5. Multimodal Autonomous Agent Application

1. Concept Introduction

1.1 What is "Autonomous Agent"?

In the `largemodel` project, **multimodal autonomous agent** is the highest form of intelligence. It no longer simply responds to a user's instruction with a single response, but is capable of **autonomously thinking, planning, and continuously calling multiple tools to complete tasks to achieve a complex goal**.

The core of this feature is the `agent_call` tool or its underlying **tool chain manager (`ToolChainManager`)**. When users make a complex request that cannot be completed through a single tool call, the autonomous agent will be activated.

1.2 Brief Implementation Principle

The autonomous agent implementation in `largemodel` follows the industry mainstream **ReAct (Reason + Act)** paradigm. Its core idea is to mimic the human problem-solving process, cycling between "thinking" and "acting".

1. **Reason**: When the agent receives a complex goal, it first calls a powerful language model (LLM) to "think". It asks itself: "To complete this goal, what should be my first step? Which tool should I use?" The LLM's output is not the final answer, but an action plan.
2. **Act**: The agent executes the corresponding action based on the LLM's thinking result—calling `ToolsManager` to run the specified tool (such as `visual_positioning`).
3. **Observe**: The agent obtains the result of the previous action ("observation"), for example `{"result": "Found the cup, location at [120, 300, 180, 360]"}`.
4. **Think Again**: The agent submits this observation result, along with the original goal, to the LLM again for a second round of "thinking". It asks itself: "I have found the cup's location, what should I do next to know its color?" The LLM might generate a new action plan, for example `{"thought": "I need to analyze the image of the cup's area to determine color", "action": "seewhat", "args": {"crop_area": [120, 300, 180, 360]}}`.

This **Think -> Act -> Observe** cycle continues until the original goal is achieved, at which point the agent will generate and output the final answer.

2. Code Analysis

Key Code

1. Agent Core Workflow (`largemodel/utils/ai_agent.py`)

The `_execute_agent_workflow` function is the main execution loop of the Agent, defining the core process of "Plan -> Execute".

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

class AIAgent:
    # ...

    def _execute_agent_workflow(self, task_description: str) -> Dict[str, Any]:
        """
        Executes the agent workflow: Plan -> Execute.
        """
        try:
            # Step 1: Task planning
            self.node.get_logger().info("AI Agent starting task planning phase")
            plan_result = self._plan_task(task_description)

            # ... (Return early if planning fails)

            self.task_steps = plan_result["steps"]

            # Step 2: Execute all steps in sequence
            execution_results = []
            tool_outputs = []

            for i, step in enumerate(self.task_steps):
                # 2.1. Before execution, process data references in parameters
                processed_parameters =
self._process_step_parameters(step.get("parameters", {}), tool_outputs)
                step["parameters"] = processed_parameters

                # 2.2. Execute single step
                step_result = self._execute_step(step, tool_outputs)
                execution_results.append(step_result)

                # 2.3. If step succeeds, save its output for subsequent step
reference
                if step_result.get("success") and
step_result.get("tool_output"):
                    tool_outputs.append(step_result["tool_output"])
                else:
                    # If any step fails, abort the entire task
                    return { "success": False, "message": f"Task terminated
because step '{step['description']}' failed." }

            # ... Summarize and return final result
            summary = self._summarize_execution(task_description,
execution_results)
```

```
            return { "success": True, "message": summary, "results":
execution_results }

        # ... (Exception handling)
```

2. Task Planning and LLM Interaction (`largemodel/utils/ai_agent.py`)

The core of the `_plan_task` function is to build a sophisticated Prompt, using the large model's own reasoning capabilities to generate structured execution plans.

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

class AIAgent:
    # ...
    def _plan_task(self, task_description: str) -> Dict[str, Any]:
        """
        Uses the large model for task planning and decomposition.
        """
        # Dynamically generate available tool list and their descriptions
        tool_descriptions = []
        for name, adapter in
self.tools_manager.tool_chain_manager.tools.items():
            # ... (Get tool description from adapter.input_schema)
            tool_descriptions.append(f"- {name}({params}): {description}")
        available_tools_str = "\\n".join(tool_descriptions)

        # Build highly structured planning Prompt
        planning_prompt = f"""
As a professional task planning Agent, please break down the user task into a
series of specific, executable JSON steps.

**# Available Tools:**
{available_tools_str}

**# Core Rules:**
1.  **Data Transfer**: When subsequent steps need to use outputs from previous
steps, you **must** use the `{{{{steps.N.outputs.KEY}}}}` format for reference.
    - `N` is the step ID (starting from 1).
    - `KEY` is the specific field name in the previous step's output data.
2.  **JSON Format**: Must strictly return JSON object.

**# User Task:**
{task_description}
"""

        # Call large model for planning
        messages_to_use = [{"role": "user", "content": planning_prompt}]
        # Note: This calls the universal text inference interface
        result = self.node.model_client.infer_with_text("",
message=messages_to_use)

        # ... (Parse JSON response and return step list)
```

3. Parameter Processing and Data Flow Implementation (`largemodel/utils/ai_agent.py`)

The `_process_step_parameters` function is responsible for parsing placeholders and implementing data flow between steps.

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

class AIAgent:
    # ...
    def _process_step_parameters(self, parameters: Dict[str, Any],
previous_outputs: List[Any]) -> Dict[str, Any]:
        """
        Parses parameter dictionary, finds and replaces all {{...}} references.
        """
        processed_params = parameters.copy()
        # Regular expression used to match {{steps.N.outputs.KEY}} format
placeholders
        pattern = re.compile(r"\\{\\{steps\\.(\\d+)\\.outputs\\.(.+?)\\}\\}")

        for key, value in processed_params.items():
            if isinstance(value, str) and pattern.search(value):
                # Use re.sub and a replacement function to handle all found
placeholders
                # The replacement function will look up and return values from
the previous_outputs list
                processed_params[key] = pattern.sub(replacer_function, value)

        return processed_params
```

Code Analysis

AI Agent is the "central brain" of the system, transforming high-level and sometimes vague tasks proposed by users into a series of precise, ordered tool calls. Its implementation does not rely on any specific model platform, but is built on a universal, scalable architecture.

1. **Dynamic Task Planning**: The core capability of Agent lies in the `_plan_task` function. It does not rely on hard-coded logic, but dynamically generates task plans through interaction with large models.
    - **Self-awareness and Prompt Building**: At the beginning of planning, Agent first checks all currently available tools and their descriptions. Then, it packages this tool information, user tasks, and strict rules (such as data transfer format) into a highly structured `planning_prompt`.
    - **Model as Planner**: This Prompt is sent to a universal text large model. The model reasons based on the provided context and returns a JSON-formatted action plan containing multiple steps. This design is extremely scalable: when tools are added or modified in the system, Agent's planning capability automatically updates without modifying Agent's own code.
2. **Tool Chain and Data Flow**: Real-world tasks often require collaboration between multiple tools, for example "take photo and describe" requires using the output of "take photo" tool (image path) as input for "describe" tool. AI Agent elegantly implements this through the `_process_step_parameters` function.
    - **Data Reference Placeholders**: In the planning phase, the large model will embed special placeholders in parameter values that need to pass data, such as

`{{steps.1.outputs.data}}`.

- **Real-time Parameter Replacement**: In the main loop of `_execute_agent_workflow`, before executing each step, `_process_step_parameters` will be called. It uses regular expressions to scan all parameters of the current step, and once placeholders are found, it will find corresponding data from the output list of previous steps and perform real-time replacement. This mechanism is key to implementing complex task automation.

3. **Supervised Execution and Fault Tolerance**: `_execute_agent_workflow` constitutes the main execution loop of Agent. It strictly executes each action in the planned step order and ensures data is correctly passed between steps.
   - **Atomic Steps**: Each step is treated as an independent "atomic operation". If any step fails to execute, the entire task chain will immediately abort and report an error. This ensures system stability and predictability, avoiding continued execution in error states.

In summary, the universal implementation of AI Agent demonstrates an advanced software architecture: it does not directly solve problems, but builds a framework that lets an external, universal reasoning engine (large model) solve problems. Through the two core mechanisms of "dynamic planning" and "data flow management", Agent is able to orchestrate a series of independent tools into complex workflows that can complete advanced tasks.

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 platform configuration
openrouter_api_key: "sk-xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"
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 text_chat_mode:=true
   ```

2. **Send text commands**:
   Open another terminal and run the following command,

   ```
   ros2 run text_chat text_chat
   ```

   Then start typing text: "Generate a similar scene image based on the current environment".

3. **Observe results**:

   In the first terminal running the main program, you will see log output showing that the system received the text command, called the `aiagent` tool, then provided prompt to LLM, and LLM will analyze detailed steps for calling tools. For this question, it will call the seewhat tool to get the image, then provide the image to LLM for analysis, the parsed text will be given to LLM as new image content for image generation.