

# 6. Multimodal Table Scanning Application

---

## 6. Multimodal Table Scanning Application

### 1. Concept Introduction

#### 1.1 What is "Multimodal Table Scanning"?

#### 1.2 Brief Implementation Principle

### 2. Code Analysis

#### Key Code

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

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

#### Code Analysis

### 3. Practical Operations

#### 3.1 Configure Online LLM

#### 3.2 Start and Test Function

## 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 these extracted information, complementing each other to collectively improve the intelligence level of document processing.

### 1.2 Brief Implementation Principle

#### 1. Table Detection and Content Recognition

- Use computer vision technology to locate tables in documents and convert text within tables into editable format through OCR technology.
- Use deep learning methods to parse table structure (row/column division, merged cells, etc.) and generate structured data representation.

#### 2. Multimodal Fusion

- Integrate visual (such as table layout), text (OCR results), and possibly existing metadata (such as file type, source) to form a comprehensive data view.
- Use specially designed multimodal models (such as LayoutLM) to simultaneously process these different types of data for more accurate understanding of table content and its contextual relationships.

## 2. Code Analysis

---

## Key Code

### 1. Tool Layer Entry (`largemode1/utils/tools_manager.py`)

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

```
# From largemode1/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 = "Please carefully analyze this image, identify the
                table within it, and return its content in Markdown format."
            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 result)

            # 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

#### (`largemode1/utils/large_model_interface.py`)

The `infer_with_image` function in this file is the unified entry point for all image-related tasks.

```
# From largemode1/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

Table scanning functionality is a typical application that converts unstructured image data into structured text data. Its core technology is still **guiding model behavior through Prompt Engineering**.

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

- The `scan_table` function is the business process controller for this functionality. It receives an image containing a table as input.
- The most critical operation of this function is **building a goal-oriented Prompt**. This Prompt directly instructs the large model to perform two tasks: 1. Identify the table in the image. 2. Return the identified content in Markdown format. This mandatory requirement for output format is key to achieving unstructured to structured conversion.
- After building the Prompt, it calls the `infer_with_image` method of the model interface layer, passing both the image and this formatting instruction.
- After receiving the returned Markdown text from the model interface layer, it executes a file operation: writing this text content into a new `.md` file.
- Finally, it returns structured data containing the new file path and table content.

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

- The `infer_with_image` function continues to serve as a unified "dispatch center". It receives images and Prompts from `scan_table`, and dispatches tasks to the correct backend model implementation based on current system configuration (`self.llm_platform`).
- Regardless of the backend model, the task of this layer is to handle communication details with specific platforms, ensuring image and text data are correctly sent, then returning the plain text returned by the model (here, Markdown formatted text) to the tool layer.

In summary, the general flow of table scanning is: `ToolsManager` receives an image and builds a "convert the table in this image to Markdown" instruction -> `ToolsManager` calls model interface -> `model_interface` packages the image and this instruction, and sends it to the corresponding model platform based on configuration -> Model returns Markdown formatted text -> `model_interface` returns text to `ToolsManager` -> `ToolsManager` saves text as `.md` file and returns result. This process demonstrates how to use the format-following capability of large models to use them as powerful OCR (Optical Character Recognition) and data structuring tools.

## 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/largemode1/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-v1:free" # Model to use,
e.g., "google/gemini-pro-vision"
```

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

```
vim ~/yahboom_ws/src/largemode1/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. Prepare image file:

Place a test image file in the following path:

```
/home/sunrise/yahboom_ws/src/largemode1/resources_file/scan_table
```

Then name the image `test_table.jpg`

#### 2. Start the `largemode1` main program:

Open a terminal, then run the following command:

```
ros2 launch largemode1 largemode1_control.launch.py
```

### 3. Testing:

- **Wake up:** Speak to the microphone: "Hello, Yahboom."
- **Dialogue:** After the speaker responds, you can say: `Analyze the table`
- **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, and find an MD document saving table information in the path  
`/home/sunrise/yahboom_ws/src/largemode1/resources_file/scan_table`.