadata (943 bytes)
Downloading weaviate_client-4.16.1-py3-none-any.whl (451 kB)
Downloading authlib-1.6.1-py2.py3-none-any.whl (239 kB)
Downloading deprecation-2.1.0-py2.py3-none-any.whl (11 kB)

2025 Weaviate

```
Downloading grpcio-1.73.1-cp313-cp313-win_amd64.whl (4.3 MB)
----- 4.3/4.3 MB 64.0 MB/s eta 0:00:00
Downloading grpcio_health_checking-1.73.1-py3-none-any.whl (18 kB)
Downloading httpx-0.28.1-py3-none-any.whl (73 kB)
Downloading httpcore-1.0.9-py3-none-any.whl (78 kB)
Downloading protobuf-6.31.1-cp310-abi3-win_amd64.whl (435 kB)
Downloading pydantic-2.11.7-py3-none-any.whl (444 kB)
Downloading pydantic_core-2.33.2-cp313-cp313-win_amd64.whl (2.0 MB)
----- 2.0/2.0 MB 57.2 MB/s eta 0:00:00
Downloading validators-0.35.0-py3-none-any.whl (44 kB)
Downloading annotated_types-0.7.0-py3-none-any.whl (13 kB)
Downloading h11-0.16.0-py3-none-any.whl (37 kB)
Downloading typing_extensions-4.14.1-py3-none-any.whl (43 kB)
Downloading typing_inspection-0.4.1-py3-none-any.whl (14 kB)
Downloading anyio-4.9.0-py3-none-any.whl (100 kB)
Downloading idna-3.10-py3-none-any.whl (70 kB)
Downloading sniffio-1.3.1-py3-none-any.whl (10 kB)
Downloading certifi-2025.7.14-py3-none-any.whl (162 kB)
Downloading cryptography-45.0.5-cp311-abi3-win_amd64.whl (3.4 MB)
----- 3.4/3.4 MB 64.2 MB/s eta 0:00:00
Downloading cffi-1.17.1-cp313-cp313-win_amd64.whl (182 kB)
Downloading packaging-25.0-py3-none-any.whl (66 kB)
Downloading pycparser-2.22-py3-none-any.whl (117 kB)
Installing collected packages: validators, typing-extensions, sniffio, pycparser, protobuf, packaging, idna,
h11, grpcio, certifi, annotated-types, typing-inspection, pydantic-core, httpcore, grpcio-health-checking,
deprecation, cffi, anyio, pydantic, httpx, cryptography, authlib, weaviate-client
Successfully installed annotated-types-0.7.0 anyio-4.9.0 authlib-1.6.1 certifi-2025.7.14 cffi-1.17.1
cryptography-45.0.5 deprecation-2.1.0 grpcio-1.73.1 grpcio-health-checking-1.73.1 h11-0.16.0 httpcore-
1.0.9 httpx-0.28.1 idna-3.10 packaging-25.0 protobuf-6.31.1 pycparser-2.22 pydantic-2.11.7 pydantic-
core-2.33.2 sniffio-1.3.1 typing-extensions-4.14.1 typing-inspection-0.4.1 validators-0.35.0 weaviate-
client-4.16.1
```

F:\app\weaviate>

Creating Client Connection Wrapper

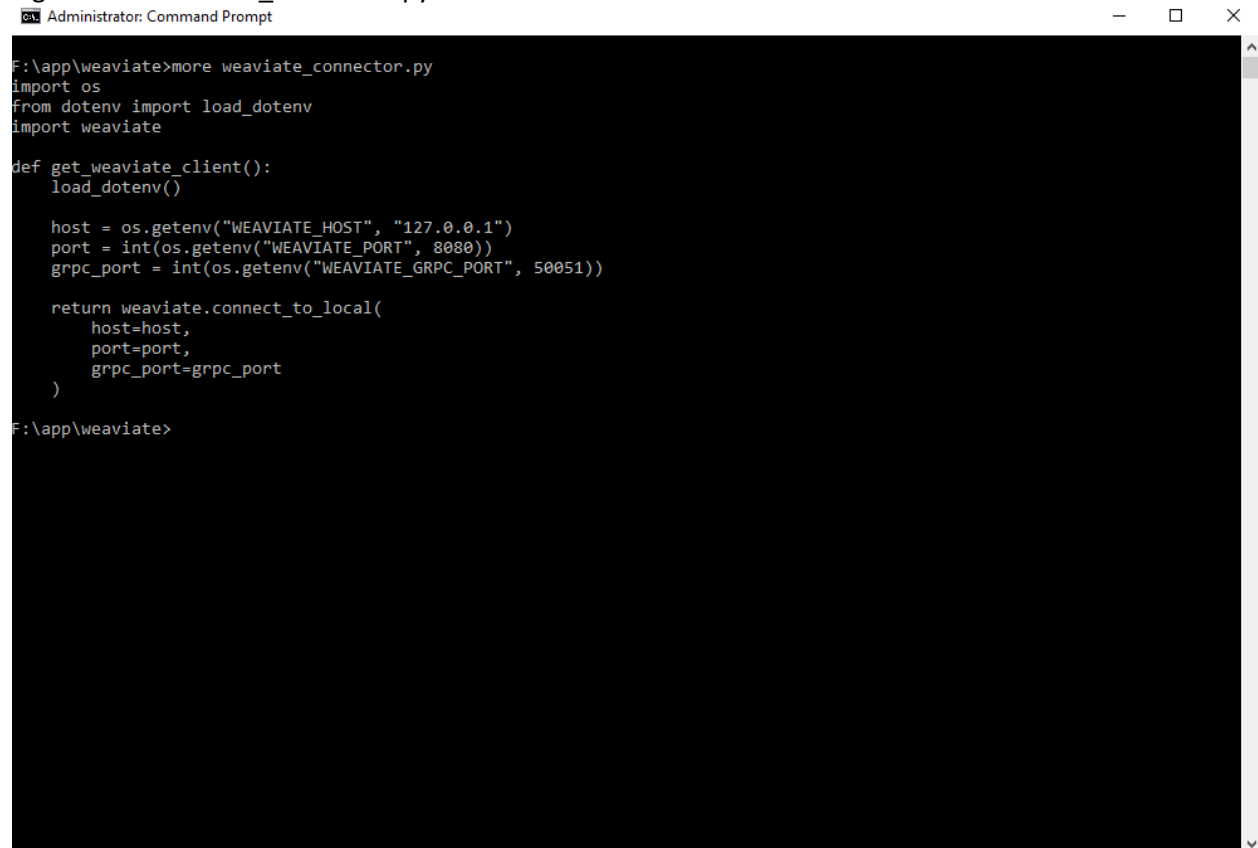
Run this command to install dotenv module that reads .env file

pip install dotenv

Create the connection wrapper as shown in Figure 12 below. Name the wrapper as
“**weaver_connector.py**”

2025 Weaviate

Figure 12 – weaviate_connector.py



```
Administrator: Command Prompt
F:\app\weaviate>more weaviate_connector.py
import os
from dotenv import load_dotenv
import weaviate

def get_weaviate_client():
    load_dotenv()

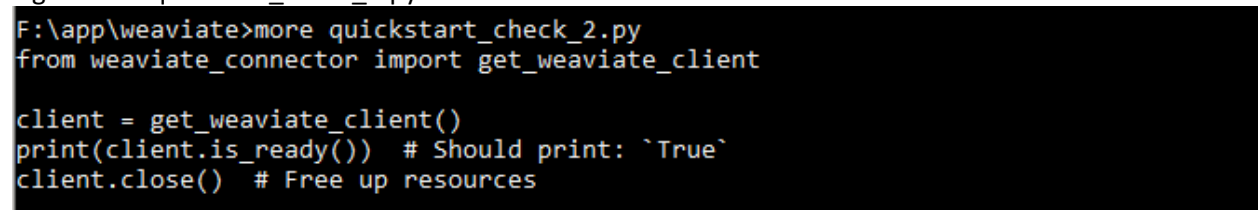
    host = os.getenv("WEAVIATE_HOST", "127.0.0.1")
    port = int(os.getenv("WEAVIATE_PORT", 8080))
    grpc_port = int(os.getenv("WEAVIATE_GRPC_PORT", 50051))

    return weaviate.connect_to_local(
        host=host,
        port=port,
        grpc_port=grpc_port
    )

F:\app\weaviate>
```

Create the connection check script called **quickstart_check_2.py** as shown in Figure 13 below

Figure 13 – quickstart_check_2.py



```
F:\app\weaviate>more quickstart_check_2.py
from weaviate_connector import get_weaviate_client

client = get_weaviate_client()
print(client.is_ready()) # Should print: `True`
client.close() # Free up resources
```

You may give the script a different name to your liking

Run the script to check if the connection wrapper works

py quickstart_check_2.py

The expected output is “True”, indicating the connection wrapper works, as shown in Figure 14 below.

Figure 14 – Connection check output

```
F:\app\weaviate>py quickstart_check_2.py
True

F:\app\weaviate>
```

Injecting Data Using Client Connection Wrapper

Create a script **quickstart_create_collection.py** as shown in Figure 15 below

Figure 15 - quickstart_create_collection.py

```
F:\app\weaviate>more quickstart_create_collection.py
import weaviate
from weaviate.classes.config import Configure
from weaviate.connector import get_weaviate_client

client = get_weaviate_client()
# client = weaviate.connect_to_local()

questions = client.collections.create(
    name="Question",
    vector_config=Configure.Vectors.text2vec_ollama(      # Configure the Ollama embedding integration
        api_endpoint="http://host.docker.internal:11434",  # Allow Weaviate from within a Docker container to contact
        your Ollama instance
        model="nomic-embed-text",                        # The model to use
    ),
    generative_config=Configure.Generative.ollama(      # Configure the Ollama generative integration
        api_endpoint="http://host.docker.internal:11434",  # Allow Weaviate from within a Docker container to contact
        your Ollama instance
        model="llama3.2",                                # The model to use
    )
)

client.close() # Free up resources
F:\app\weaviate>
```

This script creates the vector space in the database using two models, one is **nomic-embed-text**; the other, **llama3.2**. The vector space name is “**Question**”. In Python, the data type of the vector space is **collections**.

Run the script **quickstart_create_collection.py** to create the vector space.

py quickstart_create_collection.py

Next, create the script **quickstart_import.py** to inject the raw dataset into the vector database. See the script in Figure 15 below.

Figure 15 - quickstart_import.py

```
F:\app\weaviate>more quickstart_import.py
import weaviate
import requests, json

# client = weaviate.connect_to_local()
from weaviate_connector import get_weaviate_client
client = get_weaviate_client()

resp = requests.get(
    "https://raw.githubusercontent.com/weaviate-tutorials/quickstart/main/data/jeopardy_tiny.json"
)
data = json.loads(resp.text)

questions = client.collections.get("Question")

with questions.batch.fixed_size(batch_size=200) as batch:
    for d in data:
        batch.add_object(
            {
                "answer": d["Answer"],
                "question": d["Question"],
                "category": d["Category"],
            }
        )
        if batch.number_errors > 10:
            print("Batch import stopped due to excessive errors.")
            break

failed_objects = questions.batch.failed_objects
if failed_objects:
    print(f"Number of failed imports: {len(failed_objects)}")
    print(f"First failed object: {failed_objects[0]}")

client.close() # Free up resources
F:\app\weaviate>
```

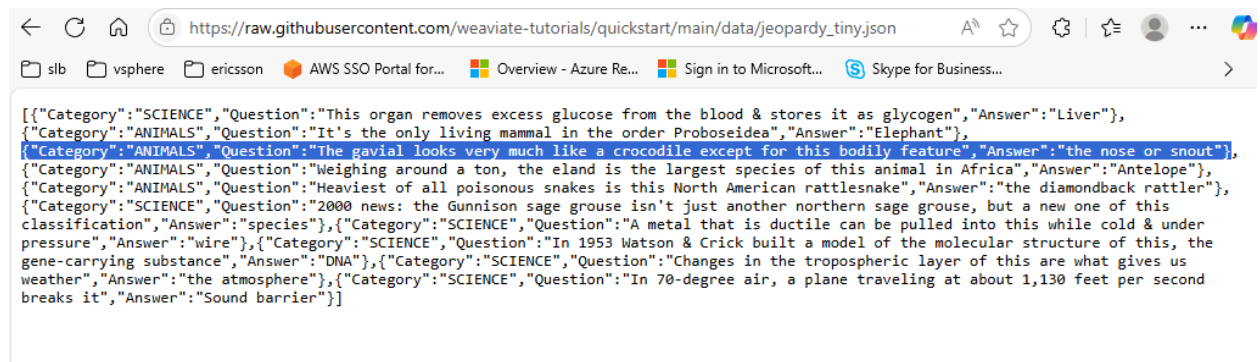
Run the script to load the dataset into the vector space.

py quickstart_import.py

The raw data set is a JSON file. It contains 10 records. Each record has three elements. Each element is a name:value pair. The first element name is "Category", the second, "Question", and the third, "Answer".

See Figure 16 below

Figure 16 – The raw dataset



The screenshot shows a web browser window with the URL https://raw.githubusercontent.com/weaviate-tutorials/quickstart/main/data/jeopardy_tiny.json. The browser displays the raw JSON content of the dataset. The JSON is an array of objects, each containing three fields: "Category", "Question", and "Answer". The first object is: `{ "Category": "SCIENCE", "Question": "This organ removes excess glucose from the blood & stores it as glycogen", "Answer": "Liver" },` and the last object is: `{ "Category": "SCIENCE", "Question": "In 70-degree air, a plane traveling at about 1,130 feet per second breaks it", "Answer": "Sound barrier" }`.

2025 Weaviate

When running, the script reads the dataset, loops through it, and adds each element pair in each record as an object into the vector database.

For clarity, see Figure 17 below the raw data in Excel format

Figure 17 – The raw data set in Excel format

Element 1 - Category	Element 2 - Question	Element 3 - Answer
Category:SCIENCE	Question:This organ removes excess glucose from the blood & stores it as glycogen	Answer:Liver
Category:ANIMALS	Question:It's the only living mammal in the order Proboscidea	Answer:Elephant
Category:ANIMALS	Question:The gavia looks very much like a crocodile except for this bodily feature	Answer:the nose or snout
Category:ANIMALS	Question:Weighing around a ton, the eland is the largest species of this animal in Africa	Answer:Antelope
Category:ANIMALS	Question:Heaviest of all poisonous snakes is this North American rattlesnake	Answer:the diamondback rattler
Category:SCIENCE	Question:2000 news: the Gunnison sage grouse isn't just another northern sage grouse, but a new one of this classification	Answer:species
Category:SCIENCE	Question:A metal that is ductile can be pulled into this while cold & under pressure	Answer:wire
Category:SCIENCE	Question:In 1953 Watson & Crick built a model of the molecular structure of this, the gene-carrying substance	Answer:DNA
Category:SCIENCE	Question:Changes in the tropospheric layer of this are what gives us weather	Answer:the atmosphere
Category:SCIENCE	Question:In 70-degree air, a plane traveling at about 1,130 feet per second breaks it	Answer:Sound barrier

Now that we see what's in the raw dataset and how the script works when loading the raw dataset into the vector database, let's proceed to making a couple of query scripts and see what those scripts will produce.

Run Semantic Query Using Client Connection Wrapper

Create a script `quickstart_neartext_query.py` as shown in Figure 18 below

Figure 18 - `quickstart_neartext_query.py`

```
F:\app\weaviate>more quickstart_neartext_query.py
Cannot access file F:\app\weaviate\quickstart_neartext_query.py

F:\app\weaviate>more quickstart_neartext_query.py
import weaviate
import json
from weaviate_connector import get_weaviate_client

client = get_weaviate_client()
# client = weaviate.connect_to_local()

questions = client.collections.get("Question")

response = questions.query.near_text(
    query="biology",
    limit=2
)

for obj in response.objects:
    print(json.dumps(obj.properties, indent=2))

client.close() # Free up resources
F:\app\weaviate>
```

The script will conduct a semantic search by querying with the word “biology” in the vector space

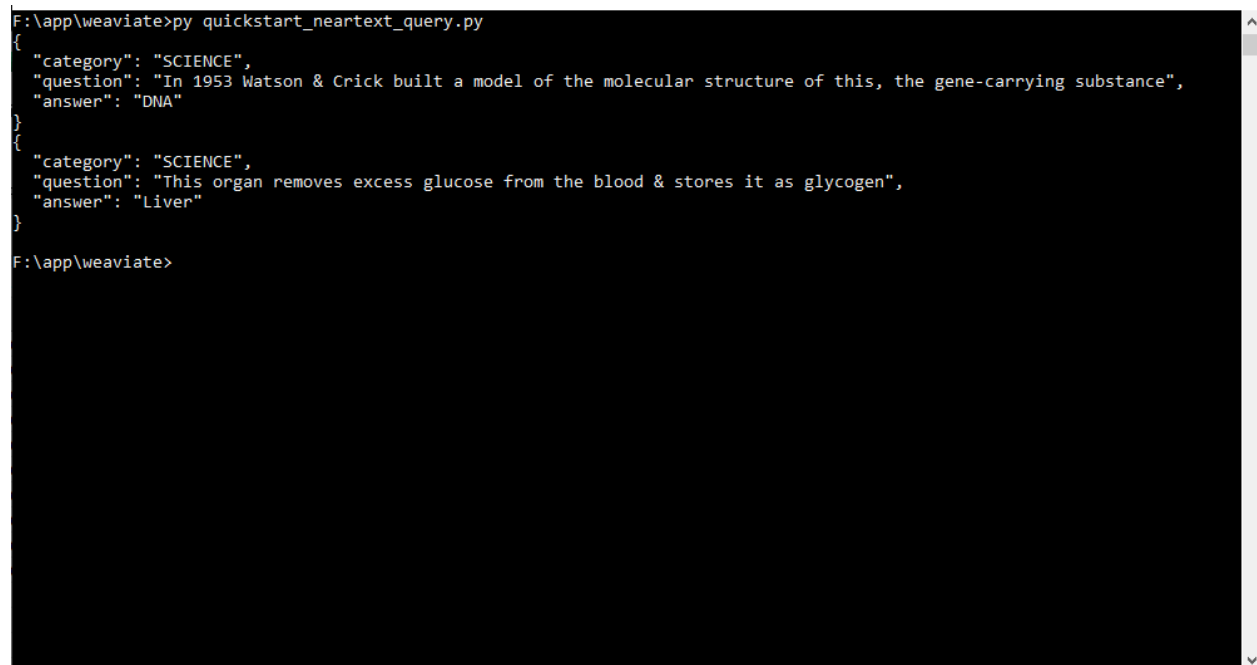
Question that contains the imported raw dataset as we saw in the previous section. We know the word “biology” does not exist in the dataset. We expect the database to respond by finding text similar in meaning of “biology”. Hence, this search is called semantic search, not a keyword search by “biology”, which is hardcoded in the script.

Run the following command to conduct the semantic search

py quickstart_neartext_query.py

See the response in Figure 19 below

Figure 19 – Semantic search response by the vector database



```
F:\app\weaviate>py quickstart_neartext_query.py
[
  {
    "category": "SCIENCE",
    "question": "In 1953 Watson & Crick built a model of the molecular structure of this, the gene-carrying substance",
    "answer": "DNA"
  },
  {
    "category": "SCIENCE",
    "question": "This organ removes excess glucose from the blood & stores it as glycogen",
    "answer": "Liver"
  }
]
F:\app\weaviate>
```

The response includes two texts, one is “DNA”, the other, “Liver”, both are of **biology**. The script is hardcoded to provide two texts in its response.

We can revise the script by removing the hardcoded query and limit. The revised script accepts the query subject word and the limit as command line arguments, thereby allowing us to play with the database using the script much more freely than otherwise. Please see the revised script **quickstart_neartext_query_2.py** in Figure 20 below.

Figure 20 quickstart_neartext_query_2.py

```
F:\app\weaviate>more quickstart_neartext_query_2.py
# quickstart_neartext_query_2.py
import weaviate
import json
import argparse
import string
from weaviate_connector import get_weaviate_client

# Accept query and limit values as arguments from command line
parser = argparse.ArgumentParser(description="Pass an integer value for limit from the command line")
parser.add_argument("--query", type=str, default="Biology", required=False, help="A text value for the query")
parser.add_argument("--limit", type=int, default=1, required=False, help="An integer value for the limit")

args = parser.parse_args()
query=args.query
limit=args.limit

print(f"The response query is set to {query}")
print(f"The response limit is set to {limit}")

client = get_weaviate_client()
questions = client.collections.get("Question")

response = questions.query.near_text(
    query=args.query,
    limit=args.limit
)

for obj in response.objects:
    print(json.dumps(obj.properties, indent=2))

client.close() # Free up resources
F:\app\weaviate>
```

Let's play with the revised script **quickstart_neartext_query_2.py** by giving different queries and limits and see what the vector database will come up with.

Query 1

```
F:\app\weaviate>py quickstart_neartext_query_2.py --query "Medicine" --limit 2
```

The response query is set to **Medicine**

The response limit is set to 2

```
{
  "category": "SCIENCE",
  "question": "This organ removes excess glucose from the blood & stores it as glycogen",
  "answer": "Liver"
}
{
  "category": "SCIENCE",
```

2025 Weaviate

```
"question": "In 1953 Watson & Crick built a model of the molecular structure of this, the gene-carrying substance",
```

```
"answer": "DNA"
```

```
}
```

Query 2

```
F:\app\weaviate>py quickstart_neartext_query_2.py --query "Iron" --limit 2
```

The response query is set to **Iron**

The response limit is set to 2

```
{
```

```
"category": "SCIENCE",
```

```
"question": "A metal that is ductile can be pulled into this while cold & under pressure",
```

```
"answer": "wire"
```

```
}
```

```
{
```

```
"category": "ANIMALS",
```

```
"question": "Weighing around a ton, the eland is the largest species of this animal in Africa",
```

```
"answer": "Antelope"
```

```
}
```

Query 3

```
F:\app\weaviate>py quickstart_neartext_query_2.py --query "Truck" --limit 2
```

The response query is set to **Truck**

The response limit is set to 2

```
{
```

```
"category": "ANIMALS",
```

```
"question": "Weighing around a ton, the eland is the largest species of this animal in Africa",
```

```
"answer": "Antelope"
```

```
}  
  
{  
  "category": "ANIMALS",  
  "question": "Heaviest of all poisonous snakes is this North American rattlesnake",  
  "answer": "the diamondback rattler"  
}
```

Query 4

```
F:\app\weaviate>py quickstart_neartext_query_2.py --query "Car" --limit 2
```

The response query is set to **Car**

The response limit is set to 2

```
{  
  "category": "ANIMALS",  
  "question": "The gavial looks very much like a crocodile except for this bodily feature",  
  "answer": "the nose or snout"  
}
```

```
{  
  "category": "SCIENCE",  
  "question": "In 70-degree air, a plane traveling at about 1,130 feet per second breaks it",  
  "answer": "Sound barrier"  
}
```

Query 5

```
F:\app\weaviate>py quickstart_neartext_query_2.py --query "Rain" --limit 1
```

The response query is set to **Rain**

The response limit is set to 1

```
{
```

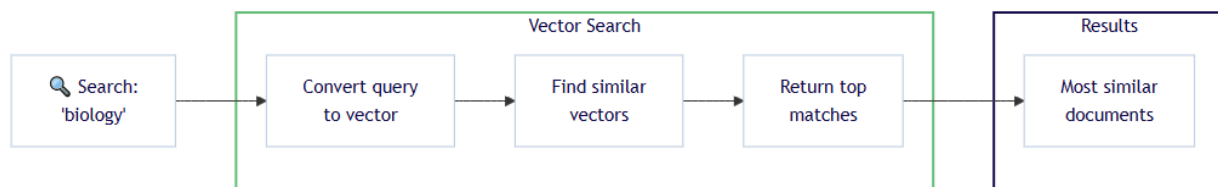
2025 Weaviate

```
"category": "SCIENCE",  
"question": "Changes in the tropospheric layer of this are what gives us weather",  
"answer": "the atmosphere"  
}
```

As you saw, we submitted 5 queries above. In each query, the query subject word was provided. None of the query subject words exist in the dataset imported into the vector database and yet the database consistently responded with texts closest in meaning to the queried subject word. Hence, the queries were all semantic searches, and not keyword searches.

Figure 21 below shows the workflow of semantic search in Weaviate

Figure 21 – Weaviate semantic search workflow



Source: [Weaviate Get Started](#)

Run RAG Using Client Connection Wrapper

In this section, we will make a different script to run **Retrieval Augmented Generation (RAG)** queries.

Please see Figure 21 below for the script `quickstart_rag_2.py`

Figure 21 – `quickstart_rag_2.py`

```
F:\app\weaviate>more quickstart_rag_2.py
# quickstart_rag_2.py
import weaviate
import json
import argparse
import string
from weaviate_connector import get_weaviate_client

# Accept query, limit, and task values as arguments from command line
parser = argparse.ArgumentParser(description="Pass an integer value for limit from the command line")
parser.add_argument("--query", type=str, default="Biology", required=False, help="A text value for the query")
parser.add_argument("--limit", type=int, default=1, required=False, help="An integer value for the limit")
parser.add_argument("--task", type=str, default="Write a tweet with emojis about these facts.", required=False, help="A text value for the query")

args = parser.parse_args()
query=args.query
limit=args.limit
grouped_task=args.task

print(f"The response query is set to {query}")
print(f"The response limit is set to {limit}")
print(f"The response task is set to {grouped_task}")

client = get_weaviate_client()

questions = client.collections.get("Question")

response = questions.generate.near_text(
    query=args.query,
    limit=args.limit,
    grouped_task=args.task
)

print(response.generated) # Inspect the generated text

client.close() # Free up resources

F:\app\weaviate>
```

Now let's run the script

RAG Query 1

```
F:\app\weaviate>py quickstart_rag_2.py
```

The response query is set to **Biology**

The response limit is set to **1**

The response task is set to **Write a tweet with emojis about these facts.**

"Did you know? 🤔 In 1953, James Watson & Francis Crick built a model of the DNA molecule 💡! The gene-carrying substance that holds our genetic code is one fascinating piece of science! 📖 #DNA #Genetics #ScienceHistory"

RAG Query 2

```
F:\app\weaviate>py quickstart_rag_2.py --query "Rain" --limit 2
```

The response query is set to **Rain**

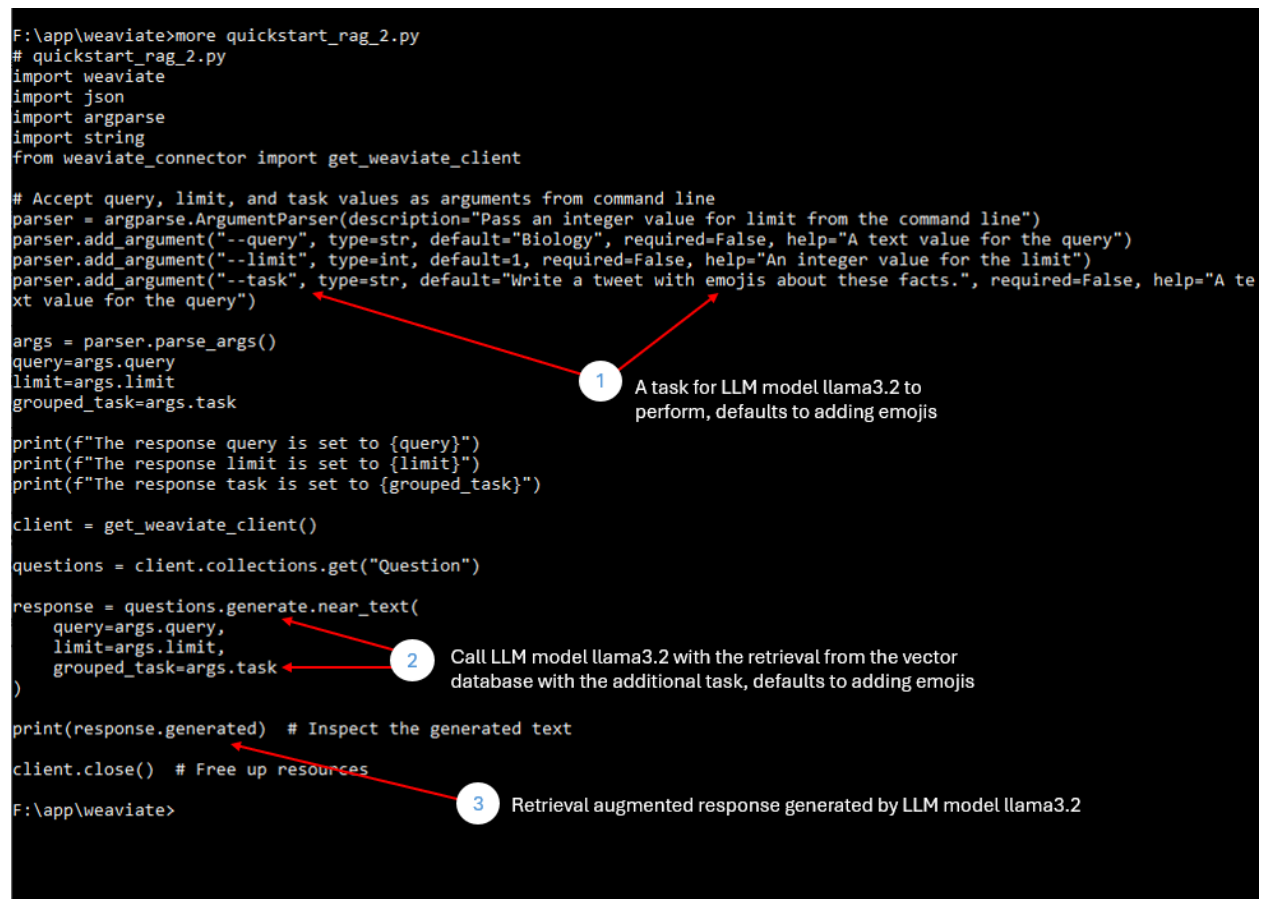
The response limit is set to **2**

The response task is set to **Write a tweet with emojis about these facts.**

"🧠 Did you know? 😬 Changes in the tropospheric layer of the atmosphere give us WEATHER 🌧️! And if you thought that was fast... a plane can break the sound barrier in just 70-degree air, reaching speeds of 1,130 feet per second! 🚀 #Weather #Science"

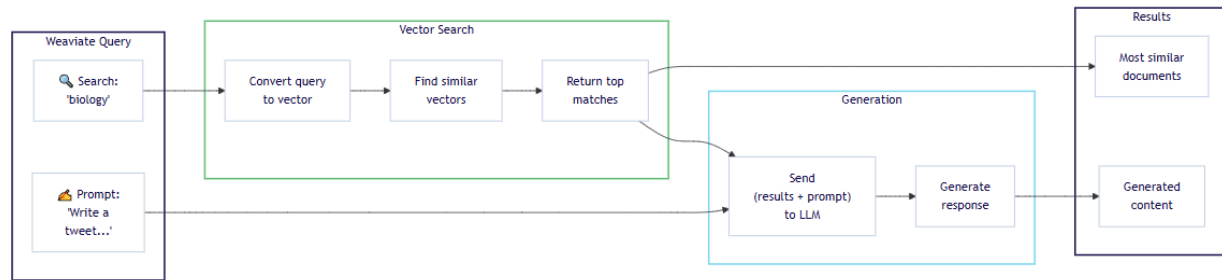
We see something new – emojis were added to the responses. Where did those emojis come from? They were generated by **LLM model llama3.2**. See Figure 22 below that explains how RAG works in the script **quickstart_rag_2.py**.

Figure 22 – How RAG works in **quickstart_rag_2.py** script



See Figure 23 below that shows the RAG workflow in Weaviate.

Figure 23 – Weaviate RAG Workflow



Source: [Weaviate Get Started](#)

Summary

This short article touched upon Weaviate, a popular Opensource vector database. Vector databases are primarily used in RAG systems. A RAG system uses LLMs in two ways. First, when LLMs process user's internal text, not found on the Internet, they convert it into dense vectors that capture semantic meaning. Those vectors are stored in vector databases. Vector databases let user retrieve documents or text chunks that are closest in meaning to the user's query. Second, the retrieval is augmented to the user's query and is sent to LLMs. Based on such a retrieval augmented query or prompt, LLMs generate response back to the user. RAG allows LLMs to work better for users because their responses are based on not only LLMs' own trained data but also users' own data. In the lab, Weaviate was downloaded, installed, and ran as a Docker container locally on a PC. The Weaviate vector database was loaded with the Weaviate provided sample dataset that represents user's internal dataset. When the dataset was imported, it was processed by llama3.2, the LLM model selected to use in the lab, into dense vectors for semantic search. In the lab, a few Python scripts were used, based on the Weaviate provided scripts, revised to add capabilities, such as using a custom port instead of the default port for Weaviate database and Weaviate client communication, and allowing the query scripts flexibility to accept various query search words and number of text chunks to be returned as command line arguments.

The lab was intended for AI beginners to deepen their understanding of semantic search and RAG.

Revision History

Created on July 22, 2025

Revised on July 25, 2025