Run Tests (CI) passing

RAG Assistant for Industrial Audio

> **Project tagline:** *Ask natural-language questions about factory machine sounds.*

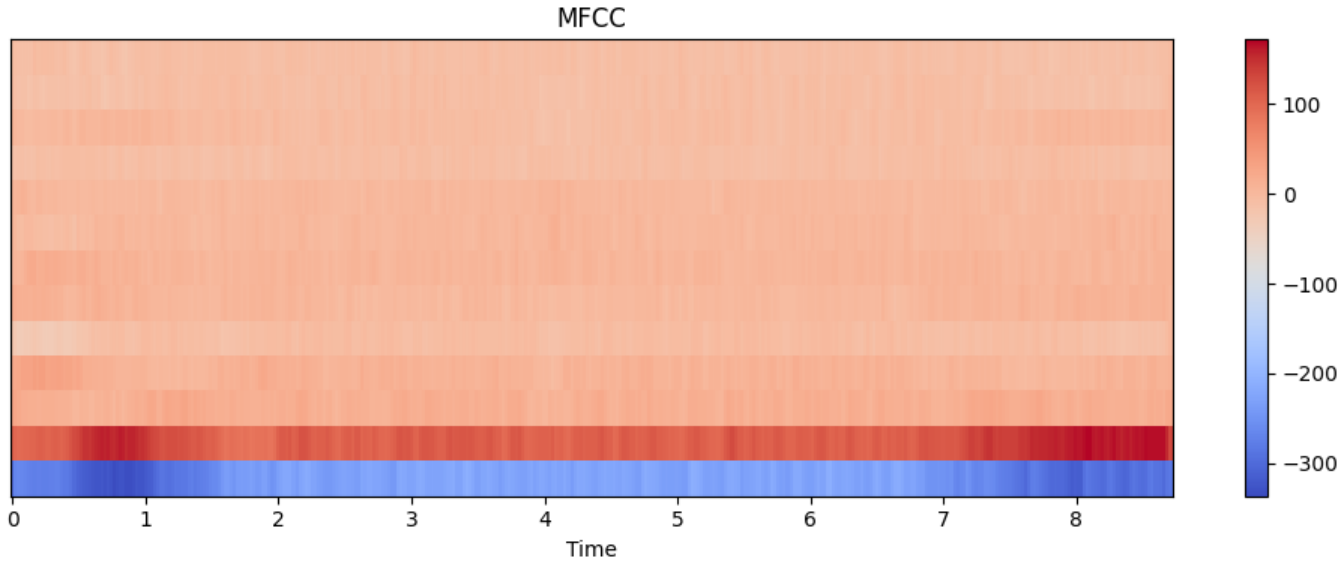
This walkthrough shows how to turn 2 GB of **DCASE 2024 Task-2** audio logs into an interactive Retrieval-Augmented-Generation (RAG) service powered by an open-source LLM and **Qdrant** vector search. Along the way we will some advanced signal processing, fast batch embedding techniques, and wrap the whole thing into a production-grade FastAPI backend, together with snapshot-based MLOps.

What is this about

When you are faced with a large dataset made of texts, LLMs and RAG techniques represent a clear choice of techniques. After all, LLMs are all about predicting what word comes next, after a given context. Industrial datasets are a whole different beast. They are rarely based on texts. For example, you might be faced with a bunch of sensor recordings. They could be wav files (coming from arrays of microphones) or accelerometer data. These continuous signals are actually not so far from texts: after all, once they enter the computer, the signals are discretized (think: streams of 0 and 1), so you could imagine that LLM might enter the scene and reason about these "texts" made of 0 and 1. In other words, you would work with LLMs directly on the raw signals. But you could do something different. By performing **numeric feature extraction** (RMS, FFT peaks), you could produce more meaningful streams of signals, and by combining them with a language model, you could directly query those raw sensor streams in plain English:

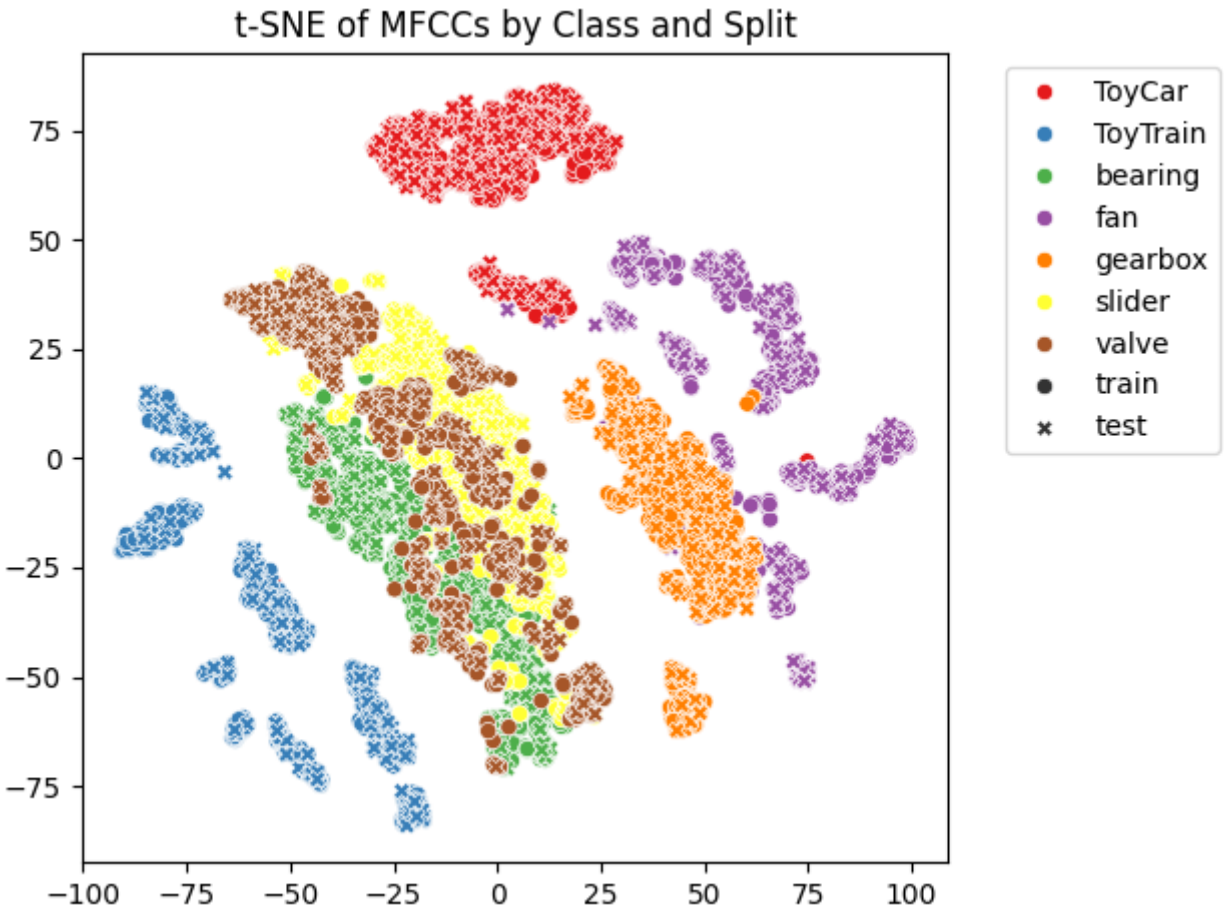
> *"Which anomalous bearing clips in section 00 had a dominant frequency above 900 Hz?"*

The features chosen here are quite simple. For the study of sounds, it is quite common to take a specialized version of spectrograms, called **mel-spectrogram**. Here is one such example:



Visualizing the entire dataset

If you want to have a global panorama of the entire dataset, you will have to make some choices. After all, visualizing means projecting to a flat 2d screen the entire dataset. One nice way to do that is to compute the mel-spectrograms for each sound snippet, obtain like this a set of large matrices (that you can view as vectors in a high dimensional space), and then project to the plane to visualize. The so-called **tsne** embedding is a common choice of such (nonlinear) projection. The result looks like this:



Architecture and flow

In this DCase dataset we have 2024 thousands of one-second WAV clips recorded from bearings, valves and other industrial machines. We want to make those audio clips instantly searchable, as if we had some kind of "Google for sounds" available to us. We want to be able to "find all files whose spectral stats & metadata resemble this noisy valve", without listening to them one by one each time we ask a question. The central piece is the script `dcase_indexer.py`: basically it ingests the entire folder full of WAVs, computes some light-weight audio features for each sound snippet, concatenates these features with filename metadata, embeds the result with a Sentence Transformer, and finally shelves the result nicely into a Qdrant collection.

Indexing the files

```
flowchart LR
        A[WAV files]-->|torch+numpy|B
        B[Feature Extractor RMS/FFT] --> C[SentenceTransformer embedder]
        C -->|vectors + JSON| D[Qdrant]
```

The audio features constitute like a simplified **fingerprint** for the audio clip. For the purpose of this project, we chose very simple ones, lightweight features, but one could easily imagine more refined choices.

Chosen feature	Why
RMS	overall loudness

Chosen feature	Why
Dominant freq (Hz)	main mechanical resonance
SNR (dB)	health proxy—faulty bearings are often noisy
Duration (s)	catches truncated files

Technically what is stored inside Qdrant looks like this:

```
{
  "id": "0d6fec7b-5a4d-4d87-9fd9-5c913a3c2d4f",
  "vector": [ -0.027, 0.154, ..., -0.041 ],          // 1 024 floats
  "payload": {
    "machine_type": "bearing",
    "section": "01",
    "domain": "source",
    "split": "train",
    "state": "normal",
    "clip_id": "000231",
    "rms": 0.018,
    "dominant_freq_hz": 49.8,
    "snr_db": 32.4,
    "duration_sec": 1.0,
    "file": "Data/Dcase/bearing/
…/bearing_01_source_train_normal_000231.wav"
  }
}
```

The **vector** part of this data corresponds to an embedding of the payload part. It gives us access to a kind of "fuzzy search" ("find sounds similar to this one"): points that are close to each other in the embedding space correspond to similar objects. The **payload** part allows some convenient filtering (eg "get all bearings") that the vector part could not offer. The two aspects complement each other.

Querying

Now let us say that the user wants to retrieve "bearing clip with loud 50 Hz hum". The query is normalized into a json `{"machine_type":"bearing","dominant_freq_hz":50,"rms":"high",...}` and then sent to the embedder.

```
flowchart LR
        E[User → /ask?q=…]--> F[Retriever Qdrant top k]
        F --> G[LLM Ollama]
        G --> H[FastAPI response]
```

Qdrant's search API returns the most relevant points. The LLM now receveives the user's original prompt together with the snippets (or feature tables) from the retrieved clips. It then returns its final answer. And

that concludes the oevrview of the entire pipeline!

- **Indexer script:** `dcase_indexer.py` (runs once; ~3 min on M1).
- **API service:** `rag_api.py` (<40 LOC).
- **Snapshots:** one command restores the full collection in seconds.

Quick-start and install

The quickstart instructions cover the situation where you run the pipeline for the first time. The indexing operations take quite a bit of time, so there are further instructions at the bottom of the page to re-use the snapshots created.

#	Command (from repo root)	What it does
1	`conda env create -f env.yml && conda activate ml_py310`	Creates + activates the Python 3.10 env
2	`bash scripts/get_dcase24.sh`	Downloads & unzips the DCASE-24 dev set (≈ 2 GB) into `Data/Dcase/`
3	`docker run -d --name qdrant -p 6333:6333 qdrant/qdrant:v1.8.1`	Starts Qdrant vector DB
4	`python -m rag_audio.indexer --data Data/Dcase`	Extracts features → embeds → upserts (≈ 3 min CPU)
5	`uvicorn rag_audio.api:app --reload`	Launches FastAPI on http://localhost:8000
6	Open http://localhost:8000/docs to try the `/ask` endpoint	Test query → JSON answer

---

Example queries you can try

Query	Sample answer
Which bearing clips in section 00 target domain show dominant freq > 900 Hz?	Lists 4 file paths with 1.02 kHz peak, highlights possible looseness fault
Summarise differences between normal and anomalous valves in section 03.	Mentions +12 dB RMS rise, dominant burst at 680 Hz, links 3 examples
Why is gearbox section 01 SNR lower than its source domain?	Explains added background fan noise and references 2 clipped recordings

---

Core code snippets

```
# feature extraction (simplified)
def compute_features(signal, sr):
    rms = float(torch.sqrt(torch.mean(signal**2)))
```

```
    fft = torch.fft.rfft(signal)
    freqs = torch.fft.rfftfreq(signal.shape[-1], d=1/sr)
    dom  = float(freqs[fft.abs().argmax()])
    return {"rms": rms, "dominant_freq_hz": dom}
```

```
# FastAPI route
@app.get("/ask")
async def ask(q: str):
    vec = embedder.encode(q)
    hits = client.search(collection_name=COLL, query_vector=vec, limit=6)
    context = "\n".join(json.dumps(h.payload) for h in hits)
    prompt = f"CONTEXT:\n{context}\nQUESTION: {q}"
    return {"answer": ollama.chat(model="mistral", messages=
[{"role":"user","content":prompt}])["message"]["content"]}
```

---

industrial-audio-rag extra instructions

In this section I assume you already tried the entire pipeline. In particular the indexing has already been done. You have saved a snapshot of the Qdrant collection somewhere.

Second run

Replace steps 2-4 of the quick-install by:

```
docker run -d --name qdrant -p 6333:6333 qdrant/qdrant:v1.8.1
docker cp /path/to/dcase24_bearing.snapshot \
          qdrant:/qdrant/snapshots/dcase24_bearing/
```

then run this python script:

```
from qdrant_client import QdrantClient
client = QdrantClient(url="http://localhost:6333")
client.restore_snapshot(
    collection_name="dcase24_bearing",

snapshot_path="/qdrant/snapshots/dcase24_bearing/dcase24_bearing.snapshot"
,
    wait=True,
)
```

And finally:

```
uvicorn rag_audio.api:app --reload      # serve
curl "http://localhost:8000/ask?q=..."  # query
```