In this methodology, we used the LLM **'Mistral-Nemo-Instruct-2407'** and the embedding model **'sentence-transformers/all-mpnet-base-v2**'. We chose the all-mpnet-base-v2 embedding model from sentence transformers as it provides the best quality, i.e., it has the highest score in encoding sentences, in performing a semantic search, and highest average performance compared to all Sentence Transformer embedding models. (https://www.sbert.net/docs/sentence_transformer/pretrained_models.html)

The **'create_connection'** function connects to a PostgreSQL database by taking parameters for database configuration. If the connection is successful, it prints 'Connection to PostgreSQL DB successful'.

Following this, the **'placeholder_data'** function executes multiple SQL queries (customer_query,transaction_type_summary,alert_query,transaction_query, detection_query, and transaction_summary_query) to gather comprehensive information about the customer, their transactions, alerts, and rule violations. We store specific columns containing customer details, transaction summaries, alert specifics, and detection information from these database tables in a structured dictionary called 'data'.

As the next step, we load and process the 'Alert Narrative' file. In the function **'load_and_parse_document',** we input the PDF file and split it into logical sections based on predefined markers, i.e., in this case, headers. Based on the structure of the Alert Narrative Document, we observe six sections. The title header "Alert Narrative" contains the 'Alert number' and 'create date' of the alert; the 'Focal Entity' contains the 'Focal Entity', i.e., the customer name, CIN: Customer Identification Number, and 'Review Scope': the period for which this alert has been generated. The 'Determination/Rationale' section includes a statement on reviewing internal and external sources conducted on the customer and the rules the customer has broken as part of this alert. Following this, there are two untitled sections, one about the customer's transactions and details that led to the alert and associated with the alert and another about the reason why the alerts were generated.

To incorporate the information from these sections into our SAR narrative, we added two additional headers to the document called "Transaction Details" and "SAR Recommendations." This step aims to isolate the transaction details, which are in "Transaction details:" and the SAR recommendation, which is in "SAR Recommendation:" and pass these specific sections through RAG. The advantage of passing a specific section of the document through RAG rather than the entire document is to :

(1) Reduce hallucinations
(2) Reduce RAG latency of answering questions from irrelevant document sections.
(3) Streamline a minimum viable solution to achieve the objective of this project.

Now, the node parser will identify headers mentioned in the 'section_markers' dictionary, grouping the text from each document part into its relevant parts. These are "ALERT NARRATIVE", "Focal Entity:", "Determination / Rationale:", "Transaction details:" and "SAR Recommendation:". However, "Determination/Rationale:" also contains the suspicious activities. As we will need to use the specific suspicious activities later on for RAG, we will store these in a separate list called ‘suspicious activities’. For the purpose of this project, these include 'Cash Structuring,' 'Large Wire from a High-Risk Jurisdiction,' 'Concentration Account,' 'Rapid Movements of Funds', and 'New Account Rule.' We store this list of rules in ‘suspcious_keywords.'

We need to find an efficient method to parse the Alert Narrative document. There are several types of node parsers. After trying various node parsers, we found the node parser ‘SentenceSplitter’ from the text_splitter library in the llama Index to be most helpful, for this use case (https://docs.llamaindex.ai/en/stable/module_guides/loading/node_parsers/modules/). SentenceSplitter helps process the text while maintaining context through chunk overlaps and chunk size parameters. Each section is stored in a mapping structure with its full text and chunks, making it easy to access specific parts of the document later. We will use full_text and not the chunks for our prompts since the prompt considers the entire part we extract as one chunk which would not be helpful for our prompts. Hence, we will use the full_texts version of the sections.

Next, we run the function **‘unique_values_for_prompt’,** which creates a dictionary called ‘unique_values’ to store all the unique_values of columns from our SQL queries. We also store the “transaction_details” and “sar_narrative” sections of the Alert Narrative we parsed.

After this, the function **‘generate_sar’** is processed. Within this function, we store the function ‘placeholder_data’ results for the customer_id and alert_id we want. In the variable ‘document_sections’, we store the results of the function ‘load_and_parse_documents’. In ‘formatted_data’, we store the results of the function ‘unique_values_for_prompt’, passing in the above variables ‘data’ and ‘document_sections’. In ‘suspicious_activities_list,’ we store the suspicious activities from ‘document_sections’.

In the list ‘formatted_rules,’ we im – m. dentify suspicious activities from the document sections and format them according to specific rules. Each activity (such as "Cash Structuring" or "Rapid Movements of Funds") is formatted with a specific phrase structure and numbered sequentially. This creates a standardized list of suspicious activities for the report.

Now that we have all the raw data prepared for the prompt, we decide on the structure of the prompts to generate the report.

After analyzing the sample SAR Report, we noticed that both sample SAR reports that were shared (LLM SAR 1 & LLM SAR A-2) follow a specific pattern. We could identify 5 critical sections in the SAR:

1. **Introduction section** *(Paragraph 1)***:** containing details of the bank filing the report and a summary of alerts for which the bank is filing this SAR.
   (Static)
2. **Transaction details section** *(Paragraph 2)***:** Describing transaction details and dates on which all the incoming and outgoing transaction alerts have been raised (**Dynamic)**

3. **Customer's KYC and Internal/ External** research section *(Paragraph 3)* - investigations which contradict validation & reasoning for the above transactions. **(Dynamic)**
4. **SAR recommendation** *(Paragraph 4)***:** Why a SAR is being filed based on **(Dynamic)**
5. **Concluding section** *(Paragraph 5)***:** Bank's Contact Details for inquiries on the SAR: **(Static)**

We use a template structure for Paragraph 1 and Paragraph 5 using 'Prompt_1' and 'Prompt_5 'respectively. As these are static parts of the SAR, a minimum viable solution in automating SAR generation may not require these sections to be generated through RAG. Additionally, it can be argued that generating this section through RAG would utilize unnecessary computation resources and electric power, which eventually translates to additional costs. Furthermore, As a bank we understand that data in the SAR report needs to be accurate and running this section through RAG may create hallucinations. Hence, using placeholders and static text may be the most suitable solution to generate these report sections, as this approach eliminates hallucinations.

To generate paragraphs 2, 3, and 4, we create three separate prompts and run them through RAG to retrieve and generate them from the Alert Narrative PDF. We also create two separate vector indices: one for transaction details and another for SAR recommendations. These indices enable semantic search and context-aware responses when generating different parts of the report. Each index is created from the relevant section of the document.

After these three different query engines are configured with specific parameters, First engine generates a chronological summary of transactions; the Second engine focuses on KYC information and research findings; the Third engine handles suspicious activity reporting. Each query engine generates its portion of the report based on specific requirements:

- 'Prompt_2' is based on 'query_engine_1' and vector index 'index'. It Summarizes transaction details chronologically.
- 'Prompt_3' is based on 'query_engine_1' and vector index 'index'. It formats KYC information and internal/external research findings.
- 'Prompt_4' is based on 'query_engine_2' and the vector index 'index_2'. Based on the alert narrative, it lists suspicious activities and their implications.

We concatenate the individual outputs of the five prompts into the string 'combined_response'. The final output is a structured SAR report that combines database information, document analysis, and formatted responses in a standardized format suitable for regulatory reporting purposes.

While creating the query engine, we passed certain as_query_engine parameters, such as 'similarity_top_k', 'custom_vars', 'response_mode', and 'response_kwargs'.

The ***'similarity_top_k'*** controls how many of the most similar chunks/documents to retrieve. A higher number provides more context for the response but potentially less relevant information, whereas a lower number provides a more focused response but might miss context. We kept the 'similarity_top_k' as 2, wanting a precise response.

The ***'custom_vars'*** dictionary defines custom variables used or accessed in the prompt. This narrows the RAG context and allows for answering questions about a specific section or paragraph of a document.

> For 'query_engine_2', we have set the custom_var to include the 'customer_name', extracted from the database, and text as the full-text version of the transaction_details. This is used in Prompt_2.
>
> For 'query_engine_3',, we have set the custom_var to include the 'customer_name' extracted from the database; and text as the full-text version of the sar_recommendations. This is used in Prompt_3.

The ***'response mode'*** parameter controls how the LLM processes and formats its responses. There are different types of response modes:

- The **'refine'** mode – generates and improves an answer by sequentially reviewing each retrieved text chunk. This entails making an individual LLM call for every Node/retrieved chunk.
- The **'compact'** mode combines all retrieved information into a concise response. It concatenates the chunks beforehand, resulting in fewer LLM calls.
- **The 'Tree_summarize'** response mode first retrieves relevant content, creating a structured and hierarchical response. It prompts multiple times to ensure all concatenated chunks are queried, ultimately generating several answers that can recursively serve as chunks in a tree_summarize LLM call. This process continues until only one chunk remains, resulting in a single final answer. This response builder merges text chunks recursively and summarizes them in a bottom-up manner, akin to constructing a tree from the leaves to the root.
- The **'accumulate'** response mode gathers all relevant information and provides more detailed, extended responses, preserving more context.
  - (https://docs.llamaindex.ai/en/stable/module_guides/querying/response_synthesizers/)

We tested all four methods and found that 'tree_summarize' and 'compact' best suit our use case. We chose 'compact' because 'tree_summarize' takes longer to run the query but generates the same output.

The ***'response_kwargs'*** (response keyword arguments) controls the response format. This include parameters such as:

- **'temperature':** This parameter controls the randomness of the models output.
- **'separator'**: This parameter enables adding spacing, such as printing out new lines between sections' responses.
- **'truncate' (true or false) -** can be used to determine whether or not to truncate responses.
- **'template'** – can be used to follow a custom response template.
- **'streaming'** (true or false) : When set to True, this parameter enables the response to be streamed, allowing for real-time or incremental output generation. This can be useful for applications that require immediate feedback or continuous data processing. (docs.llamaindex.ai)

Other parameters we tried were:

- ‘max_tokens’, sets the maximum tokens in the response.
- ‘truncate’ can be set to True or False,
- ‘verbose’ = True to see the pattern of the execution, the response generation steps and prompts used, and the chunks retrieved.

Finally, we concatenate the results of all individual components (Prompt_1, Prompt_2, Prompt_3,Prompt_4, Prompt_5) into a single cohesive report. The final output is a structured SAR report that combines database information, document analysis, and formatted responses in a standardized format suitable for regulatory reporting purposes. Throughout its execution, the function maintains error handling and ensures proper resource cleanup; if any errors occur, they are caught and reported appropriately.

Recommendations and Next Steps:

As part of next steps, we would recommend the following suggestion:

- Understand how to use ‘context_template’: This parameter defines the template used for formatting the context in the response.

- Using ‘filters’ sets the metadata filters for retrieval. This enables us to narrow down the search results based on specific metadata attributes associated with the documents or nodes in your index. By applying these filters, you can ensure that only documents meeting certain criteria are retrieved, which enhances the relevance and accuracy ofthe search results.
- Analyse and test with other Alert Narratives, Customer and Alert numbers using the same prompts.
- Analyse how to shorten the context length, context window based on the instructions in each prompt to RAG without changing the output
- Analyse how to utilize other parameters of the as_query_engine function.