**AI-BASED PROMPT ENGINEERING**

**FOR CONTEXTUAL-QUERY RESPONSES**

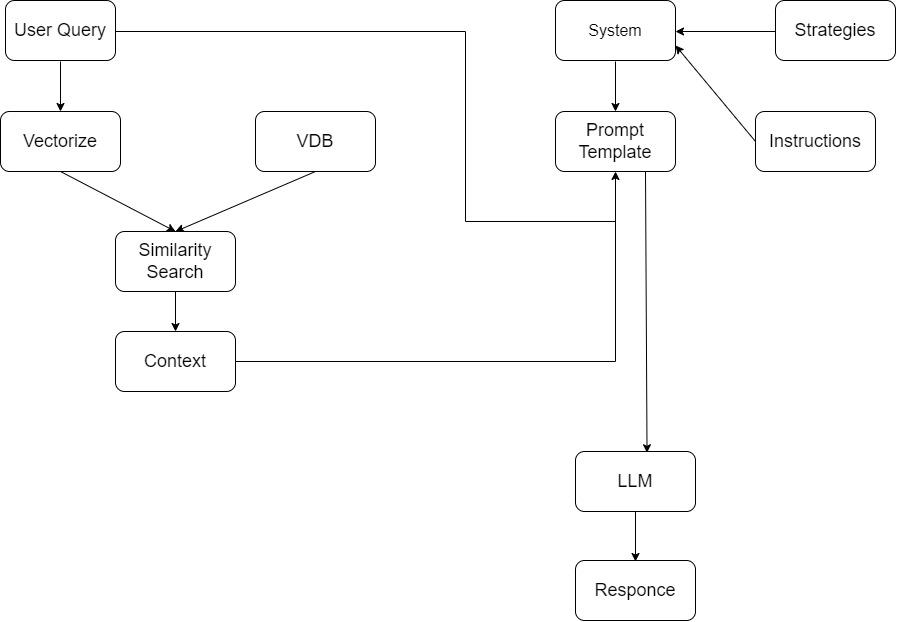
**PRESENTED BY:**

**VENKATA SOMA ADITHYA**

**SURYA TEJA**

**KALYAN KUMAR**

**Approach to the statement:**

****

**Context Retrieval**

**Text Extraction:**

* Extracted raw text from various PDFs.
* Cleaned and preprocessed the extracted text by removing unnecessary characters, symbols, or formatting issues
* Used Libraries like PyPdf2 for the pdf files.

**Text Chunking:**

* Split the extracted text into manageable chunks to facilitate efficient retrieval during query processing.
* We tokenized the extracted text into smaller, manageable chunks using **sentence tokenization**.
* This helps to balance context retention with retrieval efficiency, making it easier to search and return relevant text during retrieval

**Storing:**

* Stored the chunked text in Delta tables for efficient querying and retrieval.
* We created a Data Frame with both the **sentences** and their corresponding **embedding’s** and store this data in a Delta table.
* Each row in the table will have the original sentence (or chunk) and it’s embedding.

**Vector Search:**

* Created the Embedding for the query and searched it across the Data Frame.
* Perform vector search on the stored chunk embedding’s to find the most relevant chunks for the user’s query.
* Perform vector search on the stored chunk embedding’s to find the most relevant chunks for the user’s query.
* We retrieve the Top-K embedding’s.

**Generate Context:**

* Top k similar sentences are combined and returned back as the context to the system.
* This context is used for generating the prompt template according to the query.

**System Design for Optimal Prompt Generation**

The system is designed to generate an optimal prompt for a user's query, ensuring the best possible response from the Large Language Model (LLM). The process involves extracting the intent of the query, selecting an appropriate strategy, designing a strategy for prompt generation, and incorporating context and instructions to guide the LLM.

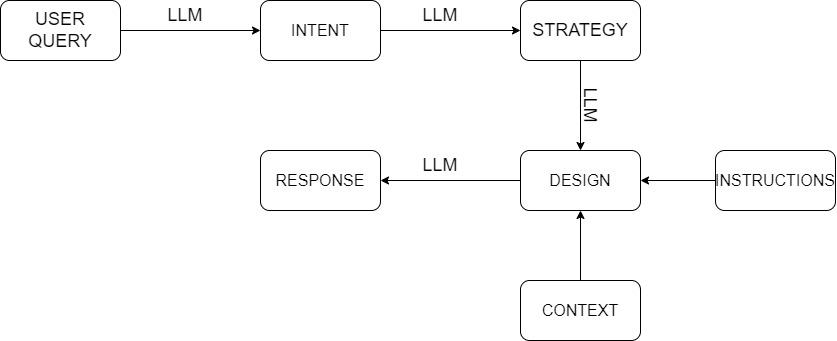
Key Components:

1. Intent Extraction

2. Strategy Selection

3. Strategy Design

4. Final Prompt Generation



**Intent Extraction from Query**

The first step involves understanding the user's query by extracting its intent. This is crucial for identifying the goal behind the query, which will dictate how the prompt should be generated.

Approach:

* LLM Used: databricks-meta-llama-3-1-405b-instruct
* The query is fed into the LLM to infer the intent.
* The system employs specific strategies (such as dynamic prompting, guided context prompts) to ensure the intent extraction is accurate and matches the user's goal.

Objective:

To accurately identify the purpose of the query, whether it’s a request for information, a recommendation, a question, or a summarization, so that further steps can be tailored.

**Strategy Selection**

Once the intent is extracted, the system selects a strategy that aligns with the user’s query. This strategy defines how the prompt should be structured to achieve the best response from the LLM.

Approach:

LLM Used: databricks-meta-llama-3-1-405b-instruct

The extracted intent is passed to the LLM, which analyzes it and suggests an appropriate strategy for prompt generation.

Strategies may include:

* Conversational Prompting: For dialogue-driven queries.
* Instruction-Based Prompting: For tasks requiring detailed steps.
* Contextual Prompting: For queries needing additional context.

Objective:

To select a strategy that optimizes the way the query is framed, ensuring that the LLM responds with the most relevant and precise information.

**Strategy Design**

After selecting the strategy, the system designs it based on the LLM’s suggestions. This involves instructing the LLM to detail the specific steps or guidelines needed for generating the prompt.

Approach:

* LLM Used: databricks-meta-llama-3-1-405b-instruct
* The system provides the selected strategy name to the LLM, along with instructions to produce a strategy design
* This design includes how the prompt should be structured, what elements to emphasize, and any additional steps needed to optimize the prompt generation process.

Objective:

To create a clear, structured strategy design that defines how to construct the final prompt to guide the LLM’s response effectively.

**Final Prompt Generation**

The last step involves constructing the final prompt by incorporating additional context from the user query and providing specific instructions to the LLM. This ensures that the LLM has all the necessary information to deliver an optimal response.

Approach:

* The system retrieves relevant context based on the user's query.
* The context is combined with the strategy design to form the final prompt.
* Specific instructions are provided within the prompt to guide the LLM, such as:

"Use a formal tone."

"Provide a brief summary."

"Include relevant examples."

Objective:

To generate a fully-optimized prompt that ensures the LLM produces a response that meets the user's expectations in terms of accuracy, relevance, and clarity.

**REST API**

**Detailed Documentation for the Flask APIs**

The provided Flask application serves two primary purposes:

Triggering **a Data bricks job to process a chatbot query** (/chatbot).

Fetching **a prompt template from memory** (/api/prompts).

**Setup and Configuration**

**Flask** is used as the web server framework.

**Flask-CORS** is enabled for Cross-Origin Resource Sharing, allowing requests from different origins.

The app communicates with a **Databricks Job API** to execute a notebook job.

**Authentication** is handled via a **Databricks token**.

**Constants**

**DATABRICKS\_HOST:** The base URL of the Databricks workspace.

**DATABRICKS\_TOKEN:** The token used to authenticate API requests to Databricks.

**DATABRICKS\_JOB\_ID:** The ID of the Databricks job that will be triggered for each user query.

**Prompt template:** A dictionary to store the generated prompt templates for each user query.

**API 1: Chatbot Query Endpoint**

**Route: /chatbot**

* **Method**: POST
* **Description**: This endpoint accepts a user query, triggers a Databricks job to process the query, waits for job completion, and returns the chatbot's response.

**Request Format:**

* **Content-Type**: application/json
* **Request Body**:

{

"user\_query": "Your question here"

}

**Flow**:

The **user query** is extracted from the request body.

The query is then sent as a parameter to a Databricks job using the trigger\_databricks\_job() function.

The Data bricks job is monitored for completion using the check\_job\_status() function.

Once the job is completed, the output is fetched using the get\_task\_run\_output() function.

 The chatbot's response is sent back to the client.

**Response:**

* **On Success (200)**:

{

"reply": "Chatbot's response here"

}

{ "error": "Error message" }

**Functions Used**:

 **trigger\_databricks\_job(user\_query)**: Sends a POST request to Databricks to trigger the job. Returns the job's run ID.

 **check\_job\_status(run\_id)**: Monitors the job's state, checking every 5 seconds until it completes. Raises an error if the job fails.

 **get\_task\_run\_output(run\_id)**: Fetches the final output from the Databricks job after successful completion.

API 2: **Fetch Prompt Template**

**Route: /api/prompts**

* **Method**: GET
* **Description**: This endpoint retrieves the prompt template that was generated during the chatbot query processing and stored in prompt\_template.

**Request Parameters:**

* **Query Parameter**: query - The original user query to retrieve its associated prompt.

/api/prompts?query=Your+question+here

**Flow:**

1. The **user query** is extracted from the request URL's query parameters.
2. The prompt corresponding to the query is retrieved from the prompt\_template dictionary.
3. The prompt is returned to the client as JSON.

### **Detailed Documentation for the Flask APIs**

The provided Flask application serves two primary purposes:

1. **Triggering a Databricks job to process a chatbot query** (/chatbot).
2. **Fetching a prompt template from memory** (/api/prompts).

### Setup and Configuration

* **Flask** is used as the web server framework.
* **Flask-CORS** is enabled for Cross-Origin Resource Sharing, allowing requests from different origins.
* The app communicates with a **Databricks Job API** to execute a notebook job.
* **Authentication** is handled via a **Databricks token**.

### Constants

* DATABRICKS\_HOST: The base URL of the Databricks workspace.
* DATABRICKS\_TOKEN: The token used to authenticate API requests to Databricks.
* DATABRICKS\_JOB\_ID: The ID of the Databricks job that will be triggered for each user query.
* prompt\_template: A dictionary to store the generated prompt templates for each user query.

### API 1: **Chatbot Query Endpoint**

#### **Route**: /chatbot

* **Method**: POST
* **Description**: This endpoint accepts a user query, triggers a Databricks job to process the query, waits for job completion, and returns the chatbot's response.

#### **Request Format**:

* **Content-Type**: application/json
* **Request Body**:

json

Copy code

{

"user\_query": "Your question here"

}

#### **Flow**:

1. The **user query** is extracted from the request body.
2. The query is then sent as a parameter to a Databricks job using the trigger\_databricks\_job() function.
3. The Databricks job is monitored for completion using the check\_job\_status() function.
4. Once the job is completed, the output is fetched using the get\_task\_run\_output() function.
5. The output is split based on the delimiter "rgukt\_basar", with the part before the delimiter being returned as the chatbot's response, and the part after stored in prompt\_template for later use.
6. The chatbot's response is sent back to the client.

#### **Response**:

* **On Success (200)**:

json

Copy code

{

"reply": "Chatbot's response here"

}

* **On Error (500)**:

json

Copy code

{

"error": "Error message"

}

#### **Functions Used**:

* **trigger\_databricks\_job(user\_query)**: Sends a POST request to Databricks to trigger the job. Returns the job's run ID.
* **check\_job\_status(run\_id)**: Monitors the job's state, checking every 5 seconds until it completes. Raises an error if the job fails.
* **get\_task\_run\_output(run\_id)**: Fetches the final output from the Databricks job after successful completion.

### API 2: **Fetch Prompt Template**

#### **Route**: /api/prompts

* **Method**: GET
* **Description**: This endpoint retrieves the prompt template that was generated during the chatbot query processing and stored in prompt\_template.

#### **Request Parameters**:

* **Query Parameter**: query - The original user query to retrieve its associated prompt.

bash

Copy code

/api/prompts?query=Your+question+here

#### **Flow**:

1. The **user query** is extracted from the request URL's query parameters.
2. The prompt corresponding to the query is retrieved from the prompt\_template dictionary.
3. The prompt is returned to the client as JSON.

#### **Response**:

* **On Success (200)**:

{

"prompt": "Generated prompt here"

}

**On Error (500)**:

{ "error": "Error message" }

**Error Handling**

 Both APIs include a generic error handler that returns a 500 status code and an error message if any exceptions occur during the execution of the API.

 If a **Databricks job** fails at any point, an exception is raised, and an appropriate error message is returned.

### Sample Use Cases:

#### **Use Case 1: Chatbot Query**

* **Request**:

POST /chatbot

{

"user\_query": "What is the weather like today?"

}

**Response**:

json

Copy code

{

"reply": "The weather today is sunny with a chance of rain."

}

#### **Use Case 2: Fetch Prompt Template**

* **Request**:

vbnet

Copy code

GET /api/prompts?query=What+is+the+weather+like+today?

* **Response**:

json

Copy code

{

"prompt": "template information after chatbot processing"

}

### Summary of the Application Flow:

1. User sends a chatbot query to /chatbot.
2. The query triggers a Data bricks notebook job that processes the query.
3. The response from the notebook is split into a chatbot reply and a prompt template.
4. The chatbot reply is returned to the user, while the prompt template is stored.
5. The user can then retrieve the prompt template for the same query using the /api/prompts endpoint.

**TECH STACK:**

1. PYTHON
2. LLM APIS
3. ANGULAR
4. FLASK
5. POSTMAN
6. DATA BRICKS