4. Multimodal Visual Voice Interaction

1. Concept Introduction

1.1 What is "Visual Understanding"?

In the `largemodel` project, **multimodal visual understanding** functionality refers to enabling robots to not only "see" a pixel matrix, but to truly "understand" the content, objects, scenes, and their relationships within images. This is like giving robots a pair of thinking eyes.

The core tool of this functionality is `seewhat`. When users issue commands like "look what's here", the system calls this tool, triggering a series of background operations, and finally feeds back the AI's analysis results of real-time scenes to users in natural language form.

1.2 Brief Implementation Principle

Its basic principle is to input two different types of information—**image (visual information)** and **text (language information)**—into a powerful multimodal large model (such as LLaVA).

1. **Image Encoding**: The model first converts the input image into digital vectors that computers can understand through a visual encoder. These vectors capture features such as color, shape, and texture of the image.
2. **Text Encoding**: At the same time, the user's question (such as "What's on the table?") is also converted into text vectors.
3. **Cross-modal Fusion**: The most critical step, the model fuses image vectors and text vectors in a special "attention layer". The model learns to "focus" on parts of the image relevant to the question here. For example, when asked about "table", the model will pay more attention to areas in the image that match "table" features.
4. **Generate Answer**: Finally, a large language model (LLM) part generates a descriptive text as an answer based on the fused information.

Simply put, it's **using text to "light up" corresponding parts of the image, then using language to describe the "lit" parts**.

2. Project Architecture

Key Code

1. Tool Layer Entry (`largemodel/utils/tools_manager.py`)

The `seewhat` function in this file defines the execution flow of this tool.

```
# From largemodel/utils/tools_manager.py

class ToolsManager:
    # ...

    def seewhat(self):
        """
        Capture camera frame and analyze environment with AI model.

        :return: Dictionary with scene description and image path, or None if
failed.
        """
        self.node.get_logger().info("Executing seewhat() tool")
        image_path = self.capture_frame()
        if image_path:
            # Use isolated context for image analysis.
            analysis_text = self._get_actual_scene_description(image_path)

            # Return structured data for the tool chain.
            return {
                "description": analysis_text,
                "image_path": image_path
            }
        else:
            # ... (Error handling)
            return None

    def _get_actual_scene_description(self, image_path, message_context=None):
        """
        Get AI-generated scene description for captured image.

        :param image_path: Path to captured image file.
        :return: Plain text description of scene.
        """
        try:
            # ... (Build Prompt)

            # Force use of a plain text system prompt with a clean, one-time
context.
            simple_context = [{
                "role": "system",
                "content": "You are an image description assistant. ..."
            }]

            result = self.node.model_client.infer_with_image(image_path,
scene_prompt, message=simple_context)
            # ... (Process result)
            return description
        except Exception as e:
            # ...
```

2. Model Interface Layer (`largemodel/utils/large_model_interface.py`)

The `infer_with_image` function in this file is the unified entry point for all image understanding tasks, responsible for calling specific model implementations based on configuration.

```python
# From largemodel/utils/large_model_interface.py

class model_interface:
    # ...
    def infer_with_image(self, image_path, text=None, message=None):
        """Unified image inference interface."""
        # ... (Prepare messages)
        try:
            # Decide which specific implementation to call based on the value of
self.llm_platform
            if self.llm_platform == 'ollama':
                response_content = self.ollama_infer(self.messages,
image_path=image_path)
            elif self.llm_platform == 'tongyi':
                # ... Logic for calling Tongyi model
                pass
            # ... (Logic for other platforms)
        # ...
        return {'response': response_content, 'messages': self.messages.copy()}
```

Code Analysis

The implementation of this functionality involves two main layers: the tool layer defines business logic, and the model interface layer is responsible for communicating with large language models. This layered design is key to achieving platform universality.

1. **Tool Layer (`tools_manager.py`):**
    - The `seewhat` function is the business core of the visual understanding functionality. It encapsulates the complete process of the "seeing" action: first calling the `capture_frame` method to get the image, then calling `_get_actual_scene_description` to prepare a command (Prompt) for requesting the model to analyze the image.
    - The most critical step is that it calls the `infer_with_image` method of the model interface layer. It doesn't care which model is used at the bottom, only responsible for passing the two core data of "image" and "analysis command".
    - Finally, it packages the analysis result (plain text description) received from the model interface layer into a structured dictionary and returns it. This makes it convenient for upper-level applications to use the analysis results.
2. **Model Interface Layer (`large_model_interface.py`):**
    - The `infer_with_image` function plays the role of a "dispatch center". Its main responsibility is to check the current platform configuration (`self.llm_platform`) and distribute tasks to corresponding specific processing functions (such as `ollama_infer` or `tongyi_infer`) based on the configuration value.
    - This layer is key to adapting to different AI platforms. All platform-specific operations (such as data encoding, API call formats, etc.) are encapsulated in their respective processing functions.

- Through this approach, the business logic code in `tools_manager.py` can support multiple different backend large model services without any changes. It only needs to interact with the unified, stable interface `infer_with_image`.

In summary, the execution flow of the `seewhat` tool demonstrates a clear responsibility separation pattern: `ToolsManager` is responsible for defining "what to do" (get image and request analysis), while `model_interface` is responsible for defining "how to do it" (select appropriate model platform based on current configuration and complete interaction with it). This makes the tutorial analysis universal, regardless of whether users are in online or offline mode, the core code logic is consistent.

3. Practical Operations

3.1 Configure Online LLM

1. **First obtain API Key from OpenRouter platform**
2. **Then update the key in the configuration file, open the model interface configuration file** `large_model_interface.yaml`:

   ```
   vim ~/yahboom_ws/src/largemodel/config/large_model_interface.yaml
   ```

3. **Enter your API Key**:
   Find the corresponding section and paste the API Key you just copied. Here we use Tongyi Qianwen configuration as an example

   ```
   # large_model_interface.yaml

   # OpenRouter平台配置 (OpenRouter Platform Configuration)
   openrouter_api_key: "sk-xxxxxxxxxxxxxxxxxxxxxxxxxxxx"
   openrouter_model: "nvidia/nemotron-nano-12b-v2-vl:free" # Model to use,
   e.g., "google/gemini-pro-vision"
   ```

4. **Open the main configuration file** `yahboom.yaml`:

   ```
   vim ~/yahboom_ws/src/largemodel/config/yahboom.yaml
   ```

5. **Select the online platform to use**:
   Modify the `llm_platform` parameter to the platform name you want to use

   ```
   # yahboom.yaml

   model_service:
     ros__parameters:
       # ...
       llm_platform: 'openrouter'              # Currently selected large model
   platform
       # Available platforms: 'ollama','openrouter'
   ```

Recompile

```
cd ~/yahboom_ws/
colcon build
source install/setup.bash
```

3.2 Start and Test Function

1. **Start the `largemodel` main program**:
   Open a terminal, then run the following command:

   ```
   ros2 launch largemodel largemodel_control.launch.py
   ```

2. **Testing**:
   - **Wake up**: Speak to the microphone: "Hello, Yahboom."
   - **Dialogue**: After the speaker responds, you can say: `What do you see?`
   - **Observe logs**: In the terminal running the `launch` file, you should see:
     1. The ASR node recognizes your question and prints it out.
     2. The `model_service` node receives the text, calls the LLM, and prints out the LLM's response.
   - **Listen to answer**: Shortly after, you should hear the answer from the speakers.