

1 A Details of the NESYRO Framework

2 NESYRO enables exploration through a recursive composition of Neuro-symbolic Code Verification
 3 and Neuro-symbolic Code Validation. A policy code produced by the LLM is first subjected to explicit
 4 symbolic verification, which statically checks logical consistency against the task specification. The
 5 verified policy code then enters interactive validation, where each skill is evaluated in neuro-symbolic
 6 manner; if an unmet precondition is detected, the system synthesizes exploratory safe probe policy
 7 code to gather the missing observations. Every probe is fed back through the same verification and
 8 validation (V&V) cycle, producing a policy tree whose nodes are recursively grounded until all skills
 9 achieve a satisfactory Neuro-symbolic Confidence Score. This recursive V&V framework guarantees
 10 that the policy code is both executable and environmentally grounded, even under dynamic, partially
 11 observable conditions.

12 A.1 NESYRO Algorithm

13 We provide the full pseudocode for our neuro-symbolic task execution pipeline in Algorithm 1 and
 14 Algorithm 2. Below, we briefly describe the functional roles of each procedure and its interaction
 15 within the recursive planning framework.

Algorithm 1 Task Execution Pipeline

Agent:

env — environment interface \mathcal{D} — domain knowledge
 $\mathcal{E}_{\text{demo}}$ — demonstration set ϵ — confidence threshold
 a — primitive action $o_{\leq t}$ — observation history up to t
 \mathcal{A} — action set

Returns:

τ — executed trajectory π_{main} — policy code

procedure RUNTASK(env, \mathcal{D} , $\mathcal{E}_{\text{demo}}$, ϵ)

$(o_{\leq 0}, g) \leftarrow \text{env.reset}()$
 $\tau \leftarrow ()$ ▷ Initialize empty trajectory
 $(\pi_{\text{main}}, o_{\leq t}, \text{env}, \tau) \leftarrow \text{NESYRO}(g, o_{\leq 0}, \mathcal{D}, \mathcal{E}_{\text{demo}}, \epsilon, 0, \text{env}, \tau)$
 $(\tau_{\text{exe}}, o_{\leq t+1}, \text{env}) \leftarrow \text{EXE}(\pi_{\text{main}}, o_{\leq t}, \text{env})$
 $\tau \leftarrow \tau \cup \tau_{\text{exe}}$
return $(\tau, o_{\leq t+1}, \pi_{\text{main}})$

end procedure

procedure EXE(π , $o_{\leq t}$, env)

$\tau_{\text{exe}} \leftarrow []$
for each f **in** π **do**
 $\mathcal{A} \leftarrow \text{EXPANDSKILL}(f, o_{\leq t})$
 for each a **in** \mathcal{A} **do**
 $o_{\text{next}} \leftarrow \text{env.step}(a)$
 $\tau_{\text{exe}} \leftarrow \tau_{\text{exe}} \cup (a, o_{\text{next}})$
 $o_{\leq t} \leftarrow o_{\leq t} \cup o_{\text{next}}$
 end for

end for

return $(\tau_{\text{exe}}, o_{\leq t}, \text{env})$

end procedure

16 **RunTask and execution loop.** The RUNTASK procedure (Algorithm 1) initializes the environment
 17 and launches the neuro-symbolic reasoning process. It first resets the environment to obtain the
 18 initial observation $o_{\leq 0}$ and instruction g , and initializes an empty trajectory τ . The main grounding
 19 routine NESYRO is then called to synthesize a grounded policy code π_{main} based on the instruction
 20 and current context. Once obtained, the policy is executed via EXE, which expands symbolic skills
 21 into primitive actions and steps through the environment. The complete trajectory τ and updated
 22 observation history $o_{\leq t}$ are returned. The EXE procedure handles the execution of symbolic skills. For

23 each skill f in π , it calls EXPANDSKILL to retrieve the sequence of corresponding low-level actions.
 24 These are executed sequentially in the environment, and the resulting observations are appended to
 25 both the trajectory and the observation history.

Algorithm 2 Recursive Neuro-symbolic Verification & Validation

```

procedure NESYRO( $g, o_{\leq t}, \mathcal{D}, \mathcal{E}_{\text{demo}}, \epsilon, k, \text{env}, \tau$ )
  ( $\mathcal{T}_{\text{spec}}, \pi_{\text{main}}$ )  $\leftarrow$  NEURO_SYMBOLIC_VERIFICATION( $g, o_{\leq t}, \mathcal{D}, k$ )
  ( $\pi_{\text{main}}, o_{\leq t}, \text{env}, \tau$ )  $\leftarrow$ 
    NEURO_SYMBOLIC_VALIDATION( $g, o_{\leq t}, \mathcal{D}, \mathcal{E}_{\text{demo}}, \pi_{\text{main}}, \epsilon, k, \text{env}, \tau$ )
  return ( $\pi_{\text{main}}, o_{\leq t}, \text{env}, \tau$ )  $\triangleright$  grounding policy code  $\pi_{\text{main}}$ 
end procedure

procedure NEURO_SYMBOLIC_VERIFICATION( $g, o_{\leq t}, \mathcal{D}, k$ )
  ( $\mathcal{T}_{\text{spec}}, \pi_{\text{main}}$ )  $\leftarrow$   $\Phi_{\text{veri}}(o_{\leq t}, g, l_{\text{cot}}, \mathcal{D}, k)$ 
  while  $\Psi_{\text{veri}}(\mathcal{T}_{\text{spec}}, \pi_{\text{main}}) = \text{fail}$  do
     $\mathcal{F}_{\text{veri}} \leftarrow \Psi_{\text{veri}}(\mathcal{T}_{\text{spec}}, \pi_{\text{main}})$ 
    ( $\mathcal{T}_{\text{spec}}, \pi_{\text{main}}$ )  $\leftarrow$   $\Phi_{\text{veri}}(o_{\leq t}, g, l_{\text{cot}}, \mathcal{D}, \pi_{\text{main}}, \mathcal{F}_{\text{veri}}, k)$ 
  end while
  return ( $\mathcal{T}_{\text{spec}}, \pi_{\text{main}}$ )
end procedure

procedure NEURO_SYMBOLIC_VALIDATION( $g, o_{\leq t}, \mathcal{D}, \mathcal{E}_{\text{demo}}, \pi_{\text{main}}, \epsilon, k, \text{env}, \tau$ )
   $n \leftarrow k$ 
  while  $n < |\pi_{\text{main}}|$  do
     $f_n \leftarrow \pi_{\text{main}}[n]$ 
     $\text{CSC} \leftarrow \Phi_{\text{vali}}(\mathcal{D}, \mathcal{E}_{\text{demo}}, o_{\leq t}, g, f_n)$ 
     $\text{LC} \leftarrow \Psi_{\text{vali}}(\mathcal{D}, o_{\leq t}, g, f_n)$ 
     $\text{NeSyConf} \leftarrow \text{CSC} \times \text{LC}$ 
    if  $\text{NeSyConf} < \epsilon$  then
       $g_{\text{probe}} \leftarrow \text{MAKEPROBEGOAL}(f_n, \mathcal{F}_{\text{csc}}, \mathcal{F}_{\text{lc}})$ 
      ( $\pi_{\text{probe}}, o_{\leq t}, \text{env}, \tau$ )  $\leftarrow$  NESYRO( $g_{\text{probe}}, o_{\leq t}, \mathcal{D}, \mathcal{E}_{\text{demo}}, \epsilon, 0, \text{env}, \tau$ )  $\triangleright$  recursive
      ( $\tau_{\text{exe}}, o_{\leq t+1}, \text{env}$ )  $\leftarrow$  EXE( $\pi_{\text{probe}}, o_{\leq t}, \text{env}$ )
       $\tau \leftarrow \tau \cup \tau_{\text{exe}}$ 
      ( $\mathcal{T}_{\text{spec}}, \pi_{\text{main}}$ )  $\leftarrow$  NEURO_SYMBOLIC_VERIFICATION( $g, o_{\leq t+1}, \mathcal{D}, n$ )
    else
       $n \leftarrow n + 1$ 
    end if
  end while
  return ( $\pi_{\text{main}}, o_{\leq t+\alpha}, \text{env}, \tau$ )  $\triangleright$   $\alpha$ : number of recursive  $\pi_{\text{probe}}$  executed during validation.
end procedure

```

26 **Recursive neuro-symbolic reasoning.** Algorithm 2 outlines the recursive grounding logic of
 27 NESYRO. The NESYRO procedure first invokes NEURO_SYMBOLIC_VERIFICATION to obtain a
 28 symbolic task specification $\mathcal{T}_{\text{spec}}$ and initial policy code π_{main} . Logical correctness is ensured through
 29 iterative verification using Φ_{veri} and Ψ_{veri} , which checks whether π_{main} satisfies $\mathcal{T}_{\text{spec}}$. After verification,
 30 the policy is passed to NEURO_SYMBOLIC_VALIDATION for skill-wise confidence assessment. For
 31 each skill f_n , the framework computes neuro-symbolic confidence score (NeSyConf). If NeSyConf
 32 falls below threshold ϵ , a probing goal g_{probe} is generated and recursively passed into NESYRO.

33 To construct this probing goal g_{probe} , the MAKEPROBEGOAL function synthesizes a new instruction
 34 that addresses the failure feedback. Specifically, it leverages NeSyConf feedback ($\mathcal{F}_{\text{csc}}, \mathcal{F}_{\text{lc}}$) to
 35 identify missing observations. This recursive routine allows a skill that fails to exceed the confidence
 36 threshold ϵ to be refined and validated using updated observations gathered from safe probe executions.
 37 Once all skills pass validation, the final grounded policy and accumulated trajectory are returned.

38 B Environment Settings

39 B.1 RL Bench

40 We use RL Bench [1] as the simulation environment for our experiments. RL Bench offers a wide
 41 range of tabletop manipulation tasks and provides realistic simulations of both robot control and
 42 visual observations. All experiments are performed using a 7-DoF Franka Emika Panda robotic arm,
 43 which is supported natively by RL Bench. The environment is particularly suitable for evaluating
 44 planning and interaction under partial observability, as it supports randomized object configurations
 45 and sensor data, including RGB, depth, and segmentation masks. Its compatibility with Python also
 46 allows straightforward integration with our code generation and execution framework.

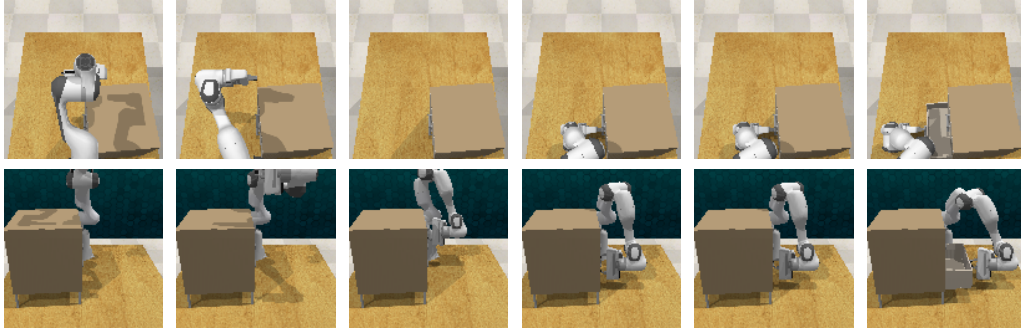


Figure 1: Example scenes illustrating the "open drawer" task in the RL Bench. The top row shows the overhead view, and the bottom row shows the front view.

47 **Object configuration.** Each episode initializes a workspace containing seven unique objects placed
 48 on a table. The objects include two tomatoes, a piece of trash, a bin, a three-level drawer, a desk
 49 lamp, and a switch for the desk lamp. The position of each object is randomized in every episode,
 50 introducing perceptual variability and scene diversity across tasks. Figure 1 shows an example scene
 51 from the RL Bench environment used in our experiments.

52 **Task composition.** We categorize the tasks into four types to enable structured evaluation, as
 53 summarized in Table 1. Each task type contains multiple language instructions, with associated probe
 54 targets indicating the source of uncertainty that must be resolved during execution.

Table 1: Task types, example of instructions, and associated probe targets in RL Bench.

Task Type	Example Instructions	Probe Target
Object Relocation	Move two tomatoes onto plate Put the trash into bin	Tomato identity Trash identity
Object Interaction	Open a drawer Open two drawers	Drawer locked/unlocked state Drawer locked/unlocked state
Auxiliary Manipulation	Move two tomatoes onto plate in dark room Open drawer in dark room	Missing visual observation (requires light activation) Missing visual observation (requires light activation)
Long-Horizon Tasks	Move a die into the drawer Move dice into the drawer	Die identity, Drawer locked/unlocked state Dice identity, Drawer locked/unlocked state

55 B.2 Real-world

56 **Environment setup.** We conducted our real-world experiments using a 7-DoF Franka Emika
 57 Research 3 robotic arm mounted on a tabletop workspace. An Intel RealSense D435 RGB-D camera
 58 was positioned above the table to provide top-down RGB and depth information. This input was
 59 processed by an object detection module to identify the categories and bounding boxes of task-relevant
 60 objects. Depth measurements were used to compute 3D coordinates, which were then transformed
 61 into the robot's coordinate frame. This setup enabled accurate object localization and real-time
 62 observation grounding, providing the necessary perception for reliable execution.

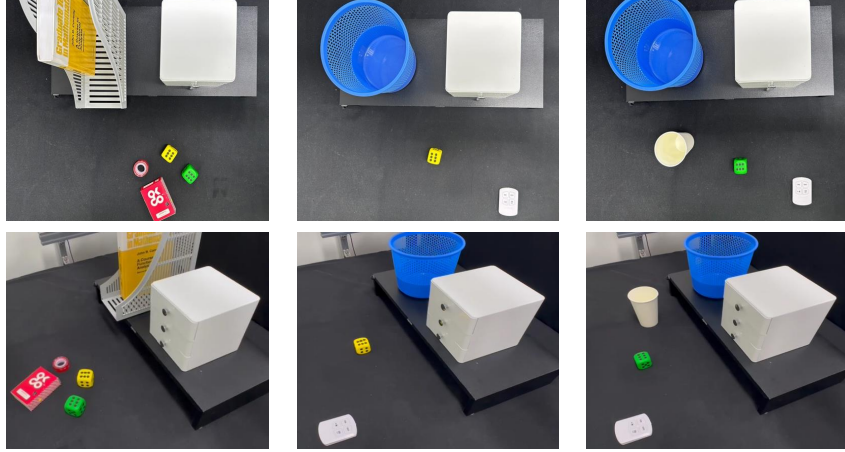


Figure 2: Example scenes from the real-world environment used in our experiments.

Object configuration. The real-world environment contains ten unique objects arranged on a tabletop workspace. These include two dice, two pieces of trash, a bin, a three-level drawer, a small cardboard box, a paper cup filled with liquid, a roll of tape, and a light switch. The initial positions of all objects are randomized for each trial, introducing diverse spatial configurations and observation conditions across task instances. This variability supports evaluation under partial observability and enables direct comparison with the RL Bench-based simulation setup. Figure 2 shows a representative setup of the real-world environment used in our experiments.

Task composition. We maintain the RL Bench task categorization in the real-world setup to ensure consistency and enable direct comparison. Each of the four task types corresponds to a distinct source of uncertainty and is associated with multiple language instructions and probe targets, as summarized in Table 2.

Table 2: Task types, example of instructions, and associated probe targets in real-world.

Task Type	Example Instructions	Probe Target
Object Relocation	Place a die into drawer Move dice into drawer	Die identity Dice identity
Object Interaction	Open a drawer Open two drawers	Drawer locked/unlocked state Drawer locked/unlocked state
Auxiliary Manipulation	Place a die into drawer in dark room Open drawer in dark room	Missing visual observation (requires light activation) Missing visual observation (requires light activation)
Long-Horizon Tasks	Place a die into drawer Move a die into drawer	Die identity, Drawer locked/unlocked state Die identity, Drawer locked/unlocked, empty/occupied state

Low-Level Control. For motion planning and control in the real-world environment, we employed MoveIt [2], an open-source motion planning framework widely used for robotic manipulation. Once the target object positions were obtained from the perception pipeline, we invoked parameterized skill primitives such as `pick`, `place`, and `open`, which are designed to operate over arbitrary object poses. Each skill was instantiated using the transformed 3D coordinates of the corresponding object and passed to the planner as goal constraints. Trajectory optimization was handled by MoveIt’s built-in planners, which computed collision-free joint-space paths that respect the robot’s kinematic limits and workspace constraints. The resulting trajectories were executed using the robot’s internal controller through ROS. Although continuous force control was not used for the gripper, we implemented discrete grasping strategies based on object geometry and semantic role (e.g., trash, dice). This ensured consistent and safe execution across a variety of physical configurations.

Unlike the RL Bench setting, where code execution is simulated through parameterized low-level APIs, our real-world system closes the loop by grounding skill calls with real sensor observations and executing planned trajectories on physical hardware. This setup allows us to evaluate the reliability of the proposed planning framework under real-world uncertainties.

89 C Experiment Details

90 C.1 Compute Resources

91 Most experiments were conducted on a local machine with an Intel(R) Core(TM) i7-9700KF CPU
92 and an NVIDIA GeForce RTX 4080 GPU (16GB VRAM). Each task instance used a single GPU,
93 and RLBench simulation was executed with up to 32GB of system memory. Symbolic verification
94 and PDDL planning were run on the CPU. For experiments using the larger language models listed in
95 Table 6 in main paper, such as Llama-3.1-8B and Qwen3-30B-A3B, we used a cloud-based CUDA
96 cluster with GPUs equipped with approximately 82GB of VRAM. All OpenAI models, including
97 GPT-4o and GPT-4.1, were accessed via the OpenAI API.

98 C.2 NESYRO Implementation

99 **LLM usage overview.** Our framework utilizes LLM as core reasoning engines in three tightly
100 integrated components of the code generation and validation pipeline:

- 101 • **Code Generation (Φ_{veri}):** Given the instruction g and the current observation history $o_{\leq t}$,
102 the verification LLM Φ_{veri} performs chain-of-thought reasoning to produce an intermediate
103 symbolic task specification $\mathcal{T}_{\text{spec}}$ and corresponding Python policy code π_{main} . If the code
104 fails symbolic verification via Ψ_{veri} , structured feedback $\mathcal{F}_{\text{veri}}$ is returned and used by the
105 LLM to iteratively revise the code. Only the unvalidated portion of π_{main} is regenerated at
106 each step, preserving previously verified components.
- 107 • **CSC Computation (Φ_{vali}):** For each skill f_n in the primary policy π_{main} , the validation LLM
108 Φ_{vali} computes CSC_{f_n} that estimates the likelihood of successful execution under the current
109 observation $o_{\leq t}$ and instruction g . The LLM is prompted with the code for f_n , domain
110 knowledge \mathcal{D} , retrieved demonstrations $\mathcal{E}_{\text{demo}}$, and task context. Token-level probabilities are
111 aggregated and transformed into a negative log-likelihood score. This value is normalized to
112 produce a scalar confidence score CSC_{f_n} used for validation. Before normalization, CSC_{f_n}
113 ranges over $[0, \infty)$; after normalization, it is scaled to the interval $[0, 1]$.
- 114 • **CSC Feedback Generation (\mathcal{F}_{csc}):** If $\text{NeSyConf}_{f_n} < \epsilon$, the skill f_n is considered to require
115 a safe probe. In such cases, CSC feedback \mathcal{F}_{csc} is constructed based on f_n , the failure context,
116 current observation $o_{\leq t}$, instruction g , and demonstrations $\mathcal{E}_{\text{demo}}$. This feedback is then used
117 to prompt the LLM to generate a safe probe policy code π_{probe} . The resulting policy code is
118 recursively verified and validated through the NESYRO pipeline before execution.

119 **Example of prompt.** Below are the representative prompts used in each stage of our framework:
120 generating executable robot code, computing CSC for each skill, and generating feedback when the
121 NeSyConf falls below a threshold.

Code Generator Prompt

Role: You are an AI assistant tasked with generating executable Python code that controls a robot in a simulated environment.

Task: Complete the executable code using the provided inputs and by implementing any missing skills. The goal is to ensure the robot can achieve the specified objective by executing a sequence of actions (plan).

Input Details:

- **Domain PDDL:** Describes the available actions and predicates in the environment. It includes information about action parameters, preconditions, and effects. This provides the symbolic action space for the planner.
Provided domain PDDL: `{{domain_pddl}}`
- **Observation (Initial State Description):** Represents the initial state of the environment in PDDL format, including locations of objects, robot position, and other relevant state descriptions.
Provided observation: `{{observation}}`

- **Goal (Natural Language Description):** The goal specifies what the robot must accomplish in plain language.
Provided goal: `{{goal}}`
- **Specification (Code Generation Guidelines):** Provides strict rules and constraints that the generated code must follow.
Provided specification: `{{spec}}`
- **Skill Code (Python Implementations of Actions):** A set of predefined Python functions that implement low-level skills (e.g., move, pick, place).
Provided skill code: `{{skill_code}}`
- **Executable Code Skeleton:** A partially completed Python file containing environment setup and control flow scaffolding.
Provided skeleton: `{{skeleton_code}}`
- **Available Skill Names:** A list of all valid skill function names that may be called in the generated code.
Provided skills: `{{available_skills}}`
- **Object List:** A list of object names that exist in the environment.
Provided object list: `{{object_list_position}}`
- **Feedback:** Corrections or issues identified from the previous code generation.
Feedback: `{{feedback}}`
Previous code: `{{prev_code}}`
- **Exploration Knowledge:** Optional knowledge to infer or explore missing observations.
Exploration knowledge: `{{exploration_knowledge}}`
- **Frozen Code:** Code that must remain unchanged. New code must follow this segment.
[Frozen Code Start] `{{frozen_code_part}}` [Frozen Code End]

Implementation Requirements:

- Use only the predefined skills (e.g., move, pick, place) from `skill_code`; do not define new functions.
- Complete the provided skeleton by inserting plan logic that achieves the specified goal.
- Preserve all existing imports and [Frozen Code] segments.
- Output should be plain text only — do not use code blocks.
- Handle errors gracefully during skill execution (e.g., invalid arguments or missing objects).
- Use the following external modules as provided:
 - `env`: Environment setup and shutdown.
 - `skill_code`: Contains all callable action implementations.
 - `video`: Tools for simulation recording.
 - `object_positions`: For retrieving object location information.

CSC Computation Prompt

Role: You are an AI assistant responsible for estimating the likelihood that a specific robotic skill will succeed in the current environment. Your evaluation will be used to compute a token-level log-probability-based common sense confidence, rather than to generate output.

Input Details:

- **Instruction:**
{instruction}
- **Demonstrations:**
Demo 1: {demo_1}
Demo 2: {demo_2}
...
- **Observation:**
{observation}
- **Object List:**
{object_list}
- **Skill Code:**
{skill_code}

124

CSC Feedback Prompt

Role: You are an AI assistant responsible for analyzing why a robotic skill code is likely to fail and for generating feedback to guide its refinement.

Task: Based on the given context and the Neuro-Symbolic Confidence Score (NeSyConf) being below threshold, provide structured feedback to help revise or improve the given skill code.

Input Details:

- **Demonstrations:**
Demo 1: {demo_1}
Demo 2: {demo_2}
...
- **Observation:**
{observation}
- **Skill Code:**
{skill_code}
- **Confidence Score**
Current Confidence Score: {NeSyConf}, Threshold: {threshold}
- **Instruction:**
{instruction}
- **Object List:**
{object_list}

Format:

- 1. Problem Identification
- 2. Justification (Why is it a problem?)
- 3. Proposed Solutions (High-level ideas)
- 4. Additional Notes (Optional)

Instruction to Model: Focus on issues such as force calibration, missing objects, logical flaws, or unsafe execution. For example: “An object declared in the code is not in the actual object list.” Your feedback will guide the regeneration of this skill step.

125

Hyperparameter setting. The only hyperparameter in our framework is the confidence threshold ϵ used during neuro-symbolic validation. For each skill, we perform five safe exploration probes under varied initial conditions to estimate its execution confidence. To determine a suitable value of ϵ for a given environment, we exclude outlier trials in which the probe failed due to non-informative reasons, which could otherwise deflate confidence estimates. This ensures that ϵ reflects a realistic and actionable lower bound of confidence for successfully grounded skills.

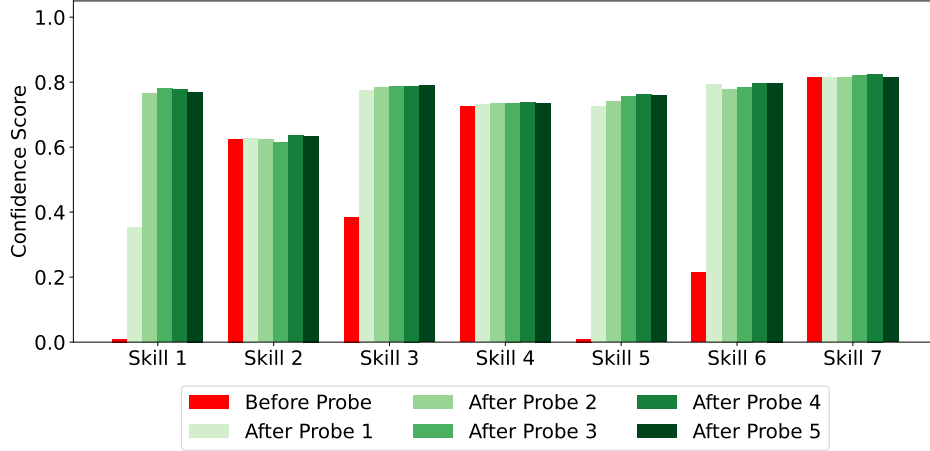


Figure 3: Confidence scores over five safe probes for each of 7 skills in a long-horizon task.

Figure 3 illustrates this process using one of the long-horizon tasks in the RLBenCh environment. Each of the 7 skills was probed five times, and confidence scores were recorded before and after each probe. The figure shows that while confidence increases with repeated probing, it typically saturates after a few trials, indicating convergence. Based on this observation, we compute the final confidence distribution by averaging only those probe outcomes that resulted in a successful skill grounding. We then set ϵ to the lower quartile of this filtered distribution, ensuring a conservative yet robust threshold that filters out unreliable executions while accepting skills with moderately confident grounding.

Demonstrations format. To support symbolic validation and LLM-based reasoning, we constructed a general-purpose demonstration library consisting of synthetic examples. These demonstrations were generated entirely via a large language model (GPT-4o) [3], given domain-level PDDL definitions and representative symbolic contexts, without requiring environment-specific execution or human annotation. Each example encodes typical task-relevant transitions across common household activities (e.g., opening a drawer, placing an object, turning on a light), and captures both successful and failure cases under varied symbolic states. In total, we synthesized approximately **500** such demonstrations spanning over **15 diverse skill types**. These examples are reused across all tasks to provide reusable prior knowledge for CSC computation and to guide safe probing decisions when symbolic grounding confidence is low.

Demonstration Structure

Each demonstration consists of a sequence of transitions. Each transition is represented as a dictionary with the following fields:

- `initial_observation`: symbolic or structured observation before executing the action.
- `action`: the skill function executed, such as a call to `pick(object)` or `open(drawer)`.
- `post_observation`: symbolic or structured observation after executing the action.
- `success`: boolean value indicating whether the action was successful.

150 C.3 Baselines Implementation

151 **Code as Policies (CaP)** [4] serves as the foundation of our framework and is implemented by invoking
152 the Code Generator Prompt with the frozen code length set to zero. Although a symbolic specification
153 is also produced, this baseline does not include any verification process.

154 **CaP w/ Lemur** [5] extends CaP by performing verification over the generated specification. This
155 process is conducted in the exact same manner as the Neuro-symbolic verification phase.

156 **CaP w/ CodeSift** [6] extends CaP by incorporating LLM-based verification and validation. In the
157 verification stage, CodeSift performs static syntax checks using language-specific tools (pylint
158 for Python, shellcheck for Bash) and prompts the LLM to summarize the code’s functionality.
159 This summary is then used in the validation stage to assess semantic alignment with the original
160 task instruction. The validation consists of multiple sub-steps: semantic similarity scoring, listing
161 all functional mismatches, and determining whether the implementation is exact. If the code fails
162 validation, the framework automatically generates refinement feedback and prompts the LLM to
163 revise the code accordingly. The entire process is orchestrated via a modular pipeline that yields
164 detailed diagnostic outputs and a refined version of the code when necessary.

165 **LLM-Planner** [7] follows the same initial procedure as CaP by generating code from the instruction
166 using a Code Generator Prompt. During execution in the environment, if an action fails, the planner
167 captures the current observation and provides it as additional context to the LLM. The previously
168 executed portion of the code is marked as frozen, and a new code segment is generated to continue
169 the task from the failure point. As in CaP, a symbolic specification is produced, but no verification or
170 validation is performed throughout the process.

171 **AutoGen** [8] adopts the same iterative replanning strategy as LLM-Planner, where code is regenerated
172 during execution upon failure by freezing the executed portion and providing the current observation
173 as context. The key difference is that it uses a dedicated reasoning model, specifically o4-mini, to
174 enhance task understanding and decision making. This improved reasoning enables more accurate
175 replanning. As with LLM-Planner, no explicit verification or validation is performed.

D Real-world Experiment Details

D.1 Figure 1 in Main Paper Details

In Figure 1 of the main paper, the bottom-right subfigures illustrating the success of the safe probe were swapped. The updated version is provided as Figure 4.

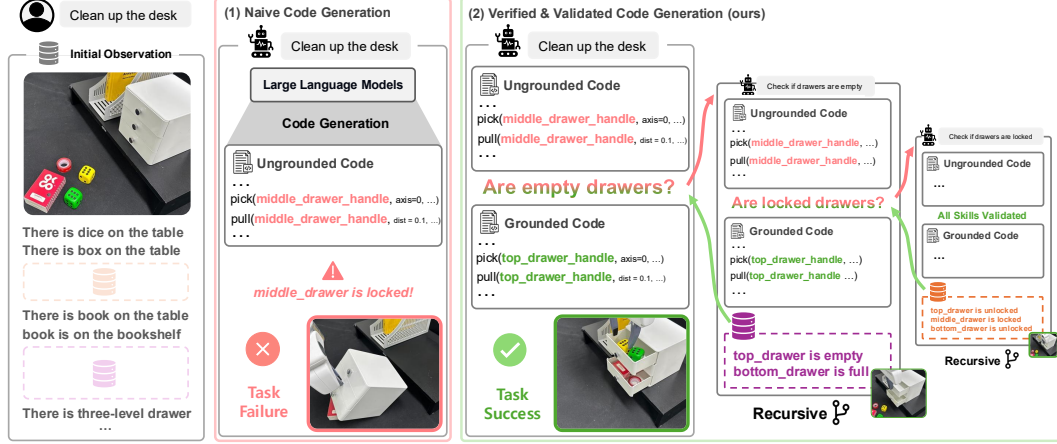


Figure 4: Concept of our NESYRO framework illustrated with an example of a room-cleaning task where drawer states are initially unknown. (Updated version)

In this section, we provide a detailed explanation of Figure 4. The complete execution sequence depicted in Figure 4 including all safe probes is illustrated in Figure 5. To demonstrate the reliability of NESYRO in a real-world setting, we tasked an embodied agent with the instruction “Clean up the desk” and compared NESYRO against a naive code generation approach. As depicted in Figure 4, the naive approach failed in partially observable environments, leading it to execute an irreversible action by not recognizing that the middle drawer might be locked. In contrast, NESYRO addresses this uncertainty using a safe probe pipeline to acquire the missing observations. It initially plans a safe probe to determine whether the drawers are empty. However, through its recursive validation phase, it subsequently identifies the need to observe the locked status of the drawers. Consequently, Safe Probe 1, which checks the locked status of the drawers, is executed first, as shown in Figure 5. Upon its completion, the agent adds observations confirming that the middle drawer is locked and that the top and bottom drawers are unlocked. Subsequently, Safe Probe 2, which checks whether the drawers are empty, is executed and adds observations confirming that both the top and bottom drawers are empty. With these observations acquired, the policy code is now grounded. NESYRO proceeds to successfully execute the “Clean up the desk” instruction.

D.2 Figure 5 in Main Paper Details

This section provides a detailed explanation of Figure 5 in main paper. The complete execution sequence depicted in Figure 5 of main paper including all safe probes is illustrated in Figure 6. To further evaluate the robustness of NESYRO in a real-world setting, we implemented the instruction “Place one dice into a drawer in a dark room” which represents partially observable environments. This requires auxiliary manipulation such as turning on the light to restore visibility before executing the main task. The initial policy code π_{main} was ungrounded, missing observations regarding drawer visibility and lock status. To resolve this, NESYRO activates its safe probe pipeline and first generates Safe Probe 1 to turn on the light, enabling the agent to perceive object locations. However, even after Safe Probe 1, the NeSyConf for the skill `pick(middle_H, ...)` remains below a threshold. In response, NESYRO generates Safe Probe 2 to check the lock status of the drawers. This probe confirms that the top and bottom drawers are unlocked, while the middle drawer is locked. Based on these observations, the code is refined by replacing the initial skills that placed one dice into the middle drawer with new skills that place it into the top drawer. As a result, the skill `pick(top_H, ...)` becomes ready to execute. Subsequently, during the validation of `pick(dice, ...)`, the agent identifies the need to check whether the drawers are empty. Consequently, NESYRO generates Safe

211 Probe 3 to check whether the drawers are empty. This probe detects trash inside the top drawer,
 212 leading to the insertion of additional code that removes it. These added skills also undergo the same
 213 validation phase in sequence. Once all skills are marked as ready to execute, the policy code is
 214 considered grounded. NESYRO proceeds to successfully execute the given instruction.

Instruction: Clean up the desk

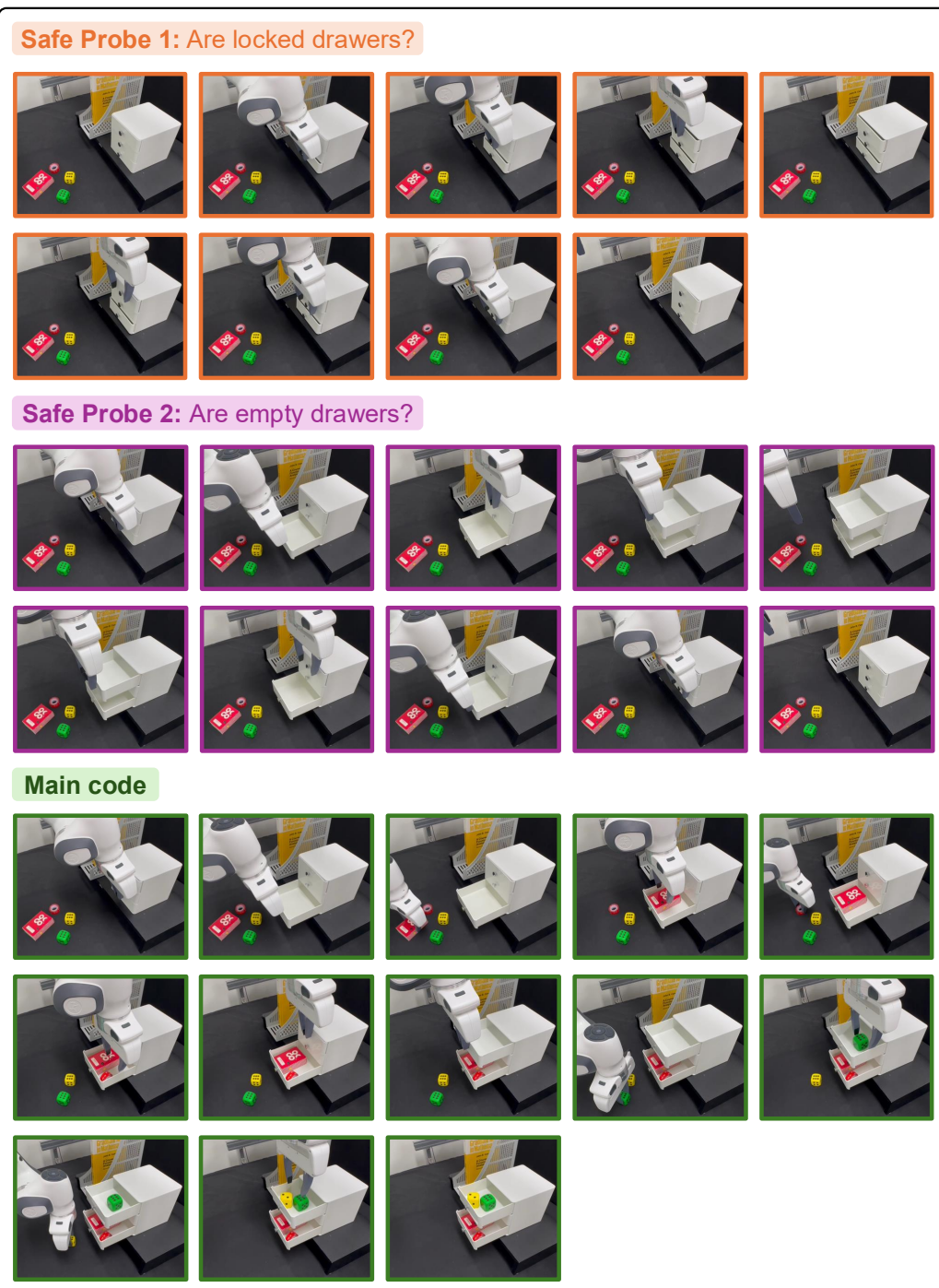


Figure 5: Real-world execution sequence of the instruction "Clean up the desk"

Instruction: Place one dice into a drawer in dark room



Figure 6: Real-world execution sequence of the instruction “Place one dice into a drawer in dark room”

References

- [1] Stephen James et al. “RLBench: The Robot Learning Benchmark & Learning Environment”. In: *IEEE Robotics and Automation Letters* 5.2 (2020), pp. 3019–3026.
- [2] David Coleman et al. “Reducing the barrier to entry of complex robotic software: a moveit! case study”. In: *arXiv preprint arXiv:1404.3785* (2014).
- [3] Lirui Wang et al. “GenSim: Generating Robotic Simulation Tasks via Large Language Models”. In: *Proceedings of the 12th International Conference on Learning Representations (ICLR)*. 2024.
- [4] Jacky Liang et al. “Code as policies: language model programs for embodied control”. In: *Proceedings of the 40th International Conference on Robotics and Automation (ICRA)*. IEEE, 2023, pp. 9493–9500.
- [5] Haoze Wu, Clark Barrett, and Nina Narodytska. “LEMUR: Integrating Large Language Models in Automated Program Verification”. In: *Proceedings of the 12th International Conference on Learning Representations (ICLR)*. 2024.
- [6] Pooja Aggarwal et al. “CodeSift: An LLM-Based Reference-Less Framework for Automatic Code Validation”. In: *arXiv preprint arXiv:2408.15630* (2024).
- [7] Chan Hee Song et al. “LLM-planner: Few-shot grounded planning for embodied agents with large language models”. In: *Proceedings of the 19th International Conference on Computer Vision (ICCV)*. 2023, pp. 2998–3009.
- [8] Qingyun Wu et al. “AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation”. In: *arXiv preprint arXiv:2308.08155* (2023).