Supplementary Material for Manipulate-Anything

Jiafei Duan [1*] **Wentao Yuan** [1*] **Wilbert Pumacay** [2] **Yi Ru Wang** [1]
Kiana Ehsani [3] **Dieter Fox** [1,4] **Ranjay Krishna** [1,3]
[1]University of Washington [2]Universidad Católica San Pablo
[3]Allen Institute for Artificial Intelligence [4]NVIDIA

Abstract: This is the supplementary material for MANIPULATE-ANYTHINGIt comprises the detailed implementation of MANIPULATE-ANYTHING prompts, simulation benchmark, and also the real-world experiments setup details. For more detailed videos on both simulation and real-world experiments, refer to our webpage: robot-ma.github.io.

Keywords: Zero-shot manipulation, multimodal language models, multiview state verification, robot skill generation, behavior cloning, robotic manipulation

1 MANIPULATE-ANYTHING implementations

Action Generation Module. We generate each action using either an agent-centric or object-centric approach. For object-centric action generation, we utilize M2T2[1], NVIDIA's foundational grasp prediction model, for `pick` and `place` actions. For 6-DoF grasping, we input a single 3D point cloud from either a single RGB-D camera (in the real world) or multiple cameras (in simulation). The model outputs a set of grasp proposals on any graspable objects, providing 6-DoF grasp candidates (3-DoF rotation and 3-DoF translation) and default gripper close states. For placement actions, M2T2 outputs a set of 6-DoF placement poses, indicating where the end-effector should be before executing a `drop` primitive action based on a VLM plan. The network ensures the object is stably positioned without collisions. We also set default values for `mask_threshold` and `object_threshold` to control the number of proposed grasp candidates. After proposing a list of template grasp poses, we use QWen-VL[2] to detect the target object by prompting the current image frame with the target object's name, translated into Chinese using a machine translation model [3]. This detection is applied to all re-rendered viewpoints or viewpoints from different cameras. We then concatenate these frames into a single image, annotating each sub-image with a number at the top right corner. Next, we call the GPT-4V API with few-shot demonstrations and the task goal to prompt GPT-4V to output the selected number of viewpoints that provide the most unobstructed views for sampling the grasp pose to achieve the sub-task. Using the selected viewpoint, we execute the grasp by moving the end-effector to the sampled grasp pose via a motion planner.

For agent-centric action generation, we first perform the same steps of viewpoint selection. Using the selected viewpoint, a few demonstration examples, and the sub-task, we prompt GPT-4V to generate an action function with code snippets that include the necessary code to perform a delta-action on the current robot pose. We then execute this by moving the end-effector based on these delta changes. This process is iterated until we obtain the most desirable code snippet function for the given sub-tasks, which is then appended to a skill library for future use.

Sub-task Verification Module The sub-task verification module helps with error recovery by ensuring all potential attempts at resolving the current step action have been tried. With the temporary goal state obtained by the action generation module, we use multi-viewpoints to sample the optimal viewpoint for answering the verification condition generated for the given sub-task during the task plan generation phase. Using the same viewpoint selection method as in the Action Generation Module, we obtain the optimal view and then perform a two-step sequence rollout of frames: one

*Equal contribution

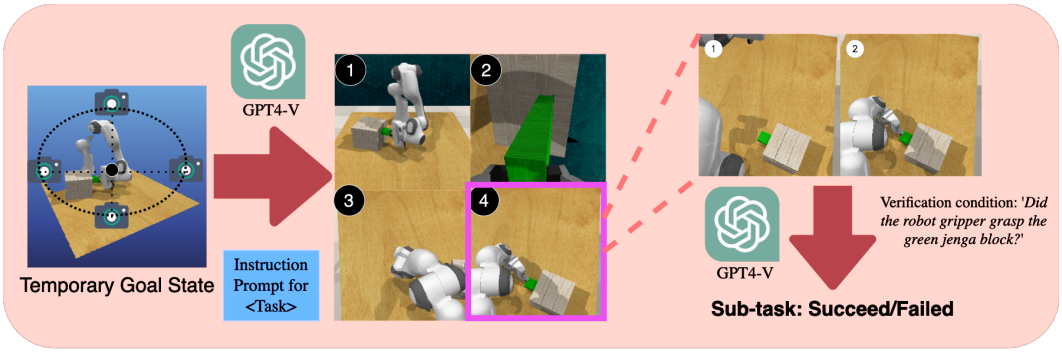

Figure 1: **Sub-task Verification Module.** We used viewpoint selection similar to action generation, to find the optimal viewpoints, and roll out the two step sequences of the previous and current frames for prompting the verification condition.

from the current frame at this viewpoint and another from the previous action step. We concatenate these two frames, annotate them with numbers to indicate their temporal relation, and use this image to prompt GPT-4V to check if the verification condition is fulfilled as shown in Fig. 1. If the answer is `Yes`, we proceed to the next sub-task. If the answer is `No`, we resample new viewpoints, generate new actions, and reattempt the entire sub-task with a different seed.

2 Simulation experiments

Simulation Setup. All the simulated experiments use a four-camera setup as illustrated in Fig. 3. The cameras are positioned at the front, left shoulder, wrist, and right shoulder. All cameras are static, except for the wrist camera, which is mounted on the end effector. We did not modify the default camera poses from the original RLBench [4]. These poses maximize coverage of the entire table, and we use a 256 x 256 resolution for better input to the VLMs.

Task Details. We describe in detail each of the 12 tasks for simulation evaluation, both for trained policies and zero-shot methods, along with their RLBench variations and success conditions. We have made some modifications to the original tasks to enhance the detection rate by Code-As-Policies and VoxPoser. The modified `.ttms` and `task.py` can be found here

2.1 `put block`

Filename: `put_block.py`

Task: Pick up the green block and place it on the red mat.

Success Metric: The success condition on the red mat detects the target green block.

2.2 `close box`

Filename: `close_box.py`

Task: Close the box.

Success Metric: The revolute joint of the specified handle is at least 60° off from the starting position.

2.3 `open box`

Filename: `open_box.py`

Task: Open the box.

Success Metric: The revolute joint of the specified handle is at least 60° off from the starting position.

2.4 play jenga

Filename: play_jenga.py

Task: Pull out the green jenga block.

Success Metric: The green jenga block is out of its pre-defined location.

2.5 open jar

Filename: open_jar.py

Task: Uncap the green jar.

Success Metric: The green jar is out of its pre-defined capped location.

2.6 pickup cup

Filename: Filename: pickup_cup.py

Task: Pick up the red cup.

Success Metric: Lift up the red cup above the pre-defined location.

2.7 take umbrella

Filename: take_umbrella_out_of_stand.py

Task: Pick up the umbrella out of the umbrella stand.

Success Metric: Lift up the umbrella out of the umbrella stand.

2.8 sort mustard

Filename: sort_mustard.py

Task: Pick up the yellow mustard bottle, and place it into the red container.

Success Metric: The yellow mustard bottle inside red container.

2.9 open wine

Filename: open_wine.py

Task: Uncap the wine bottle.

Success Metric: The wine bottle cap is out of its original position.

2.10 lamp on

Filename: lamp_on.py

Task: Turn on the lamp.

Success Metric: The lamp light up.

2.11 put knife

Filename: put_knife_on_chopping_board.py

Task: Pick up the knife and place it onto the chopping board.

Success Metric: Knife on chopping board.

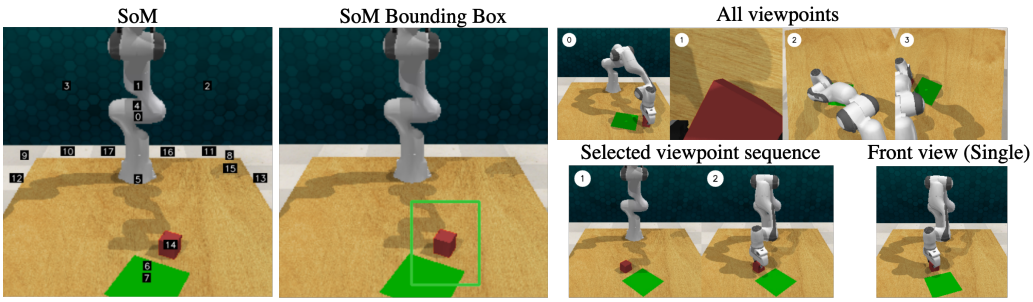

Figure 2: **Evaluation of visual prompting.** We systematic evaluate 5 different visual prompting techniques, and found that selected viewpoint sequence yields the highest performance.

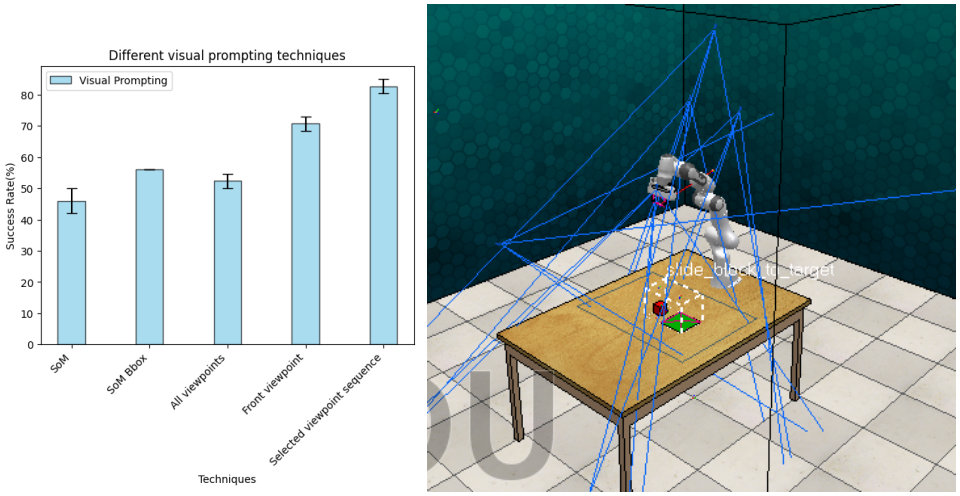

Figure 3: **Results for visual prompting techniques (Left).**We reported the various results for different visual prompting technique decision, and reported that selected viewpoint sequence yield the best performance.**Simulation scene setup (Right)**. We leverage 4 different camera for evaluation.

2.12 `push block`

Filename: `push_block_to_target.py`

Task: Push the red block down towards the green target.

Success Metric: The red block fails within the green target.

2.13 `insert block`

Filename: `insert_block.py`

Task: Push the green block into the jenga tower.

Success Metric: The green block inserted in.

2.14 `pick & lift`

Filename: `pick_and_lift.py`

Task: Pick up the red cube.

Success Metric: The red cube is lifted up.

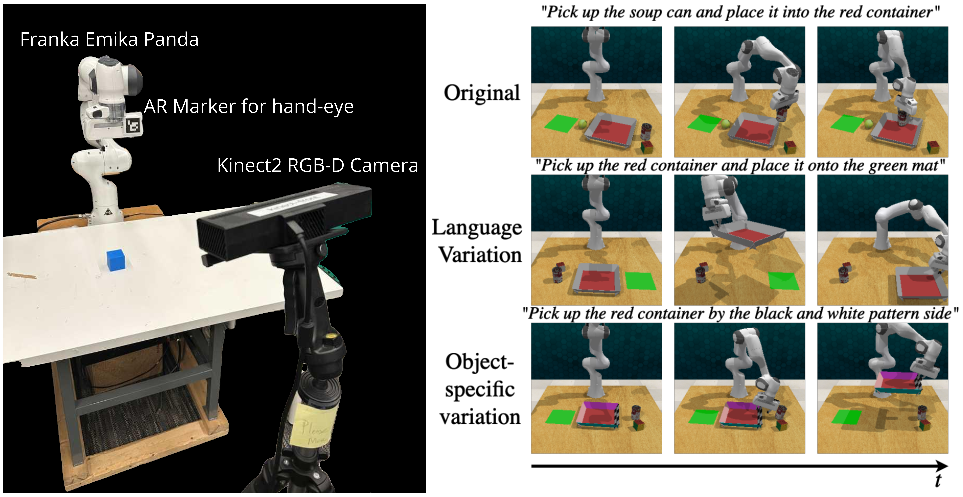

Figure 4: **Real-world experiment setup (Left).** We set up the real-world using this configuration. **Robustness and generalization evaluation.** We evaluated MANIPULATE-ANYTHING against VoxPoser for capability in generalizing to different language instructions and also object-specific manipulation.

Figure 5: **More real-world experiments.**

3 Real-world experiments

3.1 Robot hardware setup

The real-robot experiments use a Franka Panda manipulator with a parallel gripper. For perception, we use a Kinect-2 RGB-D camera mounted on a tripod, at an angle, pointing towards the tabletop. Kinect-2 provides RGB-D images of resolution 512 × 424 at 30Hz. The extrinsic between the camera and robot base-frame are calibrated with the easy hand-eye package. We use an ARUCO AR marker mounted on the gripper to aid the calibration process, as shown in Figure 4.

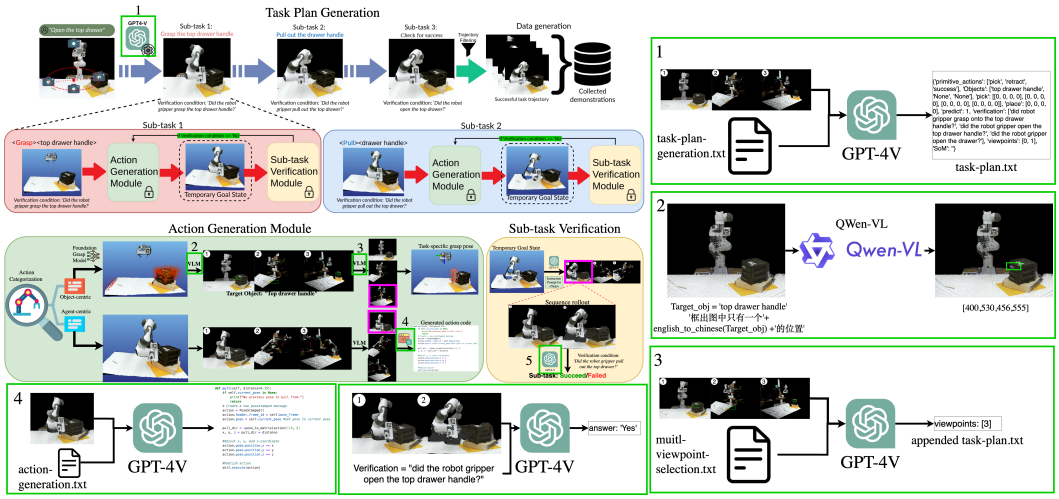

Figure 6: **In-depth Examination of VLMs for Manipulate-Anything.** The system consists of five components that utilize VLM API calls. We have provided a detailed breakdown of the inputs and outputs for each of these VLMs for reference.

3.2 Additional real-world everyday manipulation tasks

Beyond the five real-world experiments used for systematically evaluating MANIPULATE-ANYTHING, we also have additional real-world demonstrations generated in a zero-shot manner via MANIPULATE-ANYTHING. These demonstrations cover a range of tasks, from reasoning tasks to more precise everyday tasks. All of the tasks can be seen in Fig. 6.

4 Baseline implementation details

For most of the baselines we followed the original implementation with minor modifications. For the `Code as Policies` baseline we re-implemented most of the environment code using `PyRep` instead of `PyBullet`. This includes the implementation of various motion primitives that form the exposed API to the language model program. For example, one of such primitives is shown in the following Figure 7.

5 Additional Ablation Studies

We conducted two main set of ablation studies, we first look at how different visual prompting works for sub-task verification, and then we further evaluated MANIPULATE-ANYTHINGś robustness and generalization to language instructions in another set of experiments.

For evaluating different visual prompts for sub-task verification on the `put_block` task, we employed the following methods: 1) Set-of-Mark [5] on a single view, 2) Set-of-Mark with bounding box annotation, 3) Concatenated all viewpoints, 4) Front view only, and 5) Selected viewpoint sequence as shown in Fig. 2. We observed that the selected viewpoint sequences were the most effective in achieving correct sub-task verification, obtaining the highest success rate as shown in Fig. 3.

We further evaluated the generalization capabilities of our model in terms of object-specific manipulation and robustness to changes in language instructions. For language instruction variations, we altered the instructions for the same scene and found that MANIPULATE-ANYTHING outperforms VoxPoser by 60% over 25 episodes. For object-specific variations, where instructions targeted specific parts of objects, MANIPULATE-ANYTHING outperformed VoxPoser by 16%, as shown in Fig. 4.

6 Prompts

Prompts used for **multi-viewpoint VLM selection**, **task plan generation**, **action generation** and **sub-task verification** can be found below.

- **Task plan generation:** Takes in a natural language instruction, and outputs a task plan in json file with the correct format.
  Simulation: task_planner_generation.txt
- **Sub-task verification:** Takes in the selected viewpoint rollout along with the verification condition, and outputs binary 'yes' or 'no'.
  Simulation: sub-task-verification-prompt.txt
- **Action-Generation:** Takes the action primitive and generates codes for executing the action in simulation.
  Simulation: action_code_generation.txt
- **Multi-viewpoint VLM (Verification):** Takes in 4 images concatenated into a single frame with number annotated on top and returns a selection number of the most optimal viewpoint for verfying the sub-task.
  Simulation: MT_Viewpoint_Verification.txt
- **Multi-viewpoint VLM (Object centric action):** Takes in 4 images concatenated into a single frame with number annotated on top and returns a selection number of the most optimal viewpoint for filter the grasp candidate poses.
  Simulation: MT_Viewpoint_Verification_Object-centric.txt

7 Best Task Plans and Action Primitives

We ran MANIPULATE-ANYTHING with multi-processing during simulation to obtain the best set of task plans and action primitives for any given task. This set of information is then used to generate more data at scale for distilling a policy. We have compiled all the task plan JSON files and action primitive code that achieved the highest success rates in these tasks.
Best Generated Task Plans: best-task-plans.txt
Skills Library: skills-library.txt

References

[1] W. Yuan, A. Murali, A. Mousavian, and D. Fox. M2t2: Multi-task masked transformer for object-centric pick and place, 2023.

[2] J. Bai, S. Bai, S. Yang, S. Wang, S. Tan, P. Wang, J. Lin, C. Zhou, and J. Zhou. Qwen-vl: A frontier large vision-language model with versatile abilities. *arXiv preprint arXiv:2308.12966*, 2023.

[3] J. Tiedemann, M. Aulamo, D. Bakshandaeva, M. Boggia, S.-A. Grönroos, T. Nieminen, A. Raganato, Y. Scherrer, R. Vazquez, and S. Virpioja. Democratizing neural machine translation with opus-mt. *Language Resources and Evaluation*, pages 1–43, 2023.

[4] S. James, Z. Ma, D. R. Arrojo, and A. J. Davison. Rlbench: The robot learning benchmark & learning environment. *IEEE Robotics and Automation Letters*, 5(2):3019–3026, 2020.

[5] J. Yang, H. Zhang, F. Li, X. Zou, C. Li, and J. Gao. Set-of-mark prompting unleashes extraordinary visual grounding in gpt-4v. *arXiv preprint arXiv:2310.11441*, 2023.

```python
    def primitive_pick_place(self, obj1_name: str, obj2_name: str) -> None:
        """Motion primitive used for picking and placing an object

        Parameters
        ----------
            obj1_name: str
                The name of the object to pick
            obj2_name: str
                The name of the object on which to place the first
        """
        obj1_name = obj1_name.replace(" ", "_")
        obj2_name = obj2_name.replace(" ", "_")

        pick_obj = self.get_obj_if_exists(obj1_name)
        assert pick_obj is not None
        place_obj = self.get_obj_if_exists(obj2_name)
        assert place_obj is not None

        pick_pos, pick_rot = self.get_grasp_pose(
            cast(Shape, pick_obj), self.get_ee_orientation()
        )
        place_pos, place_rot = self.get_place_pose(
            cast(Shape, place_obj), self.get_ee_orientation()
        )

        pre_pick_pos = post_grasp_pos = pick_pos + np.array([0.0, 0.0, 0.2])
        pre_place_pos = place_pos + np.array([0.0, 0.0, 0.2])

        # Move to pre-pick position
        self.movep(ee_xyz=pre_pick_pos, ee_euler=pick_rot)
        while not np.allclose(pre_pick_pos, self.get_ee_position(), atol=1e-2):
            self.step()

        # Close in to pick position
        self.movep(ee_xyz=pick_pos, ee_euler=pick_rot, ignore_collisions=True)
        while not np.allclose(pick_pos, self.get_ee_position(), atol=1e-2):
            self.step()

        # Close the gripper
        while not self._robot.gripper.actuate(0.0, 0.04):
            self.step()
        self._robot.gripper.grasp(pick_obj)

        # Move to post-grab position
        self.movep(post_grasp_pos, ignore_collisions=True)
        while not np.allclose(
            post_grasp_pos, self.get_ee_position(), atol=1e-2
        ):
            self.step()

        # Move to pre-place position (to avoid collisions)
        self.movep(
            ee_xyz=pre_place_pos, ee_euler=place_rot, ignore_collisions=True
        )
        while not np.allclose(pre_place_pos, self.get_ee_position(), atol=1e-2):
            self.step()

        # Move to place position
        self.movep(ee_xyz=place_pos, ee_euler=place_rot, ignore_collisions=True)
        while not np.allclose(place_pos, self.get_ee_position(), atol=1e-2):
            self.step()

        # Open the gripper to release the object
        self._robot.gripper.release()
        while not self._robot.gripper.actuate(1.0, 0.04):
            self.step()
```

Figure 7: **Example of one of the primitives implemented for Code as Policies**