

Multimodal Autonomous Agent Voice Interaction

Multimodal Autonomous Agent Voice Interaction

1. Concept Introduction
 - 1.1 What is an "Autonomous Agent"?
 - 1.2 Implementation Principle Overview
2. Code Analysis
 - Key Code
 1. Agent Core Workflow (`largetmodel/utils/ai_agent.py`)
 2. Task Planning and LLM Interaction (`largetmodel/utils/ai_agent.py`)
 3. Parameter Processing and Data Flow Implementation (`largetmodel/utils/ai_agent.py`)
 - Code Analysis
3. Practical Operation
 - 3.1 Configuring the Offline Large Model
 - 3.1.1 Configuring the LLM Platform (`yahboom.yaml`)
 - 3.1.2 Configure the model interface (`large_model_interface.yaml`)
 - 3.1.3 Recompile
 - 3.2 Startup and Testing

1. Concept Introduction

1.1 What is an "Autonomous Agent"?

In the `largetmodel` 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 Principle Overview

The autonomous agent implementation in `largetmodel` 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 cup's location. 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. / 执行Agent工作流: 规划 -> 执行。
        """
        try:
            # Step 1: Task Planning
            self.node.get_logger().info("AI Agent starting task planning phase")
            plan_result = self._plan_task(task_description)

            # ... (If the plan fails, return early)

            self.task_steps = plan_result["steps"]

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

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

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

                # 2.3. If the step is successful, 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, the entire task is aborted.
        return { "success": False, "message": f"Task terminated
because step '{step['description']}' failed." }

        # ... Summarize and return the final results
        summary = self._summarize_execution(task_description,
execution_results)
        return { "success": True, "message": summary, "results": execution_results }

    # ... (Exception handling)

```

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

The core of the `_plan_task` function is to construct a sophisticated Prompt, leveraging the large model's own inference capabilities to generate a structured execution plan.

```

# From largetmodel/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():
            # ... (Retrieves tool description from adapter.input_schema)
            tool_descriptions.append(f"- {name}({{params}}): {{description}}")
        available_tools_str = "\n".join(tool_descriptions)

        # Building a highly structured planning Prompt
        planning_prompt = f"""

```

作为一个专业的任务规划Agent，请将用户任务分解为一系列具体的、可执行的JSON步骤。

```

***# 可用工具:***
{available_tools_str}

***# 核心规则:***
1. **数据传递**: 当后续步骤需要使用之前步骤的输出时, **必须**使用
`{{{steps.N.outputs.KEY}}}` 格式进行引用。
    - `N` 是步骤的ID (从1开始)。
    - `KEY` 是之前步骤输出数据中的具体字段名。
2. **JSON格式**: 必须严格返回JSON对象。

***# 用户任务:***
{task_description}
"""

        # calling large models for planning
        messages_to_use = [{"role": "user", "content": planning_prompt}]
        # Note that this calls a generic text reasoning interface.
        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 (`\targemode1/utils/ai_agent.py`)

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

```
# From \targemode1/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()
        # The regular expression is used to match placeholders in the format
        # {{steps.N.outputs.KEY}}.
        pattern = re.compile(r"\{\{(\w+)(\.\w+)?(.\w+)?\}\}")

        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 searches for 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 "central brain" of the system. It transforms the high-level, and sometimes even fuzzy, tasks posed by the user into a series of precise and ordered..The agent invokes tools. Its implementation is independent of any specific model platform, instead built on a general, scalable architecture.

1. **Dynamic Task Planning:** The core capability of the agent lies in the `_plan_task` function. It doesn't 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, user tasks, 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 multi-step action plan in JSON format. This design is highly scalable: when tools are added or modified in the system, the agent's planning capabilities are automatically updated without modifying 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 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.

1. Open the file in the terminal:

```
vim ~/yahboom_ws/src/largemode1/config/yahboom.yaml
```

2. Modify/confirm `llm_platform`:

```
model_service:                      # Model server node parameters
  ros__parameters:
    language: 'en'                  # Large model interface language
    useolinettis: True              # This option is invalid in text mode
    and can be ignored

    # Large model configuration
    llm_platform: 'ollama'          # Critical: Ensure this is 'ollama'
```

3.1.2 Configure the model interface (`large_model_interface.yaml`)

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

1. Open the file in the terminal

```
vim ~/yahboom_ws/src/largemode1/config/large_model_interface.yaml
```

2. Locate the Ollama-related configuration

```
#.....  
## Offline Large Language Models  
# ollama Configuration  
ollama_host: "http://localhost:11434" # ollama server address  
ollama_model: "llava" # Key: Replace this with your downloaded multimodal model,  
such as "llava"  
#.....
```

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

3.1.3 Recompile

```
cd ~/yahboom_ws/  
colcon build  
source install/setup.bash
```

3.2 Startup and Testing

1. Start the `largemode1` main program:

Open a terminal and run the following command:

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

2. After successful initialization, say the wake word and then ask a question: Check the current environment and generate a similar image based on this scenario.

3. Observe the results:

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