Python and AI/ML for Weather, Climate and Environmental Applications

Let us enjoy playing with Python and AI/ML!

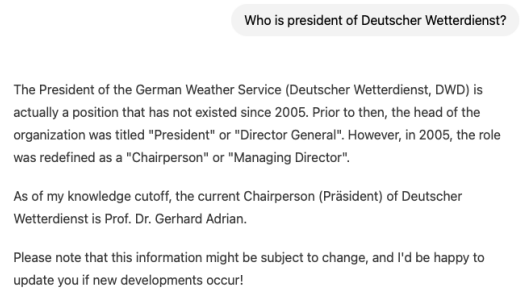
Five-Day Schedule Overview

Time	Day 1	Day 2	Day 3	Day 4	Day 5
09:00–10:00	Opening by ECMWF DG, Start: Coding & Science in the Age of AI	Neural Network Architectures	Diffusion and Graph Networks	MLOps Foundations	Model Emulation, AIFS and AICON
10:00–11:00	**Lab:** Python Startup: Basics	**Lab:** Feed-forward and Graph NNs	**Lab:** Graph Learning with PyTorch	**Lab:** Containers and Reproducibility	**Lab:** Emulation Case Studies
11:00–12:00	Python, Jupyter and APIs	Large Language Models	Agents and Coding with LLMs	CI/CD for Machine Learning	AI-based Data Assimilation
12:00–12:45	**Lab:** Work environments, Python everywhere	**Lab:** Simple Transformer and LLM Use	**Lab:** Agent Frameworks	**Lab:** CI/CD Pipelines	**Lab:** Graph-based Assimilation
12:45–13:30			**Lunch Break**		
13:30–14:30	Visualising Fields and Observations	Retrieval-Augmented Generation (RAG)	DAWID System and Feature Detection	Anemoi: AI-based Weather Modelling	AI and Physics
14:30–15:30	**Lab:** GRIB, NetCDF and Obs Visualisation	**Lab:** RAG Pipeline	**Lab:** DAWID Exploration	**Lab:** Anemoi Training Pipeline	**Lab:** Physics-informed Neural Networks
15:30–16:15	Introduction to AI and Machine Learning	Multimodal Large Language Models	MLflow: Managing Experiments	**The AI Transformation**	Learning from Observations Only
16:15–17:00	**Lab:** Torch Tensors and First Neural Net	**Lab:** Radar, SAT and Multimodal Data	**Lab:** MLflow Hands-on	**Lab: How work style could change**	**Lab:** ORIGEN and Open Discussion
17:00–20:00				Joint Dinner	

Why Retrieval-Augmented Generation (RAG)?

Limitations of standalone LLMs

- Fixed knowledge at training time
- No access to private or local data
- Risk of hallucinated answers



LLMs answer from *learned parameters only*. This can be completely off reality!

Typical real-world needs

- Large code bases
- Technical documentation
- Evolving project knowledge

Key idea:

Bring the knowledge to the model.

Core Idea of Retrieval-Augmented Generation

Standard LLM workflow

Prompt → LLM → Answer

- No external knowledge
- No verification

Pure LLMs work well for general reasoning, language understanding, summarization, and creative text generation when the required knowledge is common and static. They perform poorly when accurate, up-to-date, proprietary, or highly technical domain knowledge is required, because they cannot verify facts or access external sources. In such cases, LLMs tend to hallucinate plausible but incorrect answers.

RAG workflow

Query → Retrieve → LLM → Answer

- Search relevant documents
- Inject context into prompt
- Grounded answers

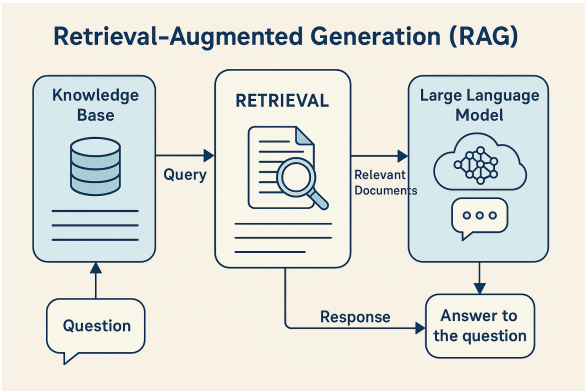
RAG Architecture Overview

RAG is more than just text search!

Main components

- Document collection
- Vector database
- Retriever
- Language model

Each component has a *clear role*.



Documents provide context for generation

Vector Databases and Transformer Embeddings

Text to vector mapping

Given a tokenized input sequence

$$(w_1, w_2, \ldots, w_n)$$

a Transformer maps tokens to embeddings:

$$x_i = E(w_i) \in \mathbb{R}^d$$

After contextualization (self-attention):

$$h_i = \mathrm{Transformer}(x_1, \ldots, x_n)$$

A sentence or paragraph embedding is typically:

$$z = \frac{1}{n} \sum_{i=1}^{n} h_i$$

Vector database view

Each paragraph is stored as:

$$z_j \in \mathbb{R}^d$$

Similarity search uses a distance measure, e.g.

$$\mathrm{sim}(z_q, z_j) = \frac{z_q \cdot z_j}{\|z_q\| \, \|z_j\|}$$

or equivalently:

$$\|z_q - z_j\|_2$$

Key point: The same embedding space is reused — now *outside* the LLM.

Elementary Vector Database — Setup

Toy example

- Small set of short sentences
- Different semantic topics
- Some paraphrases, some unrelated

We use this to build intuition before scaling to real documents.

Example sentences

```
sentences = [
  "The sun is shining brightly today.",
  "Heavy rain is falling over the city.",
  "Neural networks can learn patterns.",
  "Transformers use self-attention."
]
```

Each sentence will become one vector in a shared space.

$$\textbf{Sentence} \rightarrow \mathbf{z} \in \mathbb{R}^d$$

Text to Vector Space

Sentence embedding

A sentence s_i is mapped to a vector

$$z_i \in \mathbb{R}^d$$

using a pretrained Transformer:

$$z_i = f_\theta(s_i)$$

In our example:

$$d = 384$$

Embedding matrix

For N sentences, we obtain:

$$Z = \begin{bmatrix} z_1^\top \\ z_2^\top \\ \vdots \\ z_N^\top \end{bmatrix} \in \mathbb{R}^{N \times d}$$

Each row corresponds to *one sentence*.

Similarity Search

Query embedding

A query sentence is mapped to:

$$z_q \in \mathbb{R}^d$$

Similarity to stored sentences is measured using cosine similarity:

$$\mathrm{sim}(z_q, z_i) = \frac{z_q \cdot z_i}{\|z_q\| \, \|z_i\|}$$

Similarity search (conceptual)

```
scores = cosine_similarity(
    query_embedding,
    embeddings )

top_k = argsort(scores)[-k:]
```

Result: Sentences with closest meaning are retrieved — no keywords needed.

See example 1_vector_db_elementary.ipynb in the code07/ directory carrying out transforms explicitely. The command `SentenceTransformer("all-MiniLM-L6-v2")` loads a pretrained Transformer-based sentence embedding model that maps each input sentence to a fixed-size vector $z \in \mathbb{R}^{384}$ capturing its semantic meaning. The model is publicly available at `huggingface.co/sentence-transformers/all-MiniLM-L6-v2`.

Document Search via Embeddings

Goal

- Search *inside* a document
- Retrieve relevant pages or sections
- No keyword matching required

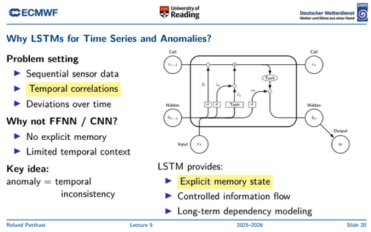
We use the same vector-space idea as in the elementary example.

"temporal pattern" find pdf →

Input

- One lecture PDF (Lecture 05)
- Pages as semantic units
- Natural-language queries

Output: Relevant pages from the PDF



From PDF Pages to Vectors

Decomposition

The PDF is split into pages:

$$\{\text{page}_1, \ldots, \text{page}_N\}$$

Each page is embedded as:

$$z_i = f_\theta(\text{page}_i) \in \mathbb{R}^d$$

In our example:

$$d = 384$$

Embedding matrix

All pages form:

$$Z \in \mathbb{R}^{N \times d}$$

Extract text from PDF pages

```
for i, page in enumerate(reader.pages):
    text = page.extract_text()
    if text and len(text.strip()) > 50:
        pages.append(text.strip())
        page_ids.append(i + 1)
```

Embedding

```
page_embeddings = model.encode(
    pages,
    convert_to_numpy=True,
    show_progress_bar=False)
```

Each row corresponds to one PDF page.
Key point: Documents are represented as collections of vectors.

Semantic Search inside a PDF

Query embedding

A user query is mapped to:

$$z_q \in \mathbb{R}^d$$

Similarity to page embeddings is computed via:

$$\mathrm{sim}(z_q, z_i) = \frac{z_q \cdot z_i}{\|z_q\| \, \|z_i\|}$$

Top-k most relevant pages are retrieved.

Semantic page search

```python
def search_pdf(query, model, pages, embeddings, page_ids, top_k=3):
    q_emb = model.encode(
        query, convert_to_numpy=True)
    scores = []
    for i, emb in enumerate(embeddings):
        score = cosine_similarity(q_emb, emb)
        scores.append((page_ids[i], score, pages[i]))

    scores.sort(key=lambda x: x[1], reverse=True)
    return scores[:top_k]
```

From Retrieval to Evidence

What we display

- Retrieved page number
- Extracted text snippet
- Original PDF page

This allows direct verification of the search result.

Why this matters

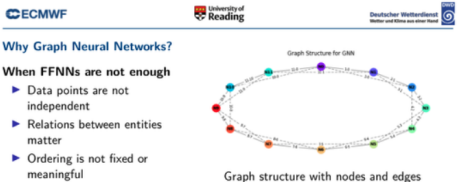
- Transparent retrieval
- No hidden reasoning
- Trust through inspectable sources

🔍 Query: **What is a graph network**

Result 1: Page 9 (similarity = 0.623)

Extracted text (snippet):

> Why Graph Neural Networks? When FFNNs are not enough ▶Data points are not independent ▶Relations between entities matter ▶Ordering is not fixed or meaningful Typical examples ▶Physical grids and meshes ▶Sensor networks ▶Molecules, social networks Graph structure with nodes and edges Core idea ▶Nodes exchange information ▶Learning respects graph structure ▶Inductive bias for relational data Roland Potthast Lecture 5 2025–2026 Slide 8



Next: **Retrieval Augmented Generation:** Retrieve first — generate later.

Why Do We Need FAISS?

Above approach

- Loop over all embeddings
- Compute similarity one by one
- Exact but very (!) slow

Cost of one query:

$$\mathcal{O}(N \cdot d)$$

Problem size

- N: number of stored vectors (pages, paragraphs, documents)
- d: embedding dimension (e.g. $d = 384$)

Each query compares against *all* vectors.

Scaling issue: Large N makes brute-force search infeasible.

Typical RAG library sizes

- **Personal projects:** 10^2–10^3 documents $\rightarrow$ $N \sim 10^4$ chunks
- **Team / institutional data:** 10^4–10^5 documents $\rightarrow$ $N \sim 10^6$–10^7 chunks
- **Enterprise-scale systems:** 10^6+ documents $\rightarrow$ $N \gg 10^7$ chunks

Chunking multiplies the number of stored vectors N.

FAISS — Vector Search at Scale

- Facebook AI Similarity Search
- Optimized nearest-neighbor search
- Designed for large vector collections

Stores vectors:

$$z_i \in \mathbb{R}^d$$

Key idea

- Build an index once
- Query many times
- Fast top-k retrieval

Search replaces explicit loops.

Hierarchical search in FAISS

FAISS accelerates nearest-neighbor search by introducing a *coarse-to-fine hierarchy* in vector space.

Step 1: Coarse partitioning

The vector space is partitioned into M regions using representative centroids:

$$\{c_1, \ldots, c_M\}, \quad c_j \in \mathbb{R}^d.$$

Each stored vector z_i is assigned to its nearest centroid:

$$z_i \mapsto c(z_i).$$

FAISS in Practice

Index construction

- Choose distance metric
- Add all embeddings
- Index lives in memory or on disk

Common choice: IndexFlatL2

Step 2: Query routing

For a query vector z_q, FAISS finds the closest centroids:

$$\mathcal{C}_q = \arg\operatorname*{top}_L \|z_q - c_j\|_2, \quad L \ll M.$$

Build and query FAISS index

```
import faiss
d = page_embeddings.shape[1]
index = faiss.IndexFlatL2(d)

index.add(page_embeddings)

distances, indices = index.search(
    query_embedding, k)
```

Step 3: Local search

Exact distances are computed only for vectors stored in the selected regions:

$$z_i \text{ with } c(z_i) \in \mathcal{C}_q.$$

Streaming LLM Responses

Standard LLM interaction

$$\text{prompt} \longrightarrow \text{full response}$$

User waits until generation is finished.

Streaming interaction

$$\text{prompt} \longrightarrow (\delta_1, \delta_2, \dots)$$

Partial tokens are delivered incrementally.

Recall that

LLMs generate tokens incrementally!

What streaming changes

- Faster perceived response
- Progressive rendering
- Interactive user experience

What streaming does not change

- Model architecture
- Reasoning capability
- Final content

OpenAI Streaming

Streaming API call

- Same prompt structure
- `stream=True`
- Tokens arrive as deltas

Each chunk contains new text:

$$\delta_k \subset \text{response}$$

There are several older OpenAI interface versions that language models may still suggest. Be careful: to avoid deprecated APIs, it is often necessary to explicitly provide a current code template.

OpenAI streaming in Jupyter

```
stream = client.chat.completions.create(
  model="gpt-4o-mini", messages=[...],
  stream=True )

accumulated = ""
handle = display(
  Markdown(""), display_id=True)

for chunk in stream:
    delta = chunk.choices[0].delta
    if delta.content:
        accumulated += delta.content
        handle.update(
          Markdown(accumulated))
```

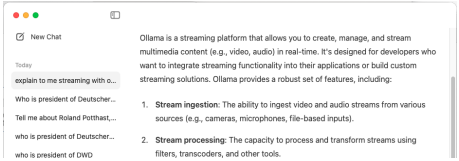
Streaming with Local LLMs (Ollama)

Local execution

- Models run on local hardware
- No external API calls
- Full data control

Streaming follows the same principle:

$$(\delta_1, \delta_2, \dots)$$



Advantages

- Privacy and compliance
- **Offline usage - train or plane**
- No usage-based cost

Trade-offs

- Smaller models
- Hardware dependent speed

List of all available models:
https://ollama.ai/library

Ollama Streaming in Python

Ollama streaming call

- Local HTTP interface
- Same message format
- Streaming enabled

Works with: `llama3`, `mistral`, `mixtral`, `deepseek-r1`

Ollama streaming in Jupyter

```
stream = ollama.chat( model="llama3",
    messages=[...], stream=True )

accumulated = ""
handle = display(
  Markdown(""), display_id=True)

for chunk in stream:
    if "content" in chunk["message"]:
        accumulated += chunk["message"]\
        ["content"]
        handle.update(Markdown(accumulated))
```

RAG Implementation: Concrete Steps

Implemented components

- text chunking (files → chunks) ✓
- sentence embeddings ✓
- FAISS index ✓
- metadata tracking ✓
- LLM query with context ✓

Execution order

1. build vector database (offline)
2. embed user query
3. retrieve top-k chunks
4. assemble prompt context
5. generate answer (optional streaming)

```
stream = client.chat.completions.create(
    model="gpt-4o-mini",
    messages=[
        {
            "role": "system",
            "content": (
                "You are a technical assistant. "
                "Answer only using the provided context."
            ),
        },
        {
            "role": "user",
            "content": (
                "Context:\n"
                f"{context}\n"
                "Question:\n"
                f"{query}"
            ),
        },
    ],
```

Top-k Retrieval: What Do We Actually Get?

FAISS output

For a query embedding z_q, FAISS returns:

$$\{(i_1, d_1), (i_2, d_2), \ldots, (i_k, d_k)\}$$

- i_j: index of a stored chunk
- d_j: distance or similarity score

These indices refer to:

$$z_{i_j} \leftrightarrow \text{chunk}_{i_j}$$

Important clarification

- FAISS returns *vectors*, not text
- Text is recovered via metadata lookup
- Ordering is by similarity score

At this stage:

- no LLM involved
- no generation
- no reasoning

From FAISS Indices to Text Chunks

Stored metadata per chunk

Each vector z_i is linked to:

- file path
- line range or section
- raw text content

This mapping is stored outside FAISS (e.g. Python lists, JSON, databases).

Lookup step

For each returned index i_j:

$$i_j \rightarrow (\text{file}, \text{lines}, \text{text})$$

This produces a ranked list:

$$(\text{chunk}_1, \ldots, \text{chunk}_k)$$

Result: Concrete, inspectable evidence.

Composing the Retrieval Context

Context construction

Retrieved chunks are:

- ordered by similarity
- concatenated into one context block

Typical structure:

- file reference
- optional score
- text snippet

Context size is explicitly limited.

Context assembly (ICON example)

```
context = ""
for r in retrieved_chunks:
  context += (
    f"[{r.file}, lines {r.start}-{r.end}]\n"
    + r.text + "\n\n" )
```

OpenSearch

What is OpenSearch?

OpenSearch is an open-source, distributed search and analytics engine.

It is designed to index and query large collections of documents with low latency.

Originally derived from Elasticsearch, OpenSearch is developed under the Apache 2.0 license.

Core capabilities

- full-text search (inverted index)
- metadata filtering and aggregation
- persistent, disk-based indices
- distributed execution across nodes
- **vector similarity search**

OpenSearch combines text-based and vector-based retrieval in a single system.

Google Search as Retrieval-Augmented Generation

Core idea

Retrieval-Augmented Generation (RAG) combines an LLM with an external information source.

In classical RAG, retrieval uses:

- vector databases (FAISS)
- local document collections

Google-based RAG

Here, retrieval is delegated to:

- Google Search
- live web documents

Same logical pipeline

$$\text{Query} \rightarrow \text{Retrieve} \rightarrow \text{Context} \rightarrow \text{LLM} \rightarrow \text{Answer}$$

Key difference

- Local RAG: curated, controlled, static
- Google RAG: open, dynamic, up-to-date

Interpretation

The LLM does *not* search — it *summarizes retrieved evidence*.

Search to LLM Pipeline

Pipeline steps

1. Issue search query
2. Retrieve top-k URLs
3. Scrape page text
4. Build context block
5. Ask LLM to answer

Answer =
LLM (Query, Web Evidence)

Important

No embeddings are required — ranking by Search Engine.

Google–LLM pipeline (schematic)

```
urls = search_google(query)
texts = [ scrape_website(url),
  for url in urls ]

prompt = f"""
Answer the question using
the sources below.
Question: {query}
Sources: {texts}
"""

response = client.chat.completions.
    create( model="gpt-4o-mini",
  messages=[{"role":"user", "content
    ":prompt}] )
```