GenAI RAG Project Documentation for Targeted Victory

May 30, 2024

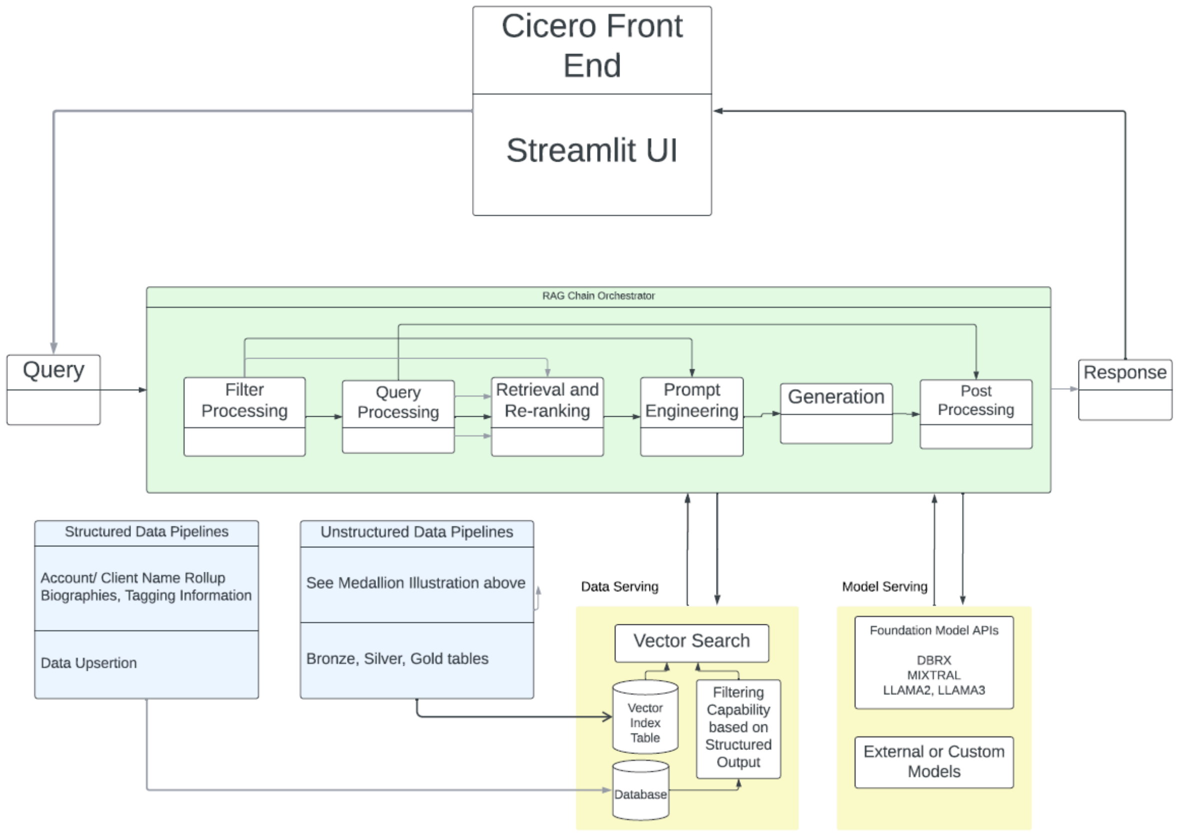
Targeted Victory POC: Alisa Brady, Alex Chang, Ted Hall
Lovelytics team: Wesley Gea and Joon Yoon

Background

This project utilizes GenAI to help generate meaningful Language Model based responses for Targeted Victory using the instruction dataset of 80000 data points by retrieving similar unstructured data using structured data to filter the relevant context and using this to help LLM learn to generate higher quality responses the users desire.
Over the course of 10 weeks, Lovelytics implemented a Retrieval Augmented Generation (RAG) based Large Language Model (LLM) solution focusing on full implementation with a comprehensive MLOps framework using Databricks. The project encompasses RAG implementation, serving continuous evaluation, focus on performance improvement, and assisting with the integration of the RAG end point to Targeted Victory's front end interface. An operational Data Pipeline was built complete with data pre-processing, CI/CD for Model Deploy, and Serving and Evaluation. RAG implementation of open-source LLMs using existing instruction dataset (expected to be around 80K values) to immediately update the model serving endpoint, define qualitative performance metrics.

Retrieval Augmented Generation (RAG) is a generative AI design pattern that enhances a large language model (LLM) by integrating it with external knowledge retrieval. This integration is crucial for connecting real-time data to generative AI applications, significantly improving their accuracy and quality by using real-time data as context during the inference phase.

RAG architecture

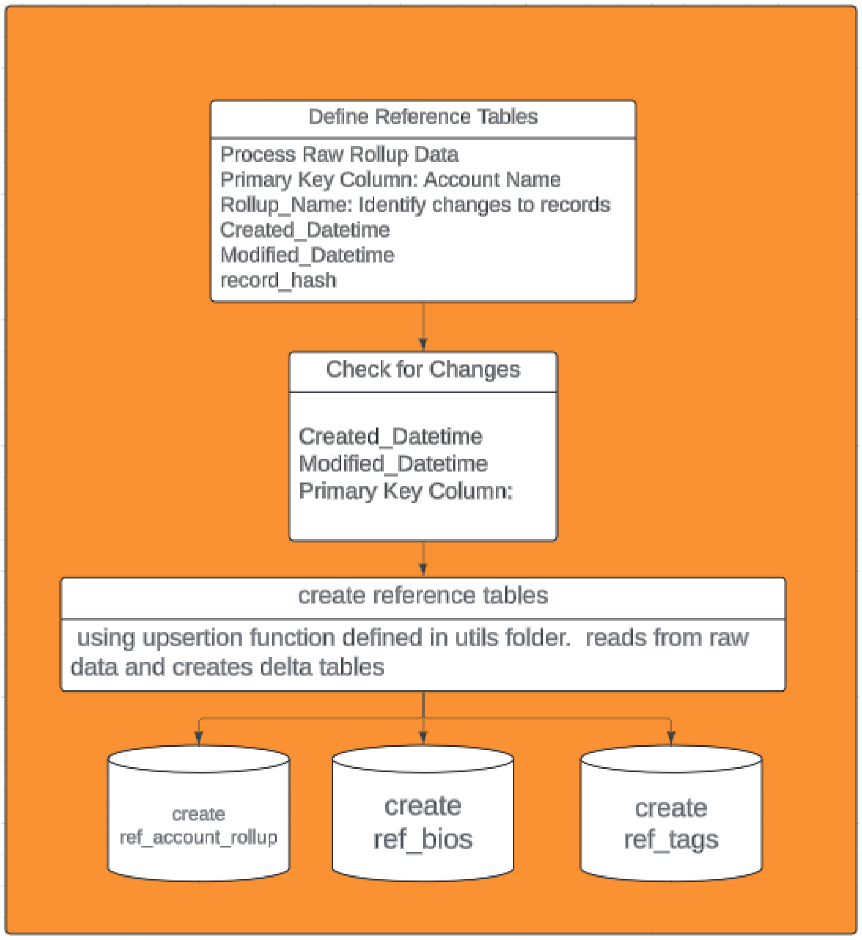


The RAG pipeline consists of two types of data, structured and unstructured data. RAG deals primarily with handling the unstructured component well in order to extract meaningful information out of it using vectorized indexing. Structured data is also used to augment and/or filter the final desired results using the RAG approach.

Processing Unstructured Data

1. Data Ingestion

Text data from Snowflake is imported into Databricks via SQL query using the external data connection called *snowflake*. The SQL query, *Text_Model_Query*, reads in data from the foreign table and catalog, *prod.business.project_results*. Basic filtering is done on the data and then written into Unity Catalog as the Delta table *models.text_model.sf_data*. This table can be treated as the bronze layer of data.



2. Data Cleaning and Parsing

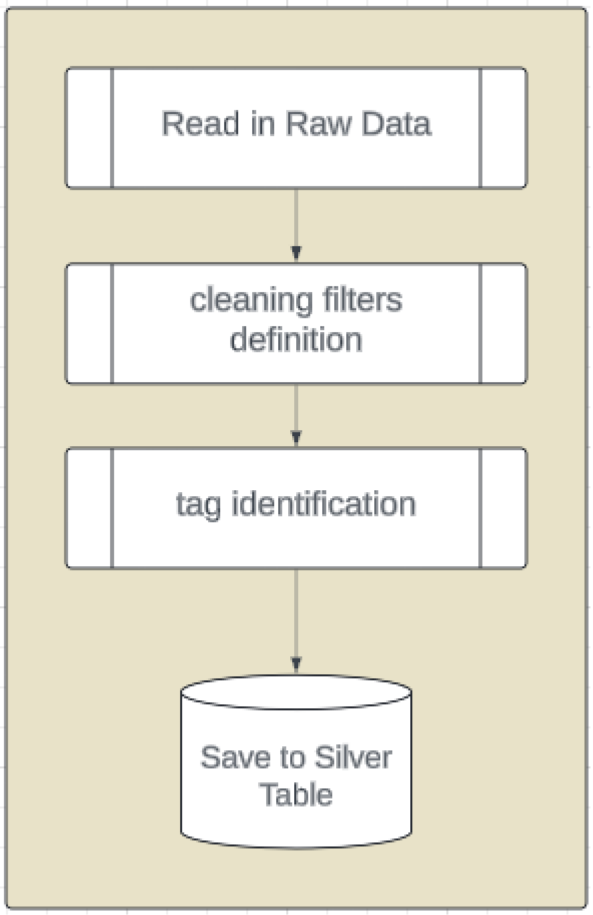
Data_Processing/Data Cleaning & Tagging Notebook

From the bronze table, the data is cleaned of any unnecessary characters such as unicode characters, characters repeating more than a set amount of times, and special text patterns. All cleaning is done using lists of regexes, where each item in the list is a separate set of characters to remove or filter out. All of the lists containing regexes to be replaced or filtered out start with the element "DUMMY_VALUE". This is to make it easier to remove or add new items to these objects. All one needs to do to add an item is to start a new line within the array or dictionary, start the line with a comma (,), and insert the desired value. To remove an item, simply delete the no longer need item including the comma. Commenting out lines works the same, just add a # at the beginning of the line and that's it. Then the data is strictly filtered for texts shorter than 1100 characters, which is small enough to be processed by the LLM's context limitations. This text length filter value is parameterized and can be changed when the notebook is run.

The text data is then parsed and tagged based on its metadata and contents. Topics, tones, and ask types are marked for each record based on a reference table created from structured data. Each tag type follows a specific pattern:

- All ask tags have the word ask as the second word (i.e. `string.split("_")[1]` is the string `ask`)
- All tone tags end with `tone` (e.g. `urgency_tone`)
- Every other tag ends with hook (e.g. `birthday_hook`)

These processes combined create the silver table, *models.lovelytics.silver_cleaned_tagged*.



3. Final Assembly, Embedding, & Indexing

Data_Processing/Text Assembler Notebook

Using the silver table, the cleaned text is assembled into its desired final state using a YAML text template located in the folder *Data_Processing/text_formats*. The YAML file informs the code how it should format the final documents/texts that will be used for either fine-tuning or for RAG prompting. These YAML files should all be located under the **text_formats** folder and follow the format

```
prompt_variables:
  - var 1
  - ...
  - var n
prompt_structure: |
  Example string %s (...) %s
```

Where there are as many variables as there are %s in the prompt structure. i.e. if there are n variables then there are n %s's in the prompt_structure. The pipe character in prompt_structure is how to declare a multi-line string in YAML format, it will not appear when the string is read in.

Pyspark's format_string function functions like Python's printf, which is why the %s syntax is used rather than the curly bracket syntax.

The code below formats strings by each component, letting it dynamically remove any parts that don't have a variable value. This means that if a record doesn't have a value for that variable, then in the final assembled text the entire portion of text associated with that variable will be removed. So for example, if this is our prompt structure and components

```
### Prompt Structure ###
A {Text_Length} {Ask_Type} text message from {Client_Name} about {Topics} written
with an emphasis on {Tones}:

{Clean_Text}

### Prompt Components ###
['A %s', ' %s', ' text message from %s', ' about %s', ' written with an emphasis on
%s', ':\n\n%s']
```

If tones is empty for a record, they would look like this

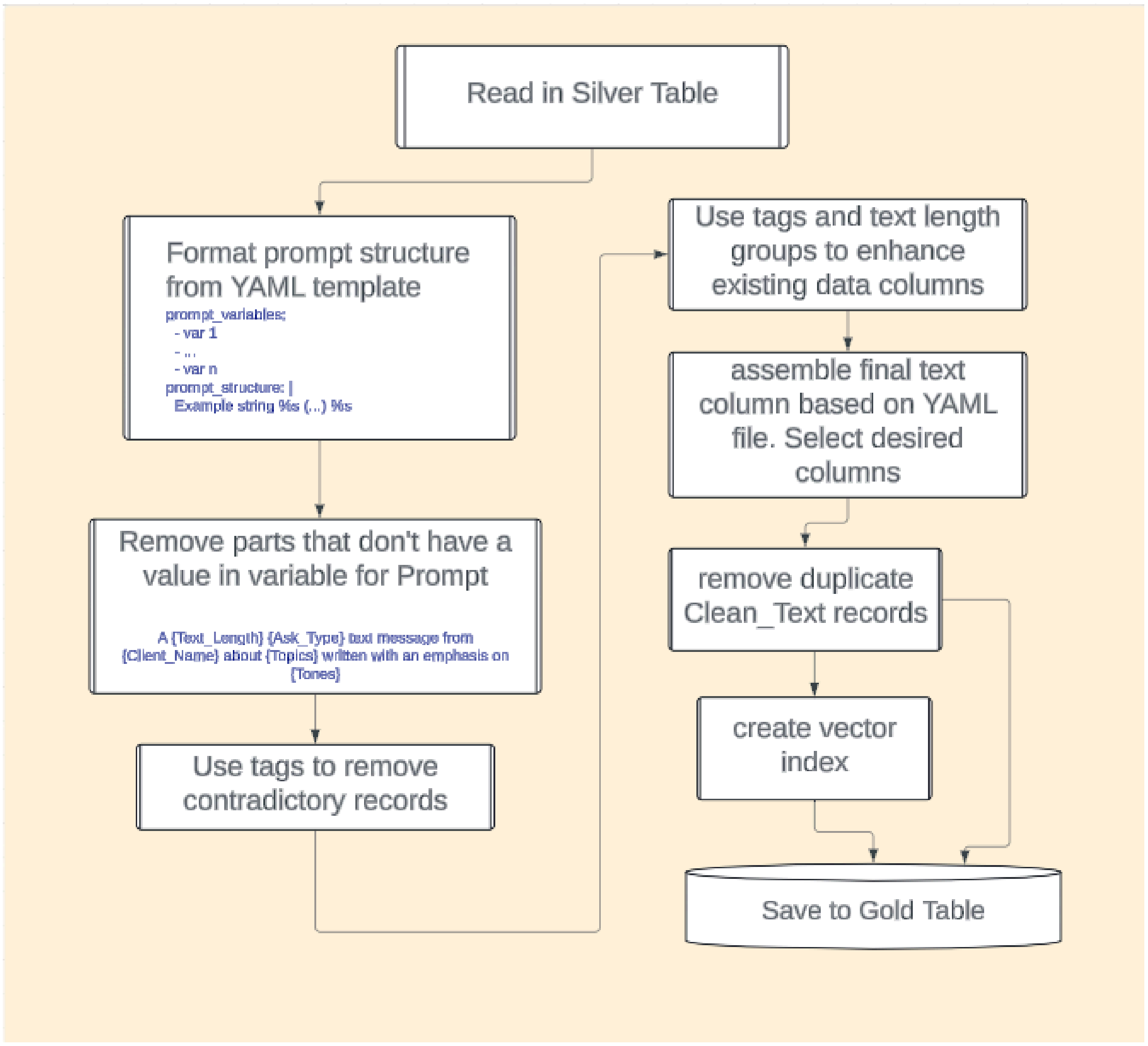
```
### Prompt Structure ###
A {Text_Length} {Ask_Type} text message from {Client_Name} about {Topics}:

{Clean_Text}

### Prompt Components ###
['A %s', ' %s', ' text message from %s', ' about %s', ':\n\n%s']
```

Using the tags created in the silver layer, the data is further refined by removing contradictory records, enhancing existing columns such as ask types and fundraising types, and by determining text length grouping. The final text column is assembled based on the YAML file and using the dataframe columns as the string components. When the dataframe is finalized, the desired columns are selected and any duplicate Clean_Text records are dropped before writing out to the gold table, *models.lovelytics.gold_text_outputs*.

Finally, the vector search endpoint *rag_llm_vector* is created if it doesn't already exist. Then the index table *models.lovelytics.gold_text_outputs_index* is created (if it doesn't already exist). Otherwise, if it exists then a sync is triggered to sync the index table with the data in the gold table.


Read in Silver Table
Format prompt structure from YAML template
prompt_variables;
- var 1
- ...
- var n
prompt_structure: |
Example string %s (...) %s
Remove parts that don't have a value in variable for Prompt
A {Text_Length} {Ask_Type} text message from {Client_Name} about {Topics} written with an emphasis on {Tones}
Use tags to remove contradictory records
Use tags and text length groups to enhance existing data columns
assemble final text column based on YAML file. Select desired columns
remove duplicate Clean_Text records
create vector index
Save to Gold Table

Processing Structured Data

In RAG chain, the structured data is not turned into embeddings but is often used to augment the overall RAG chain in conjunction with the unstructured data. Currently, all structured data are stored as reference tables, ingested and created by the Data_Processing/Create Reference Tables notebook. This notebook makes use of the upsertion function defined in the notebook utils/reusable_funcs_data_processing. Any mention of upsertions, upserting, or data being upserted means that this upsertion function was used to perform a merge, update and insert, into the target table.

1. Account/Client Name Rollup

Account names and client names that are closely related to one another are given mappings to a single name. For example, “Mercury One (TMA Direct)” and “Mercury One (TMA)” both map to “Mercury One - TMA”. In the notebook, these mappings are defined in a dictionary that is then used to create a dataframe. This dataframe is upserted into a Delta table named *models.lovelytics.ref_account_rollup*.

2. Biographies

Account/client biographies are uploaded into Databricks File System (DBFS) as a CSV file. This CSV file is read in as a spark dataframe and then upserted back into the table *models.lovelytics.ref_bios*.

3. Tagging Information

Text tag information are defined in a dictionary in the notebook. The dictionary items take the format of `“<column_name>”: “<regex_pattern>”` with the first item a dummy value that is always dropped. The dummy value makes it easier to add and remove items from the dictionary. As mentioned previously

- All ask tags have the word ask as the second word (i.e. `string.split("_")[1]` is the string `ask`)
- All tone tags end with `tone` (e.g. `urgency_tone`)
- Every other tag ends with hook (e.g. `birthday_hook`)

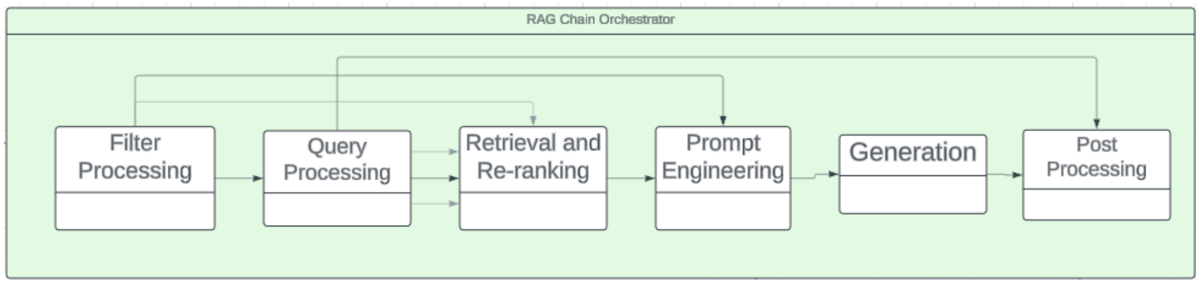
This means that the column names in the tag dictionary must either end with hook, tone, or have ask as their second word. When creating the tag reference table dataframe, new metadata columns are added in such as tag type and tag clean name.

Then once all the data is prepared, the data is upserted into the table *models.lovelytics.ref_tags*.

Data Upsertion

Data upsertion is a cleaner way to maintain changes to a table. Data is only written to the table if there are detected changes either in the form of new records or updates to existing records, thus improving runtimes by not rewriting existing unchanged data. To perform upsertions, both the target table and the update dataframe must contain a column of hash values called "record_hash". This hash value is used to track if there are any changes to the record if it already exists. This function performs upsertions using a spark dataframe onto a specified target table. If the table doesn't exist, it'll write all the contents in the dataframe to the target table instead. Otherwise it'll perform a merge onto the target table, updating the specified columns and inserting any unmatched columns based on provided key column names.

Note that this process does not perform any deletions to the table. If data needs to be deleted, it must be done another way.



RAG Chain Operation

After the index is ready, the RAG chain can be served to respond to questions.

1. Retrieval

a. Filter Combinations

Although similarity searches are still performed on the text itself, we want finer control over selecting documents using metadata filters of decreasing strictness. So for example, a user would ask for output to be generated for client A, ask type B, and topics C and D. So we would first search for documents belonging to client A, ask type B, and whose topics contain C and D. Then, if not enough documents were found, the search would continue but with filters client A, ask type B, and topics containing C. So on and so forth the filters are loosened until enough texts are retrieved from the similarity search.

This filtering method is achieved by assigning weights to each filter type and then generating every possible combination of filters. Each filter type is as follows

- Client
- Ask Type
- Text Length
- Topic
- Tone

For topics and tones, each individual topic and tone is given a weight. So if the topic weight is X, then two topics have the combined weight of 2X and one topic only has a weight of X. Same thing for tones. With every filter combination generated and every filter item given a weight, the set of filter combinations is ordered from highest weight to lowest weight. This gives us the order of importance of filters, with the strictest filters being the most important. When documents are retrieved, it is done so in the order of highest weight filters to lowest weight filters.

b. Embedding Questions

Incoming questions / queries are embedded using the same model that was used for the data embeddings, ensuring that the query and the data are compatible for similarity searches. This embedding isn't done manually, but is bundled together with the similarity search when the vector index is invoked.

c. Vector Search

When it comes time to search for and retrieve text documents, we iterate over the set of filter combinations in descending order of weight/importance. Each individual filter item in the combination is used to perform a hard filter on the document metadata. Then the final set of primary key values are added to a list to prevent retrieving duplicates from future iterations and are collected into a separate list. This separate list of primary key values is iterated through in batches, where each batch is used in the similarity search's filter statement so we only look at records we know match the hard filters we've provided. The similarity search's purpose here is to calculate a text's relevance to the prompt/question we want for outputs. The reason why results are processed in batches is because Databricks vector search client can only handle a maximum of 1024 items in a filter statement, otherwise it will throw an error. So to prevent errors, results retrieved by the initial filtering are batched into groups smaller than 1024.

And in case Databricks vector search fails for any other reason, there is a backup vector search option implemented. FAISS is used instead and performs the same function of returning documents and their similarity score to the prompt.

Once documents are retrieved from the similarity search, only texts that have a similarity score above a set threshold are kept. If the desired number of documents are found, the loop exits.

d. Auxiliary Data Retrieval

In some cases, biographical information may be included at the behest of the user. In this case, it must be fetched from the corresponding reference table which is then inserted into the system prompt.

2. Generation

a. Prompt Engineering

A many-shot prompt template is used following the prompt formatting provided by the Meta Llama 3 documentation. This template begins with a system prompt, consisting of a static string detailing instructions for the model to follow as well as dynamic inclusion of bio information if it is available. Then comes the many-shot reference texts. These texts are treated as part of a chat history with the model, where it is structured in the format of user-assistant responses in a multiple turn conversation. The number of example texts used in this chat history is set by the user, defaulting to 15, and are randomly sampled from a pool of documents created by the previous document retrieval step. The pool contains a number of documents also set by the user, defaulting to 30, and must be at least as many as the number of desired examples in use in the chat history. Then at the end of the prompt template is the question the real user will ask the model. This also takes the form of a prompt (e.g. Please write me a…) and include headline(s) provided by the user if applicable.

Each prompt, including the real user prompt, will make mention of the desired output length. So for example, for short text messages the prompt will include a line saying the text message should be less than 160 characters long.

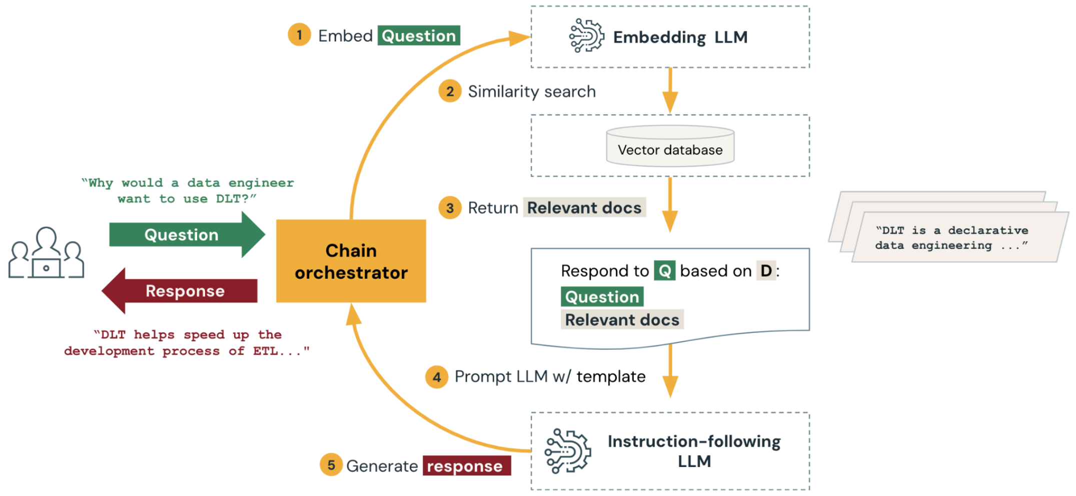
b. Response Generation

As part of the Databricks version of the RAG architecture, three different models are invoked with the prompt. This is useful for comparing different model qualities, although at the moment because we are using the Llama 3 prompt styling, our setup is currently biased towards making the Llama 3 model's output better. The three models in use are all invoked via the Databricks provided endpoints

- Model Name (Endpoint Name)
- DBRX Instruct (databricks-dbrx-instruct)
- Meta Llama 3 70B Instruct (databricks-meta-llama-3-70b-instruct)
- Mixtral-8x7B Instruct (databricks-mixtral-8x7b-instruct)

Right before the LLMs are invoked, the example documents are chosen. For short and medium-length texts, a simple random sample of the document pool is performed. For long texts, a custom sampling function is called that randomizes the order of the document pool and selects, from beginning of the pool to the end, the texts that are most dissimilar from each other. This method is to help create greater output variance so multiple outputs of the same prompt won't look too alike from one another.

Also, for short and medium-length texts, the model is invoked to output a batch of text messages. So for example, if the user wants 5 short text messages, the prompt to the model will ask "Please write me five short text messages". But for long texts, the model is invoked iteratively to generate the number of desired outputs. So if the user wants 5 long text messages, the prompt will ask "Please write me a long text message" and the model will be called 5 times using this prompt. Note that for each iterative call of the model, a different randomized set of example texts will be used as the chat history.



3. Monitoring & Orchestration

When the generation step is finished, all model outputs and the prompts provided to the models are saved to a table, *models.lovelytics.rag_outputs*. To help organize the table, each run's set of generated outputs and prompts are assigned the same batch number. There is also a timestamp attached to each record and this timestamp value is the same across a batch. The text data is stored under the column Output_Content and there is also an Output_Source column that says which model generated the output or if the content is the prompt.

There are two Databricks jobs that orchestrate the data processing and RAG.

- Data Cleaning & Text Assembly
  - This job runs the SQL query, **Text_Model_Query**, that reads in the raw Snowflake data and saves it into Databricks. It's followed by the reference table creation notebook, **Data_Processing/Create Reference Tables**, which is needed for the next step of cleaning and tagging the data. The

    Data_Processing/Data Cleaning & Tagging notebook is called next and lastly is the **Data_Processing/Text Assembler** notebook
  - This job is scheduled to run at 7:30am EDT on Monday to Friday
  - A notification is sent out to Alexander Chang, Wyatt Carpenter, Alisa Brady, and Ted Hall on job failure
  - A notification is sent to Alexander Chang on job success
- Run RAG Prompting
  - This job starts with an if statement to determine if it should run the data processing job first. The if statement evaluates a boolean variable, so simply set the variable to true to run data processing and then RAG, or set it to false to have it only run RAG. For the RAG portion, it calls only the notebook **RAG/Generate Outputs**
  - This job is not schedule to run
  - No notifications are sent out

Both of these jobs run as Alexander Chang in order to inherit his permissions.