On-Device Vision Language Model Based Natural Language Mission Execution Framework

Korea Aerospace University, Dept. of Aerospace Mechanical Engineering

Seunggyu Lee, Seungjun Oh, Sangwoo Jeon,

Youngju Park, Siwon Lee, Ghilmo Choi, Dae-Sung Jang

27 Oct. 2025 ICCAIS

Table of Contents

Introduction

Motivation

Fig.1 A humanoid robot is folding a laundry[1]

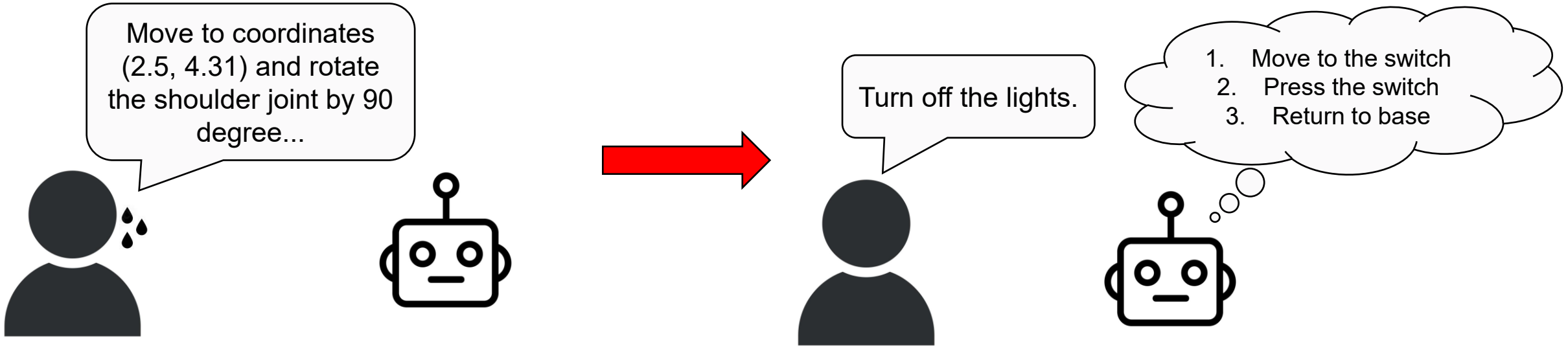
Fig.2 A humanoid robot is cleaning the floor with a vacuum[2]

As machine learning technology advances, robots that interact with humans and assist with daily tasks will emerge soon.

Image source
1) https://www.cnet.com/tech/computing/watch-figure-02-humanoid-fold-laundry-in-new-ai-demo/
2) https://tech.yahoo.com/general/articles/softer-humanoid-robot-home-120015267.html

Introduction

Motivation



To use autonomous robots conveniently in everyday life, robots must understand situational context and natural language. Also, they should be able to plan their actions on their own.

Introduction

Motivation

- We present a framework that enables robots to generate and execute action plans using an on-device Vision–Language Model (VLM).

- This on-device approach eliminates cloud dependency, enabling autonomous operation even in environments with unstable internet or strict privacy requirements.

System Configuration

Acquiring Environment Information

To generate an accurate plan, sufficient environmental context is required.

When a corresponding function is called, objects are detected using a YOLO model, and their positions are calculated using a depth image acquired from a depth camera.

The results are saved in JSON format with object ID and class.

<YOLO Object detection>

<Depth image(example)>

Localization Process

```
"objects": [
    {"id": “bottle#1",  "class": “bottle“, “map_xy”: [2.01, 3.46]}
]
```

<Generated JSON content>

System Configuration

Acquiring Environment Information

To generate an accurate plan, sufficient environmental context is required.

When a corresponding function is called, objects are detected using a YOLO model, and their positions are calculated using a depth image acquired from a depth camera.

The results are saved in JSON format with object ID and class.

→ Also, the position of the objects are visualized on a 2D map obtained through SLAM.

```
"objects": [
    {"id": “bottle#5",  "class": “bottle“, “map_xy”: [5.01, 9.46]}
]
```

<Generated JSON content>

System Configuration

Plan Generation

Control functions	Description
move_to(object)	Move to the detected object's location
pick(object)	Pick up the object (Executable only when near the object)
place(object)	Place the object at its designated position
search(object)	Move to search for the objects required for the task and update the environment information.
return_to_base()	Return to the starting point

< A table of control functions >

< Mobile robot >

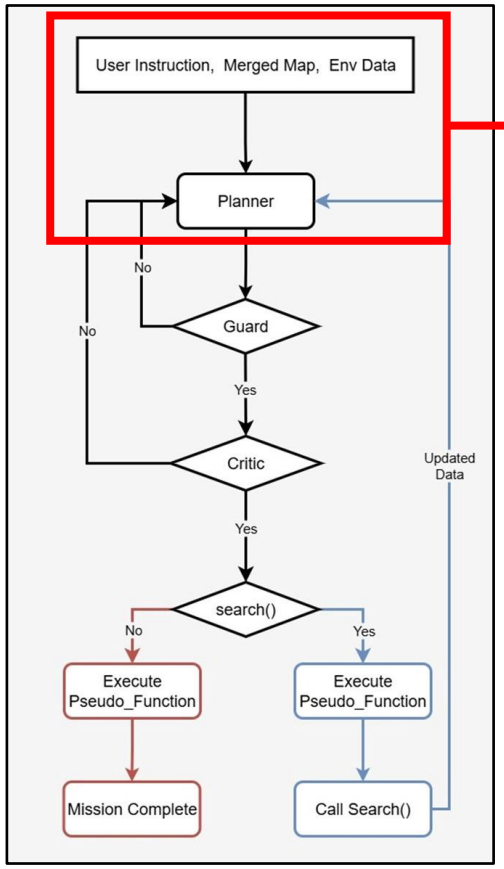
When a VLM receives a user command, it generates a plan by referring to contextual information.

The plan takes the form of a sequence of functions shown in the table.

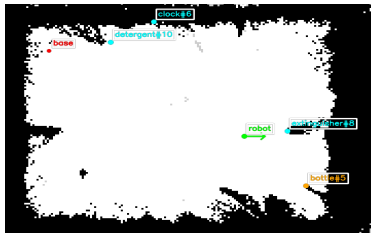
Each function is linked to the low-level control layer of a mobile robot, resulting in actual physical actions.

System Configuration

Plan generation and validation - Planner



User Instruction:
"Bring the water bottle and the apple to the person."



```
<World JSON>
        "objects": [
  {"id": "laundry#1",   "class": "laundry"},
  {"id": "washing_machine#1",  "class": "washing_machine"}
        ]
<Inventory JSON>
        "holding": [
  { "id": "laundry#1",   "class": "laundry"},
  { "id": "sponge#1",  "class": "sponge"}
        ]
```

in

Planner

out

Initial Plan

Planner prompt structure

Function Explanation: Defines how each control function works.

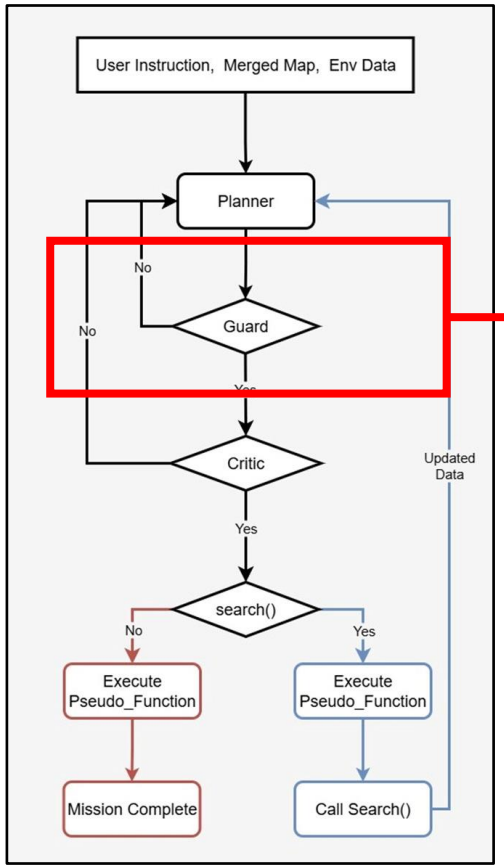
Constraints: Enforces strict rules to ensure the planner outputs only valid, consistent, and safe function-based action plans.

Output format: Ensures consistent and parseable responses.

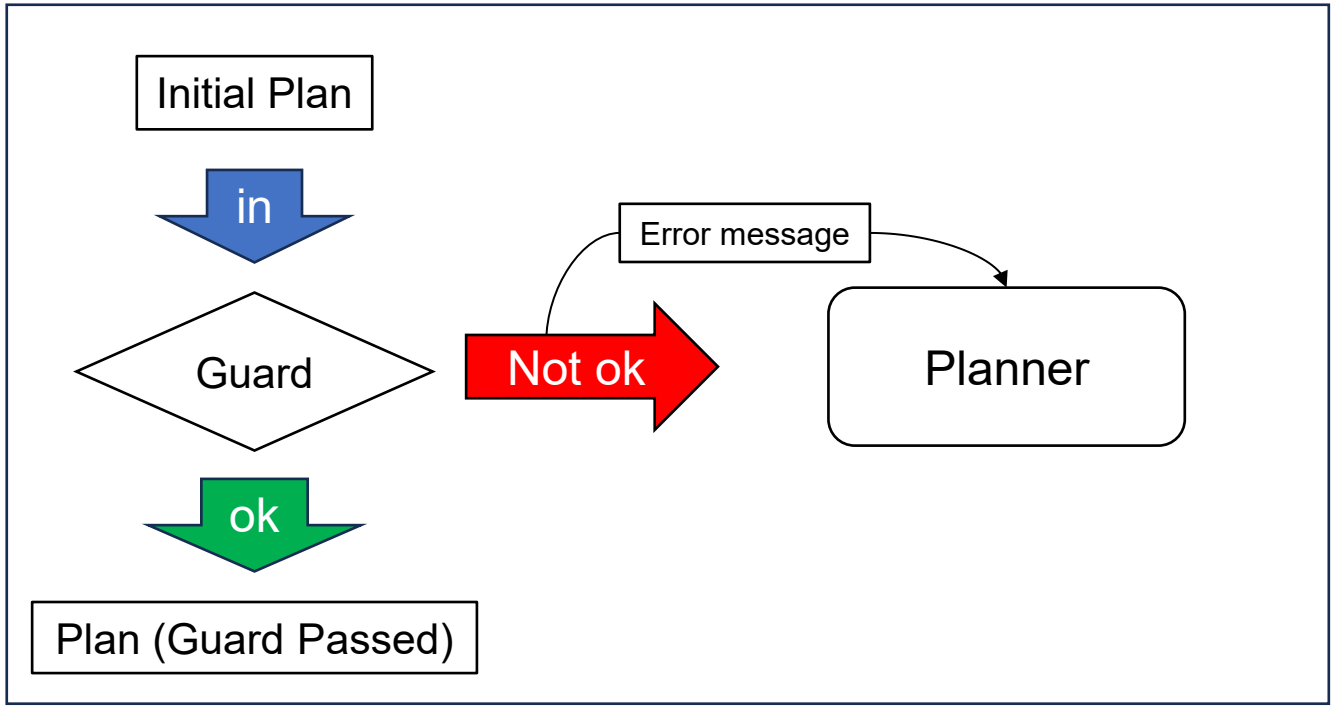
Two simple examples: Demonstrates desired input–output behavior.

System Configuration

Plan generation and validation - Guard

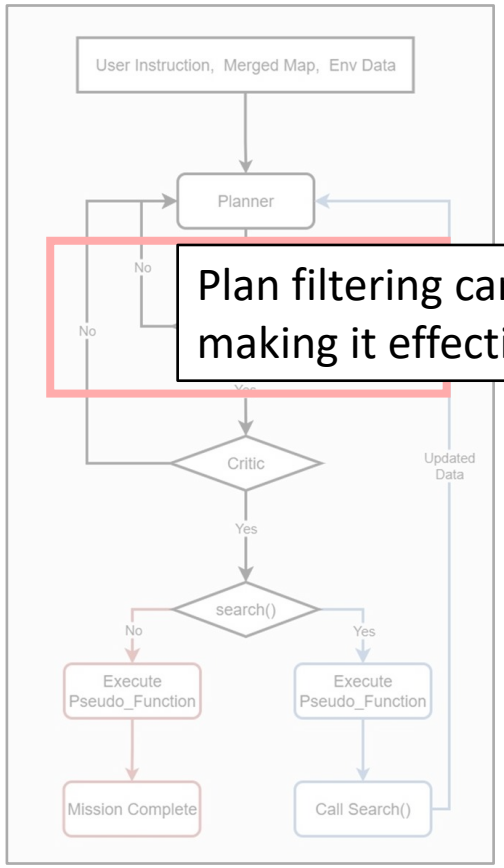


Verification of function validity based on predefined rules
Example: When a plan includes the function place("apple") for an item not possessed (“apple”), the system triggers an error message and requests the Planner to replan.



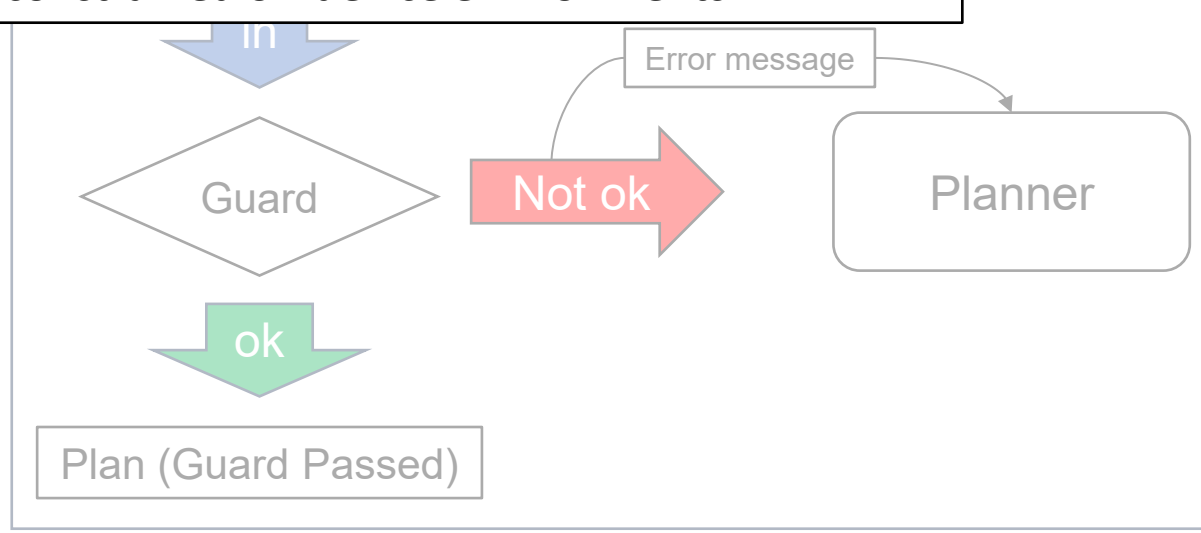
System Configuration

Plan generation and validation - Guard



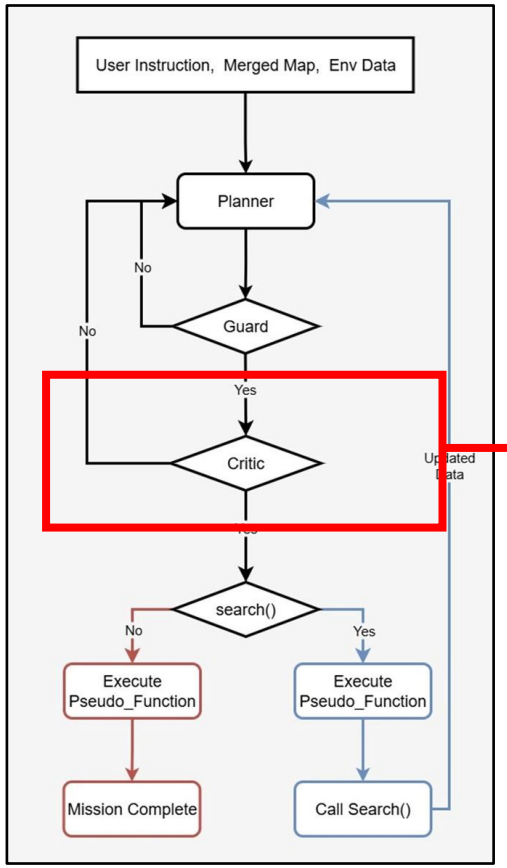
Verification of function validity based on predefined rules
Example: When a plan includes the function place("apple") for an item not possessed (“apple”), the system triggers an error message and requests the Planner to replan.

Plan filtering can be performed **without using computationally expensive VLMs**, making it effective for **resource-constrained on-device environments**.

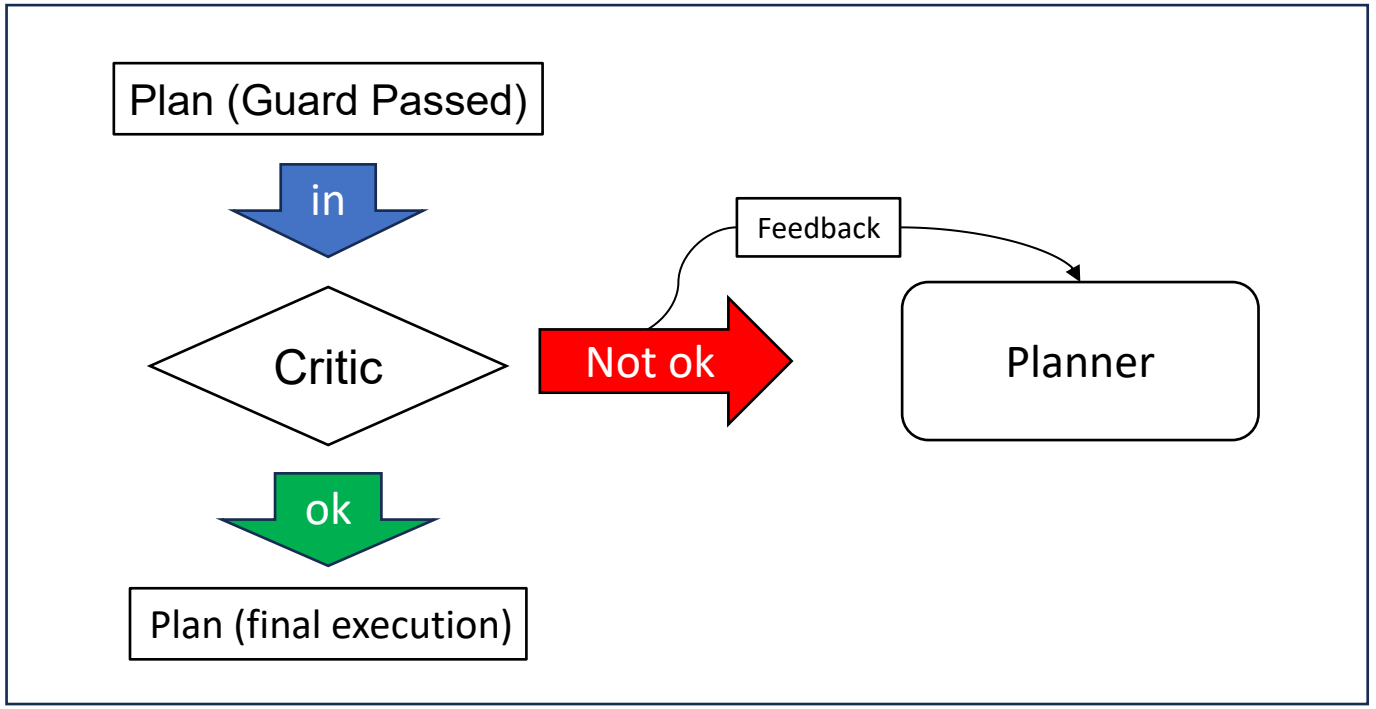


System Configuration

Plan generation and validation

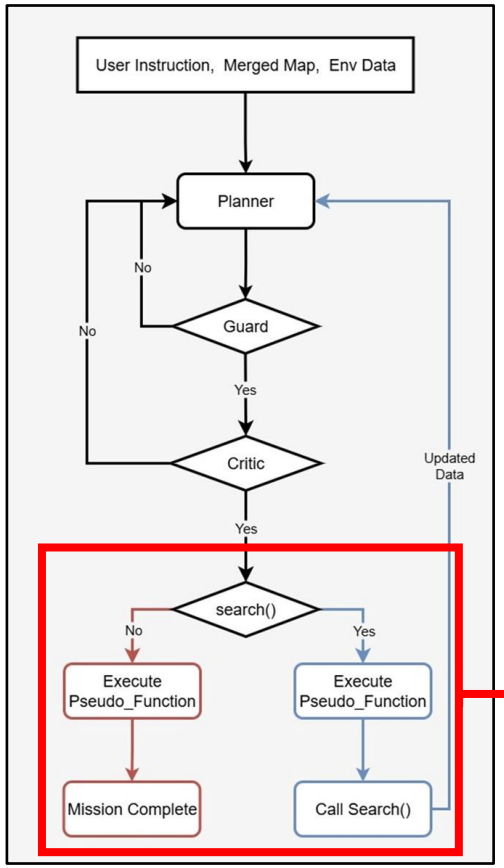


Critic evaluate the effectiveness of the generated plan in executing the given command. It receives the same user instruction and environmental information as the Planner, ensuring consistent contextual understanding.
If the plan is considered inappropriate, the Critic provides feedback and requests the Planner to replan.



System Configuration

Plan generation and validation



search(object)	Move to search for the objects required for the task and update the environment information.

Determination of Mission Completion Based on the Presence of the search() Function

1. When search() is included

→ Indicates that the mission cannot be completed with the current environmental information. After executing the search() function, the system replans using the updated environment data.

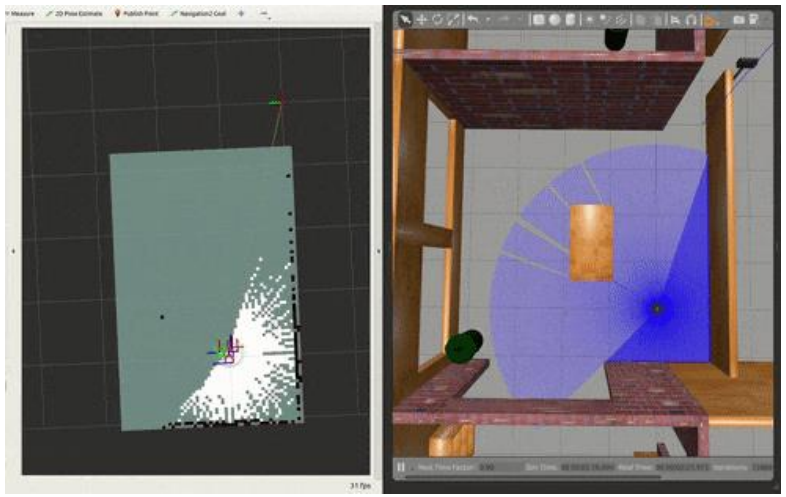
2. When search() is not included

→ Indicates that the mission can be completed with the given environmental information. The system executes all generated functions and then terminates the mission.

System Configuration

Implementation of Control Functions

Control functions	Description
move_to(object)	Move to the detected object's location
pick(object)	Pick up the object (Executable only when near the object)
place(object)	Place the object at its designated position
search(object)	Move to search for the objects required for the task and update the environment information.
return_to_base()	Return to the starting point



The movement control functions utilize the ROS2 Nav2 stack,

→ Automatically generates a safe navigation path when destination coordinates are provided.

Since the environmental information stores the positions of all objects, the Planner can make the robot move to the desired object's location without directly entering coordinates.

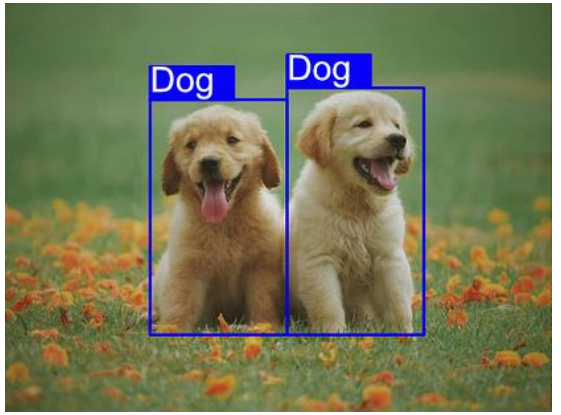
Image/GIF source: https://docs.nav2.org

System Configuration

Implementation of Control Functions

Control functions	Description
move_to(object)	Move to the detected object's location
pick(object)	Pick up the object (Executable only when near the object)
place(object)	Place the object at its designated position
search(object)	Move to search for the objects required for the task and update the environment information.
return_to_base()	Return to the starting point

Result when the function search("Dog") was called

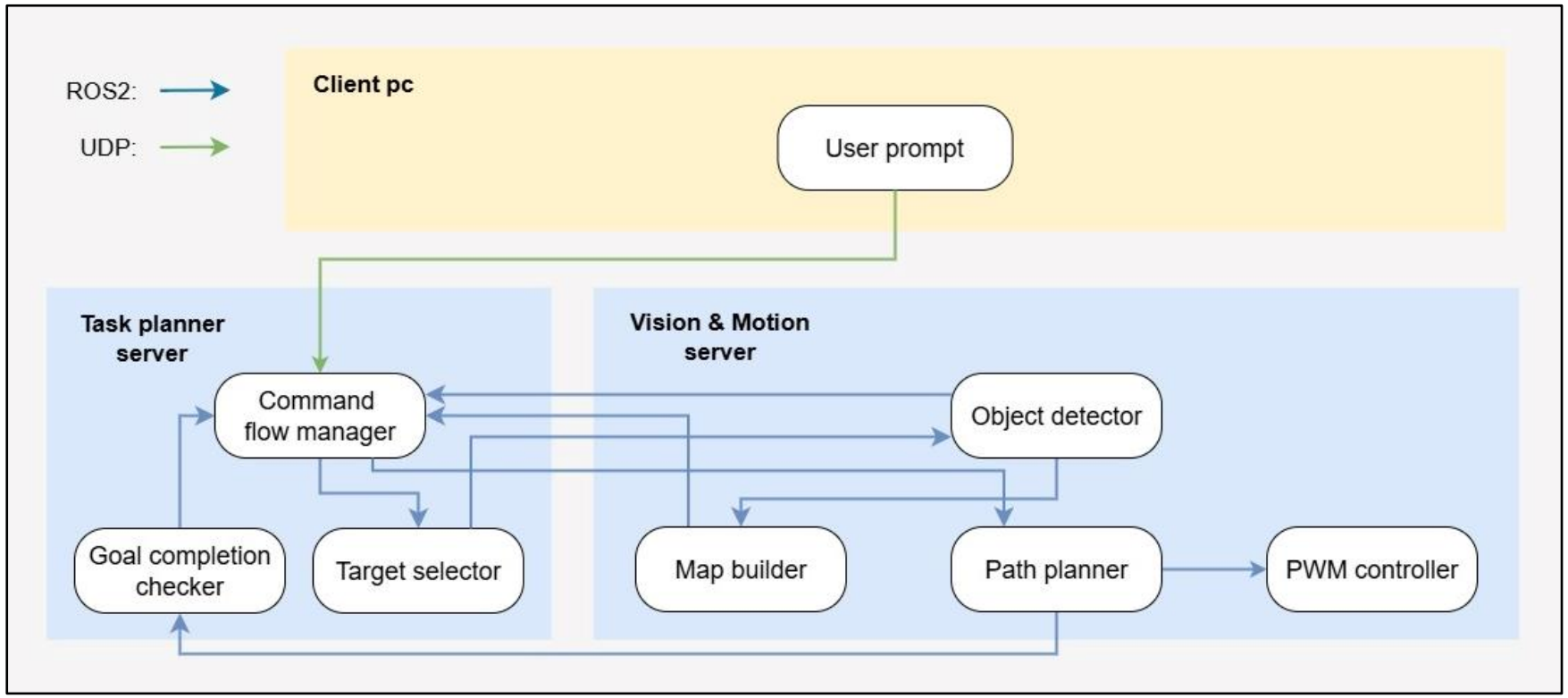


search() Function

- Utilizes the YOLO model for object detection.
- The Planner designates the target object, enabling detection of that specific item only.
- ∴ Enables selective information processing and reduces attention dispersion.
- The movement itself is manually controlled.

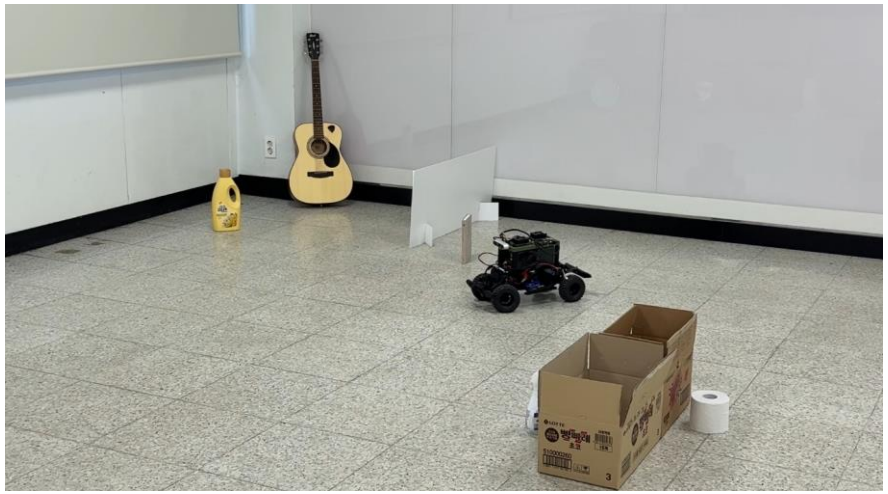
System Configuration

Overall architecture of the framework



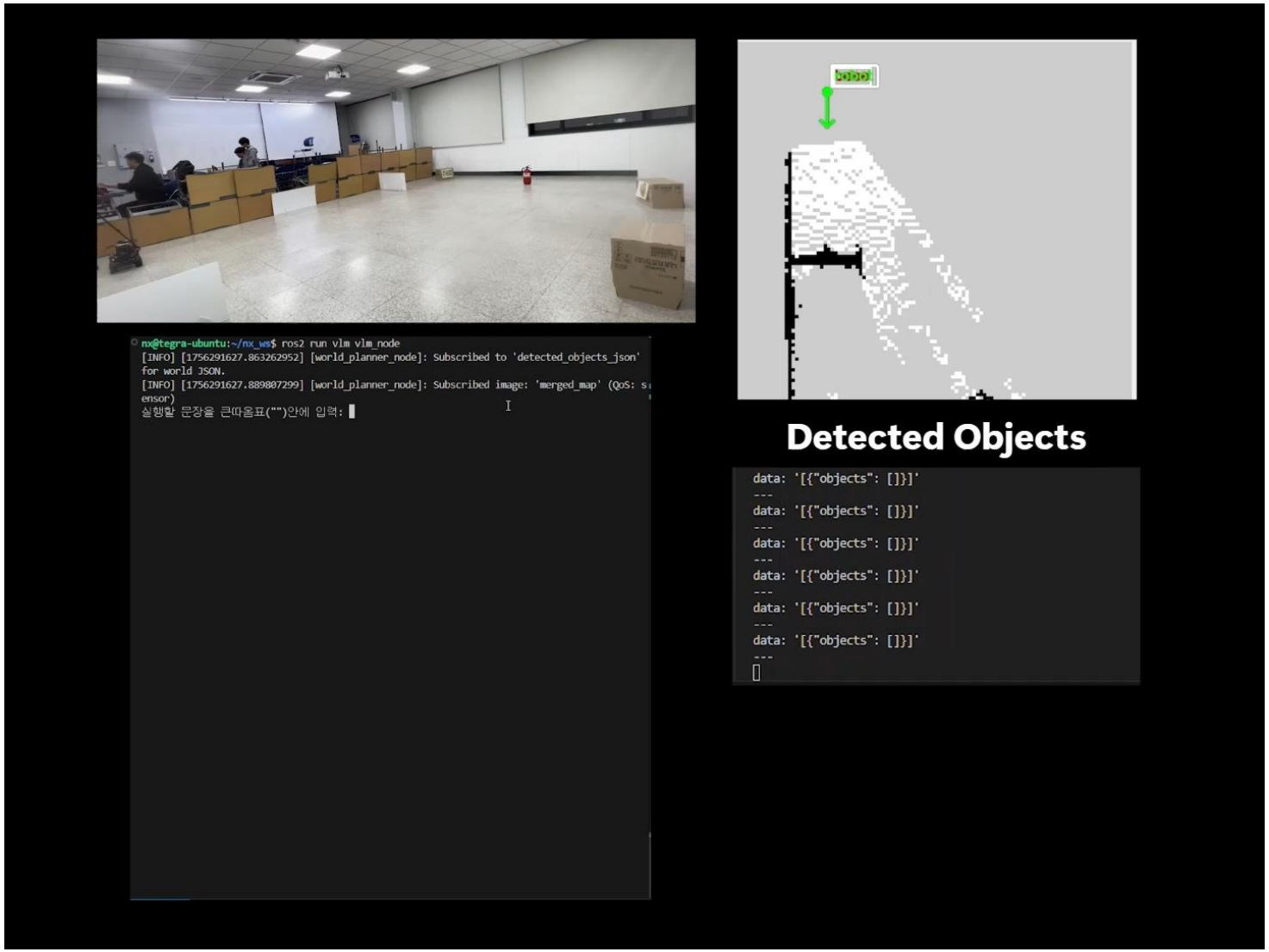
Experiments

Test Environment



To evaluate the proposed framework, we built a test space with objects and walls arranged to simulate an indoor environment.

Experiments



In this test, we used a Jetson Orin Nx 16GB board to run Qwen2.5VL(7B) model.

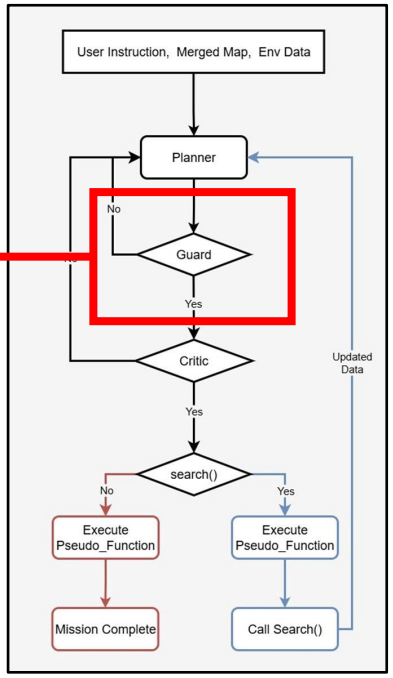
On average, one Planner–Guard–Critic loop took 15 seconds to complete.

Experiments

Results

User Instruction: “Go to the trash bin.”

===== PLAN (initial) =====
Reasoning: (omitted)
Pseudo_Function
```python
move_to(“trash bin#4”)
place(“ trash bin#4”)
return_to_base() ```
==========
[RUN] Guard failed → Replanning attempt 1
[GUARD] invalid: place(“trash bin#4”)
Place target ‘trash bin#4’ is not present in inventory_json.holding. Currently holding: []
===== PLAN (guard replan 1) =====
Reasoning: (omitted)
Pseudo_Function
```python
move_to(“ trash bin#4”)
return_to_base()```
==========




This example shows the Guard detecting an invalid plan from the Planner and sending feedback. The Planner then successfully generates a correct plan.

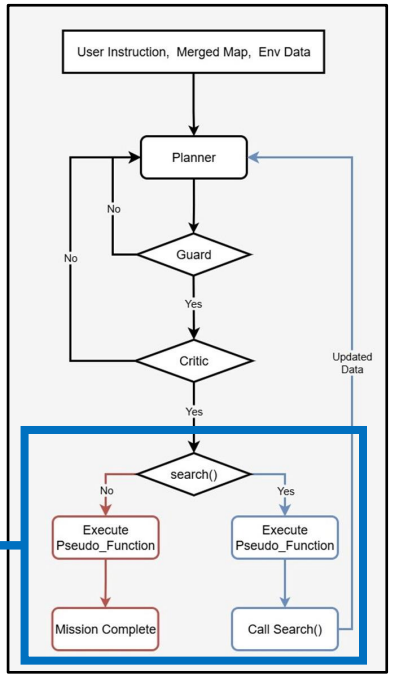
Rule-based filtering ensures basic execution stability under on-device constraints with minimal computational cost.

# Experiments

## Results

**[User command]**
“Bring the water bottle and the apple to the person.”
**[Initial environment information]**
" objects " : [
{ " id " : “person#1 " , "class": “person“}
]

**[Plan 1]**
search(“bottle”)
**[Plan 2]**
move_to(“bottle#1”)
pick(“bottle#1”)
move_to(“person#1”)
place(“bottle#1”)
search(“apple”)
**[Plan 3]**
move_to(“apple#1”)
pick(“apple#1”)
move_to(“person#1”)
place(“apple#1”)
return_to_base()




This example demonstrates a case where the Planner performs a relatively complex task.

- When the Planner determines that the current information is insufficient, it calls search() to explore the environment and updates the plan accordingly.
- Using search() as a planning checkpoint reduces the burden of generating too many commands at once.

# Experiments

## Limitations

- The Planner often failed to produce valid plans.
- Feedback from the Guard or Critic was not always effectively reflected by the Planner.
- The Critic occasionally misjudged valid or invalid plans.

→ Future work will focus on fine-tuning the VLM for Planner and Critic modules to overcome the observed limitations and improve overall planning reliability under on-device constraints.

# Conclusion

- We proposed an on-device vision–language model–based framework for natural language mission execution.
- The Planner–Guard–Critic pipeline shows potential to enhance plan validation and stability under limited resources.
- Experiments showed that rule-based filtering and structured feedback can improve planning with low computational cost.
- Future work aims for more robust and adaptive planning, integrating a robotic arm to expand system capabilities.

< Concept design of the future vehicle >

# Q & A

Every source code will be uploaded here!
https://github.com/ruga1026