clearbox AI

humans && machines

Building a Retrieval-Augmented Generation (RAG) System with Haystack

C.so Castelfidardo,30/a
10129, Torino (Italy)
info@clearbox.ai

VAT ID: (IT)12161430017

Summary

Written by Simona Mazzarino

Introduction

This guide provides a comprehensive overview of how to design and implement a Retrieval-Augmented Generation (RAG) system using Haystack. It aims to equip developers and organizations with practical knowledge and actionable steps for building efficient, scalable, and accurate RAG solutions, enabling seamless integration of powerful AI-driven information retrieval and generation capabilities. Most of the content is based on the Haystack documentation available here.

You can find the notebooks mentioned in this guide here.

What is the Retrieval-Augmented Generation?

Retrieval-Augmented Generation (RAG) is an advanced framework in the field of natural language processing (NLP) that combines the strengths of information retrieval and generative language models to enhance the quality and relevance of responses.
It addresses the limitations of generative models, which sometimes produce inaccurate or irrelevant information, by integrating them with a retrieval mechanism that provides access to external knowledge sources. At its core, RAG works by first retrieving relevant information from a knowledge base or document repository and then using this information as context for a generative
model to produce a response.

The retrieval step ensures that the model has access to accurate, up-to-date, and context-specific information, rather than relying solely on the static training data of the generative model.

This approach is particularly beneficial when dealing with queries requiring factual accuracy, domain-specific knowledge, or responses grounded in real-world data.

Which is the main architecture?

The process typically involves two main components: the retriever and the generator. The retriever searches a database or corpus for documents or passages relevant to the query. The generator, often a pre-trained model, takes the retrieved information and the original query as input to generate a coherent and informed response. By dynamically incorporating external knowledge, RAG systems can produce answers that are both creative and factually reliable.

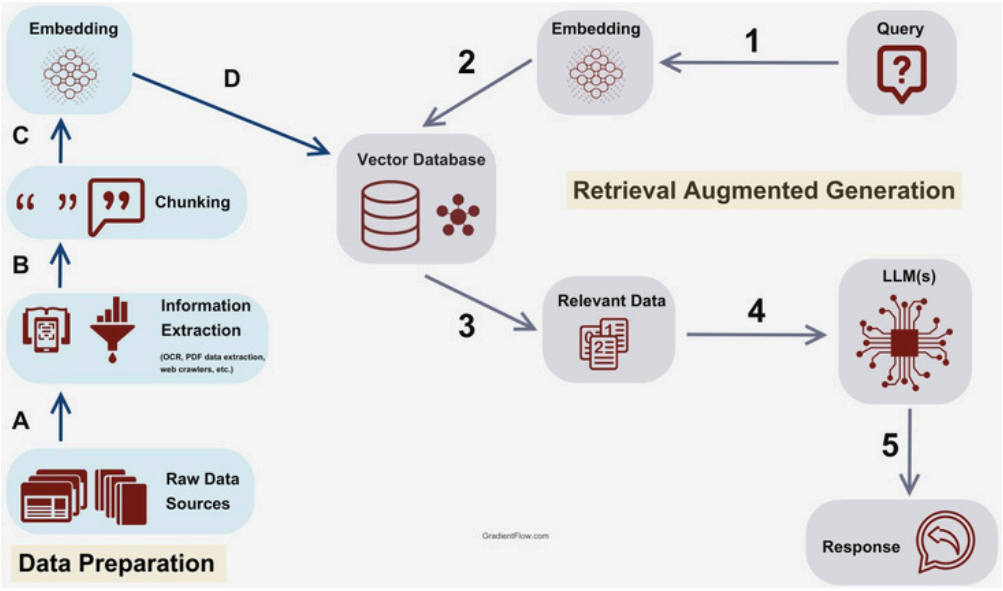


Image Source: https://gradientflow.substack.com/p/best-practices-in-retrieval-augmented

Which is the difference between RAG and fine-tuning?

The difference between Retrieval-Augmented Generation (RAG) and fine-tuning lies in how they address the limitations of pre-trained models and incorporate domain-specific or task-specific knowledge.

1. **Retrieval-Augmented Generation (RAG)**:

RAG is a framework that dynamically combines a retrieval mechanism with a generative model. Instead of relying solely on static training data, RAG retrieves

relevant external information from a knowledge base or corpus at runtime. This retrieved data is then used to inform the generative model's response. RAG does not modify the underlying generative model itself but rather enhances its output by supplying context-specific and up-to-date information from external sources. The key advantage of RAG is its ability to adapt to new information without retraining.

If the knowledge base is updated, the system can immediately leverage the new data without requiring the model to be re-trained. This makes it ideal for scenarios requiring real-time, domain-specific, or constantly evolving knowledge.

2. **Fine-tuning**:

Fine-tuning involves retraining or further training a pre-trained model, like GPT or BERT, on a specific dataset to specialize it for a particular task or domain. In fine-tuning, the model parameters are adjusted to learn the patterns and specifics of the new data. Once fine-tuned, the model becomes static—it incorporates the new

knowledge into its weights, but it cannot adapt to new information unless fine-tuned again. Fine-tuning is advantageous when the domain knowledge is stable and doesn't require frequent updates.

However, it can be computationally expensive, requires a significant amount of labeled data, and lacks flexibility to adapt to changing information unless retrained.

Key Differences:

- **Dynamic vs. Static Knowledge**: RAG retrieves and incorporates knowledge dynamically at runtime, whereas fine-tuning statically embeds knowledge nto the model during training.
- **Adaptability**: RAG can adapt to new or updated information by simply modifying the knowledge base, while fine-tuned models require retraining to reflect new knowledge.
- **Complexity and Cost**: Fine-tuning demands computational resources and labeled data, while RAG relies on an existing retriever and does not require model retraining for new knowledge.
- **Use Cases**: RAG is better for applications where the information changes frequently or is too vast to be encoded in model parameters, while fine-tuning is suitable for stable, domain-specific tasks.

What is Haystack?

Haystack is an open source framework for building production-ready *LLM applications*, *retrieval-augmented generative pipelines* and *state-of-the-art search systems* that work intelligently over large document collections.

Key concepts in Haystack

When we work with Haystack we have to keep in mind two fundamental concepts: **COMPONENTS** and **PIPELINES**

- **COMPONENTS**: Components are the building blocks of a pipeline. They perform tasks such as preprocessing, retrieving, or summarizing text while routing queries through different branches of a pipeline (https://docs.haystack.deepset.ai/docs/components).
- **PIPELINES**: Pipelines orchestrates the flow of data and operations. A pipeline in Haystack consists of different components, such as retrievers, readers, generators, and other modules, that work together to process queries and provide accurate, meaningful results (https://docs.haystack.deepset.ai/docs/pipelines).

To create the pipeline, the necessary components must first be added and then properly connected to each other to ensure seamless functionality. This integration ensures that data flows correctly between the components within the pipeline.
These two concepts will allow us to create a rag system.

Tech-Box:
How to install RAG Course's materials

- Go to https://downgit.evecalm.com/#/home
- Copy the following path and download the folder: https://github.com/Clearbox-AI/clearbox-ai-academy/tree/main/RAG_Course
- Open VSCode
- Open Terminal and write **pip install uv**
- Open RAG_Course folder
- In Terminal, in the RAG_Course folder, write **uv venv** → this will create a new virtual environment.
- Activate the virtual environment with **.venv\Scripts\activate** or **source .venv\Scripts\activate**
- Then, write **uv sync** OR **uv pip install -r pyproject.toml**
- Open the first notebook and select as kernel your virtual environment.

First Step:
Document Processing and Indexing

As we already said, when we create a RAG system, we have some documents that we want to retrieve from the database in order to generate a text consistent with this information.

In Haystack, this database is called **DOCUMENT STORE**.

Haystack allows you to work with several document stores.

In the first notebook (**1_indexing_search_pipeline.ipynb**), we use the **InMemoryDocumentStore**.

The **InMemoryDocumentStore** is a very simple document store with no extra services or dependencies. It is great for experimenting with Haystack, however we do not recommend using it for production.

Once we have our document store, we need to **prepare** our **documents to be indexed.**

The indexing process consists in organizing and structuring data in a way that enables efficient retrieval of relevant information from a database or document collection.

To prepare the documents, we need several components:

- **CONVERTER**: extract data from files in different formats and cast it into the unified document format.

Below there is a list of some available converters in Haystack.

AzureOCRDocumentConverter	Converts PDF (both searchable and image-only), JPEG, PNG, BMP, TIFF, DOCX, XLSX, PPTX, and HTML to documents.
CSVToDocument	Converts CSV files to documents.
DOCXToDocument	Convert DOCX files to documents.
HTMLToDocument	Converts HTML files to documents.
JSONConverter	Converts JSON files to text documents.
MarkdownToDocument	Converts markdown files to documents.
OpenAPIServiceToFunctions	Transforms OpenAPI service specifications into a format compatible with OpenAI's function calling mechanism.
OutputAdapter	Helps the output of one component fit into the input of another.
PDFMinerToDocument	Converts complex PDF files to documents using pdfminer arguments.
PPTXToDocument	Converts PPTX files to documents.
PyPDFToDocument	Converts PDF files to documents.
TikaDocumentConverter	Converts various file types to documents using Apache Tika.
TextFileToDocument	Converts text files to documents.
UnstructuredFileConverter	Converts text files and directories to a document.

- **CLEANER**: It removes extra whitespaces, empty lines, specified substrings, regexes, page headers, and footers in this particular order. This is useful for preparing the documents for further processing by LLMs. This component is used in the pipeline after the converter and before the splitter. The following parameters are boolean.
  - unicode_normalization normalizes Unicode characters to a standard form. The parameter can be set to NFC, NFKC, NFD, or NFKD.
  - ascii_only removes accents from characters and replaces them with their closest ASCII equivalents.
  - remove_empty_lines removes empty lines from the document.
  - remove_extra_whitespaces removes extra whitespaces from the document.
  - remove_repeated_substrings removes repeated substrings (headers/footers) from pages in the document. Pages in the text need to be separated by form feed character "\f", which is supported by TextFileToDocument and AzureOCRDocumentConverter.
  - In addition, you can specify a list of strings that should be removed from all documents as part of the cleaning with the parameter remove_substring.
  - You can also specify a regular expression with the parameter remove_regex and any matches will be removed.

- **SPLITTER:** divides a list of text documents into a list of shorter text documents. This is useful for long texts that otherwise wouldn't fit into the maximum text length of language models and can also speed up question answering.

- **DocumentSplitter** expects a list of documents as input and returns a list of documents with split texts. It splits each input document by split_by after split_length units with an overlap of split_overlap units. These additional parameters can be set when the component is initialized:
  - **split_by** can be "word", "sentence", "passage" (paragraph), "page" or "function".
  - **split_length** is an integer indicating the chunk size, which is the number of words, sentences, or passages.
  - **split_overlap** is an integer indicating the number of overlapping words, sentences, or passages between chunks.
  - **split_threshold** is an integer indicating the minimum number of words, sentences, or passages that the document fragment should have. If the fragment is below the threshold, it will be attached to the previous one.

To index the documents, we need several components:

- **EMBEDDER**: transform texts or documents into vector representations using pre-trained models. The embeddings produced are fixed-length vectors. They capture contextual information and semantic relationships within the text.

  - Embeddings in isolation are only used for information retrieval purposes (to do semantic search/vector search). You can use the embeddings in your pipeline for tasks like question answering.
  - In Haystack there are two types of Embedders: text and document.
    - Text Embedders work with text strings and are most often used at the beginning of query pipelines. They convert query text into vector embeddings.
    - Document Embedders embed Document objects and are most often used in indexing pipelines, after Converters, and before a DocumentWriter.
  - You must use the same embedding model for the query and the documents.
- **WRITER**: DocumentWriter writes a list of documents into a Document Store of your choice. It's typically used in an indexing pipeline as the final step after preprocessing documents and creating their embeddings.

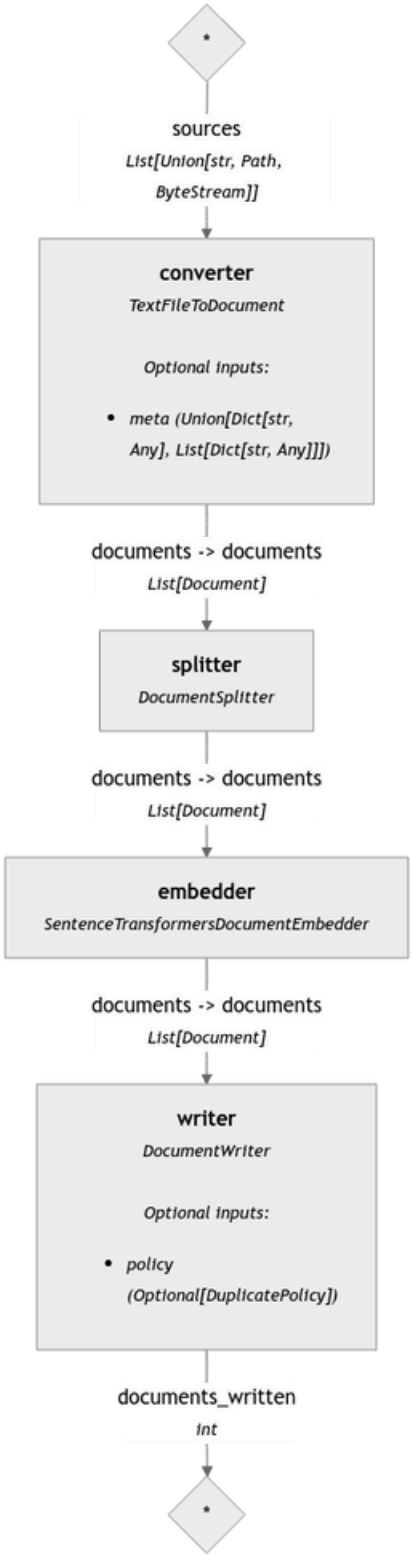


Image of a document processing and indexing pipeline.

TECH-BOX
What is a Word Embedding?

An embedding is a numerical representation of data (such as words, sentences, documents) in a continuous vector space, where similar items are placed closer together. In the context of natural language processing (NLP), embeddings are particularly used to represent words, phrases, or documents in a way that captures their semantic meaning and relationships.

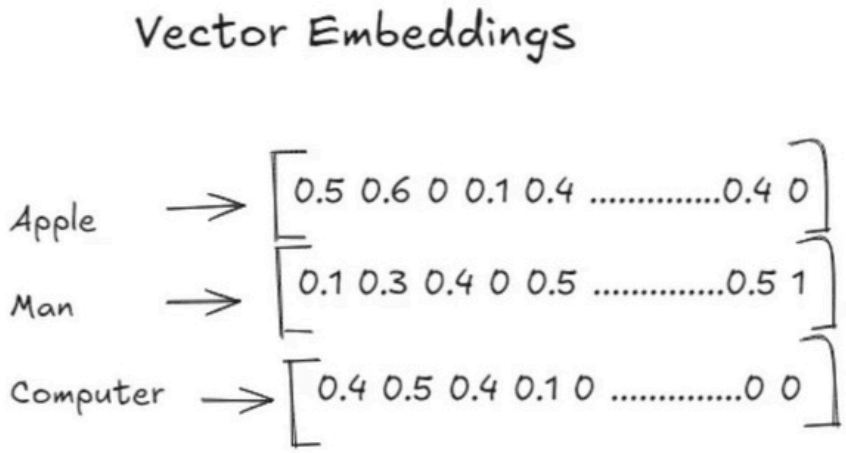


Image Source: https://pub.towardsai.net/a-complete-guide-to-embedding-for-nlp-generative-ai-llm-4a24301aba97

How Embeddings Work

Embeddings are created by mapping discrete data, like text, into high-dimensional vector spaces. Each item, such as a word or sentence, is represented as a point in this space, described by a vector of numbers. These vectors encode semantic and syntactic information about the data, allowing models to understand and manipulate language more effectively.

For example, in a word embedding space, the words “king“ and “queen“ may have
vectors that are close to each other because they share similar semantic meaning,
while words like “dog“ and “computer“ would be farther apart.

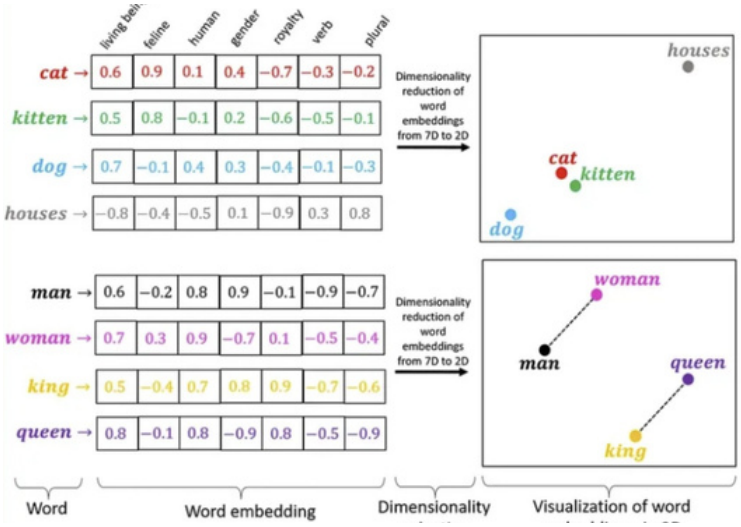


Image Source: https://ayselaydin.medium.com/9-understanding-word-embeddings-in-nlp-1c86a46f7942

Embeddings are used, in this context, because they allow to perform the Similarity Search. Embeddings enable tasks like retrieving documents or finding similar items by measuring distances between vectors, such as cosine similarity or Euclidean distance.

Types of Embeddings in NLP

1. **Word Embeddings**: Represent individual words in a vector space, e.g., Word2Vec, GloVe, FastText.
2. **Sentence Embeddings**: Represent entire sentences to capture higher-level meaning, e.g., Sentence-BERT.
3. **Document Embeddings**: Represent larger texts, like articles or documents, for broader context understanding.

Second Step: Retrieving

After the indexing process, we can create a document search pipeline. This pipeline takes as input a query, transforms it into an embedding with the SentenceTransformersTextEmbedder, and retrieves documents similar to the query from the document store.

How Do Retrievers Work?

Retrievers are the basic components of the majority of search systems. They are used in the retrieval part of the retrieval-augmented generation (RAG) pipelines; they're at the core of document retrieval pipelines.

When given a query, the Retriever sifts through the documents in the Document Store, assigns a score to each document to indicate how relevant it is to the query, and returns top candidates. It then passes the selected documents on to the next component in the pipeline or returns them as answers to the query.

Nevertheless, it's important to note that most Retrievers based on dense embedding do not compare each document with the query but use approximate techniques to achieve almost the same result with better performance.

Retriever Types

Depending on how they calculate the similarity between the query and the document, you can divide Retrievers into sparse keyword-base and dense embedding-based. Several Document Stores can be coupled with different types of Retrievers.

Sparse Keyword-Based Retrievers

The sparse keyword-based Retrievers look for keywords shared between the documents and the query using the BM251 algorithm or similar ones. This algorithm computes a weighted world overlap between the documents and the query. This method is simple and can be used when precise wording is required. It needs work with any language, but it doesn't take into account semantic aspects.

NB: This method DOES NOT require embedding.

[1]BM25 is a ranking function used by search engines to estimate the relevance of documents to a given search query.

Dense Embedding-Based Retrievers

Dense embedding-based Retrievers work with embeddings. Dense Retrievers need an Embedder first to turn the documents and the query into vectors. Then, they calculate the vector similarity of the query and each document in the Document Store to fetch the most relevant documents. This method is computational heavier than a sparse keyword-based method, but
takes into account semantic and syntax aspects of the document. They are language-specific.

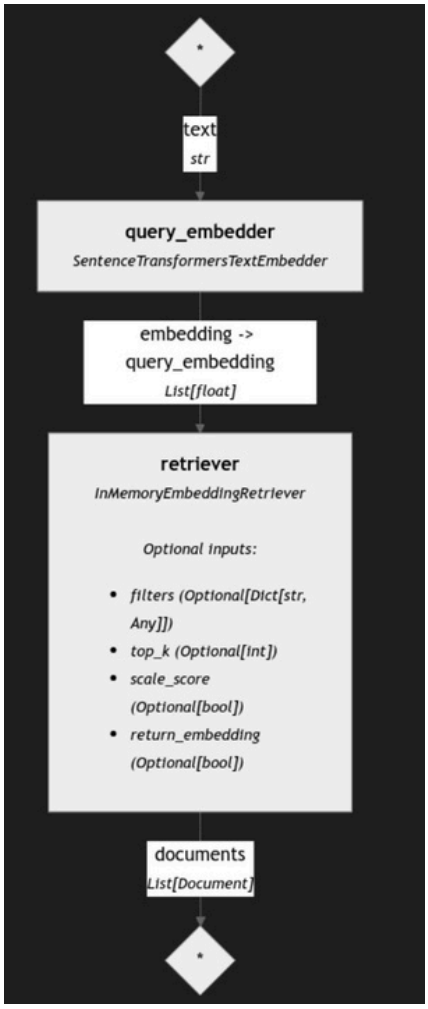


Image of a document search pipeline

Different Document Stores

In the first notebook we used a document store called the InMemoryDocumentStore, but this document store is not suitable for production.
In the second notebook, **2_different_doc_store_fetcher.ipnyb** we present two different document stores:

- **ElasticsearchDocumentStore**
- **QDrantDocumentStore**

Elasticsearch is a distributed, open-source search and analytics engine. It is designed for high-speed searching and indexing of large volumes of data, whether structured, semi-structured, or unstructured. Elasticsearch is highly scalable, allowing it to handle distributed clusters and large datasets efficiently.
It integrates seamlessly with tools like **Kibana** (for visualization) and **Logstash** (for data ingestion), forming part of the popular Elastic Stack.
While primarily used for text and log search, Elasticsearch also supports **vector-based similarity** searches, though it is not a dedicated vector database.
Common use cases include building search engines, log monitoring, real-time analytics, and powering recommendation systems.

Haystack allows you to use the ElasticsearchDocumentStore **locally**.

To do it, we will need to use Docker.

If you don't have Docker, follow the steps below:

- Download Docker Desktop https://www.docker.com/products/docker-desktop/
- Sign in or Register with an email address
- Open the app
- Go to VS Code
- Open the terminal and run the command docker-compose up.

Otherwise:

- Open the Docker Desktop
- Go to VS Code
- Open the terminal and run the command docker-compose up.

This will create a container (that you can see in the Docker desktop app) in which your Elastic Document Store is running and a Kibana dashboard in which you can have an overview of your document store.

PLEASE NOTE: This setup disables security for both Elasticsearch and Kibana, making it suitable for local development or testing. For production use, enable security by configuring authentication and SSL/TLS.

Another different document store is Qdrant.
Qdrant is an open-source, high-performance vector search engine and database designed for storing, searching, and managing embeddings and vectors generated by machine learning models.
To use **Qdrant**, you need to follow the following steps:

- Go to https://qdrant.tech/
- Log in or register
- In the Overview page, create your cluster. This could take some minutes.
- Once the cluster is created, you need to create an API token to access your cluster remotely. Copy the API token in a safe place and don't share it with anyone else.
- Now go to the Cluster section, click on the name of cluster and you obtain the cluster details.
- In Cluster details, you can find the link to add into your notebook to remotely reach your cluster. The link is the endpoint. At the end of the link, you have to add the port in which you'll find the cluster, that is :6333
- Now, you can access your cluster from your notebook.

REMIND: each document store has its proper Embedding and BM25 retrievers.

Fetcher

There are several ways to add documents to a document store. Until now, we added txt file directly from a folder in our laptop. But it is possible to directly **fetch documents from the web**.

Haystack provides a specific component to do that: **LinkContentFetcher**.

LinkContentFetcher fetches the contents of the urls you give it and returns a list of content streams. Each item in this list is the content of one link it successfully fetched in the form of a ByteStream object. Each of these objects in the returned list has metadata that contains its content type (in the content_type key) and its URL (in the url key).

It may happen that some sites block LinkContentFetcher from getting their content. In
that case, it logs the error and returns the ByteStream objects that it successfully
fetched.

Often, to use this component in a pipeline, you must convert the returned list of
ByteStream objects into a list of Document objects. To do so, you can use the
HTMLToDocument component.

You can use LinkContentFetcher at the beginning of an indexing pipeline to index the contents of URLs into a Document Store. You can also use it directly in a query pipeline, such as a retrieval-augmented generative (RAG) pipeline, to use the contents of a URL as the data source.

Third Step: Generating

The previous steps focused more on the indexing and retrieving part of a RAG system. This means that, as output of your pipeline, you got only the retrieved documents as they were stored in the document store.

Now, we will add a new step to our pipeline: **generation step**. (notebook:
3_build_customized_rag_rerank.ipynb)

The generation step involves taking the retrieved documents and creating a more comprehensive response using a generative model.

Haystack provides several generators. A comprehensive list of available generators is reported here.

When you work with generators, you need to create a **prompt**.

In generative AI, a prompt is the input provided to the AI system that directs or instructs it to produce a specific output.

Key Features of a Prompt in Generative AI

1. **Instruction**: It tells the AI what to do, such as generating text, creating an image, or completing a task.
2. **Context**: It provides the information the AI needs to produce a relevant response. In the RAG system, the retrieved documents are also part of the context.
3. **Constraints**: Prompts can include specific requirements or limitations to refine the output.

Haystack provides the **PromptBuilder** component. Renders a prompt filling in any variables so that it can send it to a Generator.
The prompt uses Jinja2 template syntax.
An example prompt for a RAG system can be the following one:

```
prompt = " " "
Answer the question based on the provided context.

Context:

{% for doc in documents %}

    {{ doc.content }}

{% endfor %}

Question: {{ query }}

" " "
```

In this prompt, you are saying to the model that it has to use the retrieved documents as context to make a response.
Now we can create the Generation pipeline.
The Generation Pipeline needs:

- A **query embedder** that embeds (= create a vector representation of) the query. The embedder used to embed the query **MUST BE THE SAME** that you use to embed your documents in the document store.
    - The query embedder and document embedder need to be the same because they must produce embeddings in the same vector space. This ensures that the query and document vectors can be meaningfully compared (e.g., via cosine similarity or dot product). If they use different models or embedding schemes, the generated vectors might not align, leading to poor or nonsensical matching results.
- A **retriever** to retrieve significant documents from the document store.
- A **prompt_builder** to construct the prompt to give instruction to the generator.
- A **generator** to create the response.

PLEASE NOTE: to use a generator, it could be that you need to create an API token. The creation of this token depends on the generator that you want to use.

TIPS: Best practises in prompt engineering

1. Be Clear and Specific

- **Define the Task Explicitly**: Clearly state what you want the AI to do. Avoid ambiguity.
  ✅ Good: "Summarize this article in 3 bullet points."
  ❌ Bad: "Summarize."
- **Include Context**: Provide any necessary background information or examples.
  ✅ Good: "Explain quantum computing to a high school student."
  ❌ Bad: "Explain quantum computing."

- **Set a Tone or Style**: Specify the tone or style if relevant.
  ✅ Good: "Write a formal email to a client explaining a project delay."
  ❌ Bad: "Write an email."

2. Provide Constraints or Rules

- **Set Word or Time Limits**: If the output should be brief or concise, state it explicitly.
  ✅ Good: "Write a summary of this article in 100 words or less."
  ❌ Bad: "Summarize this article."
- **Specify Output Format**: Indicate how you want the output structured.
  ✅ Good: "List the key benefits of renewable energy in a table format."
  ❌ Bad: "Explain the benefits of renewable energy."
- **Define Audience**: Tailor the response to a specific audience if needed.
  ✅ Good: "Explain machine learning to a 10-year-old."
  ❌ Bad: "Explain machine learning."

3. Use Examples (Few-shot Learning)

- **Provide Samples**: If you're asking for a specific style, tone, or format, include examples in the prompt.
  ✅ Good:

```
Write a joke in the style of a pun:
Example: Why don't skeletons fight each other? They don't have
the guts.
Now, write a new joke.
```

❌ Bad: “Write a pun.“

- **Demonstrate Output**: Show the AI how you expect the answer to look.

✅ Good:

```
Q: What is 2 + 2?

A: 4

Q: What is 3 + 5?

A:
```

4. Use Instructions Instead of Questions

- **Command-Based Prompts**: Framing your input as a command rather than a question often yields better results.

✅ Good: “List 3 reasons why solar energy is important.“
❌ Bad: “Why is solar energy important?“

5. Avoid Overloading the Prompt

- **Focus on One Task**: Don’t combine multiple tasks unless necessary. If combining tasks, break them into logical steps.

✅ Good: “First summarize the article in 100 words. Then provide 3 discussion points.”

❌ Bad: “Summarize and discuss the article.“

6. Handle Ambiguity

- **Resolve Vague Instructions**: Anticipate where the AI might misinterpret and clarify in the prompt.

  ✅ Good: "Generate a story about a cat who saves a village. Include a beginning, middle, and end."

  ❌ Bad: "Write a story about a cat."

7. Experiment and Iterate

- **Refine Through Testing**: Test variations of the prompt to see what produces the best results.

  1. If the output isn't what you want, adjust the wording, add more context, or simplify the instructions.

8. Use System Messages (Chat Models Only)

If you're using a chat-based AI model, include a system message to define the assistant's role and behavior.
Example:

```
System: You are a helpful assistant skilled in writing professional business emails.
User: Write an email apologizing for a delayed shipment.
```

9. Test Edge Cases

- **Clarify Uncertainties**: If there's potential for misunderstanding, clarify directly in the prompt.

  ✅ Good: "Write a poem about the ocean, focusing on its beauty and not its dangers."

  ❌ Bad: "Write a poem about the ocean."

Ranking

Rankers are a group of components that order documents by given criteria. Their goal is to improve your document retrieval results.

Haystack provides several rankers, that can be found here.

In this workshop we will use the **TransformersSimilarityRanker.**

TransformersSimilarityRanker ranks documents based on how similar they are to the
query. It uses a **pre-trained cross-encoder model** from the Hugging Face Hub to embed both the query and the documents. It then compares the embeddings to determine how similar they are. The result is a list of Document objects in ranked order, with the Documents most similar to the query appearing first.

T**ransformersSimilarityRanker** is most useful in query pipelines, such as a retrieval-augmented generation (RAG) pipeline or a document search pipeline, to ensure the retrieved documents are ordered by relevance. You can use it **after a Retriever** (such as the InMemoryEmbeddingRetriever) to improve the search results.

When using TransformersSimilarityRanker with a Retriever, consider **setting the Retriever's top_k to a small number**. This way, the Ranker will have fewer documents to process, which can help make your pipeline faster.

Ranker top_k

The Retriever's top_k specifies how many documents it returns. The Ranker then orders these documents.

You can set the same or a smaller top_k value for the Ranker. The Ranker's top_k is the number of documents it returns (if it's the last component in the pipeline) or forwards to the next component. In the pipeline example above, the Ranker is the last component, so the output you get when you run the pipeline are the top two documents, as for the Ranker's top_k.

Adjusting the top_k values can help you optimize performance. In this case, a smaller top_k value of the Retriever means fewer documents to process for the Ranker, which can speed up the pipeline.

TECH-BOX What is a Cross-Encoder Model

A **cross-encoder** is a type of neural network architecture, often used for tasks like sentence similarity, ranking, and re-ranking in NLP, that evaluates a pair of inputs (like a query and a document) together.
Unlike a bi-encoder, which encodes each input independently, **a cross-encoder combines both inputs before processing them**, allowing it to directly capture interactions between the inputs.

How Cross-Encoders Work:

1. **Input Concatenation**: The cross-encoder takes a pair of inputs, typically a query and a document, and concatenates them into a single input sequence. For example, with a BERT cross-encoder, you might format the inputs as: `[CLS] query [SEP] document [SEP]`.
2. **Joint Encoding**: The entire concatenated input sequence is fed into a transformer model (like BERT), which jointly processes the query and document. This joint processing lets the model learn interactions between specific words and phrases in both inputs.
3. **Relevance Scoring**: The cross-encoder outputs a score representing the relevance or similarity between the query and document. This score is often obtained from the `[CLS]` token's embedding, which summarizes the relationship between the query and document after processing.

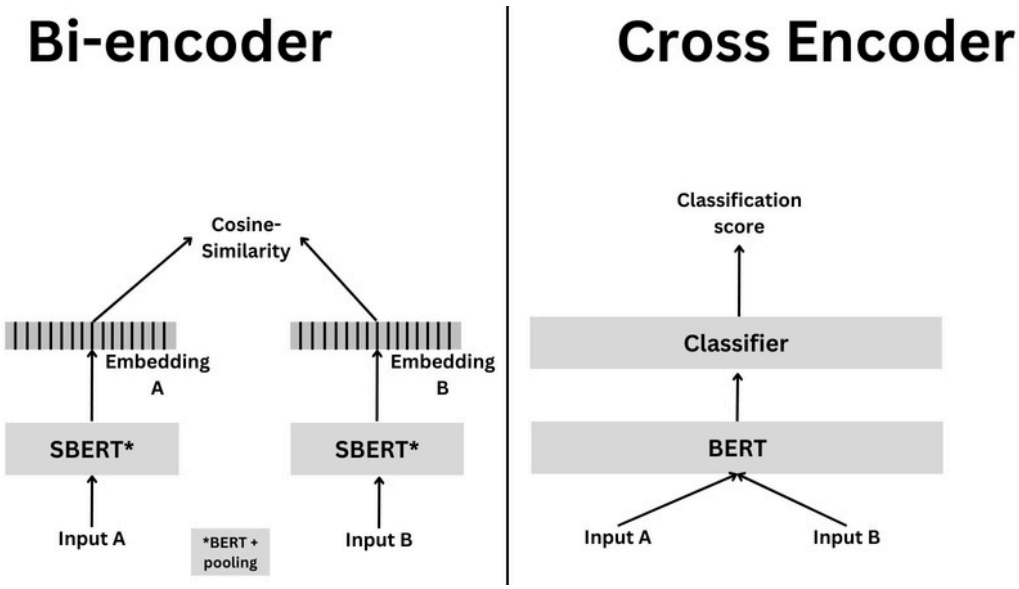


Image Source:
https://osanseviero.github.io/hackerllama/blog/posts/sentence_embeddings2/

Why Cross-Encoders Are Effective for Ranking:

- C**apturing Fine-Grained Interactions**: Cross-encoders look at each query-document pair as a single input, which allows them to capture fine-grained, contextual interactions. This is especially useful for ranking tasks, where subtle differences in wording or phrasing can determine relevance.
- **High Accuracy**: Since they jointly process the query and document, cross-encoders are generally more accurate for ranking tasks compared to bi-encoders, where each input is encoded independently.

Drawbacks of Cross-Encoders:

Computationally Intensive: Since each query-document pair must be processed together, cross-encoders are slower and more computationally expensive than bi-encoders, especially when working with a large number of documents. For this reason, they are usually applied to re-rank a smaller subset of documents that were first retrieved by a faster bi-encoder or BM25 retriever.

Limited Context

Routers & Branching

It could happen that the documents within the document store don't have enough information to reply to the user's query.

To address this problem, the model can be designed to retrieve information from the web. This requires creating a pipeline that first checks the retrieved documents for a response. If sufficient information is found, it provides an answer to the user.
If not, the pipeline searches the web for relevant information to generate an appropriate response.

In Haystack, this is possible thanks to the **Router** components (notebook **4_routers.ipynb**). The list of available routers in Haystack is available here.

One type of router is the **ConditionalRouter**.

ConditionalRouter routes your data through different paths down the pipeline by
evaluating the conditions that you specified.

To use ConditionalRouter you need to define a **list of routes**.

Each route is a **dictionary** with the following elements:

'**condition**': A Jinja2 string expression that determines if the route is selected.
'**output**': A Jinja2 expression defining the route's output value.

'**output_type**': The expected type of the output data (for example, str or List[int]).
This doesn't change the actual output type and is only needed to validate the connection with other components.
'**output_name**': The name under which the output value of the route is published. This name is used to connect the router to other components in the pipeline.

```
routes = [
        {
            "condition": "{{query|length > 10}}",
            "output": "{{query}}", "output_name":
            "ok_query", "output_type": str,

        }, {

            "condition": "{{query|length <= 10}}",
            "output": "{{query}}", "output_name":
            "too_short_query", "output_type": str,

        },
    ]
```

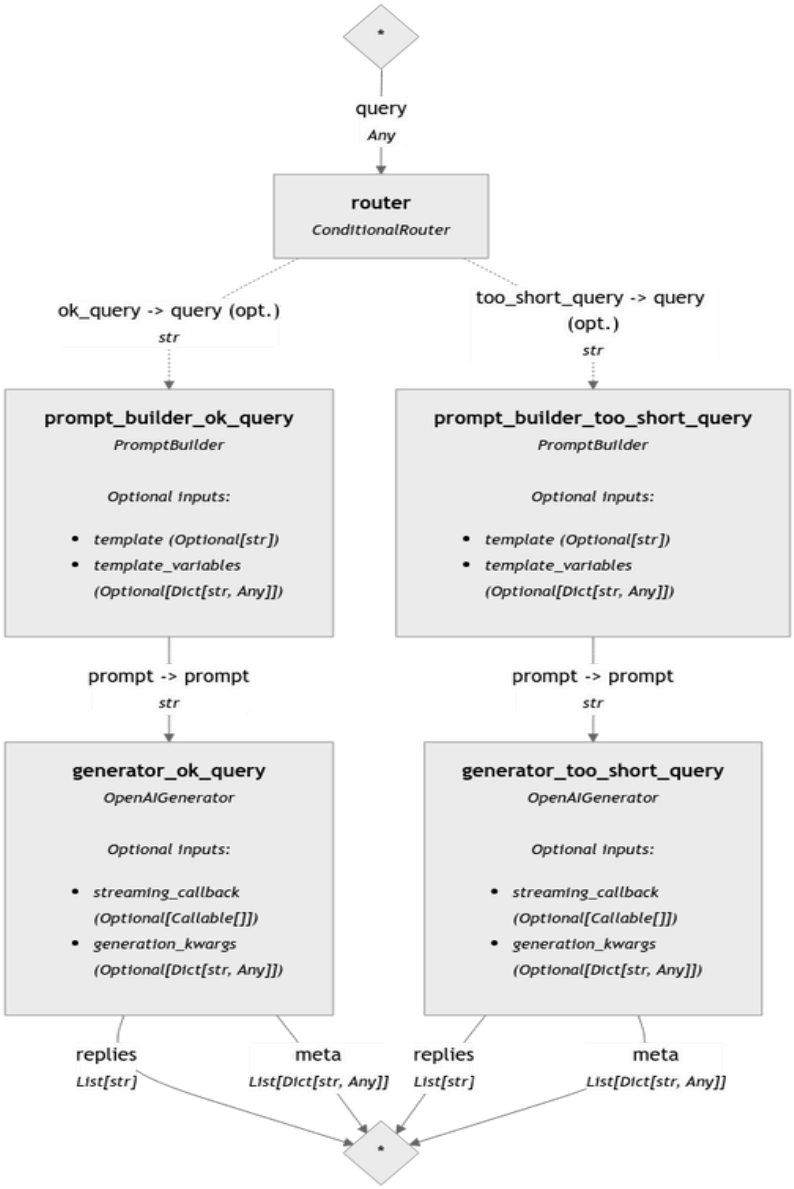
The above case shows two different routes based on two different conditions. The first condition is 'activated' if the length of the query is more than 10 characters otherwise the second one is activated.

This router points to two different branches of the pipeline:

- If the query is > 10 (in characters), then the query is passed to the branch in which there is a prompt builder for queries longer than 10 characters. In this case the prompt is "Answer the following query"

- If the query is <= 10, the query is passed to a different branch in which the is a different prompt builder specific for short query that is "Invent a max 6 words query based on the following short query and answer it."

Then, both branches have their generators that create the response.



If the document store lacks sufficient information to provide a good answer, we can use the ConditionalRouter to direct the pipeline to search the web for the required information (notebook: 5_fallbacks_branching.ipynb).

First of all, we need to create a prompt in which we specify that we want a specific token (in this case no_answer, if the model doesn't know how to reply).

PLEASE NOTE: Not all models act the same. In this guide and notebooks we use OpenAI's models. So, you need to create an OpenAI API Token.

The prompt could be like this:

```
rag_prompt_template = """
Answer the following query given the documents.
If the answer is not contained within the documents, reply with
'no_answer'
Query: {{query}}
Documents:
{% for document in documents %}
  {{document.content}}
{% endfor %}
"""
```

Then we create the usual rag pipeline with:

- query_embedder
- retriever
- prompt builder
- generator

If the model has enough information to provide a response, it will generate an answer. However, if it cannot find sufficient information, it will return **no_answer**. In such cases, the pipeline should be directed to search the web for additional information.
To do that we create the routes in the following way:

```
routes = [
    {
        "condition": "{{'no_answer' in replies[0]|lower}}",
        "output": "{{query}}", "output_name":
        "go_to_websearch", "output_type": str,

    }, {

        "condition": "{{'no_answer' not in replies[0]|lower}}",
        "output": "{{replies[0]}}", "output_name": "answer",
        "output_type": str,

    },
]
```

These routes show that:

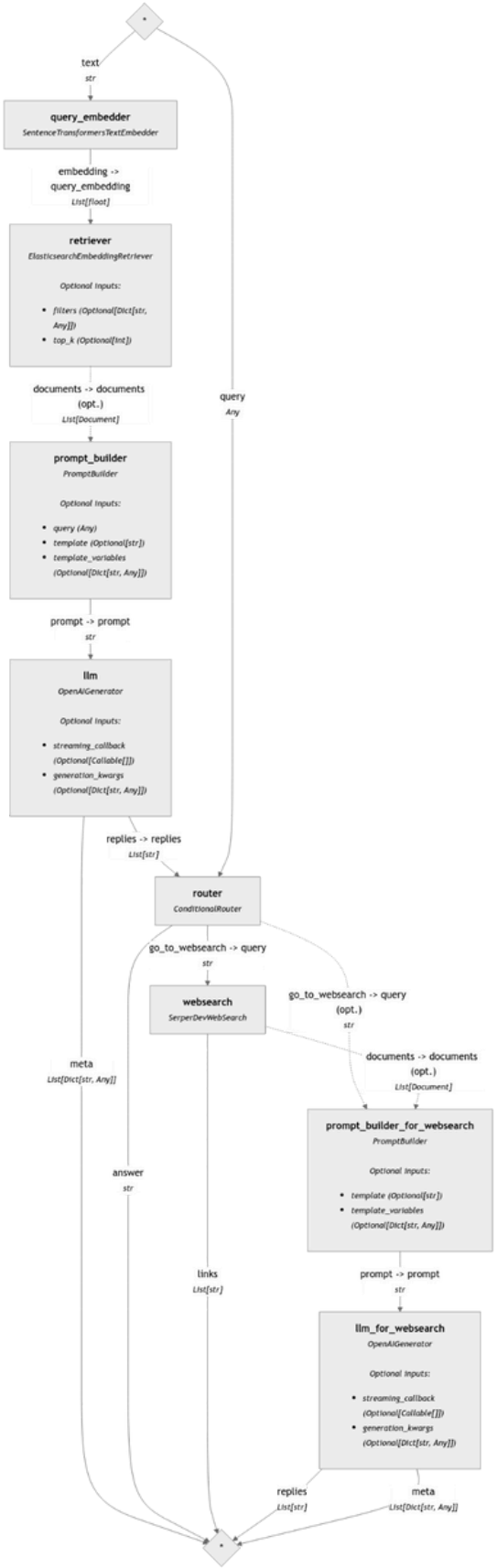
- If the text **no_answer** is NOT included in the response provided by the generator, then the pipeline provides as output the answer created by the generator.
- If the text **no_answer** is included in the response provided by the generator, then the pipeline has to continue through the **go_to_websearch** branch.

So, the ConditionalRouter component is added to the pipeline. To actually go to the web to do a search, we need to add a new component to the pipeline, the **SerperDevWebSearch**.

When you give SerperDevWebSearch a query, it returns a list of the URLs most relevant to your search. It uses page snippets (pieces of text displayed under the page title in search results) to find the answers, not the whole pages.

You need an API key to use it. You can create it by signing in here.

Once the pipeline retrieves the relevant information from the web, it passes them to a new prompt builder (added after the SerperDevWebSearch component) and a new generator will generate a response.


text
str
query_embedder
SentenceTransformersTextEmbedder
embedding -> query_embedding
List[float]
retriever
ElasticsearchEmbeddingRetriever
Optional inputs:
filters (Optional[Dict[str, Any]])
top_k (Optional[int])
documents -> documents (opt.)
List[Document]
query
Any
prompt_builder
PromptBuilder
Optional inputs:
query (Any)
template (Optional[str])
template_variables (Optional[Dict[str, Any]])
prompt -> prompt
str
llm
OpenAIGenerator
Optional inputs:
streaming_callback (Optional[Callable[]])
generation_kwargs (Optional[Dict[str, Any]])
replies -> replies
List[str]
router
ConditionalRouter
go_to_websearch -> query
str
websearch
SerperDevWebSearch
go_to_websearch -> query (opt.)
str
documents -> documents (opt.)
List[Document]
meta
List[Dict[str, Any]]
prompt_builder_for_websearch
PromptBuilder
Optional inputs:
template (Optional[str])
template_variables (Optional[Dict[str, Any]])
answer
str
links
List[str]
prompt -> prompt
str
llm_for_websearch
OpenAIGenerator
Optional inputs:
streaming_callback (Optional[Callable[]])
generation_kwargs (Optional[Dict[str, Any]])
replies
List[str]
meta
List[Dict[str, Any]]

Conversational RAG

Since this point, we work with a linear pipeline. This means that each query is independent from the other ones and there is no memory that saves all the previous query.

To better understand:

Linear Pipeline

In a **linear pipeline**, each query is processed independently, without considering the context or content of previous queries. This means:

- The system treats every query as a standalone request.
- There is no memory or mechanism to link queries together.
- Pronouns or references in follow-up queries (e.g., "he," "it") lack context, which can lead to misunderstandings or irrelevant results.

Example:

1. First query: "Who is Tolkien?"
    - The system retrieves documents related to J.R.R. Tolkien.

2. Second query: "What did he write?"
    - The retriever doesn't associate "he" with Tolkien because it doesn't remember the first query. This can lead to retrieving irrelevant or incomplete results.

Adding Query History for Context

To address this limitation, a system can maintain query history or a conversation memory. This allows:

- Storing information from previous queries.
- Linking related queries by carrying over context.
- Ensuring references (e.g., "he") are correctly resolved to earlier entities (e.g., Tolkien).

Example with Query History:

1. First query: "Who is Tolkien?"
    - The system identifies "Tolkien" as J.R.R. Tolkien and stores this context.
1. Second query: "What did he write?"
    - The system uses the stored context to understand that "he" refers to Tolkien, ensuring the retriever focuses on relevant documents about Tolkien's writings.

To create a conversational RAG system (**6_conversational_rag.ipynb**), we need to use as generator a ChatGenerator component.

The differences between Generator and **ChatGenerator** are:

Generators

- **Purpose**: Generators create text based on a single input prompt, often treating each request independently.
- **Use Case**: They are designed for tasks like summarization, story generation, code writing, or Q&A where there is no conversational flow.
- **Context Handling**: They process inputs in isolation and do not retain memory of previous prompts or outputs.

Chat Generators

- **Purpose**: Chat generators are specifically designed for multi-turn conversational tasks, where responses depend on both the current query and the history of the conversation.
- **Use Case**: They excel in scenarios requiring context retention, such as customer support, interactive dialogue, or long conversations.
- **Context Handling**: Chat generators maintain conversation memory, linking past exchanges to produce coherent responses. They manage pronouns, references, and evolving queries effectively.

Haystack provides several ChatGenerator for different models.

When you use a ChatGenerator, there are different messages coming from different roles:

- System
- Assistant
- User

1. System Role

- Purpose: The system role provides overarching instructions or guidelines that shape the assistant's behavior throughout the conversation.
- What it Does:
  - Sets the tone, style, or domain-specific constraints.
  - Instructs the assistant on how to respond.
  - Remains constant for the duration of the interaction.

Example:

```
System: "You are a helpful assistant specialising
in technology-related queries. Answer concisely
and provide examples when possible."
```

2. User Role

- Purpose: Represents the input or queries provided by the user interacting with the system.
- What it Does:
  - Provides the context or question to which the assistant must respond.
  - Can be a single query or a series of follow-ups.

Example:

```
User: "What is machine learning?"
```

3. Assistant Role

- **Purpose**: Represents the response generated by the ChatGenerator based on the user's input and the system instructions.
- **What it Does**:
  - Delivers information, solutions, or guidance as per the system's instructions.
  - Maintains the conversation context for multi-turn dialogues.

Example:

```
Assistant: artificial intelligence that focuses on
training computers to learn from data and make
predictions or decisions without explicit
programming."
```

We also need a different PromptBuilder, that in this case it will be a **ChatPromptBuilder**.

The **ChatPromptBuilder** component creates prompts using static or dynamic templates written in Jinja2 syntax, by processing a list of chat messages. The templates contain placeholders like **{{ variable }}** that are filled with values provided during runtime.

To use it, start by providing a list of ChatMessage objects as the template. ChatMessage is a data class that includes message content, a role (who generated the message, such as user, assistant, system), and optional metadata.

The builder looks for placeholders in the template and identifies the required variables. You can also list these variables manually. During runtime, the run method takes the template and the variables, fills in the placeholders, and returns the completed prompt. If required variables are missing and/or if the template is invalid, the builder raises an error.

Prompt Template for Rephrasing User Query

In conversational systems, simply injecting memory into the prompt is not enough to perform RAG effectively. There needs to be a mechanism to **rephrase the user's query** based on the conversation history to **ensure relevant documents are retrieved**. For instance, if the first user query is "*What's the first name of Einstein?*" and the second query is "*Where was he born?*", the system should understand that "he" refers to Einstein. The rephrasing mechanism should then modify the second query to "*Where was Einstein born?*" to retrieve the correct documents.

We can use an LLM to rephrase the user's query.

Now, let's incorporate query rephrasing into our pipeline by adding a new PromptBuilder, an OpenAIGenerator, and an OutputAdapter. The OpenAIGenerator will rephrase the user's query for search, and the OutputAdapter will convert the output from the OpenAIGenerator into the input for the ElasticEmbedderRetriever.

In our case, the OutputAdapter adapts the output of the generator (the new query) in a str to be passed as text to the embedder.

Creating Custom Components

With Haystack, you can easily create any custom components for various tasks (7_custom_component.ipynb).

Requirements

Here are the requirements for all custom components:

- **@component**: This decorator marks a class as a component, allowing it to be used in a pipeline.

- **run()**: This is a required method in every component. It accepts input arguments and returns a dict. The inputs can either come from the pipeline when it's executed, or from the output of another component when connected using connect(). The run() method should be compatible with the input/output definitions declared for the component.

- **Inputs and Outputs** :

Next, define the inputs and outputs for your component.

Inputs

You can choose between three input options:

- **set_input_type**: This method defines or updates a single input socket for a component instance. It's ideal for adding or modifying a specific input at runtime without affecting others. Use this when you need to dynamically set or modify a single input based on specific conditions.
- **set_input_types**: This method allows you to define multiple input sockets at once, replacing any existing inputs. It's useful when you know all the inputs the component will need and want to configure them in bulk. Use this when you want to define multiple inputs during initialization.

- Declaring arguments directly in the run() method. Use this method when the component's inputs are static and known at the time of class definition.

Outputs

You can choose between two output options:

- **@component.output_types**: This decorator defines the output types and names at the time of class definition. The output names and types must match the dict returned by the run() method. Use this when the output types are static and known in advance. This decorator is cleaner and more readable for static components.
- **set_output_types**: This method defines or updates multiple output sockets for a component instance at runtime. It's useful when you need flexibility in configuring outputs dynamically. Use this when the output types need to be set at runtime for greater flexibility.

Query Expansion

Query expansion is the process of enhancing a user's search query by adding additional terms or phrases to improve retrieval performance and ensure more relevant results (7_query_expansion.ipynb).

How Query Expansion Works

1. Analyzing the Original Query:

- The system examines the user's input to identify the main concepts or keywords.

2. Adding Related Terms:

- Synonyms, related words, or alternate spellings are added to the query to broaden its scope.

- For example, expanding “car“ to include “automobile“ or “vehicle.“

3. Reformulating the Query:

- The expanded query is reformulated to match a wider range of documents or data points.

Benefits of Query Expansion

- **Improves Recall**: Retrieves a broader set of relevant documents.
- **Handles Ambiguity**: Reduces errors caused by ambiguous or incomplete queries.
- **Enhances Synonym Matching**: Accounts for different ways users may refer to the same concept.

Example

Original Query: **“solar power“**
Expanded Query: `"solar power" or "solar energy" or "renewable energy"`

Query expansion makes retrieval systems more robust and capable of handling
diverse search behaviors and language variations.

In Haystack, it is necessary to create two new components:

- **QueryExpander** to create a number of additional queries, similar to the original user query. There is an LLM that does this expansion. So the component has a prompt in which you can set the logic for the expansion.

- **MultiQueryElasticEmbeddingRetriever** to use embedding retrieval for each (expanded) query in turn. This component also handles the same document being retrieved for multiple queries and will not return duplicates.
- PLEASE NOTE: in this case we create a **MultiQueryElasticEmbeddingRetriever** because our document store is an ElasticDocument Store and we need to use the proper embedding retriever to work in the same vector space. If you use a different document store you also need to rewrite this component accordingly.

Logging

Logging is an essential tool for understanding the behavior of an application, identifying issues, and capturing insights about its operations (**8_logging.ipynb**).

Logging Setup

First, the logging configuration is cleared by removing any existing handlers attached to the root logger. This ensures a clean slate, preventing duplicate or conflicting logging outputs.

Next, the **logging.basicConfig** function is used to set up a custom logging configuration. This function specifies:

1. **Logging Level**: **level=logging.DEBUG** ensures that all events, from debugging information to critical errors, are captured.

2. **Log Format**: The format includes the timestamp, log level (e.g., DEBUG, WARNING), logger name, and the log message.

3. Handlers:

- **FileHandler**: Writes logs to a file named **log.txt**. The mode **w** ensures the log file is overwritten each time the program runs, keeping it clean and relevant to the current session.
- **StreamHandler**: Outputs logs to the console, allowing real-time monitoring during development.

The result is a dual-output logging system: logs are visible in the console for immediate debugging and saved to a file for later analysis.
Haystack provides its own logging configuration, which is applied using `haystack.logging.configure_logging(use_json=True)`

This enables JSON-formatted logging for structured output, making it easier to analyze logs programmatically or with logging tools.
A logger specific to Haystack is retrieved with **logging.getLogger("haystack")**.
The logging level for this logger is set to **DEBUG**, ensuring that all events within the Haystack framework are captured. For additional clarity, tracing is set up using **haystack.tracing**. Tracing logs the
flow of data between pipeline components, and the **LoggingTracer** is initialized with color-coded tags to visually distinguish component input and output logs in the console.
A manual warning log is added with **logging.warning**, producing a general warning message. Similarly, the Haystack logger logs a debug message, demonstrating how specific loggers can be used to differentiate between general and framework-specific logs.

At the end of execution, the logs are saved to **log.txt**. The message "Log entries saved to log.json" is printed, reflecting that logging has successfully captured the relevant events throughout the pipeline's execution.

Monitoring

Langfuse is an open-source LLM engineering platform that helps teams collaboratively debug, analyze, and iterate on their LLM applications. All platform features are natively integrated to accelerate the development workflow (9_monitoring_pipeline.ipynb).

LangfuseConnector component allows you to easily trace your Haystack pipelines with the Langfuse UI.

Simply install the component with pip install langfuse-haystack, then add it to your pipeline.

Create an account on Langfuse.

Langfuse captures detailed information about pipeline runs, like API calls, context data, prompts, and more. Use this component to:

- Monitor model performance, such as token usage and cost.
- Find areas for pipeline improvement by identifying low-quality outputs and collecting user feedback.
- Create datasets for fine-tuning and testing from your pipeline executions.

To work with the integration, add the LangfuseConnector to your pipeline, run the pipeline, and then view the tracing data on the Langfuse website.
Don't connect this component to any other – LangfuseConnector will simply run in your pipeline's background.

These are the things that you need before working with LangfuseConnector:

Make sure you have an active Langfuse account.

- Set the **HAYSTACK_CONTENT_TRACING_ENABLED** environment variable to true – this will enable tracing in your pipelines.
- Set the **LANGFUSE_SECRET_KEY** and **LANGFUSE_PUBLIC_KEY** environment variables with your Langfuse secret and public keys found in your account profile.

```
os.environ["LANGFUSE_HOST"] = "https://cloud.langfuse.com"
os.environ["HAYSTACK_CONTENT_TRACING_ENABLED"] = "True"
os.environ["TOKENIZERS_PARALLELISM"] = "True"

LANGFUSE_SK_TOKEN = os.getenv("LANGFUSE_SK_TOKEN")
LANGFUSE_PK_TOKEN = os.getenv("LANGFUSE_PK_TOKEN")
```

Evaluation

Evaluation measures performance using metrics providing a clear picture of your pipeline's strengths and weaknesses using LLMs or ground-truth labels. Evaluating RAG systems can help understand performance bottlenecks and optimize one component at a time, for example, a Retriever or a prompt used with a Generator.

1. Human Evaluation

As the first step, perform manual evaluation. Test a few queries (5-10 queries) and manually assess the accuracy, relevance, coherence, format, and overall quality of your pipeline's output.

This will provide an initial understanding of how well your system performs and highlight any obvious issues.

To trace the data through each pipeline step, debug the intermediate components using the **include_outputs_from** parameter. This feature is particularly useful for observing the retrieved documents or verifying the rendered prompt. By examining these intermediate outputs, you can pinpoint where issues may arise and identify specific areas for improvement, such as tweaking the prompt or trying out different models.

2. Metrics

Haystack provides a wide range of Evaluators which can perform 2 types of evaluations:

- **Model-Based evaluation**
- **Statistical evaluation**

Model-Based evaluation

Model-based evaluation in Haystack uses a language model to check the results of a Pipeline. This method is easy to use because it usually doesn't need labels for the outputs. It's often used with Retrieval-Augmented Generative (RAG) Pipelines, but can work with any Pipeline.

Using LLMs for Evaluation

A common strategy for model-based evaluation involves using a Language Model (LLM), such as OpenAI's GPT models, as the evaluator model, often referred to as the golden model. The most frequently used golden model is GPT-4. We utilize this model to evaluate a RAG Pipeline by providing it with the Pipeline's results and sometimes additional information, along with a prompt that outlines the evaluation criteria.

This method of using an LLM as the evaluator is very flexible as it exposes a number of metrics to you. Each of these metrics is ultimately a well-crafted prompt describing to the LLM how to evaluate and score results.

Common metrics are faithfulness and context relevance, and so on.

Model-Based Evaluation Pipelines in Haystack

There are two ways of performing model-based evaluation in Haystack, both of which leverage Pipelines and Evaluator components.

- You can create and run an evaluation Pipeline independently. This means you'll have to provide the required inputs to the evaluation Pipeline manually. We recommend this way because the separation of your RAG Pipeline and your evaluation Pipeline allows you to store the results of your RAG Pipeline and try out different evaluation metrics afterward without needing to re-run your RAG Pipeline every time.
- As another option, you can add an evaluator component to the end of a RAG Pipeline. This means you run both a RAG Pipeline and evaluation on top of it in a single pipeline.run() call.

Model-based Evaluation of Retrieved Documents

ContextRelevanceEvaluator

Context relevance refers to how relevant the retrieved documents are to the query. An LLM is used to judge that aspect. It first extracts statements from the documents and then checks how many of them are relevant for answering the query.

You can use the **ContextRelevanceEvaluator** component to evaluate documents retrieved without ground truth labels. The component breaks up the context into multiple statements and checks whether each statement is relevant for answering a question. The final score for the context relevance is a number from 0.0 to 1.0 and represents the proportion of statements that are relevant to the provided question.

The default model for this Evaluator is gpt-4o-mini. You can override the model using the api_params key during initialization.

Model-based Evaluation of Generated or Extracted Answers

FaithfulnessEvaluator

Faithfulness, also called groundedness, evaluates to what extent a generated answer is based on retrieved documents. An LLM is used to extract statements from the answer and check the faithfulness for each separately. If the answer is not based on the documents, the answer, or at least parts of it, is called a hallucination.

A higher faithfulness score is better, and it indicates that a larger number of statements in the generated answers can be inferred from the contexts. The faithfulness score can be used to better understand how often and when the Generator in a RAG pipeline hallucinates.

SASEvaluator (Semantic Answer Similarity)

The **SASEvaluator** evaluates answers predicted by Haystack pipelines using ground truth labels. It checks the semantic similarity of a predicted answer and the ground truth answer using a transformers-based model. This metric is called semantic answer similarity.

Note that only one predicted answer is compared to one ground truth answer at a time. The component does not support multiple ground truth answers for the same question or multiple answers predicted for the same question.

Statistical Evaluation

Statistical evaluation in Haystack compares ground truth labels with pipeline predictions, typically using metrics such as precision or recall. It's often used to evaluate the Retriever component within Retrieval-Augmented Generative (RAG) pipelines, but this methodology can be adapted for any pipeline if ground truth labels of relevant documents are available.

When evaluating answers, the ground truth labels of expected answers are compared to the pipeline's predictions.

For assessing answers generated by LLMs with one of Haystack's Generator components, we recommend model-based evaluation instead.

Statistical Evaluation Pipelines in Haystack

Statistical Evaluation of Retrieved Documents

DocumentRecallEvaluator

Recall measures how often the **correct document** was **among the retrieved documents** over a set of queries. For a single query, the output is binary: either the correct document is contained in the selection, or it is not. Over the entire dataset, the recall score amounts to a number between zero (no query retrieved the right document) and one (all queries retrieved the right documents).

In some scenarios, there can be multiple correct documents for one query. The metric **recall_single_hit** considers whether **at least one** of the correct documents is retrieved, whereas **recall_multi_hi**t takes into account **how many** of the multiple correct documents for one query are retrieved.

Note that recall is affected by the number of documents that the Retriever returns. If the Retriever returns few documents, it means that it is difficult to retrieve the correct documents. Make sure to set the Retriever's top_k to an appropriate value in the pipeline that you're evaluating.

DocumentMRREvaluator (Mean Reciprocal Rank) In contrast to the recall metric, mean reciprocal rank takes the position of the top correctly retrieved document (the "rank") into account. It does this to account for the fact that a query elicits multiple responses of varying relevance. Like recall, MRR can be a value between zero (no matches) and one (the system retrieved a correct document for all queries as the top result).

DocumentMAPEvaluator (Mean Average Precision)

Mean average precision is similar to mean reciprocal rank but takes into account the position of every correctly retrieved document. Like MRR, mAP can be a value between zero (no matches) and one (the system retrieved correct documents for all top results). mAP is particularly useful in cases where there is more than one correct answer to be retrieved.

Statistical Evaluation of Extracted or Generated Answers

AnswerExactMatchEvaluator Exact match measures the proportion of cases where the predicted Answer is identical to the correct Answer. For example, for the annotated question-answer pair "What is Haystack?" + "A question answering library in Python", even a predicted answer like "A Python question answering library" would yield a zero score because it does not match the expected answer 100%.

Recap Table:

valuator Components

Evaluator	Evaluates Answers or Documents	Model-based or Statistical	Requires Labels
AnswerExactMatchEvaluator	Answers	Statistical	Yes
ContextRelevanceEvaluator	Documents	Model-based	No
DocumentMRREvaluator	Documents	Statistical	Yes
DocumentMAPEvaluator	Documents	Statistical	Yes
DocumentRecallEvaluator	Documents	Statistical	Yes
FaithfulnessEvaluator	Answers	Model-based	No
LLMEvaluator	User-defined	Model-based	No
SASEvaluator	Answers	Model-based	Yes

In the notebook 9_evaluation_rag_pipeline.ipynb, we create a separate evaluation pipeline. Here we added only two evaluation metrics because we don't have groundtruth labels.
It is possible to visualise in a pandas dataframe the overall average score for all the queries, or the score for each query.

To improve your pipeline after evaluation you can make some enhancements on both retrieval and generation:

- Retrieval Improvements: Ensure data quality with cleaning and structured metadata, utilize metadata filtering, explore different embedding models, and adopt advanced techniques like hybrid retrieval and sparse embeddings.

- Generation Improvements: Apply ranking mechanisms to prioritize relevant documents, reorder by similarity or diversity to enhance response accuracy, and address issues like the "Lost in the Middle" problem. Experiment with various language models and optimize prompts with detailed instructions or few-shot examples for better results.

humans && machines

C.so Castelfidardo,30/a
10129, Torino (Italy)
info@clearbox.ai

VAT ID: (IT)12161430017

clearbox.ai