9. Multimodal Autonomous Agent Applications

Note: Due to performance limitations, Raspberry Pi 5 2GB/4GB, RDK X5 4GB, and Jetson Orin Nano 4GB versions cannot run offline. Please refer to the online largemodel tutorial.

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 single user command, but is capable of **autonomously thinking, planning, and continuously invoking 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 the user makes a complex request that cannot be completed with a single tool call.

1.2 Implementation Principle Overview

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

1. **Thinking (Reason):** When the agent receives a complex goal, it first invokes a powerful Language Model (LLM) to "think". It asks itself: "What should I do first to achieve this goal? Which tool should I use?" The output of the LLM is not the final answer, but an action plan.
2. **Action (Act):** Based on the LLM's thinking results, the agent executes the corresponding action—calling `ToolsManager` to run the specified tool (such as `visual_positioning`).
3. **Observe (Observe):** The agent obtains the result of the previous action ("observation"), for example, `{"result": "The cup was found, located at [120, 300, 180, 360]"}`. 4. **Rethinking**: The agent submits the observations, along with the initial goal, to the LLM for a second round of "thinking." 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 **thinking -> action -> observation** cycle 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: 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. Processing data references in parameters before execution
                processed_parameters =
self._process_step_parameters(step.get("parameters", {}), tool_outputs)
                step["parameters"] = processed_parameters

                # 2.2. Executing 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 planning prompt
        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 Transfer**: 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 Task:**
{task_description}
"""

        # Calling the Large Model 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 "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. By leveraging two core mechanisms, dynamic programming and data flow management, the Agent orchestrates a series of independent tools into complex workflows capable of completing advanced tasks.

3. Practical Operations

3.1 Configuring the Offline Large Model

3.1.1 Configuring the LLM Platform (`yahboom.yaml`)

This file determines which large model platform the `model_service` node loads as its primary language model.

Raspberry Pi 5 requires entering a Docker container, while RDK X5 and Orin controllers do not.:

```
./ros2_docker.sh
```

If you need to enter other commands in the same Docker container later, simply enter ./ros2_docker.sh again in the host terminal.

1. **Open the file** in the terminal:

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

2. **Modify/Confirm** `llm_platform`:

```
model_service:                              #Model server node parameters
  ros__parameters:
    language: 'zh'                          #Large Model Interface Language
    useolinetts: True                       #This item is invalid in text mode
and can be ignored

    # Large model configuration
    llm_platform: 'ollama'                  # KEY: Make sure this is 'ollama'
    regional_setting : "China"
```

3.1.2 Configuration model interface (`large_model_interface.yaml`)

This file defines which visual model to use when the platform is selected as `ollama`.

1. Open the file in a terminal.

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

2. Find the Ollama-related configuration

```
#.....
## Offline Large Language Models
# Ollama Configuration
ollama_host: "http://localhost:11434"  # Ollama server address
ollama_model: "llava"              # Key: Change this to the multimodal model you
downloaded, such as "llava"
#.....
```

Note: Please ensure that the model specified in the configuration parameters (e.g., `llava`) can handle multimodal input.

3.2 Starting and Testing the Function

Note: Using a model with a small number of parameters or when running at a low memory usage limit will result in poor performance. For a better experience with this function, please refer to the corresponding section in <Online Large Model (Text Interaction)>.

1. **Start the `largemodel` main program**:
   Open a terminal and enter the Docker container, then run the following command:

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

2. After successful initialization, say the wakeup word and begin asking questions based on the current environment. Save the generated environment description to a text file.

3. **Observation**:
   In the first terminal running the main program, you will see log output indicating that the system receives the text command, invokes the `aiagent` tool, and then provides a prompt to LLM. LLM will analyze the detailed steps of invoking the tool. For example, for this question, the seewhat tool will be called to obtain the picture, and then the picture will be provided to LLM for parsing. The parsed text will be saved in the ~/yahboom_ws/src/largemodel/resources_file/documents folder.