Robotics-LLM Reading Party

- Chat with the Environment: Interactive Multimodal Perception using Large Language Models
- Code as Policies: Language Model Programs for Embodied Control

Weiqin Zu
ShanghaiTech University

2023.4.21

Chat with the Environment: Interactive Multimodal Perception using Large Language Models

Xufeng Zhao*, Mengdi Li, Cornelius Weber, Muhammad Burhan Hafez, and Stefan Wermter

The authors are with the Knowledge Technology Group, Department of Informatics, Universität Hamburg, 22527 Hamburg, Germany. E-mail: {xufeng.zhao, cornelius.weber, burhan.hafez, stefan.wermter}@uni-hamburg.de, mengdi.li@studium.uni-hamburg.de.

[Chat with the Environment: Interactive Multimodal Perception using Large Language Models]

Background

- How do humans perceive the surroundings to uncover latent properties?
  - Humans naturally perform multimodal observations and examinations using common sense and established knowledge in daily life.

➢ Robot well-equipped with multiple sensors and LLMs
  - ➢ choose stimuli to attend to, avoiding eagerly being bogged down into details
  - ➢ respond accordingly to the resulting sensations in the context of a specific task.

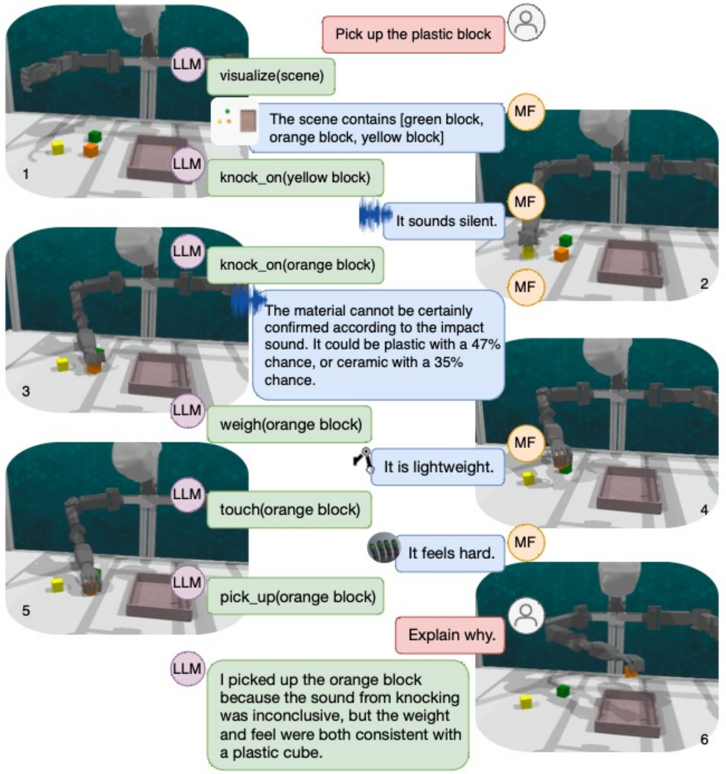


Fig. 1: Given instruction from a human, the robot recurrently "chats" with the environment to obtain sufficient information for achieving the task. An LLM generates action commands to interactively perceive the environment; And in response, the environment provides multimodal feedback (MF) through multimodal perception modules.

立志成才 报国裕民

Introduction

- Interactive Multimodal Perceptions
  - Like humans, robots can perceive the environment in either a passive or an interactive way
  - Interactive perception is complex and requires a mediating system to handle multiple types of sensory data.

- Chatting with the Environment
  - In terms of generalizability, the knowledge of LLMs allows a behavioral agent to adapt efficiently to novel concepts and environmental structures.

Matcha

- Matcha (multimodal environment chatting agent)
  - be able to interactively perceive (“chat” with) the environment through multimodal perception when the information from passive visual perception is insufficient for completing an instructed task.

- LLM backbone
- multimodal perception modules
- a low-level command execution policy.

- They connect via language as intermediate representation for information exchange.

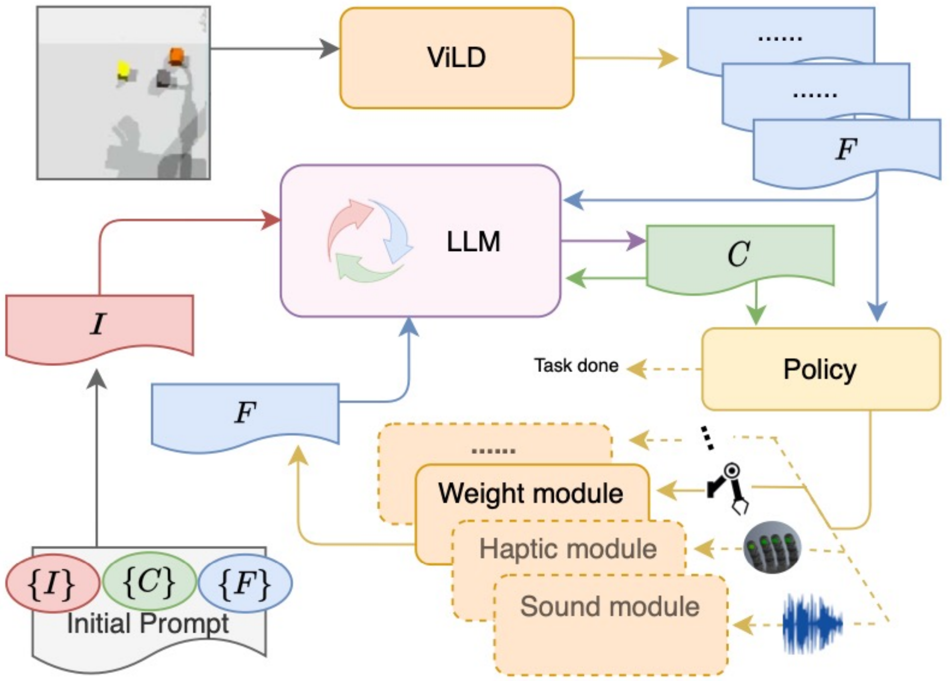


Fig. 2: Overview of Matcha. The framework contains an LLM, multimodal perception modules, and a language-conditioned policy. These components communicate with each other with natural language as the intermediate representation. Three types of language information are involved in composing the prompt: I is a language instruction from the user, C is a language command produced by the LLM, and F is semantic feedback from multimodal perceptions. Dotted lines indicate possibly evoking paths.

Matcha

- Vision
  - detect objects with their categories and positions in the scene. Then, the results will be delivered to a policy module for identification and execution.
- Impact Sound
- Weight
- Haptics
- Execution Policy

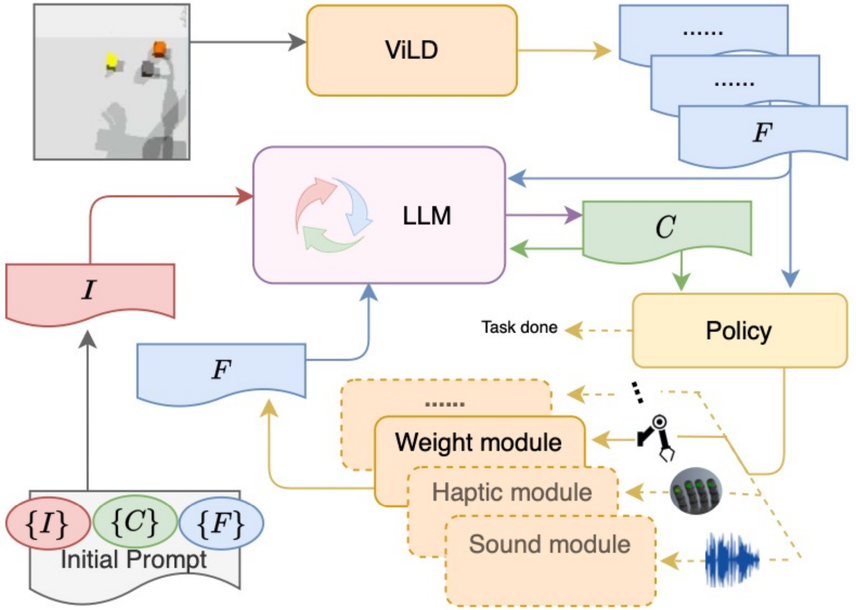


Fig. 2: Overview of Matcha. The framework contains an LLM, multimodal perception modules, and a language-conditioned policy. These components communicate with each other with natural language as the intermediate representation. Three types of language information are involved in composing the prompt: I is a language instruction from the user, C is a language command produced by the LLM, and F is semantic feedback from multimodal perceptions. Dotted lines indicate possibly evoking paths.

Experiments

上海科技大学 ShanghaiTech University

- Experimental Setup
  - a robot is instructed to pick up an object that is referred to by a latent property – material – which is, however, not visually distinguishable.
  - several blocks in various colors, materials, weights and surface textures and placed on the table next to a brown container
- Experiments are done in CoppeliaSim3 simulations

Materials	Impact Sound	Haptics	Weight
Metal	"resonant and echoing", "metallic", "ringing"	"hard and cold", "rigid, cold, and smooth"	"heavy", "300g"
Glass	"tinkling", "tinkling and brittle"	"hard", "hard and smooth", "cold and smooth"	"a little bit heavy", "150g"
Ceramic	"clinking and rattling", "rattling", "tinkling and brittle"	"hard", "tough"	"average weight", "not too light nor not too heavy", "100g"
Plastic	"dull", "muffled"	"hard", "soft"	"light", "30g"
Fibre	"muted", "silent"	"soft", "flexible"	"lightweight", "underweight", "10g"

TABLE I: Property descriptions of different materials.

LLM	Type of Description	Success Rate
text-ada-001	Indistinct	19.05%
	Distinct	28.57%
text-davinci-003	Indistinct	56.67%
	Distinct	90.57%

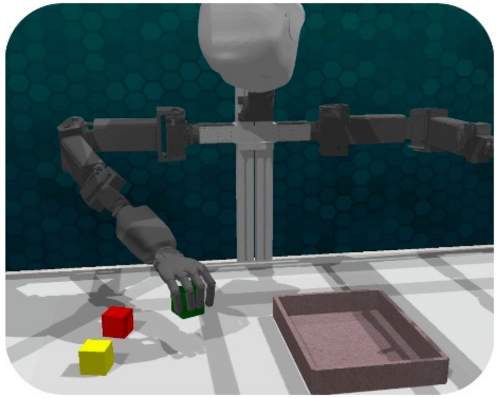
TABLE II: Effect of different LLMs on success rate.

立志 成才 报国 裕民

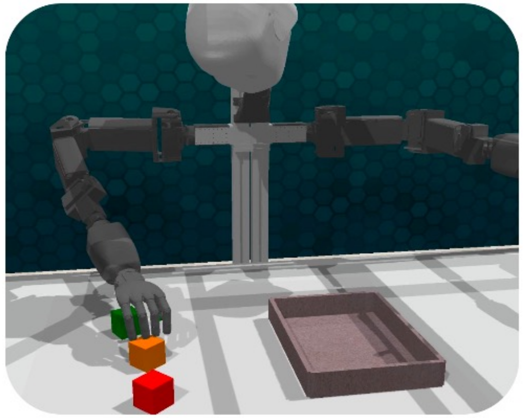
Experiments

- Case Studies

Human: "pick up the fibre block" in the scene that contains [yellow block, red block, green block].
AI: *robot.touch(red block)*
Feedback: It feels cold and smooth.
AI: *robot.touch(green block)*
Feedback: It feels flexible.
AI: *robot.knock_on(green block)*
Feedback: It sounds muted.
AI: *robot.weigh(green block)*
Feedback: It weighs underweight.
AI: *robot.pick_up(green block)*
Human: Explain why.
AI: The object is likely to be made of fibre due to its weight, flexibility, muted sound and smooth feeling. Therefore, I selected the green block as the fibre block and used the robot.pick up() skill to pick it up.
done()



Human: "pick up the metal block" in the scene that contains [orange block, red block, green block].
AI: *robot.knock_on(orange block)*
Feedback: The material cannot be certainly confirmed according to the impact sound. It could be plastic with a 30% chance, or metal with a 24% chance.
AI: *robot.weigh(orange block)*
Feedback: It weighs heavy.
AI: *robot.touch(orange block)*
Feedback: It feels rigid, cold, and smooth.
AI: *robot.pick_up(orange block)*
Human: Explain why.
AI: The sound, weight, and feeling of the object all indicate that it is a metal object. Therefore, I chose to pick up the orange block.
done()



Summary & Discussion

- demonstrate the superiority of using an LLM to realize interactive multimodal perception.
- propose Matcha, a multimodal interactive agent augmented with LLMs, and evaluate it on the task of uncovering object latent properties.

- Generalizability Evaluation
  - Property Combination
  - Utility Reasoning

Code as Policies: Language Model Programs for Embodied Control

Jacky Liang, Wenlong Huang, Fei Xia, Peng Xu, Karol Hausman, Brian Ichter, Pete Florence, Andy Zeng

Robotics at Google

Code as Policies: Language Model Programs for Embodied Control

Background

- Robots require the language to be grounded in order to establish a connection between the physical world, words, percepts, and actions.
  - rule-based methods: Use lexical analysis to understand language and inform policies, but struggle with new instructions.
  - data-driven methods: Learn language-to-action directly, but need a lot of data and can be expensive on real robots.

- how can LLMs be applied beyond just planning a sequence of skills?
  - orchestrating planning, policy logic, and control

- Code-completion synthesizes Python programs from docstrings.

- Models can be reused to write robot policy code using natural language commands.

- Policy code can process perception outputs and control primitive APIs.

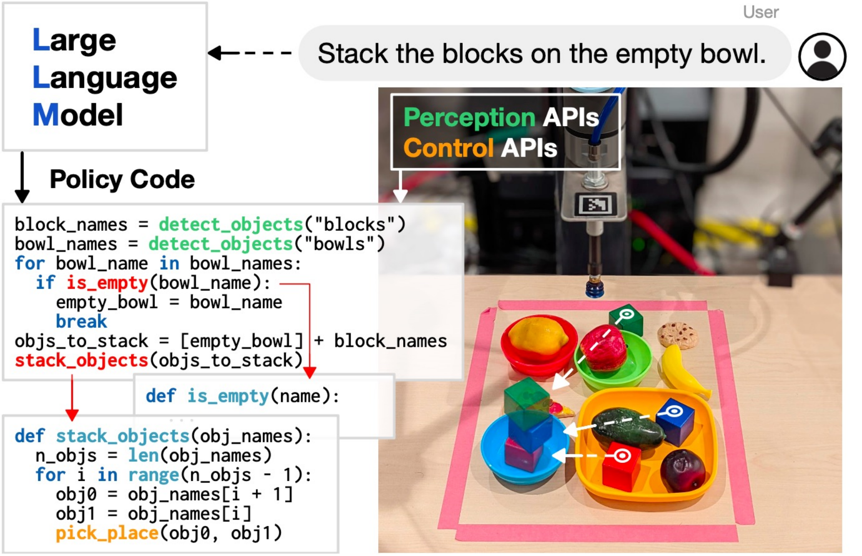


Fig. 1: Given examples (via few-shot prompting), robots can use code-writing large language models (LLMs) to translate natural language commands into robot policy code which process perception outputs, parameterize control primitives, recursively generate code for undefined functions, and generalize to new tasks.

Code as Policies

- CaP: a robot-focused approach to executing language model-generated programs (LMPs) on real systems.

- Pythonic LMPs can define sophisticated policies using:
  - Classic logic structures such as sequences, selection (if/else), and loops (for/while) to assemble new behaviors at runtime.
  - Third-party libraries for spatial-geometric reasoning, such as NumPy for point interpolation and Shapely for shape analysis and generation.

- LMPs can be hierarchical, allowing for the recursive definition of new functions, the accumulation of libraries, and self-architecting a dynamic codebase over time.

- Across multiple robot systems, they show that LLMs can independently interpret natural language commands to generate LMPs representing reactive low-level policies (e.g., PD or impedance controllers) and waypoint-based policies (e.g., for vision-based pick and place or trajectory-based control).

Code as Policies

- react to perceptual
- parameterize control primitive APIs
- are directly compiled and executed on a robot

```
# stack the blocks in the empty bowl.
empty_bowl_name = parse_obj('empty bowl')
block_names = parse_obj('blocks')
obj_names = [empty_bowl_name] + block_names
stack_objs_in_order(obj_names=obj_names)
```

- Functions defined by LMPs can progressively accumulate over time

```
# define function stack_objs_in_order(obj_names).
def stack_objs_in_order(obj_names):
    for i in range(len(obj_names) - 1):
        put_first_on_second(obj_names[i + 1], obj_names[i])
```

Code as Policies

- Prompting Language Model Programs
  - Hint: import statements that inform the and type hints on how to use those APIs.
  - Examples: instruction-to-code pairs that present few-shot "demonstrations" of how natural language instructions should be converted into code.

```
from utils import get_pos, put_first_on_second
...
# move the purple bowl toward the left.
target_pos = get_pos('purple bowl') + [-0.3, 0]
put_first_on_second('purple bowl', target_pos)
objs = ['blue bowl', 'red block', 'red bowl', 'blue block']
# move the red block a bit to the right.
target_pos = get_pos('red block') + [0.1, 0]
put_first_on_second('red block', target_pos)
# put the blue block on the bowl with the same color.
put_first_on_second('blue block', 'blue bowl')
```

- Example Language Model Programs (Low-Level)
  - Third-party libraries
  - First-party libraries
  - Language reasoning

```
objs = ['blue bowl', 'red block', 'red bowl', 'blue block']
# the bowls.
ret_val = ['blue bowl', 'red bowl']
# sea-colored block.
ret_val = 'blue block'
# the other block.
ret_val = 'red block'
```

Code as Policies

- Example Language Model Programs (High-Level)
  - Control flows: allow using control structures.
  - LMPs can be composed via nested function calls.
  - LMPs can hierarchically generate functions.
    - High-level LMPs can also follow good abstraction practices and avoid "flattening" all the code logic onto one level.
  - We use the function- generating LMP to write these undefined functions and add them to the scope.
  - Combining control flows, LMP composition, and hierarchical function generation.

```
# while the red block is to the left of the blue bowl, move it to the
right 5cm at a time.
while get_pos('red block')[0] < get_pos('blue bowl')[0]:
    target_pos = get_pos('red block') + [0.05, 0]
    put_first_on_second('red block', target_pos)
```

```
objs = ['red block', 'blue bowl', 'blue block', 'red bowl']
# while the left most block is the red block, move it toward the right.
block_name = parse_obj('the left most block')
while block_name == 'red block':
    target_pos = get_pos(block_name) + [0.3, 0]
    put_first_on_second(block_name, target_pos)
    block_name = parse_obj('the left most block')
```

```
import numpy as np
from utils import get_obj_bbox_xyxy
# define function: total = get_total(xs).
def get_total(xs):
    return np.sum(xs)
# define function: get_objs_bigger_than_area_th(obj_names, bbox_area_th).
def get_objs_bigger_than_area_th(obj_names, bbox_area_th):
    return [name for name in obj_names
            if get_obj_bbox_area(name) > bbox_area_th]
# define function: get_obj_bbox_area(obj_name).
def get_obj_bbox_area(obj_name):
    x1, y1, x2, y2 = get_obj_bbox_xyxy(obj_name)
    return (x2 - x1) * (y2 - y1)
```

Code as Policies

- Language Model Programs as Policies
  - In the context of robot policies, LMPs can compose perception- to-control feedback logic given natural language instructions, where the high-level outputs of perception model(s) (states) can be programmatically manipulated and used to inform the parameters of low-level control APIs (actions).

- The benefits of LMP-based policies are threefold
  - they can adapt policy code and parameters to new tasks and behaviors specified by unseen natural language instructions.
  - can generalize to new objects and environments by bootstrapping off of open-vocabulary perception systems and/or saliency models.
  - Don’t require any additional data collection or model training.

Experiments

- CaP: Drawing Shapes via Generated Waypoints
- CaP: Pick & Place Policies for Table-Top Manipulation
- CaP: Table-Top Manipulation Simulation Evaluations
- CaP: Mobile Robot Navigation and Manipulation

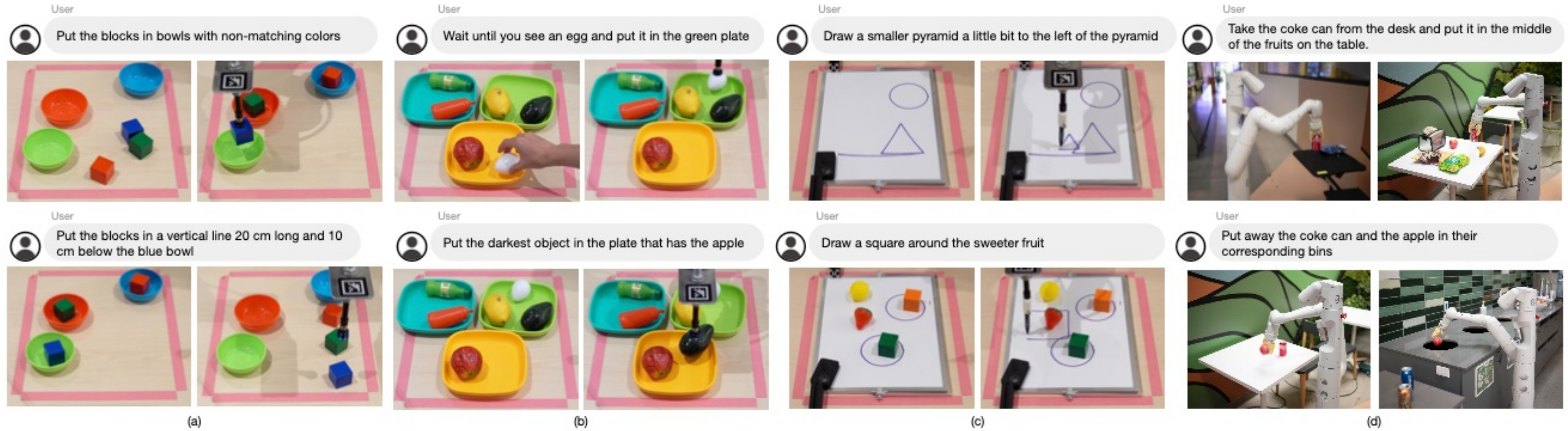


Fig. 2: Code as Policies can follow natural language instructions across diverse domains and robots: table-top manipulation (a)-(b), 2D shape drawing (c), and mobile manipulation in a kitchen with robots from Everyday Robots (d). Our approach enables robots to perform spatial-geometric reasoning, parse object relationships, and form multi-step behaviors using off-the-shelf models and few-shot prompting with no additional training. See full videos and more tasks at code-as-policies.github.io

Discussion & Limitation

- CaP generalizes at a specific layer in the robot stack.
- CaP fits into systems with factorized perception and control.
- Our method inherits LLM capabilities unrelated to code writing, such as supporting instructions with non-English languages or emojis (Appendix N).
- CaP can express cross-embodied plans that perform the same task differently depending on the available APIs .

- Limitation
  - Perception APIs limitations, e.g. visual-language models cannot describe some trajectories, and only limited primitive parameters can be adjusted.
  - Struggles with longer or complex commands, or those operating at different abstraction levels from given examples.
  - Assumes all given instructions are feasible and cannot predict correct response beforehand.

Thanks