LEARNING WHAT'S MISSING: FAILURE-DRIVEN SKILL DISCOVERY VIA PREDICATE BRIDGES

Anonymous authors
Paper under double-blind review

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 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. A more detailed review of related work is presented in Appendix A.

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

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.

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 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.

2 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.

2.1 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.

2.2 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.

2.3 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.

2.4 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?*

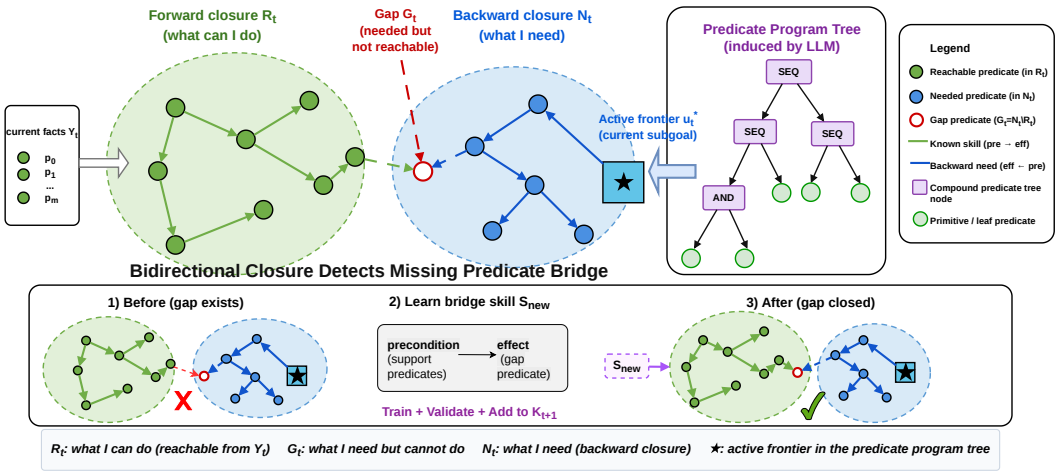

Figure 1: Overview of bidirectional Predicate Bridge Discovery. Given the current predicate facts Y_t and the predicate program tree induced by the instruction, PBD performs forward closure over the current skill library to obtain the reachable set R_t and backward closure from the active frontier $u_t^\star$ 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.

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

PBD therefore turns to skill acquisition. It selects from G the first missing predicate on the active dependency branch whose preceding predicates are already satisfied or reachable.

We denote the selected bridge predicate by $p^\star$, where

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

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 where PBD differs 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.

3 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 asked to learn a policy that makes the missing predicate true rather than to solve the original long-horizon task from scratch.

3.1 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:

`BarrierCleared(barrier_2)` $\mapsto$ `BarrierCleared(x)`.

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

After selecting a missing predicate $p^\star$, PBD uses it as the effect of a candidate skill and derives reachable support conditions as its preconditions. These preconditions define the candidate skill's initiation conditions and the low-level policy $\pi^\star$ is trained to achieve $p^\star$ from such states. The resulting skill is $\kappa^\star = \langle o^\star, \pi^\star, \rho^\star \rangle$, where $o^\star$ contains the generalized preconditions and effect, and $\rho^\star$ is estimated through validation before library admission. This yields predicate-level skills reusable across object bindings, rather than instance-specific repairs.

3.2 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:

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

The skill-learning objective is

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

During training, the policy receives a reward of 1 at the first time the target predicate becomes true, and each executed action also incurs a penalty of λ_{step}. The resulting reward is

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

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

Although predicate achievement provides only a sparse positive signal, the bridge formulation substantially narrows the learning problem compared with solving the original task from scratch. The target predicate is local, the initial states are selected to satisfy the candidate preconditions, and the policy is trained specifically on the missing capability identified through symbolic analysis. In this way, failure provides useful supervision by defining a low-level learning problem that is both more tractable and more reusable than the original failed task.

3.3 Validation Across Bindings

A candidate skill is evaluated on multiple bindings of its target predicate before it is added to the library. This validation tests whether the learned policy can achieve the intended effect beyond the instance that triggered learning.

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 define the validation success rate as

$$\hat{\rho}_{\text{val}}(\kappa^\star) = \frac{1}{|\mathcal{B}_{\text{val}}| M} \sum_{b \in \mathcal{B}_{\text{val}}} \sum_{m=1}^{M} \mathbb{I}\left[\exists t \le H \;\text{ s.t. }\; p^\star[b] \in C(s_t^{(m)})\right].$$

The skill is accepted if $\hat{\rho}_{\text{val}}(\kappa^\star) \ge \rho_{\text{val}}$. 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 compute the validation rate across all sampled bindings. A candidate is added to the library when this rate meets the acceptance threshold, providing evidence that its policy can achieve the target predicate under different bindings and be reused in later tasks.

3.4 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. When more complex instructions expose additional gaps, PBD repeats the procedure by localizing the first missing predicate, training the corresponding skill, validating it across bindings, and replanning.

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 only after validation across bindings, PBD grows the library through targeted, interpretable, and compositional skills.

4 Experiment

4.1 Experimental Setup

We evaluate PBD on BabyAI (Chevalier-Boisvert et al., 2019) and MiniHack (Samvelyan et al., 2021). These experiments test whether 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. BabyAI provides controlled gridworld tasks involving objects, colors, doors, keys, and spatial relations. We use ten simple tasks for skill acquisition, followed by three compositional tasks—MiniBossLevel, SynthSeq, and BossLevel. MiniHack introduces richer dynamics such as inventory, monsters, and hidden preconditions. We use six skill-acquisition tasks and two compositional tasks, WoD-Hard and Quest-Easy. Figure 2 provides a schematic illustration of the BabyAI curriculum. The full environment lists and curriculum stages are provided in Appendix D.1.

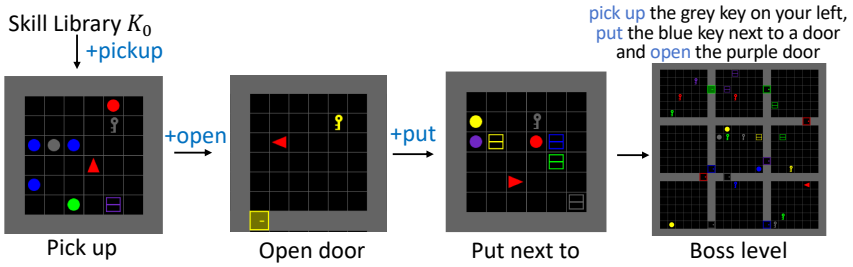

Figure 2: Illustrative BabyAI curriculum for skill accumulation and compositional reuse.

Baselines. We compare PBD with imitation learning (IL), flat RL, fixed-library HRL, and Voyager (Wang et al., 2023). Training protocols are detailed in Appendix D.4 and D.5.

4.2 Main Results

We evaluate whether PBD can identify and acquire the missing bridge skills and reuse them to solve compositional tasks. We first report PBD's final success rates on complex compositional tasks

alongside the baselines. We then examine how success changes as PBD discovers and learns bridge skills over the curriculum, and whether those skills are reused in successful executions. Complete results of all environments are provided in Appendix B.2.

Compositional Task Success Rate. For each task, we compute the success rate as the fraction of 100 held-out evaluation episodes in which the full task is completed. PBD's learned skill library is frozen during evaluation. Table 1 reports success rates on compositional environments and PBD outperforms baselines across all reported environments. Compared with the strongest baseline (Voyager), PBD improves success by 10–13 percentage points on BabyAI and by 11 points on MiniHack-QUEST-EASY and 59 points on MiniHack-WOD-HARD.

Table 1: Success rate on compositional environments over 100 held-out episodes.

Environment	IL	RL	HRL	Voyager	PBD
BabyAI-MiniBossLevel	0.36	0.10	0.13	0.66	**0.78**
BabyAI-SynthSeq	0.26	0.02	0.08	0.63	**0.73**
BabyAI-BossLevel	0.30	0.01	0.07	0.54	**0.67**
MiniHack-WoD-Hard	0.00	0.00	0.14	0.41	**1.00**
MiniHack-Quest-Easy	0.5	0.00	0.38	0.58	**0.69**

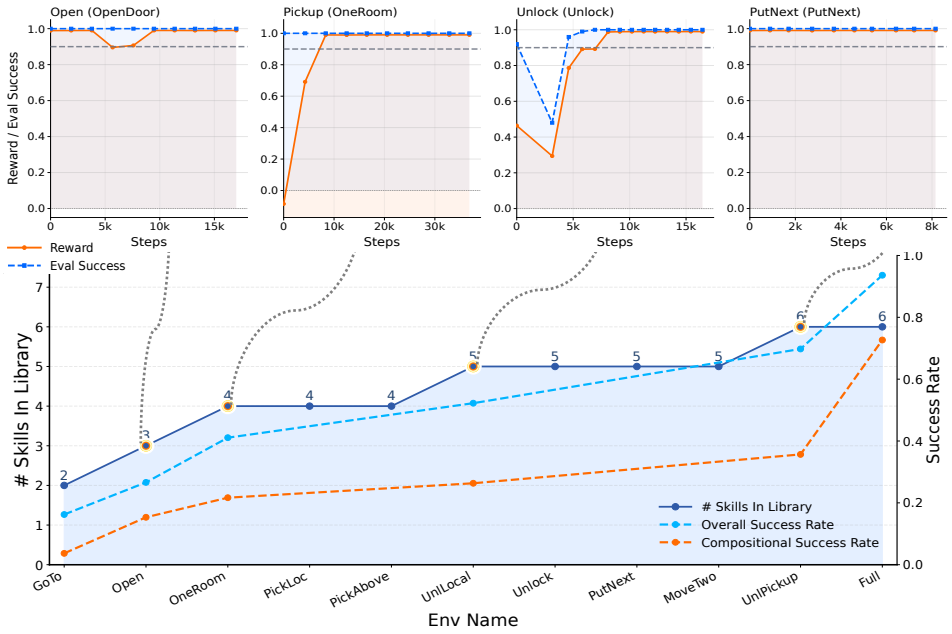

Figure 3: Skill library expansion over the BabyAI curriculum.

Skill Library Growth and Skill Reuse. We examine how curriculum training expands the executable skill library and whether the acquired skills are reused in compositional tasks. Figure 3 and 6 (Appendix B) evaluate successive library snapshots as validated bridge skills are added during the curriculum. As the number of available skills increases, both the success rate on compositional tasks and the overall success rate across all evaluated environments increase. The increase on compositional tasks is particularly relevant because these tasks require capabilities acquired earlier from local predicate gaps. Later library snapshots can therefore solve compositional episodes that remain unsuccessful with the initial library.

The reuse heatmaps in Figure 4 and 7 (Appendix B) show how these skills are used as the library expands. Newly acquired skills are invoked in subsequent tasks rather than remaining unused in the library. In BabyAI, PICKUP, OPENDOOR, and PUTNEXTTO are first learned to resolve local

predicate gaps and are later invoked in MINIBOSSLEVEL, SYNTHSEQ, and BOSSLEVEL. The MiniHack heatmap likewise shows reuse of skills for inventory, lava traversal, tool use, and combat. Together, the snapshot evaluations and invocation records show that improvements in success occur alongside the acquisition and reuse of executable bridge skills.

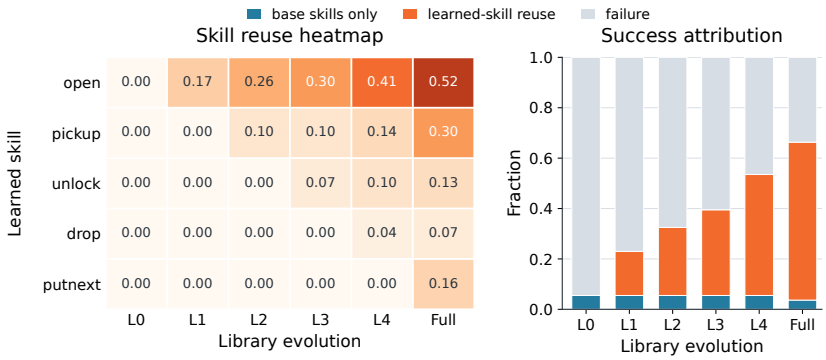

Figure 4: Skill reuse and success attribution across library growth in BabyAI. Heatmaps report the fraction of evaluation episodes in which each learned skill is invoked at least once under each library snapshot. Bar plots decompose episodes into successes using only base skills, successes using at least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

Training budgets. We report supervised and online training frames separately in Table 2. For PBD, supervised frames cover behavior-cloning warm-up, while online frames cover PPO training of bridge skills and those used for periodic skill validation during acquisition. Under this accounting, PBD achieves higher success with fewer online transitions than full-task RL. Rather than repeatedly exploring the entire compositional task, PBD focuses interaction on learning bridge skills and reuses them in later tasks.

Table 2: BabyAI-BossLevel training budgets and success rates over 100 held-out episodes. Frame counts are reported in thousands (k).

Method	Supervised frames	Online frames	SR@100
IL	16,400	0	0.30
RL	0	14,016	0.01
HRL	0	7,043	0.07
PBD	7	114	0.67

4.3 CASE STUDY

Figure 5 illustrates one BabyAI-BossLevel case and the corresponding bridge detection process. This case requires the agent to open a blue door and then reach a blue key. With the current library, the agent can use a movement skill to reach the door, making `Near(door)` reachable. Completing the predicate program also requires `DoorOpened(door)`, but no available skill can establish it. Forward reasoning therefore reaches the door, while backward reasoning identifies opening it as a necessary step. The mismatch isolates `DoorOpened(door)` as the bridge gap.

PBD uses this gap to define a candidate skill that achieves the effect from a state in which the precondition holds. After the skill is learned and added to the library, the agent can open the door, continue with the remaining plan, and complete the task successfully. Figure 8 (Appendix B) provides an additional case study of MiniHack.

4.4 ABLATION STUDIES

We ablate curriculum-based skill acquisition by comparing PBD-curr with PBD-direct. The two variants share the same PBD pipeline and differ only in the skill-training environment. PBD-direct

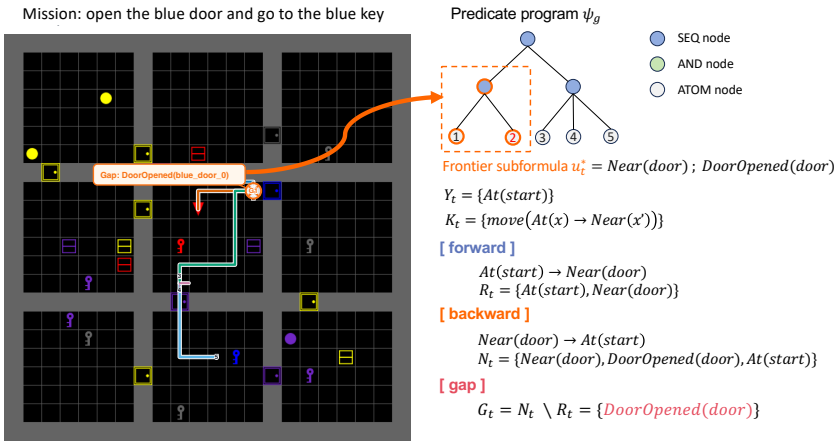

Figure 5: BabyAI-BossLevel case study. PBD detects `DoorOpened(door)` as the missing bridge predicate because it is needed by the active frontier but not reachable with the current library. After the corresponding skill is learned and added to the library, the gap is closed and the task succeeds.

trains them directly in the target compositional environment, while PBD-curr trains them in simpler acquisition environments before reuse. Additional ablation studies are provided in Appendix C.

As shown in Table 3, removing the curriculum substantially reduces success rates on the three BabyAI compositional tasks by 26–33 percentage points. Compared with curriculum acquisition, direct training requires the agent to learn skills for the same predicates in longer-horizon compositional environments, where useful initiation states are harder to reach. On MiniHack, PBD-direct matches PBD-curr 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.

Table 3: Compositional task success with and without curriculum.

	BabyAI			MiniHack	
Method	MiniBossLevel	SynthSeq	BossLevel	WoD-Hard	Quest-Easy
PBD-direct	0.46	0.40	0.41	1.00	0.69
PBD-curr	0.78	0.73	0.67	1.00	0.69

5 CONCLUSION AND LIMITATIONS

PBD is a failure-driven framework that learns missing capabilities from failed plans. When execution encounters a gap, PBD identifies a bridge predicate, trains and validates a candidate parameterized skill, and adds the validated skill to an executable and reusable library. In this way, PBD incrementally expands its capabilities rather than relying on task-specific, one-off fixes. Experiments on BabyAI and MiniHack show that PBD acquires reliable skills and compositionally reuses them to solve complex tasks, achieving higher task success rates than the evaluated baselines. By localizing failures, PBD reduces long-horizon end-to-end learning to targeted skill acquisition, resulting in substantially lower training cost. Each acquired skill is also associated with an explicit missing predicate, providing an interpretable record of how the agent's capabilities evolve. While the current implementation uses rule-based predicate extractors and structured predicate programs, these design choices provide a controlled foundation for evaluating the core PBD framework. Extending PBD to learned predicate grounding and noisier symbolic environments is a natural next step that could further broaden its applicability to more open-ended domains.

AI USE STATEMENT

In this work, we used generative AI tools to polish our manuscript, including improving its organization, clarity, and readability. We have reviewed all AI-assisted text. Separately, our proposed method uses an LLM to generate predicate programs from task instructions. This component is described in the paper and appendix. We take responsibility for the final content of this work.

REFERENCES

Constructions Aeronautiques, Adele Howe, Craig Knoblock, ISI Drew McDermott, Ashwin Ram, Manuela Veloso, Daniel Weld, David Wilkins Sri, Anthony Barrett, Dave Christianson, et al. Pddl—the planning domain definition language. *Technical Report, Tech. Rep.*, 1998.

Michael Ahn, Anthony Brohan, Noah Brown, Yevgen Chebotar, Omar Cortes, Byron David, Chelsea Finn, Chuyuan Fu, Keerthana Gopalakrishnan, Karol Hausman, et al. Do as i can, not as i say: Grounding language in robotic affordances. *arXiv preprint arXiv:2204.01691*, 2022.

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

Pierre-Luc Bacon, Jean Harb, and Doina Precup. The option-critic architecture. In *Proceedings of the AAAI conference on artificial intelligence*, volume 31, 2017.

Maxime Chevalier-Boisvert, Dzmitry Bahdanau, Salem Lahlou, Lucas Willems, Chitwan Saharia, Thien Huu Nguyen, and Yoshua Bengio. BabyAI: First steps towards grounded language learning with a human in the loop. In *International Conference on Learning Representations*, 2019.

Rohan Chitnis, Tom Silver, Joshua B Tenenbaum, Tomas Lozano-Perez, and Leslie Pack Kaelbling. 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.

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

Benjamin Eysenbach, Abhishek Gupta, Julian Ibarz, and Sergey Levine. Diversity is all you need: Learning skills without a reward function. *arXiv preprint arXiv:1802.06070*, 2018.

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

Caelan Reed Garrett, Tomás Lozano-Pérez, and Leslie Pack Kaelbling. 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.

Caelan Reed Garrett, Rohan Chitnis, Rachel Holladay, Beomjoon Kim, Tom Silver, Leslie Pack Kaelbling, and Tomás Lozano-Pérez. Integrated task and motion planning. *Annual review of control, robotics, and autonomous systems*, 4(1):265–293, 2021.

Karol Gregor, Danilo Jimenez Rezende, and Daan Wierstra. Variational intrinsic control. *arXiv preprint arXiv:1611.07507*, 2016.

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

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.

Mikayel Samvelyan, Robert Kirk, Vitaly Kurin, Jack Parker-Holder, Minqi Jiang, Eric Hambro, Fabio Petroni, Heinrich Küttler, Edward Grefenstette, and Tim Rocktäschel. Minihack the planet: A sandbox for open-ended reinforcement learning research. *arXiv preprint arXiv:2109.13202*, 2021.

Archit Sharma, Shixiang Gu, Sergey Levine, Vikash Kumar, and Karol Hausman. Dynamics-aware unsupervised discovery of skills. *arXiv preprint arXiv:1907.01657*, 2019.

Tom Silver, Rohan Chitnis, Joshua Tenenbaum, Leslie Pack Kaelbling, and Tomás Lozano-Pérez. 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.

Tom Silver, Ashay Athalye, Joshua B Tenenbaum, Tomás Lozano-Pérez, and Leslie Pack Kaelbling. Learning neuro-symbolic skills for bilevel planning. *arXiv preprint arXiv:2206.10680*, 2022.

Tom Silver, Rohan Chitnis, Nishanth Kumar, Willie McClinton, Tomás Lozano-Pérez, Leslie Kaelbling, and Joshua B Tenenbaum. Predicate invention for bilevel planning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 37, pp. 12120–12129, 2023.

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

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models. *arXiv preprint arXiv:2305.16291*, 2023.

Qiang Yang, Kangheng Wu, and Yunfei Jiang. Learning action models from plan examples with incomplete knowledge. In *ICAPS*, pp. 241–250, 2005.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*, 2022.

A 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.

B 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 3 and 6. 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 4 and 5. 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 7. Finally, we provide case studies of gap detection in Figures 5 and 8, showing how PBD identifies the missing bridge predicate from a failed plan and uses it to guide subsequent skill learning.

B.1 Overall Skill Library Expansion

Figures 3 and 6 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.

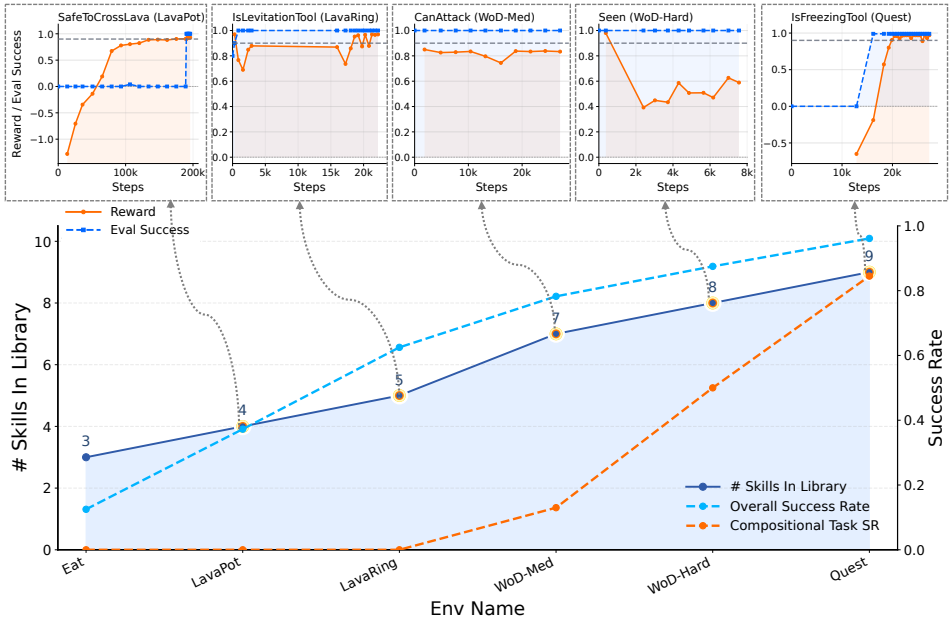

Figure 6: Skill library expansion over the Minihack curriculum.

B.2 Performance Evolution with Skill-Library Growth

Tables 4 and 5 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.

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.

Table 4: 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 5: 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

B.3 SKILL REUSE ANALYSIS

Figures 4 and 7 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 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.

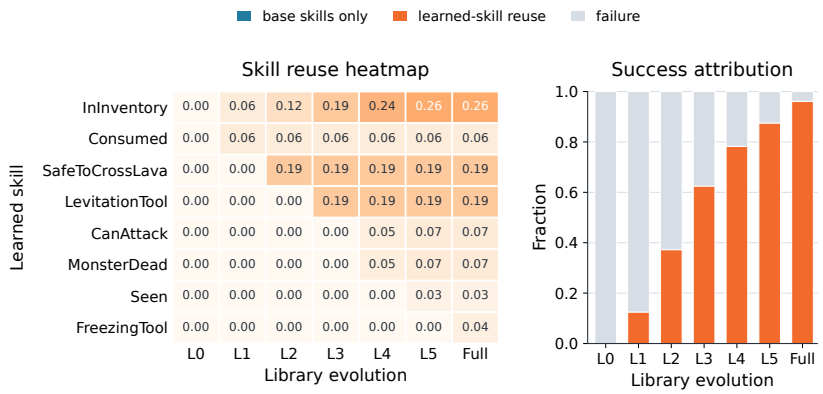

Figure 7: Skill reuse and success attribution across library growth in MiniHack. Heatmaps report the fraction of evaluation episodes in which each learned skill is invoked at least once under each library snapshot. Bar plots decompose episodes into successes using only base skills, successes using at least one learned skill, and failures. As the library grows, learned-skill reuse increases and the failure rate decreases.

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

B.4 Additional Gap Detection Example

Figures 5 and 8 provide 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.

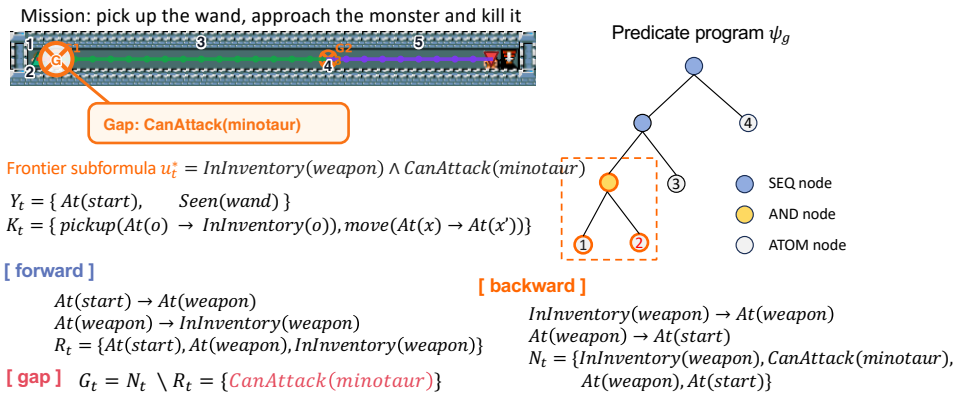

Figure 8: Case study of PBD in MiniHack-WoD-Medium.

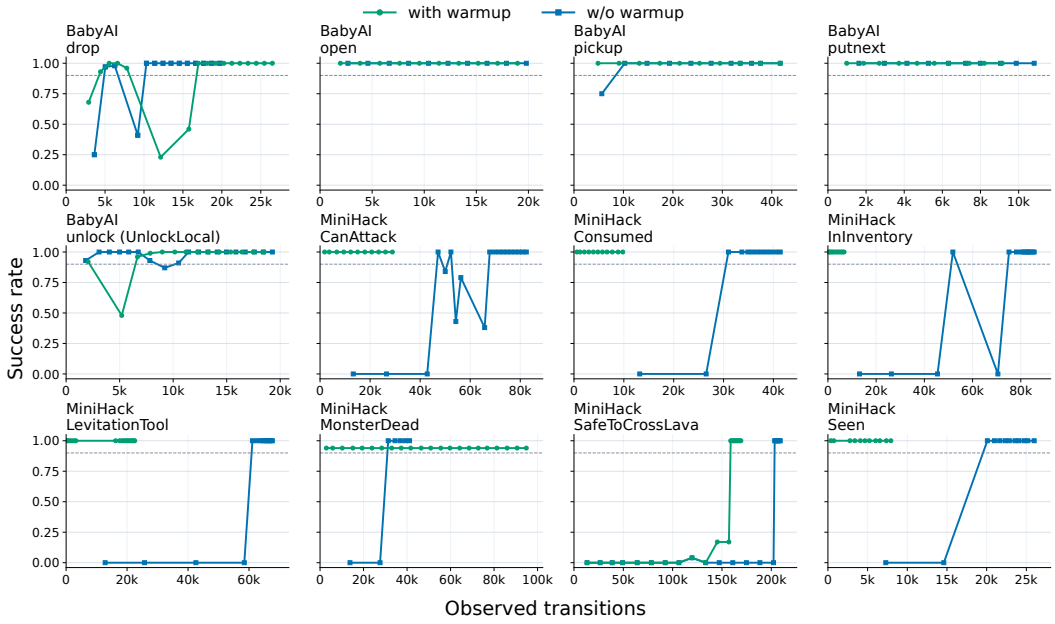

Figure 9: Effect of behavior-cloning warm-up on bridge-skill training under curriculum learning. Each panel shows the validation success rate of one learned bridge skill as a function of observed environment transitions. We compare PPO training initialized with warm-up (with warmup) against PPO from 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.

C Additional Ablation Study

C.1 Warm-up for Skill-Policy Training

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 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 9.

C.2 Skill validation 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 6 reports the effect of varying this threshold.

Table 6: 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.3 Bridge Selection

We compare PBD with a frontier-only selector. PBD selects a bridge using the difference between backward subgoal requirements and forward reachability under the current library, whereas frontier-only directly selects the unsupported active-frontier predicate. Table 7 shows that PBD selected the prerequisite InInventory(wand), yielding an accepted skill and 48/100 held-out successes. Frontier-only selection instead targeted MonsterDead(minotaur) before its prerequisite was reachable; the candidate was rejected and achieved 0/100 successes. Under this controlled setting, the results suggest that selecting an appropriate intermediate bridge is critical for formulating a learnable skill-acquisition problem.

Table 7: Bridge-target selection on MiniHack-WoD-Medium.

Selection rule	Selected bridge	New skill accepted	SR@100
PBD	`InInventory(wand)`	Yes	0.48
Frontier-only	`MonsterDead(minotaur)`	No	0.00

D Implementation and Experimental Setup Details

D.1 Curriculum and Environment Split

Figure 10 provides a visual summary of this curriculum for both BabyAI and MiniHack. Table 8 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.

D.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. The checker therefore provides the symbolic state used by PBD. In our experiments, C is implemented as an environment-level checker. 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

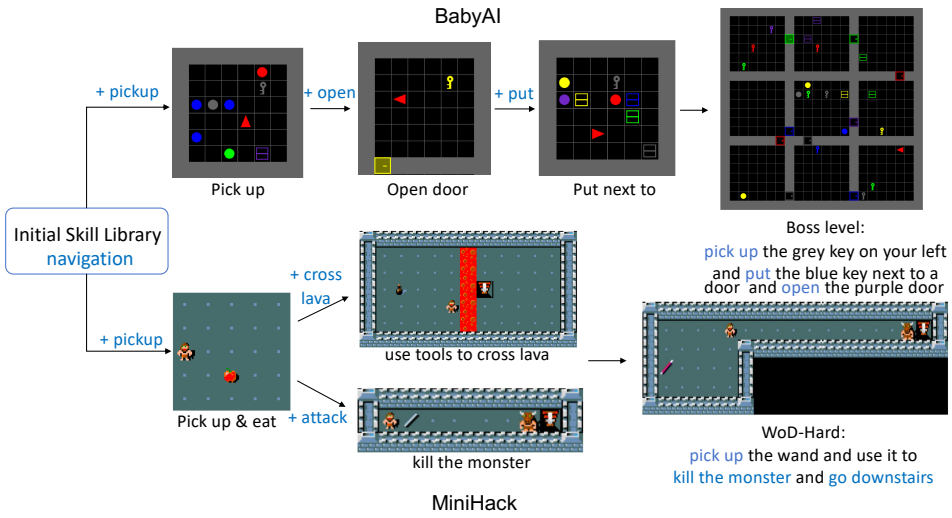

Figure 10: Illustrative curriculum for skill accumulation and compositional reuse.

Table 8: 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

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.

D.3 SENSITIVITY TO PREDICATE NOISE

We evaluate the sensitivity of PBD to errors in predicate checking on BabyAI-OpenDoor. We inject independent 5% or 10% false-positive (FP) or false-negative (FN) noise into the grounded predicate observations. We measure changes in bridge-target selection and active-frontier identification, whether the intended skill is acquired, and final success rate on 100 held-out episodes. For each condition, we train with three seeds and report mean success rate and standard deviation.

The results in Table 9 show that predicate noise perturbs bridge-target selection and active-frontier identification, but PBD still acquires the intended skill in all runs. At 5% noise, final SR remains at least 92.3%; at 10%, FP noise reduces SR to 81.0%, whereas FN noise retains 100%. Thus, the injected noise degrades some decisions and outcomes without causing complete skill-learning failure.

Table 9: Sensitivity to predicate noise on BabyAI-OpenDoor.

Noise	Bridge agreement (%) ↑	Frontier changed (%) ↓	Skill acquisition (runs) ↑	SR (%) mean ± std. ↑
0%	100.0	0.0	3/3	100.0 ± 0.0
FP-5%	94.0	6.0	3/3	92.3 ± 1.5
FN-5%	95.0	10.8	3/3	100.0 ± 0.0
FP-10%	88.8	11.2	3/3	81.0 ± 3.0
FN-10%	89.6	21.2	3/3	100.0 ± 0.0

D.4 Training Protocols for baselines

IL. BabyAI uses 10k official demonstration episodes, while MiniHack uses 40 manually collected demonstration episodes per environment.

Flat-RL. 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.

HRL. 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.

Voyager. For Voyager (Wang et al., 2023), we use BabyAI and MiniHack adaptations of its executable code generation, self-verification, and skill-memory loop. Starting from an empty learned skill library, the baseline uses GPT-5.4 to propose subgoals and generate reusable Python skill programs for interacting with the environment. Each subgoal allows up to four code-generation attempts with execution feedback; a program is added to the library only after an independent critic verifies its outcome and execution completes without error. Previously learned skills are retrieved using Qwen3-Embedding-8B. Training targets 160 action-agent prompts, with a maximum of 100 episodes and eight curriculum tasks per episode. We then freeze the learned skill library and evaluate on 100 held-out episodes per environment.

D.5 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 predicate rewards. The policy receives a success reward when the target predicate becomes true with a small step penalty. 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..

D.6 Training Hyperparameters

Table 10 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.

Table 10: 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

E Predicate-Program Planner

In all reported experiments, we use GPT-5.4 to generate predicate programs from task instructions. The model receives the instruction, the initial environment state and the allowed predicate vocabulary. It outputs a JSON tree with `SEQ` and `AND` nodes and predicate nodes. We validate each program before passing it to PBD. If a program error is detected during execution, we provide the program and failure information to the model for correction, allowing at most three attempts.

The full prompts used for BabyAI and MiniHack are shown below.

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.
```