Multimodal Autonomous Agent Applications

1. Concept Introduction

1.1 What is an "Autonomous Agent"?

In the `largemodel` project, the **multimodal autonomous agent** is the highest level of intelligence. It no longer simply responds to a user's command once, but is able to **autonomously think, plan, and continuously invoke multiple tools to complete a task to achieve a complex goal**.

The core of this functionality is the `agent_call` tool or its underlying **toolchain manager (`ToolChainManager`). The autonomous agent is activated when a user makes a complex request that cannot be completed with a single tool call.

1.2 Implementation Principles

The autonomous agent implementation in `largemodel` follows the industry-leading **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 invokes a powerful Language Model (LLM) to "think." It asks itself, "What should my first step be to achieve this goal? Which tool should I use?" The LLM's output is not a final answer, but an action plan.
2. **Act**: Based on the LLM's reasoning, the agent executes the corresponding action—calling `ToolsManager` to run the specified tool (such as `visual_positioning`).
3. **Observe**: The agent retrieves the result of the previous action ("observation"), for example, `{"result": "The cup was found, located at [120, 300, 180, 360]"}` .
4. **Rethink**: The agent submits the observation results, along with the original goal, back to the LLM for a second round of "reasoning." It asks itself, "I've found the location of the cup. What should I do next to find out 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 **think -> act -> observe** loop continues until the initial goal is achieved, at which point the agent generates and outputs the final answer.

2. Code Analysis

Key Code

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

The `_execute_agent_workflow` function is the Agent's main execution loop, defining the core "planning -> execution" process.

```
# 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 order
            execution_results = []
            tool_outputs = []

            for i, step in enumerate(self.task_steps):
               # 2.1. Processing 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. Mission planning and LLM interaction (`largemodel/utils/ai_agent.py`)

The core of the `_plan_task` function is to build a sophisticated prompt and use the reasoning ability of the large model 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.
        """
        # 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 the 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)

        # ... (parses the JSON response and returns 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 will look up and return the value from
the previous_outputs list
                processed_params[key] = pattern.sub(replacer_function, value)

        return processed_params
```

Code Analysis

The AI Agent is the system's "central brain," transforming user-provided, high-level, and sometimes even fuzzy, tasks into a series of precise, ordered tool calls. Its implementation does not rely on any specific model platform but is built on a general, scalable architecture.

1. **Dynamic Task Planning**: The core capability of the Agent lies in the `_plan_task` function. It does not rely on hard-coded logic but dynamically generates task plans through interaction with a large model.

- **Self-Awareness and Prompt Construction**: At the start of planning, the Agent first examines all available tools and their descriptions. Then, it packages this tool information, the user task, and strict rules (such as data transmission formats) into a highly structured `planning_prompt`.
- **Model as Planner**: This Prompt is sent to a general text-based large model. The model infers based on the provided context and returns a JSON-formatted action plan containing multiple steps. This design is highly scalable: when tools are added or modified in the system, the Agent's planning capabilities are automatically updated without requiring modifications to the Agent's own code.

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

- **Data Reference Placeholders:** During the planning phase, the large model embeds special placeholders, such as `{{steps.1.outputs.data}}`, into the parameter values that need to transmit data.
- **Real-time Parameter Replacement:** In the main loop of `_execute_agent_workflow`, `_process_step_parameters` is called before each step. It uses regular expressions to scan all parameters of the current step. Once a placeholder is found, it retrieves 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 correct data transfer between steps.

- **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: it doesn't directly solve the problem, but rather builds a framework for an external, general-purpose inference engine (large model) to solve it. Through the two core mechanisms of "dynamic programming" and "data flow management," the Agent can orchestrate a series of independent tools into complex workflows capable of performing advanced tasks.

3. Practical Operation

3.1 Configuring an 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 you want to use**:
   Change the `llm_platform` parameter to the platform name 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. Starting the `largemodel` main program:

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

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

2. **Sending Text Commands**:

Open another terminal, enter the same Docker container, and run the following command:

```
ros2 run text_chat text_chat
```

Then start entering the text: "Generate an image similar to the current scene based on the current environment."

3. **Observations:**

In the first terminal running the main program, you will see log output showing that the system received a text command, invoked the `aiagent` tool, and then provided a prompt to the LLM. The LLM will analyze the detailed steps of the tool invocation. For example, in this question, the `seewhat` tool will be invoked to capture the image, which will then be provided to the LLM for parsing. The parsed text will then be sent back to the LLM as content for generating a new image.