MySu ChatBot Architecture, V1

Architecture Overview:

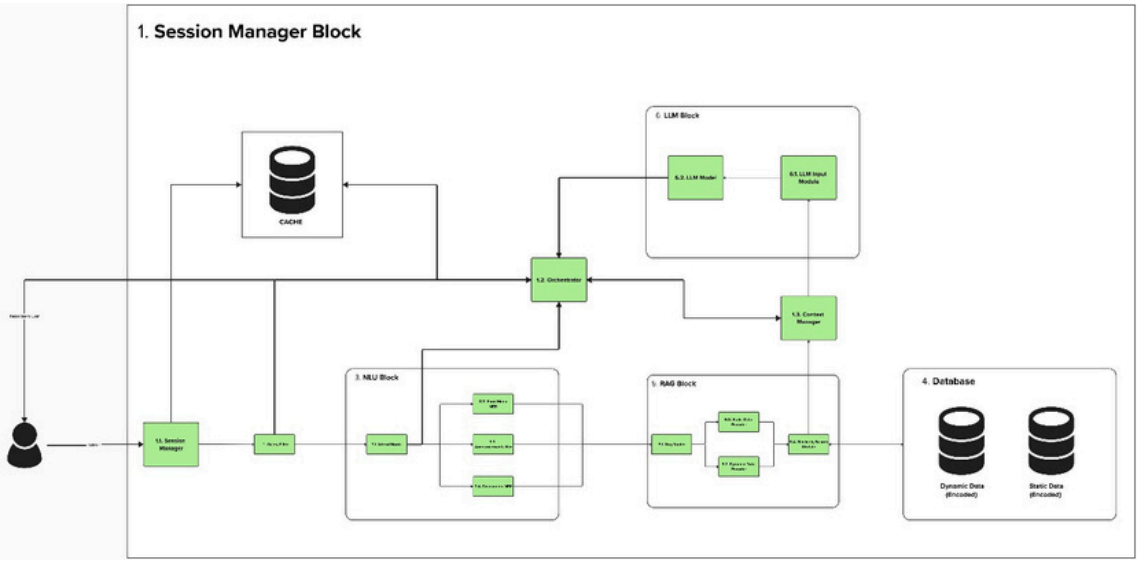

Figure 1, MySu ChatBot Architecture.

The diagram1 above illustrates the architecture of the MySu ChatBot. It highlights several components, each responsible for specific tasks. The architecture adopts a modular approach by dividing these responsibilities, and each module along with its functionalities will be explained in detail in this document.

Note:

This architecture and the modules described in this document are subject to future modifications. For now, the document serves as an overview to explain the ChatBot's overall logic, its capabilities, and the mechanisms it employs to handle specific scenarios.

Fevzi Kagan Becel, Author

[1] Chatbot Diagram, Mural

1. Session Manager:

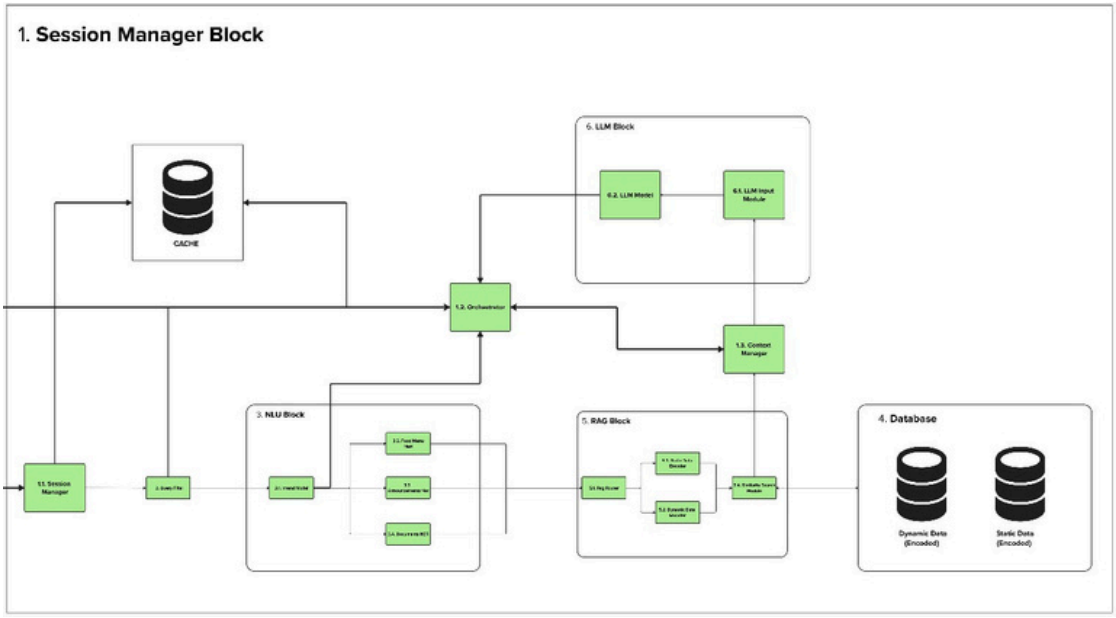

Figure 2, Session Manager Architecture

The Session Manager is the cornerstone of the chatbot architecture, responsible for enabling dynamic, context-aware interactions. Unlike a one-shot question-answering system, the Session Manager facilitates multi-turn conversations by maintaining context over a user session. It achieves this by creating and managing user sessions, ensuring seamless communication throughout the interaction.

When a user sends a query via the /message endpoint, the Session Manager generates or extends a session. Sessions are identified using user credentials or IP addresses and have a default expiration time of 30 minutes. Within each session, the user's credentials and interaction history are stored, allowing the chatbot to deliver personalized, contextually relevant responses.

The Session Manager Block consists of four distinct modules:

1.1. Session Manager Module:

This module is the backbone of the Session Manager Block, tasked with creating and managing user sessions.

- When a user sends a query, the module checks for an existing session. If none exists, it initializes a new session and sets its expiration time to 30 minutes. For active sessions, it refreshes the expiration timer upon each new query.

- All session data, including user credentials and interaction history, is stored in the Cache. This ensures quick retrieval during active sessions and efficient cleanup after expiration.
- The session data is typically stored in a lightweight, structured format (e.g., JSON) to facilitate easy access and modification.

1.2. Orchestrator Module:

The Orchestrator Module serves as the control center for the chatbot's logic and data flow.

- It governs the routing of user messages through various pathways, such as the NLU, RAG, or LLM blocks, based on the requirements of the query.
- It handles the storage of user queries and corresponding responses in the Cache for future reference, ensuring that past interactions can be utilized for context.
- Additionally, it acts as the intermediary for delivering LLM-generated responses back to the user, ensuring the chatbot operates smoothly and efficiently.

1.3. Context Manager Module:

The Context Manager Module ensures the chatbot maintains contextual awareness during multi-turn conversations.

- Before sending a user query to the LLM, the Context Manager retrieves relevant past interactions from the Cache. These interactions are formatted into a structured prompt that provides the LLM with the necessary context to generate a coherent and relevant response.
- The module also validates the output of other blocks, such as the RAG module. For instance, if the RAG retrieves an outdated document, the Context Manager flags this and incorporates it into the LLM prompt, allowing the LLM to inform the user appropriately.
- This functionality is critical for ensuring the chatbot avoids redundant or irrelevant responses and remains focused on delivering accurate and meaningful interactions.

1.4. Cache:

The Cache Module provides temporary storage for session data, including user credentials, interaction history, and context flags.

- By storing this information, the Cache ensures that the chatbot maintains context-awareness without overburdening system resources.
- A tool like Redis2, a widely-used, high-performance caching solution, can be utilized to implement this module. Redis is well-suited for managing session data in real-time applications due to its scalability and speed.
- Data is stored only for the duration of the session (e.g., 30 minutes) and is automatically purged upon expiration, keeping the system efficient and secure.

2. Query Filter:

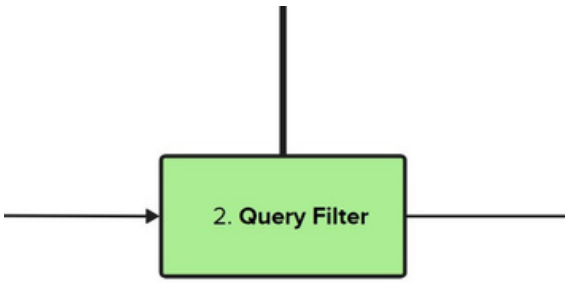

Figure 3, Query Filter Module

The Query Filter is a straightforward yet essential block, ensuring the chatbot remains focused on school-related topics.

- It evaluates user queries to determine whether they pertain to the chatbot's scope. Queries unrelated to school are flagged and routed through the Orchestrator and Context Manager modules to generate LLM responses indicating the chatbot's limitations.
- Greeting and follow-up messages (e.g., "Thank you," "How are you?") are similarly flagged but directed to the LLM with instructions to provide conversational, human-like responses.

[2] https://redis.io/

- School-related queries are forwarded to the NLU Block for further processing.

The Query Filter employs predefined keyword matching and semantic similarity analysis using tools like *Sentence-BERT* 3 or *spaCy*4. This ensures efficient and accurate filtering while maintaining flexibility for diverse user inputs.

3. NLU Block:

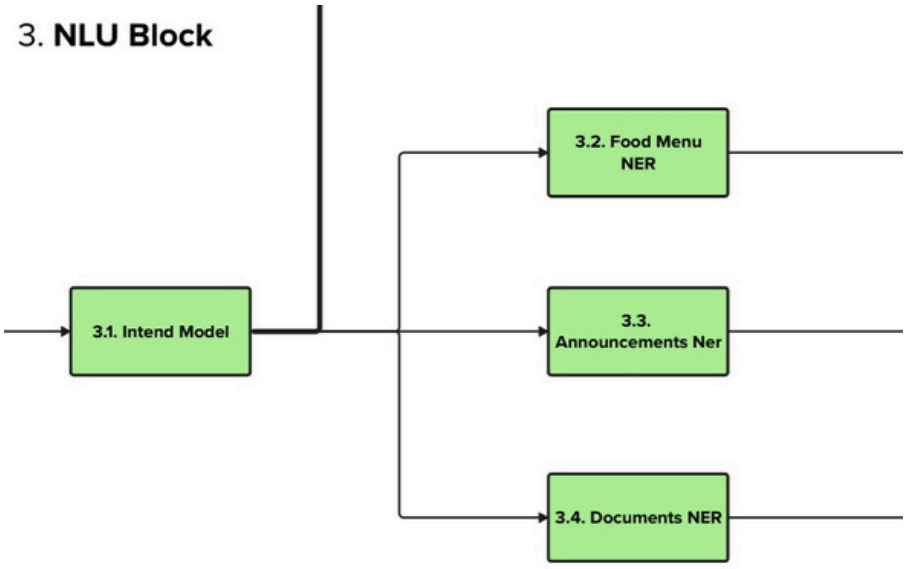

Figure 4, NLU Block

The NLU Block allows the chatbot to comprehend the semantics of user messages, making it capable of delivering contextually accurate responses. It is composed of two layers: the Intent Layer and the NER Layer.

3.1. Intent Model:

As the sole component of the Intent Layer, the intent model identifies the user's objective by determining the query's topic: Documents, Food Menu, or Announcements. Once the intent is recognized, the message is routed to the corresponding NER module for entity extraction.

The intent recognition model employs a fine-tuned BERT 5model - or a similar transformer-

[3] https://huggingface.co/efederici/sentence-bert-base
[4] https://spacy.io/
https://huggingface.co/docs/transformers/model_doc/bert
[5]

based architecture - to classify the intent with high accuracy. It uses a multi-class classification approach, trained on labeled queries related to the chatbot's supported topics.

3.2. NER Layer:

The NER Layer comprises three distinct modules: Food Menu NER (3.2.), Announcements NER (3.3.), and Documents NER (3.3.). These modules extract relevant entities to facilitate effective similarity searches in the RAG block. For instance, when a user asks for the "Final Exam Calendar," the NER extracts entities like "Final Exam."

Each NER module employs semantic similarity-based entity recognition using Sentence-BERT or a fine-tuned transformer. Instead of relying solely on predefined entity labels, this approach dynamically identifies entities by comparing query terms against a dynamically updated vocabulary or embeddings. This ensures adaptability to evolving queries and domain-specific terms.

4. Database:

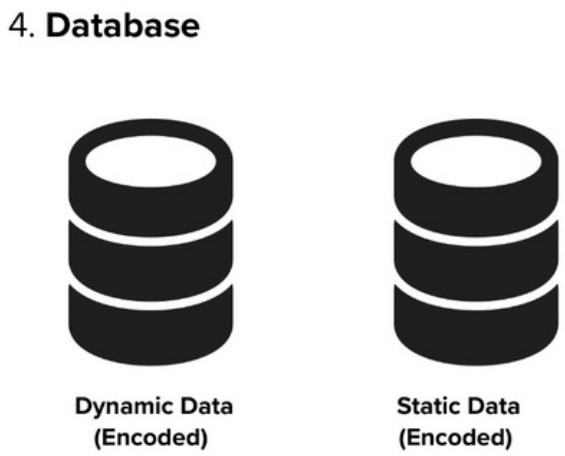

Figure 5, Database

The chatbot database is designed to support the Retrieval-Augmented Generation (RAG) block. It consists of two data types: Dynamic Data and Static Data, stored separately to accommodate their differing characteristics and use cases. Both types are indexed and vectorized to enable efficient similarity searches during RAG operations.

4.1. Dynamic Data:

Dynamic Data consists of information that is frequently updated, such as the daily food menu and recent announcements. These updates ensure the chatbot provides users with the most current information.

- Storage Method: Dynamic data is converted into embeddings using a sentence embedding model like *Sentence-BERT.* These embeddings are stored in a vector database (e.g., *Pinecone[6] or FAISS[7]*) for fast similarity searches. Update Mechanism:
- Data is automatically updated through APIs or manual uploads at scheduled intervals. Newly added data is indexed, while obsolete data is removed to maintain relevance.

4.2. Static Data:

Static Data includes information that rarely changes, such as school policies, course catalogs, or historical documents.

- Storage Method: Similar to dynamic data, static data is vectorized and stored in the same vector database but under separate namespaces or collections. This ensures logical separation and allows the RAG block to query specific data types as needed.
- Usage in RAG: Static data embeddings are queried when user questions fall under document-related intents. The chatbot retrieves the most relevant entries using cosine similarity scores.

By storing both dynamic and static data in a unified vectorized format, the RAG block can seamlessly handle diverse queries while ensuring data relevance and efficiency.

[6] https://www.pinecone.io/
[7] https://github.com/facebookresearch/faiss

5. RAG Block:

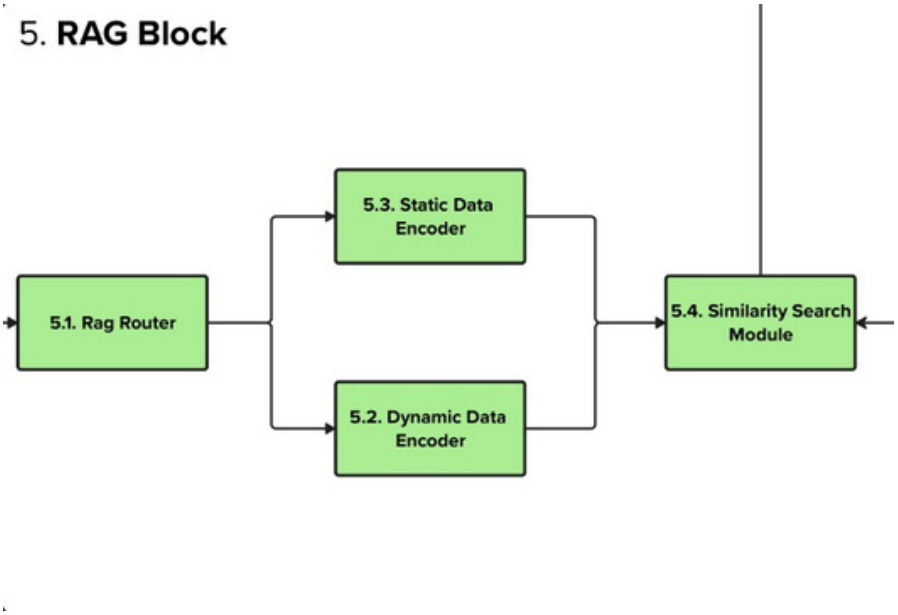

Figure 6, RAG Block

The RAG Block (Retrieval-Augmented Generation) is responsible for retrieving the most relevant data and augmenting the response generation process. It bridges the gap between raw data stored in the database (both dynamic and static) and the Language Model (LLM), allowing for context-aware and accurate answers to user queries.
It consists of four modules that manage the process from routing to similarity search:

5.1. RAG Router:

The RAG Router serves as the traffic controller of the RAG block. Upon receiving the entities and intents from the NLU block, it determines the type of query (e.g., related to dynamic data or static data) and routes the request accordingly.

- Function: The router ensures that the appropriate encoder is activated based on the user's query. It will route dynamic queries (like food menu or announcements) to the Dynamic Data Encoder and static queries (like document-related questions) to the Static Data Encoder.

5.2. Static Data Encoder:

The Static Data Encoder is responsible for encoding the static dataset into embeddings, making it ready for similarity search. Static data includes rarely updated content, such as policy documents, course catalogs, and other relevant archives.

- Function: The encoder takes the static content and transforms it into dense vector representations, typically using models like Sentence-BERT or similar. These embeddings are stored in a vector database for efficient retrieval. When queried, the encoder pulls the most relevant embeddings based on the semantic similarity of the query.

5.3. Dynamic Data Encoder:

The Dynamic Data Encoder handles the encoding of data that is frequently updated, such as daily food menus and recent announcements. These datasets require regular refreshment to stay up-to-date with the latest information.

- Function: Similar to the static data encoder, the dynamic data encoder converts content into vector embeddings. However, it also requires a mechanism to update the embeddings regularly, ensuring that new content (e.g., new food menu items) is accurately represented. These dynamic embeddings are also stored in a vector database for fast retrieval when needed.

5.4. Similarity Search Module:

The Similarity Search Module is responsible for finding the most relevant data from the vector database based on the embeddings generated by the encoders. Once the query is routed and the appropriate encoding is performed, this module compares the query's embedding with those stored in the respective database (dynamic or static).

- Function: It performs a similarity search using cosine similarity or another distance metric to match the query with the closest entries. The result of this search is then returned to the Context Manager for further processing and ultimately fed to the LLM for generating the final response. The module ensures that only the most relevant, contextually accurate data is retrieved based on the query.

6. LLM module:

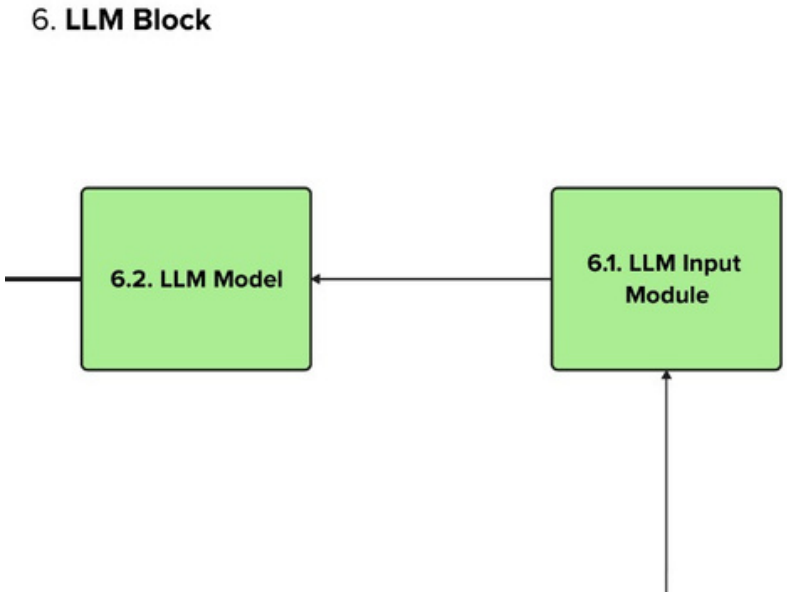

Figure 7, LLM Block

This section is crucial for making the ChatBot communicate in a natural, human-like manner. It consists of two interconnected components:

6.3. LLM Input Module:

This module prepares data for the LLM by refining the input received from the Context Manager. While the Context Manager focuses on ensuring the integrity of the data and incorporating the user's previous messages, the LLM Input Module optimizes the data formatting for the specific requirements of the LLM. This process ensures that the input is structured, clear, and ready for the model to process efficiently, enhancing the quality of the responses.

6.4. LLM Model:

The LLM Model is the centerpiece of the ChatBot's conversational capabilities. This module utilizes advanced language models, such as *LLaMa8* or *GPT-2*[9], to interpret user inputs and

[8] https://www.llama.com/
[9] https://huggingface.co/openai-community/gpt2

generate contextually relevant and coherent responses. The choice of model depends on factors such as computational resources, response time, and the desired complexity of the interactions. *LLaMa*, for example, may be used for lightweight applications, while *GPT*-2 or more advanced models could be employed for richer, more dynamic conversations. By combining context awareness and powerful language processing, this module ensures that the ChatBot provides meaningful and engaging interactions.