Institute of
Artificial Intelligence

Leibniz
Universität
Hannover

Electrical Engineering and Computer Science,
Institute of Artificial Intelligence (LUHAI)
Appelstraße 9a
30167 Hannover

Machine Learning Project Report

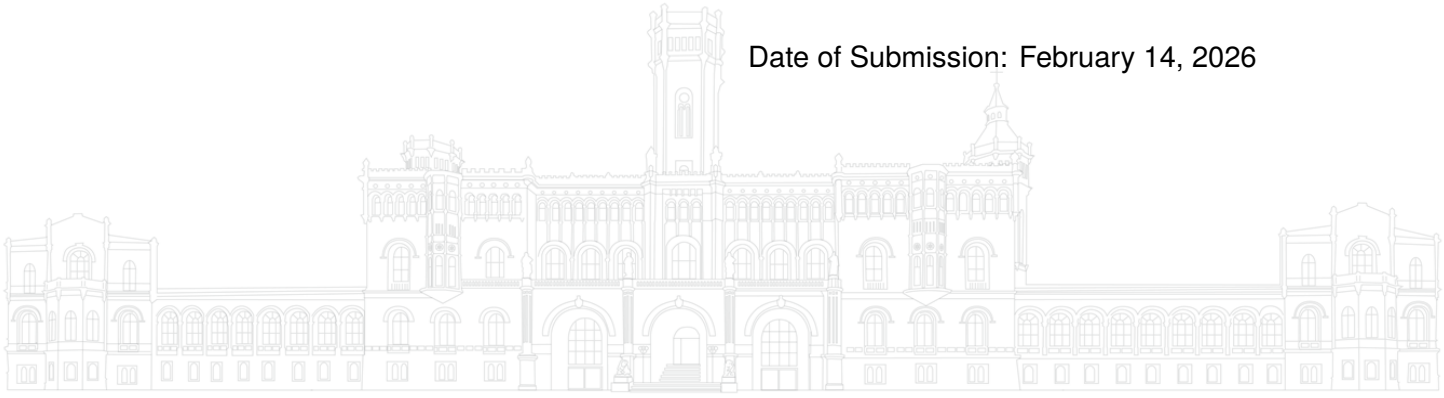
Robo Voyager: Autonomous Skill Learning for Robotic Manipulation in Isaac Sim

Johny Tarbouch

1. Reviewer Prof. Dr. rer. nat. Marius Lindauer
Electrical Engineering and Computer Science
Leibniz University Hannover

Supervisors Dr. Theresa Eimer

Date of Submission: February 14, 2026



Johny Tarbouch
Robo Voyager:
Autonomous Skill Learning for Robotic Manipulation in Isaac Sim
Machine Learning Project Report, February 14, 2026
Reviewers: Prof. Dr. rer. nat. Marius Lindauer
and Prof. Dr. rer. nat. Henning Wachsmuth
Supervisors: Dr. Theresa Eimer
Automated Machine Learning (AutoML)
Institute of Artificial Intelligence (LUHAI)
Electrical Engineering and Computer Science
Welfengarten 1
30167 Hannover

Contents

1 Introduction

Robotic manipulation, grasping, moving, and arranging objects, remains a central challenge in robotics. Traditional reinforcement learning policies and imitation learning require many per-task effort and generalize poorly (Ahn et al., 2022), in contrast to human manipulation where a single grasping skill transfers naturally across tasks.

Large language models (LLMs) offer a different paradigm: rather than learning policies from rewards or demonstrations, LLMs can *generate executable code* that controls a robot directly. *Code as Policies* (Liang et al., 2023) showed that LLMs can translate natural language instructions into robot API programs, while (Ahn et al., 2022) grounded such plans in a robot's affordances via a learned value function.

Voyager (Wang et al., 2024), an open-ended agent in Minecraft, pushed this paradigm furthest with three innovations: (1) an **automatic curriculum** where the LLM proposes its own tasks, (2) **iterative code refinement** with self-reflection, and (3) a **persistent skill library** for storing and reusing successful programs.

We ask the following research question: *can Voyager's paradigm work for physical robotic manipulation?* We present **Robo-Voyager**, an autonomous skill learning agent for a Franka Panda arm (Haddadin et al., 2022) in NVIDIA Isaac Sim (NVIDIA, 2024). Unlike Minecraft, manipulation introduces continuous state spaces, contact dynamics, and millimeter-level precision requirements.

Robo-Voyager operates in a closed-loop cycle in which an LLM proposes tasks, generates code, executes it in a sandboxed simulation, and verifies outcomes via an LLM-based critic that compares pre- and post-execution states. Verified skills are stored in a Qdrant-backed vector database using sentence-transformer embeddings, enabling semantic retrieval for future tasks. The system supports hierarchical skill composition, allowing complex manipulation behaviors (e.g., stacking or arrangement) to be constructed from previously learned primitives such as pick-and-place. Overall, this provides a complete Voyager-style pipeline for robotic manipulation, combining open-ended curriculum generation, safe code execution, automated state-based verification, scalable experience accumulation, and reusable skill composition. The full source code is available at: `https://github.com/JohnyTarbouch/voyager-robotic-isaac.git`.

2 System Architecture

Robo-Voyager follows an iterative learning loop in which each cycle proposes a task, generates code, executes it in simulation, and evaluates the outcome 2.1. The system is implemented and interacts with Isaac Sim through a custom robot API wrapper. The following subsections detail each component and the design decisions behind them.

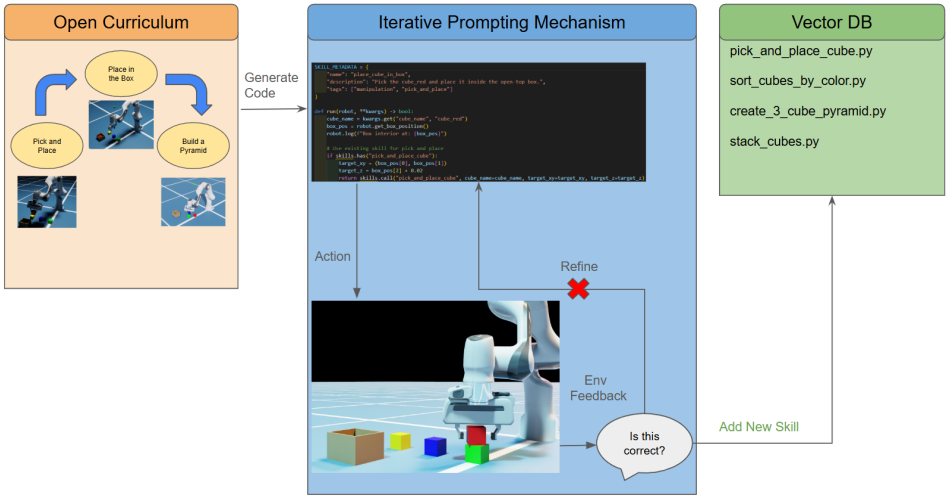


Figure 2.1.: Architecture of the Robo-Voyager learning loop. The agent cycles through task proposal, code generation, sandboxed execution, and LLM-based self-verification. Successful skills are stored for future reuse, failures trigger self-reflection and retry.

2.1 Environment: Isaac Sim and the Franka Robot

The simulation environment is built on NVIDIA Isaac Sim (NVIDIA, 2024), a GPU-accelerated robotics simulator with PhysX-based rigid body dynamics. The robot is a 7-DoF Franka Emika Panda (Haddadin et al., 2022), one of the most widely used research platforms in manipulation. End-effector motion is computed by **RMPflow** (Ratliff et al., 2018), a reactive motion policy framework that handles joint-level inverse kinematics, allowing generated code to operate in Cartesian space (target XYZ positions) rather than joint space.

The gripper uses **physics-based contact control**: rather than closing to a fixed width, it closes until contact force is detected, meaning it naturally stops at the cube surface. The robot API exposes the following methods to generate code:

- `move_ee(target_xyz, pos_tolerance)` -> End-effector motion
- `open_gripper(width, steps)` / `close_gripper(steps)` -> physics-based gripper control
- `get_object_position(name)` / `list_objects()` -> scene queries
- `get_observation()` -> full state dictionary (joint positions, EE pose, all object positions)
- `log(msg)` -> in-simulation logging for debugging

2.2 Open-Ended Curriculum

Rather than providing a fixed task list, Robo-Voyager uses an **LLM-driven curriculum** that autonomously proposes new tasks. The curriculum sends three informations to the LLM: (1) the list of objects currently in the scene (obtained via the robot API), (2) a summary of all previously learned skills with their descriptions, and (3) a history of previously attempted tasks and their outcomes (success/failure). The LLM is prompted to propose tasks of *gradually increasing difficulty*.

2.3 Planner: LLM-Based Code Generation

The Planner is responsible for translating a task description into executable Python code. It sends a structured prompt to the LLM containing:

- A **robot API specification**: the complete list of available methods with type signatures and behavioral notes
- **Domain-specific constraints**: guidance on grasp heights, lift tolerances, placement verification
- A **list of available skills** for composition, including each skill's name, description, tags, and accepted keyword arguments
- The **current observation**: a full state snapshot including end-effector position, gripper width, and all object positions

The LLM is explicitly instructed to write *generalizable* skills: parameters like cube names and target positions should be passed through `kwargs` with task-specific defaults, and skill names must be generic.

Self-Reflection on Failure. When code execution fails,either due to a runtime error, a timeout, or critic rejection, the Planner performs *self-reflection*. The error traceback or critic reasoning is appended to the original prompt, and the LLM is asked to generate a corrected version. This iterative refinement continues for up to N attempts (default $N{=}3$), with each attempt receiving the cumulative feedback from all prior failures.

2.4 Sandbox: Safe Code Execution

Since the LLM generates arbitrary Python code, executing it without safeguards would be dangerous, the code could contain uninstalled libraries, make network requests, or enter infinite loops. Robo-Voyager addresses these risks with a multi-layered safety system. The LLM is restricted to using only the `math` library, basic type constructors, and the Robot API.

2.5 Critic: LLM-Based Verification

After execution, the Critic agent determines whether the task was completed successfully. Unlike simply trusting the `run()` function's return value (which may be incorrect due to LLM reasoning errors), the Critic performs an **independent state-based assessment**. It receives:

- The original task description and success criteria
- The pre-execution environment state (all object positions, end-effector position, gripper width)
- The post-execution environment state
- The code that was executed
- Execution logs from `robot.log()` calls

The critic is instructed to be strict but fair, small position errors ($< 0.05\,\text{m}$) are acceptable due to robot precision limits, but the physical outcome must meaningfully match the task goal.

This dual-verification design (return value from code *and* independent critic assessment) improves reliability, catching cases where the LLM-generated code reports success but the physical outcome does not match, where the code returns `False` due to an overly strict self-check but the task was actually completed.

2.6 Skill Library and Retrieval

If a skill passes the code check and the Critic's check, we store it in two systems:

1. **File storage**: Each skill is saved as a standalone.

2. **Vector database**: Skills are indexed in Qdrant (Qdrant, 2024), based on the skill discription NOT the source code. Each skill is embedded using a sentence-transformer model (`all-MiniLM-L6-v2`, producing 384-dimensional embeddings) (Reimers et al., 2019).

When a new task arrives, the Planner queries the vector database with the task description and retrieves the top-k most relevant skills by cosine similarity based on the discription (not the function name) of the skill. These skills are included in the LLM prompt as available tools, complete with their names, descriptions, and accepted keyword arguments.

3 Experiments and Results

3.1 Environment Setup

All experiments are conducted in NVIDIA Isaac Sim (NVIDIA, 2024), using the Omniverse Kit runtime with GPU-accelerated PhysX physics. The robot is a 7-DoF Franka Emika Panda arm (Haddadin et al., 2022) controlled via RMPflow (Ratliff et al., 2018) for Cartesian end-effector motion planning. The scene contains four colored cubes (red, green, blue, yellow, each $0.05\,\mathrm{m}^3$) placed on a (empty) surface and an open box, all within the robot's reachable workspace ($x \in [0.3, 0.7]$, $y \in [-0.4, 0.4]$).

The LLM backend is **Llama 3.3 70B Instruct** (Touvron et al., 2023). Skill generation calls use temperature 0.25 and a limit of 2 500 tokens, critic verification calls use temperature 0.1 and 500 tokens. All experiments use the same prompt templates described in Section 2.3.

3.2 Evaluation Protocol

We evaluate Robo-Voyager across **five independent runs**, each starting from scratch with only the four seed skills. In each run the open-ended curriculum proposes tasks autonomously and the agent attempts each with up to 3 retry attempts. We report per-task metrics (success/failure, attempts, LLM calls, token usage, latency) and aggregate statistics across all runs. One representative run (Run 5) is analyzed in detail.

Over the five runs, the curriculum generated nine distinct task types spanning three difficulty tiers: Basic manipulation, Multi-object arrangement then Complex structures.

3.3 Results

Table 3.1 summarizes the aggregate performance across all five runs.

Metric	Value
Independent runs	5
Total tasks attempted	39
Total tasks completed	37
Overall success rate	94.9%
Mean success rate per run	95.1%
Mean attempts per task	1.37
First-attempt success rate	71.8%
Total tokens consumed	$\sim$522K
Mean tokens per successful task	$\sim$13 700
Mean session duration	$\sim$21 min

Table 3.1.: Aggregate evaluation results across five independent runs. Three runs achieved 100% success, two had one failure each (an L-shape and a square pattern).

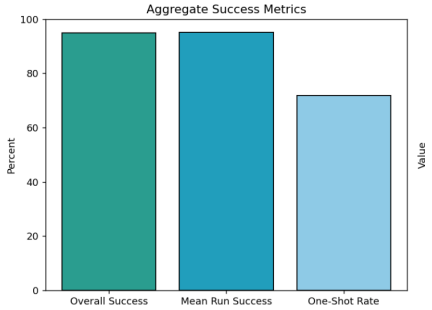


Figure 3.1.: Success rate (94.9%), mean per-run success (95.1%), and one-shot rate (71.8%).

Table 3.2 provides a per-run results. Runs 1, 3, and 5 achieved perfect success rates. In Runs 2 and 4, the single failed task (L-shape and square pattern) used all three retry attempts, showings that complex multi-object arrangements with tight geometric constraints remain the most challenging task type.

3.4 Detailed Run Analysis (Run 5)

Table 3.3 presents the per-task results from Run 5, which completed all 9 tasks with a 100% success rate.

Run	Tasks	Done	Fail	Rate	LLM Calls	Tokens	Duration (s)
1	6	6	0	100%	14	61K	640
2	7	6	1	85.7%	20	89K	1 129
3	7	7	0	100%	14	67K	724
4	10	9	1	90%	32	156K	2 216
5	9	9	0	100%	30	148K	1 519

Table 3.2.: Per-run summary. Longer runs (Runs 4 and 5) attempt more tasks and generate more complex structures, consuming more tokens due to retry-driven prompt expansion.

Task	Diff.	Att.	LLM	Lat. (s)	Exec. (s)
Pick & Place Cube	6	2	4	58.9	35.8
Place Cube in Box	6	1	2	26.3	20.2
Stack Two Cubes	6	3	6	81.9	111.2
Create Line Pattern	6	2	4	99.4	79.9
Build 3-Cube Pyramid	8	1	2	57.5	63.8
Sort Cubes by Color	8	2	4	58.2	132.0
Create L-Shape	6	1	2	24.7	62.7
Create Square Pattern	8	1	2	31.5	165.0
Create Staircase	7	2	4	36.5	222.1
Total / Session	–	15	30	474.9	892.8

Table 3.3.: Per-task results from Run 5. Five tasks succeeded on the first attempt, four required self-reflection and retry. Total session time was $\sim$25 minutes.

3.5 Analysis

Main Findings Table 3.4. The system prompt ($\sim$6 500 tokens) was iteratively refined over eight development sessions. Table 3.4 summarizes the key techniques discovered and their impact on task success. Three findings stand out. First, **negative examples outperform positive-only instructions**: despite explicit "reuse existing skills" directives, the LLM still re-implemented pick-and-place from scratch until a "BAD example" section was added showing what *not* to do. Second, **physics-aware verification is essential**: per-step verification passes but the final structure can be wrong because placing a new cube can knock a previously placed one, a mandatory final verification loop that re-reads all object positions solved this. Third, **prompt instructions alone are insufficient**: We had to explain to the LLM how the generated code should be (function should always start with run and have different parameters), and how generated skills could be used to create new ones. **We tried to keep our prompt general and not deterministic.**

Self-Reflection Enables Recovery. An iterative refinement loop enabled recovery from initial failures. In Run 5, stacking two cubes required three attempts: early

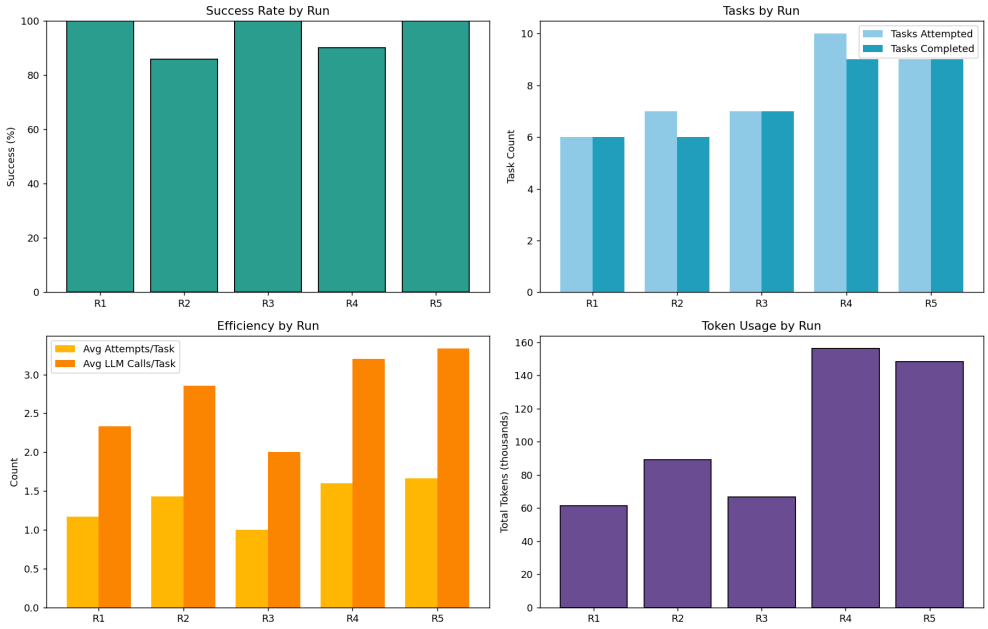


Figure 3.2.: Per-run breakdown of success rates, task counts, efficiency metrics, and token consumption. Runs 2 and 4 show lower success rates due to one failed task each. Token usage scales with run length and retry count.

trials had incorrect grasp, knocked other objects or placement heights, but critic feedback comparing pre- and post-execution object states guided the Planner to generate corrected code. Across all runs, a mean of 1.37 attempts per task indicates that most failures were resolved within the retry budget.

Skill Reuse Eliminates Redundant Code Generation. Once an easy task is learned, subsequent tasks reuse it A.4.

Critic Verification Accuracy. The critic correctly classified 37 successes across all runs, with confidence values ≥ 0.95. In Run 5's pick-and-place task (attempt 2), the critic verified that `cube_red` moved from $(0.500, 0.150, 0.025)$ to $(0.594, 0.004, 0.025)$a placement error of only $0.007\,\text{m}$ from the target. State-based comparison also detects unintended side effects (e.g., displacing a neighboring cube during placement) that return-value checks would miss.

Failure Modes. The two failures across 39 tasks share a common pattern: complex multi-object arrangements where the LLM repeatedly generates code with incorrect geometric computations (e.g., L-shape vertex positions or square corner coordinates) that the critic rejects. These failures exhaust the 3-attempt budget.

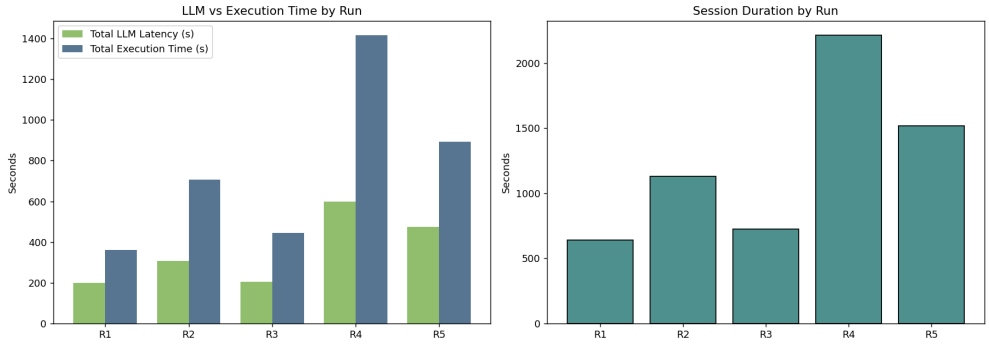


Figure 3.3.: Left: LLM latency vs. physics execution time by run. Execution time consistently dominates, confirming that task complexity (not LLM speed) drives session duration. Right: total session duration by run.

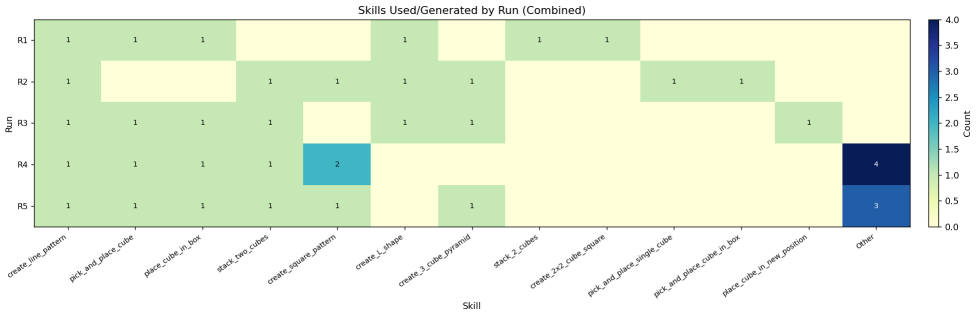


Figure 3.4.: Skills generated and reused across all five runs. Core skills, appear consistently, while the "Other" column aggregates task-specific wrapper skills. Run 4 shows the highest variety due to its longer session with more complex tasks.

Increasing the retry limit or providing geometric computation examples in the prompt could reduce this failure mode.

Prompt Engineering is Critical. The results depend on a carefully engineered system prompt ($\sim$6 500–8 500 prompt tokens per call) encoding domain knowledge: physics-based gripper behavior, position offset compensation ($\sim$0.05 m), tolerance guidelines, and composition patterns. This prompt was iteratively refined and is specific to this robot platform.

Technique	Problem Addressed	Impact
Tolerance calibration (`pos_tolerance` ≥ 0.06)	Robot's ~ 0.05 m position offset caused false `move_ee` failures	Eliminated most "failed to reach" errors
Lift retry + z verification	Cubes dropped during lift phase	Reduced stacking failures from $\sim 60\%$ to $\sim 10\%$
Collision-free placement rules ($\geq$ 0.07 m clearance)	Gripper knocked adjacent objects during placement	Prevented structure collapse
Object-aware	Place a cube on an unused cube	Ensure placement on unused space
Final structure verification loop	Physics displaced previously placed objects	Raised multi-object task success to $\sim 100\%$
Skill reuse rules	LLM re-implemented existing skills from scratch	Forced meaningful skill composition
Skill reuse in a new task with negative examples	reimplementing sub skills in the main skill	Ensure using skills (pyramid_base) in newly generated skill (pyramid)
Ordered curriculum progression	Agent proposed already-completed or regressed tasks	Ensured monotonic difficulty increase
Programmatic tolerance correction	LLM generated unrealistic success criteria (< 0.05 m)	Auto-corrected to feasible tolerances

Table 3.4.: Summary of prompt engineering techniques and their impact. Each row represents a failure mode discovered during development and the prompt-level fix that addressed it.

Conclusion

4

We introduced **Robo-Voyager**, adapting the Voyager paradigm (Wang et al., 2024) to physics-based manipulation in Isaac Sim. An LLM acts as curriculum proposer, code-generating planner, and critic verifier.

Our main finding is that **skill composition enables scalability**. After learning a verified primitive, later tasks are solved by composing parameterized calls rather than regenerating low-level manipulation code. This reduces complexity (e.g., $\sim 200 \rightarrow \sim 20$ lines for line arrangement) and lowers generation error rates. Vector-database retrieval helps surface relevant skills as the library grows.

4.1 Limitations

- **Simplified, state-access setup**: identical cubes, and ground-truth poses from the simulator (no vision).
- **Prompt dependence**: performance relies on a long, hand-crafted system prompt encoding robot and simulator specifics, reducing portability.

4.2 Future Work

- **Vision-language grounding**: replace ground-truth state with RGB-D perception (detection, pose estimation, tracking) to support real robots.
- **Vision-Language-Action models (VLA)**: introduce a curriculum that trains or fine-tunes a VLA policy to map RGB-D observations and language goals to action primitives, progressively expanding its skill repertoire and robustness.
- **Scaling and maintenance**: richer objects/tasks (articulated, deformable, tool use), multi-arm coordination, and continual skill re-verification/adaptation under environment changes.

Appendix A

A.1 Prompt Templates

This section summarizes the four system prompts used by Robo-Voyager. Full prompts total $\sim 8\,000$ tokens.

A.1.1 Skill Writer (Planner) Prompt

The Planner prompt ($\sim 6\,500$ tokens) instructs the LLM to generate executable Python skills. It contains:

1. **Robot API specification** – All available methods (`move_ee`, `open_gripper`, `close_gripper`, `get_object_position`, etc.) with exact signatures and return types.

2. **Robot characteristics** – The ~ 0.05 m position offset, physics-based gripper behavior (stops on contact), and required tolerance values (`pos_tolerance` ≥ 0.06).

3. **Skill composition rules** – Instructions to check `skills.list()` and reuse existing skills via `skills.call()` rather than reimplementing, includes good/bad examples.

4. **Scene description** – All objects (4 colored cubes, open-top box) and robot with initial positions, sizes, and creative manipulation possibilities.

5. **Safety constraints** – Collision-aware placement rules requiring ≥ 0.07 m clearance, free-space search fallback when targets are blocked.

6. **Multi-object verification** – Mandates a final verification loop re-reading all object positions after placement, since physics can displace previously placed objects.

A.1.2 Curriculum Proposer Prompt

The Curriculum prompt instructs the LLM to propose the next task based on the current skill library, environment state, and difficulty progression. Output is a structured JSON object with `name`, `description`, `success_criteria`, `difficulty` (1–10), and `required_skills`.

A.1.3 Critic (Verifier) Prompt

The Critic prompt (500 tokens) judges task success by comparing pre- and post-execution states. Key guidelines:

- Judge by *physical outcome*, not code return values , the code may return `False` due to overly strict self-checks while the task physically succeeded.
- Allow up to $0.05\,\mathrm{m}$ per-object error.
- For pyramids/towers, verify that the top cube is elevated ($z \geq 0.03\,\mathrm{m}$) above base cubes.
- Cite concrete numeric evidence from the post-execution state in reasoning.
- Output: `{success, reasoning, confidence}`.

A.1.4 Self-Reflection (Debugger) Prompt

The Self-Reflection prompt (~ 150 tokens) is invoked when a skill fails. Given the failed code, error message, and robot state, it outputs an `analysis`, `root_cause`, list of `suggested_fixes`, and a `should_retry` flag. This analysis is appended to the Planner's context on the next retry attempt, enabling targeted corrections rather than blind regeneration.

A.2 Task in Isaacsim

A.3 Generated Skill Examples

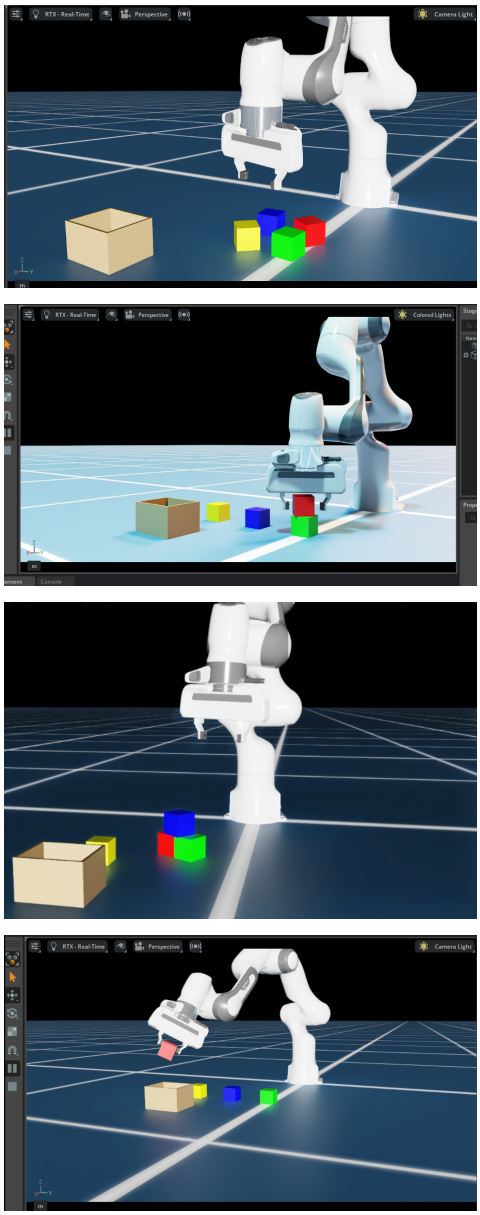
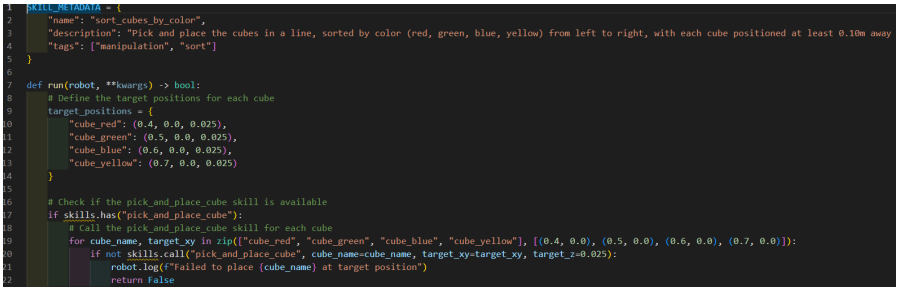


Figure A.1.: Task in IsaacsimÖ Squere, Stacking, Pyramid, lifting

```python
SKILL_METADATA = {
    "name": "pick_and_place_cube",
    "description": "Pick up a cube and place it at a target position",
    "tags": ["manipulation", "pick_and_place"]
}

def run(robot, **kwargs) -> bool:
    cube_name = kwargs.get("cube_name", "cube_red")
    target_xy = kwargs.get("target_xy", (0.60, 0.00))
    target_z = kwargs.get("target_z", 0.02)

    # Get cube position
    cube = robot.get_object_position(cube_name)
    robot.log(f"Cube {cube_name} at {cube}")

    # Approach from above
    above = (cube[0], cube[1], cube[2] + 0.10)
    if not robot.move_ee(above, pos_tolerance=0.06, timeout_s=15.0):
        robot.log("Failed to reach above cube")
        return False

    robot.open_gripper(width=0.08, steps=120)

    # Descend to grasp
    grasp = (cube[0], cube[1], cube[2] + 0.02)
    if not robot.move_ee(grasp, pos_tolerance=0.06, timeout_s=15.0):
        robot.log("Failed to descend to cube")
        return False

    robot.close_gripper(steps=200)
    robot.wait(30)

    # Lift
    lift = (cube[0], cube[1], 0.20)
    lift_ok = robot.move_ee(lift, pos_tolerance=0.03, timeout_s=15.0)
    if not lift_ok:
        robot.log("Lift timeout, retrying lift to z=0.22")
        robot.move_ee((cube[0], cube[1], 0.22), pos_tolerance=0.03, timeout_s=10.0)

    # Check actual cube position
    new_pos = robot.get_object_position(cube_name)
    robot.log(f"Cube {cube_name} now at {new_pos}")
    dz = new_pos[2] - cube[2]
    robot.log(f"Lift delta z: {dz:.3f}m")
    if dz < 0.04:
        robot.log("Failed to lift cube")
        return False

    # Move to target
    above_target = (target_xy[0], target_xy[1], 0.15)
    robot.move_ee(above_target, pos_tolerance=0.06, timeout_s=15.0)

    # Place
    place_pos = (target_xy[0], target_xy[1], target_z)
    robot.move_ee(place_pos, pos_tolerance=0.06, timeout_s=15.0)
    robot.open_gripper(width=0.08, steps=120)
    robot.wait(30)

    # Move up to clear
    robot.move_ee(above_target, pos_tolerance=0.06, timeout_s=10.0)

    # Verify cube is at target
    final_pos = robot.get_object_position(cube_name)
    robot.log(f"Cube {cube_name} final position: {final_pos}")
    dist = ((final_pos[0] - target_xy[0])**2 + (final_pos[1] - target_xy[1])**2)**0.5
    robot.log(f"Distance from target: {dist:.3f}m")
    return dist < 0.08
```

Figure A.2.: LLM-generated code for picl and place.



```python
SKILL_METADATA = {
    "name": "sort_cubes_by_color",
    "description": "Pick and place the cubes in a line, sorted by color (red, green, blue, yellow) from left to right, with each cube positioned at least 0.10m away
    "tags": ["manipulation", "sort"]
}

def run(robot, **kwargs) -> bool:
    # Define the target positions for each cube
    target_positions = {
        "cube_red": (0.4, 0.0, 0.025),
        "cube_green": (0.5, 0.0, 0.025),
        "cube_blue": (0.6, 0.0, 0.025),
        "cube_yellow": (0.7, 0.0, 0.025)
    }

    # Check if the pick_and_place_cube skill is available
    if skills.has("pick_and_place_cube"):
        # Call the pick_and_place_cube skill for each cube
        for cube_name, target_xy in zip(["cube_red", "cube_green", "cube_blue", "cube_yellow"], [(0.4, 0.0), (0.5, 0.0), (0.6, 0.0), (0.7, 0.0)]):
            if not skills.call("pick_and_place_cube", cube_name=cube_name, target_xy=target_xy, target_z=0.025):
                robot.log(f"Failed to place {cube_name} at target position")
                return False
```

Figure A.3.: Sort Cubes

```python
SKILL_METADATA = {
    "name": "create_pyramid_base",
    "description": "Pick two cubes and place them side by side to form the base of a pyramid, with a gap of 0.05m between them.",
    "tags": ["pyramid", "base", "cubes"]
}

def run(robot, **kwargs) -> bool:
    # Set default cube names and target position
    cube1_name = kwargs.get("cube1_name", "cube_red")
    cube2_name = kwargs.get("cube2_name", "cube_green")
    target_xy = kwargs.get("target_xy", (0.4, 0.0))

    # Get current cube positions
    cube1_pos = robot.get_object_position(cube1_name)
    cube2_pos = robot.get_object_position(cube2_name)

    # Calculate target positions for the base cubes
    target1_xy = (target_xy[0] - 0.025, target_xy[1])
    target2_xy = (target_xy[0] + 0.025, target_xy[1])

    # Place the first cube
    if not skills.call("pick_and_place_cube", cube_name=cube1_name, target_xy=target1_xy, target_z=0.02):
        return False

    # Place the second cube
    if not skills.call("pick_and_place_cube", cube_name=cube2_name, target_xy=target2_xy, target_z=0.02):
        return False

    # Verify the base cubes are in position
    final_pos1 = robot.get_object_position(cube1_name)
    final_pos2 = robot.get_object_position(cube2_name)
    dist1 = ((final_pos1[0] - target1_xy[0])**2 + (final_pos1[1] - target1_xy[1])**2)**0.5
    dist2 = ((final_pos2[0] - target2_xy[0])**2 + (final_pos2[1] - target2_xy[1])**2)**0.5
    gap = ((final_pos1[0] - final_pos2[0])**2 + (final_pos1[1] - final_pos2[1])**2)**0.5
    robot.log(f"Cube1 distance from target: {dist1:.3f}m")
    robot.log(f"Cube2 distance from target: {dist2:.3f}m")
    robot.log(f"Gap between cubes: {gap:.3f}m")
    return dist1 < 0.08 and dist2 < 0.08 and abs(gap - 0.05) < 0.01
```

```python
SKILL_METADATA = {
    "name": "create_full_pyramid",
    "description": "Pick and place the cubes to form a full pyramid structure with a base of two cubes and one cube on top, elevated.",
    "tags": ["pyramid", "manipulation"]
}

def run(robot, **kwargs) -> bool:
    # Check if create_pyramid_base skill exists
    if skills.has("create_pyramid_base"):
        # Call create_pyramid_base skill to place the base cubes
        base_xy = (0.5, 0.0)  # Target base position
        if not skills.call("create_pyramid_base", cube1_name="cube_red", cube2_name="cube_green", target_xy=base_xy):
            return False

        # Get the actual positions of the base cubes
        base_cube1_pos = robot.get_object_position("cube_red")
        base_cube2_pos = robot.get_object_position("cube_green")

        # Calculate the target position for the top cube
        top_xy = ((base_cube1_pos[0] + base_cube2_pos[0]) / 2, (base_cube1_pos[1] + base_cube2_pos[1]) / 2)
        top_z = max(base_cube1_pos[2], base_cube2_pos[2]) + 0.05  # Elevated by at least 0.05m

        # Call pick_and_place_cube skill to place the top cube
        if skills.has("pick_and_place_cube"):
            if not skills.call("pick_and_place_cube", cube_name="cube_blue", target_xy=top_xy, target_z=top_z):
                return False
```

Figure A.4.: LLM-generated code for pyramid construction. Top: base layer placement (`build_pyramid_base`). Bottom: full pyramid assembly (`build_3_cube_pyramid`), which composes the base skill with a stacking call.

Bibliography

Michael Ahn, Anthony Brohan, Noah Brown, Yevgen Chebotar, Omar Cortes, Byron David, Chelsea Finn, Chuyuan Fu, Keerthana Gopalakrishnan, Karol Hausman, et al. (2022). "Do as i can, not as i say: Grounding language in robotic affordances". In.

Sami Haddadin, Sven Parusel, Lars Johannsmeier, Saskia Golz, Simon Gabl, Florian Walch, Mohamadreza Sabaghian, Christoph Jähne, Lukas Hausperger, and Simon Haddadin (2022). "The franka emika robot: A reference platform for robotics research and education". In: *IEEE robotics & automation magazine* 29.2.

Jacky Liang, Wenlong Huang, Fei Xia, Peng Xu, Karol Hausman, Brian Ichter, Pete Florence, and Andy Zeng (2023). "Code as policies: Language model programs for embodied control". In: *arXiv preprint arXiv:2209.07753*.

NVIDIA (2024). *Isaac Sim – Robotics Simulation Platform*. `https://developer.nvidia.com/isaac-sim`.

Qdrant (2024). *Qdrant – Vector Similarity Search Engine*. `https://qdrant.tech`.

Nathan D Ratliff, Karl Van Wyk, Mandy Kleinke, and Natalia Gonzalez (2018). "Riemannian Motion Policies". In: *arXiv preprint arXiv:1801.02854*.

Nils Reimers and Iryna Gurevych (2019). "Sentence-bert: Sentence embeddings using siamese bert-networks". In: *Proceedings of the 2019 conference on empirical methods in natural language processing and the 9th international joint conference on natural language processing (EMNLP-IJCNLP)*.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. (2023). "Llama: Open and efficient foundation language models. arXiv 2023". In: *arXiv preprint arXiv:2302.13971* 10.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar (2024). "Voyager: An Open-Ended Embodied Agent with Large Language Models". In: *arXiv preprint arXiv:2305.16291*.

Colophon

This thesis was typeset with LaTeX 2_ε. It uses the *Clean Thesis* style developed by Ricardo Langner.

Download the *Clean Thesis* style at `http://cleanthesis.der-ric.de/`.