Alex JIANG

Okayama University
4-chome-7-51 Tsushimahigashi, Kita Ward, Okayama, 700-0082

2nd year engineering internship report: Development of Machine Learning Model(s) for Analyzing Log Data

From June 3, 2024 to August 27, 2024

supervised by Dr. NOGAMI, Dr. Kodera, Dr. Huda

Contents

Table des figures

1 Acknowledgements

I would like to express my deep and sincere gratitude to my internship supervisor, Professor KODERA Yuta, for welcoming me in his laboratory during my stay in Japan. I am grateful for all his support and help on my project, allowing me to discover more about how LLM and RAG really work. I really enjoyed working on my topic of research. Although, I did many mistakes, he was always here to calmly accept them and try to find a solution. I was also able to work on a Jetson and on the school server to execute the RAG Application.

I would like to express my gratitude also to ISOKAWA Tanoshimaru and KAWADA Yuta, taking their precious time showing me how to setup a Jetson, helping me using Docker and the school server. Also, to Shamin and Bisri, in PhD course, for taking their time to give me some dataset, and also discuss about Deep Learning in general. I am extremely grateful to all my laboratory mates I cannot name here. Everyone was very warm-welcoming, and made me discover parts of Japanese culture, in addition to be very altruistic, especially when I could not find what I needed. I really had a nice time, and I hope to stay in touch with them.

Then, I particularly want to express my deepest gratitude to Marianne GENTON who allowed to make contact with Okayama University and helped me to deal with all of the administrative procedure. Last but not the least, thanks to Professor Matthieu CHABANAS, former teacher at ENSIMAG, who recommended me to Okayama University.

2 Introduction

In recent years, the field of natural language processing (NLP) has witnessed many advancements, largely driven by the development and deployment of large language models (LLMs) such as ChatGPT by OpenAI. These models, characterized by their massive scale and deep architectures, have demonstrated remarkable capabilities in generating human-like text, solving complex tasks, and so on. Among the various applications of LLMs, Retrieval-Augmented Generation (RAG) stands out as a particularly promising approach, combining the strengths of retrieval-based methods and generative models to enhance the quality and relevance of generated content.

Retrieval-Augmented Generation leverages the vast amounts of information available in large external databases or knowledge repositories to inform and improve the generation process. By integrating retrieval mechanisms into the generative model, RAG models can access and utilize external knowledge, enabling them to produce more accurate, contextually relevant, and informative responses. This approach addresses some of the limitations of purely generative models pre-trained from time to time, such as the tendency to produce hallucination and the challenge of maintaining coherence over the time.

By using these two complex architectures, the main goal is to create an application capable to analyse log data especially for cybersecurity. Indeed, being capable to detect more or less some issues encounter in the log data or even creating statistical report about how most of the time, attackers proceed can be very useful. This is why, by mixing these two technologies, it might be possible to create this kind of application.

This paper aims to explore the advancements, methodologies, and application of Retrieval-Augmented Generation. It is really important to focus on how LLM to understand why RAG is useful. Some words or techniques are also used only in RAG application, they will be define during the whole explanation. First, the paper addresses some basic but important knowledge about LLMs in general, including their architecture, training processes, and key features. Next, it explains the fundamentals needed to understand how a basic Retrieval-Augmented Generation application works and how to implement it. Additionally, the paper explores various methodologies to improve RAG applications.

Finally, it presents a real-world application involving SSH logs, demonstrating how advanced RAG methods can be used to solve complex tasks. The goal being **analyzing Ssh Log Data** using a machine learning model, especially to produce a statistical report.

3 Large Language Model(s) (LLMs)

Nowadays, Large Language Models (LLMs) are widely used in various domains of Deep Learning such as **sentiment analysis**, **text-to-text generation**, **image generation**, and so on. LLMs are large, pre-trained models containing billions of parameters. They can solve complex tasks by breaking them down into smaller ones. A Large Language Model consists of three core concepts known as **generative**, **pre-trained**, and **transformer** (Figure 1)[13].

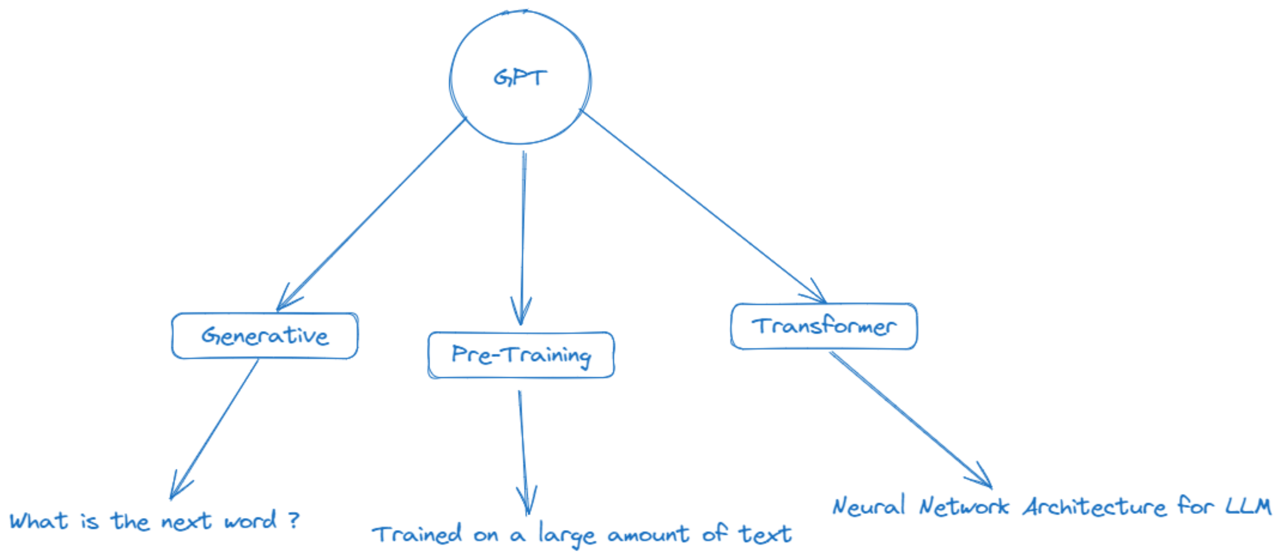

GPT
Generative
Pre-Training
Transformer
What is the next word ?
Trained on a large amount of text
Neural Network Architecture for LLM

Figure 1: Basic LLM Components

3.1 Generative

An LLM is capable of predicting the next word in a sequence. To generate text, it recursively predicts the next word based on the sentence generated in the previous step of recursion (Figure 2. Within the neural network, the final vector of the last layer, called **Logits**, is transformed into **Probabilities**. Finally, the next word is randomly sampled based on these probabilities [2]. The generation process stops either when all probabilities are too low to be relevant for text generation or when the maximum number of words is reached.

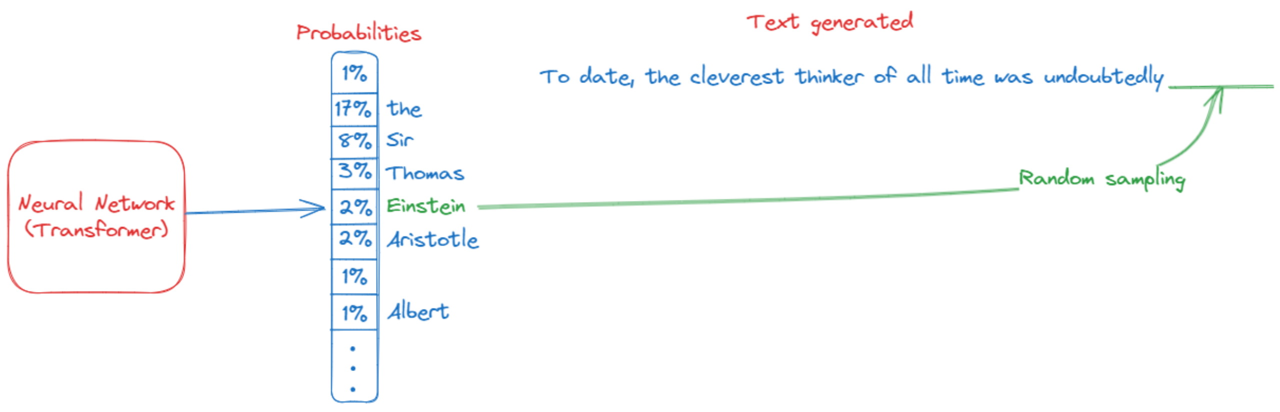


Figure 2: Generation in LLM

3.2 Pre-Training

LLMs are advanced versions of pre-trained language models (PLMs), distinguished by their larger parameter counts (often exceeding one billion). The primary objective is to train these models on vast datasets to enable them to generate text by predicting the subsequent words. These datasets are typically sourced from extensive collections of text found on the internet, in books, research papers, and other sources. Over time, the model becomes proficient at predicting the next word in any given sequence. This training process is known as *self-supervised learning*.

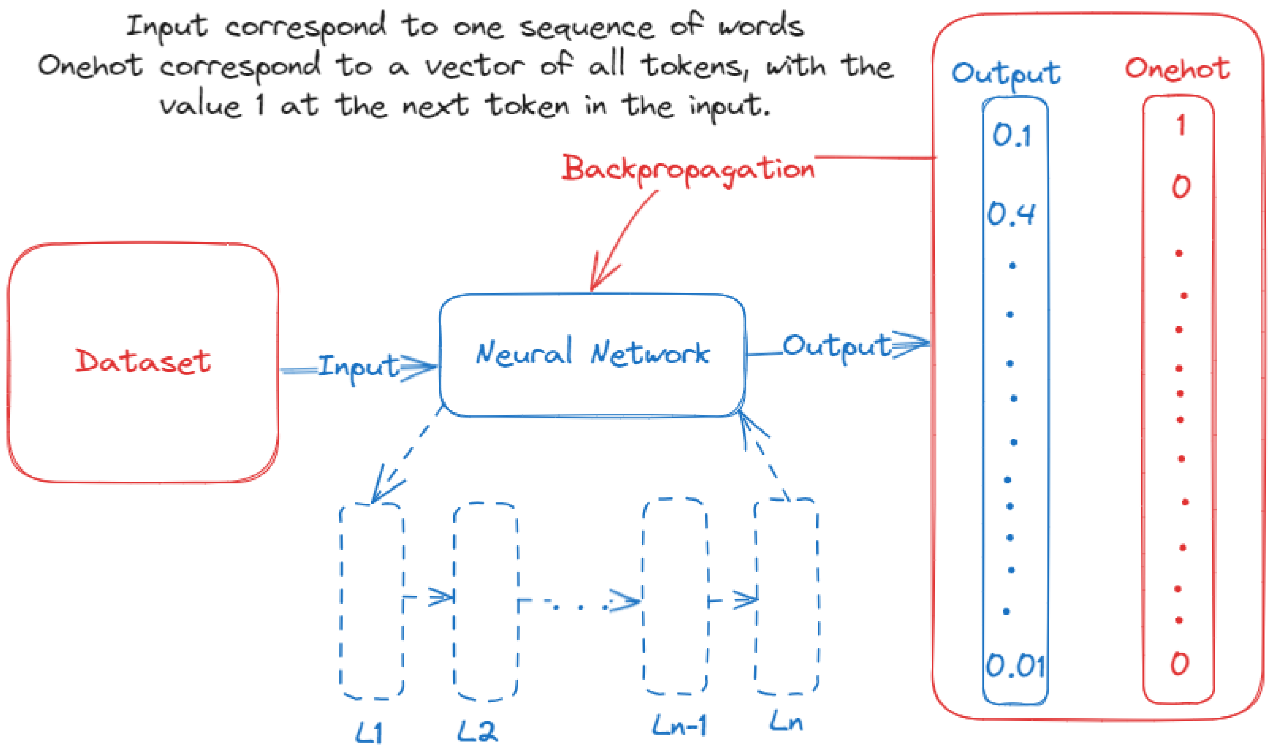

Input correspond to one sequence of words
Onehot correspond to a vector of all tokens, with the value 1 at the next token in the input.
Backpropagation
Dataset
Input
Neural Network
Output
Output
Onehot
0.1
0.4
0.01
1
0
0
L1
L2
Ln-1
Ln

Figure 3: Basic Training

However, during pre-training, several challenges arise. Models like ChatGPT are quite good to solve difficult tasks such as **Question-Answering** or even **Summarizing**. Yet, training only to predict the next word doesn't allow a model to perform these complex tasks. For instance, after the generation: "How much does an engineer earn ?", it's possible for the LLM to generate like another question when predicting the next word such as "How much time does an engineer work ?" which is not suitable for creating a ChatGPT-like conversational model. To address this issue, one effective approach is to fine-tune the model specifically for tasks like question-answering, ensuring it learns to generate coherent and contextually appropriate responses.

3.2.1 Supervised instruction fine-tuning and Reinforcement Learning from Human Feedback (RLHF)

To develop an AI assistant rather than just a text completion AI, fine-tuning its parameters becomes crucial to tackle complex tasks effectively. Summarizing is trained thanks to well-summarized sources like research papers and books, allowing the model to perform reasonably well. However, for question-answering, the model requires training on a new dataset distinct from its pre-training data. This dataset is much smaller and consists of high-quality question-answer pairs crafted by humans, enabling the model to learn how to provide accurate responses to queries.

Furthermore, RLHF aims to align the LLM's output and human values (censorship, ...) and preferences by fine-tuning again the model. All these stages lead to massive improvements over the LLMs, in order to create the best AI assistant ever.

3.3 Transformer

A Transformer is a specific type of **Neural Network Architecture**. It's the core invention of the current breakthrough of AI. It contains four important stages such as Embedding, Attention Block, Multi-Layer Perceptron and finally Unembedding[2]. Let's explain the four important stages of how transformer works.

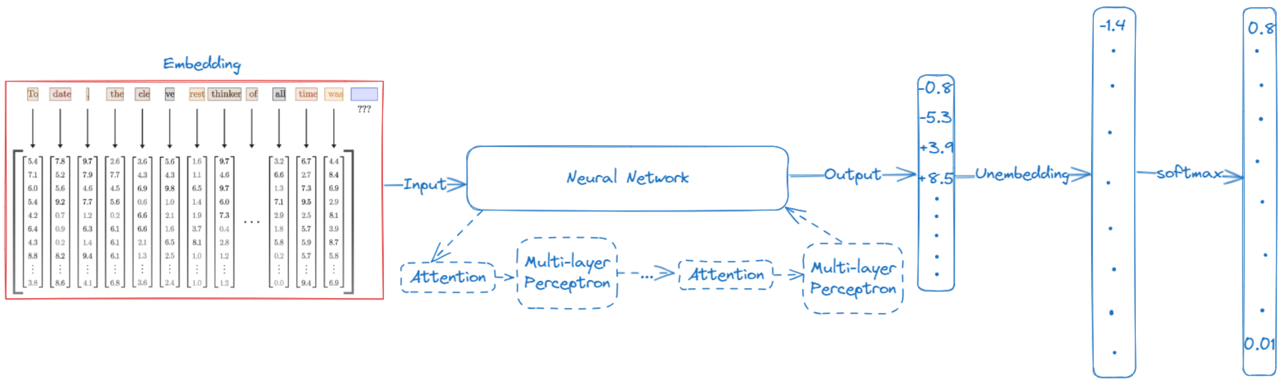


Figure 4: High-Level Transformer

3.3.1 Embedding, Positional Encoding and Unembedding

The embedding process in NLP aims to convert each word in a sequence into a fixed-dimensional vector. These vectors are generated using an embedding matrix within the neural network, where each column corresponds to a unique word. This matrix is pre-trained to optimize the word representations. Before passing these embeddings to the first attention block, they undergo in a module called **Positional Encoding**. This step embeds the word positions within the input sentence. Various techniques exist for positional encoding; a fundamental approach uses sine and cosine functions. This encoding helps capture relationships between words in the sequence; words closer together in the vector space are more similar. Embeddings can also represent complex concepts like the semantic differences between words, such as the difference between men and women (the gender) (Figure 5).

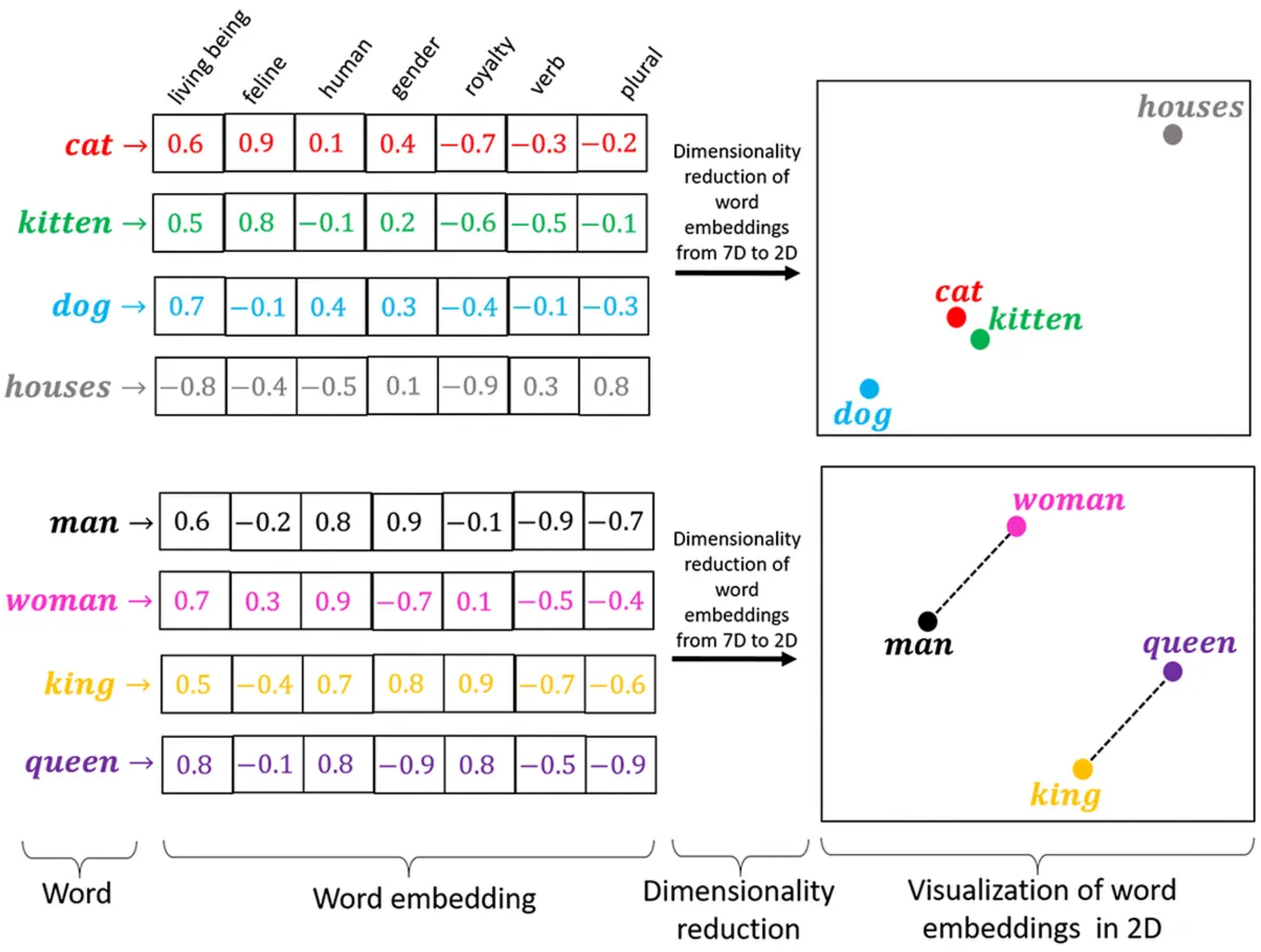

living being
feline
human
gender
royalty
verb
plural
cat → 0.6 0.9 0.1 0.4 −0.7 −0.3 −0.2
kitten → 0.5 0.8 −0.1 0.2 −0.6 −0.5 −0.1
dog → 0.7 −0.1 0.4 0.3 −0.4 −0.1 −0.3
houses → −0.8 −0.4 −0.5 0.1 −0.9 0.3 0.8
man → 0.6 −0.2 0.8 0.9 −0.1 −0.9 −0.7
woman → 0.7 0.3 0.9 −0.7 0.1 −0.5 −0.4
king → 0.5 −0.4 0.7 0.8 0.9 −0.7 −0.6
queen → 0.8 −0.1 0.8 −0.9 0.8 −0.5 −0.9
Dimensionality reduction of word embeddings from 7D to 2D
houses
cat
kitten
dog
woman
man
queen
king
Word
Word embedding
Dimensionality reduction
Visualization of word embeddings in 2D

Figure 5: Word Embedding - Space Representation by [14]

The dot product is one of the way to evaluate the similarity between two words. For instance, cat and kitten are strongly linked, therefore have a big dot product. In the example the dot product between cat and kitten is 1.69 while between dog and houses is -0.6. Negative means opposite and positive means similar. The magnitude of the dot product indicates the degree of similarity between two words: the larger the value, the more similar the words are.

The unembedding step aims to decode the final vector from the last layer, which has a fixed embedding dimension, back to its original dimension equal to the size of the vocabulary. By applying the softmax on this decoded vector, it generates **probabilities** for all words used during the **generation** of word. The unembedding matrix is pretrained with data to effectively decode a vector from a lower-dimensional space back to its original representation.

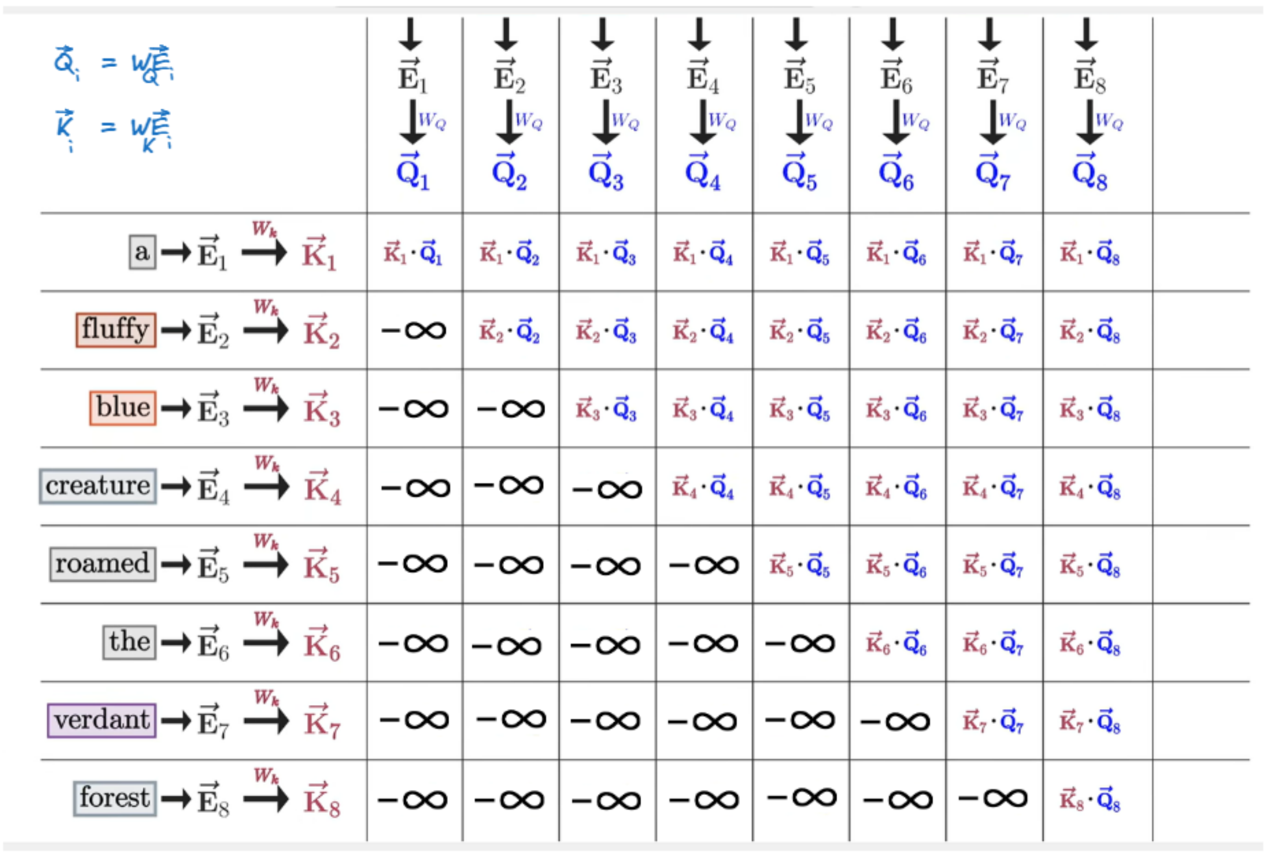
3.3.2 Single Head Attention & Attention Blocks

Many words have different meaning depending on the context. Yet, in the embedding matrix, every single word has a unique vector to represent it. In order to change the vector depending on the context, all vectors are given as an input to attention blocks. Attention blocks aim to transform each vector by taking into account the context of other words. Attention blocks is divided in few steps: pre-training and then computing on a sequence of words.

There are 3 important matrices **Query Matrix** W_Q, **Key Matrix** W_K and **Value Matrix** W_V.

All of these matrices contain weights and are tuned in the pre-training process with back propagation. Then, for each word embedding E_i, a query vector, key vector and value vector are compute $W_X \times E_i = X_i$ where X correspond to Q, K or V, with $Q_i \in \mathbf{R}^{\mathrm{d_k}}$, $K_i \in \mathbf{R}^{\mathrm{d_k}}$ and $V_i \in \mathbf{R}^{\mathrm{d_v}}$. By concatenating all vectors (respectively by Q, K and V) from the previous formula, they are transformed into matrices Q, K and V.

The dot product between Q and K allows to see what is the weight that a word have to another one in the sequence for all words. The matrix obtained is called the **Attention Pattern** (Figure 6.

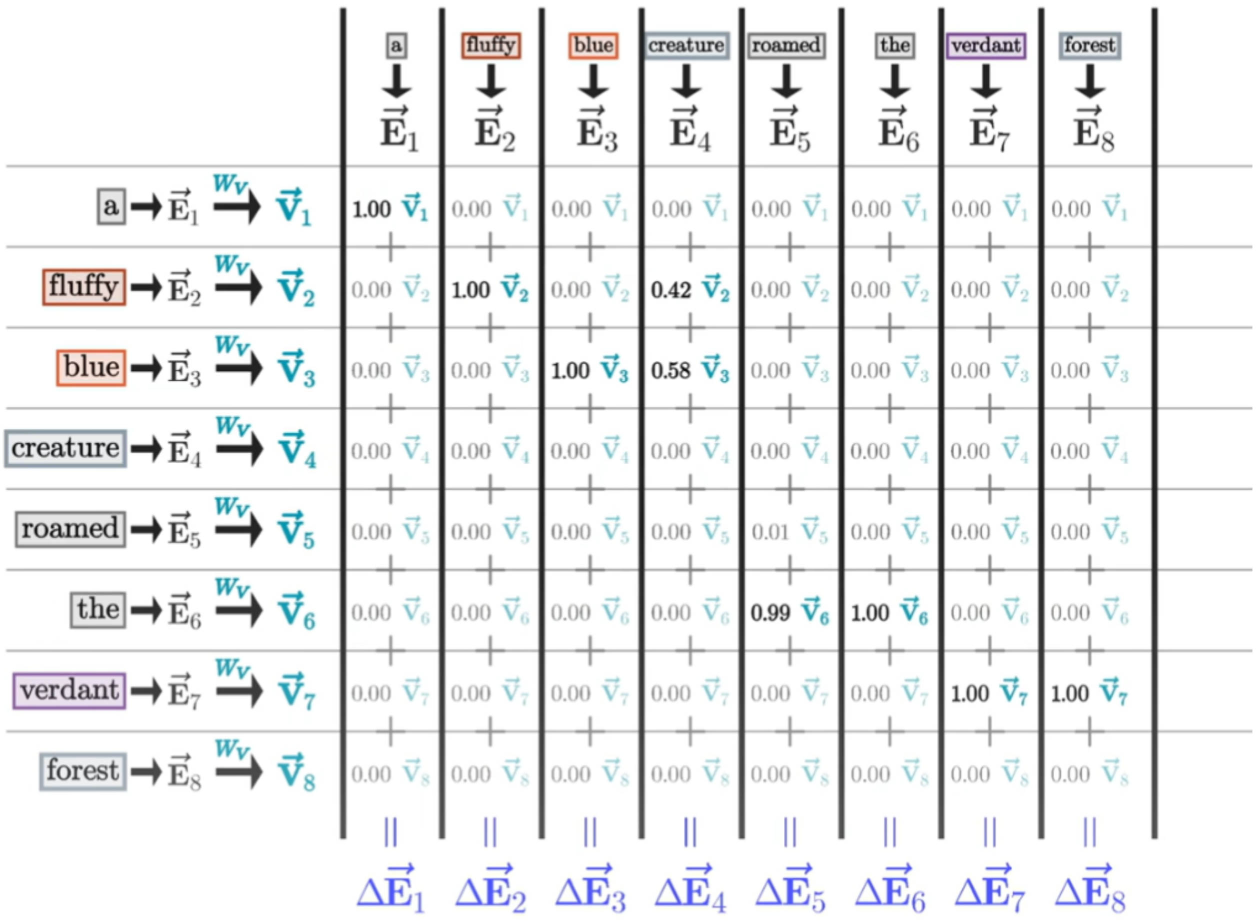


$\vec{Q}_i = W_Q \vec{E}_i$ $\vec{K}_i = W_K \vec{E}_i$	$\vec{E}_1 \xrightarrow{W_Q} \vec{Q}_1$	$\vec{E}_2 \xrightarrow{W_Q} \vec{Q}_2$	$\vec{E}_3 \xrightarrow{W_Q} \vec{Q}_3$	$\vec{E}_4 \xrightarrow{W_Q} \vec{Q}_4$	$\vec{E}_5 \xrightarrow{W_Q} \vec{Q}_5$	$\vec{E}_6 \xrightarrow{W_Q} \vec{Q}_6$	$\vec{E}_7 \xrightarrow{W_Q} \vec{Q}_7$	$\vec{E}_8 \xrightarrow{W_Q} \vec{Q}_8$
a $\rightarrow \vec{E}_1 \xrightarrow{W_k} \vec{K}_1$	$\vec{K}_1\cdot\vec{Q}_1$	$\vec{K}_1\cdot\vec{Q}_2$	$\vec{K}_1\cdot\vec{Q}_3$	$\vec{K}_1\cdot\vec{Q}_4$	$\vec{K}_1\cdot\vec{Q}_5$	$\vec{K}_1\cdot\vec{Q}_6$	$\vec{K}_1\cdot\vec{Q}_7$	$\vec{K}_1\cdot\vec{Q}_8$
fluffy $\rightarrow \vec{E}_2 \xrightarrow{W_k} \vec{K}_2$	$-\infty$	$\vec{K}_2\cdot\vec{Q}_2$	$\vec{K}_2\cdot\vec{Q}_3$	$\vec{K}_2\cdot\vec{Q}_4$	$\vec{K}_2\cdot\vec{Q}_5$	$\vec{K}_2\cdot\vec{Q}_6$	$\vec{K}_2\cdot\vec{Q}_7$	$\vec{K}_2\cdot\vec{Q}_8$
blue $\rightarrow \vec{E}_3 \xrightarrow{W_k} \vec{K}_3$	$-\infty$	$-\infty$	$\vec{K}_3\cdot\vec{Q}_3$	$\vec{K}_3\cdot\vec{Q}_4$	$\vec{K}_3\cdot\vec{Q}_5$	$\vec{K}_3\cdot\vec{Q}_6$	$\vec{K}_3\cdot\vec{Q}_7$	$\vec{K}_3\cdot\vec{Q}_8$
creature $\rightarrow \vec{E}_4 \xrightarrow{W_k} \vec{K}_4$	$-\infty$	$-\infty$	$-\infty$	$\vec{K}_4\cdot\vec{Q}_4$	$\vec{K}_4\cdot\vec{Q}_5$	$\vec{K}_4\cdot\vec{Q}_6$	$\vec{K}_4\cdot\vec{Q}_7$	$\vec{K}_4\cdot\vec{Q}_8$
roamed $\rightarrow \vec{E}_5 \xrightarrow{W_k} \vec{K}_5$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$\vec{K}_5\cdot\vec{Q}_5$	$\vec{K}_5\cdot\vec{Q}_6$	$\vec{K}_5\cdot\vec{Q}_7$	$\vec{K}_5\cdot\vec{Q}_8$
the $\rightarrow \vec{E}_6 \xrightarrow{W_k} \vec{K}_6$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$\vec{K}_6\cdot\vec{Q}_6$	$\vec{K}_6\cdot\vec{Q}_7$	$\vec{K}_6\cdot\vec{Q}_8$
verdant $\rightarrow \vec{E}_7 \xrightarrow{W_k} \vec{K}_7$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$\vec{K}_7\cdot\vec{Q}_7$	$\vec{K}_7\cdot\vec{Q}_8$
forest $\rightarrow \vec{E}_8 \xrightarrow{W_k} \vec{K}_8$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$\vec{K}_8\cdot\vec{Q}_8$

Figure 6: Attention Pattern - Entry of the Softmax by [1]

Then, the value matrix adjusts the current vector of one word weighted by the previous dot product[1, 16] by adding it with the word embedding (Figure 7). That process is called **Single Head Attention**. This is how the next formula works[16]:

$$Attention(Q, K, V) = Softmax(\frac{QK^T}{\sqrt{d_k}})V \tag{1}$$



a fluffy blue creature roamed the verdant forest
$\vec{E}_1$ $\vec{E}_2$ $\vec{E}_3$ $\vec{E}_4$ $\vec{E}_5$ $\vec{E}_6$ $\vec{E}_7$ $\vec{E}_8$
a → $\vec{E}_1 \xrightarrow{W_V} \vec{V}_1$ 1.00 $\vec{V}_1$ 0.00 $\vec{V}_1$ 0.00 $\vec{V}_1$ 0.00 $\vec{V}_1$ 0.00 $\vec{V}_1$ 0.00 $\vec{V}_1$ 0.00 $\vec{V}_1$ 0.00 $\vec{V}_1$
fluffy → $\vec{E}_2 \xrightarrow{W_V} \vec{V}_2$ 0.00 $\vec{V}_2$ 1.00 $\vec{V}_2$ 0.00 $\vec{V}_2$ 0.42 $\vec{V}_2$ 0.00 $\vec{V}_2$ 0.00 $\vec{V}_2$ 0.00 $\vec{V}_2$ 0.00 $\vec{V}_2$
blue → $\vec{E}_3 \xrightarrow{W_V} \vec{V}_3$ 0.00 $\vec{V}_3$ 0.00 $\vec{V}_3$ 1.00 $\vec{V}_3$ 0.58 $\vec{V}_3$ 0.00 $\vec{V}_3$ 0.00 $\vec{V}_3$ 0.00 $\vec{V}_3$ 0.00 $\vec{V}_3$
creature → $\vec{E}_4 \xrightarrow{W_V} \vec{V}_4$ 0.00 $\vec{V}_4$ 0.00 $\vec{V}_4$ 0.00 $\vec{V}_4$ 0.00 $\vec{V}_4$ 0.00 $\vec{V}_4$ 0.00 $\vec{V}_4$ 0.00 $\vec{V}_4$ 0.00 $\vec{V}_4$
roamed → $\vec{E}_5 \xrightarrow{W_V} \vec{V}_5$ 0.00 $\vec{V}_5$ 0.00 $\vec{V}_5$ 0.00 $\vec{V}_5$ 0.00 $\vec{V}_5$ 0.01 $\vec{V}_5$ 0.00 $\vec{V}_5$ 0.00 $\vec{V}_5$ 0.00 $\vec{V}_5$
the → $\vec{E}_6 \xrightarrow{W_V} \vec{V}_6$ 0.00 $\vec{V}_6$ 0.00 $\vec{V}_6$ 0.00 $\vec{V}_6$ 0.00 $\vec{V}_6$ 0.99 $\vec{V}_6$ 1.00 $\vec{V}_6$ 0.00 $\vec{V}_6$ 0.00 $\vec{V}_6$
verdant → $\vec{E}_7 \xrightarrow{W_V} \vec{V}_7$ 0.00 $\vec{V}_7$ 0.00 $\vec{V}_7$ 0.00 $\vec{V}_7$ 0.00 $\vec{V}_7$ 0.00 $\vec{V}_7$ 0.00 $\vec{V}_7$ 1.00 $\vec{V}_7$ 1.00 $\vec{V}_7$
forest → $\vec{E}_8 \xrightarrow{W_V} \vec{V}_8$ 0.00 $\vec{V}_8$ 0.00 $\vec{V}_8$ 0.00 $\vec{V}_8$ 0.00 $\vec{V}_8$ 0.00 $\vec{V}_8$ 0.00 $\vec{V}_8$ 0.00 $\vec{V}_8$ 0.00 $\vec{V}_8$
$\Delta\vec{E}_1$ $\Delta\vec{E}_2$ $\Delta\vec{E}_3$ $\Delta\vec{E}_4$ $\Delta\vec{E}_5$ $\Delta\vec{E}_6$ $\Delta\vec{E}_7$ $\Delta\vec{E}_8$

Figure 7: One head attention by [1]

Having high value of dot product can lead in the softmax values near 1 and therefore since it's a distribution other values near 0. Thus, leading to small gradient which might cause issues during the tuning process. That is why in practice, the dot product used is a scaled dot-product attention by $\frac{1}{\sqrt{d_k}}$. Besides, in a sequence, future words in a specific sentence shouldn't attend to current words. To solve this, before applying the Softmax function, the attention value of future words attending on current words are set to $-\infty$ to force their weights to 0. This processus is called **masking**.

Finally, in practice, one attention block correspond to a Multi-Head Attention which is a set of independent Single Head Attention previously introduced. Thanks to independence, all computations of the adjustments can be parallelized to eventually add them to one word embedding (Figure 8).

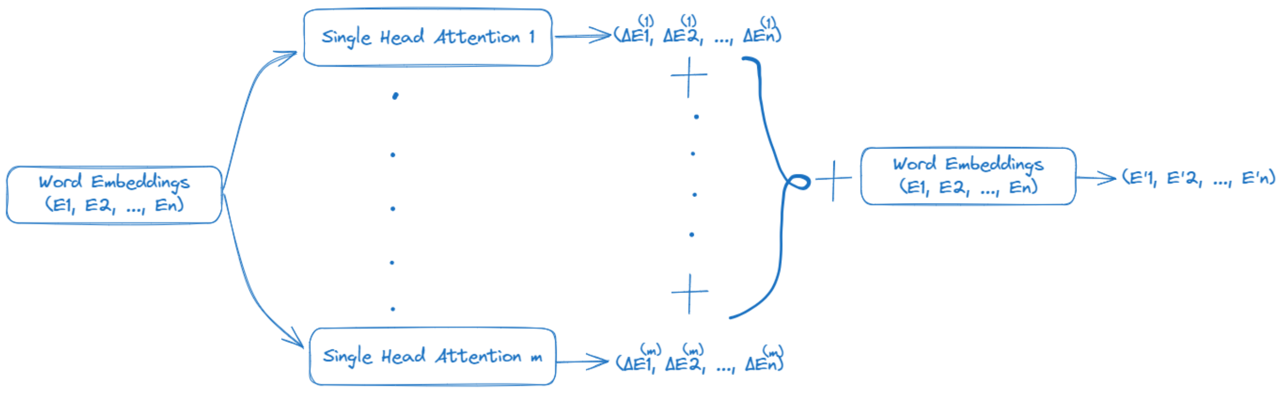

Single Head Attention 1
$(\Delta E1^{(1)}, \Delta E2^{(1)}, ..., \Delta En^{(1)})$
Word Embeddings (E1, E2, ..., En)
Word Embeddings (E1, E2, ..., En)
(E'1, E'2, ..., E'n)
Single Head Attention m
$(\Delta E1^{(m)}, \Delta E2^{(m)}, ..., \Delta En^{(m)})$

Figure 8: One Attention Block, Multi-Head Attention

3.3.3 Multi-Layer Perceptron (MLP) or Position-Wide Feed Forward

The multi-layer perceptron is the basic neural network that can be found in machine learning. It's several successive hidden layer having multiple perceptrons. It has 3 important components which are: **weights**, **biases** and **activation functions** (Figure 9). In GPT Model, the activation function is a $ReLU(x) = max(0, x)$. In this model, it takes as an input, the output of the attention block after applying the position encoding. And then, return a vector as an output and give it to the next attention block.

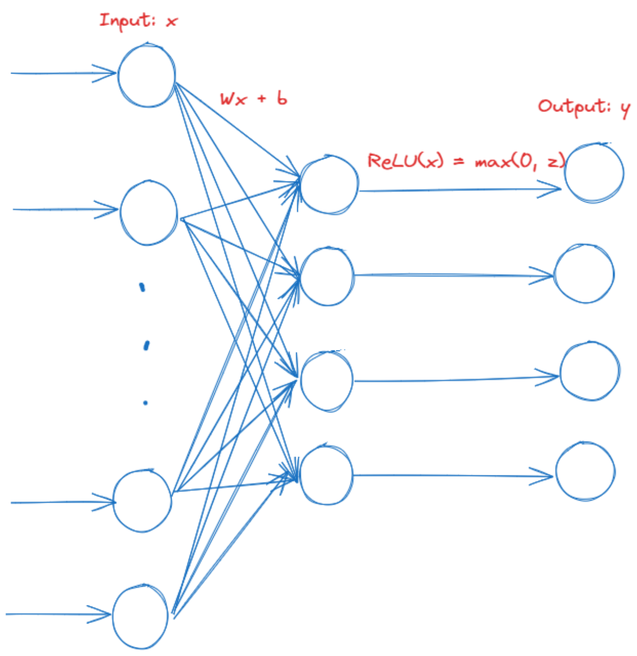


Figure 9: Basic one layer perceptron

The attention block aims to illustrate the relation between words in a sequence depending to their position and meaning. While, the MLP aims to specialize, by training, the model into an assistant AI, like being capable to summarize texts, question-answer task, and so on.

Finally, the last vector of the last layer correspond to the probability distribution of all tokens.

4 Retrieval Augmentated Generation (RAG)

All the following terms are from the Llama-Index documentation [9]; they will be defined before their usage.

Some answers of some specific questions can change through the time, therefore pre-trained models might generate a wrong answer depending their dataset. Moreover, since an LLM has billions of parameters (GPT3 has 176B parameters), it takes too much time and cost too much to re-train the model. The main purposes of the RAG application is to augment the LLM model with their own data by using them as external knowledge. Doing that, it allows the model to generate good answer without going through a re-training process.

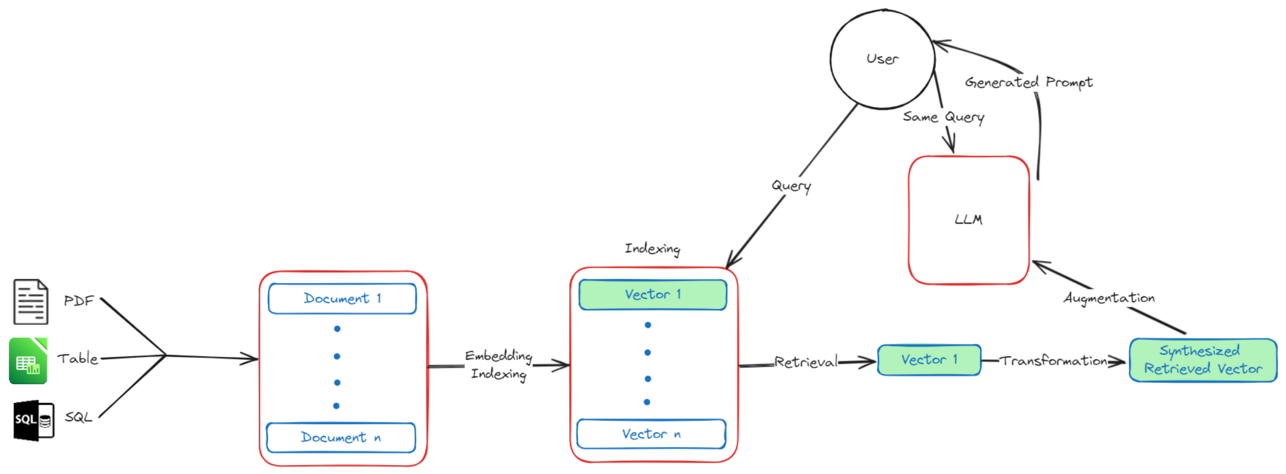


Figure 10: Basic RAG process

What is important inside the basic RAG application (Figure 10):

- One user has any type of documents (external knowledge) that are stored inside a folder for instance.
- All these documents are embedded during the Indexing step: transformed into a vector to be able to compute or do logical operation on them.
- The user inputs a query and then specific documents similar to the user query are retrieved.
- The retrieved documents are then transformed and used as a context for the input given to the LLM alongside with the initial query.
- LLM generates an answer using additional information got in the retrieved documents.

Let's now focus on each important steps to prototype a RAG.

4.1 Indexing Stage

All of the three following steps constitute the **Indexing Stage**. This stage creates an index, which is a data structure that stores the vectorized nodes.

4.1.1 Loading

The primary goal of the **loading step** is to load data provided by the user and, if required, chunk it into smaller documents before proceeding to the indexing phase.

During this step, it is crucial to be capable of reading various file types provided by the user, such as CSV files or SQL databases, and transforming them into a compatible format suitable for embedding.

Let's define the key components in this step:

- **Nodes**: Chunks of a document that will be embedded in the next step, representing atomic units of a document.
- **Connectors**: Tools that serve as readers for different data types, transforming them into nodes.

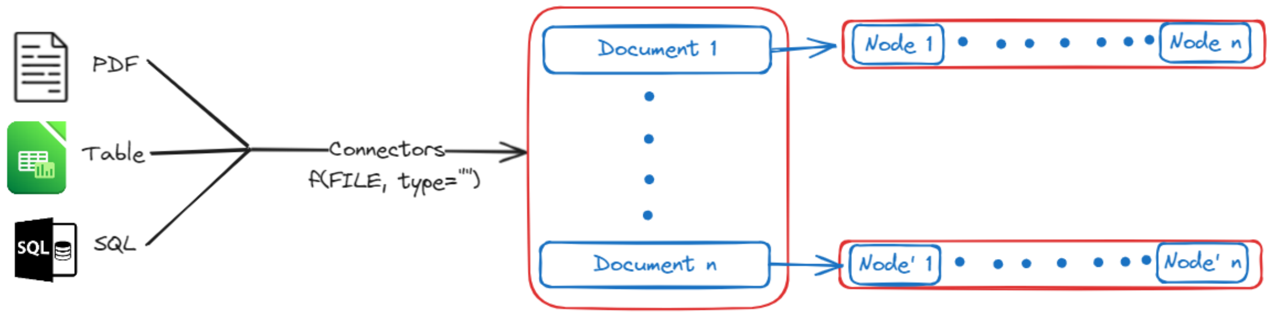


Figure 11: RAG Loading

4.1.2 Indexing

The main purpose of the **indexing step** is to transform nodes into vectors of numbers that will be stored in a Vector Database. Performing mathematical and logical operations directly to texts can be challenging. Therefore, in Natural Language Processing (NLP), texts are typically vectorized using word embedding methods in transformers to facilitate these operations.

Embedding is a method to vectorize data from any type of file. There are various embedding methods, each tailored to different types of input data.

4.1.3 Storing

The **storing step** main purpose is to store the vectors into a database known as the Index. This step is quite beneficial because it allows to avoid re-indexing all nodes each time needed for the retrieval. Instead, only new documents need to be transformed and stored.

4.2 Querying Stage

The **querying step** is focused on retrieving relevant documents from the **Index** created during the **Indexing Stage**. To compare the user's input to the nodes, both need to have the same semantics, therefore both need to be vectorized. Then, by using some operation that represent the similarity between two objects, for instance the **dot product** in k-dimension, it's possible to know which nodes are similar to the user query and thus needs to be retrieved. **Retrievers** are those responsible for fetching contextually relevant information based on a user query.

Node post-processors are modules that, for all nodes retrieved by the Retriever, transform, filter and re-rank them. Indeed, during the first retrieval, some nodes have useless context that might bias the retrievers. Therefore, is it up to those modules to filter these useless contexts and remove them. All nodes are then retrieved and re-ranked without useless context. Therefore, the ranking is much more accurate since it actually takes what is important in each node.

4.3 Evaluation

There are two different type of evaluation: **Retrieval Evaluation** and **Response Evaluation**.

For **Retrieval Evaluation**, two metrics—Hit Rate and Mean Reciprocal Rank (MRR)—are commonly used. The Hit Rate measures the proportion of queries where the top-k retrieved documents are relevant. On the other side, the MRR evaluates how well the retriever ranks the most relevant document.

The Hit Rate counts the number of queries where the answer of the user query can be found. Its goal is to check if during the Retrieval step, they aren't any useless queries that are retrieved.

For the MRR, first a judge LLM or a human judgement get the most relevant documents for the user query inside the database. And then, its ranking by the retrieval is checked. The goal is to check if the most relevant has the best rank by the retrieval.
The score given is: $\frac{1}{n}$ where n is the ranking of the document.

For the **Response Evaluation**, the faithfulness and the relevancy of the generated response are evaluated. Indeed, the faithfulness measures if the response generated can be trusted or not, in other word if it's a **hallucination** or not. The relevancy measures if the response and the retrieved context actually answers the query.

The Faithfulness evaluator (Figure 12) compares the answer by the model used and a ground-truth answer. Most of the time, the ground-truth answer is generated by a much bigger LLM such as GPT-4 which is supposed as a true answer[15].

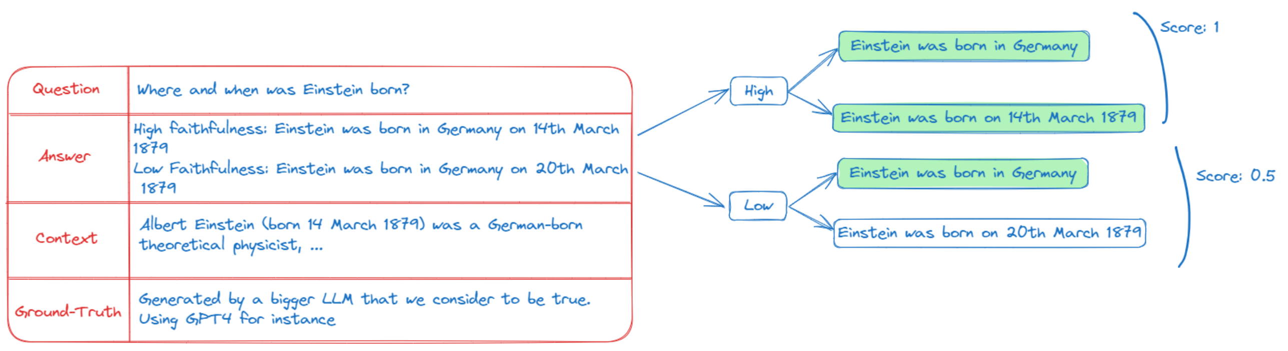
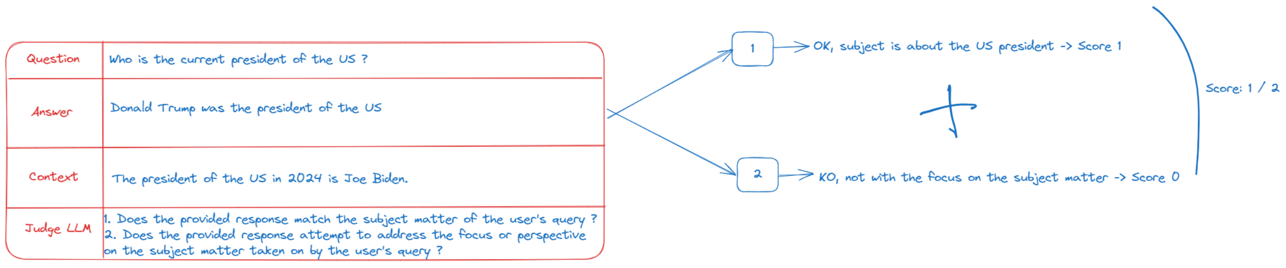

Question
Where and when was Einstein born?
Answer
High Faithfulness: Einstein was born in Germany on 14th March 1879
Low Faithfulness: Einstein was born in Germany on 20th March 1879
Context
Albert Einstein (born 14 March 1879) was a German-born theoretical physicist, ...
Ground-Truth
Generated by a bigger LLM that we consider to be true. Using GPT4 for instance
High
Einstein was born in Germany
Einstein was born on 14th March 1879
Score: 1
Low
Einstein was born in Germany
Einstein was born on 20th March 1879
Score: 0.5

Figure 12: Faithfulness score

There are two types of relevancy evaluation: **AnswerRelevancy** and **ContextRelevancy**. For each type of relevancy evaluation, two questions are asked to a judge LLM, a bigger LLM where its answer is considered as the truth. For instance, for the answer relevancy (Figure 13):

- Does the provided response match the subject matter of the user's query?
- Does the provided response attempt to address the focus or perspective on the subject matter taken on by the user's query?

A score is then given to each relevancy by summing up the score given by the judge LLM for each questions.



Question
Who is the current president of the US ?
Answer
Donald Trump was the president of the US
Context
The president of the US in 2024 is Joe Biden.
Judge LLM
1. Does the provided response match the subject matter of the user's query ?
2. Does the provided response attempt to address the focus or perspective on the subject matter taken on by the user's query ?
1
OK, subject is about the US president -> Score 1
+
2
KO, not with the focus on the subject matter -> Score 0
Score: 1 / 2

Figure 13: Answer relevancy score

5 Basic implementation using Llama-index

For the data that is used, it's the data given by Llama-Index: an essay by Paul Graham [3]

5.1 LLM Model with a naive RAG

All the following code was made by Llama-Index [9] about implementing all stages of a RAG application. Here's the code for the indexing stage:

- Loading by using the following reader for txt: SimpleDirectoryReader
- Embedding the model by using an open source model from HuggingFaceEmbedding and by using an LLM
- Indexing and Storing the documents with VectorStoreIndex inside index

```python
from llama_index.core import VectorStoreIndex, SimpleDirectoryReader,
    Settings, load_index_from_storage
from llama_index.embeddings.huggingface import HuggingFaceEmbedding
from llama_index.llms.ollama import Ollama
from llama_index.core.storage import StorageContext
import os

documents = SimpleDirectoryReader("data").load_data()

# bge-base embedding model
Settings.embed_model = HuggingFaceEmbedding(model_name="BAAI/bge-base-en-
    v1.5")

# ollama
Settings.llm = Ollama(model="llama3", request_timeout=360.0)

# Indexing & Storing
if not os.path.exists("index_storage"):
    index = VectorStoreIndex.from_documents(
        documents,
    )
    index.storage_context.persist("index_storage")
# Loading if already stored
else:
    storage_context = StorageContext.from_defaults(persist_dir="
        index_storage")
    index = load_index_from_storage(storage_context)
```

For the **Indexing Stage**, the loading step is done by the **connector "SimpleDirectoryReader"**. Then both indexing and storing step are done with one module **VectorStoreIndex**. To load a VectorStoreIndex, load_from_storage function is used.

The query engine is a generic interface that allows any user to ask a question.

Here's the code for the querying stage:

- Index seen as query engine meaning "taking account the whole database"
- Generating with the database index as context with a specific prompt with Llama3

```
query_engine = index.as_query_engine()
response = query_engine.query("What did the author do growing up?")
print(response)
```

5.2 Implementing Evaluation

In order to compare which models are the best or to follow the Evaluation Driven Development method, it is a must to have the same samples of test to evaluate (loading and saving the dataset).

5.2.1 Generation Evaluation

The main way is to generate QA dataset by an LLM. Each query are associated with relevant contexts and answer. Then, each query are given as prompt for the RAG application, where its output is compared to the content in the QA dataset. The output of the generation evaluator are four types of evaluations

- Correctness: evaluate the relevance and correctness of a generated answer against a reference answer
- Faithfulness: measure if the response matches any source nodes, check the hallucination
- Relevancy: evaluate the relevancy of retrieved contexts.
- Context Similarity: evaluate the quality of a QA system by comparing

The dataset generated is a set of the class **LabelledRagDataExemple** which have multiple attributes: one query, the reference_contexts of this query and finally the reference_answer.

```
from llama_index.llms.openai import OpenAI

from llama_index.core import SimpleDirectoryReader, Settings,
   VectorStoreIndex
from llama_index.core.llama_dataset.generator import RagDatasetGenerator
from llama_index.core.llama_dataset import LabelledRagDataset
from llama_index.core.llapa_pack import download_llama_pack
from llama_index.llms.ollama import Ollama
from llama_index.embeddings.huggingface import HuggingFaceEmbedding
from llama_index.core.node_parser import SentenceSplitter

import os

# Initialization
Settings.llm = Ollama(model="llama3", request_timeout=360.0)
Settings.embed_model = Ollama(model_name="BAAI/bge-base-en-v1.5")
RagEvaluatorPack = download_llama_pack("RagEvaluatorPack", "./pack")
node_parser = SentenceSplitter(chunk_size=512)
nodes = node_parser.get_nodes_from_documents(documents)
for idx, node in enumerate(nodes):
    node.id_ = f"node_{idx}"


# Loading Step
documents = SimpleDirectoryReader("data").load_data()
index = VectorStoreIndex(nodes)

query_engine = index.as_query_engine()

# Instantiate a DatasetGenerator and Create a QA dataset for the
   Generation
dataset_generator = RagDatasetGenerator.from_documents(documents, llm=
   OpenAI(model="gpt-4"), num_questions_per_chunk=2)

if os.path.exists("rag_dataset.json"):
    rag_dataset = LabelledRagDataset.from_json("rag_dataset.json")
else:
    rag_dataset = dataset_generator.generate_dataset_from_nodes()
    rag_dataset.save_json("rag_dataset.json")

rag_evaluator = RagEvaluatorPack(query_engine=query_engine, rag_dataset=
   rag_dataset, judge_llm=Settings.llm)

benchmark_df = rag_evaluator.run()
print(benchmark_df)
```

5.2.2 Retrieval Evaluation

For the evaluation of the retrieval, a judge llm will generate a dataset pairs of question and context. Then, this dataset is given to the LLM used to return a **Hit Rate** value and **MRR** value. Here's the implementation to display through a dataframe the answer.

```
# Code from Llama-index in RetrieverEvaluator
def display_results(name, eval_results):
    """Display results from evaluate."""

    metric_dicts = []
    for eval_result in eval_results:
        metric_dict = eval_result.metric_vals_dict
        metric_dicts.append(metric_dict)

    full_df = pd.DataFrame(metric_dicts)

    columns = {
        "retrievers": [name],
        **{k: [full_df[k].mean()] for k in ["hit_rate", "mrr"] },
    }

    metric_df = pd.DataFrame(columns)

    return metric_df
```

Here's the implementation for the retrieval evaluation.

```
import asyncio
from llama_index.llms.openai import OpenAI

from llama_index.core import SimpleDirectoryReader, Settings,
   VectorStoreIndex
from llama_index.llms.ollama import Ollama
from llama_index.embeddings.huggingface import HuggingFaceEmbedding
from llama_index.core.evaluation import RetrieverEvaluator,
   EmbeddingQAFinetuneDataset, generate_question_context_pairs
from llama_index.core.node_parser import SentenceSplitter

import os

# Initialization
Settings.llm = Ollama(model="llama3", request_timeout=360.0, base_url=
   ollama_base_url)
Settings.embed_model = HuggingFaceEmbedding(model_name="BAAI/bge-base-en-
   v1.5")

# Loading step
documents = SimpleDirectoryReader("data").load_data()
node_parser = SentenceSplitter(chunk_size=512)
nodes = node_parser.get_nodes_from_documents(documents)
for idx, node in enumerate(nodes):
    node.id_ = f"node_{idx}"

index = VectorStoreIndex(nodes)
retriever = index.as_retriever(similarity_top_k=2)

# Instantiate a dataset for the Retrieval.
if os.path.exists("retriever_dataset.json"):
    retriever_dataset = EmbeddingQAFinetuneDataset.from_json("
       retriever_dataset.json")
else:
    retriever_dataset = generate_question_context_pairs(documents, llm=
       OpenAI("gpt-4"), num_questions_per_chunk=2)
    retriever_dataset.save_json("retriever_dataset.json")

retriever_evaluator = RetrieverEvaluator.from_metric_names(["hit_rate", "
   mrr"], retriever=retriever)

async def get_retriever_evaluation():
    eval_results = await retriever_evaluator.aevaluate_dataset(
       retriever_dataset)
    return eval_results

eval_results = asyncio.run(get_retriever_evaluation())
print(display_results("top-2 eval", eval_results))
```

6 Improving RAG application for Production purposes

All of the methods are explained in the Llama-Index docs [5, 9]. They are **advanced methods to improve RAG application**.

6.1 Decoupling chunks for retrieval and synthesis

A basic retrieval will after embedding the documents, retrieved the top-k documents that are the most relevant to the user query. However, some words and sentences (filler words) are actually useless but are still taken in account and therefore bias which one are the most relevant. To mitigate this issue, one approach is to separate the retrieval and synthesis stages

One method to achieve this separation is by summarizing the documents (**Embed Summary**)(Figure 14) to remove filler words. The system identifies the top-k relevant documents based on these summaries for retrieval. Finally, the full raw text of these top-k documents is provided to the language model for synthesis.

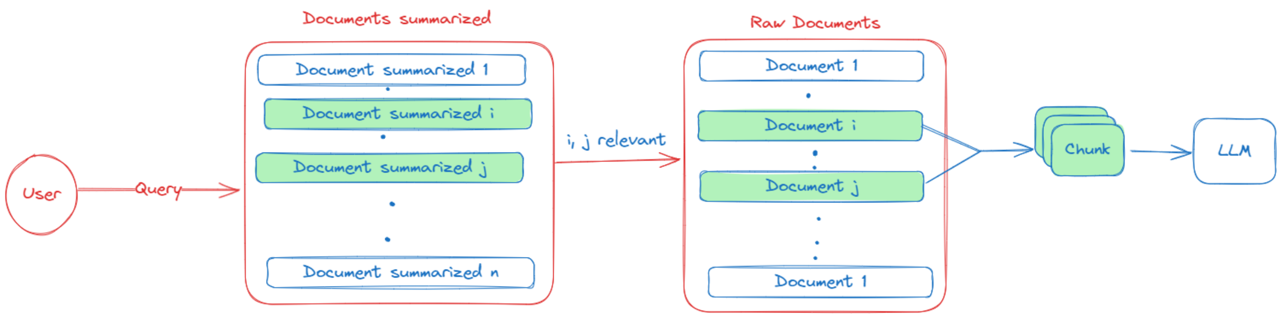


Figure 14: Decoupling retrieval and synthesis by summarizing

Another method to decouple is to evaluate sentence by sentence which one are the most relevant (**Embed Sentence**)(Figure 15). After that, for the top-k sentences, a large windows centered on the retrieved sentence is given to the model in order to give more context.

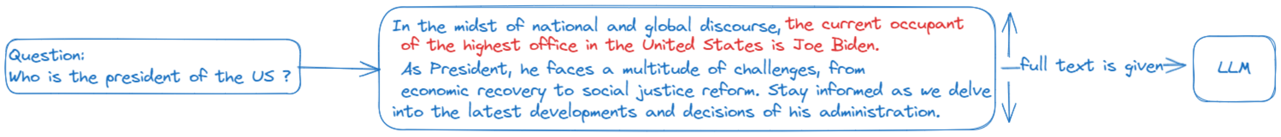

Question:
Who is the president of the US ?
In the midst of national and global discourse, the current occupant of the highest office in the United States is Joe Biden.
As President, he faces a multitude of challenges, from economic recovery to social justice reform. Stay informed as we delve into the latest developments and decisions of his administration.
full text is given
LLM

Figure 15: Decoupling retrieval and synthesis by sentence

Last but not the least is called **Re-Ranking**. The main purpose of this, consists of two steps that will decouple chunks for retrieval and synthesis. Indeed, during the retrieval step, a basic way of retrieval can be done here using for instance a Sentence Splitter retriever (=Embed Sentence). All of these nodes that were retrieved are considered as a candidate to be used as relevant to answer to the input question. These nodes are a couple of text and relevancy score. Therefore, re-ranking means using a more accurate method after the retrieval to compute a new relevancy score to see among the candidate, which one is the most important. Re-ranking is one type of **Node Post-Processor modules** seen in the fundamentals.

6.2 Structured Retrieval for Larger Documents Sets

Having documents structured can drastically improve the Retrieval. Indeed, structuring documents mean to give more details about the data and how to organize them. Thus, improving the performance of the retrieval.

One approach is to use **Metadata filters and Auto-Retrieval** (**Routers**). Indeed, structuring by Metadata explains what a document is about. So for retrieving, documents' metadata that match with the user query's metadata are retrieved, and only then, the relevant node are retrieved. It can enhance retrieval performance, since it reduce the number of candidate that might be retrieved. For instance, in the context of SSH Logs data, two type of connections are possible (succeeded or failed). Therefore, one structure that can be done, is to split these connections into two different SQLTables. Then, an LLM generates metadata for each table. The Auto-Retrieval will then, checks thanks to Metadata associated to each table, which table to use in order to retrieve relevant queries.

```
from pydantic import BaseModel, Field
from llama_index.core.program import LLMTextCompletionProgram

# Creating the Metadata generator and the Metadata schema.
class TableInfo(BaseModel):
    """Information regarding a structured table."""
    table_name: str = Field(..., description="table_name (must be
       underscores and NO spaces)")
    table_summary: str = Field(..., description="short, concise summary/
       caption of the table")

prompt_str = """\
You are given a ssh logs data.
Give me a summary of the table only by their name.
Name:
{table_name}
Summary:"""

program = LLMTextCompletionProgram.from_defaults(
    output_cls = TableInfo,
    llm=Settings.llm,
    prompt_template_str=prompt_str,
)
```

```
# Generating the metadata for each table.
import json, os
from pathlib import Path

table_infos = []
if not Path("table_json_dir").exists():
    os.makedirs("table_json_dir")

for table_name in table_names:
    if os.path.exists(f"table_json_dir/{table_name}.json"):
        # Loading
        table_info = TableInfo.parse_file(Path(f"{table_name}.json"))
    else:
        table_info = program(table_name=table_name)
        # Storing
        out_file = f"table_json_dir/{table_name}.json"
        json.dump(table_info.dict(), open(out_file, "w"))
    table_infos.append(table_info)
```

6.3 Dynamically Retrieve Chunks depending on the task given

There are multiples ways to retrieve chunks. RAG isn't just about QA (Question-Answering) but can do other tasks as well such as summarizing or comparing. Therefore, selecting the right query_engine (index containing the important nodes) is important in order to execute well a specific task. The process of choosing the right query_engine is called **Routing**. Here's a Query-Time Table Retrieval where tables are dynamically retrieved depending to the user query. From which table with Object Indexing: by embedding each tables thanks to the metadata associated to its table, and which rows to retrieved with Row Embedding: by embedding every single rows of the table, and Retrieving [8]. By doing so, with a RAG application using SQL as external knowledge, it is possible to retrieve from the right table, the relevant rows.

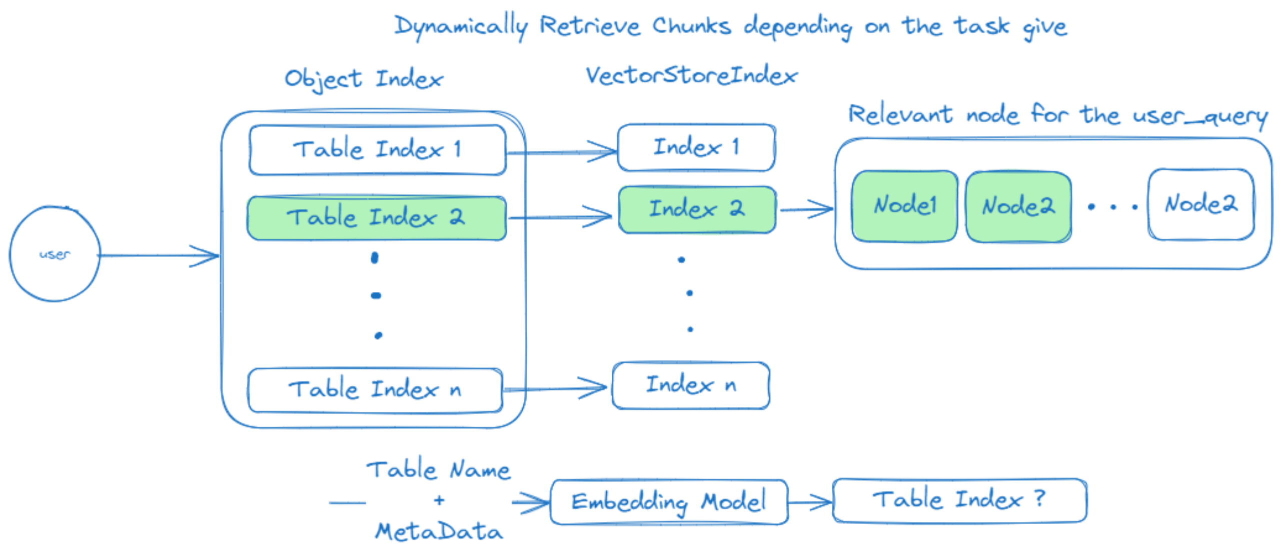


Figure 16: Dynamically Retrieval depending the task and the Metadata

```
from llama_index.core.objects import (
    SQLTableNodeMapping, ObjectIndex, SQLTableSchema
)
from llama_index.core import SQLDatabase
from llama_index.core.base.base_retriever import BaseRetriever
from typing import List, Any

def object_indexing(engine: Engine, table_infos: List[Any]) ->
   BaseRetriever:
    """Object Index (Index for SQLTable)"""
    sql_database = SQLDatabase(engine)
    table_node_mapping=SQLTableNodeMapping(sql_database)
    table_schema_objs = [SQLTableSchema(table_name=t.table_name,
       context_str=t.table_summary) for t in table_infos]

    obj_index =
    ObjectIndex.from_objects(table_schema_objs, table_node_mapping,
       VectorStoreIndex)

    obj_retriever = obj_index.as_retriever(similarity_top_k=3)
    return obj_retriever
```

```
from llama_index.core import SQLDatabase, VectorStoreIndex, Settings
from llama_index.core.schema import TextNode
from llama_index.embeddings.huggingface import HuggingFaceEmbedding
from typing import Dict, List
from sqlalchemy import text

def index_all_tables(
    sql_database: SQLDatabase, table_names: List[str]
) -> Dict[str, VectorStoreIndex]:
    """Index all tables."""
    Settings.embed_model = HuggingFaceEmbedding(model_name="BAAI/bge-
       small-en-v1.5)

    # Creation d'un dictionnaire de VectorStoreIndex:
    vector_index_dict = {}
    engine = sql_database.engine

    for table_name in table_names:
        with engine.connect() as conn:
            result = conn.execute(text(f"SELECT * FROM {table_name}"))
            rows = [row[1:] for row in result.fetchall()]

            rows_to_tuple = []
            for row in rows:
                rows_to_tuple.append(tuple(row))

        nodes = [TextNode(text=str(t)) for t in rows_to_tuple]

        index = VectorStoreIndex(nodes, show_progress=True)
        vector_index_dict[table_name] = index
    return vector_index_dict
```

6.4 Optimizing context embedding

Finally, the last method to improve the model is to directly fine-tune the embedding model to match the documents. Indeed, having a fine-tuned embedding model allows better retrieval, because it can match semantically better and therefore have much more relevant nodes. The most common way to do that is to generate a Question Answer (QA) Dataset based on the external knowledge. This QA

Dataset, will then be used to train the embedding model. After the training, it can be evaluate using the Test Dataset [10]. Compared to fine-tuning a whole LLM, fine-tuning the embedding model cost a lot less since it has smaller size of parameters.

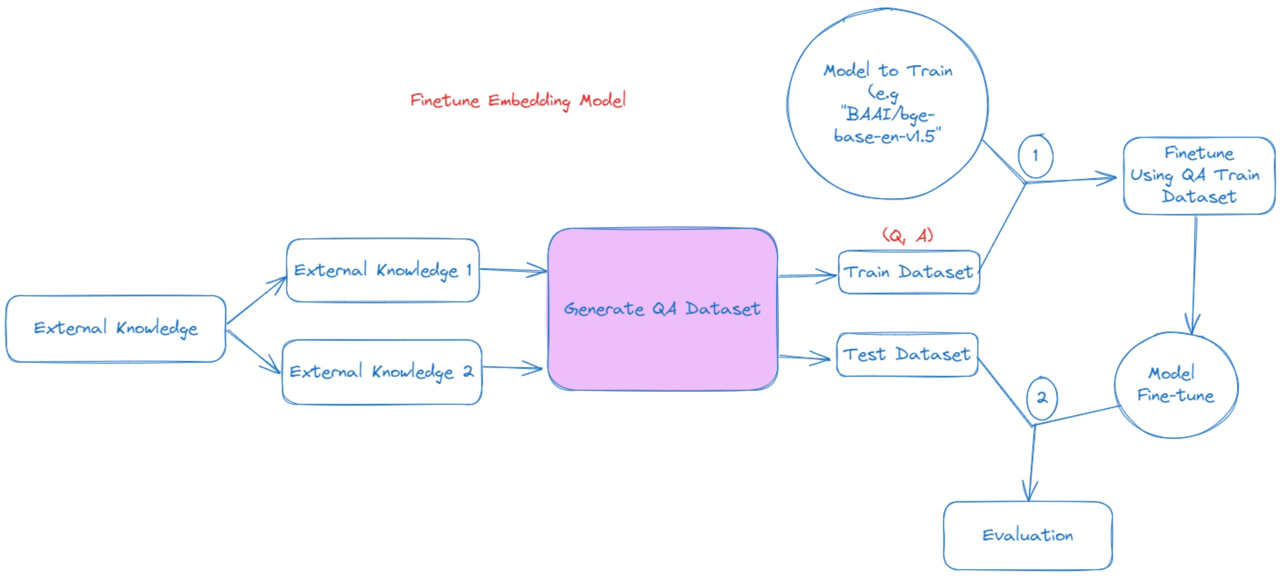
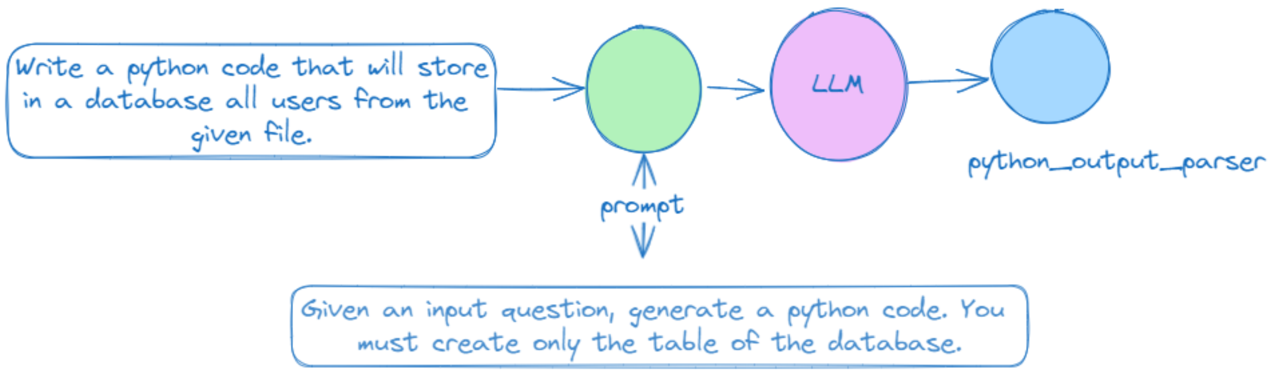


Figure 17: Finetuning Embedding Model and Evaluation

6.5 Query Pipeline

A Query Pipeline can be seen as a graph, where each vertex is one step that the application needs to follow. The main goal is to guide the application through instructions to solve complex tasks. Each vertex is associated to a specific role needed for the complex task.

Let's illustrate it with a basic pattern of query pipeline. Let's say all the input question are about storing some data into a database. Therefore, asking the LLM to focus only on the table creation first is important. The query pipeline will take as input the input question and then, create prompt that will be given to the LLM. Finally, the output generated by the LLM will be parsed to get an executable python code.



Write a python code that will store in a database all users from the given file.
LLM
python_output_parser
prompt
Given an input question, generate a python code. You must create only the table of the database.

Figure 18: Query Pipeline focused only on the table creation

The idea is to put in sequence several query pipelines to tackle a complex task.

7 Advanced RAG Application (ChatBot) for Cybersecurity purposes

After all these explanations, let's focus on how to implement a RAG Application in the case of Log Analysis. What the user wants the most is to create an application capable to make statistical report or analysis from a dataset. Focusing on SSH Logs Data for instance, several things are quite interesting such as which username is the most use to try to log-in. Look in the appendix, for the whole implementation.
Using the Basic RAG Application explained will most of the time failed for very complex tasks. Indeed, the Retrieval step will often retrieve wrong nodes. The main purposes of an **Advanced RAG Application** are to improve this **Retrieval Step** in order to give the best information possible to one LLM.

Doing statistics on a text file is very challenging. That is why, it's important to transform the logs into a simpler format where these operations can be easily done such as SQL Database. The strategy that is done in our implementation is: first, transform the log data into a sqlite database using LLM: **Database Generation Step**, and then, with another LLM, generate some SQL Request to answer to the query **Decision Step**. These two steps are both two separates RAG Application using each different type of external knowledge

So given a pair of input questions, the first one will go through the first rag application **SSH Database**, where by generating some python code, a temporary SQLite database will be generated. The idea is to create dynamically a database for each input question, where each rows and tables inside the database can be used to answer the input question. Then, after giving the schema of each table to the **SSH Analyzer**, the correct table will be used to generate a SQL Request to answer the input question.

Splitting into a pair of question for each RAG Application is very important, because each RAG application must get a input question specific to them. Otherwise, one won't even perform correctly the task and tends to generating non executable python code. For instance, asking the database to "list all users" might change totally its behaviours while "store all users" will make the LLM much more capable.

It is important to note that the SSH Logs are crude. Meaning, there was no pre-processing, and immediately retrieved from a real logs from Okayama University.

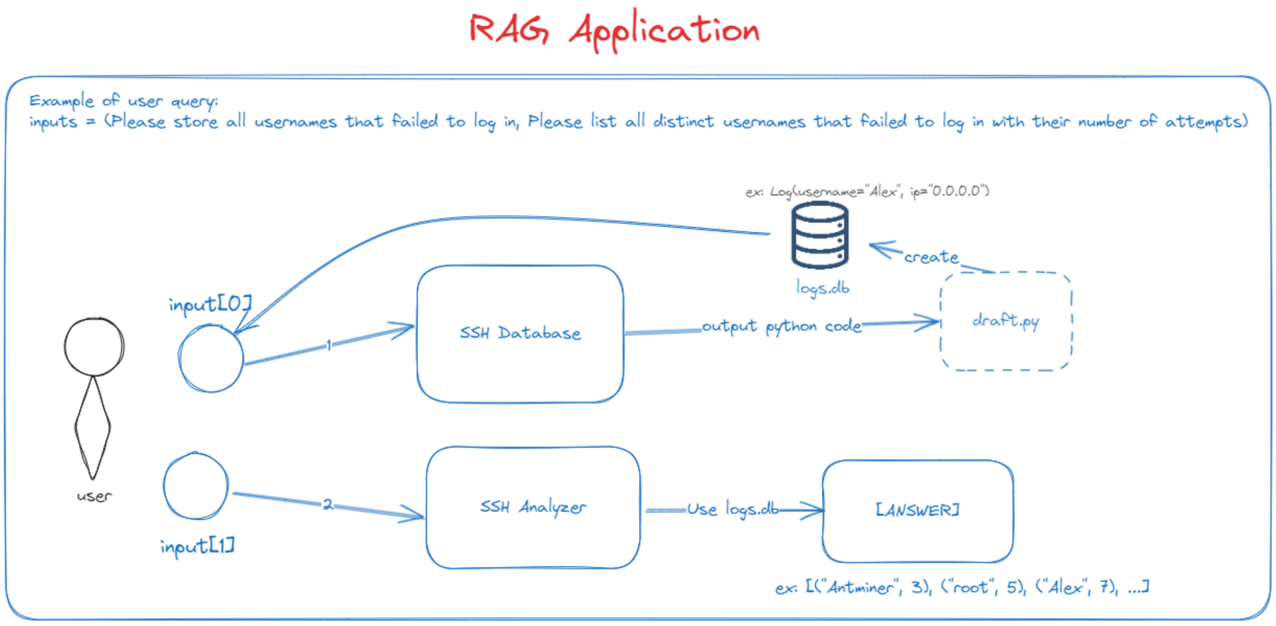

RAG Application
Example of user query:
inputs = (Please store all usernames that failed to log in, Please list all distinct usernames that failed to log in with their number of attempts)
ex: Log(username="Alex", ip="0.0.0.0")
logs.db
create
input[0]
1
SSH Database
output python code
draft.py
user
input[1]
2
SSH Analyzer
Use logs.db
[ANSWER]
ex: [("Antminer", 3), ("root", 5), ("Alex", 7), ...]

Figure 19: Architecture of RAG Application for Log Analysis

All query pipelines and the RAG Application for Log Analysis architecture were strongly inspired by **Zero-Shot Learning - Chain of Thought**[17], where the idea is to split the complex task into much smaller one to do them as good as possible. Let's explain the two rag applications.

7.1 SSH Database

SSH Database is a RAG Application where the external knowledge is a text-file (ssh logs, here). The main goal of this RAG is to improve the retrieval for text-file and then, generate a python code that will generate the sql database using SQLAlchemy.

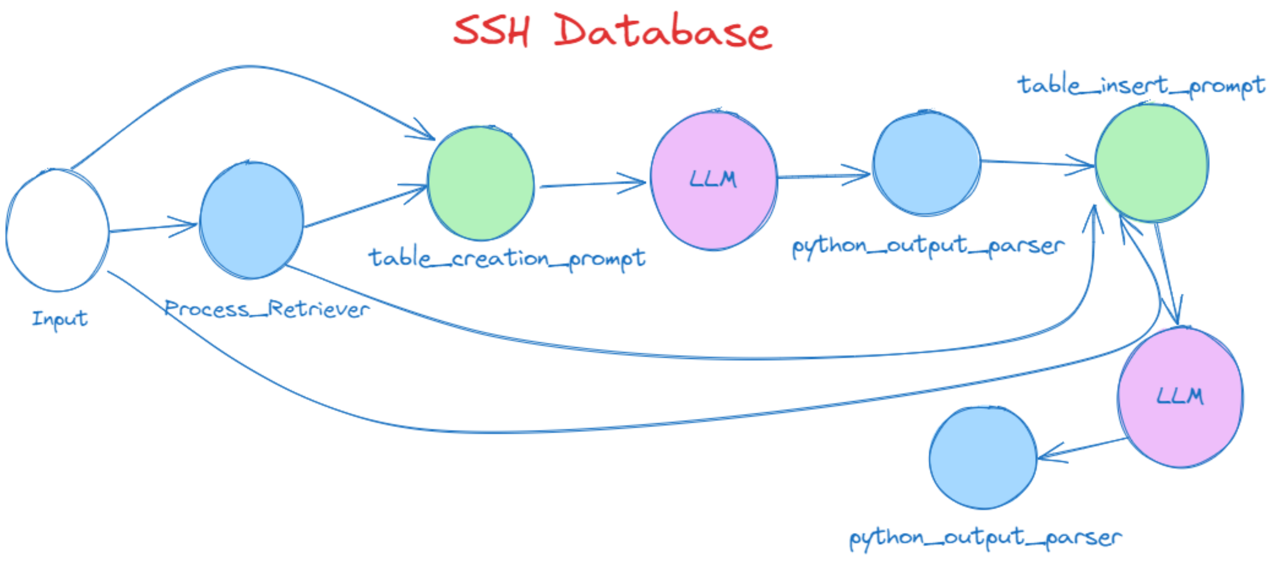


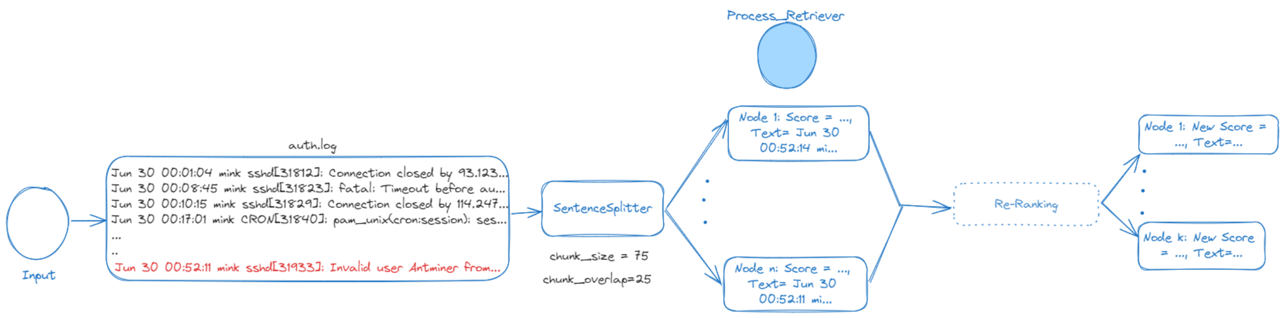
Figure 20: RAG Application for Database Generation

7.1.1 Process Retriever

The process retriever is using two retrievers. One is a **Node-Postprocessor** (Re-Ranking) and the other one is the SentenceSplitter.

The Sentence Splitter a method of splitting the document into chunks. These chunks are then retrieved depending if they are relevant to the input question. Choosing the correct chunk sizes is very important. Indeed, the chunk sizes decide how the document will be cut and in the case of ssh logs, two consecutive lines are often independent, meaning one line can be a failed connection while the second line be a successful one, Since one input question is one sided (failed or successful), it is important to retrieve only one row. Therefore, having a big chunk sizes, during retrieval, might wrong the relevant nodes, which is why, in this case, it's important to make the chunk sizes small enough.

Then, for the postprocessing, the Re-Ranking is done by using LLM which goal is to re-rank (give new scores and filter) each candidate nodes from the Sentence Splitter whether or not they are relevant to the input question. Using the a Node-Postprocessor should improve the performance of the retriever.


Process_Retriever
Input
auth.log
SentenceSplitter
chunk_size = 75
chunk_overlap=25
Node 1: Score = ..., Text= Jun 30 00:52:14 mi...
Node n: Score = ..., Text= Jun 30 00:52:11 mi...
Re-Ranking
Node 1: New Score = ..., Text=...
Node k: New Score = ..., Text=...

```
Jun 30 00:01:04 mink sshd[31812]: Connection closed by 93.123...
Jun 30 00:08:45 mink sshd[31823]: fatal: Timeout before au...
Jun 30 00:10:15 mink sshd[31829]: Connection closed by 114.247...
Jun 30 00:17:01 mink CRON[31840]: pam_unix(cron:session): ses...
...
..
Jun 30 00:52:11 mink sshd[31933]: Invalid user Antminer from...
```

Figure 21: Query Pipeline for Database Generation

7.1.2 Basic Query Pipeline

The two next chains of modules are made of a basic query pipeline explained in the production RAG part. Given an input question and relevant nodes, it's given to a prompt and feed to an LLM. The output parser is here is clean the response every time for robustness purposes. To solve the task of database generation, the complex task is split in two smaller tasks (the two basic query pipeline), first focus only on creating the table given the input question and relevant nodes. Then, focus only on the insertion given the input question, relevant nodes and the table that was created. An LLM is much more capable when it is focused on only one specific task. Indeed, Multi-Tasking have often lead to a worse score.

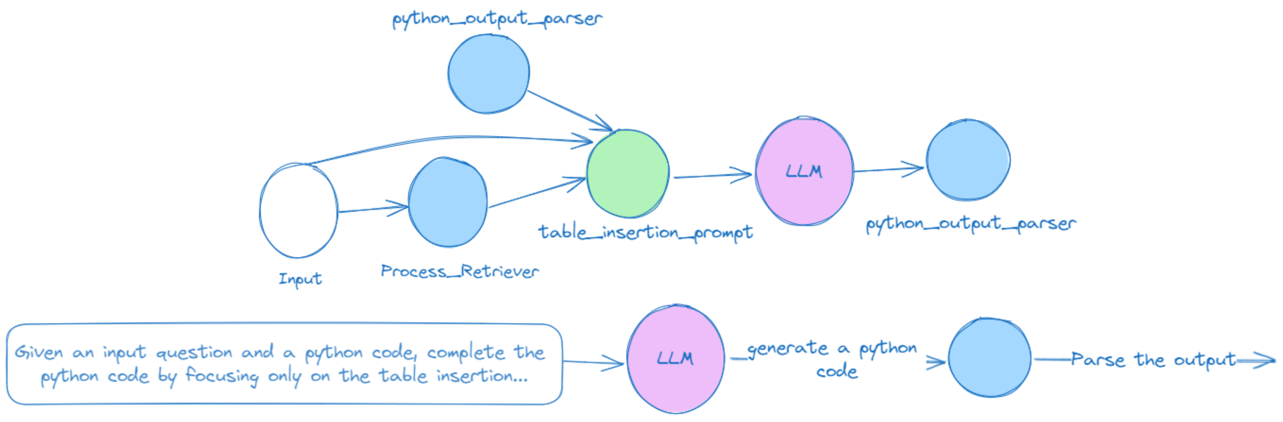


Figure 22: Example of a basic query pipeline in the case of SQL Table Insertion

7.2 SSH Analyzer

The SSH Analyzer is a RAG Application where the external knowledge is a SQLDatabase (the one generated by the SSH Database). The main goal of this RAG is to improve the retrieval for sqldatabase-file and then, generate a SQLRequest that will extract all the data needed to the input question. The main goal here is to improve a RAG Application not for a text-file, but rather a sqldatabase-file. Therefore, other methods must be used in order to improve the retriever.

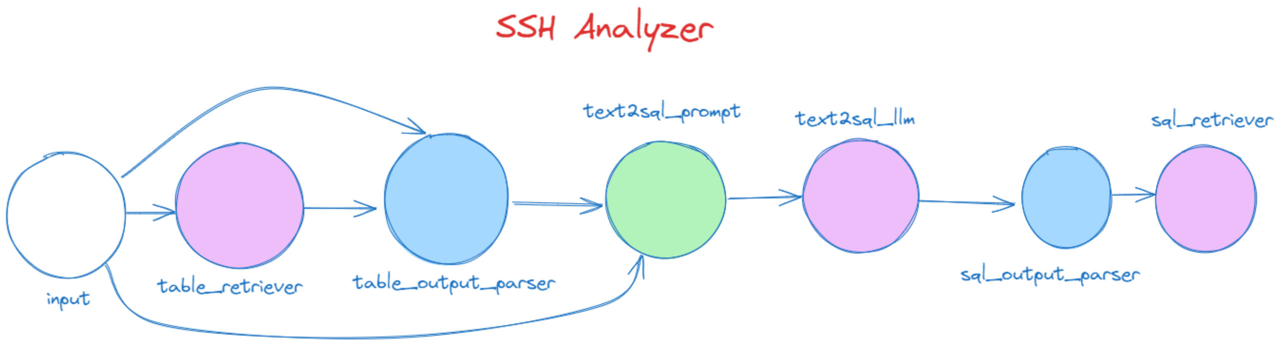


Figure 23: Query Pipeline for SSH Analysis

7.2.1 Text-to-SQL with Query-Time Row Retrieval (along with Table Retrieval)

In the initialization, to improve the table retrieval, three methods are used **Structuring Table**, **Object Indexing** and **Row Embedding**.

Structuring Table means for each table in the SQL Database, a metadata is generate and use as a filters. This method was presented previously in this paper **Structured Retrieval for Larger Documents Sets** in the case of SQL Database.

Then, thanks to these metadata, **Dynamically Retrieve Chunks depending on the task given** can be used here. To do so, the concatenation of the table name and the metadata previously generated are first embedded. Then, during the retrieval process, only relevant table similar to the input question are used to get information. For instance, in the case of Log data analysis, imagine two tables failed and successful connection. If the input question is *Please give me all users that* *failed to logged in*, it's a must to have the retriever only focus on the failed table, because retrieving in the successful one might make the answer wrong. This is called **Object Indexing**. Although, this example aren't supposed to happens because the table are dynamically created, meaning there won't be two opposite tables, this was done for more robustness purposes because one LLM might still generate useless tables.

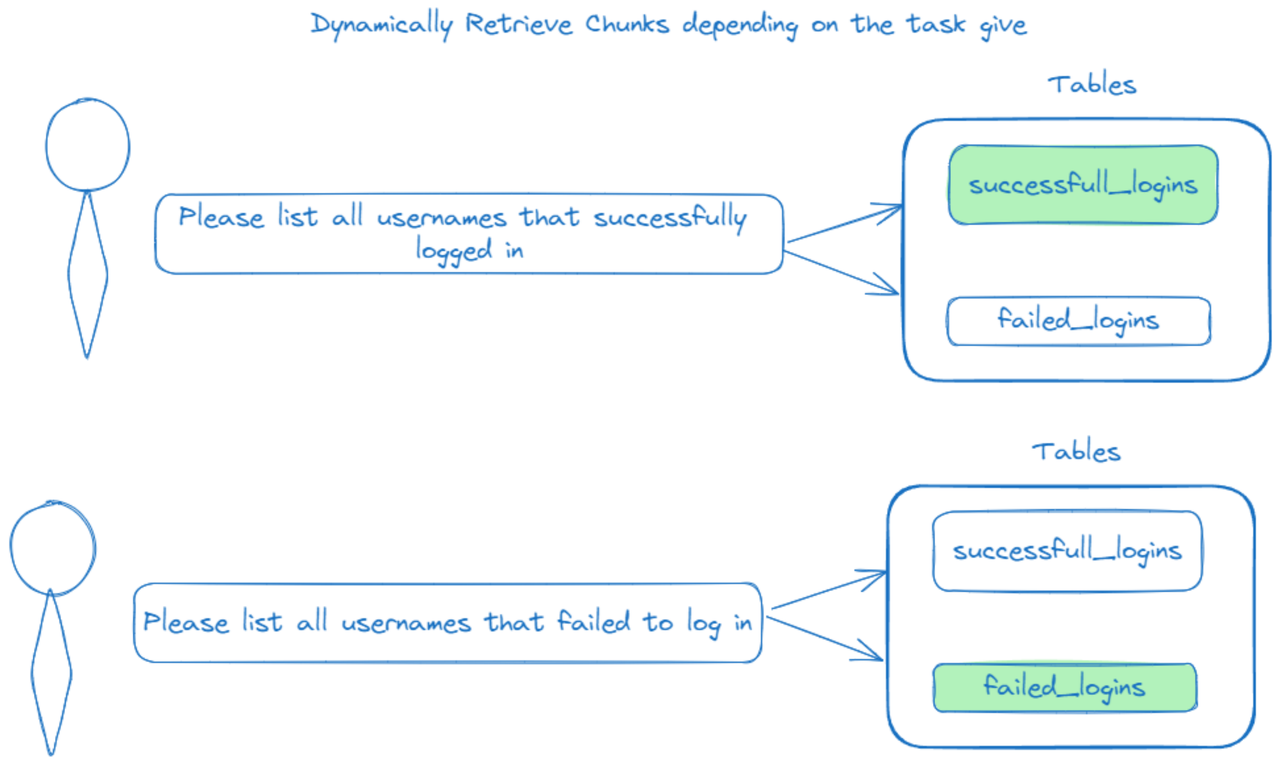

Dynamically Retrieve Chunks depending on the task give
Tables
Please list all usernames that successfully logged in
successfull_logins
failed_logins
Tables
Please list all usernames that failed to log in
successfull_logins
failed_logins

Figure 24: Object Indexing

Finally, for **Row Embedding** introduced as well in the **Dynamically Retrieve Chunks depending on the task given**, each row from each table is embedded. The reason for that, is to make the retriever capable to retrieve only the right row in the right table. Also, to give much more relevant examples to the LLM. For instance, this solve the issue of case sensitivity. For instance, if the input question is to retrieve a specific username, let's say "rain" but inside the table successful logins, it's written "RAIN", it's important to be able to give relevant nodes from this table. Without the Row-embedding, it's only possible to give random rows as relevant nodes which might not fit in this case.

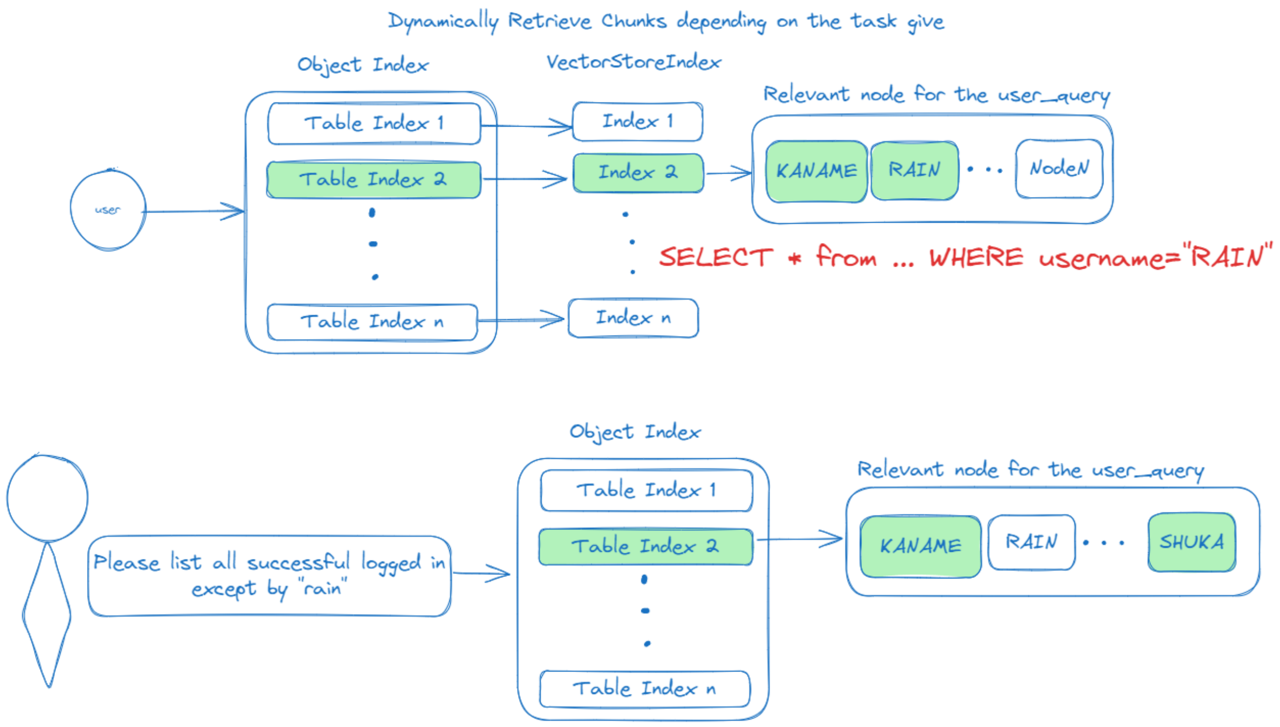

Dynamically Retrieve Chunks depending on the task give
Object Index
VectorStoreIndex
Relevant node for the user_query
Table Index 1
Table Index 2
Table Index n
Index 1
Index 2
Index n
KANAME
RAIN
NodeN
SELECT * from ... WHERE username="RAIN"
user
Object Index
Please list all successful logged in except by "rain"
Table Index 1
Table Index 2
Table Index n
Relevant node for the user_query
KANAME
RAIN
SHUKA

Figure 25: Row Embedding

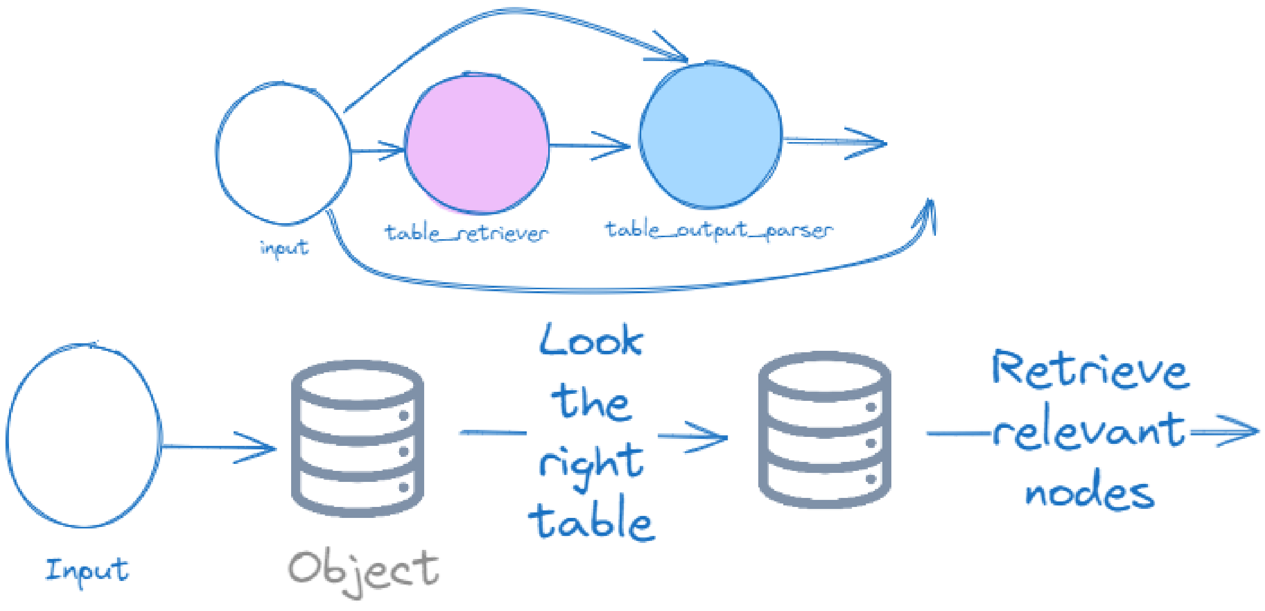


Figure 26: Table Retriever

7.2.2 Basic Query Pipeline

Others modules are basic query pipeline explained the production RAG part. The plan made here is, first get the relevant nodes, then from this nodes and an input question, create a SQLQuery. Finally execute the SQLQuery and return the SQLResponse.

7.3 Implementation Choices

7.3.1 Robustness purposes

For more robustness in the application, a retry method has been used in this application. For instance, if the database is empty, then, the generation of the database is called once again, until 3 attempts. To do so, in the output of the program generated, a function, that drop all tables that are empty, is called after execute the whole program. Having empty tables dropped does also improve the quality of answer, since the analyzer won't make any request to an empty table.

Finally, each tables created is forced to have an id as primary_key, and to not use any kind of foreign_key, in order to make the row insertion easier. Yet, without foreign_key, it is still possible to make some complex SQL request, so this limitation might not be a real issue.

However, this type of robustness comes with a cost, where although the application will generated more often responses, it might be wrong answer. The only way to check, is to have a human-judgment about the whole process that the input question went through to see whether or not the answer is correct.

7.3.2 Performance of the application

To improve the efficiency of this program, the instruction prompt often comes with a python code template, where the only thing that the LLM needs to do is to fill the important section of code missing. Indeed, for instance if we are using python code and SQL, one framework that is a must is SQL Alchemy, and the coding pattern is often the same.

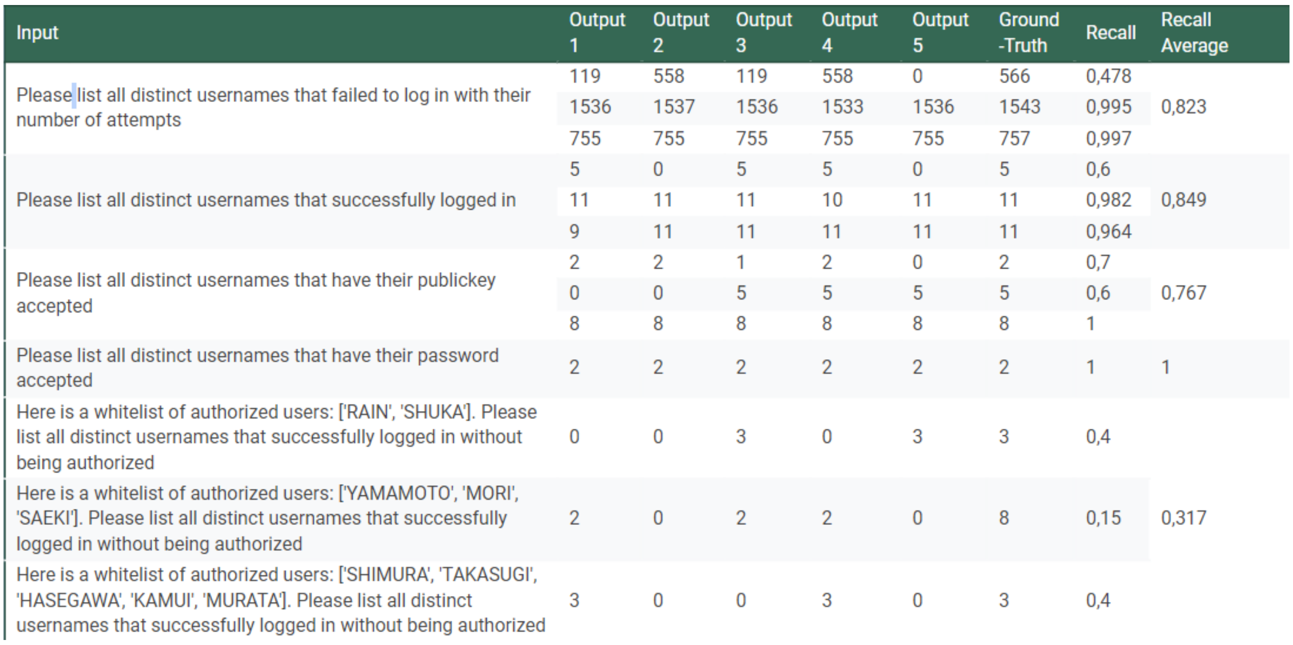
8 Results

8.1 Correctness Evaluation

The LLM model used for this evaluation is: **Llama3-8B**, with the embedding model: **BAAI/bge-small-en-v1.5** and using python code. Besides, the metric used for Correctness is the **Recall** metrics because, the output is composed of several classes of usernames. All of evaluation was done on 3 different files of SSH logs.

8.1.1 Chunk sizes without Re-Ranking

Here, the total average for recall with the whitelist is: 75.12% and without is: 85.60%.

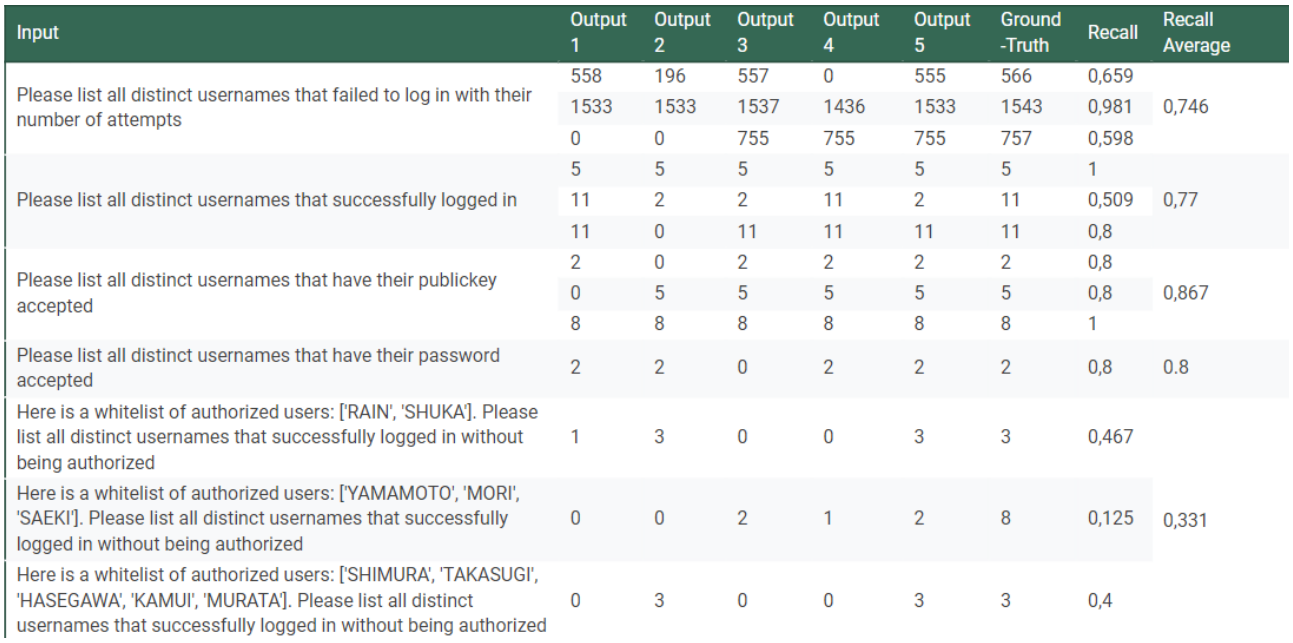


Input	Output 1	Output 2	Output 3	Output 4	Output 5	Ground -Truth	Recall	Recall Average
Please list all distinct usernames that failed to log in with their number of attempts	119	558	119	558	0	566	0,478	
	1536	1537	1536	1533	1536	1543	0,995	0,823
	755	755	755	755	755	757	0,997	
Please list all distinct usernames that successfully logged in	5	0	5	5	0	5	0,6	
	11	11	11	10	11	11	0,982	0,849
	9	11	11	11	11	11	0,964	
Please list all distinct usernames that have their publickey accepted	2	2	1	2	0	2	0,7	
	0	0	5	5	5	5	0,6	0,767
	8	8	8	8	8	8	1	
Please list all distinct usernames that have their password accepted	2	2	2	2	2	2	1	1
Here is a whitelist of authorized users: ['RAIN', 'SHUKA']. Please list all distinct usernames that successfully logged in without being authorized	0	0	3	0	3	3	0,4	
Here is a whitelist of authorized users: ['YAMAMOTO', 'MORI', 'SAEKI']. Please list all distinct usernames that successfully logged in without being authorized	2	0	2	2	0	8	0,15	0,317
Here is a whitelist of authorized users: ['SHIMURA', 'TAKASUGI', 'HASEGAWA', 'KAMUI', 'MURATA']. Please list all distinct usernames that successfully logged in without being authorized	3	0	0	3	0	3	0,4	

Figure 27: Table Retrieving results without Re-Ranking

8.1.2 Chunk sizes with Re-Ranking

Here, the total average for recall with the whitelist is: 70.28% and without is: 79.58%. Less compared to without re-ranking, which is an issue. One possible reason for that, is that the LLM used is too weak to really re-ranked the nodes retrieved.



Input	Output 1	Output 2	Output 3	Output 4	Output 5	Ground -Truth	Recall	Recall Average
Please list all distinct usernames that failed to log in with their number of attempts	558	196	557	0	555	566	0,659	
	1533	1533	1537	1436	1533	1543	0,981	0,746
	0	0	755	755	755	757	0,598	
Please list all distinct usernames that successfully logged in	5	5	5	5	5	5	1	
	11	2	2	11	2	11	0,509	0,77
	11	0	11	11	11	11	0,8	
Please list all distinct usernames that have their publickey accepted	2	0	2	2	2	2	0,8	
	0	5	5	5	5	5	0,8	0,867
	8	8	8	8	8	8	1	
Please list all distinct usernames that have their password accepted	2	2	0	2	2	2	0,8	0.8
Here is a whitelist of authorized users: ['RAIN', 'SHUKA']. Please list all distinct usernames that successfully logged in without being authorized	1	3	0	0	3	3	0,467	
Here is a whitelist of authorized users: ['YAMAMOTO', 'MORI', 'SAEKI']. Please list all distinct usernames that successfully logged in without being authorized	0	0	2	1	2	8	0,125	0,331
Here is a whitelist of authorized users: ['SHIMURA', 'TAKASUGI', 'HASEGAWA', 'KAMUI', 'MURATA']. Please list all distinct usernames that successfully logged in without being authorized	0	3	0	0	3	3	0,4	

Figure 28: Table Retrieving results with Re-Ranking

9 Future Work and Conclusion

9.1 Future Work

One method improving the retrieval process that wasn't implemented is **Optimizing context embedding**. The reason is that, generated a whole QA dataset for the ssh logs dataset is too long. Indeed, one mistake that has been done during the fine-tuning, is that the generated QA dataset had its chunk sizes on 1024 instead of 75. Therefore, the question generated are really bad for the models to be fine-tuned on. Eventually, with more resources and time, one way to improve the application is therefore to first generated a QA dataset with a small chunk sizes and then fine-tuned the embedding model on this.

For the task about using a whitelist, one way to improve its **Recall** is to transform the whitelist given into some SQL database before starting the database creation and analysis. Meaning, to add by hand the whitelist to the database generated itself, then thanks to the **Object Indexing**, for tasks where the whitelist are not needed, they won't be using this table, so the recall would remain the same for these kind of tasks while for the one which needs it, it should perform better. Indeed, it should work the same as splitting the input question into two input one for database and the other one for analysis.

The application is also only capable to do some extraction and do statistical report, but not really analyze the whole document itself. For instance, if one's asks the following question: **Please list all users that tried to brute force attack**, the LLM itself will try to retrieve words that matches with "brute force attack" which doesn't appear in the logs. One way to do so might be to ask the application to store logs from a specific timestamp and then, to retrieve the logs message from one user that tried and failed more than X attempts.

Creating an UI is also very important. Indeed, since it is using an LLM, which answer can be sometimes wrong because of hallucination, a human must always verify the output by hand to check if there was no issue during the whole process. The UI must therefore display the three importants steps which are: the relevant nodes that was retrieved, the whole program created by the database and then finally, check the process of the ssh analyzer.

9.2 Conclusion

The application is pretty capable to extract with a good recall some basic information such as generating a statistical report often correct. However, still struggle to do some tasks such as for the whitelist. Then, since it's using an LLM, which is a tool where its answer must always be verified by a human like ChatGPT answers, it will be very important to give to one's users the whole process of database, analysis and retrieval very single time to check the faithfulness of one generated answer.

10 Bibliography

References

[1] 3Blue1Brown. Attention in transformers, visually explained | chapter 6, deep learning. https://www.youtube.com/watch?v=eMlx5fFNoYc&t=640s, 2024.

[2] 3Blue1Brown. But what is a gpt? visual intro to transformers | chapter 5, deep learning. https://www.youtube.com/watch?v=wjZofJX0v4M&t=845s, 2024.

[3] Paul Graham. Essay. 2021. https://raw.githubusercontent.com/run-llama/llama_index/main/docs/docs/examples/data/paul_graham/paul_graham_essay.txt.

[4] Prompt Engineering Guide. Retrieval augmented generation (rag) for llm. 2024. https://www.promptingguide.ai/research/rag.

[5] Jerry Liu. Jerry liu–llamaindex – practical data considerations for building production-ready llm applications. https://www.youtube.com/watch?v=g-VvYLhYhOg, 2023.

[6] Jerry Liu. Build agents from scratch (building advanced rag, part 3). https://www.youtube.com/watch?v=T0bgevj0vto&t=1677s, 2024.

[7] Jerry Liu. Introduction to query pipelines (building advanced rag, part 1). https://www.youtube.com/watch?v=CeDS1yvw9E4&t=795s, 2024.

[8] Jerry Liu. Llms for advanced question-answering over tabular/csv/sql data (building advanced rag, part 2). https://www.youtube.com/watch?v=L1o1VPVfbb0, 2024.

[9] Llama-Index. Framework for building context-augmented llm applications. 2023. https://docs.llamaindex.ai/en/stable/understanding/using_llms/using_llms/.

[10] AI Makerspace. High-performance rag with llamaindex. High-performanceRAGwithLlamaIndex, 2023.

[11] Christopher Manning et al. Stanford cs224n nlp with deep learning | winter 2021. https://www.youtube.com/watch?v=rmVRLeJRkl4&list=PLoROMvodv4rMFqRtEuo6SGjY4XbRIVRd4, 2021.

[12] Ehsan Minaee et al. Large language models: A survey. 2024. https://arxiv.org/pdf/2402.06196.

[13] Andreas Stöffelbauer. How large language models work, from zero to chatgpt. 2023. https://medium.com/data-science-at-microsoft/how-large-language-models-work-91c362f5b78f.

[14] Barsha Rani Swain. Embeddings/word vectors • langchain in chains. 2024. https://medium.com/@varsha.rainer/embeddings-word-vectors-langchain-in-chains-9ba4fd9f4221.

[15] Ravi Theja. Evaluate rag with llamaindex. 2023. https://cookbook.openai.com/examples/evaluation/evaluate_rag_with_llamaindex.

[16] Ashish Vaswani et al. Attention is all you need. 2017. https://arxiv.org/abs/1706.03762.

[17] Lei Wang et al. Plan-and-solve prompting: Improving zero-shot chain-of-thought reasoning by large language models. 2023. https://arxiv.org/abs/2305.04091.

11 Appendix

- Github: `https://github.com/Haruyukis/rag-application`
- Presentation: `https://docs.google.com/presentation/d/1OmthhYtuSzK7IRd542H8Viu-tSXcrDQqXGySiNhWS30/edit?usp=sharing`