Task Title: Build a Conversational AI ROS System with Dockerization (C++ Focus)

Objective:

Develop a ROS-based system with three nodes, implementing all functionality in C++, and containerize the application using Docker for seamless deployment. The system will function as an end-to-end chatbot system, enabling users to input speech and receive synthesized voice responses.

This includes:

1. Converting speech input into text using an ASR (Automatic Speech Recognition) node.
2. Processing the transcribed text through a decision-making node that interacts with multiple chatbot APIs.
3. Generating and playing the response as speech using a TTS (Text-to-Speech) node.

Note: You are not required to use the suggested gTTS or ASR (STT) engines; they are provided as recommendations. Also, you do not have to purchase a ChatGPT API.

Timeline: 3 days maximum from the time the email is sent (Saturday 12/14 at 1pm).

Requirements:

Programming Language:

- All nodes must be implemented in C++ to evaluate your proficiency in the language.

ROS Framework:

- Use either ROS1 or ROS2 (your choice) with strict adherence to ROS best practices in C++.

Docker Support:

- Containerize the entire application using Docker and Docker Compose.

Task Description:

1. Whisper ASR Node

- **Programming:** Implement the node in C++.
- **Inputs:** Process audio input (from a .wav file or microphone).
- **Outputs:** Publish transcribed text to a ROS topic `/recognized_speech`.
- **Requirements:**
    - Integrate Whisper using a compatible library (e.g., whisper.cpp).
    - Handle various audio formats and manage error scenarios robustly.

2. Google TTS Node

- **Programming:** Implement the node in C++, integrating the Python-based gTTS API through a service or system call.
- **Inputs:** Subscribe to the `/text_to_speak` ROS topic for input text.
- **Outputs:** Play synthesized speech or save it as a .wav file.
- **Requirements:**
    - Demonstrate proficiency in interfacing C++ nodes with external tools like Python scripts.

3. Decision-Making Node

- **Programming:** Implement the node in C++.
- **Inputs:**
    - Subscribed text from `/recognized_speech`.
    - Responses from three chatbot APIs (e.g., OpenAI GPT, Hugging Face, or a local LLM).
- **Outputs:** Publish the selected response to `/final_response`.
- **Requirements:**
    - Implement mock integrations for three LLM APIs. Clearly note that no real API subscriptions or payments are expected unless otherwise you decide to do that—you should only create placeholder files simulating API calls and mock the responses.
    - Define and apply a selection algorithm to choose the best chatbot response based on criteria such as latency, sentiment, or relevance.
    - Include fallback logic for handling API failures.

4. Dockerization

1. **Dockerfile for Each Node:**
    - Each node (Whisper ASR, Google TTS, and Decision-Making) must have its own Docker image with C++ dependencies configured.
2. **Docker Compose Setup:**
    - Define services for all nodes, ensuring communication within the ROS environment.

   - Specify container names for clarity (e.g., `whisper_asr_node`, `google_tts_node`, `decision_maker_node`).
   - Include all three node containers in the `docker-compose.yml` file.
3. **Networking:**
   - Ensure proper configuration of ROS-specific networking variables (`ROS_MASTER_URI`, `ROS_IP`) in the Docker Compose setup.
4. **Build and Run Instructions:**
   - Provide a step-by-step guide for building Docker images and launching the system using Docker Compose.

Deliverables:

1. **C++ Implementation:**
   - Source code for all three nodes, written in C++ and adhering to ROS standards.
2. **Dockerfiles:**
   - Three individual Dockerfiles (one per node), optimized for size and performance.
     - **Purpose:** Each Dockerfile should encapsulate the dependencies and runtime environment for its respective node.
3. **docker-compose.yml:**
   - Orchestration for all containers, ensuring seamless communication.
   - Include explicit container names and service configurations as described.
4. **Documentation:**
   - A `README.md` file explaining:
     - How to build and run the system using Docker.
     - Design choices, challenges faced, and assumptions made.
     - The importance of this file is emphasized—it should provide detailed insight into the approach and reasoning behind the solution.
5. **Testing Instructions:**
   - Provide a test file or a set of steps to test the functionality of the system.
   - Explain how to verify the performance of each node and the overall integration.
6. **GitHub Repository:**
   - Deliver all code, Dockerfiles, and documentation via a GitHub repository.
   - Ensure the repository is well-organized with clear folder structures and appropriate use of version control.

Stretch Goals (Optional but Bonus for Creativity):

1. Add dynamic switching between LLMs at runtime.
2. Incorporate multithreading in C++ for improved performance.
3. Optimize Docker image sizes using multi-stage builds.
4. Implement advanced error-handling strategies in the C++ nodes.

Evaluation Criteria:

C++ Focused Criteria:

1. **Code Quality:**
   - Clear, modular, and well-documented C++ code.
2. **ROS Best Practices:**
   - Adherence to ROS conventions in C++ (e.g., nodelets, parameters, and callbacks).
3. **Error Handling:**
   - Robust handling of API failures, latency issues, and audio format inconsistencies.

General Criteria:

- Docker containerization quality.
- Ease of deployment using Docker Compose.
- Overall system functionality and integration.

Timeline and Additional Notes:

- Timebox the exercise to a maximum of three days from the time the email is sent.
- It is acceptable to ask clarifying questions if the requirements are unclear. Encourage communication to ensure expectations are met.