Learning What's Missing: Failure-Driven Skill Discovery via Predicate Bridges

Yanwen Liu [1] **Wendi Ren** [1] **Haozhou Gao** [1] **Shuang Li** [1]

Abstract

Agents deployed in new environments often fail not because they plan incorrectly, but because they lack the skills required to execute their plans. Some tasks can be solved with existing primitives, while others expose missing capabilities that must be acquired on demand. However, current approaches largely sidestep this problem: planners assume a fixed skill library, and skill discovery methods add behaviors without linking them to the cause of failure. We propose **Predicate Bridge Discovery (PBD)**, a framework that *treats failure as supervision for learning missing skills*. Given an instruction, the agent constructs a predicate program and performs bidirectional reasoning. Forward reasoning estimates which predicates are executable from the current state, while backward reasoning identifies which predicates are required for success. When these sets do not connect, their mismatch reveals a bridge gap—the minimal predicate that blocks execution. PBD learns directly from this gap. It turns the missing predicate into a parameterized skill, and adds it to the library only after it reliably achieves its effect across instances. This yields a compact, executable, and reusable skill set that grows as new failures are encountered. We evaluate PBD on BabyAI and MiniHack, spanning tasks from single-step behaviors to compositional problems that require acquiring new skills. PBD consistently converts failures into targeted skill acquisition and improves generalization to unseen bindings and tasks. These results show how agents can not only act, but identify and learn what they are missing.

[1]School of Data Science, The Chinese University of Hong Kong (Shenzhen), Shenzhen, China. Correspondence to: Shuang Li <lishuang@cuhk.edu.cn>.

Accepted to the Workshop on Failure Modes of Agentic AI at ICML 2026, Seoul, South Korea. Copyright 2026 by the author(s).

1 Introduction

Agents deployed in new environments often encounter tasks that cannot be solved with their existing capabilities. Some tasks can be completed by composing known skills, while others fail because a required capability is simply missing. This phenomenon appears across embodied and language-guided agents, where success depends not only on planning but also on having the right set of executable skills (Ahn et al., 2022; Wang et al., 2023). An adaptive agent should therefore do more than search over a fixed set of behaviors. It should be able to identify what is missing and acquire new skills accordingly.

Existing approaches largely sidestep this problem. Classical and language-based planners assume a fixed skill library and treat failure as a search or inference issue (Fikes & Nilsson, 1971; Aeronautiques et al., 1998; Yao et al., 2022), even when the required capability is absent. On the other hand, skill discovery methods in reinforcement learning introduce new options or latent behaviors through hierarchical abstractions or intrinsic objectives (Sutton et al., 1999; Bacon et al., 2017; Eysenbach et al., 2018). While these methods can produce reusable behaviors, they are typically not grounded in the specific reason a task failed, making it unclear whether the learned skills address the actual bottleneck.

In this work, we propose a different perspective: failure provides structured supervision for learning missing capabilities. We formalize this idea in *Predicate Bridge Discovery* (PBD), a framework that identifies the minimal capability required to resolve a failed plan and learns it as a reusable skill.

Our approach represents tasks as predicate programs and skills as parameterized operators with preconditions and effects, following the tradition of symbolic planning and neuro-symbolic abstraction (Konidaris et al., 2018; Silver et al., 2021). Given a task, the agent performs bidirectional reasoning over its current skill library. Forward reasoning estimates which predicates are reachable from the current state, while backward reasoning identifies which predicates are required to satisfy the goal. When these two processes fail to connect, their mismatch localizes a *bridge gap*, defined as the smallest predicate that blocks completion. This formulation turns failure diagnosis into a precise localization problem over symbolic structure.

Rather than proposing arbitrary new behaviors, PBD derives a new skill directly from this gap. The missing predicate is factorized into a reusable template and instance-specific bindings, separating functional structure from task-specific details. A parameterized policy is then trained to realize this predicate, and the resulting skill is added to the library only after demonstrating both execution success and reuse across different bindings. This ensures that library growth remains compact, interpretable, and compositional.

This design addresses two key challenges in skill learning: when to learn and what to learn. The need for a new skill is triggered by a precise symbolic mismatch between reachability and necessity, and the content of the skill is determined by the minimal missing predicate rather than heuristic or unsupervised discovery. Together, these properties allow the agent to expand its capabilities in a targeted and reusable manner.

We evaluate PBD on BabyAI (Chevalier-Boisvert et al., 2019) and MiniHack (Samvelyan et al., 2021), two environments with structured tasks of varying complexity. BabyAI provides a controlled setting for compositional instruction following, while MiniHack introduces richer dynamics such as tool use and hidden preconditions. Across these settings, we examine not only task success but also whether the agent correctly identifies missing predicates, learns reusable skill templates, and composes them to solve more complex tasks.

In summary, this paper makes three contributions. First, we formulate failure-driven skill acquisition as a bidirectional predicate bridge discovery problem, which localizes minimal missing capabilities through the mismatch between reachability and necessity. Second, we introduce adaptive predicate factorization, which enables reusable skill templates that generalize across bindings. Third, we demonstrate through experiments in BabyAI and MiniHack that this approach leads to more effective and compositional skill learning compared to fixed-library planning and unconstrained skill discovery methods.

PBD formalizes a simple form of adaptive skill growth. When an agent fails on a task, the failure may have *two* different causes: the agent may have composed its existing skills incorrectly, or *the task may require a capability that is not yet in the library*. PBD distinguishes these cases by comparing two symbolic objects: what the current skill library can make true, and what the task requires to become true. When these two sets do not meet, their mismatch exposes a *bridge predicate*: the *first missing condition* that prevents the current task from being solved.

The framework has three ingredients. First, the world is described by predicates that can be checked in the current state. Second, language instructions are converted into predicate programs that expose the dependencies among subgoals. Third, skills are represented by symbolic preconditions and effects, together with low-level policies that attempt to realize those effects. The *central contribution* is the *bridge-discovery step* that connects these ingredients: it turns a failed plan into a concrete target for skill learning.

2 Related Work

A central question in this work is how an agent should respond when its current capabilities are insufficient for a task. Prior work on planning and embodied agents has made significant progress in reasoning over structured representations and composing skills, but typically assumes that the available skill set is fixed. Classical planning frameworks such as STRIPS and PDDL (Fikes & Nilsson, 1971; Aeronautiques et al., 1998) formalize actions through preconditions and effects, enabling efficient reasoning about reachability and goal satisfaction. Task-and-motion planning extends this paradigm by integrating symbolic reasoning with continuous feasibility checks (Garrett et al., 2020; 2021). More recently, language-based planners leverage large language models to decompose instructions, select actions, or generate executable policies (Ahn et al., 2022; Liang et al., 2023; Yao et al., 2022), while systems such as Voyager maintain growing libraries of reusable code-based skills (Wang et al., 2023). Despite these advances, these approaches primarily treat failure as a search or grounding issue, and do not explicitly address whether the required capability is missing from the current library.

A complementary line of work studies skill learning through hierarchical reinforcement learning and option discovery. The options framework introduces temporally extended actions with initiation and termination conditions (Sutton et al., 1999), and methods such as the option-critic architecture learn such skills end-to-end (Bacon et al., 2017). Unsupervised skill discovery methods further aim to learn diverse or controllable behaviors using intrinsic objectives such as mutual information or state coverage (Gregor et al., 2016; Eysenbach et al., 2018; Sharma et al., 2019). While these approaches can produce reusable behaviors and improve exploration, they are typically not conditioned on task-specific failures. As a result, the discovered skills are not guaranteed to address the actual bottlenecks encountered during task execution, and there is no mechanism for identifying which capability is missing when a plan fails.

Another line of work focuses on learning structured abstractions and symbolic models that support planning. Methods such as Konidaris et al. (2018) derive symbolic representations from low-level behaviors, while operator learning and relational transition models learn abstractions that generalize across objects and tasks (Silver et al., 2021; Chitnis et al., 2022). Bilevel and neuro-symbolic planning approaches integrate symbolic operators with learned policies

and samplers (Silver et al., 2022), and predicate invention methods aim to discover useful abstractions for planning (Silver et al., 2023). Related work on action-model learning infers preconditions and effects from execution traces (Yang et al., 2005; Amir & Chang, 2008), and plan-repair methods update plans when execution deviates from expectations. These approaches improve representations and models for planning, but typically assume that the underlying set of capabilities is sufficient. In contrast, our focus is on identifying when this assumption fails and expanding the skill library accordingly.

Finally, object-centric representations and compositional environments provide a useful setting for evaluating capability acquisition. Object-oriented formulations of reinforcement learning support generalization across entities and relations (Diuk et al., 2008), while environments such as BabyAI (Chevalier-Boisvert et al., 2019) and MiniHack (Samvelyan et al., 2021) expose tasks with increasing compositional structure and hidden dependencies. We use these environments not only to evaluate task success, but to test whether an agent can identify missing predicates, learn reusable skills, and compose them across tasks.

In contrast to prior work, PBD does not assume a fixed or pre-learned skill set. By explicitly comparing what is reachable with what is required, it localizes the minimal missing predicate and uses it as supervision for learning a new skill. This connects planning, abstraction, and skill learning through a single mechanism that enables targeted and compositional capability growth.

3 Model

PBD formalizes a simple form of adaptive skill growth. When an agent fails on a task, the failure may have *two* different causes: the agent may have composed its existing skills incorrectly, or *the task may require a capability that is not yet in the library*. PBD distinguishes these cases by comparing two symbolic objects: what the current skill library can make true, and what the task requires to become true. When these two sets do not meet, their mismatch exposes a *bridge predicate*: the *first missing condition* that prevents the current task from being solved.

The framework has three ingredients. First, the world is described by predicates that can be checked in the current state. Second, language instructions are converted into predicate programs that expose the dependencies among subgoals. Third, skills are represented by symbolic preconditions and effects, together with low-level policies that attempt to realize those effects. The *central contribution* is the *bridge-discovery step* that connects these ingredients: it turns a failed plan into a concrete target for skill learning.

3.1 Predicate States

Let s be the current environment state and let $\mathcal{P}$ denote the set of grounded predicates available in a task instance. A predicate is a statement about the current world, such as `Near(obj)`, `Holding(obj)`, `Open(gate)`, `Has(tool)`, or `At(target)`. A predicate checker

$$C(s) \subseteq \mathcal{P}$$

returns the set of predicates that are true in state s. The checker may be exact (e.g., extracted from simulator state) or approximate (e.g., predicted from observations). In our experiments, we use an environment-level checker.

The checker is used to read the current state, determine goal satisfaction, and evaluate skill success. All bridge discovery operates directly on these predicates, without requiring a learned world model.

3.2 Predicate Programs

Given a language instruction g, PBD constructs a predicate program ψ_g that describes the conditions required for success. A program is composed of atomic predicates, conjunctions, and temporal composition:

$$\psi ::= p \mid \psi_1 \wedge \psi_2 \mid \psi_1; \psi_2.$$

Here p is a grounded predicate, $\wedge$ means that several predicates must hold, and ; means that one subgoal must be achieved before another.

The program induces a dependency structure over predicates. During execution, PBD focuses on the earliest subformula of ψ_g that is not yet satisfied by $C(s)$. We call this subformula the active frontier. Intuitively, the frontier is the current point of contact between the instruction and the agent's abilities: predicates before the frontier have already been achieved, while predicates at the frontier still need to be made true.

In our implementation, the predicate program is produced by a prompted language model. The prompt provides the predicate vocabulary, constraints on valid predicate expressions, and examples of instruction-to-program translations. The language model is used only to propose symbolic structure. It does not certify that a skill exists and it does not provide the low-level policy. Whether a proposed predicate transition is executable is decided later by bridge discovery, learning, and validation.

3.3 Skills and Symbolic Operators

PBD separates symbolic operators from executable skills. A symbolic operator describes a possible transition in predicate space:

$$o = \langle \mathrm{pre}(o), \mathrm{eff}(o) \rangle,$$

where $\mathrm{pre}(o)$ is the set of predicates that should hold before the transition, and $\mathrm{eff}(o)$ is the set of predicates expected after successful execution. An executable skill is a symbolic operator paired with a low-level policy: $\kappa = \langle o_\kappa, \pi_\kappa, \rho_\kappa \rangle$. Here o_κ specifies the symbolic preconditions and effects, π_κ is the policy that attempts to realize the effect, and ρ_κ is the empirical validation success rate of the skill.

This distinction is important. The language model may propose an operator saying that a tool can clear a barrier, but the agent does not possess the corresponding skill until it has trained and validated a policy that actually clears barriers. Thus, PBD uses symbolic operators as hypotheses about task structure, while the skill library $\mathcal{K}$ contains only validated executable capabilities.

Operators and skills may contain variables. A binding b replaces these variables with concrete entities in the current task. For example, the general predicate `Holding(x)` can be instantiated as `Holding(red_object)` or `Holding(tool_3)`. This separation between a reusable predicate form and a concrete binding is what allows one learned skill to apply across many task instances.

3.4 Bidirectional Bridge Discovery

Figure 1 illustrates an overview of bidirectional Predicate Bridge Discovery. The bridge-discovery step compares reachability and necessity. Let $Y = C(s)$ be the predicates true in the current state. The *reachable set* R contains the predicates that can be made true from Y using the current executable skill library $\mathcal{K}$. It is computed by repeatedly applying the effects of validated skills whose preconditions are already reachable: $R = \mathrm{Reach}(Y, \mathcal{K})$. Thus, R answers the question: *what can the agent currently make true?*

The *needed set* N contains the predicates required to satisfy the active frontier of the predicate program. It is computed by reasoning backward through the symbolic operators: starting from the active frontier, PBD adds the preconditions of operators whose effects would help satisfy the frontier:

$$N = \mathrm{Need}(\mathrm{Frontier}(\psi_g, Y), \mathcal{O}),$$

where $\mathcal{O}$ is the set of symbolic operators proposed for the domain. Thus, N answers the question: *what must become true for the task to proceed?*

The bridge gap is the difference $G = N \setminus R$. If $G = \emptyset$, then every required predicate is reachable with the current skill library. The agent can extract a symbolic skill sequence and execute it. If $G \neq \emptyset$, then planning alone is insufficient: at least one predicate required by the task cannot be achieved by any validated skill.

PBD selects from G the first missing predicate on the active dependency branch whose preceding predicates are already satisfied or reachable.

We denote this predicate by

$$p^\star = \mathrm{Bridge}(G, \psi_g, R).$$

This is the bridge predicate. It is minimal in the sense that predicates before it on the current dependency branch can already be reached, while predicates after it should not yet be learned because they depend on this unresolved capability.

For example, suppose a task requires moving an object into a protected region. Backward reasoning may reveal that `BarrierCleared(barrier)` must become true before the object can be placed at the target. Forward reasoning may show that the agent can reach the barrier and obtain the necessary tool, but no accepted skill can make `BarrierCleared(barrier)` true. The bridge is therefore not the entire task, nor the final placement predicate, but the missing predicate `BarrierCleared(barrier)`. Learning this predicate yields a reusable capability that can later be composed with navigation, object manipulation, and placement skills.

This is the key difference from fixed-library planning and unconstrained skill discovery. A fixed-library planner can only fail when G is nonempty. An unsupervised skill-discovery method can introduce new behaviors, but it does not know which behavior would repair the failed task. PBD uses the need–reach mismatch to specify both *when* learning is necessary and *what* should be learned.

4 Learning

Once PBD identifies a bridge predicate $p^\star$, it converts that predicate into a new candidate skill. The learning problem is deliberately narrow: the agent is not asked to solve the original long-horizon task from scratch. It is asked to learn a policy that makes the missing predicate true.

4.1 From a Bridge Predicate to a Skill Template

The bridge predicate is first observed in a grounded form, such as `Holding(obj_7)` or `BarrierCleared(barrier_2)`. PBD generalizes this grounded predicate by replacing concrete entities with variables:

`Holding(obj_7)` $\mapsto$ `Holding(x)`.

The generalized predicate defines the effect of the new candidate skill, while the original entities define the current binding.

The preconditions of the candidate skill are taken from the symbolic operator that introduced the bridge predicate during backward reasoning. The preconditions of the candidate skill are derived from the program structure around $p^\star$: PBD

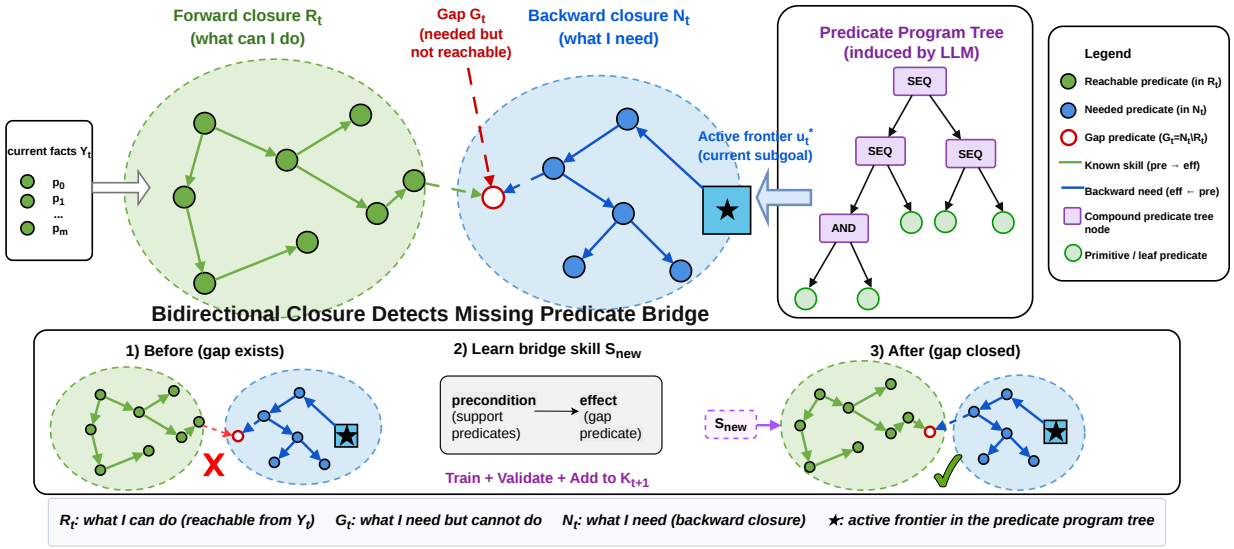


Figure 1. Overview of bidirectional Predicate Bridge Discovery. Given the current predicate facts Y_t and the predicate program tree induced by the instruction, PBD performs forward closure over the current skill library to obtain the reachable set R_t and backward closure from the active frontier $u_t^\star$ to obtain the needed set N_t. Their mismatch $G_t = N_t \setminus R_t$ identifies a missing bridge predicate: a condition required by the current subgoal but not achievable by the current library. PBD then converts this predicate into a new skill target, trains and validates the corresponding low-level policy, and adds the accepted skill to the library. After the skill is added, the previous gap becomes reachable and the predicate program can continue.

uses the nearest predecessor subformula under the local SEQ relation.

These preconditions are not learned from scratch by the low-level policy; they define where skill execution should begin. The low-level policy is responsible for closing the remaining gap between the reachable preconditions and the missing effect. The candidate skill is therefore

$$\kappa^\star = \langle o^\star, \pi^\star, \rho^\star \rangle,$$

where $o^\star$ contains the generalized preconditions and effect, $\pi^\star$ is the policy to be trained, and $\rho^\star$ is assigned after validation.

This factorization prevents library growth from degenerating into one-off repairs. The agent does not learn a separate skill for every object, barrier, tool, or target. It learns a predicate-level template, and different task instances supply different bindings.

4.2 Sparse Predicate-Reward Learning

The policy $\pi^\star$ is trained in the environment with the bridge predicate as its success condition. For a sampled binding b, let $p^\star[b]$ denote the grounded target predicate for that episode. The episode has horizon H. The skill succeeds if the target predicate becomes true at any time during the episode:

$$\mathrm{Succ}(\pi^\star, b) = \mathbb{I}\left[\exists t \le H \ \text{ s.t. } \ p^\star[b] \in C(s_t)\right].$$

Equivalently, the policy receives a sparse reward, reduced by a small step penalty λ_{step}, at the first time the target predicate becomes true and zero otherwise:

$$r_t = \begin{cases} 1 - \lambda_{\text{step}}, & p^\star[b] \in C(s_t), \\ & p^\star[b] \notin C(s_j), \quad \forall j < t, \\ 0, & \text{otherwise.} \end{cases}$$

Thus, success is defined over the episode, not by a dense change in a learned predicate score.

The skill-learning objective is

$$\max_{\pi^\star} \ \mathbb{E}_b \left[\mathrm{Succ}(\pi^\star, b)\right].$$

In our experiments, we optimize this objective with reinforcement learning. When warm-up demonstrations are available, they initialize the policy before reinforcement learning, but the acceptance criterion remains predicate success in the environment.

Although the reward is sparse, the bridge formulation makes the learning problem substantially more focused than solving the original task. The target predicate is local, the starting states are chosen so that the candidate preconditions hold, and the policy is trained only on the capability that the symbolic analysis identified as missing. This is why failure becomes supervision: it specifies a low-level learning problem that is smaller and more reusable than the failed task itself.

4.3 Validation Across Bindings

A candidate skill is not added to the library merely because it solves the instance that triggered learning. It must succeed across multiple bindings of the same predicate template.

Let $\mathcal{B}_{\text{val}}$ be the validation bindings for the candidate template, and let M be the number of evaluation episodes per binding. We compute a single validation success rate:

$$\hat{\rho}_{\text{val}}(\kappa^\star) = \frac{1}{|\mathcal{B}_{\text{val}}|M} \sum_{b \in \mathcal{B}_{\text{val}}} \sum_{m=1}^{M} \text{Succ}^{(m)}(\pi^\star, b).$$

The skill is accepted if $\hat{\rho}_{\text{val}}(\kappa^\star) \geq \rho_{\text{val}}$. When accepted, it is inserted into the library:

$$\mathcal{K} \leftarrow \mathcal{K} \cup \{\kappa^\star\}, \qquad \rho_{\kappa^\star} \leftarrow \hat{\rho}_{\text{val}}(\kappa^\star).$$

We use this aggregate validation rate directly over all sampled bindings. There is no separate instance-level score and reuse-level score. A candidate becomes a skill only if it works as a reusable predicate template, not as a memorized repair for the original binding.

4.4 Library Growth and Replanning

After a skill is accepted, PBD recomputes reachability with the expanded library. The bridge predicate that was previously outside R can now be reached, and this may make later predicates reachable as well. A task that originally ended at a bridge gap can therefore become solvable after the corresponding skill is learned.

This process produces an incremental form of capability growth. Early failures expose simple missing predicates. Once learned, those skills become building blocks for later tasks. More complex instructions then reveal deeper gaps, and PBD repeats the same procedure: localize the first missing predicate, train the corresponding skill, validate it across bindings, and replan.

The experimental curriculum instantiates this loop. Earlier stages isolate individual bridge predicates so that the agent can acquire compact skills. Later stages test whether the accumulated library can solve tasks that require several accepted skills to be ordered and composed. Thus, the evaluation measures not only final task success, but whether the learned library grows in the way the method predicts.

Summary. PBD turns planning failure into a skill-learning target. The predicate checker tells the agent what is true, the predicate program tells it what must become true, and the skill library tells it what it can already make true. The bridge predicate is the first condition that lies on the task's dependency path but outside the reach of the current library. By learning a policy for exactly that predicate and accepting it after validation across bindings, PBD grows the library through targeted, interpretable, and compositional skills.

5 Experiment

5.1 Experimental Setup

We evaluate PBD in BabyAI and MiniHack, two structured environments with compositional goals and discrete predicate-level task structure. BabyAI provides controlled gridworld tasks involving objects, colors, doors, keys, and spatial relations, while MiniHack introduces richer dynamics such as inventory, tool use, lava, monsters, and hidden preconditions. We use 13 BabyAI environments: 10 skill-acquisition environments that expose localized predicate gaps, and 3 compositional skill-reuse environments, MINI-BOSSLEVEL, SYNTHSEQ, and BOSSLEVEL. For MiniHack, we use 6 skill-acquisition environments and 2 compositional environments, QUEST-EASY and WOD-HARD. Unlike BabyAI, MiniHack compositional environments may also expose new missing predicates during training. However, for both BabyAI and MiniHack, final evaluation is conducted on held-out seeds with the learned skill library fixed, and no additional skill discovery or training is allowed. The full list of environments and complete per-environment results are provided in Appendix A.2.

Predicate Checker. Because neither environment directly exposes the symbolic interface required by PBD, we implement rule-based predicate extractors from environment states. The extractors ground task-relevant entities and agent state into predicates, e.g., `Adjacent(obj)`, `HoldingObj(obj)`, `DoorOpened(door)`, `SafeToCrossLava()`, and `MonsterDead(monster)`. These grounded predicates are used for bridge detection, skill validation, and library reuse. To verify that PBD is not inherently tied to hand-coded checkers at inference time, we additionally evaluate a variant in which predicates are replaced by learned classifiers trained on labels generated by the rule-based extractors. Results (Appendix B.1) show that PBD with learned grounding achieves comparable performance, indicating compatibility with approximate predicate checkers. Further details are provided in Appendix C.2.

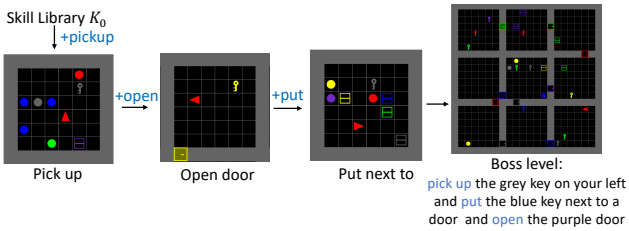


Figure 2. Illustrative BabyAI curriculum for skill accumulation and compositional reuse.

Curriculum. The curriculum proceeds from local skill-acquisition tasks to compositional reuse tasks. Early stages

Table 1. Closed-loop success rate on compositional environments over 100 held-out episodes. During evaluation, the learned library is frozen and no additional skill discovery or training is allowed. PBD-direct trains discovered bridge skills in the target task, whereas PBD-Curriculum learns them in simpler acquisition environments before reuse.

Environment	IL	RL	HRL	PBD-direct	PBD-Curriculum
BabyAI-MiniBossLevel	0.36	0.10	0.13	0.46	**0.78**
BabyAI-SynthSeq	0.26	0.02	0.08	0.40	**0.73**
BabyAI-BossLevel	0.3	0.01	0.07	0.41	**0.67**
MiniHack-WoD-Hard	0.00	0.00	0.14	**1.00**	**1.00**
MiniHack-Quest-Easy	0.5	0.00	0.38	**0.69**	**0.69**

provide clean environments for training missing bridge skills, such as object pickup, door opening, and lava crossing. Later stages require multiple learned skills to be ordered and reused in long-horizon predicate programs. Figure 2 illustrates representative curriculum stages in BabyAI, and the full curriculum stages of BabyAI and MiniHack are listed in Appendix C.1.

The main experimental question is whether, as task difficulty increases, PBD can identify the intermediate capabilities missing from the current skill library, convert them into reusable skills, and ultimately solve complex tasks that require composing multiple learned skills.

5.2 Results

We report results on the compositional environments, which test whether the learned library can be reused rather than only solving isolated acquisition tasks. For BabyAI, the compositional tasks are MINIBOSSLEVEL, SYNTHSEQ, and BOSSLEVEL. For MiniHack, they are QUEST-EASY and WOD-HARD. All success rates are computed over 100 held-out evaluation episodes with the learned library fixed.

Baselines and PBD variants. For BabyAI, IL is trained on 10k official expert-demonstration episodes, and RL uses the BabyAI PPO implementation. For MiniHack, IL is trained on 40 manually collected demonstrations per environment, and RL uses the native PolyBeast baseline. The HRL baseline uses the same fixed skill library available at PBD evaluation time, but replaces predicate-level planning and bridge discovery with a learned high-level actor-critic controller. We compare two PBD variants: PBD-DIRECT, which trains discovered bridge skills directly in the target compositional environment, and PBD-CURRICULUM, which first learns bridge skills in simpler acquisition environments and then reuses the resulting library on the compositional task. Each discovered bridge skill is trained as a goal-conditioned low-level PPO policy with the bridge predicate as its success condition. We warm-start each skill policy by behavior cloning for 5 epochs, using 100 demonstration episodes per skill in BabyAI and 40 per skill in MiniHack.

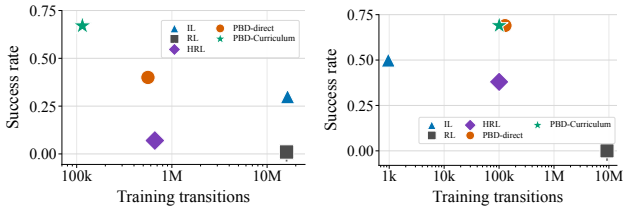
Success Rate. Table 1 reports closed-loop success rates on compositional environments. PBD outperforms IL, RL, and HRL across all environments. On BabyAI, PBD-Curriculum achieves the best performance, improving over PBD-direct by 0.26–0.33 absolute success rate. On MiniHack, PBD-direct matches PBD-Curriculum, reaching 0.69 on QUEST-EASY and 1.00 on WOD-HARD.

Sample efficiency. Figure 3 compares success rate as a function of total training transitions. For IL, we count demonstration transitions; for RL and HRL, we count online environment transitions used during policy training. For PBD, we count the full cost of acquiring every learned bridge skill used by the final library:

$$N_{\text{PBD}} = \sum_{\kappa \in \mathcal{K}_{\text{used}}} \left(N^{(\kappa)}_{\text{warmup}} + N^{(\kappa)}_{\text{PPO}} \right),$$

where $N^{(\kappa)}_{\text{warmup}}$ is the number of behavior-cloning warm-up transitions for skill κ, and $N^{(\kappa)}_{\text{PPO}}$ is the number of PPO environment transitions consumed before the skill is accepted or early-stopped. If the same skill is reused multiple times during evaluation, its training cost is counted only once. Evaluation episodes are excluded from all training-transition budgets.

Under this accounting, PBD achieves higher success with fewer transitions than full-task RL. Rather than repeatedly exploring the entire compositional task, PBD spends interaction on localized bridge-skill learning and reuses the acquired skills in later tasks.



(a) BabyAI-BossLevel *(b)* MiniHack-Quest-Easy

Figure 3. Sample efficiency on compositional evaluation.

Ablation Study: Effect of curriculum. We ablate curriculum-based skill acquisition by comparing PBD-CURRICULUM with PBD-DIRECT. The two variants share the same PBD pipeline and differ only in the skill-training environment. PBD-DIRECT trains them directly in the target compositional environment, while PBD-CURRICULUM trains them in simpler acquisition environments before reuse. Additional ablation studies are provided in Appendix B.

As shown in Table 1, removing the curriculum substantially reduces BabyAI success from 0.78/ 0.73/ 0.67 to 0.46/ 0.40/0.41 on MINIBOSSLEVEL/SYNTHSEQ/BOSSLEVEL. Figure 3 further shows that direct training requires more transitions for lower final success. This suggests that compared with curriculum acquisition tasks, direct training must learn the same predicates inside longer-horizon compositional environments, where useful initiation states are rarer and other subgoals interfere with exploration.

On MiniHack, PBD-DIRECT matches PBD-CURRICULUM on both QUEST-EASY and WOD-HARD. This indicates that the MiniHack target tasks already provide a sufficiently learnable signal once the missing predicate is localized. Thus, curriculum mainly affects how easily a selected bridge skill can be learned, while bidirectional bridge discovery determines what skill should be learned.

5.3 Skill Library Growth and Skill Reuse

We analyze whether PBD improves performance by expanding the executable skill library rather than merely replanning with the initial skills. Figures 4 and 8 (Appendix A) shows library snapshots during training, together with episode-level success attribution and per-skill invocation frequency.

Across BabyAI and MiniHack, base skills account for only a small fraction of successful compositional episodes. As the library grows, learned-skill reuse increases while failures decrease. BabyAI skills such as PICKUP, OPENDOOR, and PUTNEXTTO are first learned from local predicate gaps and later reused in compositional tasks. MiniHack shows the same trend for inventory, lava, tool-use, and combat skills. This indicates that PBD expands the reachable predicate set through validated bridge skills, which then serve as reusable operators in later compositional plans.

Figure 5 illustrates one BabyAI-BossLevel case and the corresponding bridge detection process. Figure 9 (Appendix A) provides an additional case study of MiniHack. Figures 6 and 7 (Appendix A) further show skill library expansion over the BabyAI and MiniHack curriculum. More experiment results and details are shown in Appendix A and C.

6 Conclusion and Limitation

PBD is a failure-driven framework that learns missing skills from failed plans by identifying bridge predicates and training reusable parameterized skills to close execution gaps. By validating each learned skill before adding it to the library, PBD incrementally builds an executable and reusable skill set rather than relying on one-off fixes. Experiments on BabyAI and MiniHack demonstrate improved closed-loop success and sample efficiency, with gains driven by effective skill reuse. Results also suggest that curriculum learning is particularly beneficial when bridge skills are difficult to acquire directly in complex compositional tasks. By grounding each library update in an explicit missing predicate, PBD makes capability acquisition traceable, exposing both the symbolic bottleneck that triggered skill learning and the downstream contexts in which the learned skill is reused.

Our framework currently assumes a predefined predicate vocabulary, which bounds the skills PBD can discover to capabilities expressible within the vocabulary and limits transfer across domains with different predicate sets. Future work will extend predicate grounding to noisier and more perceptual observation settings, and relax the fixed-vocabulary assumption by learning or adapting predicate abstractions across domains.

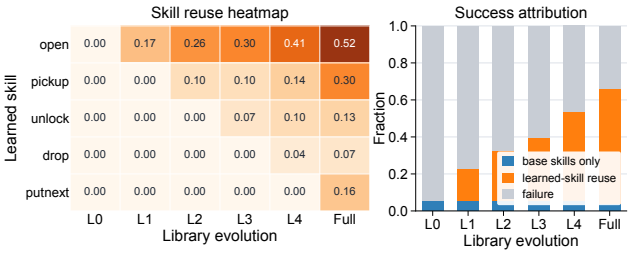


Figure 4. Skill reuse and success attribution across library growth in BabyAI. Heatmaps report the fraction of evaluation episodes in which each learned skill is invoked under each library snapshot. Bar plots decompose episodes into successes using only base skills, successes using learned skill, and failures. As the library grows, learned-skill reuse increases and the failure rate decreases.

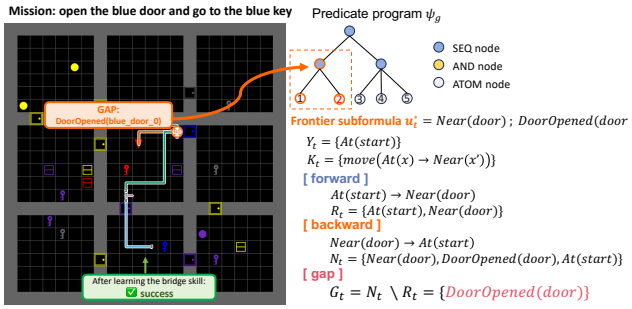


Figure 5. BabyAI-BossLevel case study. PBD detects `DoorOpened(door)` as the missing bridge predicate because it is needed by the active frontier but not reachable with the current library. After the corresponding skill is learned and added to the library, the gap is closed and the task succeeds.

References

Aeronautiques, C., Howe, A., Knoblock, C., McDermott, I. D., Ram, A., Veloso, M., Weld, D., Sri, D. W., Barrett, A., Christianson, D., et al. Pddl—the planning domain definition language. *Technical Report, Tech. Rep.*, 1998.

Ahn, M., Brohan, A., Brown, N., Chebotar, Y., Cortes, O., David, B., Finn, C., Fu, C., Gopalakrishnan, K., Hausman, K., et al. Do as i can, not as i say: Grounding language in robotic affordances. *arXiv preprint arXiv:2204.01691*, 2022.

Amir, E. and Chang, A. Learning partially observable deterministic action models. *Journal of Artificial Intelligence Research*, 33:349–402, 2008.

Bacon, P.-L., Harb, J., and Precup, D. The option-critic architecture. In *Proceedings of the AAAI conference on artificial intelligence*, volume 31, 2017.

Chevalier-Boisvert, M., Bahdanau, D., Lahlou, S., Willems, L., Saharia, C., Nguyen, T. H., and Bengio, Y. BabyAI: First steps towards grounded language learning with a human in the loop. In *International Conference on Learning Representations*, 2019.

Chitnis, R., Silver, T., Tenenbaum, J. B., Lozano-Perez, T., and Kaelbling, L. P. Learning neuro-symbolic relational transition models for bilevel planning. In *2022 IEEE/RSJ international conference on intelligent robots and systems (IROS)*, pp. 4166–4173. IEEE, 2022.

Diuk, C., Cohen, A., and Littman, M. L. An object-oriented representation for efficient reinforcement learning. In *Proceedings of the 25th international conference on Machine learning*, pp. 240–247, 2008.

Eysenbach, B., Gupta, A., Ibarz, J., and Levine, S. Diversity is all you need: Learning skills without a reward function. *arXiv preprint arXiv:1802.06070*, 2018.

Fikes, R. E. and Nilsson, N. J. Strips: A new approach to the application of theorem proving to problem solving. *Artificial intelligence*, 2(3-4):189–208, 1971.

Fu, H., Sharma, P., Stengel-Eskin, E., Konidaris, G., Roux, N. L., Côté, M.-A., and Yuan, X. Language-guided skill learning with temporal variational inference. *arXiv preprint arXiv:2402.16354*, 2024.

Garrett, C. R., Lozano-Pérez, T., and Kaelbling, L. P. Pddlstream: Integrating symbolic planners and blackbox samplers via optimistic adaptive planning. In *Proceedings of the international conference on automated planning and scheduling*, volume 30, pp. 440–448, 2020.

Garrett, C. R., Chitnis, R., Holladay, R., Kim, B., Silver, T., Kaelbling, L. P., and Lozano-Pérez, T. Integrated task and motion planning. *Annual review of control, robotics, and autonomous systems*, 4(1):265–293, 2021.

Gregor, K., Rezende, D. J., and Wierstra, D. Variational intrinsic control. *arXiv preprint arXiv:1611.07507*, 2016.

Konidaris, G., Kaelbling, L. P., and Lozano-Perez, T. From skills to symbols: Learning symbolic representations for abstract high-level planning. *Journal of Artificial Intelligence Research*, 61:215–289, 2018.

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 *2023 IEEE International conference on robotics and automation (ICRA)*, pp. 9493–9500. IEEE, 2023.

Samvelyan, M., Kirk, R., Kurin, V., Parker-Holder, J., Jiang, M., Hambro, E., Petroni, F., Küttler, H., Grefenstette, E., and Rocktäschel, T. Minihack the planet: A sandbox for open-ended reinforcement learning research. *arXiv preprint arXiv:2109.13202*, 2021.

Sharma, A., Gu, S., Levine, S., Kumar, V., and Hausman, K. Dynamics-aware unsupervised discovery of skills. *arXiv preprint arXiv:1907.01657*, 2019.

Silver, T., Chitnis, R., Tenenbaum, J., Kaelbling, L. P., and Lozano-Pérez, T. Learning symbolic operators for task and motion planning. In *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 3182–3189. IEEE, 2021.

Silver, T., Athalye, A., Tenenbaum, J. B., Lozano-Pérez, T., and Kaelbling, L. P. Learning neuro-symbolic skills for bilevel planning. *arXiv preprint arXiv:2206.10680*, 2022.

Silver, T., Chitnis, R., Kumar, N., McClinton, W., Lozano-Pérez, T., Kaelbling, L., and Tenenbaum, J. B. Predicate invention for bilevel planning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 37, pp. 12120–12129, 2023.

Sutton, R. S., Precup, D., and Singh, S. Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Artificial intelligence*, 112(1-2): 181–211, 1999.

Wang, G., Xie, Y., Jiang, Y., Mandlekar, A., Xiao, C., Zhu, Y., Fan, L., and Anandkumar, A. Voyager: An open-ended embodied agent with large language models. *arXiv preprint arXiv:2305.16291*, 2023.

Yang, Q., Wu, K., and Jiang, Y. Learning actions models from plan examples with incomplete knowledge. In *ICAPS*, pp. 241–250, 2005.

Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K., and Cao, Y. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*, 2022.

A Additional Experiment Results

This appendix provides a detailed analysis of how PBD expands its skill library and how this expansion affects downstream task performance. We first show the overall growth of the learned skill library across the curriculum in Figures 5 and 9. We then report per-environment success rates under each intermediate library snapshot, showing how task coverage improves as new bridge skills are accepted in Tables 2 and 3. We also analyze skill reuse and success attribution to verify that the performance gains come from reusable learned skills rather than from the initial skill library alone in Figures 4 and 8. Finally, we provide case studies of gap detection, showing how PBD identifies the missing bridge predicate from a failed plan and uses it to guide subsequent skill learning.

A.1 Overall Skill Library Expansion

Figures 6 and 7 show how the skill library expands over the curriculum. As PBD proceeds through curriculum stages, newly encountered failures reveal bridge gaps. For each accepted bridge predicate, PBD trains and validates a corresponding skill, adds it to the library, and makes it available to all subsequent tasks. These figures trace the capability growth induced by the curriculum. As training progresses, PBD repeatedly encounters new predicate-level gaps, converts validated bridge predicates into executable skills, and incorporates them into the library. The expanding library is accompanied by a steady increase in overall task success, suggesting that the acquired skills improve the agent's effective task coverage rather than remaining isolated solutions to individual curriculum stages. The gains are especially reflected in later compositional tasks, where previously learned skills serve as reusable building blocks for longer predicate program.

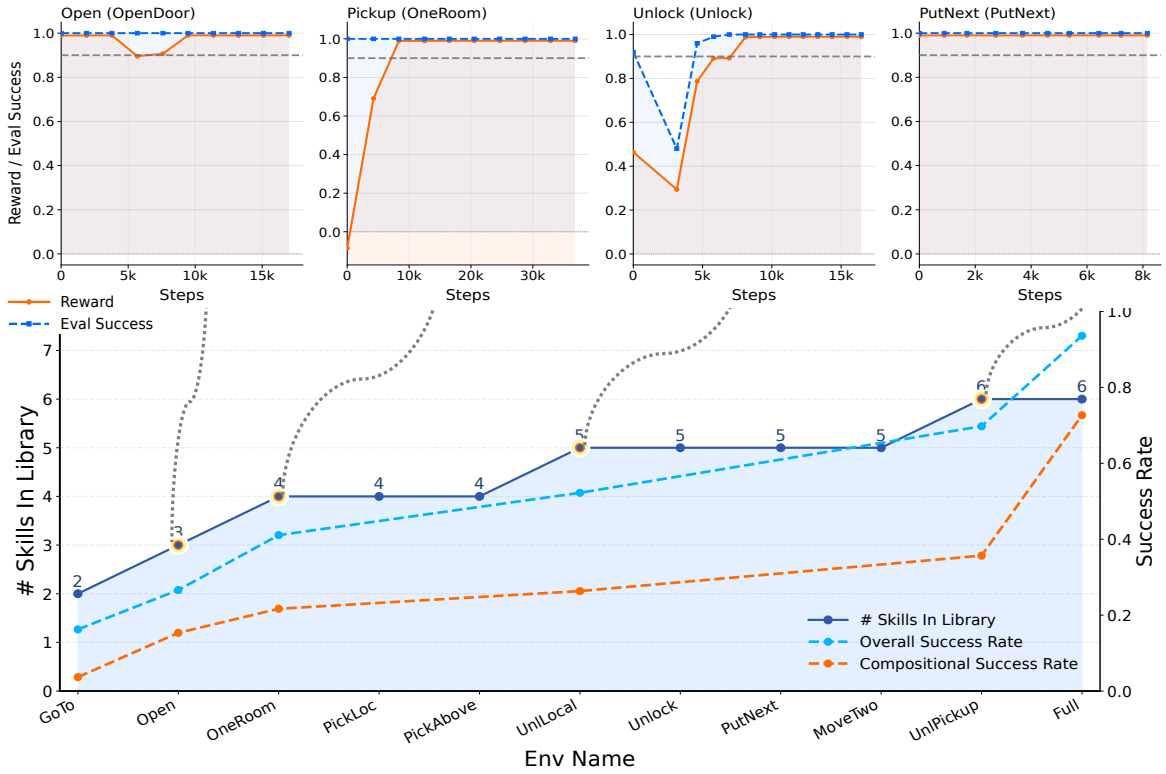


Figure 6. Skill library expansion over the BabyAI curriculum.

A.2 Performance Evolution with Skill-Library Growth

Tables 2 and 3 list all environments used in our experiment and report the per-environment success rates for each evaluated library snapshot. L_0 denotes the initial library, and L_i denotes the library after the i-th accepted bridge discovered by PBD. For each snapshot, we freeze the corresponding library and evaluate it on every environment for 100 episodes, with no additional skill discovery or training during evaluation.

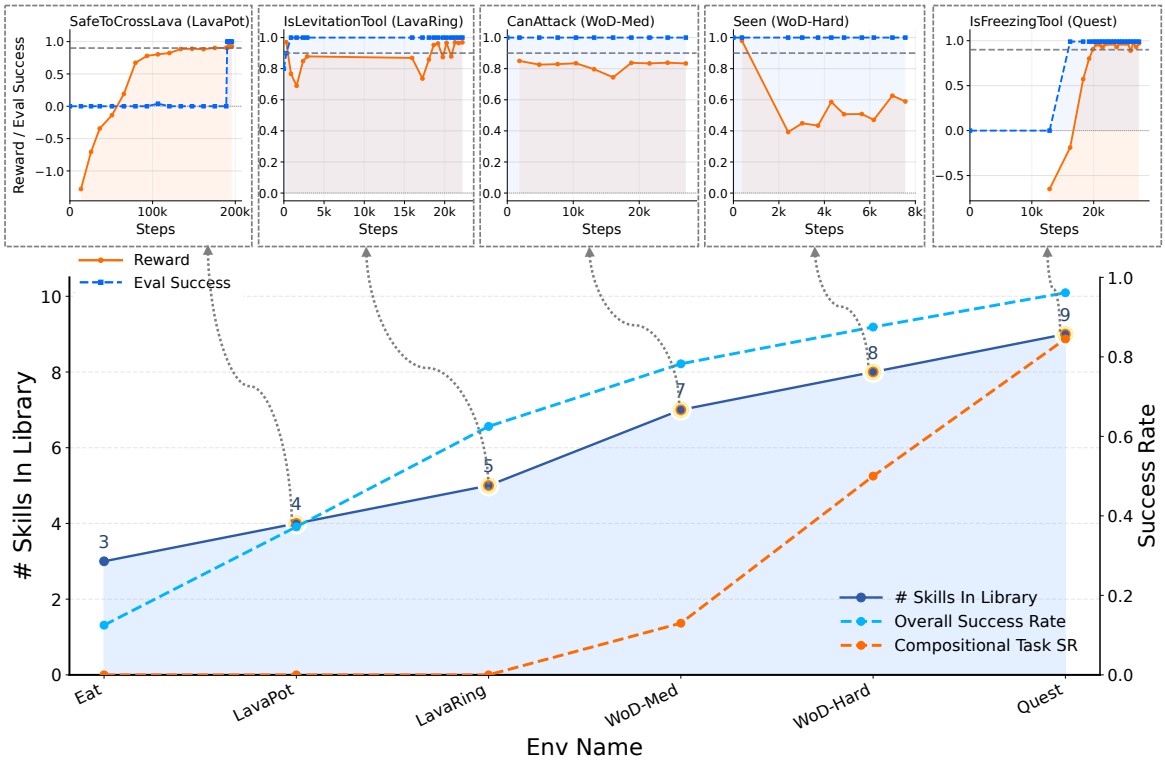


Figure 7. Skill library expansion over the Minihack curriculum.

The results show that skill-acquisition environments are typically solved once their corresponding bridge gaps have been filled. These tasks often expose a small number of localized missing predicates, so adding the relevant learned skill is sufficient to raise success to near-perfect levels. For example, tasks involving door opening, object pickup, unlocking, lava crossing, or tool use become solvable immediately after the corresponding skill is accepted into the library. This indicates that the learned skills close real execution gaps in the original library rather than merely improving performance through additional training.

The compositional environments exhibit a more gradual improvement pattern. Unlike local acquisition tasks, these environments require multiple learned skills to be ordered and reused within longer predicate programs. As the curriculum progresses and the library accumulates more bridge skills, success rates on compositional tasks such as MINIBOSSLEVEL, SYNTHSEQ, BOSSLEVEL, WOD-HARD, and QUEST-EASY steadily increase. This suggests that the learned skills remain useful beyond the tasks in which they were acquired and can be composed to cover a growing portion of the downstream task distribution.

Overall, these results provide a fine-grained view of capability expansion under PBD. The library first closes localized gaps in skill-acquisition environments, and these acquired skills then become reusable components for solving more complex compositional tasks. This detailed breakdown complements the aggregate results in the main text by showing how each stage of library growth expands the set of tasks the agent can successfully complete.

A.3 Skill Reuse Analysis

Figures 4 and 8 analyze whether the accepted bridge skills are reused after being added to the library and whether they contribute to final task success. For each library snapshot, the heatmap reports the fraction of evaluation episodes in which each learned skill is invoked at least once. This allows us to inspect whether an accepted bridge skill remains useful beyond the curriculum stage in which it was first learned.

The results show that learned skills are repeatedly invoked in later tasks rather than being isolated solutions to their original training environments. As the curriculum progresses, newly added skills become available to subsequent predicate programs

Table 2. Per-environment success rate after successive library updates for BabyAI. L_0 denotes the initial library. L_1–L_4 denote libraries after successive accepted bridge skills, and Full denotes the final library used for evaluation.

Environment	L_0	L_1	L_2	L_3	L_4	Full
GoToLocal	1.00	1.00	1.00	1.00	1.00	1.00
GoToObj	1.00	1.00	1.00	1.00	1.00	1.00
OpenDoor	0.00	1.00	1.00	1.00	1.00	1.00
PickupLoc	0.00	0.00	1.00	1.00	1.00	1.00
PickupAbove	0.00	0.00	0.69	1.00	1.00	1.00
UnlockLocal	0.00	0.00	0.00	1.00	1.00	1.00
UnlockPickup	0.00	0.00	0.00	0.00	1.00	1.00
BlockedUnlockPickup	0.00	0.00	0.00	0.00	1.00	1.00
PutNextLocal	0.00	0.00	0.00	0.00	0.00	1.00
MoveTwoAcrossS5N2	0.00	0.00	0.00	0.00	0.00	0.99
MiniBossLevel	0.06	0.25	0.36	0.45	0.62	0.78
SynthSeq	0.00	0.00	0.00	0.00	0.00	0.73
BossLevel	0.05	0.21	0.29	0.34	0.45	0.67

Table 3. Per-environment success rate after successive library updates for MiniHack. L_0 denotes the initial library. L_1–L_5 denote libraries after successive accepted bridge skills, and Full denotes the final library used for evaluation.

Environment	L_0	L_1	L_2	L_3	L_4	L_5	Full
MiniHack-Eat	0.00	1.00	1.00	1.00	1.00	1.00	1.00
MiniHack-Lava-Ring-Inv	0.00	0.00	0.00	1.00	1.00	1.00	1.00
MiniHack-Lava-Potion-Inv	0.00	0.00	1.00	1.00	1.00	1.00	1.00
MiniHack-Lava-Ring-Pickup	0.00	0.00	0.00	1.00	1.00	1.00	1.00
MiniHack-Lava-Potion-Pickup	0.00	0.00	0.98	1.00	1.00	1.00	1.00
MiniHack-WoD-Medium	0.00	0.00	0.00	0.00	1.00	1.00	1.00
MiniHack-WoD-Hard	0.00	0.00	0.00	0.00	0.26	1.00	1.00
MiniHack-Quest-Easy	0.00	0.00	0.00	0.00	0.00	0.00	0.69

and are reused in combination with previously acquired skills. This pattern indicates that PBD expands the library with transferable predicate-level capabilities, instead of accumulating one-off behaviors tied to specific instances.

The success-attribution bars provide a complementary view. They decompose evaluation episodes into three categories: successes achieved using only the initial base skills, successes that require at least one learned skill, and failures. Across library snapshots, the fraction of successes involving learned skills increases, while the failure fraction decreases. In contrast, successes using only base skills remain limited. This suggests that the learned skills are not merely added to the library, but are actively responsible for solving tasks that the initial skill set cannot handle

A.4 Additional Gap Detection Example

Figures 5 and 9 provide additional case studies of gap detection in BabyAI and MiniHack. Both examples show how PBD compares the predicates reachable under the current library with the predicates required by the active subgoal, and then selects the missing bridge predicate as the next skill-learning target. The selected bridge predicate is then used as the target effect for learning the corresponding attack-enabling skill. After this skill is learned and added to the library, the agent can close the gap and proceed toward the final goal.

B Additional Ablation Study

B.1 Learned Predicate Grounding

To assess whether PBD requires hand-coded predicate checkers, we replace the rule-based extractors with learned predicate grounders. Each grounder predicts predicate truth values from domain observations, using the same predicate vocabulary as the rule-based checker. We implement the learned grounder as a query-conditioned binary classifier, where a shared three-layer MLP takes an observation encoding and a grounded predicate query as input and predicts the truth probability of that predicate. The query features encode the predicate name, argument attributes, and relative geometry between the agent and predicate arguments. The grounder is trained for 20 epochs on 100 episodes, with training labels generated automatically

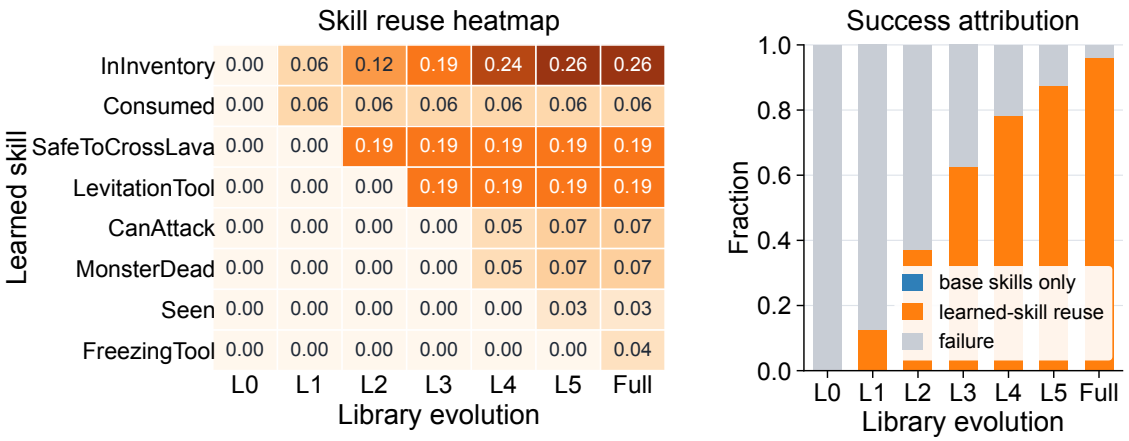


Figure 8. Skill reuse and success attribution across library growth in MiniHack. Heatmaps report the fraction of evaluation episodes in which each learned skill is invoked at least once under each library snapshot. Bar plots decompose episodes into successes using only base skills, successes using at least one learned skill, and failures. As the library grows, learned-skill reuse increases and the failure rate decreases.

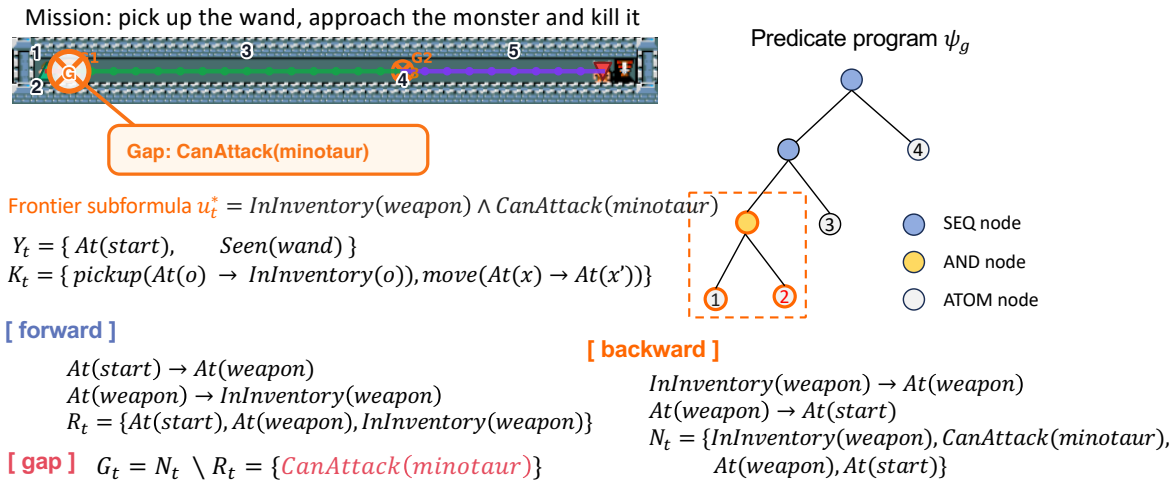


Figure 9. Case study of PBD in MiniHack-WoD-Medium.

by the rule-based checker. Table 4 reports held-out predicate prediction performance and shows that the learned grounder closely matches the rule-based checker on the evaluated predicates.

We further evaluate whether PBD can operate with learned predicate estimates during bridge detection and skill acquisition. For example, in the BabyAI-OpenDoor environment, using the learned grounder leads PBD to identify the same missing bridge predicate as the rule-based checker. PBD then trains the corresponding OpenDoor skill using grounder-based reward, termination, and validation signals. The learned skill reaches 100% real-environment evaluation success, matching the oracle-checker version with comparable training cost. This result suggests that learned predicate estimates can support PBD gap detection and skill acquisition.

B.2 Curriculum and Warm-up for Skill-Policy Training

We analyze two factors in bridge-skill policy training: whether the skill is trained in a localized curriculum environment, and whether PPO is initialized with warm-up demonstrations. This gives four settings: curriculum with warm-up, curriculum without warm-up, direct training without curriculum with warm-up, and direct training without curriculum without warm-up. This ablation focuses on the stage after a missing predicate has already been selected. All variants train a policy for the same target bridge predicate and use the same predicate reward, termination condition, and validation protocol, but differ in

Predicate	Precision	Recall	F1
`Adjacent`	1.00	1.00	1.00
`DoorOpened`	1.00	1.00	1.00
`DoorLocked`	1.00	1.00	1.00
`CanUnlock`	1.00	1.00	1.00
`HoldingObj`	1.00	1.00	1.00
`HoldingKey`	1.00	1.00	1.00
`HandEmpty`	1.00	1.00	1.00
`Seen`	1.00	1.00	1.00
`CanPutNext`	0.79	0.74	0.76
Overall	0.9999	0.9998	0.9998

Table 4. Learned predicate grounding results on held-out BabyAI states. The learned classifiers predict predicate truth values from symbolic state vectors using labels automatically generated by the rule-based checker.

whether the policy is initialized with demonstration trajectories before PPO training.

In the *with warm-up* setting, the skill policy is initialized using demonstration transitions collected for the corresponding bridge predicate. In the *without warm-up* setting, the same PPO skill learner is trained from scratch. The skill-training curves for representative bridge skills in both BabyAI and MiniHack are provided in Figure 10.

B.3 Different library threshold.

PBD inserts a trained bridge skill into the library only if its validation success rate exceeds the threshold ρ_{val}, and we set $\rho_{\text{val}} = 0.9$ in all experiments. This threshold is used as a fixed reliability gate for library construction, rather than tuned on the final compositional evaluation tasks.

The threshold controls a trade-off between library coverage and skill reliability. A low threshold can admit more candidate skills, but some of them may only satisfy the target predicate inconsistently. This is harmful in compositional tasks because the planner treats accepted skills as executable symbolic operators. If an unreliable skill is used inside a long-horizon plan, a single failed skill execution can invalidate all later subgoals. On the other hand, an overly strict threshold can reject useful skills due to occasional stochastic validation failures. This reduces library coverage and may leave later predicate programs unsolved even when a usable bridge skill has been learned. We therefore choose $\rho_{\text{val}} = 0.9$ as a conservative reliability gate: it is high enough to filter unstable skills, but not so strict that it requires perfect validation success.

Table 5 reports the effect of varying this threshold.

Table 5. Sensitivity to the library acceptance threshold. Thresholds up to 0.9 accept the same library, while stricter thresholds reject useful MiniHack skills and reduce MiniHack success. We use $\rho_{\text{val}} = 0.9$ in all experiments.

ρ_{val}	Accepted skills	Rejected skills	BabyAI SR	MiniHack SR
0.70	13/13	0	0.936	0.961
0.80	13/13	0	0.936	0.961
0.90	13/13	0	0.936	0.961
0.95	12/13	1	0.936	0.711
1.00	11/13	2	0.936	0.625

C Implementation and Experimental Setup Details

C.1 Curriculum and Environment Split

Figure 11 provides a visual summary of this curriculum for both BabyAI and MiniHack. Table 6 shows the full curriculum and environment split used in the experiments. Each row lists the main bridge predicate exposed by that stage and the corresponding skill added to the library after validation. The skill-acquisition stages are designed to provide localized training contexts for individual bridge skills, while the compositional stages require multiple learned skills to be ordered and reused. For MiniHack, some later environments, such as WoD-Hard, both reuse previous skills and expose additional missing predicates during training.

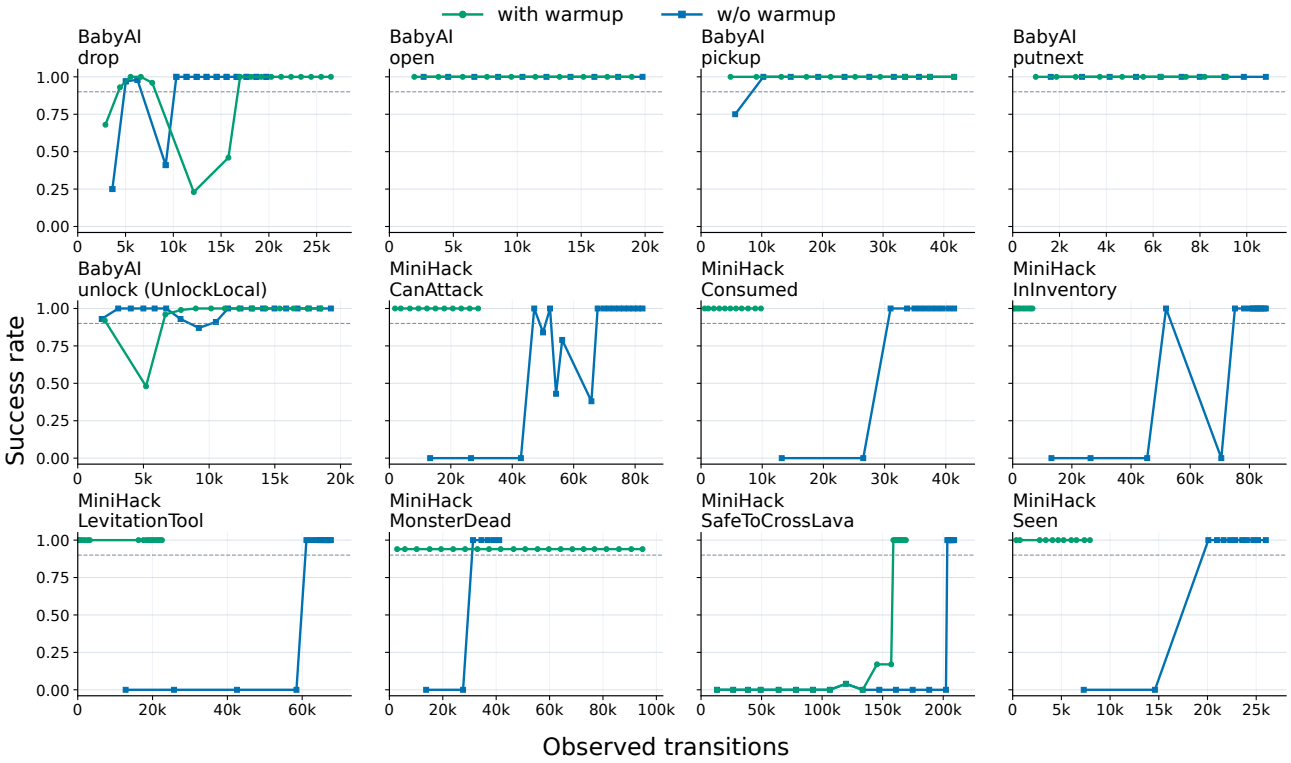


Figure 10. Effect of behavior-cloning warm-up on bridge-skill training under curriculum learning. Each panel shows the validation success rate of one learned bridge skill as a function of observed environment transitions. We compare PPO training initialized with warm-up (with warmup) against PPO from scratch (w/o warmup). BC warm-up generally improves early learning speed and stability, while both variants can eventually reach high predicate-level success for many bridge skills.

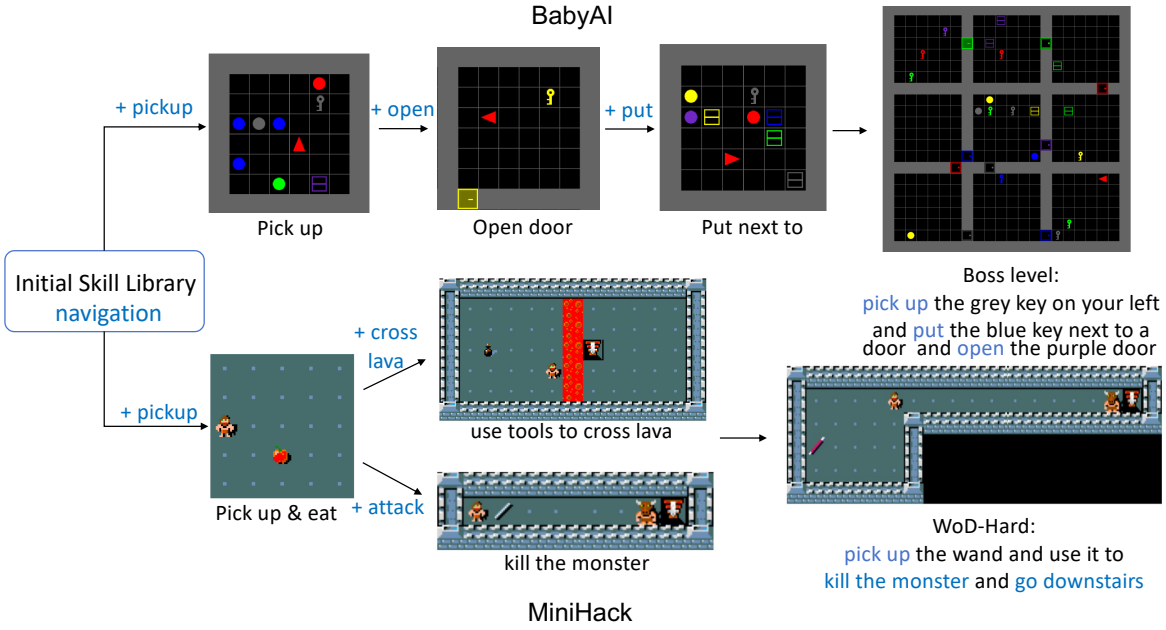


Figure 11. Illustrative curriculum for skill accumulation and compositional reuse.

C.2 Symbolic Predicate Grounding

PBD operates in a fixed predicate semantic space defined by a domain-specific predicate vocabulary. This vocabulary specifies the predicate schemas used by task programs, runtime state checking, skill preconditions, skill effects, and bridge-gap detection. For each task instance, the predicate schemas are grounded with the concrete entities in that instance to form the grounded predicate set P. The predicate checker C maps an environment state s_t to the subset of grounded predicates that are true $C(s_t) \subseteq P$. The checker therefore provides the symbolic state used by PBD. In our experiments, C is implemented as an environment-level checker, rather than a learned latent predictor or a reward-derived signal. The task predicate program ψ_g is constructed using the same predicate vocabulary. Thus, predicates appearing in the program have the same semantics as predicates evaluated by the checker. At execution time, PBD evaluates the active frontier of ψ_g against the current predicate state $C(s_t)$.

In BabyAI and MiniHack, we use different environment-specific implementations of the same checker interface. BabyAI predicates are grounded from the MiniGrid symbolic state, while MiniHack predicates are grounded from NLE observations and auxiliary environment state. These implementations differ only in how predicate truth is read from the environment. The downstream PBD procedure remains unchanged because task parsing, state checking, reachability, need computation, and skill validation all operate over the same fixed predicate semantic space.

C.3 Training Protocols for baselines

For imitation learning, BabyAI uses 10k official demonstration episodes, while MiniHack uses 40 manually collected demonstration episodes per environment. For flat reinforcement learning, BabyAI uses the PPO implementation from the BabyAI benchmark, and MiniHack uses the native PolyBeast baseline. Both flat RL baseline are trained as full-task policies for roughly 10^7 environment steps, without access to PBD's predicate-level bridge diagnosis.

The HRL baseline uses the same fixed skill library available at PBD evaluation time and trains a high-level actor-critic controller over grounded skills and composite options. The high-level controller is a two-layer MLP actor-critic with 512-dimensional hashed symbolic input features and hidden size 256, trained with Adam (3×10^{-4}), discount factor $\gamma = 0.99$, entropy coefficient 0.01, and value-loss coefficient 0.5. Training uses 200 updates with 16 episodes per update. This gives HRL access to the same executable skills used by PBD at test time, but it does not perform bridge discovery, train new bridge skills, or expand the skill library during learning. Thus, HRL tests whether a learned option-selection policy over a fixed library is sufficient for the compositional tasks. In future work, we will further compare PBD with skill-discovery methods such as LAST (Fu et al., 2024) under a unified experimental protocol.

Table 6. Full curriculum and environment split used in the experiments. Skill-acquisition stages expose localized missing predicates, while compositional stages evaluate reuse of the learned skill library.

Stage	Missing predicate	Learned skill
BabyAI		
BabyAI-GoToLocal / GoToObj	`Adjacent(obj)`	NAVIGATE
BabyAI-PickupLoc	`HoldingObj(obj)`	PICKUP
BabyAI-OpenDoor / UnlockLocal	`DoorOpened(door)`	OPENDOOR
BabyAI-UnlockPickup	`HoldingKey(key)`	UNLOCK
BabyAI-PutNextLocal	`NextTo(obj1,obj2)`	PUTNEXTTO
BabyAI-MiniBossLevel / SynthSeq	–	compositional reuse
BabyAI-BossLevel	–	compositional reuse
MiniHack		
MiniHack-Eat	`InInventory(obj),Consumed(food)`	PICKUP, EAT
MiniHack-LavaCross-Levitate-Ring-Inv	`IsLevitationTool(ring)`	USERINGTOLEVITATE
MiniHack-LavaCross-Levitate-Potion-Inv	`IsLevitationTool(potion)`	USEPOTIONTOLEVITATE
MiniHack-WoD-Medium	`MonsterDead(monster)`	ATTACK
MiniHack-WoD-Hard	`CanAttack(monster)`	EXPLORE / compositional reuse
MiniHack-Quest-Easy	–	compositional reuse

Table 7. Hyperparameters used for bridge-skill policy training. The library acceptance threshold is applied to held-out validation rollouts before a trained skill is inserted into the reusable skill library.

Hyperparameter	Value
Batch size	32
Warm-up epochs	5
Skill horizon	64
Learning rate	3×10^{-4}
GAE parameter λ	0.99
Entropy coefficient	10^{-2}
Value loss coefficient	0.5
Gradient clipping	1.0
PPO rollout size	16 episodes
PPO optimization epochs	4
PPO clipping coefficient	0.2
PPO target KL	0.05
Library acceptance threshold ρ_{val}	0.9
Step penalty	-0.01
Success reward	1.0
Early stopping criterion	10 successful evaluation episodes

C.4 Bridge Skill Policy Training

When PBD detects a missing bridge predicate $p^\star$, it constructs a candidate bridge skill whose symbolic form is a precondition-to-effect transition. The effect of the candidate skill is the missing predicate $p^\star$, and its preconditions are obtained from the executable prefix and the local support predicates identified before the failure. During each failed episode, PBD records the sequence of skills that can still be successfully executed before reaching the gap. These recorded skills are later used as a setup policy for training the new bridge skill.

For each bridge-skill rollout, we first execute the recorded prefix skills to bring the environment into an initiation region where the candidate preconditions hold. The new low-level policy is then trained only on the remaining missing transition, namely making the target predicate true from this initiation region. Thus, the bridge policy is not trained to solve the entire original task from the initial state. Instead, it is trained on the localized execution gap isolated by predicate bridge detection.

Importantly, the target is not tied to the particular object instance that exposed the failure. Although the gap is first observed as a grounded predicate, PBD generalizes it into a predicate template and matches this template against all valid entities in the environment. During training, different bindings of the same predicate template can be sampled, so the skill learns to achieve the predicate-level effect rather than memorizing a specific object or location. This parameterization allows one

learned bridge skill to be reused across different task instances that require the same capability.

We optimize the bridge policy with PPO using sparse predicate rewards. The policy receives a success reward when the target predicate becomes true and a small step penalty otherwise. Before PPO, we optionally apply behavior-cloning warm-up for 5 epochs, using 100 demonstration episodes per skill in BabyAI and 40 demonstration episodes per skill in MiniHack. These demonstrations are used only for initialization and are not used during PPO updates. After warm-up, the skill continues training through online environment interaction. A learned skill is added to the library only when its validation success rate over held-out predicate bindings reaches the fixed threshold $\rho_{\text{val}} = 0.9$. The acceptance criterion is therefore predicate success under the generalized skill template, rather than reward on the original task instance that triggered the failure..

C.5 Training Hyperparameters

Table 7 summarizes the default skill policy training settings. Our experiments were conducted on a workstation with 4 NVIDIA RTX 3090 GPUs, each with 24GB memory. Training runs for the proposed model and imitation baselines were typically executed on a single GPU, with different environments or random seeds run in parallel across GPUs when available.

D Predicate-Program Planner prompt

We use a fixed constrained prompt to compile each instruction and fully observed symbolic state into a grounded predicate-program tree. The full prompts of BabyAI and MiniHack are shown in the following two boxes.

Planner prompt for MiniHack

```
You are an expert planner for the MiniHack environment. Your task is to read the initial state of a
SkillHack task and produce a high-level plan predicate tree.
The plan should outline the key state changes described by predicates that must be satisfied to complete the
task, based on the initial map and state information provided.
Predicate vocabulary:
{predicate vocabulary}
Compositional operators:
- AND
- SEQ
The style of the parser tree should be:
- Return one JSON tree with fields: type, predicate, children.
- type is ATOM, SEQ, or AND.
- Use SEQ when one subgoal must happen before the next.
- Use AND only when subgoals are independent and order-free.
- Use exact entity/terrain ids from the state whenever an argument refers to an object, monster, door,
stairs, or terrain region.
- Use the natural-language map_description as the primary map information; do not infer object types from
NetHack symbols.
- Use clear high-level predicates that describe completed skill effects or required task milestones.
- Do not output low-level NetHack actions or explanations.
- For ground-object interactions such as `pickup`, the agent should stand on the item's location, which
corresponds to predicate `At(item)`.
- For inventory-based interactions such as `zap`, `wear`, `quaff`, `wield`, or `apply`, the item only needs
to be in the agent's inventory, which corresponds to predicate `InInventory(item)`.
- For monster-killed, when the agent seen the monster, it should attack the monster.
For example:
Mission: Get the wand, defeat the monster, then reach the stairs.
For the Wand-of-Death task, you should zap the wand to kill the monster and finally go to the down stair.
If there isn't a wand in the inventory, you should first find the wand and pick it up, and then go near to
the monster and zap your wand in inventory to the monster.
A good output is like:
{
  "type": "SEQ",
  "predicate": null,
  "children": [
    {"type": "ATOM", "predicate": {"name": "At", "args": ["wand"]}, "children": null},
    {"type": "ATOM", "predicate": {"name": "InInventory", "args": ["wand"]}, "children": null},
    {"type": "ATOM", "predicate": {"name": "Seen", "args":["monster"]}, "childern":null},
    {"type": "ATOM", "predicate": {"name": "MonsterDead", "args":["monster"]}, "children":null},
    {"type": "ATOM", "predicate": {"name": "At", "args": ["stairs"]}, "children": null},

  ]
}
State summary:
{state}
```

Planner prompt for BabyAI

```
You are a symbolic instruction compiler and instance-specific planner for BabyAI-style grid environments.

Your job is to convert:
1. a natural-language instruction
2. a fully observed environment description for one concrete map
into a COMPLETE, INSTANCE-SPECIFIC symbolic predicate program tree for THIS MAP.

Your output must be a concrete symbolic plan over the exact runtime object ids used by the environment.

Important:
- You can only use the atomic predicates and the items appear in the map.
- You should fully understand the mission, respect the explicit clause order first.
- As long as you need to cross a door to enter another room, the door should be added to the plan.
Atomic predicates:
{predicate vocabulary}
Compositional operators:
- AND
- SEQ
Semantic conventions:
- Adjacent(X) means the agent will go to X
- HoldingObj(X) means the agent is carring X, and HoldingKey(K) means X is a key.
- DoorClosed(D) means the door is closed but not locked; and DoorLocked(D) means D is locked.
- NextTo(X, Y) means the item X is next to item Y
- CanUnlock(key, door) means the key matches the door by color
Instruction interpretation rules:
- The instruction follows the BFN grammar:
 ⟨Sent⟩ |= ⟨Sent1⟩ | ⟨Sent1⟩ ',' then ⟨Sent1⟩ | ⟨Sent1⟩ after you ⟨Sent1⟩
 ⟨Sent1⟩ |= ⟨Clause⟩ | ⟨Clause⟩ and ⟨Clause⟩
 ⟨Clause⟩ |= go to ⟨Descr⟩ | pick up ⟨DescrNotDoor⟩ | open ⟨DescrDoor⟩ |put ⟨DescrNotDoor⟩ next to ⟨Descr⟩
  this tells you how to decompose the instruction.
- Compile explicit clauses first, then insert only the minimum hidden prerequisites for each clause.
- When the instruction has temporal composition, default to SEQ, not AND.
Output STRICT JSON ONLY.
JSON format:
Internal nodes:
{
  "type": "SEQ" | "AND",
  "children": [ ... ]
}
Leaf nodes:
{
  "type": "ATOM",
  "predicate": {
    "name": "Adjacent" | "HoldingKey" | "HoldingObj" | "DoorOpened" |"HandEmpty" | "At" | "CanPutNext" |
    "NextTo" | "CanUnlock",
    "args": ["..."]
  }
}
Examples:
{examples}
Output STRICT JSON ONLY.
Do not output markdown.
Do not output explanations.
```