Multimodal Table Scanning Applications

1. Concept Introduction

1.1 What is "Multimodal Table Scanning"?

Multimodal table scanning is a technology that uses image processing and artificial intelligence to identify and extract table information from images or PDF documents. It not only focuses on visual table structure recognition but also combines multimodal data such as text content and layout information to enhance table understanding. **Large Language Models (LLMs)** provide powerful semantic analysis capabilities for understanding this extracted information; the two complement each other to improve the intelligence level of document processing.

1.2 Brief Description of Implementation Principles

1. **Table Detection and Content Recognition**

- Use computer vision technology to locate tables in documents and use OCR technology to convert the text within the tables into an editable format.
- Use deep learning methods to parse the table structure (row and column division, cell merging, etc.) and generate a structured data representation.

2. **Multimodal Fusion**

- Integrates visual elements (such as table layout), text (OCR results), and any existing metadata (such as file type and source) into a comprehensive data view.
- Uses a specially designed multimodal model (such as LayoutLM) to process these different types of data simultaneously for a more accurate understanding of table content and its context.

2. Code Analysis

Key Code

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

The `scan_table` function in this file defines the execution flow of the tool, specifically how it constructs a Prompt that requests a Markdown format return.

```
# From largemodel/utils/tools_manager.py
class ToolsManager:
    # ...
    def scan_table(self, args):
        """
        Scan a table from an image and save the content as a Markdown file.

        :param args: Arguments containing the image path.
        :return: Dictionary with file path and content.
        """
        self.node.get_logger().info(f"Executing scan_table() tool with args:
{args}")
        try:
            image_path = args.get("image_path")
            # ... (Path checking and fallback)

            # Construct a prompt asking the large model to recognize the table
and return it in Markdown format.
            if self.node.language == 'zh':
                prompt = "请仔细分析这张图片，识别其中的表格，并将其内容以Markdown格式返
回。"
            else:
                prompt = "Please carefully analyze this image, identify the
table within it, and return its content in Markdown format."

            result = self.node.model_client.infer_with_image(image_path, prompt)

            # ... (Extract Markdown text from the results)

            # Save the recognized content to a Markdown file.
            md_file_path = os.path.join(self.node.pkg_path, "resources_file",
"scanned_tables", f"table_{timestamp}.md")
            with open(md_file_path, 'w', encoding='utf-8') as f:
                f.write(table_content)

            return {
                "file_path": md_file_path,
                "table_content": table_content
            }
        # ... (Error Handling)
```

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-related tasks.

```
# 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 message)
        try:
            # Determine 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 the Tongyi model
                pass
            # ... (Logic for other platforms)
        # ...
        return {'response': response_content, 'messages': self.messages.copy()}
```

Code Analysis

The table scanning function is a typical application of converting unstructured image data into structured text data. Its core technology remains **guiding model behavior through Prompt Engineering**.

1. **Tool Layer (`tools_manager.py`):**

- The `scan_table` function is the business process controller for this function. It receives an image containing a table as input.
- The most crucial operation of this function is **constructing a targeted Prompt**. This Prompt directly instructs the large model to perform two tasks: 1. Recognize the table in the image. 2. Return the recognized content in Markdown format. This requirement for the output format is key to achieving the unstructured to structured conversion.
- After constructing the Prompt, it calls the `infer_with_image` method of the model interface layer, passing the image and this formatting instruction along with it.
- After receiving the returned Markdown text from the model interface layer, it performs a file operation: writing the text content into a new `.md` file.

Finally, it returns structured data containing the new file paths and table contents.

2. **Model Interface Layer (`large_model_interface.py`):**

The `infer_with_image` function continues to act as a unified "dispatch center." It receives the image and prompt from `scan_table` and dispatches the task to the correct backend model implementation based on the current system configuration (`self.llm_platform`).

Regardless of the backend model, this layer's task is to handle the communication details with the specific platform, ensuring that image and text data are sent correctly, and then return the plain text (in this case, Markdown format) returned by the model to the utility layer.

In summary, the general workflow for table scanning is as follows: `ToolsManager` receives the image and constructs a command to "convert the table in this image to Markdown" -> `ToolsManager` calls the model interface -> `model_interface` packages the image and the command, and sends it to the corresponding model platform according to the configuration -> The model returns Markdown formatted text -> `model_interface` returns the text to `ToolsManager` -> `ToolsManager` saves the text as a `.md` file and returns the result. This workflow demonstrates how to leverage the formatting capabilities of large models as a powerful OCR (Optical Character Recognition) and data structuring tool.

3. Practical Operation

3.1 Configuring Online LLM

Raspberry Pi 5 requires entering a Docker container; RDK X5 and Orin mainframes do not require this:

```
./ros2_docker.sh
```

If you need to enter other commands within the same Docker container later, simply type `./ros2_docker.sh` again in the host machine's terminal.

1. **Then you need to 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
   ```

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

   ```
   # large_model_interface.yaml

   ## Thousand Questions on Tongyi
   qianwen_api_key: "sk-xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx" # Paste your Key
   qianwen_model: "qwen-vl-max-latest" # You can choose the model as needed,
   such as qwen-turbo, qwen-plus
   ```

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

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

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

   ```
   # yahboom.yaml

   model_service:
     ros__parameters:
       # ...
       llm_platform: 'tongyi'  #Optional platforms: 'ollama', 'tongyi',
   'spark', 'qianfan', 'openrouter'
   ```

   After modifying the configuration file, you need to recompile and source it in the workspace:

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

3.2 Starting and Testing the Functionality

1. **Preparing the Image File**:

Place the image file you want to test in the following path:
`~/yahboom_ws/src/largemodel/resources_file/scan_table`

Then name the image `test_table.jpg`

2. **Starting the `largemodel` Main Program**:

Open a terminal, enter the Docker container, and then run the following command:

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

3. **Testing**:

- **Wake-up**: Say into the microphone, "Hello, Xiaoya."
- **Dialogue**: After the speaker responds, you can say: `Analyze the table`
- **Observing the Logs**: In the terminal running the `launch` file, you should see:

1. The ASR node recognizes your problem and prints it out.
2. The `model_service` node receives the text, calls LLM, and prints LLM's response.

- **Listen to the answer**: Shortly after, you should be able to hear the answer through the speaker and find a md file containing the table information in the path `~/yahboom_ws/src/largemodel/resources_file/scan_table`.