CLUSTER REPLY

POLIMI - 2025 SPECIAL PROJECT

13/11/2025

REPLY CLUSTER

TEAM

Alessandro Belotti
al.belotti@reply.it

Vincenzo Graziani
v.graziani@reply.it

Davide Pietrasanta
d.pietrasanta@reply.it

INNOVATOR’S LAB

EXAM TRAINER AGENT

REPLY
CLUSTER

SUMMARY

1.

Introduction

DP0

What is an LLM and why the need of RAG?

2.

RAG architecture

What are the main components of a RAG architecture?

3.

RAG benefits over fine tuning

Pros and cons of a RAG

4.

Hand's on lab

Implement your own exam trainer agent.

Slide 4

DP0 Sii coerente con i colori

Davide Pietrasanta, 2025-10-23T09:17:32.061

Introduction

- LLM definition
- LLM and its limitations
- RAG definition
- Why we need RAG

Introduction: LLM definition DP0

- A large language model (**LLM**) is a type of AI model that can recognize and generate text among other tasks
- LLMs are very large models that are pre-trained on vast amounts of data
- Built on **transformer architecture** is a set of neural network that consist of an encoder and a decoder with **self-attention** capabilities
- It can perform completely different tasts such as answering questions, summarizing documents, translating languages and completing sentences.

Examples are GPT-4.1 mini, **Llama 3.2** or **GPT-5**.

Slide 6

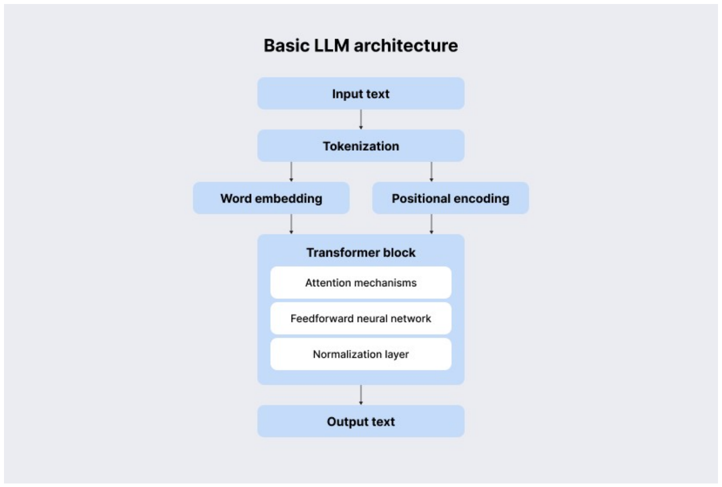
DP0 Non ci spendere troppo tempo

Davide Pietrasanta, 2025-10-23T09:17:55.127

Introduction: LLM definition

In a simple terms, an LLM is a ML model that has been fed enough examples to be able to **recognize** and **interprete** human language or other types of complex data.

The quality of the samples impacts how well LLMs will learn natural language, so an LLM’s programmers may use a **curated dataset**.



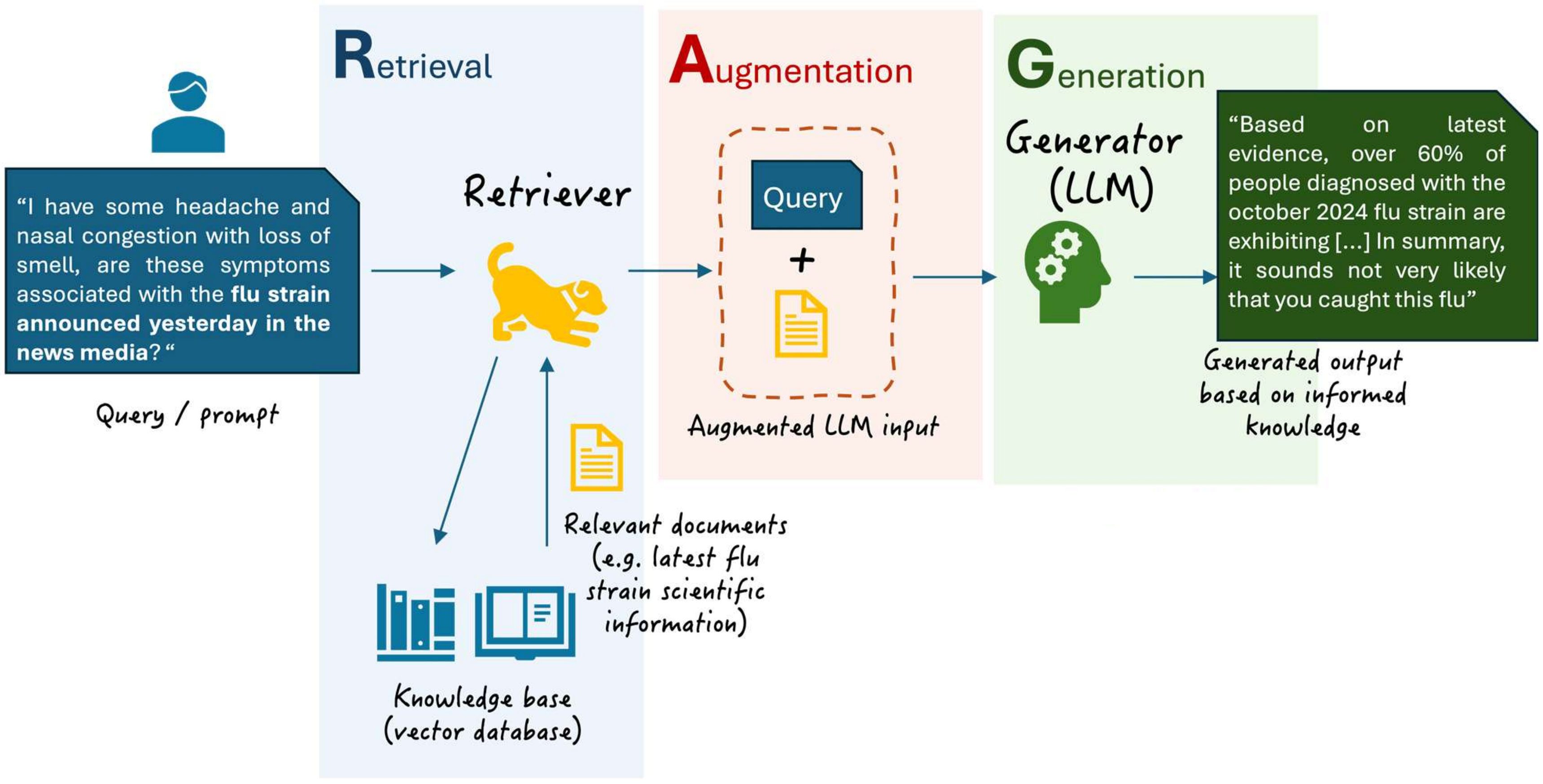
Introduction: LLM limitations

- **Outdated knowledge**: Models only know data up to their training data
- **Hallucinations**: May produce confident but factually wrong answers
- **Lack of specificity**: Struggles with domain or organization specific details
- **No source citations**: Hard to trace or verify information origins
- **Costly updates**: Retraining with new data is slow and resource-intensive
- **False answers**: May generate plausible text even when unsure.

Introduction: RAG definition

- **RAG** (Retrieval Augmented Generation) is an advanced technique used in LLMs
- RAG combines retrieval and generation processes to enhance the capabilities of LLMs
- In RAG the model retrieves **relevant information** from a **knowledge base** or external sources
- This retrieved information is then used in conjunction with the model's internal knowledge to generate coherent and contextually **relevant responses**
- RAG enabels LLMs to produce higher quality and more **context-aware outputs** compared to traditional generation methods

Introduction: RAG definition


Retrieval
Augmentation
Generation
"I have some headache and nasal congestion with loss of smell, are these symptoms associated with the **flu strain announced yesterday in the news media**? "
Query / prompt
Retriever
Knowledge base (vector database)
Relevant documents (e.g. latest flu strain scientific information)
Query
+
Augmented LLM input
Generator (LLM)
"Based on latest evidence, over 60% of people diagnosed with the october 2024 flu strain are exhibiting [...] In summary, it sounds not very likely that you caught this flu"
Generated output based on informed knowledge

RAG architecture

- Overall architecture
- Preparing for the hand’s on lab
- Retrieval components
  - Document processing
  - Vector embedding
  - Semantic search retrieval
  - Ranking algorithms
- Generative compoment

RAG architecture

- The two main RAG components are retriever and generation
- Suppose a student walks into a library and asks the employee, “When did the French Revolution start?”
- The employee doesn’t know the exact date, so he searches through the history books to find the correct information (**retrieval part**)
- After checking the books, he confidently replies, “The French Revolution started in 1789” (**generation part**)

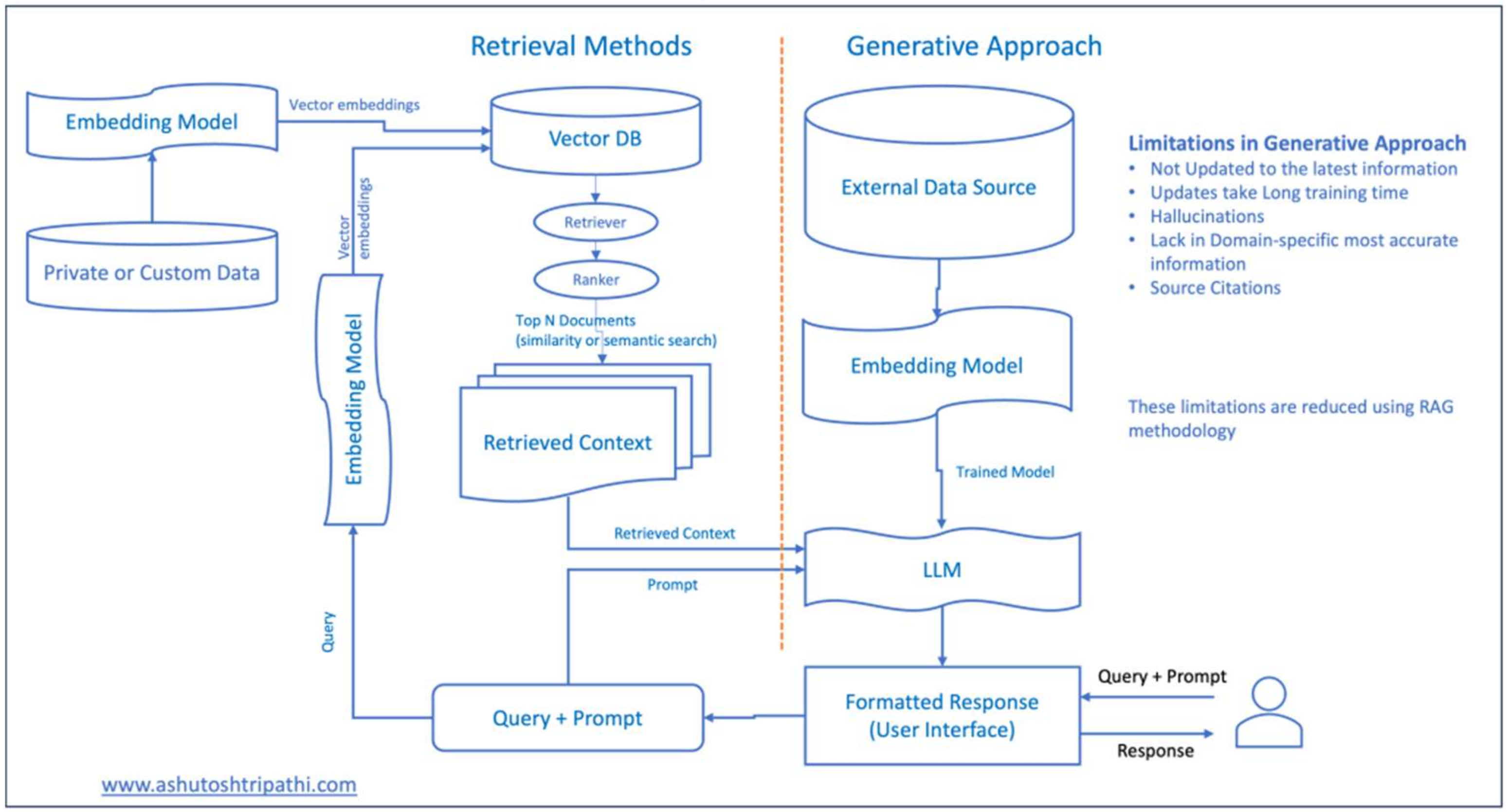
Preparing for the hand's on lab

Before you start building your first assistant you need to:

- Install VS code and python (link)
- Install and configure git on VS code (link)
- Clone the github repository
- Install docker and pull qdrant image from docker hub
- Install dependencies (*pip install -r requirements.txt*)
- Start qdrant locally (*docker run -p 6333:6333 qdrant/qdrant*)
- Run the app (*streamlit run app.py*)

RAG architecture: summary

Retrieval Augmented Generation



Limitations in Generative Approach

- Not Updated to the latest information
- Updates take Long training time
- Hallucinations
- Lack in Domain-specific most accurate information
- Source Citations

These limitations are reduced using RAG methodology

RAG architecture: Retrieval component

Document processing

The processing phase is divided into

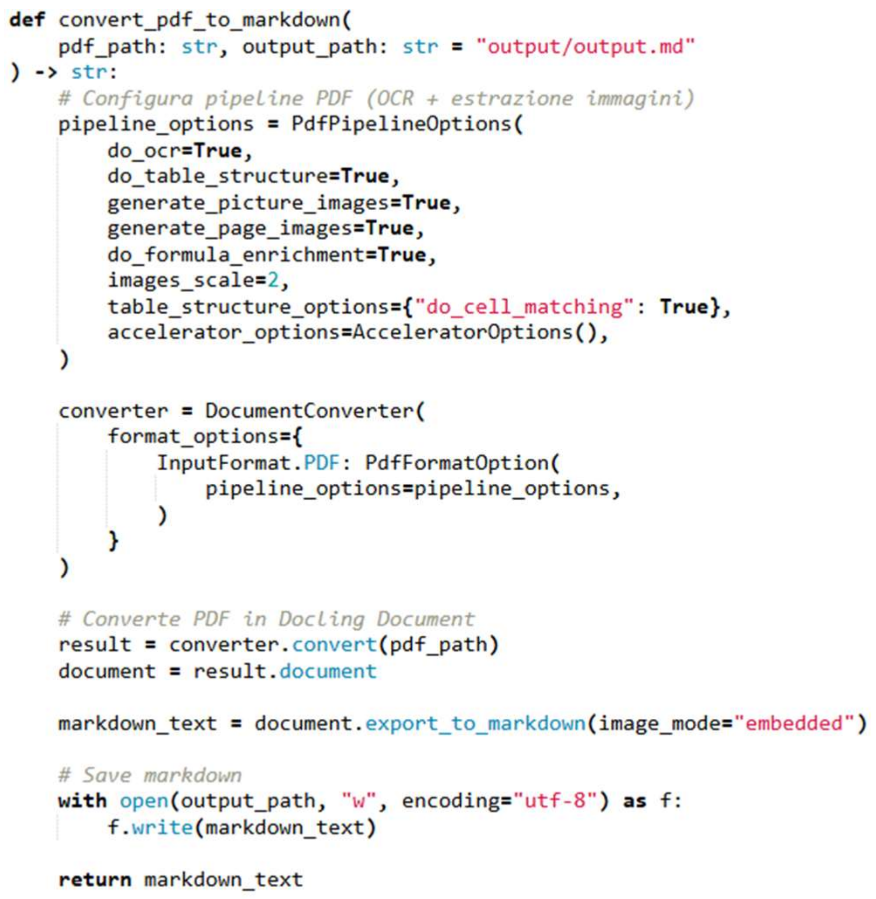
- Document scanning with **OCR**
- Segmentation into **chunks**

The document has to be convert into text, this is done using the Optical Character Recognition (OCR) tool. There are many libraries that are able to do it:

- **Docling**
- Tesseract OCR
- EasyOCR

RAG architecture : Retrieval component

DP0



```python
def convert_pdf_to_markdown(
    pdf_path: str, output_path: str = "output/output.md"
) -> str:
    # Configura pipeline PDF (OCR + estrazione immagini)
    pipeline_options = PdfPipelineOptions(
        do_ocr=True,
        do_table_structure=True,
        generate_picture_images=True,
        generate_page_images=True,
        do_formula_enrichment=True,
        images_scale=2,
        table_structure_options={"do_cell_matching": True},
        accelerator_options=AcceleratorOptions(),
    )

    converter = DocumentConverter(
        format_options={
            InputFormat.PDF: PdfFormatOption(
                pipeline_options=pipeline_options,
            )
        }
    )

    # Converte PDF in Docling Document
    result = converter.convert(pdf_path)
    document = result.document

    markdown_text = document.export_to_markdown(image_mode="embedded")

    # Save markdown
    with open(output_path, "w", encoding="utf-8") as f:
        f.write(markdown_text)

    return markdown_text
```

src/utils.py

Slide 16

DP0 Qui vai sul codice? Io non mostrerei codice sulle slide

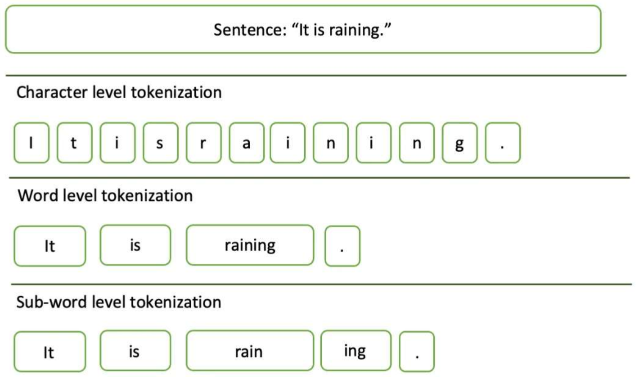
Davide Pietrasanta, 2025-10-23T09:18:43.533

RAG architecture: Retrieval component

Document processing

The second phase of this process is the **tokenization**, so after that the tokenized text is converted into smaller chuncks. Python libraries like nltk or **AutoTokenizer** are able to implement these tokenizer strategies:

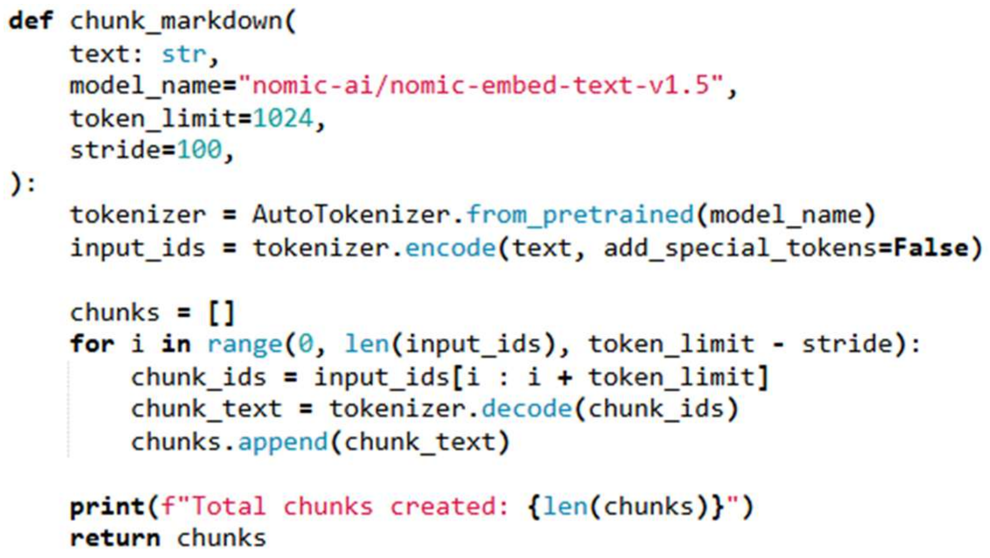
- Word or whitespace level
- Character level
- **Subword tokenization**


Sentence: “It is raining.”
Character level tokenization
I t i s r a i n i n g .
Word level tokenization
It is raining .
Sub-word level tokenization
It is rain ing .

The main advantage is that you ensure that each chunk fits into model’s input limit and the semantic and syntactic integrity is preserved.

So remember to tokenize before apply chunking !!!

RAG architecture : Retrieval component



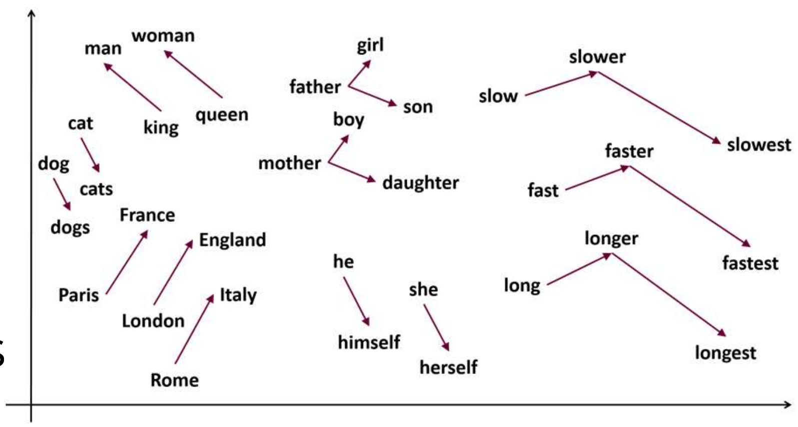
```python
def chunk_markdown(
    text: str,
    model_name="nomic-ai/nomic-embed-text-v1.5",
    token_limit=1024,
    stride=100,
):
    tokenizer = AutoTokenizer.from_pretrained(model_name)
    input_ids = tokenizer.encode(text, add_special_tokens=False)

    chunks = []
    for i in range(0, len(input_ids), token_limit - stride):
        chunk_ids = input_ids[i : i + token_limit]
        chunk_text = tokenizer.decode(chunk_ids)
        chunks.append(chunk_text)

    print(f"Total chunks created: {len(chunks)}")
    return chunks
```

src/chunk_embed.py

RAG architecture : Retrieval component

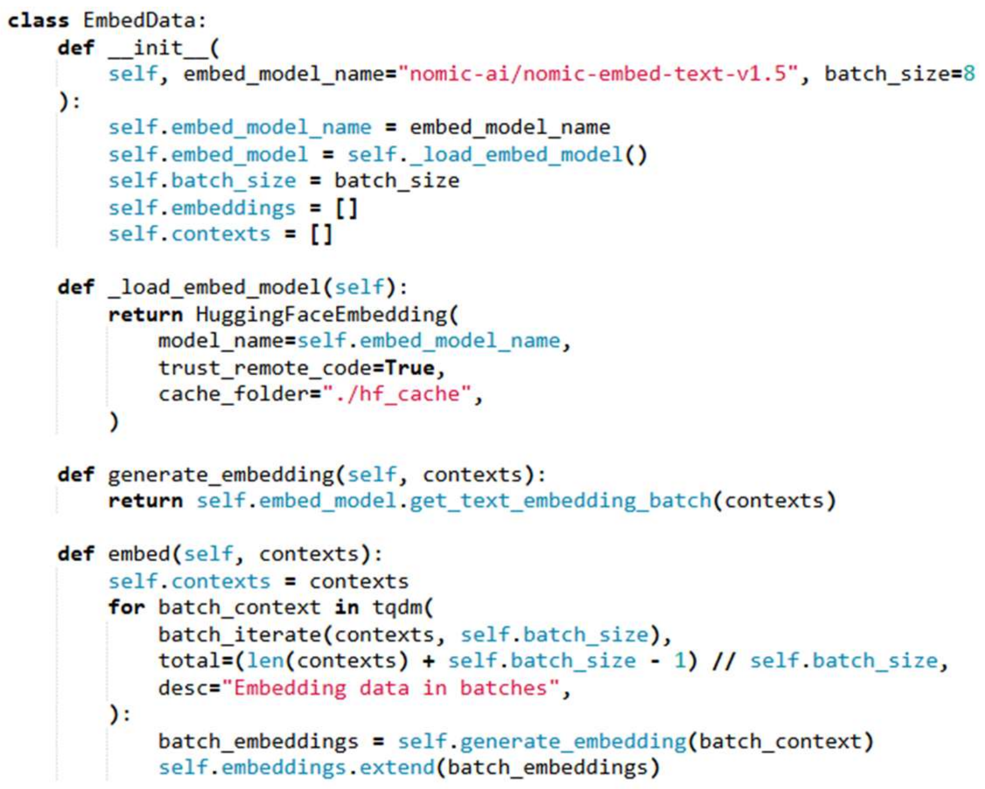
Vector embedding

- An embedding is a numerical representation (**high dimensional vector**) that represent an item of information
- Similar items ar mapped to nearby vectors (the **semantic** determine the position)



- Emeddings enables semantic similarity search
- Common approaches for embedding construction:
  - OpenAI embeddings (support directly semantic search)
  - Static word embeddings (spaCy, GloVe or Word2Vec libraries)
  - **SentenceTransformers** (HuggingFace, choice for local embeddings)

RAG architecture : Retrieval component



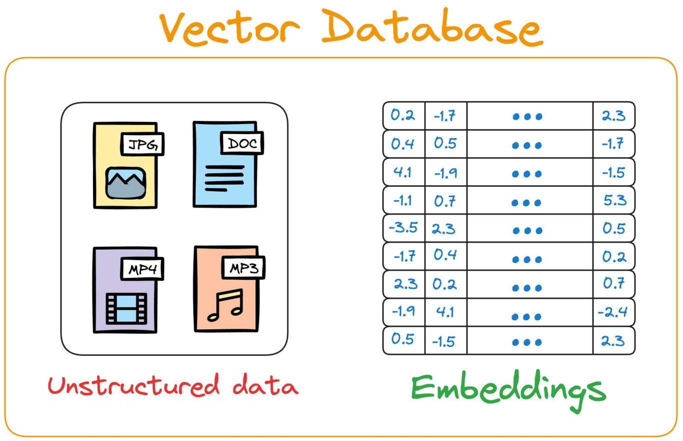
```python
class EmbedData:
    def __init__(
        self, embed_model_name="nomic-ai/nomic-embed-text-v1.5", batch_size=8
    ):
        self.embed_model_name = embed_model_name
        self.embed_model = self._load_embed_model()
        self.batch_size = batch_size
        self.embeddings = []
        self.contexts = []

    def _load_embed_model(self):
        return HuggingFaceEmbedding(
            model_name=self.embed_model_name,
            trust_remote_code=True,
            cache_folder="./hf_cache",
        )

    def generate_embedding(self, contexts):
        return self.embed_model.get_text_embedding_batch(contexts)

    def embed(self, contexts):
        self.contexts = contexts
        for batch_context in tqdm(
            batch_iterate(contexts, self.batch_size),
            total=(len(contexts) + self.batch_size - 1) // self.batch_size,
            desc="Embedding data in batches",
        ):
            batch_embeddings = self.generate_embedding(batch_context)
            self.embeddings.extend(batch_embeddings)
```

src/chunk_embed.py

RAG architecture : Retrieval component

Vector embedding

- Each chunck is turn into a vector
- Each vector is stored into a vector DB
- Some popular vector DB are:

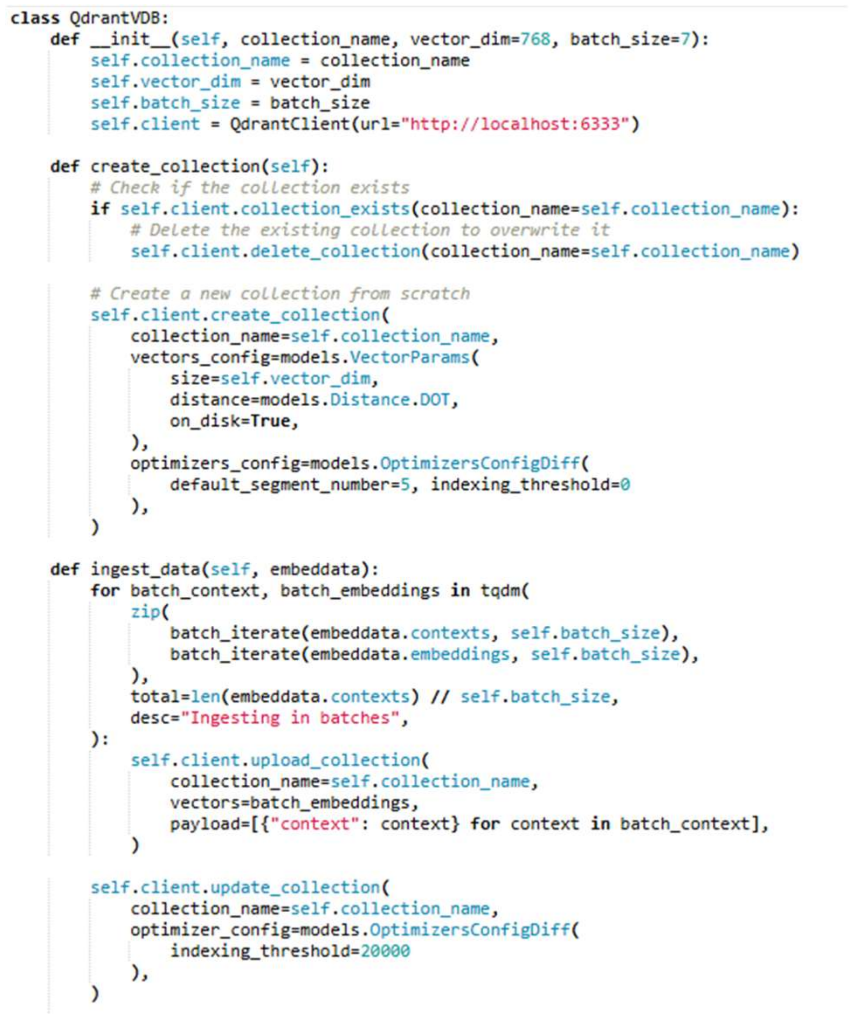


0.2	-1.7	•••	2.3
0.4	0.5	•••	-1.7
4.1	-1.9	•••	-1.5
-1.1	0.7	•••	5.3
-3.5	2.3	•••	0.5
-1.7	0.4	•••	0.2
2.3	0.2	•••	0.7
-1.9	4.1	•••	-2.4
0.5	-1.5	•••	2.3

  - FAISS: A library from Meta for efficient similarity search and clustering of dense vectors
  - Qdrant: Open-source vector **search engine and database**, popular for RAG workflows
  - ChromaDB: Lightweight and user friendly, in-memory database with persistence

Qdrant

RAG architecture : Retrieval component



```python
class QdrantVDB:
    def __init__(self, collection_name, vector_dim=768, batch_size=7):
        self.collection_name = collection_name
        self.vector_dim = vector_dim
        self.batch_size = batch_size
        self.client = QdrantClient(url="http://localhost:6333")

    def create_collection(self):
        # Check if the collection exists
        if self.client.collection_exists(collection_name=self.collection_name):
            # Delete the existing collection to overwrite it
            self.client.delete_collection(collection_name=self.collection_name)

        # Create a new collection from scratch
        self.client.create_collection(
            collection_name=self.collection_name,
            vectors_config=models.VectorParams(
                size=self.vector_dim,
                distance=models.Distance.DOT,
                on_disk=True,
            ),
            optimizers_config=models.OptimizersConfigDiff(
                default_segment_number=5, indexing_threshold=0
            ),
        )

    def ingest_data(self, embeddata):
        for batch_context, batch_embeddings in tqdm(
            zip(
                batch_iterate(embeddata.contexts, self.batch_size),
                batch_iterate(embeddata.embeddings, self.batch_size),
            ),
            total=len(embeddata.contexts) // self.batch_size,
            desc="Ingesting in batches",
        ):
            self.client.upload_collection(
                collection_name=self.collection_name,
                vectors=batch_embeddings,
                payload=[{"context": context} for context in batch_context],
            )

        self.client.update_collection(
            collection_name=self.collection_name,
            optimizer_config=models.OptimizersConfigDiff(
                indexing_threshold=20000
            ),
        )
```

src/index.py

RAG architecture : Retrieval component

Semantic search retrieval

In the final step the documents are taken from the knowledge base and a **ranker** will choose the best one.

- The new **user query** is embedded into a vector using the same embedding model as before
- The similarity between the query vector and all the document vectors in the DB is computed using
  - **Cosine similarity** (in Qdrant)
  - Dot product
  - Euclidean distance

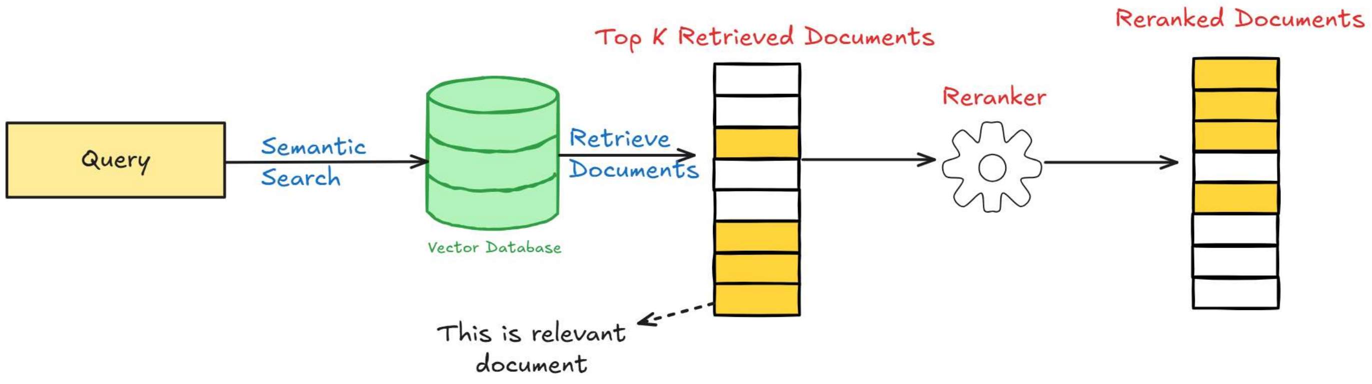
RAG architecture : Retrieval component

Ranking algorithms

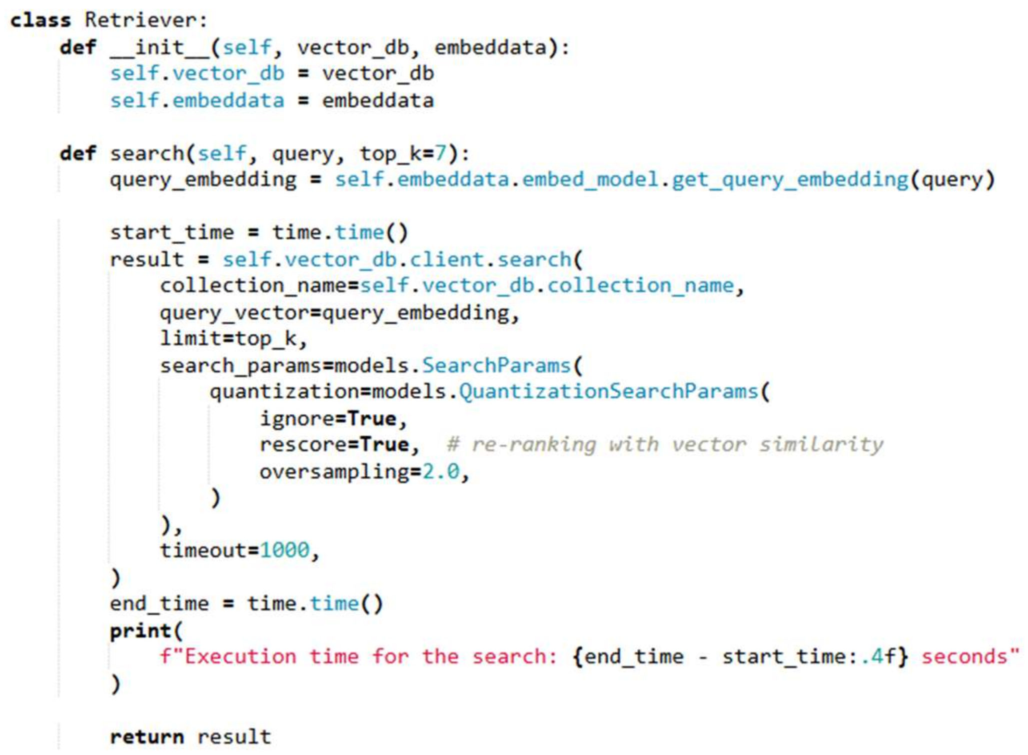
After the retrieval phase a list of **top-k** most similar documents are taken.

Now the ranker chooses only the relevant document among these. Common techniques are

- **Similarity re-ranking** (vector similarity used in qdrant)
- Lexical similarity (BM25 algorithm)



RAG architecture : Retrieval component



```python
class Retriever:
    def __init__(self, vector_db, embeddata):
        self.vector_db = vector_db
        self.embeddata = embeddata

    def search(self, query, top_k=7):
        query_embedding = self.embeddata.embed_model.get_query_embedding(query)

        start_time = time.time()
        result = self.vector_db.client.search(
            collection_name=self.vector_db.collection_name,
            query_vector=query_embedding,
            limit=top_k,
            search_params=models.SearchParams(
                quantization=models.QuantizationSearchParams(
                    ignore=True,
                    rescore=True,  # re-ranking with vector similarity
                    oversampling=2.0,
                )
            ),
            timeout=1000,
        )
        end_time = time.time()
        print(
            f"Execution time for the search: {end_time - start_time:.4f} seconds"
        )

        return result
```

src/retriever.py

RAG architecture : Generation component

The **generation** phase happens after retrieval and optional ranking.

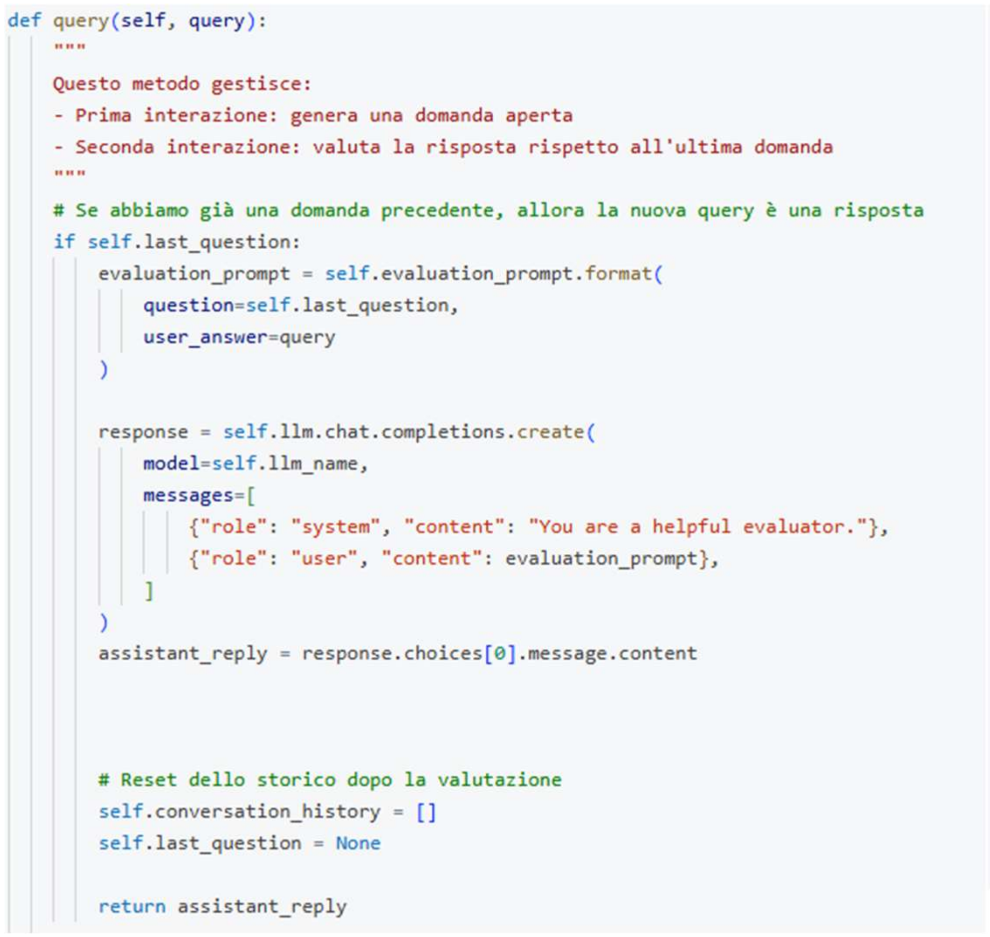
Its main purpose is to produce a natural language answer using the **retrieved context**

- Input: user query + top-k retrieved documents or chunks
- Output: final answer, summary, or explanation

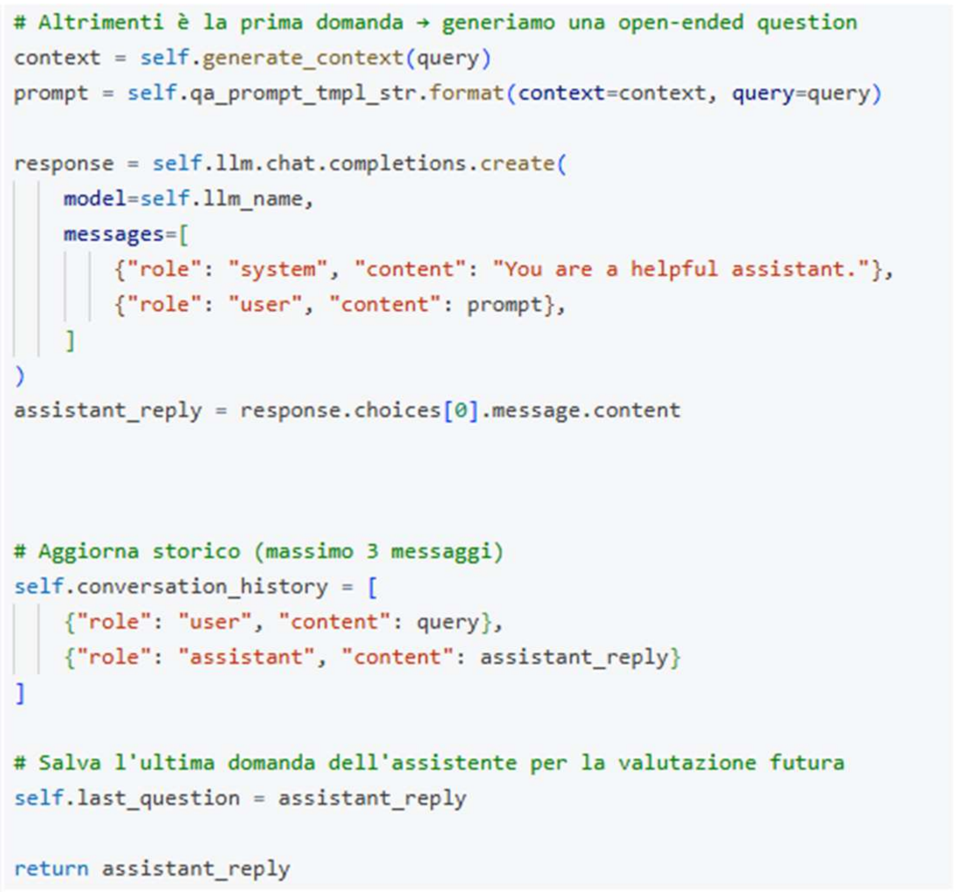
In other words, the LLM is conditioned on the retrieved context to generate a coherent response.

The user query and the retrieved context is usually incapsulated into a more **structured prompt** in order to give to the LLM additional rules.

RAG architecture : Generation component



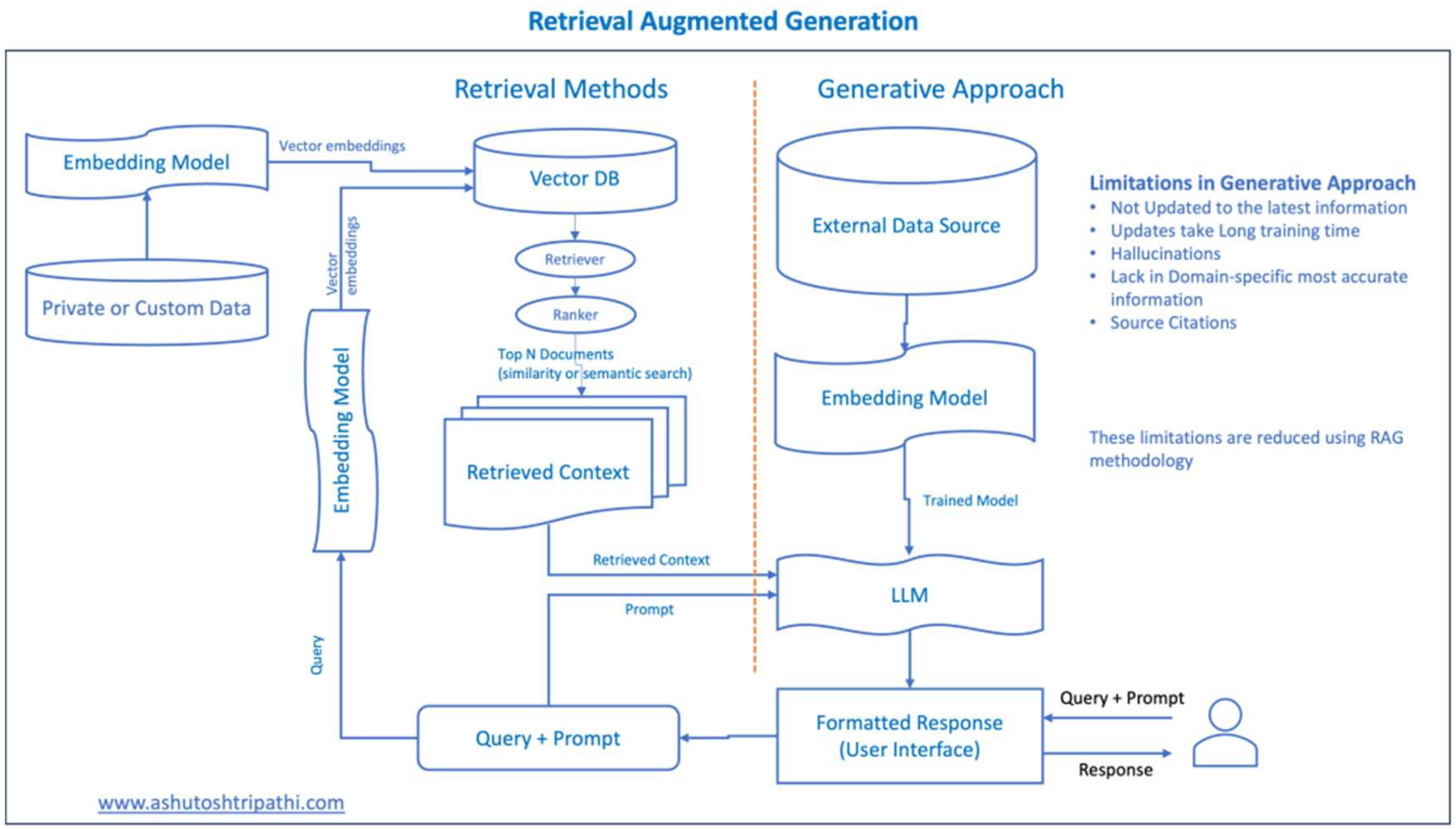
```python
def query(self, query):
    """
    Questo metodo gestisce:
    - Prima interazione: genera una domanda aperta
    - Seconda interazione: valuta la risposta rispetto all'ultima domanda
    """
    # Se abbiamo già una domanda precedente, allora la nuova query è una risposta
    if self.last_question:
        evaluation_prompt = self.evaluation_prompt.format(
            question=self.last_question,
            user_answer=query
        )

        response = self.llm.chat.completions.create(
            model=self.llm_name,
            messages=[
                {"role": "system", "content": "You are a helpful evaluator."},
                {"role": "user", "content": evaluation_prompt},
            ]
        )
        assistant_reply = response.choices[0].message.content



        # Reset dello storico dopo la valutazione
        self.conversation_history = []
        self.last_question = None

        return assistant_reply
```



```python
# Altrimenti è la prima domanda → generiamo una open-ended question
context = self.generate_context(query)
prompt = self.qa_prompt_tmpl_str.format(context=context, query=query)

response = self.llm.chat.completions.create(
    model=self.llm_name,
    messages=[
        {"role": "system", "content": "You are a helpful assistant."},
        {"role": "user", "content": prompt},
    ]
)
assistant_reply = response.choices[0].message.content



# Aggiorna storico (massimo 3 messaggi)
self.conversation_history = [
    {"role": "user", "content": query},
    {"role": "assistant", "content": assistant_reply}
]

# Salva l'ultima domanda dell'assistente per la valutazione futura
self.last_question = assistant_reply

return assistant_reply
```

src/rag_engine.py

RAG architecture

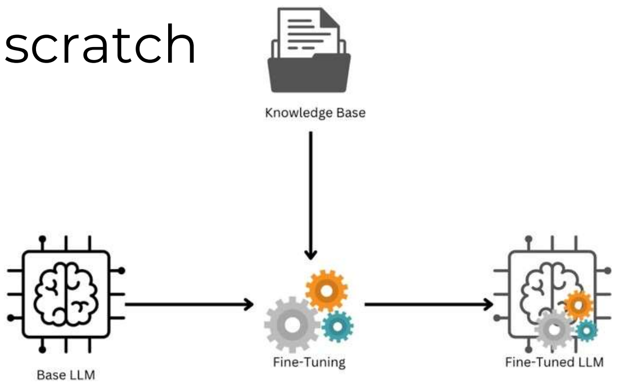


Limitations in Generative Approach
- Not Updated to the latest information
- Updates take Long training time
- Hallucinations
- Lack in Domain-specific most accurate information
- Source Citations

These limitations are reduced using RAG methodology

RAG benefits over finetuning

- Adapting a pre-trained model to a specific task or dataset to improve performance
  - Start with a pre-trained data
  - Provide a specialized dataset
  - Adjust the model's parameters
- The model weights update can be full or partial
- Reduce need for massive training data from scratch
- Risk of overfitting if dataset is too small



RAG benefits over finetuning

	Fine-tuning	**RAG**
Objective	Adapt the internal knowledge of a pre-trained model to a specific task or domain **by updating its weights**.	Improve relevance and accuracy by injecting fresh, external knowledge at inference time. **No weight updates**.
Training data	Requires labeled, task-specific datasets. **More cost**, curation effort, and compute.	Leverages a pre-trained model plus a knowledge base. Minimal or **no additional training data required**.
Adaptability	Becomes specialized for the fine-tuned task, potentially **losing general flexibility.**	Preserves general skills of the base model and adapts **dynamically** across topics depending on retrieved info.
Model architecture	Same architecture, but the internal parameters are modified and **permanently changed**.	Two-stage pipeline: a retriever fetches context; the generator produces the answer using that context. **Base model stays untouched**.

HAND’S ON LAB

REPLY
CLUSTER

Further developments

- **Implement a difficulty bar for the question generation part**
- **Implement a memory chat history**
- **Change model with a local one (try Ollama or whatever you want)**
- Expand the OCR pipeline in order to scan also images and other document types
- Implement the possibility to upload more than one document per run