

Appendix

This appendix provides detailed information about our method and experimental setup. It is organized as follows:

- In section [A](#), we first describe the prompting scheme used by different agents in InstructFlow, including both the shared initialization prompt and agent-specific templates for the planner, code generator, and symbolic constraint generator.
- In section [B.1](#), we provide additional details on our experimental setup, including how programs are instantiated and executed within the simulation pipeline. We also identify critical limitations in the original P_{Ro}C₃S evaluation protocol and introduce a VLM-based semantic check to address them.
- In section [B.2.1](#), we present an extended experiment exploring the effect of visual inputs on planning, followed by a summary of symbolic constraints induced across different manipulation tasks in section [B.2.2](#).
- In section [B.2.2](#), we include several representative case studies that illustrate how InstructFlow recovers from planning failures through symbolic reasoning and instruction graph-guided code repair.

A. InstructFlow Prompting Details

Here we provide details on the prompting scheme used for each agent in InstructFlow. The prompting template consists of two parts: a shared initialization template and agent-specific prompts. The complete structure is illustrated as follows:

Shared Prompt Templates:

```
1 {{{system_prompt}}}
2 {{{domain_setup_code}}}
3 {{{skill_preface}}}
4 {{{domain_skills}}}
```

Agent-Specific Prompt Templates:

```
1 {{{planner_role}}}
2 {{{code_generator_role}}}
3 {{{constraint_generator_role}}}
```

A.1. Shared Prompt Templates

All agents share a common initial prompt structure, consisting of: (a) `system_prompt`, (b) `domain_setup_code`, (c) `skill_preface`, (d) `domain_skills`. This shared context is constructed following the initial prompt setup introduced in PRoC3S (Curtis et al., 2024), where detailed environment and skill specifications are defined. We adopt the same structure without modification, and refer readers to the original paper for full specification details.

A.2. Agent-Specific Prompt Templates

As outlined in Section 3, we have three primary agents: the **InstructFlow Planner**, which decomposes the high-level goal into structured subgoals through instruction graph construction; the **Code Generator**, which translates the instruct graph into executable code snippets; and the **Constraint Generator**, which analyzes execution feedback to induce symbolic constraints for graph-guided code repair.

A.2.1. INSTRUCTFLOW PLANNER

Instruction Graph Construction Prompt

```
<inputs>
  {task_goal}
  {initial_state}
</inputs>
```

You are responsible for constructing an adaptive instruction graph that serves as an intermediate reasoning structure for robotic task planning. Your task is to generate a sequence of base planning nodes that decompose the task into subgoals. These nodes form the initial executable backbone of the plan. Do not include any reasoning nodes at this stage. Each node should include a semantic description of the subgoal, which can represent either:

- a concrete manipulation action (e.g., pick, place), or
- a prerequisite operation (e.g., selecting a target object, removing an obstacle).

Ensure subgoals are:

- logically ordered,
- collectively sufficient to achieve the goal.

Node Type: Planning Node

```
{ "id": "n1",
  "type": "action",
  "inputs": ["goal", "initial_state"],
  "output": "<natural language subgoal>"}
```

Instruction Graph Revision Prompt

```
<inputs>
  {task_goal}
  {initial_state}
```

```
{last_instruction_graph}
{symbolic_constraint}
</inputs>
```

Your task is to revise the instruction graph in response to symbolic constraints inferred from execution failures. This process involves constructing a reasoning flow that incrementally generates a refined instruction to update downstream planning nodes. You need to follow these steps:

Step 1: Reasoning Node Selection

You need to insert one or more reasoning nodes upstream of affected planning nodes guiding by symbolic constraints. Each reasoning node should perform a symbolic transformation relevant to the failure, and output an intermediate instruction fragment.

Step 2: Instruction Flow Construction

Sequentially process each reasoning node, using its output as an instruction fragment that incrementally updates the evolving task description. These fragments form a directed information flow. At the end of the reasoning node, concatenate all instruction fragments to form a single composite instruction that encodes the full reasoning chain and can be used to update downstream planning nodes.

Step 3: Instruction Graph Update

Use the composite instruction to update the planning nodes. Replace the original subgoal with this revised version, which incorporates both the task intent and refinements. The updated subgoals of instruction graph should then be passed as the final instruction to the code generator.

We define five types of reasoning nodes, each with its specific functionality and structure as described below:

-The spatial relation reasoning node analyzes pairwise spatial relationships between all visible objects to generate a spatial relation graph:

Node Type: *Spatial Relation Reasoning Node*

```
{ "id": "n2",
  "type": "spatial_relation_reasoning_node",
  "input": ["initial_state"],
  "output": <"spatial_relation_graph">}
```

-The object density reasoning node estimates the local spatial density around each object to reflect how crowded its surroundings are.

Node Type: *Object Density Reasoning Node*

```
{ "id": "n3",
  "type": "object_density_analysis_node",
  "input": ["initial_state"],
  "output": <"object_density_map">}
```

-The object selection reasoning node combines the goal, the output of the spatial perception node, and the output of the object density analysis node as the inputs to infer which correct manipulated object(s) should be selected to accomplish the goal.

Node Type: *Object Selection Reasoning Node*

```
{ "id": "n4",
  "type": "object_selection_reasoning_node",
  "input": ["goal", "spatial_relations_graph", "object_density_map", "symbolic_constraint"],
  "output": <"manipulated_objects">}
```

-The plan logic Reasoning Node combines the goal, the output of object selection reasoning node, and the symbolic predicate as the inputs to infer the correct execution order among the manipulated targets.

Node Type: *Plan Logic Reasoning Node*

```
{ "id": "n5",
  "type": "plan_logic_reasoning_node",
  "input": ["goal", "manipulated_objects", "symbolic_constraint"],
  "output": <"execution_order">}
```

-The parameter range reasoning node combines the goal and the symbolic predicate to adjust the ranges of action plan parameters based on explicit rules defined in symbolic predicates, and outputs instructions indicating whether to expand or shrink the ranges to meet task requirements.

Node Type: *Parameter Range Reasoning Node*

```
{ "id": "n6",
  "type": "parameter_range_adjustment_node",
  "inputs": ["goal", "symbolic_constraint"],
  "output": <"range_adjustment_instruction">}
```

A.2.2. CODE GENERATOR

Code Generator Prompt

```
<inputs>
{task_goal}
{initial_state}
{subgoals_instruction}
</inputs>
```

You are a code generation agent in a robotic planning system. Your goal is to generate two things:

First, generate a python function named 'gen_plan' that can take any discrete or continuous inputs. No list inputs are allowed and return the entire plan with all steps

included where the parameters to the plan depend on the inputs. The plan should be generated based on the initial high-level computation graph, which is composed of a sequence of subgoals. Each subgoal corresponds to either a manipulation action or a prerequisite operation.

Second, generate a python function 'gen_domain' that returns a set of bounds for the continuous or discrete input parameters. The number of bounds in the generated domain should exactly match the number of inputs to the function excluding the state input.

The function you give should always achieve the goal regardless of what parameters from the domain are passed as input. The 'gen_plan' function therefore defines a family of solutions to the problem. Explain why the function will always satisfy the goal regardless of the input parameters. Make sure your function inputs allow for as much variability in output plan as possible while still achieving the goal. Your function should be as general as possible such that any correct answer corresponds to some input parameters to the function.

The main function should be named EXACTLY 'gen_plan' and the domain of the main function should be named EXACTLY 'gen_domain'. Do not change the names. Do not create any additional classes or overwrite any existing ones. Aside from the initial state all inputs to the 'gen_plan' function MUST NOT be of type List or Dict. List and Dict inputs to 'gen_plan' are not allowed. Additionally, the input to 'gen_domain' must be exactly the 'initial:RavenBelief' argument, even if this isn't explicitly used within the function!

A.2.3. SYMBOLIC CONSTRAINT GENERATOR

Constraint Generator Prompt

```
<inputs>
  {task_goal}
  {initial_state}
  {failure_feedback}
  {generated_code}
</inputs>
```

You are a symbolic reasoning agent tasked with diagnosing execution failures in robotic manipulation tasks. Your goal is to induce generalizable symbolic constraints that explain the failure and can guide future plan correction. Your task is to perform the following reasoning steps to generate symbolic constraint(s) for plan repair:

Step 1. Failure-relevant retrieval:

Identify the exact code segment responsible for triggering the failure. In a failure feedback, "Step N" refers to the N-th action generated by the code (zero-based index). Specifically, it corresponds to the N-th `plan.append(Action(...))` call in the code. For example: "Step 1, Action: place" refers to the second action in the plan (index = 1).

Step 2. Code-level reasoning:

Extract all variables relevant to this failure, including:

- Manipulated objects.
- Object poses and spatial relations.
- Action parameters (e.g., offsets).
- Domain ranges for parameters.

Step 3. Diagnostic reasoning:

Based on the environment state and extracted variables, analyze the geometric or physical cause of failure. You must consider multiple possible causes, including but not limited to:

- Geometric violations (e.g., collisions, unstable placement, path obstruction).
 - Temporal inconsistencies (e.g., incorrect subgoal ordering, premature actions).
 - Symbolic logical errors (e.g., wrong object selection, missing prerequisite conditions).
- For physical causes, compute diagnostic metrics, including but not limited to:
- Proximity distances between relevant entities.
 - Existence of collision-free paths.
 - Stability metrics (e.g., center of mass projection).

For symbolic or logical causes, analyze:

- Whether the current action respects task-specific symbolic constraints.
 - Dependencies between subgoals as defined in the instruction graph.
 - Whether preconditions for the current subgoal are satisfied.
- Provide a concise diagnosis explaining why the failure occurred, explicitly stating:
- The type of cause (physical, temporal, symbolic).
 - The reasoning process leading to this conclusion.

You are not allowed to assume the cause based solely on the failure description. All conclusions must be verified through concrete reasoning over environment state, task semantics, or diagnostic metrics.

Step 4. Symbolic Constraint Induction:

Given the diagnosis and variables, formulate a symbolic constraint that abstracts the failure into a general, reusable rule. You must express the constraint using standard symbolic predicates, including but not limited to:

- Spatial relations: On, ClearOf, Aligned, StableOn...
- Temporal/ordering: Before, Precondition, Order...
- Semantic affordances: Reachable, Occludes, Graspable, Affords...
- Physical feasibility: ProximitySafe, PathClear, PlacementFeasible, ForceStable...

The constraint must express logical conditions involving task entities, their relationships, or parameter ranges, and be parameterized with thresholds or bounds when applicable.

B. Experiment Details and Additional Results

B.1. Experimental Setup Details

B.1.1. PROGRAM INSTANTIATION AND SIMULATION PIPELINE

Our framework follows the same two-stage code generation and execution process used in P_{RoC3S} (Curtis et al., 2024). Specifically, after generating the code that defines a task plan via LLM, the resulting Python function consists of two components: `get_plan()` and `get_domain()`.

The `get_plan()` function encodes a sequence of symbolic actions with continuous or discrete parameters, corresponding to the subgoals decomposed in the instruction graph. The `get_domain()` function specifies the sampling bounds for each parameter from a predefined sampler. These samplers (e.g., `ContinuousSampler`, `GraspSampler`) generate candidate values for plan parameters without awareness of environmental constraints such as collisions or instability.

To ground the abstract plan into an executable one, we adopt the same strategy as P_{RoC3S}: sample n parameter instantiations from the domains defined in `get_domain()`, and for each instantiation, evaluate the resulting plan in a physics-based simulator. If the plan violates any constraints (e.g., Kinematic, collisions, grasp, placement constraints), the simulator reports detailed constraint violation feedback. Once a constraint-free plan is found in simulation, it is deployed in the real environment.

B.1.2. FIXING PROTOCOL LIMITATIONS IN P_{RoC3S} EVALUATION

While our experimental setup builds directly on the original P_{RoC3S} (Curtis et al., 2024) framework and reuses its environment, skill library, and simulation interface, we identified structural limitations in its evaluation protocol. Specifically, P_{RoC3S} treats any execution that does not explicitly violate simulator constraints as successful, regardless of whether the task goal has been semantically achieved.

Incorrect but constraint-free Plan Misclassified as Successful For example, a plan may place an object in an incorrect position, fail to form the required structure (e.g., a pyramid), or manipulate the wrong object altogether. As long as no collisions or instability are triggered in simulation, such plans are incorrectly classified as successful. Figure 6 illustrates several cases where the task goal was clearly unmet, yet no feedback was generated to initiate replanning.

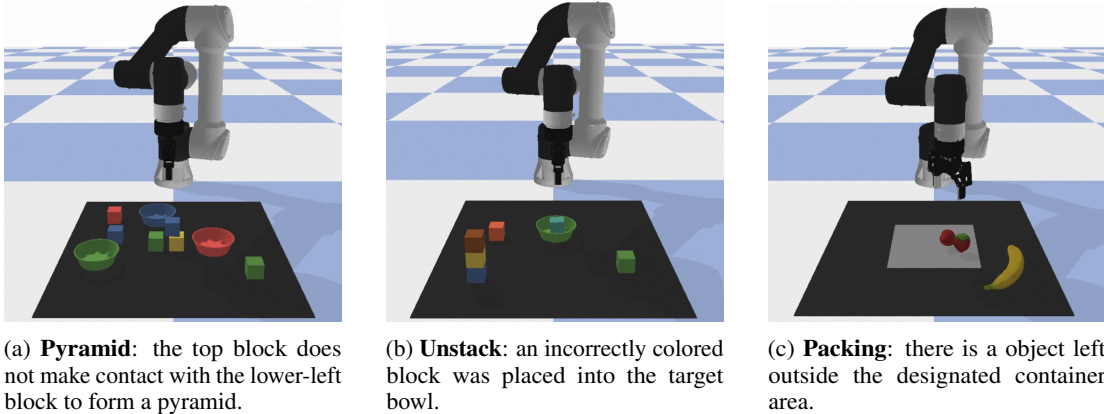


Figure 6. Task failures caused by incorrect plans that are accepted as successful by P_{RoC3S}, despite violating the task goal, because they do not trigger any constraint violations.

Fixing the Evaluation Protocol via VLM Check To address this fundamental evaluation gap, we introduce a semantic-level verification step using a vision-language model (GPT-4o). After executing each plan, we render the final scene and prompt the VLM to assess whether the natural language goal has been achieved. If not, the failure is recorded and propagated, triggering a replanning cycle. **Specifically, we explicitly instruct the VLM to evaluate structured visual conditions and provide clear success criteria along with positive and negative examples, enabling it to produce consistent and grounded success judgments.**

This fix does not modify the core P_{RoC3S} planning mechanism, but augments the evaluation logic with a reliable, goal-aware success signal. It enables all baselines—including P_{RoC3S} and InstructFlow—to be assessed under a consistent,

semantically meaningful criterion. While this change may lower the reported success rates of prior methods, we consider it essential for fair and rigorous comparison, especially in tasks with under-specified goals or ambiguous execution semantics.

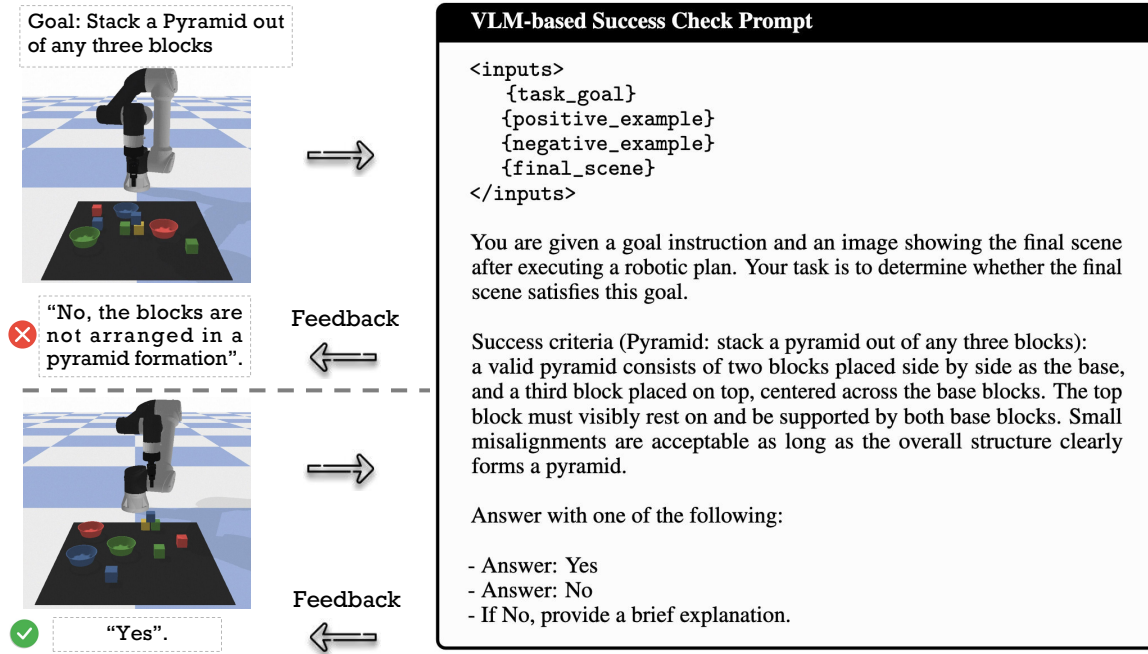


Figure 7. VLM-based success check on two execution plans for the Pyramid task. The upper plan produces a **non-standard pyramid** where the top block does not contact the left base block. The VLM detects this semantic error and returns “No” with reason. The lower plan yields a **correct pyramid** structure, and the VLM responds with “Yes”. This demonstrates the importance of incorporating VLM check as a complementary layer to evaluation protocol.

B.2. Additional Experiment Results and Cases

B.2.1. ADDITIONAL EXPERIMENT

To assess the potential of vision-language models (VLMs) for improving task planning, we conduct an additional experiment where our existing GPT-4o-based reasoning agent receive both the goal and an image of the initial scene, enabling multimodal visual reasoning.

This vision-augmented setup improves the planner’s ability to reason under partial observability or ambiguous symbolic states. As shown in Table 4, we observe performance gains in several visually grounded tasks, including *Packing (Blocks)*, *Unstack*, and *Packing (YCB)*, where early-stage visual cues help the system make better object selection or ordering decisions.

	Drawing				Arrange Blocks				Arrange YCB	
	Star	Arrow	Letters	Enclosed	Pyramid	Line	Packing	Unstack	Packing	Stacking
LLM ³ (w/ vision)	40%	40%	80%	50%	0%	40%	30%	0%	0%	10%
CaP (w/ vision)	10%	0%	40%	30%	20%	20%	20%	20%	40%	10%
PRoC3S (w/ vision)	90%	80%	80%	90%	60%	70%	60%	80%	40%	40%
InstructFlow (w/ vision)	100%	80%	100%	100%	90%	100%	90%	100%	60%	70%

Table 4. Task success rates (%) across drawing, block arrangement, and YCB manipulation tasks using VLM-based visual reasoning. Bold indicates the best-performing results.

```

1 def gen_plan(initial: RavenBelief,
2     +         x_clear1: float, y_clear1: float,
3     +         x_clear2: float, y_clear2: float,
4     +         x_offset: float, y_offset: float):
5     plan = []
6
7     # subgoal 1: Pick object_6 (the yellow block)
8     block6 = initial.objects["object_6"]
9     plan.append(Action("pick", block6.pose.point))
10
11    # subgoal 2: Place object_6 at a clear location
12    plan.append(Action("place", [x_clear1, y_clear1, block6.pose.point[2]]))
13    +
14    # subgoal 3: Pick object_5 (the teal block)
15    block5 = initial.objects["object_5"]
16    plan.append(Action("pick", block5.pose.point))
17
18    # subgoal 4: Place object_5 at a clear location
19    plan.append(Action("place", [x_clear2, y_clear2, block5.pose.point[2]]))
20
21    # subgoal 5: Pick object_4 (the green block)
22    block4 = initial.objects["object_4"]
23    plan.append(Action("pick", block4.pose.point))
24
25    # subgoal 6: Place object_4 into object_7 (the green bowl)
26    bowl7 = initial.objects["object_7"]
27    x, y, z = bowl7.pose.point
28    plan.append(Action("place", [x + x_offset, y + y_offset, z]))
29
30    return plan
    
```

Figure 8. A code snippet illustrating how InstructFlow repairs the **Pyramid** task plan: to correct the code, **InstructFlow** incorporates a **parameter range refinement node** into the instruction graph based on the inducted symbolic constraint, which guides the regenerated code to reduce the sampling range of **offset_x** in the **get_domain** function. This adjustment reduces the spacing between base blocks, enabling a **stable top placement** and successful pyramid construction.

B.2.2. CASE STUDY

In addition to the **Unstack** task discussed in section 4.4, we present several representative manipulation tasks as case studies to further demonstrate the effectiveness and generality of our method. For each task, we analyze the core planning difficulties, the symbolic constraints induced during execution, and how InstructFlow dynamically updates the instruction graph to recover from failures.

Pyramid The goal is to **stack a Pyramid out of any three blocks**, which means the robot need to construct a pyramid-like structure by selecting any three available blocks: two as the base and one stacked on top. The key challenge lies in both object selection and precise spatial configuration. Specifically, the lateral distance between the two base blocks must be carefully chosen to ensure that the top block can be stably placed across them. Furthermore, this task is especially sensitive to execution noise: even if a plan passes all physical constraint checks in simulation, the same stack may collapse in the real environment due to minor perturbations such as control inaccuracy or object pose estimation errors. **PRoC3S** (Curtis et al., 2024) lacks a feedback mechanism to detect post-simulation failures. Once a plan passes simulation, it is executed directly without verifying whether the real-world outcome satisfies the goal. Our method addresses this by introducing a VLM-based validation step: the executed scene is rendered and checked against the original goal, with replanning initiated if the structure is incorrect.

In the first round of code generation, InstructFlow receives the goal: “*Stack a pyramid out of any three blocks.*” The **InstructFlow Planner** constructs an initial instruction graph consisting of seven planning nodes, each representing a subgoal

in the pyramid assembly process. Guided by this graph, the code generator produces an executable plan. However, after executing the plan, VLM-based semantic validation reports a failure, indicating the following issue:

[Error Message]: "The blocks are not arranged in a pyramid formation"

Given the failure feedback and generated code, the **Constraint Generator** localizes the issue to the code block responsible for pyramid construction, specifically the subgoal corresponding to `subgoal5` in `get_plan()` and the `offset_x` parameter defined in `get_domain()`. By examining the sampled parameter values, InstructFlow infers that the current range of `offset_x` is too wide, causing the base blocks to be placed too far apart. As a result, the top block either falls during execution or forms a configuration that is not recognized as a pyramid by the VLM. This reasoning leads to the generation of an explicit symbolic constraint:

$$\phi_{\text{dist}} := \text{Distance}(\text{?block_bottom}) \in [0, 0.04]$$

Here, 0.04 corresponds to the environment-defined block size, ensuring that the top block can span both base blocks without falling or misalignment.

Given the symbolic constraint, **InstructFlow Planner** dynamically updates the instruction graph by introducing reasoning nodes that refine the value range of `offset_x`. Specifically, the updated graph incorporates a `parameter range refinement` node $v_{\text{param}}^{\text{reason}}$, whose output is an instruction to narrow the sampling bounds of `offset_x`.

Based on the updated instruction graph, the **Code Generator** regenerates the executable code to satisfy the induced symbolic constraints. As shown in Figure 5, the regenerated code modifies the `get_domain()` function by narrowing the sampling range of `offset_x` from the original `(BLOCK_SIZE, BLOCK_SIZE * 2)` to `(0, BLOCK_SIZE)`. This adjustment results in a tighter arrangement of the base blocks, enabling the top block to rest stably and form a recognizable pyramid structure. The final scene passes VLM validation, and the task is successfully completed.

Stacking (Arrange-YCB) This task requires the agent to stack one object on top of another, selected from a diverse set of YCB objects with varied shapes, sizes, and physical properties. The primary challenge lies in object selection: due to irregular geometries and asymmetric mass distributions, not all object pairs are feasible for stable stacking. Therefore, selecting an appropriate pair is critical to the success of the task. A second challenge arises from the lack of feedback handling in PRoC3S. Due to shape mismatch or small execution disturbances, the stacked configuration may fail in the real environment but without replanning.

In the first round of code generation, InstructFlow receives the goal: "Stack any object on any other object". The **InstructFlow Planner** constructs an initial instruction graph consisting of three planning nodes, corresponding to selecting two objects, picking the first object, and placing it on top of the second. Guided by this structure, the **Code Generator** generates an executable plan. However, after execution, VLM-based semantic validation reports a failure, indicating that the chosen object pair did not result in a valid or stable stacked configuration.

[Error Message]: "No objects are stacked on top of each other"

Given the failure feedback and generated code, the **Constraint Generator** localizes the issue to the code block responsible for selecting object pairs. Specifically, the analysis reveals that the object pair was selected via an environment-defined `DiscreteSampler()`, which randomly samples two objects without considering any factors that influence stackability. As a result, the constraint generator induces the following symbolic constraint to guide future selection:

$$\phi_{\text{stack}} := \text{AlignedForStacking}(\text{object_a}, \text{object_b}) \wedge \text{PlacementFeasible}(\text{object_a}, \text{object_b})$$

Given the symbolic constraint, **InstructFlow Planner** dynamically updates the instruction graph by introducing reasoning nodes that guide the selection of a physically compatible object pair for stacking. Specifically, the updated


```

1  - def gen_plan(initial: RavenBelief, object_name1: str, grasp1: RavenGrasp,
2      object_name2: str, grasp2: RavenGrasp):
3  + def gen_plan(initial: RavenBelief, grasp: RavenGrasp):
4
5      plan = []
6
7      #Subgoal 1: Pick the first selected object
8      plan.append(Action("pick", [object_name1, grasp1]))
9
10     #Subgoal 2: Get the pose of the second object to stack on top
11     second_object_pose = initial.objects[object_name2].pose
12
13     #Subgoal 3: Place the first object on top of the second object
14     plan.append(Action("place", [object_name1, grasp1, RavenPose(x=second_object_pose.x,
15         y=second_object_pose.y, z=second_object_pose.z + BLOCK_SIZE)]))
16
17     #Subgoal 1: Define the objects to be used in the plan
18     object_name1 = "object_2" # apple
19     object_name2 = "object_4" # power_drill
20
21     #Subgoal 2: Pick the first selected object (apple)
22     plan.append(Action("pick", [object_name1, grasp]))
23
24     #Subgoal 3: Get the pose of the second object (power_drill) to stack on top
25     second_object_pose = initial.objects[object_name2].pose
26
27     #Subgoal 4: Place the first object (apple) on top of the second object (power_drill)
28     plan.append(Action("place", [object_name1, grasp, RavenPose(x=second_object_pose.x,
29         y=second_object_pose.y, z=second_object_pose.z + BLOCK_SIZE)]))
30
31     return plan
32
33 def gen_domain(initial: RavenBelief):
34     object_ids = list(initial.objects.keys())
35     return {
36         "object_name1": DiscreteSampler(object_ids),
37         "grasp1": GraspSampler(),
38         "object_name2": DiscreteSampler(object_ids),
39         "grasp2": GraspSampler(),
40     }
41
42 + return {
43     "grasp": GraspSampler(),
44 }

```

Figure 9. A code snippet illustrating how InstructFlow repairs the **Stacking** task plan: to correct the code, **InstructFlow** updates the instruction graph based on the induced symbolic constraint. This update adjusts the subgoals from **randomly selecting stacking targets** to using a **specifically determined object pair**, guiding the regenerated code to modify the corresponding logic in **get_plan** and remove the random sampling from **get_domain**. This adjustment ensures the **physical feasibility** of the stacking operation.

graph includes a spatial_relation_reasoning_node $v_{\mathcal{T}^{spatial}}^{reason}$ to analyze relative object positions and an object_density_analysis_node $v_{\mathcal{T}^{density}}^{reason}$ to evaluate the local clutter surrounding each object. These outputs are fed into an object_selection_reasoning_node $v_{\mathcal{T}^{select}}^{reason}$, which selects object_2 and object_4 as a feasible stacking pair based on the symbolic constraint `SelectStackablePair`. A plan_logic_reasoning_node $v_{\mathcal{T}^{order}}^{reason}$ then determines the appropriate execution order, generating a subgoal sequence: “Pick object_2” followed by “Place object_2 on top of object_4.”.

Based on the updated instruction graph, the **Code Generator** regenerates the executable code to satisfy the induced symbolic constraints. As shown in Figure 6, the regenerated code eliminates the use of unconstrained object sampling in `gen_domain()`, replacing it with fixed object assignments (object_2 and object_4) determined through reasoning nodes. This ensures that the selected object pair adheres to the `AlignedForStacking` and `PlacementFeasible` constraints. The new plan simplifies the domain by sampling only the grasp for the manipulated object and directly encodes the stacking intent in the plan logic. With this targeted refinement, the resulting scene passes VLM-based check, and the stacking task is successfully completed.