

# RAG proof of concept

Requirements:

The two python files

Ubuntu 22.04 WSL via Windows\*

Python 3.12 installed in a virtual environment (venv in this case) with pip.

the following python modules (via pip):

- chromadb
- sentence\_transformers
- langchain
- pymupdf
- flask

Ollama + an LLM model (any usual quantized gguf like mistral, deepseek, etc)

OpenWebUI (this is only being used as the front-end, you can use the terminal to query without OpenWebUI)

\*this is done on my personal computer. I have no idea about IT and allowing WSL, Hyper-V, etc, on work computers

The screenshot shows a Windows PowerShell terminal window with WSL installed. The user lists installed WSL instances, including 'openwebui' which is running. They then launch the 'openwebui' instance using 'wsl -d openwebui'. Inside the WSL environment, they navigate to the '~/rag' directory and list its contents, showing files like 'rag.py', 'rag\_api.py', and 'rag\_chroma'. A callout box explains that 'py.rag' creates the chromadb database and acts as an Ollama interpreter. Another callout points to the 'wsl -d openwebui' command, stating it launches the specific instance of the openwebui WSL. A third callout points to the directory listing, noting that the '~/rag' folder contains 'rag.py' and the '~/rag/data' folder contains PDFs.

The screenshot shows the Sublime Text editor with the 'rag.py' file open. The code defines a RAG system using chromadb, sentence transformers, and langchain. Annotations explain the role of each component: 'chromadb' is used to create a database from PDFs; 'sentence transformers' are used for embeddings; 'langchain' is used by chromadb to parse data; and 'pymupdf' is used to process PDFs without manual text extraction. The code includes configuration for the document folder, chroma directory, collection name, model name, chunk size, and embedding model.

```
will@will-desktop:~/rag$ cd data
will@will-desktop:~/rag/data$ ls
a.pdf b.pdf c.pdf d.pdf
```

the PDFs of the ASHRAE handbooks. Renamed to make moving them around in the command line easier.

VENV

running the rag.py script parses the data from all the pdfs in the ~/rag/data folder

```
(rag-env) will@will-desktop:~/rag$ rag.py
rag.py: command not found
(rag-env) will@will-desktop:~/rag$ python rag.py
[1] Initializing embedder and ChromaDB...
  → Existing chunks in DB: 0
[2] Scanning folder: data

→ Processing: a.pdf
→ Processing: c.pdf
→ Processing: d.pdf
→ Processing: b.pdf

✓ Ingestion complete. New chunks added: 56165
[3] Ready for questions.
```

the script also allows you to query directly via Ollama without OpenWebUI.

Ask a question (or 'exit'): what is ethylene glycol?

→ Asking Mistral via Ollama...

Ethylene glycol is a colorless, practically odorless liquid that is miscible with water and many organic compounds. It has the ability to efficiently lower the freezing point of water, and it is often used in heat transfer fluids due to its low volatility and relatively low corrosivity when properly inhibited. Ethylene glycol can be toxic and is typically not suitable for applications involving possible human contact or where there is a risk of incidental food contamination. It is commonly available as premixed aqueous solutions from vendors.

Ask a question (or 'exit'):

```
98 prompt = f"""Use the following context to answer the question.\n\nContext:\n{context}\n\nQuestion: {query}\n\nAnswer:"""
99
100 print("\n→ Asking Mistral via Ollama...")
101 response = requests.post(
102     "http://localhost:11434/api/generate"
103     json={
104         "model": MODEL_NAME,
105         "prompt": prompt,
106         "stream": False
107     }
108 )
```

from rag.py

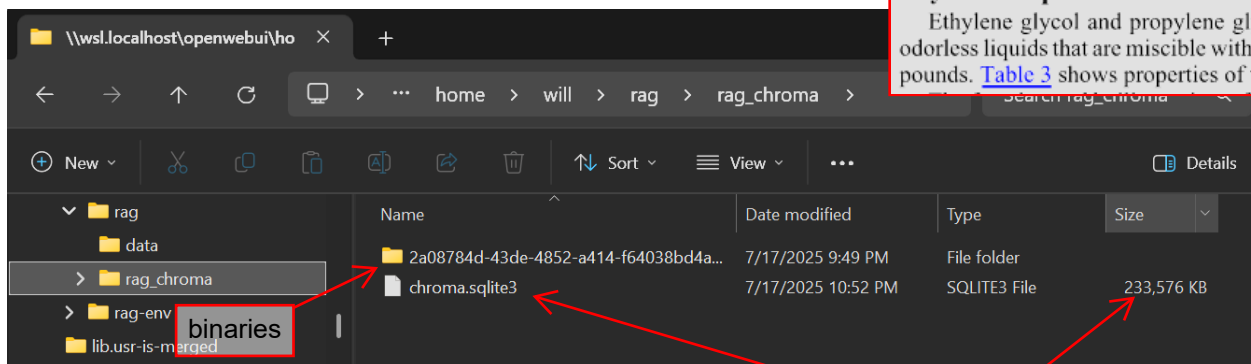
from the ASHRAE handbooks

## 2. INHIBITED GLYCOLS

Ethylene glycol and propylene glycol, when properly inhibited for corrosion control, are used as aqueous-freezing-point depressants (antifreeze) and heat transfer media. Their chief attributes are their ability to efficiently lower the freezing point of water, their low volatility, and their relatively low corrosivity when properly inhibited. Inhibited ethylene glycol solutions have better thermophysical properties than propylene glycol solutions, especially at lower temperatures. However, the less toxic propylene glycol is preferred for applications involving possible human contact or where mandated by regulations. If a heat transfer fluid may have incidental food contact, then it should be made from propylene glycol that meets U.S. Pharmacopeia (USP 2016) *Food Chemical Codex* (FCC) specifications. Avoid other, less pure grades of propylene glycol: they can contain toxic or unwanted impurities that also adversely affect performance characteristics (e.g., foaming propensity, corrosion).

### Physical Properties

Ethylene glycol and propylene glycol are colorless, practically odorless liquids that are miscible with water and many organic compounds. [Table 3](#) shows properties of the pure materials.



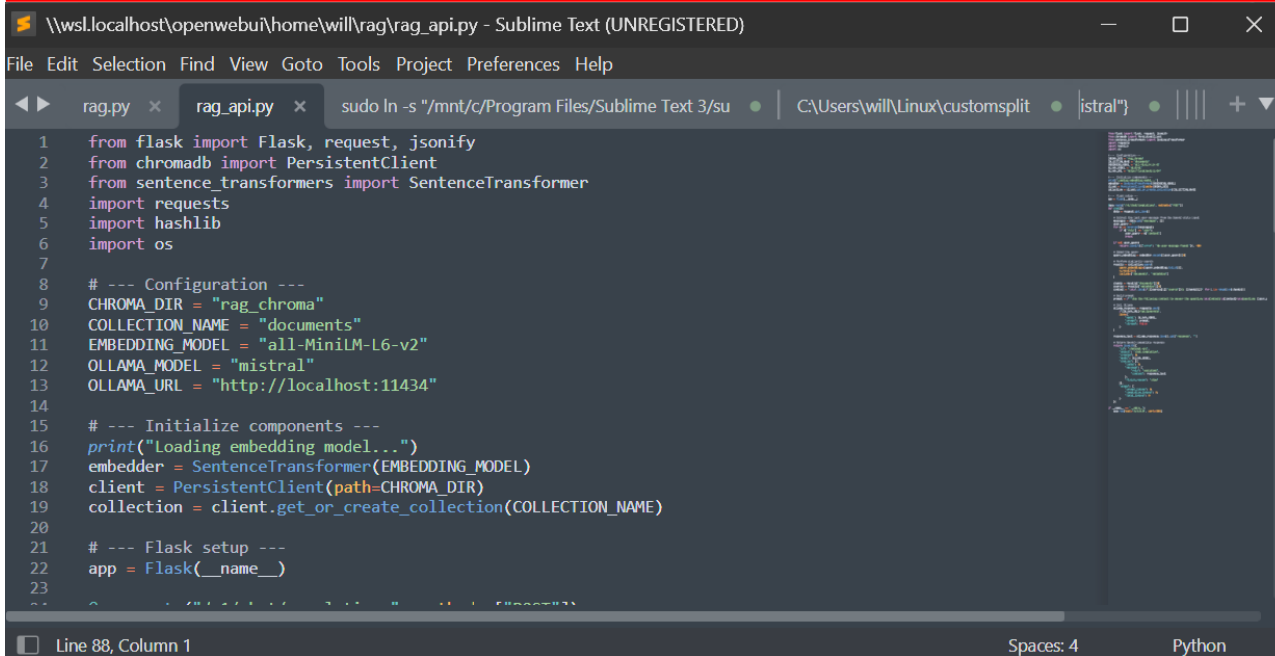
The rag.py generates a ~/rag/rag\_chroma folder with the SQL db and binaries. NOTE: The pdfs were reduced to around 50 total megabytes before processing (manually in bluebeam). The database is 6x that size with the binaries. It took approximately 1.5 hours to process on a GTX 5090 with M2 PCIe v5 NVMe and 128gb ram using PyTorch cu129. The shuffling of data between the solid state drive and gpu seemed like the bottleneck. VRAM never exceed ~4gb out of 32gb. The mistral model used is quantized to 4gb so that tracks.

# OpenWebUI as Front End

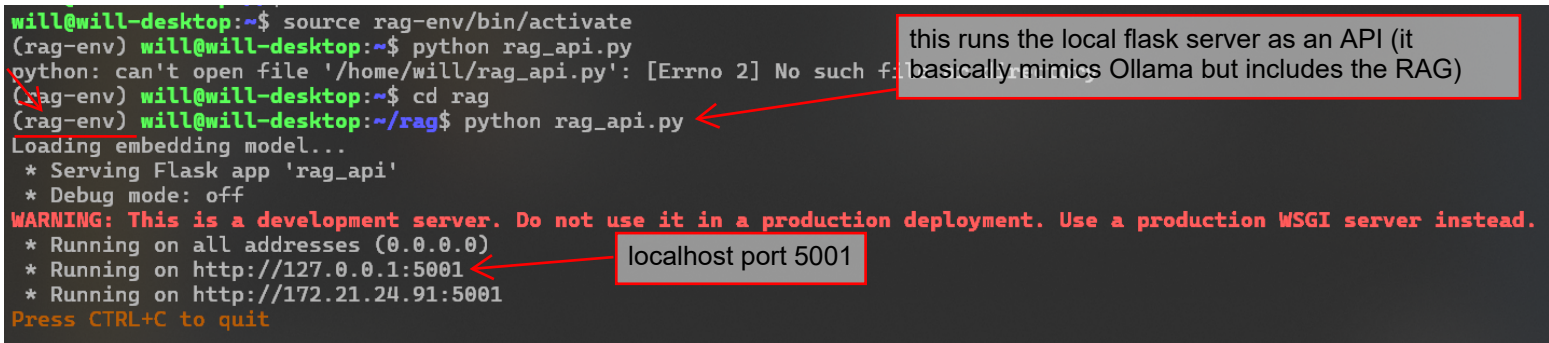
The rag\_api.py creates a local API that OpenWebUI ties to and queries instead of Ollama directly.

The flow is:

OpenWebUI queries the flask API which queries Ollama+RAG payload which uses RAG+Mistral.



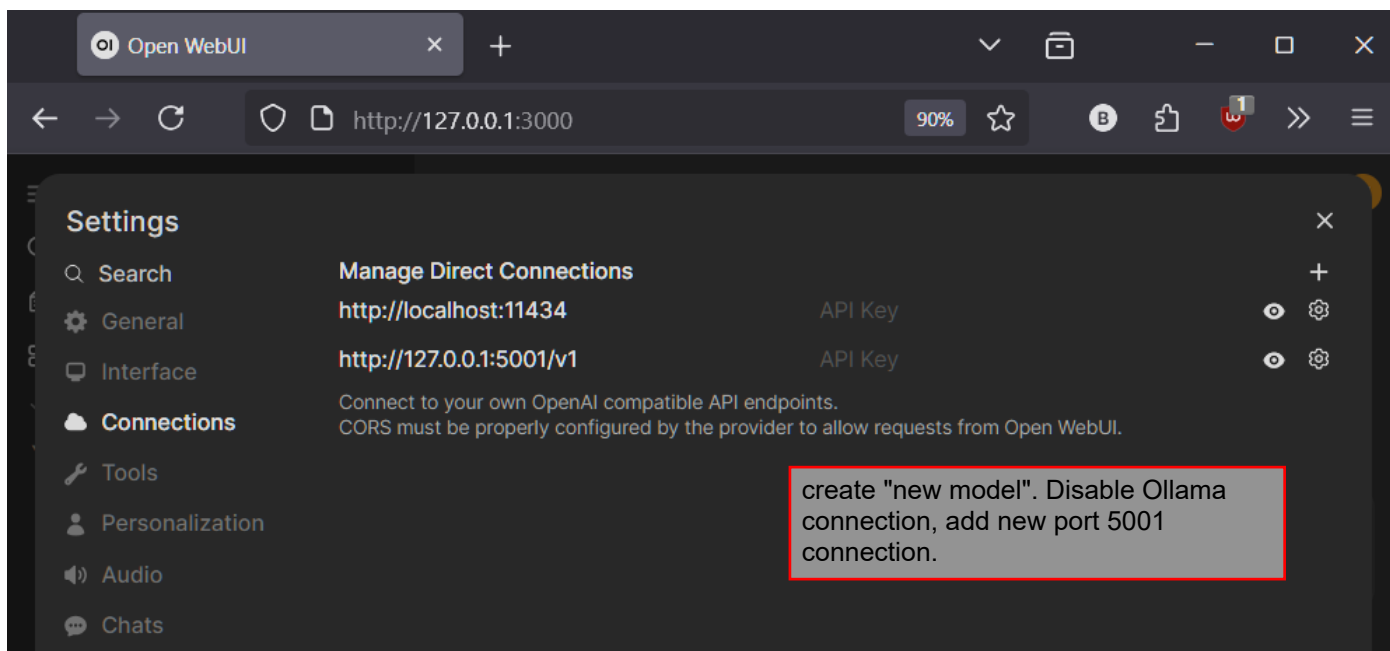
```
1 from flask import Flask, request, jsonify
2 from chromadb import PersistentClient
3 from sentence_transformers import SentenceTransformer
4 import requests
5 import hashlib
6 import os
7
8 # --- Configuration ---
9 CHROMA_DIR = "rag_chroma"
10 COLLECTION_NAME = "documents"
11 EMBEDDING_MODEL = "all-MiniLM-L6-v2"
12 OLLAMA_MODEL = "mistral"
13 OLLAMA_URL = "http://localhost:11434"
14
15 # --- Initialize components ---
16 print("Loading embedding model...")
17 embedder = SentenceTransformer(EMBEDDING_MODEL)
18 client = PersistentClient(path=CHROMA_DIR)
19 collection = client.get_or_create_collection(COLLECTION_NAME)
20
21 # --- Flask setup ---
22 app = Flask(__name__)
23
24 @app.route("/api/embed", methods=["POST"])
25 def embed():
26     data = request.get_json()
27     text = data.get("text")
28     if not text:
29         return jsonify({"error": "Text is required"}), 400
30     embedding = embedder.embed(text)
31     return jsonify({"embedding": embedding.tolist()}), 200
32
33 @app.route("/api/query", methods=["POST"])
34 def query():
35     data = request.get_json()
36     text = data.get("text")
37     if not text:
38         return jsonify({"error": "Text is required"}), 400
39     embedding = embedder.embed(text)
40     results = collection.query(query_embeddings=embedding.tolist())
41     relevant_docs = results.get("documents")
42     if not relevant_docs:
43         return jsonify({"error": "No relevant documents found"}), 404
44     relevant_docs = relevant_docs[0]
45     prompt = data.get("prompt")
46     if not prompt:
47         return jsonify({"error": "Prompt is required"}), 400
48     prompt = prompt + "\n\n" + relevant_docs
49     response = requests.post(
50         OLLAMA_URL,
51         json={
52             "model": OLLAMA_MODEL,
53             "prompt": prompt,
54             "stream": False,
55         },
56     )
57     if response.status_code != 200:
58         return jsonify({"error": "Ollama query failed"}), 500
59     response_json = response.json()
60     return jsonify({"response": response_json.get("response")}), 200
61
62 if __name__ == "__main__":
63     app.run(host="0.0.0.0", port=5001)
```

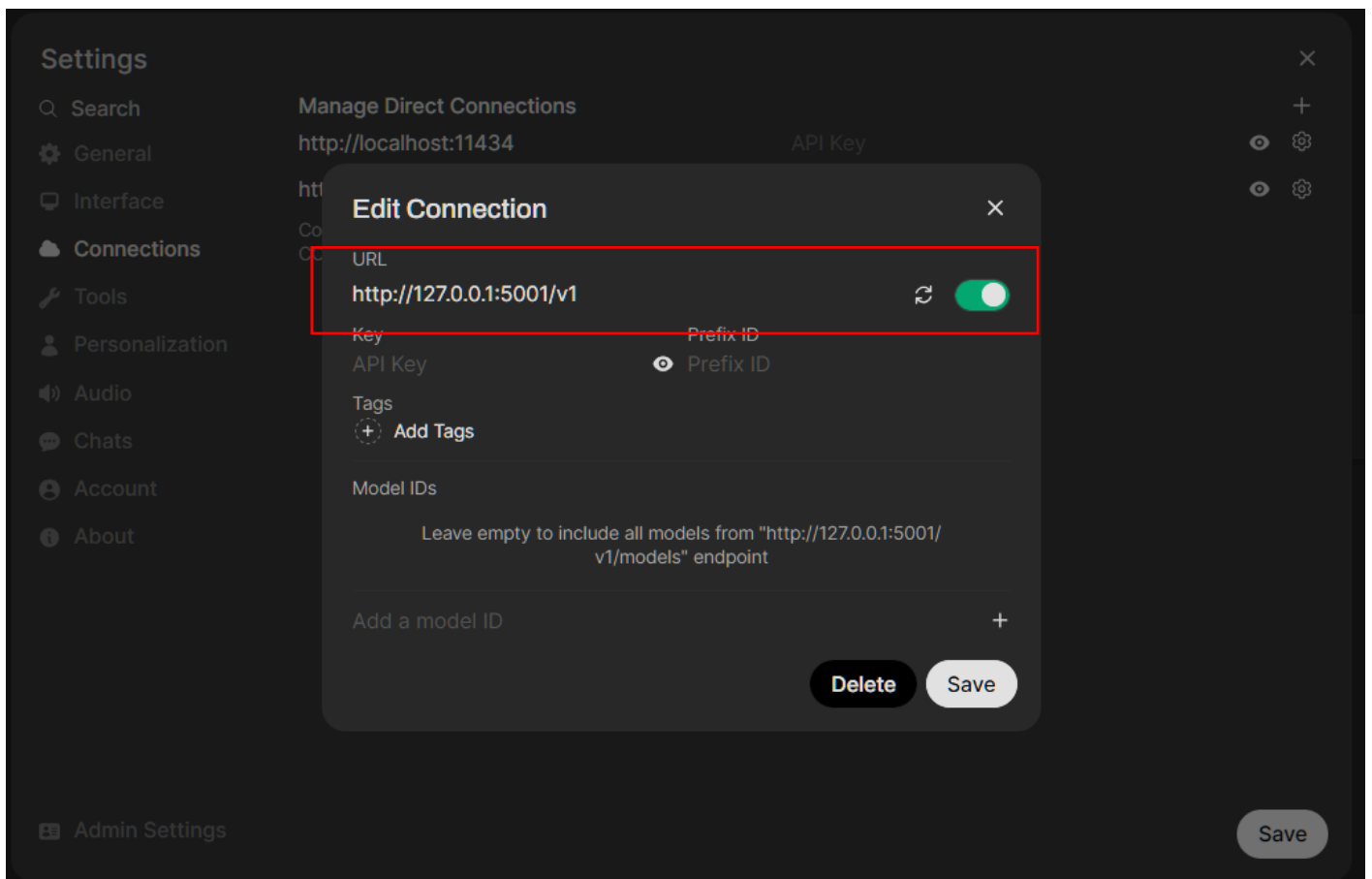
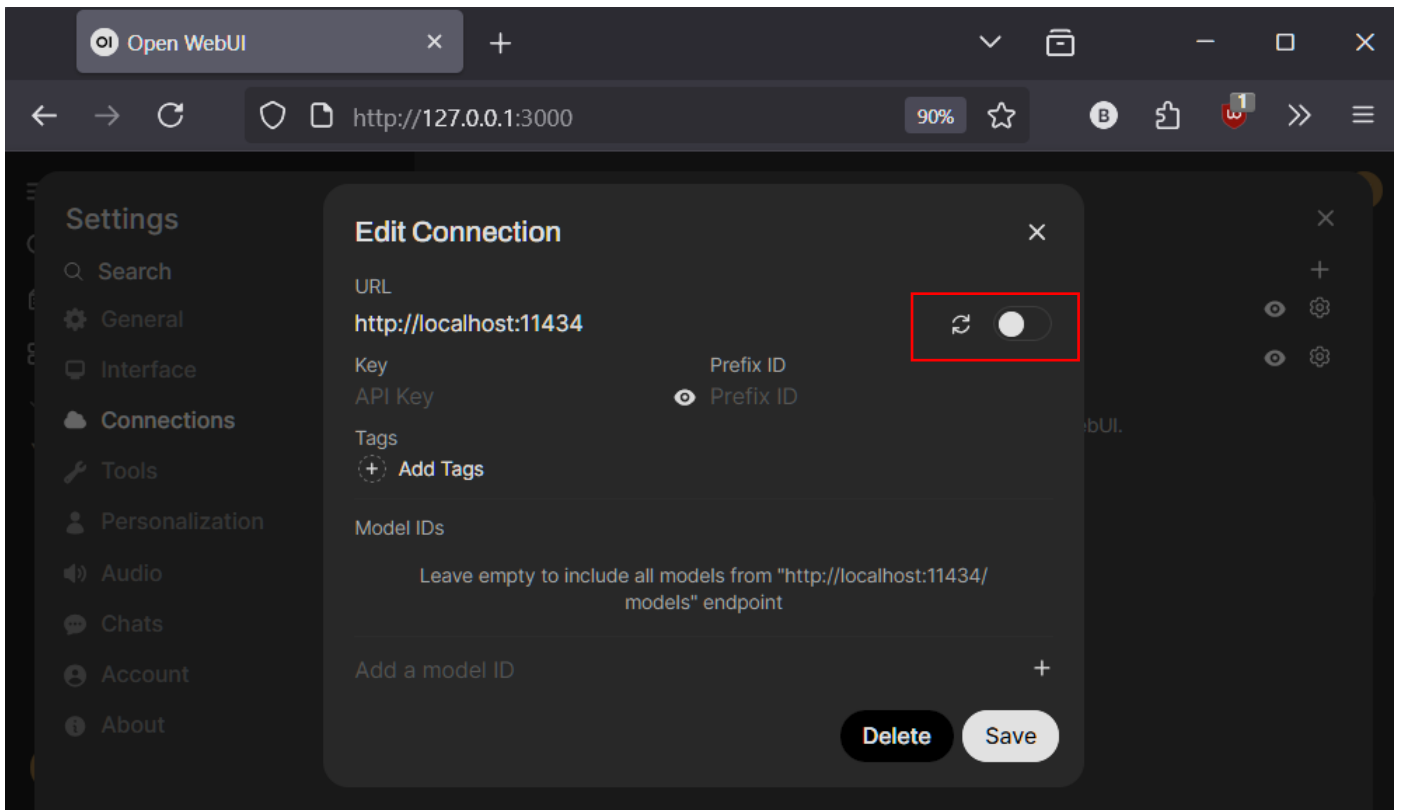


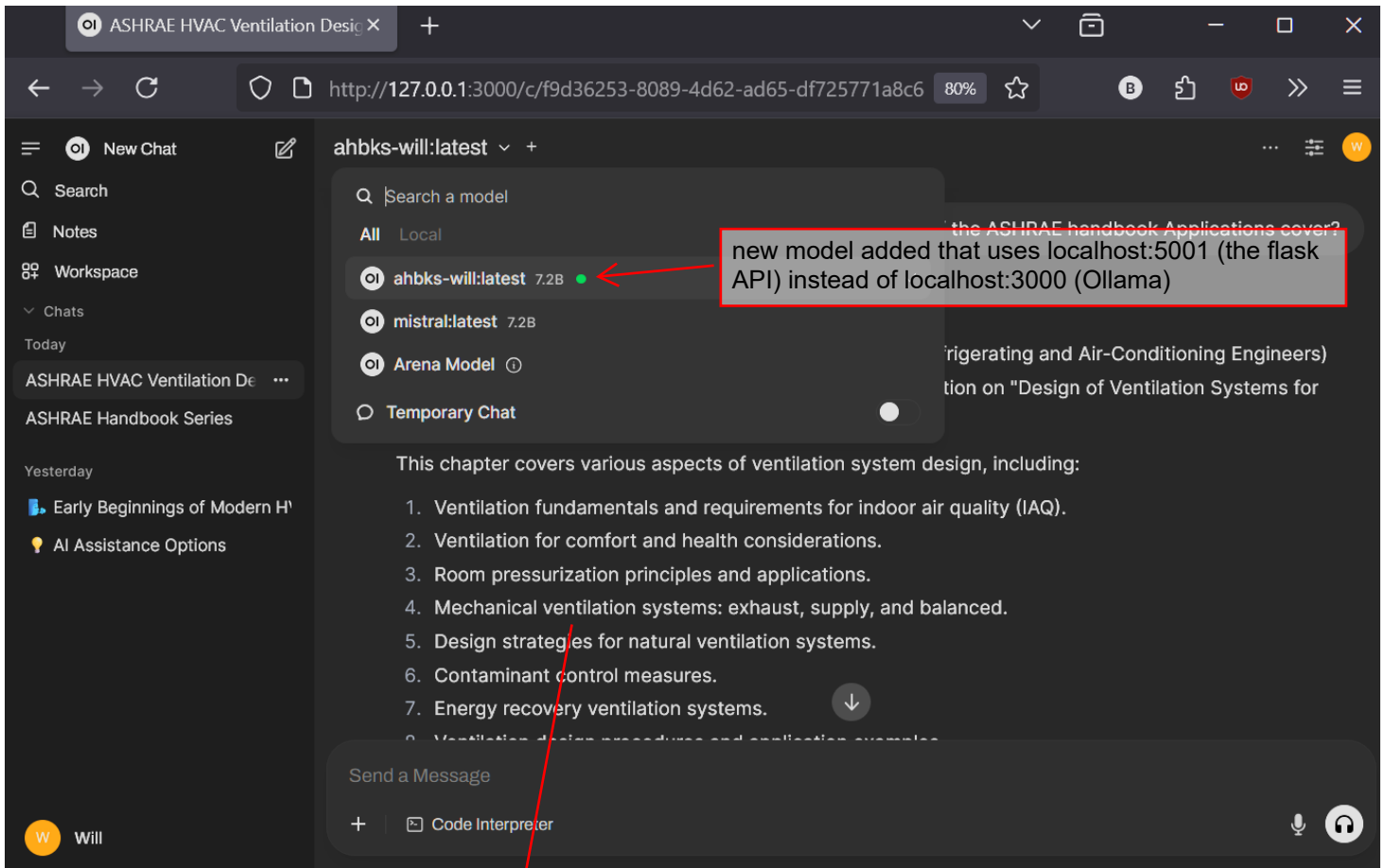
```
will@will-desktop:~$ source rag-env/bin/activate
(rag-env) will@will-desktop:~$ python rag_api.py
python: can't open file '/home/will/rag_api.py': [Errno 2] No such file or directory
(rag-env) will@will-desktop:~$ cd rag
(rag-env) will@will-desktop:~/rag$ python rag_api.py
Loading embedding model...
* Serving Flask app 'rag_api'
* Debug mode: off
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
* Running on all addresses (0.0.0.0)
* Running on http://127.0.0.1:5001
* Running on http://172.21.24.91:5001
Press CTRL+C to quit
```

this runs the local flask server as an API (it basically mimics Ollama but includes the RAG)

localhost port 5001







flask shows the urllib requests in real time when you query.

```
(rag-env) will@will-desktop:~/rag$ 127.0.0.1 - - [18/Jul/2025 13:29:36] "OPTIONS /v1/models HTTP/1.1" 404 -
127.0.0.1 - - [18/Jul/2025 13:29:37] "OPTIONS /v1/models HTTP/1.1" 404 -
127.0.0.1 - - [18/Jul/2025 13:29:37] "OPTIONS /v1/models HTTP/1.1" 404 -
(rag-env) will@will-desktop:~/rag$ 127.0.0.1 - - [18/Jul/2025 13:31:50] "OPTIONS /v1/models HTTP/1.1" 404 -
127.0.0.1 - - [18/Jul/2025 13:31:50] "OPTIONS /v1/models HTTP/1.1" 404 -
127.0.0.1 - - [18/Jul/2025 13:32:04] "OPTIONS /v1/models HTTP/1.1" 404 -
```

There are other tools than chromadb. This was mainly a proof of concept/learning exercise.