# HiBT: From Hierarchical Concepts to Behavior Trees for Reliable Robot Task Planning

Anonymous authors https://anonymous-2026.github.io/HiBT
Paper under double-blind review

## Abstract

Long-horizon robotic assembly requires planners to preserve object relations, tool states, and ordered preconditions while producing executable control structures. Direct language-model generation leaves these dependencies implicit in a flat token sequence. We propose HiBT, a learned–symbolic planner that formulates Behavior Tree (BT) synthesis as hierarchical concept planning. A learned module predicts a task-conditioned hierarchical interface from the goal and symbolic state; deterministic realization then closes domain dependencies and compiles the grounded plan into a BT. On a 60-instance assembly benchmark with four open-source backbones, HiBT solves all 60 tasks with logically coherent BTs and subsecond planning latency. Matched controls and interface ablations isolate the gains from hierarchical organization and grounding. In LIBERO-Long, HiBT reaches 89.2% final-goal success with shorter rollouts. The dependency-intensive LIBERO-Hier suite further connects these gains to compliance with execution-order and process constraints. A dependency-aware learned–symbolic interface thus improves executable BT synthesis and embodied long-horizon planning.

## 1 Introduction

Language-conditioned robotic assembly requires executable plans that preserve object relations, tool states, and ordered preconditions. Long-horizon tasks combine tool switching, re-grasping, and intermediate dependencies: an insertion, for example, may require the correct tool and grasp. Behavior Trees (BTs) express condition checks and reusable control structures (Iovino et al., 2020; 2023). Existing synthesis methods use evolutionary computation, reinforcement learning, and structured planning (Colledanchise et al., 2019; Banerjee, 2018; Li et al., 2021). Recent language-model planners generate action sequences, programs, or BTs from task descriptions (Singh et al., 2023; Song et al., 2023; Driess et al., 2023; Zhou et al., 2024; Ao et al., 2025; Merino-Fidalgo et al., 2025; Ichter et al., 2023; Yao et al., 2023b; Wang et al., 2023a). Executable assembly requires explicit dependencies among stable conditions supporting grounded actions (Wang et al., 2023b).

We propose HiBT, a learned–symbolic planner that formulates BT synthesis as hierarchical concept planning. Conditioned on the instruction and current state, a learned module predicts a hierarchical interface. Prototype-guided decoding grounds its template and slots; symbolic realization completes tool, grasp, and ordering dependencies; and a recursive compiler produces the BT. This design connects hierarchical reasoning (Wei et al., 2022; Yao et al., 2023a; Zhou et al., 2023; Khot et al., 2023; Hao et al., 2023; Chen et al., 2023a; Lyu et al., 2023) to executable task structure, with learning selecting the decomposition and symbolic operations enforcing its dependencies.

This work makes three contributions. We formulate BT synthesis through a hierarchical interface with explicit subgoal dependencies, and introduce a learned–symbolic procedure for grounding, repairing, and compiling that interface. Evaluation spans 60 symbolic assembly tasks across four backbones and embodied LIBERO rollouts. Matched flat controls and component ablations identify the gains from hierarchical structure, slot grounding, and stable-subgoal checks. HiBT consistently improves executable assembly planning across backbones and strengthens both final-goal and process-constrained success in embodied rollouts.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

## 2 Related Work

**Structured robot planning.** PDDL specifies goals and action models for search (McDermott et al., 1998); HTN planning decomposes tasks through domain methods (Erol et al., 1994); and TAMP couples task choices with geometry (Kaelbling & Lozano-Pérez, 2011). BTs provide a reactive execution interface (Iovino et al., 2020), with synthesis methods based on learning, acquired control policies, and formal specifications (Colledanchise et al., 2019; Banerjee, 2018; Li et al., 2021). HiBT learns task-conditioned guidance over symbolic predicates and actions, then compiles their dependencies into a BT for downstream execution.

**Language-conditioned planning.** LLM planners produce action sequences, symbolic plans, and executable programs (Singh et al., 2023; Song et al., 2023; Rana et al., 2023; Hazra et al., 2024; Chen et al., 2023b; Yao et al., 2023b). BT-oriented approaches include hybrid planning from human instructions (Chen et al., 2024), one-step and adaptive tree generation (Lykov & Tsetserukou, 2023; Zhou et al., 2024; Izzo et al., 2024), and execution-aware synthesis (Ao et al., 2025; Merino-Fidalgo et al., 2025). HiBT introduces an explicit intermediate hierarchy of goals, stable subgoals, and grounded methods, exposing the support relations used by the compiler before tree generation.

**Hierarchical reasoning interfaces.** Decompositional prompting and planning-oriented reasoning organize computation through intermediate states and programs (Zhou et al., 2023; Khot et al., 2023; Hao et al., 2023; Liu et al., 2023; Liang et al., 2023; Wang et al., 2023a; Gao et al., 2023; Chen et al., 2023a; Lyu et al., 2023). Recent methods also learn layered or latent representations (Gui et al., 2025; Huang et al., 2026; Su et al., 2025; Shen et al., 2025; Amos et al., 2026). HiBT connects this representation learning to BT synthesis: the instruction–state pair determines a hierarchy, prototype retrieval grounds its individual slots, and symbolic realization enforces executable dependencies. Interface ablations distinguish the roles of template routing and slot-level grounding.

## 3 Method

HiBT learns a task-conditioned planning interface and realizes it as an executable BT (Figure 1). A training-time Builder extracts concept targets from annotated plan pyramids. These targets supervise a Predictor and populate a family-filtered prototype bank. At inference, the Predictor produces a latent pyramid; prototype decoding grounds its template and slots; symbolic realization completes dependencies; and the compiler converts the grounded method graph into a BT.

### 3.1 Problem Formulation

A task is a language-conditioned assembly instance

$$\mathcal{I} = (u, s_0, \gamma, \mathcal{D}), \tag{1}$$

with instruction $u$, initial symbolic state $s_0$, target predicate $\gamma$, and domain $\mathcal{D}$. The domain specifies predicates, grounded actions, and their preconditions $P_a$ and positive/negative effects $E_a^+, E_a^-$. An applicable action updates the state as

$$\delta(s, a) = (s \setminus E_a^-) \cup E_a^+, \qquad P_a \subseteq s. \tag{2}$$

HiBT produces a BT that reaches $\gamma$ from $s_0$; Appendix B defines the domain and leaves.

### 3.2 Hierarchical Plan Pyramid

HiBT uses the four-level symbolic plan pyramid in Eq. 3:

$$P = (\gamma, \mathcal{S}, \mathcal{M}, \pi), \tag{3}$$

where $\gamma$ is the final target predicate, $\mathcal{S} = \{q_j\}_{j=1}^{J}$ is the set of stable subgoals, $\mathcal{M} = \{m_k\}_{k=1}^{K}$ is the set of grounded method nodes, and $\pi = \langle a_k \rangle_{k=1}^{K}$ is a linearized action sequence retained for reporting and action-leaf consistency evaluation. Each method node is defined as

$$m_k = (q_k, a_k, U_k). \tag{4}$$

Here $q_k \in \{\gamma\} \cup \mathcal{S}$ is the condition achieved by $a_k \in \mathcal{A}$, and $U_k \subseteq \mathcal{S}$ contains the stable subgoals that should hold before executing $a_k$. Compilation operates on the dependency structure encoded by $\mathcal{M}$, whereas $\pi$ is used only for reporting and evaluation.

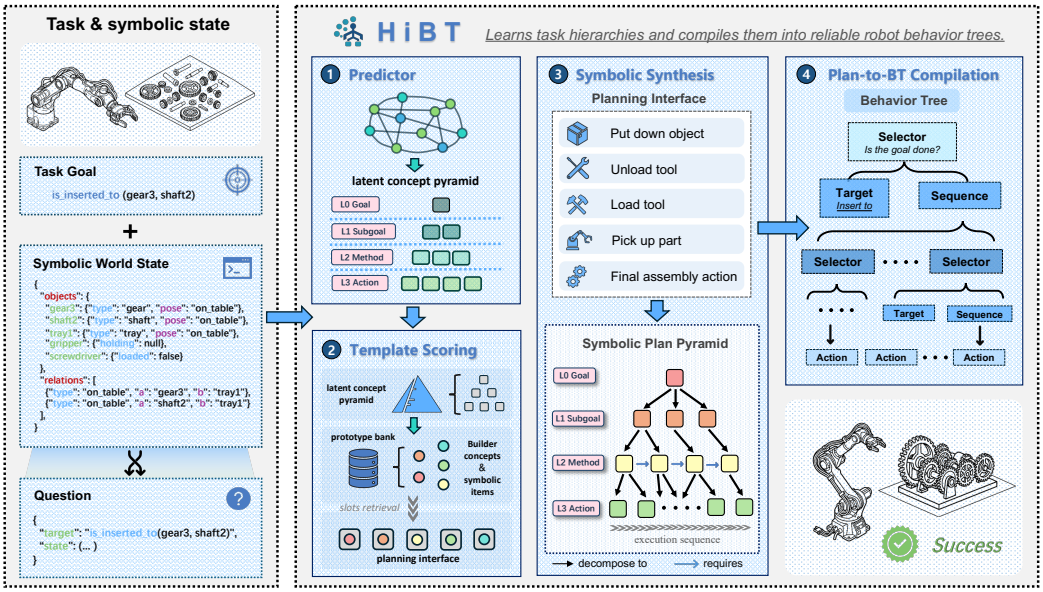



**Figure 1:** Overview. The planner predicts a hierarchical concept interface from the task instruction and initial state, completes dependency-aware support conditions, and compiles the repaired plan into an executable BT.

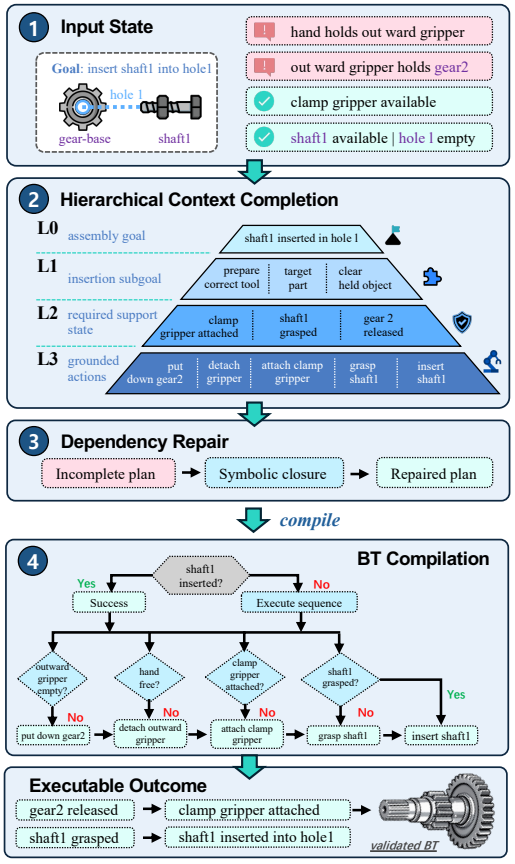



**Figure 2:** HiBT completes context and repairs support dependencies before compiling the gear-insertion BT.

**Table 1:** Main symbols used in the method.

| Symbol | Definition |
|---|---|
| $\mathcal{I}$ | Language-conditioned assembly instance. |
| $u, s_0, \gamma$ | Task instruction, initial symbolic state, and target predicate. |
| $\mathcal{D}$ | Symbolic assembly domain with predicates, actions, preconditions, and effects. |
| $\mathbf{n}$ | Slot-count vector over the concept levels. |
| $N$ | Total number of latent concept slots. |
| $h_\phi$ | Training-time Builder that extracts target concept pyramids from annotated plans. |
| $f_\theta$ | Inference-time Predictor that maps $(u, s_0)$ to a predicted concept pyramid. |
| $\Xi$ | Hierarchical planning interface induced by prototype-guided decoding. |
| $P$ | Dependency-aware plan pyramid with goal, stable subgoals, method nodes, and action sequence. |
| $\mathcal{T}$ | Output Behavior Tree. |

Figure 2 illustrates this process on a gear-insertion instance: HiBT introduces stable support conditions and clears and switches the occupied tool before the final insertion. Table 1 summarizes the main objects used in the method.

**Algorithm 1** Hierarchical Interface Induction

**Input:** task instruction $u$, initial symbolic state $s_0$, prototype bank $\mathcal{B}$
**Output:** hierarchical planning interface $\Xi = (\tau^*, \mathcal{R})$

Predict hierarchy, $\hat{\mathbf{Z}} \leftarrow f_\theta(u, s_0)$
**for** each $b_j = (\mathbf{C}_j, \tau_j, \Sigma_j) \in \mathcal{B}$ **do**
    $s_j \leftarrow \frac{1}{4}\sum_{\ell=0}^{3} \cos(\mathbf{v}(\hat{\mathbf{Z}}^{(\ell)}), \mathbf{v}(\mathbf{C}_j^{(\ell)}))$
**end for**
Select template, $j^* \leftarrow \arg\max_j s_j$
$\tau^* \leftarrow \tau_{j^*}$
$\mathcal{R} \leftarrow \emptyset$
**for** $\ell \in \{0, 1, 2, 3\}$ and $i \in [n_\ell]$ **do**
    $r_i^{(\ell)} \leftarrow \arg\max_{p \in \mathcal{B}_i^{(\ell)}(\tau^*)} \cos(\hat{\mathbf{z}}_i^{(\ell)}, \mathbf{c}(p))$
    Retrieve slot item, $\mathcal{R} \leftarrow \mathcal{R} \cup \{r_i^{(\ell)}\}$
**end for**
**return** $\Xi = (\tau^*, \mathcal{R})$

### 3.3 Concept Interface

The learned representation has four levels with slot counts $\mathbf{n} = (1, 4, 5, 5)$ and $N = 15$ total slots. Each level is a tensor $\mathbf{Z}^{(\ell)} \in \mathbb{R}^{n_\ell \times d}$, aligned with the annotated plan hierarchy (Wang et al., 2023a; Lyu et al., 2023). The Builder and Predictor produce target and predicted pyramids:

$$\mathbf{C}^* = h_\phi(u, s_0, P^*), \qquad \hat{\mathbf{Z}} = f_\theta(u, s_0), \tag{5}$$

where $P^*$ is an annotated symbolic plan pyramid, $\mathbf{C}^*$ is the target concept pyramid, and $\hat{\mathbf{Z}}$ is the predicted concept pyramid. The Predictor is trained with

$$\mathcal{L} = \lambda_Z \mathcal{L}_Z + \lambda_\pi \mathcal{L}_\pi, \tag{6}$$

Here $\mathcal{L}_Z$ is the per-level mean squared error between predicted and Builder slots, and $\mathcal{L}_\pi$ is the teacher-forced loss on symbolic plan serialization. Inference uses the predicted concept pyramid, whose level- and slot-aligned representations organize task-conditioned structure.

Prototype-guided decoding maps the predicted concept pyramid to a structured planning interface (Gao et al., 2023; Chen et al., 2023a). The prototype bank is

$$\mathcal{B} = \{b_j = (\mathbf{C}_j, \tau_j, \Sigma_j)\}_{j=1}^{M}, \tag{7}$$

where $\mathbf{C}_j$ is a Builder concept pyramid, $\tau_j$ is its structural template, and $\Sigma_j$ is the associated symbolic plan schema. Each offline bank entry pairs a Builder target with the template and slot-aligned items from its annotated training plan; family exclusion precedes supervision and bank construction. Template matching uses the template score in Eq. 8:

$$s_\tau(b_j) = \frac{1}{4}\sum_{\ell=0}^{3} \cos\left(\mathbf{v}(\hat{\mathbf{Z}}^{(\ell)}), \mathbf{v}(\mathbf{C}_j^{(\ell)})\right), \tag{8}$$

where $\mathbf{v}(\cdot)$ vectorizes a level tensor. The selected bank entry is indexed by $j^* = \arg\max_j s_\tau(b_j)$ and $\tau^* = \tau_{j^*}$. For each slot, the decoder retrieves the nearest symbolic item within that template:

$$r_i^{(\ell)} = \arg\max_{p \in \mathcal{B}_i^{(\ell)}(\tau^*)} \cos\left(\hat{\mathbf{z}}_i^{(\ell)}, \mathbf{c}(p)\right). \tag{9}$$

Let $\Phi$ denote prototype-guided decoding. The induced interface is

$$\Xi = \Phi(\hat{\mathbf{Z}}; \mathcal{B}) = (\tau^*, \mathcal{R}), \qquad \mathcal{R} = \{r_i^{(\ell)}\}_{\ell,i}. \tag{10}$$

Algorithm 1 summarizes decoding; Appendix B details bank construction and retrieval.

### 3.4 Dependency Closure

Realization grounds the learned structure and completes dependencies under the domain semantics (Lyu et al., 2023). Let $\rho_\mathcal{D}$ instantiate a sequence, $\kappa_\mathcal{D}$ close dependencies, and $\Lambda$ lift it into a pyramid:

$$\pi_0 = \rho_\mathcal{D}(\gamma, s_0, \Xi), \quad \pi^* = \kappa_\mathcal{D}(\pi_0, s_0), \quad \tilde{P} = \Lambda(\gamma, \pi^*). \tag{11}$$

The initial sequence $\pi_0$ is instantiated from the retrieved template and slot-level retrieval signals. Dependency closure then checks each action against the current symbolic state. For an action $a$ in state $s$, the missing dependency set is

$$\Delta(a, s) = P_a \setminus s. \tag{12}$$

Whenever $\Delta(a, s) \neq \emptyset$, the closure operator inserts domain-supported actions that establish the missing conditions before $a$ is executed. Closure uses deterministic domain-rule dispatch. In the assembly domain, when a candidate assembly action requires an incompatible tool state, dependency closure inserts release, detach, attach, and grasp actions before the final assembly action.

For action $a_k$ in state $s^{(t)}$, $\nu$ selects a missing predicate under the domain's repair priority, and $\zeta_{\mathcal{D}}$ supplies its repair segment:

$$\begin{aligned} q_t &= \nu\Big(P_{a_k} \setminus s^{(t)}\Big), \quad \bar{\pi}_t = \zeta_{\mathcal{D}}(q_t, s^{(t)}), \\ s^{(t+1)} &= \delta^*(s^{(t)}, \bar{\pi}_t). \end{aligned} \tag{13}$$

Here $\delta^*$ repeatedly applies Eq. 2. Repair segments are appended until $P_{a_k}$ holds, after which $a_k$ is appended and simulated (Algorithm 2). The repaired sequence is then lifted into the plan pyramid $\tilde{P} = (\gamma, \mathcal{S}, \mathcal{M}, \pi)$, where stable subgoals record support conditions checked or re-established during BT ticking. For embodied manipulation, a benchmark adapter maps task predicates to the same interface, whose subgoals guide a low-level visual policy (Appendix D.2).

### 3.5 BT Compilation

The compiler recursively maps the repaired plan pyramid to a BT:

$$\mathcal{T} = C(\tilde{P}). \tag{14}$$

For any condition $q \in \{\gamma\} \cup \mathcal{S}$, let $m(q)$ denote the method node that achieves $q$, and let $U_{m(q)}$ be its required stable subgoals. The branch $B_q$ achieves $q$ after these required stable subgoals have been compiled. The recursive compilation rule is

$$B_q = \sigma_\wedge \left[ [C(r)]_{r \in U_{m(q)}}, \psi(a_{m(q)}), \alpha(a_{m(q)}) \right]. \tag{15}$$

The compiler then wraps this branch in the selector $C(q) = \sigma_\vee[\chi(q), B_q]$, and the final tree is $\mathcal{T} = C(\gamma)$. The selector provides a target-check branch and an achievement branch for every goal or stable subgoal. Every method node first compiles its required stable subgoals, then checks the action preconditions, and finally emits the grounded action leaf.

### 3.6 Compilation Properties

**Proposition 1 (Well-formed compilation).** Under an acyclic repaired dependency graph and valid grounded action schemas, the compiler terminates and deterministically produces a schema-valid BT. Its action leaves are induced exclusively from the grounded actions in $\mathcal{M}$.

*Proof sketch.* Consider a topological order of the repaired dependency graph. A condition with no required stable subgoals compiles to a selector with a finite sequence branch. Each remaining condition expands only conditions earlier in this order, followed by its precondition and grounded-action leaves. Structural induction gives termination and preserves the BT grammar.

**Proposition 2 (Dependency representation).** If closure makes every required stable subgoal checkable or establishable before its associated action, the compiled BT represents these dependencies through explicit recursive subtrees and precondition checks.

*Proof sketch.* Equation 15 places every $C(r)$, for $r \in U_{m(q)}$, before the precondition check $\psi(a_{m(q)})$ and action $\alpha(a_{m(q)})$. The selector $C(q)$ first checks the target condition, reusing it when already satisfied and otherwise executing this ordered branch. Appendix B defines the nodes and leaves.

**Table 2:** Assembly results with shared Predictor candidates and repair (P+R).

| Backbone | Method | Exec | LC | SR | Sec. |
|---|---|---|---|---|---|
| Qwen3 | Direct BT | 60/60 | 31/60 | 29/60 | 22.54 |
| Qwen3 | ActionSeq | 60/60 | 37/60 | 37/60 | 3.17 |
| Qwen3 | HiBT | 60/60 | 60/60 | 60/60 | 0.45 |
| Qwen2.5 | Direct BT | 60/60 | 22/60 | 18/60 | 14.22 |
| Qwen2.5 | Direct BT + P+R | 60/60 | 32/60 | 32/60 | 14.25 |
| Qwen2.5 | ActionSeq | 60/60 | 43/60 | 45/60 | 2.62 |
| Qwen2.5 | ActionSeq + P+R | 60/60 | 36/60 | 36/60 | 2.91 |
| Qwen2.5 | LLM-BT | 60/60 | 54/60 | 54/60 | – |
| Qwen2.5 | HiBT | 60/60 | 60/60 | 60/60 | 0.35 |
| LLaMA-3 | Direct BT | 58/60 | 26/60 | 24/60 | 26.84 |
| LLaMA-3 | ActionSeq | 60/60 | 25/60 | 25/60 | 3.11 |
| LLaMA-3 | HiBT | 60/60 | 60/60 | 60/60 | 0.37 |
| Mistral | Direct BT | 40/60 | 23/60 | 19/60 | 28.53 |
| Mistral | ActionSeq | 34/60 | 17/60 | 18/60 | 3.60 |
| Mistral | HiBT | 60/60 | 60/60 | 60/60 | 0.39 |

**Table 3:** Realization ablations on 60 assembly tasks.

| Backbone | Variant | Exec | LC | SR | Sec. |
|---|---|---|---|---|---|
| Qwen3 | No repair | 60/60 | 24/60 | 24/60 | 0.40 |
| Qwen3 | No stable subgoals | 60/60 | 38/60 | 38/60 | 0.85 |
| Qwen3 | HiBT | 60/60 | 60/60 | 60/60 | 0.45 |
| Qwen2.5 | No repair | 60/60 | 38/60 | 38/60 | 0.67 |
| Qwen2.5 | No stable subgoals | 60/60 | 45/60 | 45/60 | 0.74 |
| Qwen2.5 | HiBT | 60/60 | 60/60 | 60/60 | 0.35 |
| LLaMA-3 | No repair | 60/60 | 13/60 | 13/60 | 0.72 |
| LLaMA-3 | No stable subgoals | 60/60 | 30/60 | 30/60 | 0.74 |
| LLaMA-3 | HiBT | 60/60 | 60/60 | 60/60 | 0.37 |
| Mistral | No repair | 60/60 | 25/60 | 25/60 | 0.73 |
| Mistral | No stable subgoals | 60/60 | 30/60 | 30/60 | 0.75 |
| Mistral | HiBT | 60/60 | 60/60 | 60/60 | 0.39 |

# 4 Symbolic Assembly Experiments

## 4.1 Task Setting and Benchmark

We evaluate language-conditioned BT synthesis on 60 symbolic assembly tasks. Each instance specifies a goal and initial world state; the generated BT is executed in a shared symbolic simulator. Tasks cover Gearset, Gearset-1, Chair, and Lamp, spanning easy through very-hard dependency structures. Appendix A gives the task distribution and difficulty partition. Embodied transfer is evaluated separately in Section 5, where the interface guides a VLA carrier in LIBERO.

## 4.2 Protocol and Metrics

We compare Direct BT, ActionSeq, and HiBT across Qwen3-8B, Qwen2.5-7B-Instruct, LLaMA-3-8B-Instruct, and Mistral-7B-Instruct v0.3. Direct BT generates a tree in one step; ActionSeq first generates an action sequence; HiBT is the full hierarchical pipeline. Two realization variants isolate repair and stable-subgoal construction: no-repair disables symbolic repair; no-stable-subgoal retains repair but omits stable subgoals.

Builder supervision, Predictor training, and prototype-bank construction all exclude evaluation-task families, including their subtask, subset, and already-satisfied variants. This family-level separation applies to every HiBT configuration. All runs use the concept geometry in Section 3; Appendix J.2 gives the filtering procedure.

We report four metrics. Exec measures whether a generated BT can be parsed, compiled, and stepped by the shared symbolic runtime without structural failure. LC additionally requires consistency between the reported action sequence and the action leaves extracted from the compiled BT, and rejects executions terminating at a precondition or target-failure node. SR measures whether symbolic execution reaches the requested target state. Generation time measures planning latency, excluding model loading and simulation-time evaluation. Given a fixed Predictor checkpoint, prototype bank, decoding configuration, and symbolic runtime, planning is deterministic. Wilson intervals quantify uncertainty over the 60 test instances. LIBERO evaluation measures both symbolic plan validity and embodied completion, with rollout efficiency quantified by steps and policy calls.

## 4.3 Main Results

HiBT solves all 60 assembly tasks with executable, logically coherent plans across the four backbones (Table 2). ActionSeq improves on direct tree generation in several settings, but its flat decomposition leaves a larger gap in coherence and task completion. HiBT also has the lowest planning latency for every backbone, ranging from 0.35 to 0.45 seconds.

The Qwen2.5-7B comparison separates the value of hierarchical organization from access to planning candidates and repair. HiBT outperforms both matched flat controls with the same Predictor

**Figure 3:** Qwen2.5-7B success: (a) Wilson 95% intervals; (b) accuracy–latency trade-off.

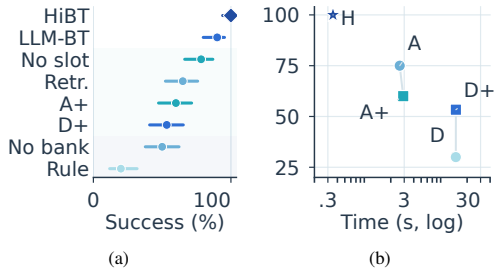



**Table 4:** Interface ablations on 60 assembly tasks. Intervals are Wilson 95% CIs.

| Variant | SR | CI (%) |
|---|---|---|
| Rule only | 12/60 | [11.8, 31.8] |
| Random | 18/60 | [19.9, 42.5] |
| No bank | 30/60 | [37.7, 62.3] |
| Classifier | 37/60 | [49.0, 72.9] |
| Retrieval only | 39/60 | [52.4, 75.8] |
| No slot retrieval | 47/60 | [66.4, 86.9] |
| Shuffled | 53/60 | [77.8, 94.2] |
| **HiBT** | **60/60** | **[94.0, 100.0]** |

In Figure 3, D denotes Direct BT, A denotes ActionSeq, H denotes HiBT, and + denotes shared Predictor candidates and repair (P+R). Template selection supplies coarse structure, while slot retrieval grounds that structure in the current task. Their successive gains, together with the drop under hierarchy shuffling, show that both levels of organization contribute to the complete interface. These results connect task-specific hierarchy and slot grounding to executable planning, complementing the matched flat controls in Table 2.

candidates and repair backend, as well as LLM-BT. The hierarchy therefore contributes beyond the shared realization components. Appendix E gives the domain breakdown; the Wilson 95% interval for full success is [94.0, 100.0]%.

### 4.4 Interface Evidence

The Qwen2.5-7B controls identify the planning signal supplied by hierarchy, templates, and grounded slots. Table 4 summarizes these comparisons; Appendix C.2 specifies each variant. Figure 3 relates interface quality to success and the accuracy–latency trade-off.

### 4.5 Difficulty Results

To examine how performance differences vary with task difficulty, we aggregate results across the four backbones within each bucket. HiBT attains 100% LC and SR in every bucket, while the largest baseline gaps occur on hard and very-hard instances, where tool switching and interacting subgoals place greater demands on dependency preservation. The complete breakdown, including Exec and latency, appears in Appendix Table 7.

### 4.6 Realization Ablations

Table 3 isolates dependency repair and stable-subgoal construction. Removing repair preserves structural executability but reduces LC and SR to 13/60–38/60 across the four backbones. Stable-subgoal removal also lowers both metrics. The gains from each component show how symbolic realization converts learned planning structure into coherent execution.

## 5 Embodied Transfer Experiments

### 5.1 LIBERO Evaluation

We next evaluate the same planning interface in LIBERO manipulation rollouts using a pi0.5/OpenPI low-level carrier. HiBT supplies dependency-aware subgoals and BT structure; the VLA policy executes low-level actions from visual observations. Each LIBERO-Long method is evaluated on 10 tasks with 120 trials per task. The adapter reads each task's language, object inventory, initial predicates, and goal predicates, then expresses placement, spatial, articulation, and appliance-state goals through the same hierarchical interface. The resulting subgoals become natural-language prompts for the fixed visual policy.

Flat instruction supplies the original task request throughout execution. ActionSeq converts extracted symbolic actions into sequential subgoal prompts. HiBT uses dependency-aware subgoals with

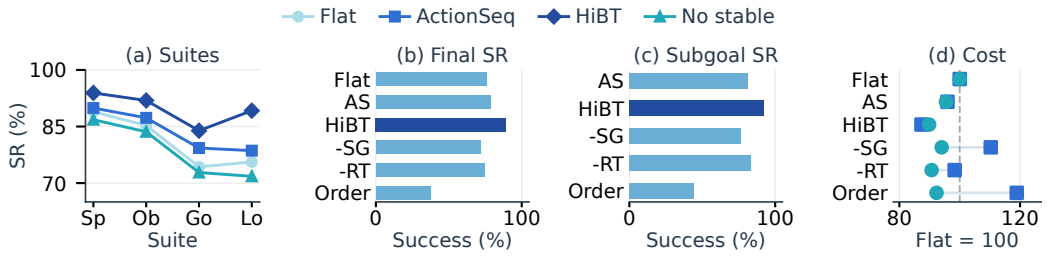



**Figure 4:** LIBERO rollout summary, 1200 episodes per suite and method. Sp/Ob/Go/Lo: Spatial/Object/Goal/Long; AS: ActionSeq; -SG/-RT: no stable checks/no retry; Order: wrong-order control. Cost markers denote steps (squares) and policy calls (circles), normalized to Flat.

**Table 5:** Complete LIBERO-Long rollout table. Bold marks the best value among Flat instruction, Action sequence, and HiBT, excluding manual and diagnostic controls.

| Method | Interface | SR ↑ | Subgoal SR ↑ | Steps ↓ | Policy Calls ↓ | Plan ms ↓ |
|---|---|---|---|---|---|---|
| *Reference methods* | | | | | | |
| Flat instruction | none | 907/1200 (75.6) | N/A | 418 | 98.7 | N/A |
| Action sequence | flat action/subgoal list | 943/1200 (78.6) | 81.3 | 401 | 94.2 | 1760 |
| Manual subgoal upper | BDDL/manual subgoals | 1089/1200 (90.8) | 93.4 | 368 | 97.8 | N/A |
| *HiBT variants and controls* | | | | | | |
| **HiBT** | learned concepts + BT + repair | **1070/1200 (89.2)** | **92.1** | **365** | **88.7** | **198** |
| HiBT without stable | concepts + sequence only | 861/1200 (71.8) | 76.4 | 461 | 92.8 | 186 |
| HiBT without retry | concepts + no fallback | 896/1200 (74.7) | 83.1 | 411 | 89.5 | 184 |
| Wrong-order control | shuffled subgoals | 447/1200 (37.3) | 44.2 | 497 | 91.1 | 181 |

stable-condition checks and retry/fallback branches. When a task combines placement and final state-changing actions, repaired guidance orders the placement subgoals before the final state change.

All rollout variants share camera preprocessing, a 650-step episode horizon, a 180-step stage budget, and replanning every five steps. Final and subgoal success measure goal attainment and intermediate-condition completion; steps and policy calls measure execution cost. Appendix D.2 gives the complete run matrix and metric definitions.

Table 5 shows that HiBT reaches 89.2% final-goal success, exceeding both flat instruction and action-sequence decomposition. The gain accompanies shorter rollouts and fewer policy calls, linking subgoal structure to both completion and execution efficiency. In planning-only checks, HiBT matches the oracle pyramid on all 40 LIBERO tasks (Appendix D.2). Figure 5 contrasts flat and hierarchical guidance; Figure 7 shows a successful task-3 flat-policy rollout.

### 5.2 Cross-Setting Analysis

The custom LIBERO-Hier suite tests whether final success coincides with process constraints such as ordering and distractor control. As shown in Table 6, HiBT achieves 72.3% strict success, a 30-point improvement over ActionSeq. The gain extends from reaching the final state to following the required execution process. Figure 6 compares the corresponding ablations and constraint-sensitive outcomes.

Figure 6 separates learned structure from execution support under a shared LIBERO-Long carrier. Learned concepts outperform rule-only, random, and shuffled guidance; removing stable checks or retries also reduces success. The wrong-order control gives the largest drop, linking these gains to preservation of task-specific dependencies during execution.

The LIBERO-Hier results in Table 6 extend this analysis to explicit process constraints. HiBT has a smaller gap between final and strict success than either flat baseline. Together with its higher order validity, this pattern connects final-goal gains to improved compliance with the specified execution process. Appendix Table 15 provides the complete constraint and distractor-displacement metrics.

Across both settings, the gains track dependency preservation. Assembly controls isolate the benefit of hierarchical structure beyond parseable outputs and shared repair candidates. Under a fixed LIBERO carrier, stable checks and retries turn that structure into successful, shorter rollouts, while LIBERO-Hier shows improved compliance with process constraints. The compilation properties

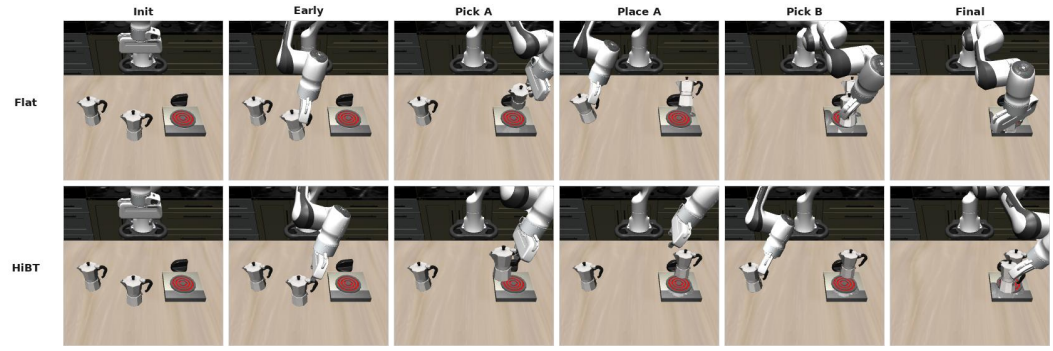



**Figure 5:** Task-8 LIBERO-Long process comparison. The flat-instruction rollout and HiBT-guided rollout are shown with aligned phase samples, illustrating how hierarchical subgoal guidance changes the execution trajectory on the same long-horizon manipulation task.

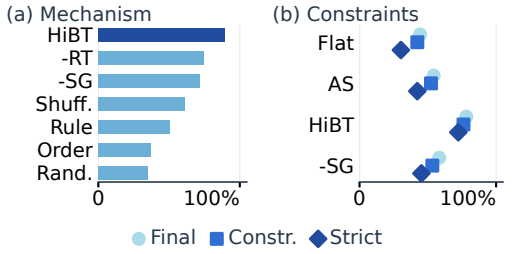



**Figure 6:** LIBERO-Long ablations (a) and LIBERO-Hier success (b), with 1200 episodes per variant. Stable checks and retries improve completion; strict success also requires the specified process constraints.

**Table 6:** LIBERO-Hier success and order validity (%, 1200 episodes). Full metrics: Appendix Table 15.

| Method | Final | Strict | Order |
|---|---|---|---|
| Flat | 44.3 | 30.3 | 58.4 |
| ActionSeq | 54.3 | 42.3 | 68.3 |
| Manual upper | 81.9 | 75.9 | 94.1 |
| **HiBT** | **78.1** | **72.3** | **87.4** |
| No stable | 58.3 | 45.3 | 69.3 |

Figure 6 uses the labels in Figure 4; Shuff./Rand. denote shuffled/random concepts. Strict success requires both the final goal and process constraints. The improvement in strict success is accompanied by higher order validity, linking stable-subgoal checks to process compliance.

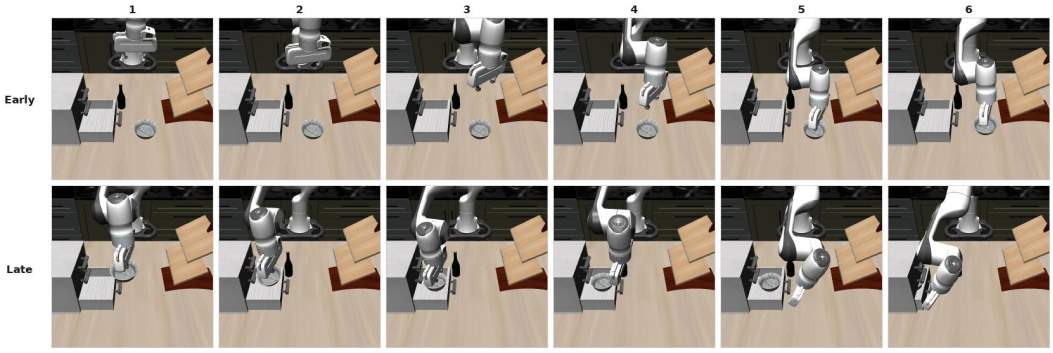



**Figure 7:** LIBERO-Long task-3 rollout under flat-instruction guidance. Early and late frames show successive manipulation stages from the initial state to completion, complementing the task-8 comparison in Figure 5.

connect these results to the method: repaired dependencies are retained as explicit checks and ordered subtrees in the executable policy.

## 6 Conclusion

HiBT formulates BT synthesis as hierarchical concept planning: learned task structure is grounded, repaired, and compiled into executable control. On 60 assembly tasks, HiBT improves logical coherence, target achievement, and planning latency over direct-generation and action-sequence baselines. Controls and interface perturbations isolate complementary gains from hierarchy, prototype grounding, and symbolic realization. LIBERO rollouts link dependency-aware subgoals to improved completion and process compliance. Hierarchical concept planning thus connects learned task decomposition with executable robot control.

## AI USE STATEMENT

OpenAI Codex assisted with manuscript editing, LaTeX formatting, and the organization of supplementary implementation details. The added implementation descriptions were checked against the existing manuscript, configuration files, and source code. This revision did not introduce new experimental measurements. The authors are responsible for the content of the submission.

## REPRODUCIBILITY STATEMENT

The task-family exclusion protocol and symbolic definitions are specified in Appendices A and B. Appendices D and H give the metrics, execution settings, and training procedure; Appendix J details the serialized interface, baseline prompt contracts, and training configuration.

## ETHICS STATEMENT

This work studies language-conditioned BT synthesis and robotic execution using simulation, symbolic task states, and offline demonstration data. It does not involve human-subject data, private personal information, or deployment on physical users. Nevertheless, HiBT can alter the behavior of a base execution policy by changing high-level action ordering and recovery structure. Any deployment beyond simulation should therefore include explicit checks on correction magnitude, workspace safety, tool changes, object-release events, and gripper state transitions before physical execution.

## REFERENCES


Ido Amos, Avi Caciularu, Mor Geva, Amir Globerson, Jonathan Herzig, Lior Shani, and Idan Szpektor. Latent Reasoning With Supervised Thinking States, 2026.

Jicong Ao, Fan Wu, Yansong Wu, Abdalla Swiki, and Sami Haddadin. LLM-as-BT-Planner: Leveraging LLMs For Behavior Tree Generation In Robot Task Planning. In *2025 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 1233–1239. IEEE, 2025. doi: 10.1109/ICRA55743.2025.11128454.

Bikramjit Banerjee. Autonomous Acquisition Of Behavior Trees For Robot Control. In *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 3460–3467. IEEE Press, 2018. doi: 10.1109/IROS.2018.8594083.

Wenhu Chen, Xueguang Ma, Xinyi Wang, and William W. Cohen. Program Of Thoughts Prompting: Disentangling Computation From Reasoning For Numerical Reasoning Tasks. *Transactions on Machine Learning Research*, October 2023a.

Xinglin Chen, Yishuai Cai, Yunxin Mao, Minglong Li, Wenjing Yang, Weixia Xu, and Ji Wang. Integrating Intent Understanding And Optimal Behavior Planning For Behavior Tree Generation From Human Instructions. In *Proceedings of the Thirty-Third International Joint Conference on Artificial Intelligence*, pp. 6832–6840, 2024. doi: 10.24963/ijcai.2024/755.

Yaran Chen, Wenbo Cui, Yuanwen Chen, Mining Tan, Xinyao Zhang, Dongbin Zhao, and He Wang. RoboGPT: An Intelligent Agent Of Making Embodied Long-Term Decisions For Daily Instruction Tasks, 2023b.

Michele Colledanchise, Ramviyas Parasuraman, and Petter Ogren. Learning Of Behavior Trees For Autonomous Agents. *IEEE Transactions on Games*, 11(2):183–189, June 2019. ISSN 2475-1510. doi: 10.1109/TG.2018.2816806.

Danny Driess, Fei Xia, Mehdi S. M. Sajjadi, Corey Lynch, Aakanksha Chowdhery, Brian Ichter, Ayzaan Wahid, Jonathan Tompson, Quan Vuong, Tianhe Yu, Wenlong Huang, Yevgen Chebotar, Pierre Sermanet, Daniel Duckworth, Sergey Levine, Vincent Vanhoucke, Karol Hausman, Marc Toussaint, Klaus Greff, Andy Zeng, Igor Mordatch, and Pete Florence. PaLM-E: An Embodied Multimodal Language Model. In *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pp. 8469–8488. PMLR, 2023.

Kutluhan Erol, James A. Hendler, and Dana S. Nau. HTN Planning: Complexity and Expressivity. In *Proceedings of the Twelfth AAAI Conference on Artificial Intelligence*, pp. 1123–1128. AAAI Press, 1994.

Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. PAL: Program-aided Language Models. In *Proceedings of the 40th International Conference on Machine Learning*, pp. 10764–10799. PMLR, 2023.

Runquan Gui, Zhihai Wang, Jie Wang, Chi Ma, Huiling Zhen, Mingxuan Yuan, Jianye Hao, Defu Lian, Enhong Chen, and Feng Wu. HyperTree Planning: Enhancing LLM Reasoning Via Hierarchical Thinking, 2025.

Shibo Hao, Yi Gu, Haodi Ma, Joshua Hong, Zhen Wang, Daisy Wang, and Zhiting Hu. Reasoning With Language Model Is Planning With World Model. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 8154–8173, Singapore, December 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.emnlp-main.507.

Rishi Hazra, Pedro Zuidberg Dos Martires, and Luc De Raedt. SayCanPay: Heuristic Planning With Large Language Models Using Learnable Domain Knowledge. *Proceedings of the AAAI Conference on Artificial Intelligence*, 38(18):20123–20133, 2024. doi: 10.1609/aaai.v38i18.29991.

Xingshuai Huang, Derek Li, Bahareh Nikpour, and Parsa Omidi. Hierarchical Chain-of-Thought Prompting: Enhancing LLM Reasoning Performance And Efficiency, 2026.

Brian Ichter, Anthony Brohan, Yevgen Chebotar, Chelsea Finn, Karol Hausman, Alexander Herzog, Daniel Ho, Julian Ibarz, Alex Irpan, Eric Jang, Ryan Julian, Dmitry Kalashnikov, Sergey Levine, Yao Lu, Carolina Parada, Kanishka Rao, Pierre Sermanet, Alexander T. Toshev, Vincent Vanhoucke, Fei Xia, Ted Xiao, Peng Xu, Mengyuan Yan, Noah Brown, Michael Ahn, Omar Cortes, Nicolas Sievers, Clayton Tan, Sichun Xu, Diego Reyes, Jarek Rettinghouse, Jornell Quiambao, Peter Pastor, Linda Luu, Kuang-Huei Lee, Yuheng Kuang, Sally Jesmonth, Nikhil J. Joshi, Kyle Jeffrey, Rosario Jauregui Ruano, Jasmine Hsu, Keerthana Gopalakrishnan, Byron David, Andy Zeng, and Chuyuan Kelly Fu. Do As I Can, Not As I Say: Grounding Language In Robotic Affordances. In *Proceedings of The 6th Conference on Robot Learning*, volume 205 of *Proceedings of Machine Learning Research*, pp. 287–318. PMLR, 2023.

Matteo Iovino, Edvards Scukins, Jonathan Styrud, Petter Ögren, and Christian Smith. A Survey Of Behavior Trees In Robotics And AI, 2020.

Matteo Iovino, Julian Förster, Pietro Falco, Jen Jen Chung, Roland Siegwart, and Christian Smith. On The Programming Effort Required To Generate Behavior Trees And Finite State Machines For Robotic Applications. In *2023 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 5807–5813. IEEE, 2023. doi: 10.1109/ICRA48891.2023.10160972.

Riccardo Andrea Izzo, Gianluca Bardaro, and Matteo Matteucci. BTGenBot: Behavior Tree Generation For Robotic Tasks With Lightweight LLMs. In *2024 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 9684–9690. IEEE, October 2024. doi: 10.1109/IROS58592.2024.10802304.

Leslie Pack Kaelbling and Tomás Lozano-Pérez. Hierarchical Task and Motion Planning in the Now. In *2011 IEEE International Conference on Robotics and Automation*, pp. 1470–1477. IEEE, 2011. doi: 10.1109/ICRA.2011.5980391.

Tushar Khot, Harsh Trivedi, Matthew Finlayson, Yao Fu, Kyle Richardson, Peter Clark, and Ashish Sabharwal. Decomposed Prompting: A Modular Approach For Solving Complex Tasks. In *The Eleventh International Conference on Learning Representations*, 2023.

Shen Li, Daehyung Park, Yoonchang Sung, Julie A. Shah, and Nicholas Roy. Reactive Task And Motion Planning Under Temporal Logic Specifications, 2021.

Jacky Liang, Wenlong Huang, Fei Xia, Peng Xu, Karol Hausman, Brian Ichter, Pete Florence, and Andy Zeng. Code As Policies: Language Model Programs For Embodied Control. In *2023 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 9493–9500. IEEE, 2023. doi: 10.1109/ICRA48891.2023.10160591.

Bo Liu, Yuqian Jiang, Xiaohan Zhang, Qiang Liu, Shiqi Zhang, Joydeep Biswas, and Peter Stone. LLM+P: Empowering Large Language Models With Optimal Planning Proficiency, 2023.

Artem Lykov and Dzmitry Tsetserukou. LLM-BRAIn: AI-driven Fast Generation Of Robot Behaviour Tree Based On Large Language Model, 2023.

Qing Lyu, Shreya Havaldar, Adam Stein, Li Zhang, Delip Rao, Eric Wong, Marianna Apidianaki, and Chris Callison-Burch. Faithful Chain-of-Thought Reasoning. In *Proceedings of the 13th International Joint Conference on Natural Language Processing and the 3rd Conference of the Asia-Pacific Chapter of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 305–329. Association for Computational Linguistics, 2023.

Drew McDermott, Malik Ghallab, Adele Howe, Craig Knoblock, Ashwin Ram, Manuela Veloso, Daniel Weld, and David Wilkins. PDDL—The Planning Domain Definition Language. Technical Report CVC TR-98-003, Yale Center for Computational Vision and Control, 1998.

Sergio Merino-Fidalgo, Celia Sánchez-Girón, Eduardo Zalama, Jaime Gómez-García-Bermejo, and Jaime Duque-Domingo. Behavior Tree Generation And Adaptation For A Social Robot Control With LLMs. *Robotics and Autonomous Systems*, 194:105165, 2025. ISSN 0921-8890. doi: 10.1016/j.robot.2025.105165.

Krishan Rana, Jesse Haviland, Sourav Garg, Jad Abou-Chakra, Ian Reid, and Niko Suenderhauf. SayPlan: Grounding Large Language Models Using 3D Scene Graphs For Scalable Robot Task Planning. In *Proceedings of The 7th Conference on Robot Learning*, volume 229 of *Proceedings of Machine Learning Research*, pp. 23–72. PMLR, 2023.

Xuan Shen, Yizhou Wang, Yufa Zhou, Xiangxi Shi, Pu Zhao, Yanzhi Wang, and Jiuxiang Gu. Efficient Reasoning With Hidden Thinking, 2025.

Ishika Singh, Valts Blukis, Arsalan Mousavian, Ankit Goyal, Danfei Xu, Jonathan Tremblay, Dieter Fox, Jesse Thomason, and Animesh Garg. ProgPrompt: Program Generation For Situated Robot Task Planning Using Large Language Models. *Autonomous Robots*, 47(8):999–1012, August 2023. ISSN 0929-5593. doi: 10.1007/s10514-023-10135-3.

Chan Hee Song, Jiaman Wu, Clayton Washington, Brian M. Sadler, Wei-Lun Chao, and Yu Su. LLM-Planner: Few-Shot Grounded Planning For Embodied Agents With Large Language Models. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, pp. 2998–3009, October 2023.

DiJia Su, Hanlin Zhu, Yingchen Xu, Jiantao Jiao, Yuandong Tian, and Qinqing Zheng. Token Assorted: Mixing Latent And Text Tokens For Improved Language Model Reasoning, 2025.

Lei Wang, Wanyu Xu, Yihuai Lan, Zhiqiang Hu, Yunshi Lan, Roy Ka-Wei Lee, and Ee-Peng Lim. Plan-and-Solve Prompting: Improving Zero-Shot Chain-of-Thought Reasoning By Large Language Models. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 2609–2634. Association for Computational Linguistics, 2023a.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-Consistency Improves Chain Of Thought Reasoning In Language Models. In *The Eleventh International Conference on Learning Representations*, 2023b.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. Chain-of-Thought Prompting Elicits Reasoning In Large Language Models. In *Advances in Neural Information Processing Systems*, volume 35, pp. 24824–24837, 2022.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik Narasimhan. Tree Of Thoughts: Deliberate Problem Solving With Large Language Models, 2023a.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing Reasoning And Acting In Language Models. In *The Eleventh International Conference on Learning Representations*, 2023b.

Denny Zhou, Nathanael Schärli, Le Hou, Jason Wei, Nathan Scales, Xuezhi Wang, Dale Schuurmans, Claire Cui, Olivier Bousquet, Quoc Le, and Ed Chi. Least-to-Most Prompting Enables Complex Reasoning In Large Language Models. In *The Eleventh International Conference on Learning Representations*, 2023.

Haotian Zhou, Yunhan Lin, Longwu Yan, Jihong Zhu, and Huasong Min. LLM-BT: Performing Robotic Adaptive Tasks Based On Large Language Models And Behavior Trees. In *2024 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 16655–16661. IEEE, May 2024. doi: 10.1109/ICRA57147.2024.10610183.

# A Benchmark Details

The 60-instance benchmark emphasizes dependency-heavy tasks. It contains 44 Gearset-1 instances, one Gearset instance, nine Chair instances, and six Lamp instances. Difficulty buckets contain 22 easy, 13 medium, 11 hard, and 14 very-hard tasks. The difficulty labels are used only as analysis buckets and are not given to the planner. For all HiBT-family evaluations on the 60-instance benchmark, the prototype bank, Builder concept extraction data, and Predictor training data are filtered with family-level exclusion. The pipeline first maps each candidate to a normalized base problem ID and then excludes any item whose normalized family overlaps with the benchmark request set. This filtering also covers derived forms such as subtasks, subsets, and already-satisfied variants, so that retrieval and training do not reuse near-duplicates of the evaluation problems.

# B Formal Details

This section expands the formal definitions used in Section 3 and provides the three algorithms used by HiBT. The notation follows the symbolic conventions in the main text: $\rho_{\mathcal{D}}$ denotes initial realization, $\kappa_{\mathcal{D}}$ denotes dependency closure, $\zeta_{\mathcal{D}}$ denotes single-predicate repair, $\Lambda$ denotes lifting, and $C$ denotes deterministic BT compilation.

## B.1 Symbolic Assembly Domain

The symbolic assembly domain is $\mathcal{D} = (\mathcal{Q}, \mathcal{A}, \mathcal{P}, \mathcal{E}^+, \mathcal{E}^-)$, where $\mathcal{Q}$ contains grounded symbolic predicates over tool occupancy, object grasp, tool availability, and assembly relations, and $\mathcal{A}$ contains the grounded manipulation actions in the domain action inventory. For action $a$, the sets $P_a$, $E_a^+$, and $E_a^-$ denote preconditions, positive effects, and negative effects. The transition function applies only when the action preconditions are satisfied:

$$\begin{aligned} \delta(s,a) = \left(s \setminus E_a^-\right) \cup E_a^+, \\ P_a \subseteq s. \end{aligned} \tag{16}$$

This domain definition separates structural executability from goal achievement, allowing both properties to be evaluated under the same transition model.

## B.2 Plan Pyramid and Graph

The repaired plan pyramid is

$$\begin{aligned} P = (\gamma, \mathcal{S}, \mathcal{M}, \pi), \\ m_k = (q_k, a_k, U_k), \end{aligned} \tag{17}$$

where $\gamma$ is the final target predicate, $\mathcal{S}$ is the stable subgoal set, $\mathcal{M}$ is the set of grounded method nodes, and $\pi$ is a linearized action sequence retained for reporting and action-leaf consistency evaluation. The dependency graph is

$$\mathcal{G}_P = (V_P, E_P), \qquad V_P = \{\gamma\} \cup \mathcal{S}, \tag{18}$$

with $(r, q) \in E_P$ if and only if $r \in U_{m(q)}$. The compiler assumes this dependency graph is acyclic. Stable subgoals are conditions that can be checked or re-established under repeated BT ticking, while method nodes encode the grounded action that achieves a goal or stable subgoal and the support conditions required by that action. For a condition $q \in V_P$, let

$$m(q) = \arg \min_{m_k \in \mathcal{M}: q_k = q} k, \qquad U(q) = U_{m(q)}. \tag{19}$$

If multiple candidate method nodes achieve the same predicate, this selector uses the first repaired method in the realized sequence; unresolved ties are therefore fixed before BT compilation. The induced partial order is

$$r \prec_P q \quad \Longleftrightarrow \quad (r, q) \in E_P^+, \tag{20}$$

where $E_P^+$ is the transitive closure of $E_P$. Acyclicity of $\mathcal{G}_P$ is the condition that makes this order well defined and allows the recursive compiler in Algorithm 3 to terminate.

### B.3 Concept Training

The concept geometry is fixed across experiments:

$$\begin{aligned} \mathbf{Z}^{(\ell)} &\in \mathbb{R}^{n_\ell \times d}, \\ \mathbf{n} &= (n_0, n_1, n_2, n_3), \\ N &= \sum_{\ell=0}^{3} n_\ell. \end{aligned} \tag{21}$$

In this work, $\mathbf{n} = (1, 4, 5, 5)$ and $N = 15$. The Builder is trained from annotated solution encodings using weighted reconstruction, slot-ordering, residual, and teacher-forced reasoning losses, with respective weights 0.1, 0.5, 1.0, and 0.5. The resulting Builder is frozen before it supplies aligned concept targets to the Predictor. Builder supervision produces target pyramids $\mathbf{C}^* = h_\phi(u, s_0, P^*)$, and the Predictor produces $\hat{\mathbf{Z}} = f_\theta(u, s_0)$. In this instantiation, concept training uses per-level MSE:

$$\mathcal{L}_Z = \frac{1}{4} \sum_{\ell=0}^{3} \frac{1}{n_\ell} \sum_{i=1}^{n_\ell} \left\| \hat{\mathbf{z}}_i^{(\ell)} - \mathbf{c}_i^{*(\ell)} \right\|_2^2. \tag{22}$$

The auxiliary token loss is teacher-forced cross entropy on the annotated solution sequence $\mathbf{y}$:

$$\mathcal{L}_\pi = -\sum_{t=1}^{|\mathbf{y}|} \log p_\theta \left( y_t \mid u, s_0, \mathbf{C}^*, y_{<t} \right). \tag{23}$$

The total loss is $\mathcal{L} = \lambda_Z \mathcal{L}_Z + \lambda_\pi \mathcal{L}_\pi$, where $\mathcal{L}_Z$ and $\mathcal{L}_\pi$ correspond to the concept reconstruction and symbolic-sequence losses above. At inference time, HiBT uses the predicted concept pyramid directly for interface decoding.

### B.4 Interface Decoding

The prototype bank is

$$\mathcal{B} = \{ b_j = (\mathbf{C}_j, \tau_j, \Sigma_j) \}_{j=1}^{M}. \tag{24}$$

Each entry stores a Builder concept pyramid, a structural template, and the associated annotated plan schema. Family-level exclusion is applied before prototype-bank construction and concept-model data preparation, preventing evaluation-family near-duplicates from appearing in either retrieval entries or concept supervision data. Template matching computes

$$s_\tau(b_j) = \frac{1}{4} \sum_{\ell=0}^{3} \cos \left( \mathbf{v}(\hat{\mathbf{Z}}^{(\ell)}), \mathbf{v}(\mathbf{C}_j^{(\ell)}) \right), \tag{25}$$

selects $j^* = \arg\max_j s_\tau(b_j)$, and sets $\tau^* = \tau_{j^*}$. Slot-level retrieval selects the nearest symbolic item within the chosen template:

$$r_i^{(\ell)} = \arg \max_{p \in \mathcal{B}_i^{(\ell)}(\tau^*)} \cos\big(\hat{\mathbf{z}}_i^{(\ell)}, \mathbf{c}(p)\big), \tag{26}$$

yielding $\Xi = (\tau^*, \mathcal{R})$ with $\mathcal{R} = \{r_i^{(\ell)}\}_{\ell,i}$.

### B.5 Dependency Closure

Dependency closure is defined over action preconditions. For an action $a$ and state $s$,

$$\Delta(a, s) = P_a \setminus s. \tag{27}$$

If this set is non-empty, the closure operator applies domain-supported repair rules that establish missing predicates before the action is appended to the repaired sequence. The implementation uses deterministic rule dispatch rather than open-ended search: missing hand, tool, and part-support predicates are mapped to the supported release, detach, attach, grasp, and assembly-support steps of the symbolic assembly domain. The map $\nu$ selects the next missing predicate according to this

fixed repair-rule priority. The closure process terminates when the sequence can be simulated without missing preconditions or when no supported repair rule exists for a missing predicate; the latter case is counted as a logical-coherence or task-success failure during evaluation. More explicitly, for a candidate action sequence $\pi_0 = \langle a_k \rangle_{k=1}^{K}$, closure maintains a state sequence $s^{(t)}$ and a repaired sequence prefix $\pi^{*(t)}$. When processing action $a_k$, it repeatedly applies

$$\begin{aligned} q_t &= \nu\Big(P_{a_k} \setminus s^{(t)}\Big), \\ \bar{\pi}_t &= \zeta_{\mathcal{D}}(q_t, s^{(t)}), \\ s^{(t+1)} &= \delta^*(s^{(t)}, \bar{\pi}_t), \end{aligned} \tag{28}$$

until $P_{a_k} \subseteq s^{(t)}$, where $\delta^*$ denotes repeated application of the one-step transition in Eq. 16. The original action is then appended and simulated:

$$\pi^{*(t+1)} = \pi^{*(t)} \circ \langle a_k \rangle, \qquad s^{(t+1)} = \delta(s^{(t)}, a_k). \tag{29}$$

Algorithm 2 gives the corresponding procedure.

### B.6 BT Compilation

The BT grammar used by the compiler is

$$\begin{aligned} \mathcal{T} ::= \sigma_{\vee}([\mathcal{T}_i]_{i=1}^{n}) \mid \sigma_{\wedge}([\mathcal{T}_i]_{i=1}^{n}) \\ \mid \chi(q) \mid \psi(a) \mid \alpha(a). \end{aligned} \tag{30}$$

Compilation recursively expands each target condition into a selector subtree whose achievement branch first establishes required stable subgoals and then executes the grounded action. It compiles the dependency graph rather than the linear action sequence, which is why repaired stable subgoals become explicit control-flow structure. The leaf semantics used by the symbolic evaluator are

$$\begin{aligned} \mathrm{eval}_s(\chi(q)) &= \text{success} \;\Leftrightarrow\; q \in s, \\ \mathrm{eval}_s(\psi(a)) &= \text{success} \;\Leftrightarrow\; P_a \subseteq s, \\ \mathrm{eval}_s(\alpha(a)) &= (\text{success}, \delta(s,a)) \quad \text{if } P_a \subseteq s. \end{aligned} \tag{31}$$

Selectors $\sigma_{\vee}$ return the first successful child, and sequences $\sigma_{\wedge}$ require all children to succeed in order. Thus the compiled subtree $C(q) = \sigma_{\vee}[\chi(q), B_q]$ first reuses an already-satisfied condition and otherwise executes the achievement branch defined by the repaired dependency graph.

### B.7 Formal Properties

**Proposition 1.** If each method node contains a valid grounded action schema and $\mathcal{G}_P$ is acyclic, then compilation terminates and returns a BT that contains only the allowed node types in the grammar above. The proof follows by structural induction over the acyclic dependency graph: leaves have no required stable subgoals and compile to one selector with a finite sequence branch; each internal condition compiles after its required subgoals, which are lower in the acyclic dependency order.

**Proposition 2.** If dependency closure ensures that every required stable subgoal of a method node can be checked or established before the corresponding action is executed, then the compiled BT represents those dependencies through explicit recursive subtrees and precondition checks. This follows because $C(q)$ recursively compiles all $r \in U_{m(q)}$ before inserting $\psi(a_{m(q)})$ and $\alpha(a_{m(q)})$.

### B.8 Algorithms

Algorithm 1 gives interface induction in the main text. The procedures below specify dependency-closed symbolic realization (Algorithm 2) and deterministic BT compilation (Algorithm 3).

**Algorithm 2** Symbolic Realization with Dependency Closure

**Input:** target predicate $\gamma$, initial state $s_0$, interface $\Xi$, domain $\mathcal{D}$
**Output:** repaired plan pyramid $\tilde{P}$

Instantiate candidate, $\pi_0 \leftarrow \rho_{\mathcal{D}}(\gamma, s_0, \Xi)$
$\pi^* \leftarrow \langle\rangle$
$s \leftarrow s_0$
**for** each $a \in \pi_0$ **do**
    $\Delta \leftarrow P_a \setminus s$
    **while** $\Delta \neq \emptyset$ **do**
        $q \leftarrow \nu(\Delta)$
        Add repair segment, $\bar{\pi} \leftarrow \zeta_{\mathcal{D}}(q, s)$
        **for** each $\bar{a} \in \bar{\pi}$ **do**
            append $\bar{a}$ to $\pi^*$
            $s \leftarrow \delta(s, \bar{a})$
        **end for**
        $\Delta \leftarrow P_a \setminus s$
    **end while**
    Commit original action, append $a$ to $\pi^*$
    $s \leftarrow \delta(s, a)$
**end for**
**return** $\tilde{P} \leftarrow \Lambda(\gamma, \pi^*)$

**Algorithm 3** Deterministic Plan-to-BT Compilation

**Input:** repaired plan pyramid $\tilde{P} = (\gamma, \mathcal{S}, \mathcal{M}, \pi)$
**Output:** executable Behavior Tree $\mathcal{T}$

**function** C($q$)
    $m(q) \leftarrow$ method node in $\mathcal{M}$ that achieves $q$
    $\mathcal{B}_q \leftarrow \langle\rangle$
    **for** each $r \in U_{m(q)}$ **do**
        append $C(r)$ to $\mathcal{B}_q$
    **end for**
    append $\psi(a_{m(q)})$ to $\mathcal{B}_q$
    append $\alpha(a_{m(q)})$ to $\mathcal{B}_q$
    **return** $\sigma_\vee[\chi(q), \sigma_\wedge[\mathcal{B}_q]]$
**end function**
**return** $\mathcal{T} \leftarrow C(\gamma)$

## C Ablation Interpretation

### C.1 Difficulty Breakdown

Table 7 reports results aggregated across the four backbones within each difficulty bucket; each denominator is the number of instances in that bucket multiplied by four. The difficulty breakdown links HiBT's gains to explicit support relations. Success remains consistent as dependencies grow, while both flat baselines lose accuracy on hard and very-hard tasks. Each bucket aggregates the same task instances across four backbones.

**Table 7:** Difficulty-bucket results aggregated across all four backbones.

| Bucket | Method | Exec | LC | SR | Time (s) |
|---|---|---|---|---|---|
| Easy | Direct BT | 80/88 | 36/88 | 36/88 | 18.63 |
| Easy | ActionSeq | 76/88 | 52/88 | 54/88 | 2.50 |
| Easy | HiBT | 88/88 | 88/88 | 88/88 | 0.41 |
| Medium | Direct BT | 48/52 | 35/52 | 30/52 | 22.43 |
| Medium | ActionSeq | 44/52 | 25/52 | 26/52 | 3.40 |
| Medium | HiBT | 52/52 | 52/52 | 52/52 | 0.35 |
| Hard | Direct BT | 40/44 | 19/44 | 14/44 | 27.37 |
| Hard | ActionSeq | 40/44 | 21/44 | 21/44 | 3.72 |
| Hard | HiBT | 44/44 | 44/44 | 44/44 | 0.41 |
| Very hard | Direct BT | 50/56 | 12/56 | 10/56 | 27.18 |
| Very hard | ActionSeq | 54/56 | 24/56 | 24/56 | 3.34 |
| Very hard | HiBT | 56/56 | 56/56 | 56/56 | 0.39 |

The interface controls in Table 8 attribute gains to task-conditioned structure: rule-only and random-concept inputs reach 12/60 and 18/60 success, while shuffling a learned hierarchy reaches 53/60. The realization ablations in Table 3 identify the complementary role of dependency completion and stable-subgoal construction. Together, the controls connect learned guidance to the support conditions checked by the compiled BT.

### C.2 Template and Interface Ablations

These Qwen2.5-7B controls use the same 60 tasks. **Retrieval-only** selects the nearest bank template before realization. **Classifier-to-template** predicts its ID from the goal and state. **No slot retrieval** retains concept prediction and template scoring, then supplies the selected template to realization without slot grounding. **No bank** supplies concept predictions to symbolic synthesis without prototype support. Table 8 includes these variants, rule-only realization, and random or shuffled concepts. Template selection alone leaves substantial room for improvement. Slot-level grounding closes this gap, while removing prototype support reduces success. The controls thus distinguish coarse structural selection from task-specific symbolic grounding.

**Table 8:** Interface evidence on the 60-instance benchmark with Qwen2.5-7B. Intervals are Wilson intervals over the 60 test instances.

| Variant | Interface signal | SR | Wilson 95% CI |
|---|---|---|---|
| Rule-only realization | none | 12/60 | [11.8, 31.8] |
| Random concepts | random hierarchy | 18/60 | [19.9, 42.5] |
| Concept without bank | learned, no bank | 30/60 | [37.7, 62.3] |
| Classifier-to-template | template ID | 37/60 | [49.0, 72.9] |
| Retrieval-only | nearest template | 39/60 | [52.4, 75.8] |
| No slot retrieval | template only | 47/60 | [66.4, 86.9] |
| Shuffled concepts | shuffled hierarchy | 53/60 | [77.8, 94.2] |
| **HiBT** | hierarchical + slots | **60/60** | **[94.0, 100.0]** |

## D Protocol Details

Exec, LC, and SR are computed by the same symbolic runtime. Exec checks that the generated tree can be parsed, compiled, and evaluated without structural failure. LC is stricter: it requires executability, action-sequence consistency between the reported action sequence and the action leaves extracted from the compiled BT, and no terminal precondition or target failure node. SR measures whether the simulator reaches the requested target state. Generation time measures planning latency. For HiBT, it includes concept prediction, template scoring, slot retrieval, symbolic synthesis/repair, and BT compilation; model loading and simulation-time evaluation are excluded for all methods.

### D.1 Compute Environment

Table 9 reports the compute environment used for the reported experiments. Generation and evaluation jobs are run with single-GPU execution unless otherwise noted; the multi-GPU server is used to schedule independent runs and repeated evaluations.

**Table 9:** Compute setup used for the reported experiments.

| Component | Configuration |
|---|---|
| Compute server | 8× NVIDIA RTX 5880 Ada GPUs, 48GB VRAM each |
| Execution mode | Single-GPU execution unless otherwise noted |

### D.2 LIBERO Extension

LIBERO evaluates the interface at two levels. Symbolic checks test the executability and coherence of plans extracted from benchmark specifications. MuJoCo rollouts test how the resulting subgoals guide a VLA carrier through embodied manipulation. Both evaluations use the same extracted goal structure. The adapter reads each LIBERO task language description together with the objects, fixtures, initial predicates, and goal predicates provided by the benchmark specification. It then normalizes goal predicates into manipulation primitives covering object placement, spatial relations, articulation state, and appliance state. The resulting request has the same role as an assembly instance in the main paper: it supplies a language-conditioned task, an initial symbolic state, and a target structure that can be passed through the HiBT interface. For planning-only evaluation, the generated BT or subgoal sequence is checked against this extracted symbolic structure. For rollout evaluation, the same high-level structure is converted into natural subgoal prompts for the VLA carrier.

The benchmark scope contains the four ten-task LIBERO suites: *LIBERO Spatial*, *LIBERO Object*, *LIBERO Goal*, and *LIBERO-10*, together with a custom long-horizon dependency stress suite, whose results are reported separately. Table 10 summarizes the metrics. Symbolic metrics match the assembly evaluation; rollout metrics assess embodied execution.

**Table 10:** LIBERO metrics for symbolic planning and embodied execution.

| Layer | Metric | Meaning |
|---|---|---|
| | | *Planning-only* |
| Symbolic | Exec | Generated structure can be parsed and checked. |
| Symbolic | LC | Action leaves and reported plan remain logically coherent. |
| Symbolic | Symbolic SR | Extracted symbolic goal predicates are satisfied. |
| Symbolic | Stable SG | Number of explicit stable subgoals in the plan. |
| Symbolic | Plan ms | Planning latency before embodied rollout. |
| | | *VLA rollout* |
| Embodied | SR | Simulator episode reaches the benchmark success condition. |
| Embodied | Subgoal SR | Planned subgoals are completed during rollout. |
| Embodied | Steps | Simulator steps used by the episode. |
| Embodied | Policy calls | Low-level carrier invocations. |
| Embodied | Latency | Mean or percentile policy-call latency. |

Table 11 summarizes LIBERO evaluation. Rollouts use `pi0.5`/OpenPI with `rotate180` preprocessing, a 650-step horizon, a 180-step stage budget, and replanning every five steps.

**Table 11:** LIBERO experiment matrix. Budget is reported as tasks times trials, yielding episodes per method. The qualitative task-8 entry corresponds to the visual comparison in Fig. 5.

| Phase | Benchmark | Methods | Budget | Reported in |
|---|---|---|---|---|
| | | *Symbolic checks* | | |
| Planning-only | 40 symbolic tasks | direct, ActionSeq, concept, oracle | 40 tasks | Table 12 |
| | | *Embodied rollouts* | | |
| VLA Long | *LIBERO-10* | flat, ActionSeq, HiBT, ablations | $10 \times 120 = 1200$ ep | Table 5 |
| Four-suite VLA | spatial / object / goal / long | flat, ActionSeq, HiBT | $4 \times 1200$ ep | Table 13 |
| Stress suite | *LIBERO-Hier10* | flat, ActionSeq, HiBT, manual | $10 \times 120 = 1200$ ep | Tables 6, 15 |
| Latency | VLA rollouts | rollout methods | per episode | Table 17 |
| Qualitative | *LIBERO-10* task 8 | flat, ActionSeq, HiBT | 30 inits | Fig. 5 |

**Table 12:** LIBERO planning-only evaluation on 40 symbolic tasks. No VLA rollout is performed. Symbolic SR is predicate-level success under the symbolic checker.

| Method | Exec ↑ | LC ↑ | Sym. SR ↑ | Stable SG | Plan ms ↓ |
|---|---|---|---|---|---|
| Direct BT | 40/40 | 31/40 | 31/40 | N/A | 16 |
| ActionSeq | 40/40 | 39/40 | 39/40 | 1.3 | 11 |
| Concept raw | 40/40 | 39/40 | 39/40 | 1.3 | 10 |
| Concept without stable | 38/40 | 38/40 | 35/40 | 1.1 | 12 |
| **Concept / HiBT** | **40/40** | **40/40** | **40/40** | 1.3 | 12 |
| Oracle pyramid | 40/40 | 40/40 | 40/40 | 1.3 | 10 |

**Table 13:** Four-suite LIBERO VLA rollout summary. Each suite cell contains 1200 episodes. The corresponding visualization appears in Figure 4.

| Method | Spatial SR ↑ | Object SR ↑ | Goal SR ↑ | Long SR ↑ | Mean Steps ↓ |
|---|---|---|---|---|---|
| Flat instruction | 1067/1200 (88.9) | 1023/1200 (85.3) | 891/1200 (74.3) | 907/1200 (75.6) | 428 |
| Action sequence | 1079/1200 (89.9) | 1047/1200 (87.3) | 951/1200 (79.3) | 943/1200 (78.6) | 415 |
| Manual subgoal upper | 1131/1200 (94.3) | 1109/1200 (92.4) | 1023/1200 (85.3) | 1089/1200 (90.8) | 368 |
| **HiBT** | **1127/1200 (93.9)** | **1103/1200 (91.9)** | **1007/1200 (83.9)** | **1070/1200 (89.2)** | **372** |
| HiBT without stable | 1041/1200 (86.8) | 1003/1200 (83.6) | 873/1200 (72.8) | 861/1200 (71.8) | 446 |

**Table 14:** Mechanism ablation on LIBERO-Long. Learned, Stable, and Retry indicate enabled components. Dominant Failure is the modal first-failure stage on failed episodes.

| Variant | Learned | Stable | Retry | SR ↑ | Subgoal SR ↑ | Dominant Failure |
|---|---|---|---|---|---|---|
| **HiBT** | yes | yes | yes | **1070/1200 (89.2)** | **92.1** | residual policy / timeout |
| No stable checks | yes | no | yes | 861/1200 (71.8) | 76.4 | state / order violation |
| No retry/fallback | yes | yes | no | 896/1200 (74.7) | 83.1 | timeout / placement |
| Rule-only realization | no | yes | yes | 603/1200 (50.3) | 63.4 | wrong decomposition |
| Random concept | random | yes | yes | 423/1200 (35.3) | 47.1 | wrong decomposition |
| Shuffled concept | perturbed | yes | yes | 729/1200 (60.8) | 70.2 | misplaced subgoal |
| Wrong-order control | yes | yes | yes | 447/1200 (37.3) | 44.2 | precondition / order |

**Table 15:** Complete custom LIBERO-Hier stress-suite table. Final SR measures only the BDDL final goal, while Strict SR requires both final-goal and process constraint success.

| Method | Final SR ↑ | Constraint SR ↑ | Strict SR ↑ | Distractor Move ↓ | Order Valid ↑ |
|---|---|---|---|---|---|
| Flat instruction | 531/1200 (44.3) | 507/1200 (42.3) | 363/1200 (30.3) | 7.6 cm | 701/1200 (58.4) |
| Action sequence | 651/1200 (54.3) | 627/1200 (52.3) | 507/1200 (42.3) | 5.4 cm | 819/1200 (68.3) |
| Manual subgoal upper | 983/1200 (81.9) | 959/1200 (79.9) | 911/1200 (75.9) | 1.4 cm | 1129/1200 (94.1) |
| **HiBT** | **937/1200 (78.1)** | **913/1200 (76.1)** | **867/1200 (72.3)** | **2.0 cm** | **1049/1200 (87.4)** |
| HiBT without stable | 699/1200 (58.3) | 639/1200 (53.3) | 543/1200 (45.3) | 4.9 cm | 831/1200 (69.3) |

**Table 16:** Cross-benchmark snapshot. Assembly uses symbolic `test60`, LIBERO planning-only uses 40 symbolic tasks, and LIBERO-Long VLA uses 1200 rollouts per method. Costs are comparable within benchmarks.

| Benchmark | Method | Exec / Valid ↑ | LC / Logic ↑ | SR ↑ | Cost ↓ |
|---|---|---|---|---|---|
| | | *Symbolic assembly* | | | |
| Assembly `test60` | Direct BT | 60/60 | 31/60 | 29/60 | 22.54 s |
| Assembly `test60` | ActionSeq | 60/60 | 37/60 | 37/60 | 3.17 s |
| Assembly `test60` | **HiBT / Concept** | **60/60** | **60/60** | **60/60** | **0.45 s** |
| | | *LIBERO expansion* | | | |
| LIBERO planning-only | Direct BT | 40/40 | 31/40 | 31/40 | 16 ms |
| LIBERO planning-only | ActionSeq | 40/40 | 39/40 | 39/40 | 11 ms |
| LIBERO planning-only | **HiBT / Concept** | **40/40** | **40/40** | **40/40** | 12 ms |
| LIBERO-Long VLA | Flat VLA | N/A | N/A | 907/1200 (75.6) | 18.2 s/ep |
| LIBERO-Long VLA | **HiBT + VLA** | N/A | 92.1 subgoal SR | **1070/1200 (89.2)** | **15.7 s/ep** |

**Table 17:** Latency accounting under the 120-trial rollout setting. Policy P50/P95 are per-call latency quantiles; Wall s/Ep excludes model loading.

| Method | Planner ms ↓ | P50 ms ↓ | P95 ms ↓ | Wall s/Ep ↓ |
|---|---|---|---|---|
| Flat instruction | N/A | 84 | 87 | 18.2 |
| Action sequence | 1760 | 84 | 87 | 16.3 |
| **HiBT** | **198** | 83 | 87 | **15.7** |
| HiBT cached plan | 24 | 83 | 87 | 15.4 |
| Manual subgoal upper | N/A | 83 | 87 | 15.5 |

The embodied extension uses a fixed pi0.5/OpenPI LIBERO policy as the low-level carrier, allowing all guidance variants to share the same execution policy and camera preprocessing. The rollout variants are organized around the amount and structure of guidance given to the VLA carrier. A flat VLA baseline receives only the original task instruction. An action-sequence variant converts extracted symbolic actions into sequential subgoal prompts. A concept-guided variant uses the dependency-aware subgoal structure produced by the HiBT interface. Repaired concept guidance additionally applies execution-aware ordering so that placement subgoals precede final state-changing actions when a task requires both spatial and state predicates. This design isolates whether high-level subgoal organization helps the same low-level carrier handle long horizons and ordering constraints.

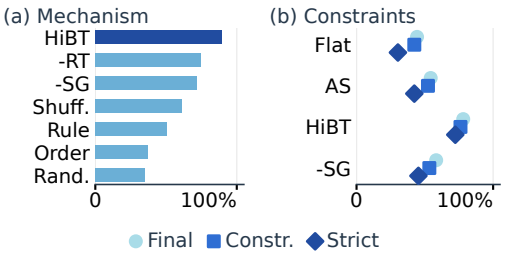



**Figure 8:** LIBERO mechanism and constraint-success summaries. Abbreviations follow Figure 4.

The mechanism controls preserve the low-level carrier while changing learned structure, stable checks, or retry behavior. Their ordering highlights how dependency-aware guidance supports execution. On LIBERO-Hier, the final, constraint, and strict criteria distinguish reaching the goal from satisfying the prescribed process. Tables 14 and 15 give the complete results.

### D.3 LIBERO Qualitative Visuals

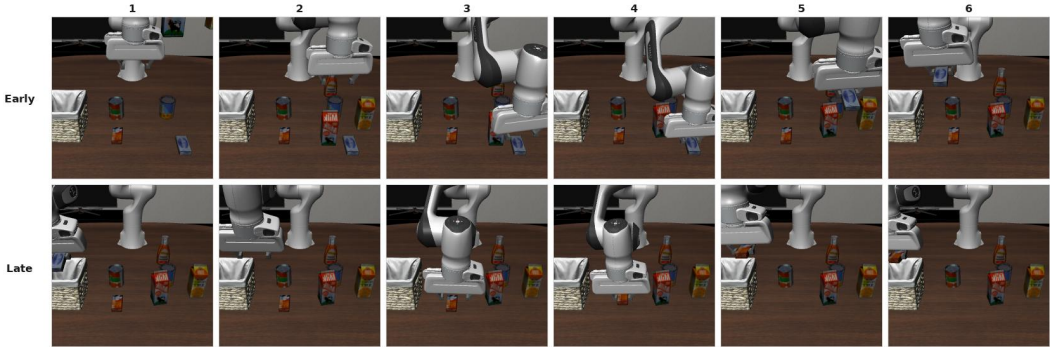



**Figure 9:** Appendix LIBERO-Long task-1 flat-policy process sheet. This successful non-task-8 trajectory complements the task-8 comparison in Fig. 5.

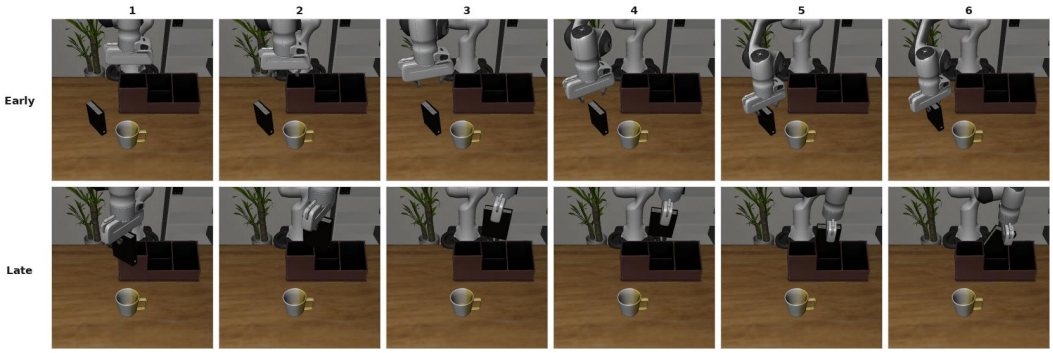



**Figure 10:** Appendix LIBERO-Long task-5 flat-policy process sheet, providing an additional rollout trajectory alongside the task-8 comparison.

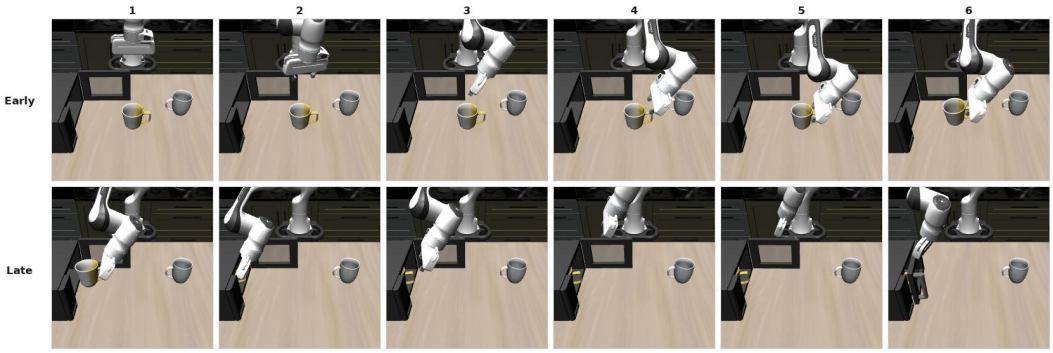
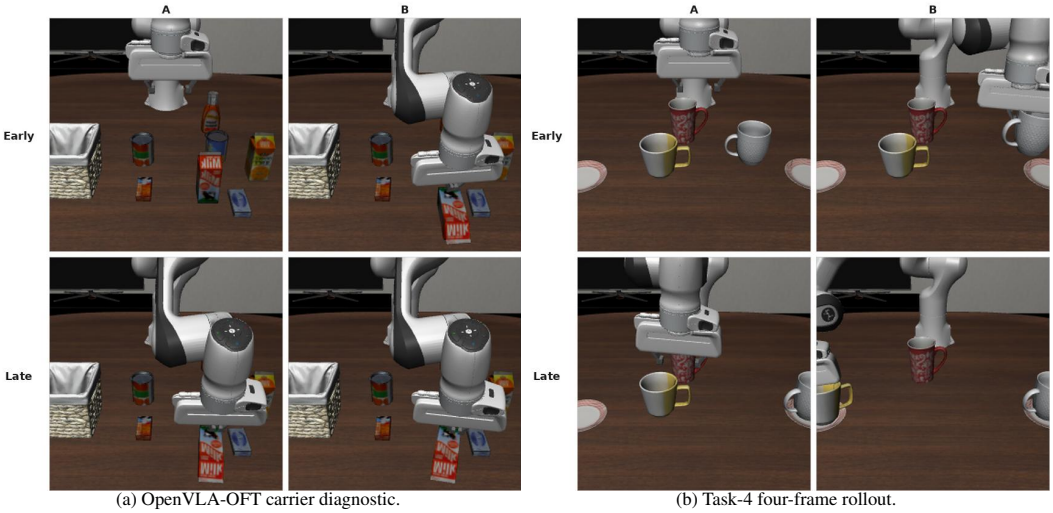



**Figure 11:** Appendix LIBERO-Long task-9 flat-policy process sheet. Together with the task-3 and task-5 sheets, this provides broader visual coverage across the successful sweep rollouts.




(a) OpenVLA-OFT carrier diagnostic. (b) Task-4 four-frame rollout.

**Figure 12:** Appendix LIBERO rollout diagnostics. The left panel shows the OpenVLA-OFT task-1 carrier diagnostic; the right panel shows a compact task-4 $2 \times 2$ rollout summary under the pi0.5/OpenPI carrier.

## E Domain Breakdown

Table 18 aggregates results by task domain across the four tested backbones. The per-domain denominators preserve the benchmark's task distribution. HiBT achieves full success in each domain.

**Table 18:** Cross-domain breakdown aggregated across the four tested backbones.

| Domain | Method | N pooled | Exec | LC | SR |
|---|---|---|---|---|---|
| Gearset | Direct BT | 4 | 4/4 | 4/4 | 4/4 |
| Gearset | ActionSeq | 4 | 4/4 | 4/4 | 4/4 |
| Gearset | HiBT | 4 | 4/4 | 4/4 | 4/4 |
| Gearset-1 | Direct BT | 176 | 162/176 | 79/176 | 73/176 |
| Gearset-1 | ActionSeq | 176 | 157/176 | 98/176 | 101/176 |
| Gearset-1 | HiBT | 176 | 176/176 | 176/176 | 176/176 |
| Chair | Direct BT | 36 | 30/36 | 11/36 | 10/36 |
| Chair | ActionSeq | 36 | 30/36 | 14/36 | 14/36 |
| Chair | HiBT | 36 | 36/36 | 36/36 | 36/36 |
| Lamp | Direct BT | 24 | 22/24 | 8/24 | 3/24 |
| Lamp | ActionSeq | 24 | 23/24 | 6/24 | 6/24 |
| Lamp | HiBT | 24 | 24/24 | 24/24 | 24/24 |

# F Failure Modes

Direct BT generation and action-sequence-first planning mostly fail through hand/tool occupancy, missing equippability, malformed action arguments, or unsupported manipulation preconditions. Unrepaired concept decoding often remains structurally executable, but fails because the required stable support relations are missing. These patterns indicate that symbolic realization contributes dependency completion beyond structural BT formatting.

# G Worked Example

We illustrate the full realization and compilation process using a successfully completed shaft-insertion instance. The initial state contains an occupied outward gripper and requires switching to a clamp gripper before the final insertion, so the realized manipulation first releases the already held gear, detaches the outward gripper, attaches the clamp gripper, grasps the shaft, and then inserts the shaft into the gear-base hole. The corresponding plan pyramid stores the final insertion goal at level 0 and places the required stable support conditions at level 1: the outward gripper has been cleared, the released gripper is again available, the left hand holds the clamp gripper, and the clamp gripper holds the shaft. Level 2 maps these support conditions to grounded method nodes, where the release, tool-detachment, tool-attachment, grasp, and insertion steps respectively establish the support relations needed by later steps. Level 3 preserves the readable action sequence. The compiler then turns each goal or stable subgoal into a selector with a target-check branch and a sequence branch, so repeated BT ticking can reuse already satisfied conditions while preserving repaired dependencies.

# H Reproducibility

The supplementary evaluation package contains prompt assets, plan schemas, prototype banks, deterministic compilation code, and shared evaluation interfaces. HiBT requires a Builder checkpoint, a Predictor checkpoint, and a fixed prototype bank. Reproducing the main results consists of verifying the 60-instance request set, evaluating Direct BT, ActionSeq, and HiBT under the same symbolic simulator, and then running the decoder/compiler and concept-robustness ablations. The tables derive from fixed main-result, complexity, domain, decoder/compiler, and concept-robustness summaries. Appendix J.1 gives the training configuration. Prototype-bank construction follows Builder training; the selected Builder and bank are then fixed for Predictor training and evaluation.

# I Transfer and Extensions

The evaluation connects dependency-heavy symbolic assembly with embodied manipulation across standard LIBERO suites and a dedicated dependency stress suite. The interface separates learned decomposition, domain-specific action schemas, and low-level execution. This organization supports adding validated skills to the prototype bank and integrating other carriers and perception systems.

# J Implementation and Reproduction Details

## J.1 Training Configuration

Table 19 collects the settings of the four-level held-out Builder and shared-backbone Predictor configurations. The concept width is 3584 for the Qwen configurations and 4096 for the LLaMA and Mistral configurations. All four configurations use 16 attention heads and a maximum sequence length of 4096. Language-model weights remain frozen; training optimizes concept-specific modules. The Predictor obtains aligned targets from the frozen Builder and shares its language backbone.

**Table 19:** Four-level held-out training settings; loss definitions appear in Appendix B.

| Setting | Builder | Predictor |
|---|---|---|
| Concept levels / slots | 4 / $(1, 4, 5, 5)$ | 4 / $(1, 4, 5, 5)$ |
| Backbone precision | bfloat16 | shared backbone |
| Training seed | 42 | 42 |
| Epochs | 1 | 1 |
| Batch size | 2 | 1 |
| Optimizer | AdamW | AdamW |
| Learning rate | $10^{-4}$ | $10^{-4}$ |
| Weight decay | 0.01 | 0.01 |
| Warmup ratio | 0.1 | 0.1 |
| Schedule after warmup | cosine | cosine |
| Gradient-norm clipping | 1.0 | 1.0 |
| Reconstruction / concept loss weight | 0.1 | 1.0 |
| Teacher-forced reasoning loss weight | 0.5 | 1.0 |
| Ordering loss weight / margin | 0.5 / 1.0 | — |
| Residual loss weight | 1.0 | — |

## J.2 Family Exclusion Procedure

The filtering utility strips the supported suffixes `_subtask_`*N*, `_subset`, and `_goal_already_satisfied` from evaluation and candidate identifiers, then excludes candidates whose base identifier occurs in the evaluation set. This family-level exclusion precedes Builder and Predictor supervision and bank construction. Request and filtered-row manifests, together with the Builder checkpoint and bank version, identify the evaluated split.

## J.3 Serialized Planning Interface

The `plan-schema-v1` JSON schema contains `schema_version`, `domain`, `input`, and `pyramid`, plus an optional `problem_id`. The input records the grounded target and symbolic initial state; the pyramid contains four ordered levels (Table 20). Predicates and actions have explicit names and arguments, with identifier references linking dependencies across levels.

**Table 20:** Serialized fields of the four-level symbolic plan pyramid.

| Level | Object | Fields and meaning |
|---|---|---|
| Goal | $\gamma$ | `id`, `type`, `predicate`, `args`; exactly one goal item. |
| Stable subgoals | $\mathcal{S}$ | `id`, `type`, `predicate`, `args`, `supports`; support relations for later steps. |
| Methods | $\mathcal{M}$ | `id`, `type`, `achieves`, `action`, `requires`; each grounded action is linked to its target and required subgoal IDs. |
| Actions | $\pi$ | `step`, `type`, `name`, `args`, `derived_from`; an ordered trace linked back to method IDs. |

The compiler follows `achieves` and `requires` links. Nodes contain `summary` and `name`; control-flow nodes also contain ordered `children`. Names combine a type and grounded call, e.g., `action: insert(left_hand, inward_claw, gear3, shaft2)`. The method graph drives compilation; the action trace supports reporting and leaf-consistency checks.

### J.4 Baseline Prompt and Output Contracts

Both generators receive the target, serialized state, domain actions and predicates, output contract, examples, and request template. Each returns only a JSON object.

**Direct BT.** The output fields are `thought`, `action_sequence`, and `behavior_tree`. Nodes use grounded calls and the runtime's selector, sequence, target, precondition, and action types. The prompt aligns action leaves with the reported sequence. Full and compact profiles preserve the same input/output contract; the compact profile uses schema rules and a skeleton BT.

**ActionSeq.** The output has `task_plan`, a list of grounded action calls, and `explanation`, a brief rationale. The implementation parses the calls, constructs a symbolic plan pyramid, and passes it to the deterministic BT compiler. In the matched P+R controls in Table 2, the baseline additionally receives the same Predictor candidates and dependency-repair backend while retaining its flat representation.

### J.5 Success-Rate Intervals

For a count $k$ of successful assembly instances out of $n = 60$, let $\hat{p} = k/n$ and $z = 1.96$. The Wilson interval reported in Table 8 and Figure 3 is

$$\left[\frac{\hat{p} + z^2/(2n) - z\sqrt{\hat{p}(1-\hat{p})/n + z^2/(4n^2)}}{1 + z^2/n},\ \frac{\hat{p} + z^2/(2n) + z\sqrt{\hat{p}(1-\hat{p})/n + z^2/(4n^2)}}{1 + z^2/n}\right]. \tag{32}$$

Endpoints are multiplied by 100 and rounded to one decimal place. For example, $k = 60$ gives $[94.0, 100.0]\%$. These intervals describe finite-test-set uncertainty for a fixed trained model. The four-backbone difficulty and domain tables aggregate outcomes on the same requests across backbones; their pooled counts are descriptive breakdowns rather than additional independent test instances.