Multimodal autonomous proxy application

1. Concept Introduction

1.1 What is an "Autonomous Agent"?

In the `largemodel` project, **multimodal autonomous agents** represent the most advanced form of intelligence. Rather than simply responding to a user's command, they are capable of **autonomously thinking, planning, and continuously invoking multiple tools to achieve a complex goal**.

The core of this functionality is the **agent_call`** tool or its underlying **`ToolChainManager`. When a user issues a complex request that cannot be accomplished with a single tool call, the autonomous agent is activated.

1.2 Implementation Principles

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

1. **Reason**: When the agent receives a complex goal, it first invokes a powerful language model (LLM) to perform "thinking." It asks itself, "To achieve this goal, what should I do first? Which tool should I use?" The output of the LLM isn't a final answer, but rather an action plan.
2. **Act**: Based on the LLM's deliberation, the agent performs the corresponding action—calling the ToolsManager to run the specified tool (e.g., `visual_positioning`).
3. **Observe**: The agent obtains the result of the previous action ("observation"), for example, `{"result": "The cup was found at [120, 300, 180, 360]"}`.
4. **Rethink**: The agent submits this observation, along with the original goal, to the LLM for a second round of "reflection." It asks itself, "Now that I've found the cup's location, what should I do next to learn its color?" The LLM might generate a new action plan, such as {"thought": "I need to analyze the image of the area where the cup is located to determine its color", "action": "seewhat", "args": {"crop_area": [120, 300, 180, 360]}}`.

This cycle of **think -> act -> observe** continues until the initial goal is achieved, at which point the agent generates and outputs a 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, which defines the core process of "planning -> execution".

```
# 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. / 执行Agent工作流：规划 -> 执行。
        """
        try:
            # Step 1: Mission 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: Follow all steps in order
            execution_results = []
            tool_outputs = []

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

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

                # 2.3. If the step succeeds, save its output for reference in subsequent steps
                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 the 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, leveraging the large model's inherent reasoning capabilities to generate a structured execution plan.

```
# 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. / 使用大模型进行
任务规划和分解。
        """
        # Dynamically generate a list of available tools 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 a highly structured plan
        planning_prompt = f"""
As a professional task planning agent, please break down user tasks into a series
of specific, executable JSON steps.

**# Available Tools:**
{available_tools_str}

**# Core Rules:**
1. **Data Passing**: When a subsequent step requires the output of a previous
step, it must be referenced using the `{{{{steps.N.outputs.KEY}}}}` format.
- `N` is the step ID (starting at 1).
- `KEY` is the specific field name in the output data of the previous step.
2. **JSON Format**: Must strictly return a JSON object.

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

        # Calling large models for planning
        messages_to_use = [{"role": "user", "content": planning_prompt}]
        # Note that the general text reasoning interface is called here
        result = self.node.model_client.infer_with_text("",
message=messages_to_use)

        # ... (Parse the JSON response and return a list of steps)
```

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 placeholders in the format
{{steps.N.outputs.KEY}}
        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 process all found
placeholders
                # The replacement function looks up and returns a value from the
previous_outputs list.
                processed_params[key] = pattern.sub(replacer_function, value)

        return processed_params
```

Code Analysis

The AI Agent is the "brain" of the system, translating high-level, sometimes ambiguous, tasks posed by the user into a precise, ordered series of tool calls. Its implementation is independent of any specific model platform and built on a general, extensible architecture.

1. **Dynamic Task Planning**: The Agent's core capability lies in the `_plan_task` function. Rather than relying on hard-coded logic, it dynamically generates task plans by interacting with a larger model.

- **Self-Awareness and Prompt Construction**: At the beginning of planning, the Agent first examines all available tools and their descriptions. It then packages this tool information, the user's task, and strict rules (such as data transfer format) into a highly structured `planning_prompt`.
- **Model as Planner**: This prompt is fed into a general text-based model. The model reasoned based on the provided context and returned a multi-step action plan in JSON format. This design is highly scalable: as tools are added or modified in the system, the Agent's planning capabilities are automatically updated without requiring code modifications.

2. **Toolchain and Data Flow**: Real-world tasks often require the collaboration of multiple tools. For example, "take a picture and describe" requires the output (image path) of the "take a picture" tool to be used as the input of the "describe" tool. The AI Agent elegantly implements this through the `_process_step_parameters` function.

- **Data Reference Placeholders**: During the planning phase, large models embed special placeholders, such as `{{steps.1.outputs.data}}`, in parameter values where data needs

to be passed.

- **Real-Time Parameter Replacement**: In the `_execute_agent_workflow` main loop, `_process_step_parameters` is called before each step. It uses regular expressions to scan all parameters of the current step. Upon discovering a placeholder, it finds the corresponding data from the output list of the previous step and replaces it in real time. This mechanism is key to automating complex tasks.

3. **Supervised Execution and Fault Tolerance**: `_execute_agent_workflow` constitutes the Agent's main execution loop. It strictly follows the planned sequence of steps, executing each action sequentially and ensuring the correct flow of data between them.

- **Atomic Steps**: Each step is treated as an independent "atomic operation." If any step fails, the entire task chain immediately aborts and reports an error. This ensures system stability and predictability, preventing continued execution in an erroneous state.

In summary, the general implementation of the AI Agent demonstrates an advanced software architecture: rather than solving problems directly, it builds a framework that enables an external, general-purpose reasoning engine (a large model) to solve the problem. Through two core mechanisms, dynamic programming and data flow management, the Agent is able to orchestrate a series of independent tools into complex workflows capable of completing advanced tasks.

3. Practical Operations

3.1 Configuring Online LLM

1. **First obtain the API Key from any platform in the previous tutorial**
2. **Then you need to update the key in the configuration file and 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. This example uses the Tongyi Qianwen configuration.
4.
```
# 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 asqwen-turbo, qwen-plus
```

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

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

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

```
# yahboom.yaml

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

3.2 Start and test the function

1. **start up**

   Note: The startup commands for CSi cameras and USB microphone cameras are different. Please run the command corresponding to your camera.

   CSI Camera

   Open a terminal and start UDP video streaming (host machine)

   ```
   ./start_csi.sh
   ```

   Open a new terminal and enter the CSI camera docker (host machine)

   ```
   ./run_csi_docker.sh
   ```

   Start topic conversion (docker)

   ```
   python3 ~/temp/udp_camera_publisher.py
   ```

   Open another new terminal and check the container id

   ```
   docker ps
   ```

   According to the container ID shown above, change the container ID of the following command to the actual ID displayed, and enter the same docker with multiple terminals

   ```
   docker exec -it container_id  /bin/bash
   ```

   Start the `largemodel` main program:

   Open a terminal and run the following command:

   ```
   ros2 launch largemodel largemodel_control.launch.py text_chat_mode:=true
   ```

   Start sending text commands:
   Open another terminal and run the following command. Modify the container ID according to your needs. Multiple terminals can access the same docker run.

   ```
   docker ps
   ```

   ```
   docker exec -it  container_id /bin/bash
   ```

```
ros2 run text_chat text_chat
```

Then start typing text: "Based on the current environment, generate a picture of a similar scene."

USB Camera

Open a terminal and access the USB camera docker (host machine)

```
./run_usb_docker.sh
```

Start the `largemodel` main program:

```
ros2 launch largemodel largemodel_control.launch.py text_chat_mode:=true
```

Open another new terminal and check the container id

```
docker ps
```

According to the container ID shown above, change the container ID of the following command to the actual ID displayed, and enter the same docker with multiple terminals

```
docker exec -it  container_id  /bin/bash
```

Start sending text commands:

```
ros2 run text_chat text_chat
```

Then start typing text: "Based on the current environment, generate a picture of a similar scene."

2. **Observations**:

In the first terminal running the main program, you'll see log output showing the system receiving the text command, calling the `aiagent` tool, and then providing a prompt to LLM. LLM will analyze the detailed steps of calling the tool. For example, in this question, the seewhat tool will be called to obtain the image, which will then be parsed by LLM. The parsed text will be given to LLM as the content of a new image, which will be generated.