InstructFlow: Adaptive Symbolic Constraint-Guided Code Generation for Long-Horizon Planning

Haotian Chi [1 2] **Zeyu Feng** [2] **Yueming Lyu** [2] **Chengqi Zheng** [2] **Linbo Luo** [3] **Yew-Soon Ong** [2 4] **Ivor Tsang** [2 4]
Hechang Chen [1] **Yi Chang** [1] **Haiyan Yin** [2]

Abstract

Long-horizon planning in robotic manipulation tasks requires translating under-specified, symbolic goals into executable control programs satisfying spatial, temporal, and physical constraints. However, language model-based planners often struggle with long-horizon task decomposition, robust constraint satisfaction, and adaptive failure recovery. We introduce **InstructFlow**, a multi-agent framework that establishes a symbolic, feedback-driven flow of information for code generation in robotic manipulation tasks. InstructFlow employs a **InstructFlow Planner** to construct and traverse a hierarchical instruction graph that decomposes goals into semantically meaningful subtasks, while a **Code Generator** generates executable code snippets conditioned on this graph. Crucially, when execution failures occur, a **Constraint Generator** analyzes feedback and induces symbolic constraints, which are propagated back into the instruction graph to guide targeted code refinement without regenerating from scratch. This dynamic, graph-guided flow enables structured, interpretable, and failure-resilient planning, significantly improving task success rates and robustness across diverse manipulation benchmarks, especially in constraint-sensitive and long-horizon scenarios.

1. Introduction

Large language models have become a prevalent approach for robotic code generation due to their ability to translate natural language instructions into executable programs (Chen et al., 2024; Liang et al., 2023; Mu et al., 2024; Tang et al., 2024). However, they often struggle with long-horizon task decomposition, adaptive failure recovery, and robust constraint satisfaction. For example, when instructed to place a target object into a bowl, LLM-based planners can detect that a grasp attempt fails due to collisions with stacked objects. But, they lack the ability to reason beyond the surface-level failure and infer the deeper structural cause, existing methods rely on blind retries or ad-hoc replanning, failing to adaptively repair the plan or refine the code to resolve the root cause.

These failures expose the fundamental limitation of relying solely on natural language prompts for planning: language is often under-specified, ambiguous, or too abstract to ground reliably in the physical world. When prompted to generate full plans directly, LLMs frequently hallucinate valid-looking but physically infeasible code or fail to recover when execution errors occur. Recent approaches have attempted to bridge the gap between language instructions and executable code by either directly generating grounded skill sequences with continuous parameters (Wang et al., 2024), or synthesizing end-to-end executable programs (Liang et al., 2023). However, these methods rely on flat planning paradigms that lack structured task decomposition or feedback-driven repair, often struggling in scenarios that require reasoning over dynamic failures or layered environment constraints.

Based on these, PRoC3S (Curtis et al., 2024) introduced a two-phase pipeline that separates plan generation and constraint checking, allowing failure-triggered replanning. While this improves robustness by incorporating environment-level feedback, its re-planning process remains reactive and lacks mechanisms to reason about the task-level causes of failures or to generalize corrective strategies across tasks.

To bridge this gap, we propose **InstructFlow**, a modular, multi-agent framework that establishes a symbolic, feedback-driven information flow for adaptive task planning and code generation. InstructFlow enables structured reasoning over execution failures, inducing interpretable symbolic

[1]Jilin University, China [2]CFAR and IHPC, Agency for Science, Technology and Research (A*STAR), Singapore [3]Xidian University, China [4]Nanyang Technological University, Singapore. Correspondence to: Haiyan Yin <yin_haiyan@cfar.a-star.edu.sg>, Hechang Chen <chenhc@jlu.edu.cn>, Yi Chang <yichang@jlu.edu.cn>.

ICML 2025 Workshop on Programmatic Representations for Agent Learning, Vancouver, Canada.

constraints that are injected back into the instruction graph. This dynamic flow of symbolic knowledge allows the system to restructure plans at both the task and code level, enabling targeted repair without full plan regeneration.

At the core of InstructFlow are two key innovations: First, InstructFlow introduces a hierarchical instruction graph that modularizes task planning into structured subgoals, while supporting dynamic, feedback-driven updates based on induced constraints. Second, its symbolic constraint induction mechanism abstracts raw execution failures into interpretable symbolic predicates, capturing both relational and physical task-level causes. These symbolic abstractions enable precise and generalizable plan and code refinement, improving robustness in complex, long-horizon, and constraint-sensitive tasks. This continuous flow of symbolic information throughout the instruction graph underpins InstructFlow's robustness and efficiency, distinguishing it from prior flat or reactive methods.

We highlight the following contributions: (i) InstructFlow framework, a modular system with hierarchical instruction graphs and symbolic reasoning for structured task decomposition and interpretable code generation; (ii) Symbolic constraint induction, feedback-driven mechanism that abstracts execution failures into reusable symbolic constraints for efficient and targeted plan repair; (iii) Comprehensive experiments on Drawing, Arrange-block, and Arrange-YCB demonstrate our method's superior success rates, robustness, and failure recovery over strong LLM-based baselines.

2. Related Works

LLM-Based Code Generation for Robotic Manipulation Recent advances highlight the potential of LLMs as general-purpose planners for robotic manipulation via code generation. CaP (Liang et al., 2023) introduces the use of LLMs to synthesize Python-based reactive controllers, integrating perception modules and control primitives. RoboScript (Chen et al., 2024) proposes a unified interface for deploying such code across simulation and real robots, focusing on deployability and modularity. LLM^3 (Wang et al., 2024) further integrates task and motion planning with LLM-driven failure reasoning for robust code generation in dynamic environments. Instruct2Act (Huang et al., 2023a) and VoxPoser (Huang et al., 2023b) combine LLMs with vision-language models, grounding language instructions into actionable code conditioned on perceptual inputs. RoboCodeX (Mu et al., 2024) introduces a tree-structured multimodal reasoning framework, decomposing language commands into object-centric manipulation code. Recent systems like OctoPack (Muennighoff et al., 2023) and RobotCode (Li et al., 2024) further enhance generalization and reliability by combining LLM-generated programs with skill libraries. While these methods demonstrate the effectiveness of LLMs when equipped with structured APIs, affordance models, and perceptual grounding, they still struggle with handling execution failures and adapting plans in constraint-sensitive or long-horizon tasks.

Symbolic Abstraction Planning Symbolic representations remain crucial for long-horizon and constraint-sensitive manipulation. Classical TAMP systems (Kaelbling & Lozano-Pérez, 2011; Curtis et al., 2022; Dantam et al., 2016; Garrett et al., 2021; Srivastava et al., 2014) integrate symbolic task planning with motion controllers but depend on domain-specific predicates. Traditional robotics planning relies on hard-coded symbolic world models (Garrett et al., 2021; Konidaris, 2019). Hybrid methods bridge language and symbolic planning, such as LLM+P (Liu et al., 2023a), which translates instructions into PDDL for optimal symbolic planning. VisualPredicator (Liang et al., 2025) learns neuro-symbolic predicates from visual inputs, while ViLaIn (Shirai et al., 2024) extracts scene-level symbolic representations from vision-language models. Other works like PlanBench (Valmeekam et al., 2023) create symbolic abstractions for plan feasibility analysis; P3IV (Zhao et al., 2022) and RLang (Rodriguez-Sanchez et al., 2023) focus on domain-agnostic symbolic representations aligned with LLM reasoning. Despite their use of symbolic abstractions, these approaches lack mechanisms for dynamic symbolic reasoning to abstract task-relevant information or diagnose failures for plan repair, which is a gap our method explicitly addresses by integrating symbolic reasoning into the code generation and planning loop.

Feedback-Driven Failure Recovery Recovering from execution failures remains a central challenge for LLM-driven robotics. Most approaches treat LLMs or VLMs as success detectors (Du et al., 2023; Ma et al., 2022; Wang et al., 2023), while recent works explore feedback-driven plan repair. REFLECT (Liu et al., 2023b) and AHA (Duan et al., 2024) leverage LLMs and VLMs for multi-modal failure explanation, enabling language-guided correction. RoboRepair (Schlesinger et al., 2024) and DoReMi (Guo et al., 2024) integrate LLMs and VLMs for execution misalignment detection and proactive repair. However, these methods primarily rely on fine-tuning models for failure understanding. To avoid fine-tuning, LLM^3(Wang et al., 2024), ProgPrompt(Singh et al., 2023), and CLAIRify (Skreta et al., 2023) propose failure-aware prompting and runtime verification to guide iterative plan repair. PRoC3S (Curtis et al., 2024) further introduces a hybrid approach that combines LLM-generated partial programs with post-hoc constraint optimization. Building on PRoC3S, our method integrates symbolic constraint induction and graph-guided plan repair into the code generation loop, enabling interpretable and adaptive failure recovery.

3. Methodology

We begin by outlining the LLM-based code generation paradigm for robotic manipulation. Then we introduce **InstructFlow**, detailing its instruction graph structure, agent roles, and symbolic constraint mechanism for structured, feedback-driven code generation and repair. An overview of InstructFlow is shown in Figure 1.

3.1. Overview

The problem of code generation for robotic manipulation involves translating natural language instructions into executable programs that operate reliably in robotic environments. Given task descriptions and initial state of the environment, a LLM is instructed to generate parameterized action plans that invoke low-level control routines to solve the task. Following a general paradigm (Curtis et al., 2024), the LLM produces two functions per task:

- ***get_plan***: a sequence of high-level actions conditioned on free parameters and environment state.
- ***get_domain***: the feasible ranges for those parameters (e.g., spatial offsets), defining the search space for plan instantiation.

At the task level, the goal is to generate code that completes the instructed objective without violating physical constraints. Execution feasibility is assessed by a continuous constraint satisfaction program (CCSP) module, which enforces four environment-level checks: *kinematic reachability*, *collision avoidance*, *grasp stability*, and *placement validity*. The LLM must reason not only over a series of *discrete action choices*, but also over *continuous numerical parameters*, which is a nontrivial requirement. Beyond sequencing skills, it must produce long-horizon code grounded in geometry, dynamics, and task-specific semantics. The core challenge lies in the lack of grounding from language prompts to low-level control logic that adheres to physical and dynamic constraints. When execution fails, LLMs often repeat or compound errors, exhibiting limited ability to adaptively repair code.

Our work aims to improve the failure recovery ability of LLM-based planners through structured planning and symbolic constraint-driven code repair.

We propose **InstructFlow**, a modular multi-agent framework for symbolic, feedback-driven code generation in robotic manipulation. The core idea is to introduce an **instruction graph** that hierarchically decomposes high-level task prompts into semantically structured subgoals, and couple this representation with a **symbolic constraint induction mechanism** for effective plan repair. InstructFlow modularizes the code-generation pipeline into three cooperative agents, each with a specialized role and local reasoning context (see Appendix A for the prompts used in our three Agents):

- **InstructFlow Planner**: Parses the task prompt and constructs a hierarchical instruction graph capturing semantic and spatial dependencies. Each node encodes a typed subgoal grounded in the robot's skill space.
- **Code Generator**: Translates each subgoal into executable Python code, producing symbolic control routines along with parameter domains for sampling feasible continuous values;
- **Constraint Generator**: Monitors execution failures and induces symbolic constraints that explain the cause. These constraints guide context-specific graph and prompt revisions, enabling targeted subgoal repair without full plan regeneration.

3.2. Instruction Graph Semantics

A central architectural contribution of InstructFlow is an hierarchical instruction graph that enables structured task decomposition and adaptive symbolic reasoning. At each interaction round t, the **InstructFlow Planner** constructs an instruction graph $\mathcal{G}_t = (\mathcal{V}_t, \mathcal{E}_t)$, conditioned on the task goal, initial state, and symbolic feedback. This graph acts as a typed, declarative scaffold for both task decomposition and constraint-aware refinement. The node set is partitioned as follows:

$$\mathcal{V}_t = \mathcal{V}_{\text{plan}} \cup \mathcal{V}_{\text{reason}}, \quad \mathcal{V}_{\text{plan}} \cap \mathcal{V}_{\text{reason}} = \emptyset, \tag{1}$$

with edges $\mathcal{E}_t \subseteq \mathcal{V}_t \times \mathcal{V}_t$ capturing symbolic or temporal dependencies. Each edge (v_i, v_j) denotes a directed flow of information, allowing parent nodes to influence the semantic context of their children.

(1) Planning nodes $v^{\text{plan}} \in \mathcal{V}_{\text{plan}}$ define grounded subgoals directly translatable into robot-executable code. Each is instantiated as a symbolic prompt:

$$v^{\text{plan}} : (\text{goal}, \text{state}) \rightarrow \text{subgoal}_{\mathcal{A}}, \quad \mathcal{A} \in \{\text{pick}, \text{place}, \ldots\}.$$

These nodes form the plan's executable backbone and anchor structural code generation.

(2) Reasoning nodes $v^{\text{reason}}_{\mathcal{T}_j} \in \mathcal{V}_{\text{reason}}$ perform typed symbolic transformations that enrich planning with task-level abstraction and constraint resolution. These typed modules abstract reusable domain knowledge, enabling modular plan revision: $v^{\text{reason}}_{\mathcal{T}_{select}} : \mathcal{I}_{\mathcal{T}_j} \rightarrow \mathcal{O}_{\mathcal{T}_j}$, where $\mathcal{I}_{\mathcal{T}_j}, \mathcal{O}_{\mathcal{T}_j}$ denote structured symbolic fields. Outputs are propagated to downstream planning nodes, injecting symbolic knowledge such as spatial adjacency or parameter tuning. We instantiate five

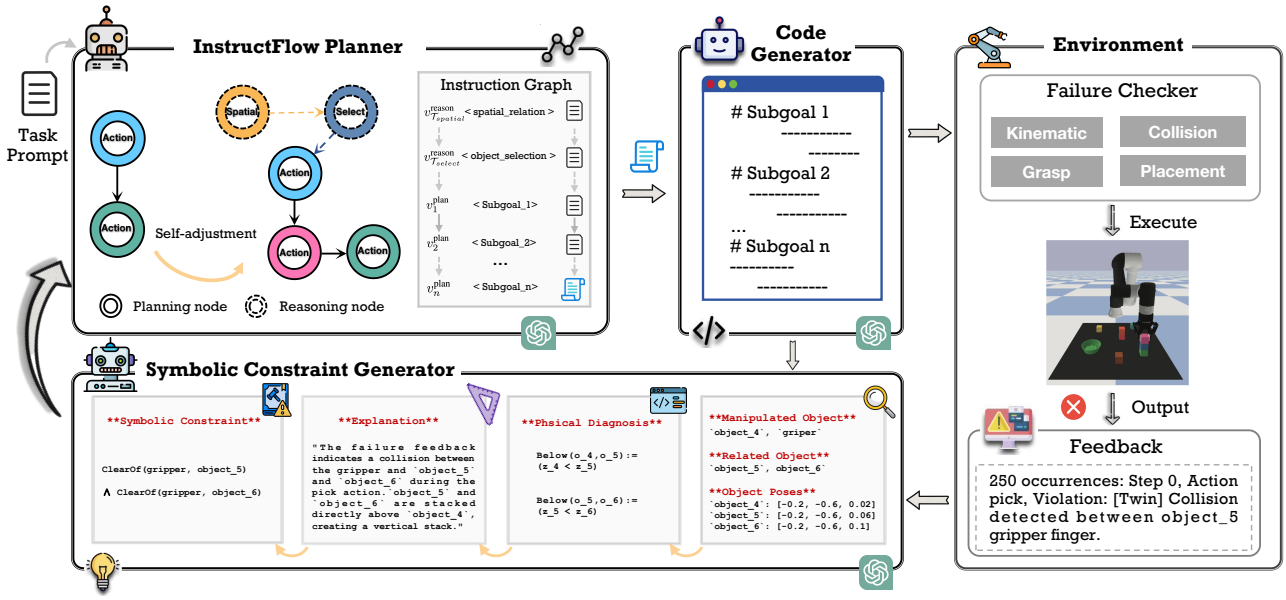


Figure 1. An overview of InstructFlow, a framework enabling multi-agent, symbolic, feedback-driven code generation for long-horizon robotic manipulation planning. The system consists of three coordinated agents: (a) **InstructFlow Planner**: Parses the task prompt and constructs a multi-level instruction graph of subgoals and reasoning nodes; (b) **Code Generator**: Generates executable code segments and samples parameter domains to instantiate the plan; (c) **Constraint Generator**: Analyzes failure traces and induces symbolic constraints that refine plan generation.

core reasoning modules:

$$\begin{aligned}
&\mathcal{T}_{\text{spatial}} : S \to \texttt{Rel}(\text{Objects, Adjacency}) && \text{(spatial relation inference)}\\
&\mathcal{T}_{\text{density}} : S \to \texttt{Rel}(\text{Objects, Density}) && \text{(local clutter estimation)}\\
&\mathcal{T}_{\text{select}} : (G, S, \Phi) \to \texttt{Select}(\text{Objects}) && \text{(target object selection)}\\
&\mathcal{T}_{\text{order}} : (G, S, \Phi) \to \texttt{Order}(\text{Actions}) && \text{(plan logic inference)}\\
&\mathcal{T}_{\text{param}} : (G, \Phi) \to \texttt{Refine}(\text{ParamDomain}) && \text{(parameter range refinement)}
\end{aligned}$$

Here, G denotes the high-level task goal, S represents the initial state, and Φ captures symbolic constraints induced from prior failures. These inputs are used by reasoning nodes to extract task-relevant abstractions for plan refinement.

Feedback-Driven Graph Update A core capability of **InstructFlow** is its ability to revise the instruction graph $\mathcal{G}_t$ based on symbolic constraint feedback and failure diagnostics. At initialization ($\texttt{constraint}_0 = \emptyset$), the graph contains only planning nodes. Upon failure (e.g., collisions, instability), the planner inserts reasoning nodes upstream of affected subgoals, dynamically composing a symbolic stack tailored to the error mode:

$$\mathcal{G}_t = \text{Planner}_{\text{LLM}}(\text{goal}, \text{state}, \text{constraint}_{t-1}). \tag{2}$$

This mechanism supports coarse-to-fine symbolic planning by injecting only the reasoning needed to refine or repair the faulty part of the task.

InstructFlow-Guided Code Generation InstructFlow translates symbolic plans into executable code by composing structured prompts along the instruction graph $\mathcal{G}_t$. Each planning node $v_i^{\text{plan}} \in \mathcal{V}_{\text{plan}}$ generates a prompt:

$$\begin{aligned}
\underbrace{\texttt{instr}^{(t)}}_{\text{Task Prompt}} &= \text{Encode}\left(\{v_{\mathcal{T}_j}^{\text{reason}}\}_{j=1}^{|v^{\text{reason}(t)}|},\ v^{\text{plan}}\right),\\
\underbrace{\texttt{code}^{(t)}}_{\text{Generated Code}} &= \text{LLM}\left(\texttt{instr}^{(t)}\right),
\end{aligned} \tag{3}$$

where $\texttt{Encode}(\cdot)$ integrates the subgoal with symbolic refinements from reasoning nodes, such as spatial relations, parameter ranges, or action dependencies. $|v^{\text{reason}(t)}|$ denotes the number of reasoning nodes providing contextual information to v^{plan}, including types such as spatial reasoning, object selection, and other reasoning nodes as defined above.

This symbolic conditioning guides the LLM to produce context-aware and physically valid code. When failures occur, **InstructFlow selectively updates the relevant subgoals and reasoning nodes impacted by the constraint violations**, avoiding unnecessary recomputation of unrelated parts of the plan. InstructFlow leverages symbolic instruction graphs to generate interpretable, constraint-compliant code, improving efficiency and robustness in long-horizon tasks.

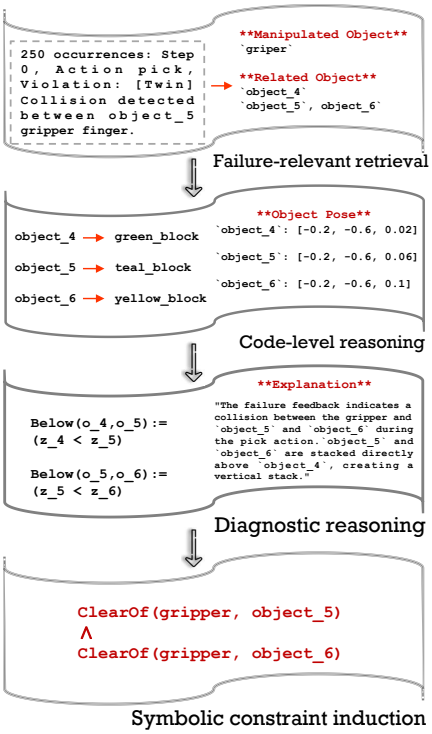


Figure 2. Failure Diagnosis Workflow of Symbolic constraint generator.

The Role of "Flow" While InstructFlow introduces multiple agents and a hierarchical instruction graph, the key distinguishing feature lies in the flow of symbolic information and feedback throughout the entire code generation loop. Unlike static prompting approaches, InstructFlow treats the prompt construction itself as a dynamic, graph-guided flow, where high-level goals, reasoning outputs, and failure-induced constraints are progressively injected into task-specific prompts at each planning node. This flow-centric prompt composition ensures that each code snippet is generated in a context-aware, failure-resilient, and constraint-compliant manner, enabling efficient plan repair without full regeneration. The flow mechanism thus operates at two intertwined levels: *Graph-level symbolic flow*: From reasoning nodes to planning nodes. *Prompt-level information flow*: From task goal, through symbolic reasoning and feedback, into structured, adaptive prompts.

3.3. Symbolic Constraint Induction from Execution Failures

LLM-based robotic planners often lack structured mechanisms for failure recovery, relying instead on naive re-prompts or implicit retries. We introduce a **Constraint Generator** that diagnoses execution failures, and abstracts them into logical constraints. These constraints serve as *symbolic corrections that guide graph restructuring and prompt refinement*, enabling interpretable, efficient, and generalizable plan repair.

Failure Diagnosis Workflow. As shown in Figure 2, the symbolic constraint induction follows a four-stage reasoning workflow: (i) **Failure-relevant entities retrieval**, which identifies failure-relevant entities from the failure trace $\mathcal{F}_t$ and executed code $\mathcal{P}_t$; (ii) **Code-level reasoning**, which instantiates involved variables and reasoning symbolic predicates that reflect the physical feasibility; (iii) **Diagnostic reasoning**, which compute geometric or geometric diagnostics based on predicates, such as collision proximity, path clearance, and placement stability; and (iv) **Symbolic constraint induction**, which abstracts diagnostic findings into declarative symbolic constraints that encapsulate the feasibility conditions violated by the current plan. This structured workflow transforms grounded execution failures into symbolic rules that guide prompt regeneration and enable interpretable, plan repair.

Physical Predicate as an Induction Basis. To enable interpretable failure diagnosis and structured symbolic constraint induction, we ground physical feasibility reasoning on a set of declarative physical predicates. These predicates abstract task-specific physical interactions into reusable logical representations, serving as the foundation for symbolic reasoning across diverse manipulation scenarios. We categorize predicates along four functional components:

(i) **Entities** ($\mathcal{E}$): Rather than pre-defining entities rigidly, InstructFlow dynamically abstracts task-relevant entities into functional roles based on the evolving task context and feedback, such as `?target` (manipulated object), `?neighbor` (potential obstacles), `?surface` (supporting structures), and `?gripper` (robot end-effector); (ii) **Relations** ($\mathcal{R}$): Symbolic relations are flexibly instantiated to capture emergent spatial and semantic interactions during task execution and diagnosis, such as `On(?a, ?b)` for support/contact, or `ClearOf(?a, ?b)` for proximity constraints, enabling contextual adaptation rather than relying on static domain rules; (iii) **Physical Functions** ($\mathcal{F}$): InstructFlow leverages a set of physical diagnostics as interpretable abstractions, such as `Dist(?a, ?b)`, `SupportArea(?obj)`, and `COMDeviation(?obj)`, which are dynamically evaluated in response to execution feedback, guiding the symbolic reasoning process without hard-coded thresholds; (iv) **Thresholds** ($\mathcal{B}$): Task-specific feasibility bounds, such as δ_{safe} for clearance margins, and η_{min} for support stability ratios, , which can be tuned or inferred based on the environment state and failure modes, allowing InstructFlow to generalize beyond fixed rule specifications.

These symbolic forms are grounded by diagnostics over physical basis, but are interpreted and manipulated as logi-

cal components of the instruction graph, which isolate the physical root causes of failure and ground them in explicit task parameters, forming the basis for symbolic abstraction.

Symbolic Constraint Induction. We formalize the symbolic constraint ϕ as a conjunction over two complementary modalities of failure correction: relational structure and physical feasibility:

$$\begin{aligned}\phi &:= \bigwedge_{c\in\mathcal{C}} c, \quad \text{where} \quad \mathcal{C}(\mathcal{E},\mathcal{R},\mathcal{F},\mathcal{B}) \\ &= \underbrace{\{R_i(e_{a_i}, e_{b_i})\}}_{\text{Relational Constraints}} \cup \underbrace{\{f_j(\Theta_j) \oplus \tau_j\}}_{\text{Physical Constraints}}. \end{aligned} \tag{4}$$

Here, for each relational constraint $R_i(e_{a_i}, e_{b_i})$, e_{a_i} and e_{b_i} are *entity instances* (e.g., `block`, `bowl`) participating in the relation R_i. For physical constraints, each term $f_j(\Theta_j) \oplus \tau_j$ represents a *feasibility condition*, where: Θ_j denotes the *variables* involved (e.g., poses, offsets), $\oplus$ is a *comparison operator* (e.g., $\leq$, $\geq$, or $=$), $\tau_j \in \mathcal{B}$ is a *task-specific threshold* (e.g., maximum allowable clearance).

This formulation allows each constraint ϕ to capture both high-level task semantics and low-level physical requirements within a unified logical form, which the **Constraint Generator** can compose into logical constraints ϕ for plan repair. For instance:

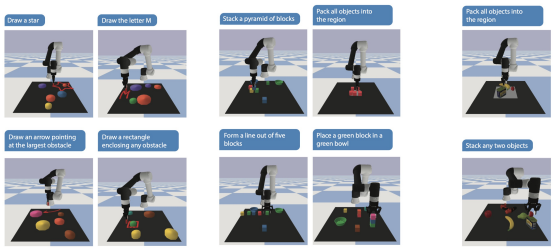
$$\begin{aligned}\phi_{\text{pick}} &:= \texttt{ProximitySafe}(?object, ?neighbor) \\ &\quad \wedge \texttt{PathClear}(?gripper, ?object), \\ \phi_{\text{place}} &:= \texttt{Dist}(?pose, ?neighbor) \geq \delta_{\text{safe}} \\ &\quad \wedge \texttt{StableOn}(?object, ?surface).\end{aligned}$$

Notably, these symbolic constraints act as structured priors for graph refinement and generalize across task instances and environments, enabling not just plan repair but modular, interpretable priors that can be reused across planning episodes. By treating failure correction as symbolic program refinement, this representation integrates seamlessly into our instruction graph and enables feedback-driven, structurally grounded prompt generation.

4. Experiments

4.1. Experimental Setup

We adopt the same environments, evaluation metrics, and protocol as PRoC3S (Curtis et al., 2024) to ensure fair comparison, while extending its core planning pipeline with symbolic reasoning and constraint-guided repair. All experiments are conducted in the Ravens simulation environment, using a 6-DoF UR5 arm with a Robotiq 2F-85 gripper in a tabletop workspace. Physics-based execution and constraint checking are handled via PyBullet. Simulations run on CPUs with 32GB RAM, with all baseline implementations integrated into a unified evaluation framework.



(a) Drawing (b) Arrange-Blocks (c) Arrange-YCB

Figure 3. Tasks in our simulated environments, along with corresponding goals.

Domains and Tasks. We evaluate our approach on three simulated domains, each designed to test different aspects of long-horizon planning with parameterized skills and physical constraints:

1. **Drawing**: The robot is equipped with a `draw_line` primitive that generates 2D trajectories to render geometric and symbolic shapes on a surface, while avoiding randomly placed objects. These tasks require precise parameter coordination under tight spatial constraints.
2. **Arrange-Blocks**: The robot stacks and arranges colored blocks and bowls to form pyramids, lines, or centered clusters. This domain tests stability, spatial accuracy, and planning under clutter and occlusions.
3. **Arrange-YCB**: The robot manipulates complex objects from the YCB dataset (e.g., banana, meat can) to perform packing and stacking. Irregular geometries introduce challenges in grasping, placement feasibility, and collision avoidance.

Constraints. Across all domains, generated plans are evaluated against a set of physical and geometric constraints that reflect real-world robotic limitations: (1) **Kinematic constraints** ensure that the robot's inverse kinematics solver produces a reachable end-effector pose, rejecting infeasible motions; (2) **Collision constraints** eliminate plans that lead to unintended contact between the robot, environment, or other objects, allowing only expected contact such as during grasps; (3) **Grasp constraints** verify that the gripper properly encloses the object and maintains stability during lifting, rejecting grasps that cause slippage or collision; (4) **Placement constraints** require that, upon release, the object remains upright and stationary, i.e., any post-placement drift or instability signals a failure of physical feasibility.

Baselines. We compare our approach against three baselines:

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

Table 1. Task success rates (%) across drawing, block arrangement, and YCB manipulation domains. Bold indicates top-performing results.

1. **PRoC3S** (Curtis et al., 2024): The original two-phase LLM-based planner that separates plan generation and constraint satisfaction using a sampling-based solver with feedback.
2. **LLM^3** (Wang et al., 2024): A recent method in which the LLM directly outputs grounded skill sequences with continuous parameters.
3. **Code-as-Policies (CaP)** (Liang et al., 2023): A program synthesis-based strategy that uses an LLM to produce complete Python programs encoding the action sequence and continuous parameters for execution.

Execution Details. Each approach is evaluated over 10 randomized seeds per simulated task. We use a maximum budget of 1000 samples per trial (10000 for drawing tasks). We limit the number of feedback iterations to 5. All methods are queried via OpenAI's GPT-4o unless otherwise stated. A task is considered successful if the final robot state satisfies the goal condition without violating any constraints (see Appendix B.1 for more details on experiments setting).

4.2. Benchmark Experiments

We benchmark InstructFlow against PRoC3S, LLM^3, and CaP across three domains. As shown in Table 1, across drawing, block arrangement, and YCB manipulation tasks, InstructFlow outperforms prior methods by 20–40% in task success rate.

This improvement stems from InstructFlow's ability to perform structured symbolic reasoning over task-specific failures, enabling targeted plan corrections at multiple levels: (i) refining parameter domains to satisfy geometric constraints, (ii) inducing symbolic relations (e.g., adjacency, clearance) to prevent repeated failures, and (iii) revising subgoal sequences based on environment feedback. For example, in *Pyramid* and *Line*, baseline methods like PRoC3S often fail due to improper block spacing. InstructFlow detects these failures and augments the instruction graph with adjacency constraints (e.g., `Adjacent(?block_i, ?block_j)`), guiding offset adjustments to improve stability without exhaustive re-planning.

Similarly, in *Packing* tasks involving irregular YCB objects,

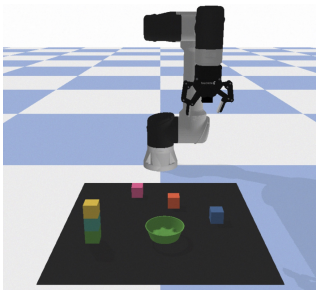


Figure 4. Illustrative image of the environment for the `Unstack` task from **Arrange Blocks**.

InstructFlow uses proximity constraints to guide precise placement corrections. When initial plans yield collision-prone configurations, violated `ClearOf` constraints are identified, and placement ranges refined to ensure collision-free, feasible solutions—capabilities absent in flat prompt-based methods (see Appendix B.2.1).

Ablation results in Table 2 highlight the distinct roles of symbolic planning and constraint induction in InstructFlow's performance. Without the Planner, the system loses its ability to structure tasks hierarchically, resulting in brittle plans and severe failures in multi-step spatial tasks (e.g., *Pyramid*, *Packing*, with up to 50% drops). Removing Constraint Induction disables feedback-driven repair, forcing the model into blind retries that struggle with physical feasibility, leading to 30–40% degradation in cluttered and precision-sensitive tasks. These results confirm that InstructFlow's robustness stems from the synergy of symbolic task decomposition and failure-informed constraint refinement.

4.3. Discovered Symbolic Constraints

Table 3 summarizes the symbolic constraints that were automatically induced based on failure feedback across different manipulation tasks. From a content perspective, we highlight the following key properties:

Structural Consistency. All induced constraints conform to the symbolic constraint formulation presented in Section 3.3. Each constraint instance can be expressed either as a relational predicate $R_i(e_a, e_b)$ or a physical threshold condition $f_j(\Theta) \oplus \tau$. This ensures that all constraints are

	Drawing				Arrange Blocks				Arrange YCB	
	Star	Arrow	Letters	Enclosed	Pyramid	Line	Packing	Unstack	Packing	Stacking
Ours	**100%**	**80%**	**100%**	**100%**	**90%**	**100%**	**90%**	**90%**	**60%**	**70%**
Ours w/o Planner Agent	90%	80%	80%	100%	50%	90%	50%	40%	40%	40%
Ours w/o Constraint Agent	100%	80%	100%	80%	40%	100%	60%	60%	30%	40%

Table 2. Ablation study results (% task success) highlighting the contributions of the InstructFlow Planner and Symbolic Constraint Generator.

Task	Symbolic Constraints
Pyramid	`StableOn`(?block_top, ?block_bottom); `ProximitySafe`(?block_bottom); `PathClear`(?gripper, ?block); `Before`(place(block_top), place(?block_bottom)); `Distance`(?block_bottom) ∈[0, 0.04] `Aligned`(?block_bottom) ∧ `On`(block_top, ?block_bottom); `Order`(Place(block_bottom1), Place(block_bottom2), Place(block_top)); `Contact`(block_top, ?block_bottom) ∧ `CloseTogether`(?block_bottom); `CenterOfMass`(block$_{top}$) ∈ `SupportArea`(*?block_bottom*)
Line	`StableOn`(?block, ?table); `Alignment`(?block); `Alignment_tolerance` ∈[-0.01, 0.01]
Packing (Blocks)	`ProximitySafe`(?block, ?boundary); `PlacementFeasible`(?block, square_region); `WithinBounds`(?block, region_center, 2*blcok_size); `ProximitySafe`(block, region_center) ∧ `WithinDistance`(?block, region_center, ?max_distance)
Unstack (Blocks)	`StableOn`(?block, ?bowl); `ProximitySafe`(?gripper, ?block); `Cleafof`(?gripper,?obstacle); `PlacementFeasible`(green_block, green_bowl); `Inside`(green_block, green_bowl); `Aligned`(block_center, bowl_center); `NotStacked`(green_block, ?obstacle) ∧ `offset_z` > 0.03
Packing (YCB)	`ProximitySafe`(?gripper, ?object); `GraspFeasible`(?grasp, ?object_pose); `ProximitySafe`(object, table_center); `Distance`(?object, table_center) < 0.06; `PlacementFeasible`(?object, center, threshold) ∧ `threshold` = 0.06; `Graspable`(?grasp, ?object); `CollisionFree`(?gripper, ?object)
Stacking (YCB)	`OnTop`(object_a, object_b); `ClearOf`(object_a, surface); `ClearSurface`(object_b); `PlacementFeasible`(object_a, object_b); `StableOn`(object_a, object_b); `AlignedForStacking`(object_a, object_b); `Graspable`(?object, ?grasp) ∧ `CollisionFree`(?object)

Table 3. Induced constraints for each task across Arrange-Blocks and Arrange-YCB domains

logically composable, interpretable, and grounded in the formal symbolic space defined by ϕ.

Diverse Coverage of Constraint Types. The constraint set spans a wide range of task-relevant constraint categories, including:

- **Spatial safety:** `ProximitySafe`, `ClearOf`, `PathClear...`
- **Placement feasibility:** `PlacementFeasible`, `Aligned`, `StableOn...`
- **Geometric parameters:** `Distance`, `Offset`, `CenterOfMass...`
- **Temporal logic:** `Before`, `Order...`

These categories capture both physical feasibility and symbolic reasoning failure modes, supporting a broad range of corrective strategies.

Notably, the table does not include constraints for the Drawing tasks. This is because the drawing tasks are comparatively simpler in structure and were typically solved by the planner in a single attempt without triggering any failure-driven refinement. As a result, no symbolic constraint induction process was invoked for these tasks, and they are therefore excluded from this table.

4.4. Case Study

We take the **Unstack** task as a case study to illustrate the effectiveness of InstructFlow. The goal is to **place the green block into the green bowl**, but the task poses hidden challenges: the green block is often buried beneath a stack, making direct access infeasible. Naive pick attempts cause collisions with blocks above, violating the constraints and leading to failure.

Existing methods such as PRoC3S can detect execution failures and make localized repairs, such as inserting obstacle removal steps. However, they struggle with multi-layered occlusions. When the green block is buried under multiple stacked objects, PRoC3S lacks a structured mechanism to reason about the correct removal order. As a result, it often generates plans with invalid sequences or actions that reintroduce collisions, ultimately failing the task.

Symbolic Constraints In the first round of code generation, InstructFlow receives the goal: place the green block into the green bowl. It constructs an initial instruction graph with two planning nodes: `pick object_4` and `place object_7`. Correspondingly, the code attempts to pick `object_4` at its current pose and place it at the target location.

Execution feedback, however, reveals repeated collisions during the `pick` action:

> **[Error Message]**: "250 occurences: Step 0, Action pick, Violation: [Twin] Collision detected between object_5 object gripper finger" and "250 occurences: Step 0, Action pick, Violation: [Twin] Collision detected between object_6 object gripper finger"

Given the **failure feedback** and **generated code**, the **Con-**

straint Generator localizes the root cause to the `pick` action on `object_4`, identifying the involved variables: the *manipulated object*, *its pose*, and *the interfering objects* (`object_5`, `object_6`). By analyzing the spatial configuration, InstructFlow infers that the gripper's approach vector intersects with the stacked obstacles, violating collision constraints. This reasoning leads to the generation of explicit **symbolic constraints**:

$$\phi_{\text{unstack}} := \texttt{ClearOf(gripper, object\_5)} \wedge \texttt{ClearOf(gripper, object\_6)}$$

These constraints distill raw collision feedback into symbolic predicates that express the essential condition: the gripper must reach the target without obstruction. This abstraction transforms a low-level failure into a reusable, structured signal for graph updates and targeted code repair.

Dynamic InstructFlow Graph Update. Given the symbolic constraints, **InstructFlow Planner** dynamically updates the instruction graph by introducing reasoning nodes that explicitly model the spatial and logical dependencies in the scene.

Specifically, the updated graph includes a `spatial reasoning node` $v^{\text{reason}}_{\mathcal{T}_{spatial}}$ that analyzes vertical stacking relations between objects, inferring that `object_5` and `object_6` are stacked above `object_4` via symbolic `below` relations. Based on this output, an `object selection reasoning node` $v^{\text{reason}}_{\mathcal{T}_{selection}}$ identifies `object_5` and `object_6` as obstacles to be removed according to the induced `ClearOf` constraints. A `logic reasoning node` $v^{\text{reason}}_{\mathcal{T}_{logic}}$ then determines the action sequence that satisfies these constraints, ensuring the objects are unstacked top-down. These reasoning nodes collectively refine the instruction graph by introducing new planning nodes to first move `object_6` (yellow), then `object_5` (teal), and finally pick `object_4` (green), reflecting the inferred symbolic dependencies.

Code Repair Based on the updated instruction graph, **Code Generator** regenerates the executable code to satisfy the induced symbolic constraints. Unlike black-box retries, the code repair is explicitly guided by InstructFlow's graph structure, ensuring that prerequisite actions (e.g., obstacle removal) are correctly sequenced before the primary task.

As shown in Fig. 5, the repaired code respects both spatial constraints (via clear placement of obstacles) and temporal dependencies (via correct unstacking order), demonstrating InstructFlow's ability to produce interpretable, constraint-compliant programmatic policies. (See Appendix B.2.2 for more case study).

```
def gen_plan(initial: RavenBelief,
+            x_clear1: float, y_clear1: float,
+            x_clear2: float, y_clear2: float,
             x_offset: float, y_offset: float):
    plan = []

    # subgoal 1: Pick object_6 (the yellow block)
    block6 = initial.objects["object_6"]
    plan.append(Action("pick", block6.pose.point))

    # subgoal 2: Place object_6 at a clear location
    plan.append(Action("place", [x_clear1, y_clear1, block6.pose.point[2]]))
+
    # subgoal 3: Pick object_5 (the teal block)
    block5 = initial.objects["object_5"]
    plan.append(Action("pick", block5.pose.point))

    # subgoal 4: Place object_5 at a clear location
    plan.append(Action("place", [x_clear2, y_clear2, block5.pose.point[2]]))

    # subgoal 5: Pick object_4 (the green block)
    block4 = initial.objects["object_4"]
    plan.append(Action("pick", block4.pose.point))

    # subgoal 6: Place object_4 into object_7 (the green bowl)
    bowl7 = initial.objects["object_7"]
    x, y, z = bowl7.pose.point
    plan.append(Action("place", [x + x_offset, y + y_offset, z]))

    return plan
```

Figure 5. A code snippet illustrating how InstructFlow repairs the `Unstack` plan by intuitively injecting a targeted object removal routine automatically derived from InstructFlow's structural reasoning process.

5. Conclusions

We presented InstructFlow, a multi-agent framework for long-horizon robotic manipulation that combines symbolic constraints reasoning with feedback-driven code repair. By introducing an adaptive instruction graph with symbolic constraints reasoning, InstructFlow decomposes complex tasks into structured subgoals, enabling interpretable, constraint-aware code generation and robust plan repair. Empirical results across challenging benchmarks validate the system's ability to handle long-horizon, constraint-sensitive scenarios with improved success rates and sample efficiency. Looking ahead, we plan to extend InstructFlow to incorporate visual grounding and multi-modal constraint induction, enabling even richer symbolic reasoning from unstructured feedback in the physical world.

Acknowledgments

This research is supported by the National Research Foundation, Singapore and Infocomm Media Development Authority under its Trust Tech Funding Initiative, Career Development Fund (CDF) of the Agency for Science, Technology and Research (A*STAR) (No: C233312007, No: C243512014), and the National Research Foundation, Singapore under its AI Singapore Programme (AISG Award No: AISG-NMLP-2024-003), and, in part, by the Key R&D Project of Jilin Province, China (No. 20240304200SF). Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not reflect the views of the National Research Foundation, Singapore, and Infocomm Media Development Authority.

References

Chen, J., Mu, Y., Yu, Q., Wei, T., Wu, S., Yuan, Z., Liang, Z., Yang, C., Zhang, K., Shao, W., et al. Roboscript: Code generation for free-form manipulation tasks across real and simulation. *arXiv preprint arXiv:2402.14623*, 2024.

Curtis, A., Fang, X., Kaelbling, L. P., Lozano-Pérez, T., and Garrett, C. R. Long-horizon manipulation of unknown objects via task and motion planning with estimated affordances. In *2022 International Conference on Robotics and Automation, ICRA 2022, Philadelphia, PA, USA, May 23-27, 2022*, pp. 1940–1946. IEEE, 2022. doi: 10.1109/ICRA46639.2022.9812057. URL https://doi.org/10.1109/ICRA46639.2022.9812057.

Curtis, A., Kumar, N., Cao, J., Lozano-Pérez, T., and Kaelbling, L. P. Trust the proc3s: Solving long-horizon robotics problems with llms and constraint satisfaction. In Agrawal, P., Kroemer, O., and Burgard, W. (eds.), *Conference on Robot Learning, 6-9 November 2024, Munich, Germany*, volume 270 of *Proceedings of Machine Learning Research*, pp. 1362–1383. PMLR, 2024. URL https://proceedings.mlr.press/v270/curtis25a.html.

Dantam, N. T., Kingston, Z. K., Chaudhuri, S., and Kavraki, L. E. Incremental task and motion planning: A constraint-based approach. In Hsu, D., Amato, N. M., Berman, S., and Jacobs, S. A. (eds.), *Robotics: Science and Systems XII, University of Michigan, Ann Arbor, Michigan, USA, June 18 - June 22, 2016*, 2016. doi: 10.15607/RSS.2016.XII.002. URL http://www.roboticsproceedings.org/rss12/p02.html.

Du, Y., Konyushkova, K., Denil, M., Raju, A., Landon, J., Hill, F., de Freitas, N., and Cabi, S. Vision-language models as success detectors. *arXiv preprint arXiv:2303.07280*, 2023.

Duan, J., Pumacay, W., Kumar, N., Wang, Y. R., Tian, S., Yuan, W., Krishna, R., Fox, D., Mandlekar, A., and Guo, Y. Aha: A vision-language-model for detecting and reasoning over failures in robotic manipulation. *arXiv preprint arXiv:2410.00371*, 2024.

Garrett, C. R., Chitnis, R., Holladay, R. M., Kim, B., Silver, T., Kaelbling, L. P., and Lozano-Pérez, T. Integrated task and motion planning. *Annu. Rev. Control. Robotics Auton. Syst.*, 4:265–293, 2021. doi: 10.1146/ANNUREV-CONTROL-091420-084139. URL https://doi.org/10.1146/annurev-control-091420-084139.

Guo, Y., Wang, Y.-J., Zha, L., and Chen, J. Doremi: Grounding language model by detecting and recovering from plan-execution misalignment. In *2024 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 12124–12131. IEEE, 2024.

Huang, S., Jiang, Z., Dong, H., Qiao, Y., Gao, P., and Li, H. Instruct2act: Mapping multi-modality instructions to robotic actions with large language model. *arXiv preprint arXiv:2305.11176*, 2023a.

Huang, W., Wang, C., Zhang, R., Li, Y., Wu, J., and Fei-Fei, L. Voxposer: Composable 3d value maps for robotic manipulation with language models. *arXiv preprint arXiv:2307.05973*, 2023b.

Kaelbling, L. P. and Lozano-Pérez, T. Hierarchical task and motion planning in the now. In *IEEE International Conference on Robotics and Automation, ICRA 2011, Shanghai, China, 9-13 May 2011*, pp. 1470–1477. IEEE, 2011. doi: 10.1109/ICRA.2011.5980391. URL https://doi.org/10.1109/ICRA.2011.5980391.

Konidaris, G. On the necessity of abstraction. *Current opinion in behavioral sciences*, 29:1–7, 2019.

Li, J., Chen, P., Wu, S., Zheng, C., Xu, H., and Jia, J. Robocoder: Robotic learning from basic skills to general tasks with large language models. *arXiv preprint arXiv:2406.03757*, 2024.

Liang, J., Huang, W., Xia, F., Xu, P., Hausman, K., Ichter, B., Florence, P., and Zeng, A. Code as policies: Language model programs for embodied control. In *IEEE International Conference on Robotics and Automation, ICRA 2023, London, UK, May 29 - June 2, 2023*, pp. 9493–9500. IEEE, 2023. doi: 10.1109/ICRA48891.2023.10160591. URL https://doi.org/10.1109/ICRA48891.2023.10160591.

Liang, Y., Kumar, N., Tang, H., Weller, A., Tenenbaum, J. B., Silver, T., Henriques, J. F., and Ellis, K. Visualpredicator: Learning abstract world models with neuro-symbolic predicates for robot planning. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net, 2025. URL https://openreview.net/forum?id=QOfswj7hij.

Liu, B., Jiang, Y., Zhang, X., Liu, Q., Zhang, S., Biswas, J., and Stone, P. LLM+P: empowering large language models with optimal planning proficiency. *CoRR*, abs/2304.11477, 2023a. doi: 10.48550/ARXIV.2304.11477. URL https://doi.org/10.48550/arXiv.2304.11477.

Liu, Z., Bahety, A., and Song, S. Reflect: Summarizing robot experiences for failure explanation and correction. *arXiv preprint arXiv:2306.15724*, 2023b.

Ma, Y. J., Sodhani, S., Jayaraman, D., Bastani, O., Kumar, V., and Zhang, A. Vip: Towards universal visual reward and representation via value-implicit pre-training. *arXiv preprint arXiv:2210.00030*, 2022.

Mu, Y., Chen, J., Zhang, Q., Chen, S., Yu, Q., Ge, C., Chen, R., Liang, Z., Hu, M., Tao, C., Sun, P., Yu, H., Yang, C., Shao, W., Wang, W., Dai, J., Qiao, Y., Ding, M., and Luo, P. Robocodex: Multimodal code generation for robotic behavior synthesis. In *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*. OpenReview.net, 2024. URL https://openreview.net/forum?id=xnQ1qoly7Q.

Muennighoff, N., Liu, Q., Zebaze, A., Zheng, Q., Hui, B., Zhuo, T. Y., Singh, S., Tang, X., Von Werra, L., and Longpre, S. Octopack: Instruction tuning code large language models. In *NeurIPS 2023 Workshop on Instruction Tuning and Instruction Following*, 2023.

Rodriguez-Sanchez, R., Spiegel, B. A., Wang, J., Patel, R., Tellex, S., and Konidaris, G. Rlang: a declarative language for describing partial world knowledge to reinforcement learning agents. In *International Conference on Machine Learning*, pp. 29161–29178. PMLR, 2023.

Schlesinger, C., Guha, A., and Biswas, J. Creating and repairing robot programs in open-world domains. *arXiv preprint arXiv:2410.18893*, 2024.

Shirai, K., Beltran-Hernandez, C. C., Hamaya, M., Hashimoto, A., Tanaka, S., Kawaharazuka, K., Tanaka, K., Ushiku, Y., and Mori, S. Vision-language interpreter for robot task planning. In *IEEE International Conference on Robotics and Automation, ICRA 2024, Yokohama, Japan, May 13-17, 2024*, pp. 2051–2058. IEEE, 2024. doi: 10.1109/ICRA57147.2024.10611112. URL https://doi.org/10.1109/ICRA57147.2024.10611112.

Singh, I., Blukis, V., Mousavian, A., Goyal, A., Xu, D., Tremblay, J., Fox, D., Thomason, J., and Garg, A. Progprompt: Generating situated robot task plans using large language models. In *IEEE International Conference on Robotics and Automation, ICRA 2023, London, UK, May 29 - June 2, 2023*, pp. 11523–11530. IEEE, 2023. doi: 10.1109/ICRA48891.2023.10161317. URL https://doi.org/10.1109/ICRA48891.2023.10161317.

Skreta, M., Yoshikawa, N., Arellano-Rubach, S., Ji, Z., Kristensen, L. B., Darvish, K., Aspuru-Guzik, A., Shkurti, F., and Garg, A. Errors are useful prompts: Instruction guided task programming with verifier-assisted iterative prompting. *CoRR*, abs/2303.14100, 2023. doi: 10.48550/ARXIV.2303.14100. URL https://doi.org/10.48550/arXiv.2303.14100.

Srivastava, S., Fang, E., Riano, L., Chitnis, R., Russell, S., and Abbeel, P. Combined task and motion planning through an extensible planner-independent interface layer. In *2014 IEEE International Conference on Robotics and Automation, ICRA 2014, Hong Kong, China, May 31 - June 7, 2014*, pp. 639–646. IEEE, 2014. doi: 10.1109/ICRA.2014.6906922. URL https://doi.org/10.1109/ICRA.2014.6906922.

Tang, H., Key, D., and Ellis, K. Worldcoder, a model-based llm agent: Building world models by writing code and interacting with the environment. *Advances in Neural Information Processing Systems*, 37:70148–70212, 2024.

Valmeekam, K., Marquez, M., Olmo, A., Sreedharan, S., and Kambhampati, S. Planbench: An extensible benchmark for evaluating large language models on planning and reasoning about change. *Advances in Neural Information Processing Systems*, 36:38975–38987, 2023.

Wang, L., Ling, Y., Yuan, Z., Shridhar, M., Bao, C., Qin, Y., Wang, B., Xu, H., and Wang, X. Gensim: Generating robotic simulation tasks via large language models. *arXiv preprint arXiv:2310.01361*, 2023.

Wang, S., Han, M., Jiao, Z., Zhang, Z., Wu, Y. N., Zhu, S., and Liu, H. Llm^3: Large language model-based task and motion planning with motion failure reasoning. In *IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2024, Abu Dhabi, United Arab Emirates, October 14-18, 2024*, pp. 12086–12092. IEEE, 2024. doi: 10.1109/IROS58592.2024.10801328. URL https://doi.org/10.1109/IROS58592.2024.10801328.

Zhao, H., Hadji, I., Dvornik, N., Derpanis, K. G., Wildes, R. P., and Jepson, A. D. P3iv: Probabilistic procedure planning from instructional videos with weak supervision. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 2938–2948, 2022.