BEAT: BOTTLENECK-GUIDED SELF-EVOLUTION OF EMBODIED AGENTS FOR LONG-HORIZON TASKS

Anonymous authors
Paper under double-blind review

ABSTRACT

Long-horizon embodied tasks require skill organization that matches controller capabilities, yet skill decompositions are often fixed in advance. We introduce BEAT, a bottleneck-guided self-evolution framework that jointly adapts the planner and controller through execution feedback. Starting from a coarse control tree, BEAT identifies bottleneck skills through closed-loop rollouts and first improves their controllers with targeted online reinforcement learning. Fresh rollouts continually refresh skill-entry states, keeping local training aligned with the evolving agent. When local learning plateaus while failures persist, the planner refines skill granularity and task organization through sequential decomposition or conditional branching. Candidate structures are evaluated against the incumbent after controller adaptation using matched closed loop rollouts. Across 30 Minecraft long-horizon tasks, BEAT achieves 58.9% average success, compared with 51.1% for the strongest evaluated baseline. Targeted controller adaptation further increases average success from 61.5% to 72.9% on a six-task subset. In Craftax, where controllers are trained from scratch, BEAT achieves 23.93% average success across four goals within an 800M-step budget, compared with 20.41% for the strongest evaluated baseline. These results support execution-driven adaptation of skill organization and controller capabilities for long-horizon tasks.

1 INTRODUCTION

Open-world environments such as Minecraft (Guss et al., 2019) or Craftax (Matthews et al., 2024) pose challenging long-horizon decision-making problems for embodied agents. Tasks in these environments require agents to coordinate sequences of interdependent behaviors—such as exploration, resource gathering, tool crafting, and recovery from unexpected events—under partial observability and substantial environmental stochasticity (Fan et al., 2022; Lin et al., 2023a). As the task horizon grows, local execution errors can accumulate, while sparse task-level feedback provides limited information about where the execution bottleneck lies. Consequently, successful long-horizon behavior depends not only on choosing an appropriate high-level procedure, but also on reliably executing each of its constituent skills.

A common approach is the hierarchical planner–controller paradigm. A high-level planner, often instantiated by a pretrained large language model (LLM) (Touvron et al., 2023; OpenAI, 2023; Yang et al., 2025; Anthropic, 2025) or vision-language model (VLM) (Achiam et al., 2023; Wang et al., 2024b; Team et al., 2023; Guo et al., 2025), converts task instructions and current environment observations into intermediate subgoals (Wang et al., 2023; 2024c;a), semantic actions (Brohan et al., 2023; Li et al., 2025; Wang et al., 2025), or other high-level control representations (Cai et al., 2024a;b; Lifshitz et al., 2024; Hafner et al., 2023), while lightweight controllers implemented by scripts or learned policies translate these abstract decisions into native environment actions. Such systems combine the reasoning ability of large models with efficient, specialized execution, reducing the need for costly large-model inference at every control step (Wang et al., 2026). Some recent approaches further employ multiple expert controllers and adaptively switch among them according to the task or current state (Wang et al., 2025; He et al., 2026). Nevertheless, in many such systems, the available skill interfaces and the functional boundaries of the corresponding controllers are specified in advance, even when skill selection and task-level planning are adaptive. Given the diversity of open-world tasks, it is difficult to determine in advance a fixed planner–controller decomposition that remains appropriate as the agent's capabilities evolve.

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.

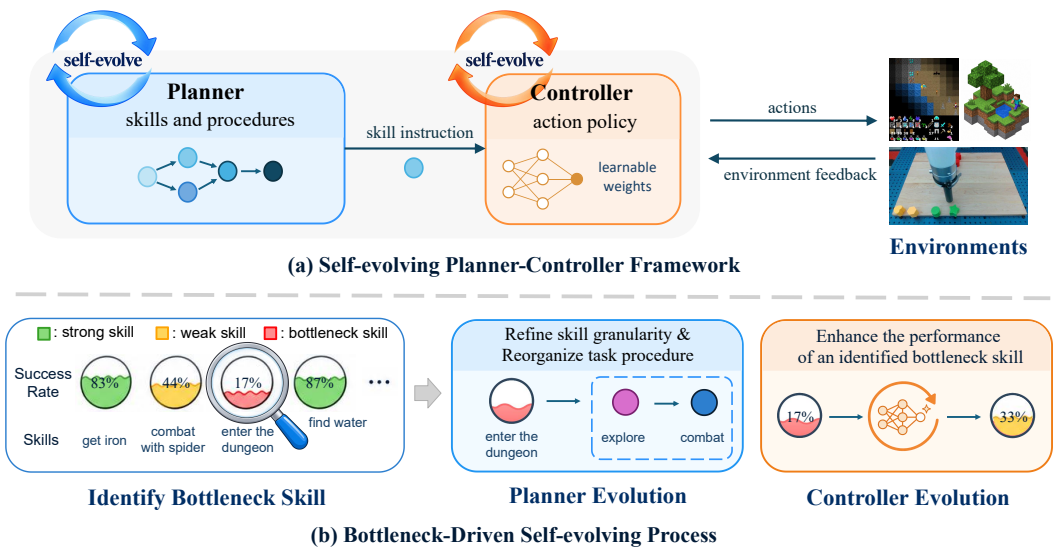


Figure 1: **Bottleneck-guided self-evolution of planner–controller agents.** (a) A hierarchical embodied agent consists of a planner that organizes skills into task procedures and a controller that executes selected skills through low-level action policies. Both components can evolve through environment feedback. (b) BEAT uses closed-loop execution to identify the current bottleneck skill and guide self-evolution at two coupled levels: *planner evolution* refines skill granularity and reorganizes task procedures, while *controller evolution* improves the execution capability of the bottleneck skill through targeted policy learning.

Meanwhile, recent self-evolving agents suggest that an agent's internal organization need not remain fixed. High-level models can create new tools, programs, subtasks, or controllers based on execution experience, reducing the need to manually specify a complete system beforehand. We use **skills** as a common term for executable behaviors implemented by scripts, lightweight policies, programs, or other control mechanisms (Xia et al., 2026; Yang et al., 2026; Zabounidis et al., 2026). Yet skill creation alone does not ensure an appropriate decomposition: generated skills can become excessively fine-grained, have unclear functional boundaries, and be costly for the planner to organize and maintain.

These limitations highlight a tension between planner organization and controller capability, particularly in the choice of skill granularity. Consider the Minecraft task of obtaining a diamond from scratch. A human-designed skill such as *chop a tree* is a semantically natural step, yet may still combine multiple behaviors that are difficult for the current controller to learn reliably. Conversely, unrestricted decomposition may fragment the same behavior into operations such as *adjust the camera*, *move toward the target*, and *slightly realign the crosshair*, effectively shifting fine-grained control decisions back to the planner. **Skills that are too coarse can be hard to learn, while skills that are too fine can make planning more difficult.**

We argue that appropriate skill granularity and organization are not determined by the task alone, but also depend on the agent's current capabilities. A stage that remains difficult to learn may benefit from decomposition, whereas a reliably executed stage need not be subdivided further. Moreover, structural deficiencies—such as missing conditional branches, overlooked resource dependencies, or absent recovery behaviors—may only become apparent through actual execution. Planner organization and controller competence should therefore evolve together: controller improvements change the states and failure patterns encountered during execution, while planner refinements redefine the learning problems assigned to individual controllers. The resulting structure should be refined progressively in response to execution bottlenecks, rather than fully specified beforehand or expanded without regard to controller capabilities.

Motivated by this observation, we propose **BEAT** (**B**ottleneck-guided **E**volution of **A**gents for long-horizon **T**asks), a framework for the self-evolution of hierarchical embodied agents. BEAT starts

from a coarse executable control tree and repeatedly evaluates the current agent closed-loop from fresh task initializations. It identifies a bottleneck limiting task completion and first performs *controller evolution*, improving the corresponding skill through targeted online reinforcement learning. Only when local learning plateaus while systematic failures persist does BEAT consider *planner evolution* to refine skill granularity and reorganize the task procedure through sequential decomposition or conditional routing. After adapting the affected policies, BEAT compares each candidate against the incumbent through matched closed-loop evaluation and either promotes the candidate or retains the incumbent. In this way, BEAT couples planner-level structural evolution with controller-level policy evolution, using execution evidence to guide when and how additional structure is introduced.

Our contributions are summarized as follows:

- We formulate long-horizon embodied-agent learning as a coupled planner–controller self-evolution problem, in which skill boundaries and task organization adapt to the capabilities of the underlying controllers rather than being fixed in advance.
- We introduce BEAT, a bottleneck-guided framework that couples controller evolution through targeted online reinforcement learning with planner evolution through bottleneck-guided refinement of skill granularity and task organization.
- We evaluate BEAT on long-horizon tasks in Minecraft and Craftax, showing improvements over the evaluated baselines in both organizing pretrained skills and learning skill policies from scratch.

2 Related Work

2.1 Hierarchical Planner–Controller Agents

Classical hierarchical reinforcement learning uses goals, temporally extended options, or manager–worker structures to provide temporal abstraction and ease long-horizon credit assignment (Vezhnevets et al., 2017; Bacon et al., 2017; Nachum et al., 2018; Nasiriany et al., 2019). Planning-based approaches compose executable skills or actions through task plans, often using symbolic operators with preconditions and effects (Yang et al., 2018; Cheng & Xu, 2023; Lin et al., 2023b; Yuan et al., 2023; Yao et al., 2022). These approaches support compositional execution but often retain predefined hierarchical forms, skill interfaces, or operator schemas.

LLM/VLM-based agents generate task plans or select skills using semantic relevance and environmental affordances (Huang et al., 2022a; Singh et al., 2022; Brohan et al., 2022; Wang et al., 2023), compose perception and control APIs into programs with functions, loops, and conditionals (Liang et al., 2022; Wang et al., 2024a), and replan using observations, execution feedback, and memory (Huang et al., 2022b; Wang et al., 2024c). Behavior Trees compose action and condition nodes through Sequence and Fallback/Selector nodes (Iovino et al., 2022), with recent methods using LLMs to generate or update these structures from task descriptions and execution context (Zhou et al., 2024; Zhang et al., 2025). Other methods learn intermediate action abstractions, such as semantic subtasks and latent goals, to connect high-level instructions with low-level control (Brohan et al., 2023; Belkhale et al., 2024; Physical Intelligence et al., 2025; Li et al., 2025; Cai et al., 2024a;b; Lifshitz et al., 2024), or study termination, memory, observation representations, and switching across action levels (Hu et al., 2026; Wang et al., 2025; He et al., 2026; Wang et al., 2026). Although planning and skill selection can be adaptive, controller interfaces and functional boundaries often remain determined by the system design. Flexible task-level composition therefore does not by itself ensure that skill boundaries match the current controllers' capabilities. BEAT instead revises skill granularity and executable organization in response to closed-loop execution bottlenecks.

2.2 Skill Discovery and Evolution

Skill-discovery methods optimize behavioral diversity or state predictability (Eysenbach et al., 2019; Sharma et al., 2020), construct skill graphs from initiation sets, termination regions, and composability relations (Bagaria et al., 2021), or induce task hierarchies from trajectories and demonstrations (Huang et al., 2019; Harvey et al., 2026; Deng et al., 2025). These methods reduce dependence

on manually specified skill boundaries, but their abstractions are generally driven by state-space coverage or trajectory structure rather than the current agent's closed-loop bottlenecks.

LLM-based agents expand persistent skill libraries through task proposal, program generation, environment-grounded validation, and retrieval (Wang et al., 2024a; Zhao et al., 2024; Xia et al., 2026; Yang et al., 2026), or compile successful multi-step behaviors into reusable higher-level skills (Zhang et al., 2023; Carta et al., 2026). These approaches support continual capability acquisition without a fully predefined library. BEAT focuses on the organization of the current task: it may retain an existing skill, decompose successive behaviors, or introduce conditional routing for distinct entry contexts while preserving the surrounding procedure.

Training and execution feedback can also guide revisions to textual guidance, rewards, and success conditions (Xia et al., 2026; Ma et al., 2024; Xie et al., 2024; Yu et al., 2023; Zhao et al., 2024). More closely related methods revise symbolic skill operators or hierarchical reward programs, including their prerequisites, effects, resource conditions, and reward specifications (Zabounidis et al., 2026; Bornemann et al., 2026). These works already adapt skill specifications through experience. BEAT emphasizes how structural revision is coordinated with controller learning: it first trains the current bottleneck and considers structural changes when local learning plateaus while failures persist. After adapting the affected controllers, BEAT compares each candidate with the incumbent through matched fresh-start evaluation, grounding structural selection in complete-task performance rather than local skill success alone.

Local policy learning also depends on the states from which skills are trained. Reset-based methods expand start-state distributions backward from goals, revisit archived states, or use guide policies to provide handoff states (Florensa et al., 2017; Ecoffet et al., 2021; Uchendu et al., 2023). BEAT restores recent bottleneck-entry states and refreshes them through closed-loop rollouts, avoiding repeated execution of reliable prefixes while tracking changes in upstream policies and routing.

3 METHOD

BEAT jointly adapts skill policies and their executable organization through execution feedback. It first trains bottleneck controllers and considers structural refinement when local learning plateaus while failures persist. Candidate structures are compared with the incumbent after controller adaptation using matched closed-loop evaluation.

3.1 PLANNER: BOTTLENECK-GUIDED SKILL ORGANIZATION

Following the problem formulation in Appendix A, we represent the agent by its skill policies Π and executable organization $\mathcal{G}$. In our implementation, $\mathcal{G}$ is a control tree $\mathcal{T}$: leaves correspond to executable skill stages, and internal nodes specify sequential or conditional composition. The tree determines which controller acts and when control is transferred, using the available observations and execution history.

BEAT begins with a coarse tree and refines it locally as execution bottlenecks emerge. Fresh closed-loop rollouts execute the current tree and policies from task initial states, recording stage-entry states, completion outcomes, and failure trajectories. The planner uses this evidence together with accumulated training history to identify a bottleneck stage b.

A bottleneck does not by itself imply that the structure should change. Its controller may simply require further training. Alternatively, the stage may combine distinct subprocesses or entry contexts that are difficult for a single controller to handle. BEAT therefore first optimizes the bottleneck controller while keeping the tree fixed. Only when local improvement stalls does the planner consider changing the stage's decomposition or routing. This ordering makes structural refinement responsive to persistent execution failures after an opportunity for policy improvement.

3.2 CONTROLLERS: BOTTLENECK-FOCUSED REINFORCEMENT LEARNING

Training exclusively from task initial states repeatedly executes prefixes that may already be reliable. When the main difficulty lies downstream, these repeated prefixes consume interaction without

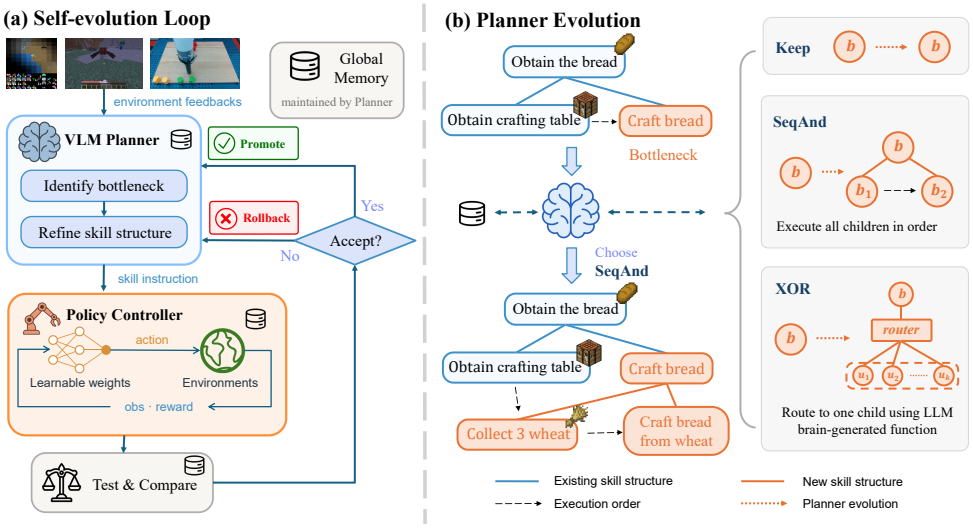


Figure 2: **Overview of BEAT.** Fresh closed-loop rollouts identify execution bottlenecks and refresh skill-entry states. Bottleneck controllers are improved through local reinforcement learning. When learning plateaus, the planner considers refinements to skill decomposition or routing. Candidate structures undergo controller adaptation before matched closed-loop evaluation determines whether they are retained.

directly training the bottleneck. BEAT therefore combines a bottleneck-local training stream with a smaller fresh closed-loop rollout stream.

Bottleneck-local training. For each stage v, we maintain a buffer $\mathcal{D}_v$ of recently encountered environment states at which the complete agent invokes v. Given bottleneck b, local rollouts restore states sampled from $\mathcal{D}_b$ and primarily update the associated policy π_b on its local objective. This concentrates learning on the current bottleneck while using entry states reached by the complete agent.

Fresh closed-loop rollouts. As upstream controllers and routing rules change, cached entry states may no longer reflect the current agent's execution. BEAT therefore periodically runs the current tree and policies from fresh task initializations. Whenever a stage is invoked, its entry state is added to the corresponding buffer. These rollouts refresh the local training distributions and provide task-level outcomes and failure evidence for bottleneck identification and structural refinement.

In our implementation, approximately 90% of the rollout budget is allocated to bottleneck-local training and 10% to fresh closed-loop rollouts. The local stream focuses policy updates on the current learning problem, while the fresh stream maintains coverage of the states encountered by the evolving complete agent.

3.3 Control-Tree Refinement and Selection

When local reinforcement learning plateaus at bottleneck b, the planner receives the current tree $\mathcal{T}$, bottleneck evidence $\mathcal{E}_b$, and memory $\mathcal{M}$ of previous training and structural attempts. It proposes

$$(z, \mathcal{T}') = \text{Planner}(\mathcal{T}, b, \mathcal{E}_b, \mathcal{M}), \qquad z \in \{\textsc{Keep}, \textsc{SeqAnd}, \text{XOR}\}. \tag{1}$$

The planner can retain the current tree or apply one of two local refinements to the bottleneck stage.

Sequential decomposition (SeqAnd). When trajectories reveal subgoals with a consistent temporal ordering, a stage can be replaced by

$$v \longrightarrow \text{SeqAnd}(v_1, \ldots, v_\ell). \tag{2}$$

The children execute in order, and completion of the sequence completes the parent stage. This refinement makes intermediate objectives explicit and divides a long behavior into smaller controller-learning problems.

Algorithm 1 BEAT: Bottleneck-Guided Planner–Controller Evolution

Require: Control tree $\mathcal{T}$; training state Θ; memory $\mathcal{M}$; patience schedule $p(\cdot)$
while training budget remains **do**
  $\mathcal{E} \leftarrow \mathrm{FreshRollout}(\mathcal{T}, \Theta)$
  Refresh entry-state buffers in Θ using $\mathcal{E}$
  $b \leftarrow \mathrm{Planner.Bottleneck}(\mathcal{T}, \mathcal{E}, \mathcal{M})$
  $\Theta \leftarrow \mathrm{LocalRL}(b, \mathcal{T}, \Theta)$
  if $\mathrm{Plateau}(b, p(d_b))$ **then**
    $(z, \mathcal{T}') \leftarrow \mathrm{Planner.Propose}(\mathcal{T}, b, \mathcal{E}_b, \mathcal{M})$
    if $z \neq$ KEEP and $\mathrm{Valid}(\mathcal{T}', \mathcal{E})$ **then**
      $\Theta' \leftarrow \mathrm{Adapt}(\mathcal{T}', \mathrm{Copy}(\Theta), \mathcal{E})$
      $(\widehat{J}, \widehat{J}') \leftarrow \mathrm{MatchedEval}(\mathcal{T}, \Theta, \mathcal{T}', \Theta')$
      if $\mathrm{Accept}(\widehat{J}, \widehat{J}')$ **then**
        $(\mathcal{T}, \Theta) \leftarrow (\mathcal{T}', \Theta')$
      end if
      Update $\mathcal{M}$ with the proposal and evaluation outcome
    end if
  end if
end while

Conditional decomposition (XOR). When different entry contexts require distinct strategies, a stage can instead be replaced by

$$v \longrightarrow \mathrm{XOR}_{f_v}(v_1, \ldots, v_\ell), \qquad i = f_v(h_{t_j}), \tag{3}$$

where h_{t_j} denotes the available execution history at stage entry and f_v is an executable routing function. The selected child v_i executes, and its completion completes the parent stage. Each branch must be supported by a minimum number of observed entry states to avoid introducing specialized branches in response to isolated failures.

Controller adaptation and candidate selection. A structural refinement changes the learning problems assigned to the affected controllers. BEAT therefore adapts new or affected policies on a copy of the incumbent training state before assessing the candidate. The incumbent and adapted candidate are then evaluated from fresh task initializations under matched conditions, using the same task-level objective. Their measured closed-loop performance determines whether the candidate is promoted or the incumbent is retained. The proposal and evaluation outcome are recorded in $\mathcal{M}$ to inform subsequent refinement attempts.

Algorithm 1 summarizes the procedure. Here, Θ denotes the training state, including the skill policies and entry-state buffers, and $p(d_b)$ specifies the plateau patience for a bottleneck at tree depth d_b. Detailed patience schedules, proposal-validity checks, branch-support thresholds, acceptance criteria, and memory bookkeeping are provided in Appendix C.

4 EXPERIMENTS

We evaluate BEAT in two complementary settings. Minecraft examines how adaptive skill organization improves long-horizon task completion with a shared pretrained controller. Craftax examines whether the framework can learn long-horizon skills from scratch. We then study adaptive organization relative to fixed task specifications, compare alternative representations for organizing skills, and assess the additional benefit of targeted controller adaptation. Additional experiments appear in Appendix D; full protocols and hyperparameters are provided in Appendix E.

4.1 MINECRAFT: ORGANIZING PRE-TRAINED SKILLS

Setup. We construct a 30-task acquisition suite in MineStudio, with ten tasks in each of three difficulty tiers: Simple, Medium, and Hard. Episodes vary in available ingredients, completed prerequisites, and nearby resources, requiring agents to organize available skills toward a target item. All methods use the same STEVE-1 controller, shared GUI skills, task initializations, and interaction budgets. The main comparison keeps the pretrained controller fixed to evaluate high-level planning and skill organization; we study controller adaptation separately in Section 4.3.

Table 1: Main results on the Minecraft task suite. We report success rates (%) for two representative tasks from each difficulty tier, together with the macro-average over all ten tasks in that tier and the overall 30-task macro-average. All methods use the same frozen STEVE-1 controller, GUI skills, task initializations, and interaction budgets, and are evaluated over 32 episodes per task. Superscripts denote standard deviations across tasks. Bold and underlined values denote the best and second-best mean results, respectively.

Method	Simple			Medium			Hard			Overall Avg.
	Bucket	Iron Nugget	*Avg.*	Shield	Tripwire Hook	*Avg.*	Jukebox	Crossbow	*Avg.*	
Baselines										
GPT-5.6 (OpenAI, 2026)	9.4	0.0	$17.5^{\pm 19.5}$	0.0	0.0	$8.1^{\pm 10.3}$	0.0	0.0	$2.5^{\pm 5.0}$	$9.4^{\pm 14.5}$
Reflexion (Shinn et al., 2023)	81.2	25.0	$81.9^{\pm 22.5}$	40.6	**50.0**	$37.5^{\pm 20.3}$	0.0	18.8	$15.0^{\pm 14.2}$	$44.8^{\pm 33.8}$
Plan4MC (Yuan et al., 2023)	78.1	**93.8**	$85.6^{\pm 10.5}$	34.4	28.1	$38.1^{\pm 17.4}$	<u>18.8</u>	25.0	<u>25.0</u>$^{\pm 10.6}$	$49.6^{\pm 29.2}$
DEPS (Wang et al., 2023)	<u>87.5</u>	62.5	$80.0^{\pm 12.7}$	31.3	31.3	$39.4^{\pm 23.6}$	0.0	<u>31.3</u>	$13.4^{\pm 14.0}$	$44.3^{\pm 32.5}$
Optimus-1 (Li et al., 2024)	50.0	37.5	$75.0^{\pm 20.9}$	<u>56.3</u>	25.0	$38.1^{\pm 19.0}$	12.5	6.3	$18.1^{\pm 19.9}$	$43.8^{\pm 30.9}$
JARVIS (Wang et al., 2024c)	84.4	68.8	$85.9^{\pm 13.1}$	18.8	0.0	$37.2^{\pm 20.1}$	<u>18.8</u>	18.8	$11.9^{\pm 9.9}$	$45.0^{\pm 34.2}$
XENON (Lee et al., 2026)	75.0	68.8	<u>89.4</u>$^{\pm 13.4}$	<u>56.3</u>	18.8	<u>44.1</u>$^{\pm 20.6}$	6.3	12.5	$20.0^{\pm 16.7}$	<u>51.1</u>$^{\pm 33.5}$
Ours										
BEAT	**90.6**	<u>90.6</u>	$\mathbf{94.1}^{\pm 5.3}$	**68.8**	<u>37.5</u>	$\mathbf{52.8}^{\pm 18.4}$	**43.8**	**37.5**	$\mathbf{29.7}^{\pm 14.7}$	$\mathbf{58.9}^{\pm 30.1}$

We compare BEAT with adaptations of JARVIS-1, XENON, DEPS, Reflection, Plan4MC and Optimus-1, as well as a GPT-5.6 baseline. All methods are adapted to the common MineStudio environment and controller interface, with GPT-5.6 at xhigh reasoning effort for high-level planning. We evaluate each method with 32 episodes per task. Table 1 reports two representative tasks per tier, macro-averages over all ten tasks in each tier, and the overall 30-task macro-average.

Results. BEAT achieves the highest macro-average in every difficulty tier: 94.1% on Simple, 52.8% on Medium, and 29.7% on Hard. Its overall success rate is 58.9%, compared with 51.1% for the strongest baseline, XENON. These results show that adaptive skill organization improves goal completion across the three difficulty tiers under a shared, fixed controller.

4.2 Craftax: Learning Skills from Scratch

Setup. We evaluate four long-horizon goals in Craftax (Matthews et al., 2024): CollectDiamond, EnterDungeon, UnlockGnomishLadder, and EnterGnomishMines. Low-level policies are trained from random initialization within an 800M-step interaction budget per run. We compare BEAT with general RL baselines (PPO, PQN, GRPO, and SAC) and Craftax adaptations of structured or skill-based methods (ARM-FM, Voyager, and SkillRL). For each method–task pair, we select the checkpoint with the highest observed batch success rate within the interaction budget, then independently evaluate it over 256 episodes from fresh task initializations. Table 2 reports these evaluation success rates and their unweighted average across the four tasks.

Results. BEAT achieves the highest four-task average (23.93% versus 20.41% for ARM-FM) and leads on three goals. Its largest absolute advantage is on CollectDiamond (24.22% versus 13.67% for ARM-FM), while EnterDungeon performance is close to SkillRL (64.45% versus 64.84%).

Figure 3 complements the task-level results with learning trajectories over the 800M-step budget. BEAT establishes a clear advantage on CollectDiamond, while several structured baselines reach similar performance on EnterDungeon. BEAT also shows progress on the Gnomish objectives later in training.

4.3 Ablation Studies

We examine how initial task specifications, the representation used for skill organization, and targeted controller adaptation affect performance. The Craftax analysis compares adaptive organization with fixed priors, while the Minecraft analyses use the six representative tasks from Table 1.

Table 2: **Craftax results (%)** within 800M environment steps. Entries report an independent 256-episode fresh-start evaluation of each checkpoint selected by batch success rate within the interaction budget. Avg. denotes the unweighted mean over the four tasks. Bold and underlined values denote the best and second-best results, respectively.

Method	Collect Diamond	Enter Dungeon	Unlock Gnomish	Enter Mines	Avg.
General RL Baselines					
PPO (Schulman et al., 2017)	5.86	42.97	1.56	0.00	12.60
PQN (Gallici et al., 2025)	8.20	58.59	0.78	0.00	16.89
SAC (Haarnoja et al., 2018)	0.00	0.00	0.00	0.00	0.00
GRPO (Shao et al., 2024)	0.00	0.39	0.00	0.00	0.10
Structured / Skill-Based Baselines					
Voyager (Wang et al., 2024a)	3.91	62.89	2.34	0.39	17.38
ARM-FM (Castanyer et al., 2026)	<u>13.67</u>	62.50	<u>4.69</u>	<u>0.78</u>	<u>20.41</u>
SkillRL (Xia et al., 2026)	8.59	**64.84**	1.56	0.00	18.75
Ours					
BEAT	**24.22**	<u>64.45</u>	**5.86**	**1.17**	**23.93**

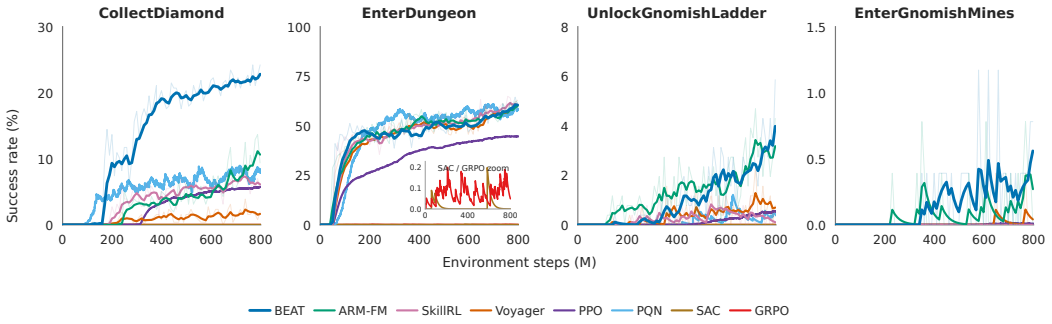


Figure 3: **Learning curves on Craftax.** BEAT is compared with structured and general RL baselines across four goals within 800M environment steps. Curves are smoothed with a 20M-step causal EWMA.

Adaptive organization and fixed priors. We compare BEAT with PPO and two variants that use fixed stage decompositions. *Weak-prior PPO* uses manually designed stages, while *Strong-prior PPO* follows Scalar (Zabounidis et al., 2026), combining rich prior world knowledge with a VLM to generate a detailed stage decomposition. Both decompositions remain fixed during training. BEAT instead starts from a single broad leaf and progressively constructs and revises its skill organization using rollout feedback.

Figure 4 shows the resulting learning trajectories within 800M environment steps. BEAT begins making progress on CollectDiamond substantially earlier than the strong-prior variant, while both attain comparable performance on EnterDungeon. The difference is more pronounced on the deeper objectives: BEAT attains higher success on UnlockGnomishLadder and maintains greater success on EnterGnomishMines throughout the later stages of training. Notably, despite starting from a single coarse skill rather than a detailed, task-specific stage decomposition, BEAT matches or surpasses the strong-prior variant across these objectives. These results demonstrate that learning the task hierarchy from interaction can recover and even improve upon the benefits of a strong predefined decomposition, particularly for harder or deeper goals.

Skill-organization representation. We compare four ways of organizing the shared pretrained skills: *Point* directly proposes a skill from the goal; *Chain* uses a linear SeqAND decomposition; *Code* expresses state-dependent control as a Python program; and *BEAT* uses SeqAND/XOR structures. Chain, Code, and BEAT receive the same rollout feedback and revision budget. The controller is fixed for this comparison, and each variant is evaluated with 32 episodes per task. As shown in

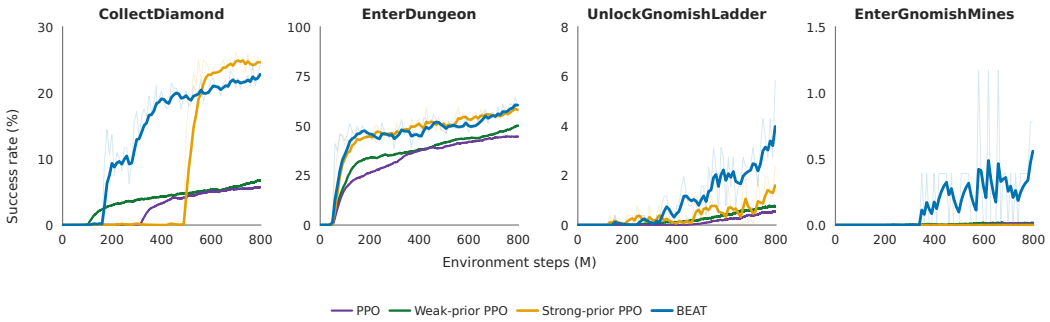


Figure 4: **Adaptive organization versus fixed priors in Craftax.** BEAT starts from a single broad leaf, whereas weak-prior and strong-prior PPO use fixed stage decompositions designed manually or generated with a VLM following Scalar (Zabounidis et al., 2026), respectively. All methods use an 800M-step budget. Curves are smoothed with a 20M-step causal EWMA.

Table 3: Ablation of high-level skill organization and low-level policy adaptation on six representative Minecraft tasks. Avg. is the unweighted mean across tasks. Notation and highlighting follow Table 1.

Variant	Bucket	Iron Nugget	Shield	Tripwire Hook	Jukebox	Crossbow	Avg.
High-level organization							
Point (GPT-5.6)	9.4	0.0	0.0	0.0	0.0	0.0	$1.6^{\pm 3.5}$
Chain (SeqAND only)	87.5	78.1	62.5	31.3	18.8	43.8	$53.6^{\pm 24.6}$
Code (Python)	81.3	**96.9**	56.3	25.0	37.5	37.5	$55.7^{\pm 25.7}$
BEAT w/o RL	**90.6**	90.6	68.8	37.5	43.8	37.5	$61.5^{\pm 23.2}$
+ Low-level policy adaptation							
BEAT w/ RL	87.5	90.6	**84.4**	**59.4**	**65.6**	**50.0**	$\mathbf{72.9}^{\pm 15.4}$

Table 3, Among the high-level organization variants, BEAT without RL achieves the highest six-task average, 61.5%, compared with 55.7% for Code and 53.6% for Chain. BEAT without RL leads on four of the six tasks among these organization variants. The results support SeqAND/XOR structures as an effective representation for reorganizing skills from execution feedback.

Targeted controller adaptation. We further enable reinforcement learning for bottleneck skills while keeping the high-level BEAT organization unchanged. All variants start from the same STEVE-1 controller and use the same evaluation protocol. Targeted adaptation raises the six-task average from 61.5% to 72.9%. The gains are concentrated on the Medium and Hard tasks. For example, success increases from 37.5% to 59.4% on Tripwire Hook and from 43.8% to 65.6% on Jukebox, with improvements also observed on Shield and Crossbow. These results show that adapting the controller provides complementary improvements beyond organizing the pretrained skills alone.

5 Conclusion

We presented BEAT, a bottleneck-guided framework that couples targeted controller learning with bottleneck-guided refinement of skill organization. BEAT first trains bottleneck controllers and considers structural changes when local learning plateaus, comparing candidate structures with the incumbent through matched closed-loop evaluation after controller adaptation. Across 30 Minecraft tasks under a shared frozen controller, BEAT achieves an average success rate of 58.9%, compared with 51.1% for the strongest evaluated baseline. In Craftax, where controllers are trained from scratch, BEAT achieves the highest average success rate among the evaluated methods and matches or surpasses a strong fixed-decomposition baseline despite starting from a single coarse skill. These results support joint planner-controller self-evolution for long-horizon task completion, with skill boundaries and task organization progressively adapted to execution bottlenecks.

AI USE STATEMENT

(This section is **required** and does not count toward the page limit.)

In this work, we used GPT-5.6 and GPT-6 to assist with translation. Additionally, we used these tools for language polishing and general code editing. We did not use generative AI tools for other research tasks requiring disclosure. The authors reviewed all AI-assisted work, checking the edited text for accuracy, translations for fidelity to the intended meaning, and code edits for consistency with the intended functionality. We take responsibility for the final content of this work, including text, claims, and artifacts produced with the aid of generative AI.

ETHICS STATEMENT

This work studies the self-evolution of embodied agents in Minecraft, Full Craftax, and the simulated Language Table environment. The experiments are conducted in simulation and do not involve human-subject studies or real-world robotic deployment. Although the proposed framework is intended to improve long-horizon planning and control, performance in these environments does not establish safety or reliability in real-world settings. In particular, automatically revised skill structures and learned controllers may exhibit unintended behaviors outside the evaluated conditions. Any deployment beyond simulation would require additional safety evaluation and appropriate human oversight.

REPRODUCIBILITY STATEMENT

Section 3 describes the BEAT framework, with the problem formulation and additional method details provided in Appendix A and Appendix C. Appendix B specifies the environments and agent interfaces. Appendix D and section 4 present the main and additional experiments, while Appendix E documents experimental configurations, comparison methods, training budgets, optimization hyperparameters, and evaluation protocols, including checkpoint selection and result aggregation. Appendix F provides concrete case studies of local control-tree refinement. Appendix G lists the Minecraft task suite, including task objectives and episode budgets. We will release our code, implementation details and experimental scripts publicly to facilitate reproducibility.

REFERENCES

Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

Anthropic. Introducing claude 4, 2025. URL https://www.anthropic.com/news/claude-4.

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

Akhil Bagaria, Jason K. Senthil, and George Konidaris. Skill discovery for exploration and planning using deep skill graphs. In *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pp. 521–531. PMLR, 2021. URL https://proceedings.mlr.press/v139/bagaria21a.html.

Bowen Baker, Ilge Akkaya, Peter Zhokov, Joost Huizinga, Jie Tang, Adrien Ecoffet, Brandon Houghton, Raul Sampedro, and Jeff Clune. Video pretraining (vpt): Learning to act by watching unlabeled online videos. *Advances in Neural Information Processing Systems*, 35:24639–24654, 2022.

Suneel Belkhale, Tianli Ding, Ted Xiao, Pierre Sermanet, Quon Vuong, Jonathan Tompson, Yevgen Chebotar, Debidatta Dwibedi, and Dorsa Sadigh. Rt-h: Action hierarchies using language. *arXiv preprint arXiv:2403.01823*, 2024.

Richard Bornemann, Pierluigi Vito Amadori, and Antoine Cully. Code-sharp: Continuous open-ended discovery and evolution of skills as hierarchical reward programs, 2026. URL https://arxiv.org/abs/2602.10085.

Anthony Brohan, Yevgen Chebotar, Chelsea Finn, Karol Hausman, Alexander Herzog, Daniel Ho, Julian Ibarz, Alex Irpan, Eric Jang, Ryan Julian, et al. Do as i can, not as i say: Grounding language in robotic affordances. In *6th Annual Conference on Robot Learning*, 2022.

Anthony Brohan, Noah Brown, Justice Carbajal, Yevgen Chebotar, Xi Chen, Krzysztof Choromanski, Tianli Ding, Danny Driess, Avinava Dubey, Chelsea Finn, et al. Rt-2: Vision-language-action models transfer web knowledge to robotic control. *arXiv preprint arXiv:2307.15818*, 2023.

Shaofei Cai, Zihao Wang, Kewei Lian, Zhancun Mu, Xiaojian Ma, Anji Liu, and Yitao Liang. Rocket-1: Mastering open-world interaction with visual-temporal context prompting. *arXiv preprint arXiv:2410.17856*, 2024a.

Shaofei Cai, Bowei Zhang, Zihao Wang, Xiaojian Ma, Anji Liu, and Yitao Liang. Groot: Learning to follow instructions by watching gameplay videos. In *The Twelfth International Conference on Learning Representations*, 2024b.

Thomas Carta, Clément Romac, Loris Gaven, Pierre-Yves Oudeyer, Olivier Sigaud, and Sylvain Lamprier. Herakles: Hierarchical skill compilation for open-ended llm agents, 2026. URL https://arxiv.org/abs/2508.14751.

Roger Creus Castanyer, Faisal Mohamed, Pablo Samuel Castro, Cyrus Neary, and Glen Berseth. Arm-fm: Automated reward machines via foundation models for compositional reinforcement learning, 2026. URL https://arxiv.org/abs/2510.14176.

Shuo Cheng and Danfei Xu. League: Guided skill learning and abstraction for long-horizon manipulation, 2023. URL https://arxiv.org/abs/2210.12631.

Jingwen Deng, Zihao Wang, Shaofei Cai, Anji Liu, and Yitao Liang. Open-world skill discovery from unsegmented demonstrations. *arXiv preprint arXiv:2503.10684*, 2025.

Adrien Ecoffet, Joost Huizinga, Joel Lehman, Kenneth O. Stanley, and Jeff Clune. First return, then explore. *Nature*, 590(7847):580–586, 2021. doi: 10.1038/s41586-020-03157-9. URL https://doi.org/10.1038/s41586-020-03157-9.

Benjamin Eysenbach, Abhishek Gupta, Julian Ibarz, and Sergey Levine. Diversity is all you need: Learning skills without a reward function. In *International Conference on Learning Representations*, 2019. URL https://arxiv.org/abs/1802.06070.

Linxi Fan, Guanzhi Wang, Yunfan Jiang, Ajay Mandlekar, Yuncong Yang, Haoyi Zhu, Andrew Tang, De-An Huang, Yuke Zhu, and Anima Anandkumar. Minedojo: Building open-ended embodied agents with internet-scale knowledge. *Advances in Neural Information Processing Systems Datasets and Benchmarks*, 2022.

Carlos Florensa, David Held, Markus Wulfmeier, Michael Zhang, and Pieter Abbeel. Reverse curriculum generation for reinforcement learning. In *Proceedings of the 1st Annual Conference on Robot Learning*, volume 78 of *Proceedings of Machine Learning Research*, pp. 482–495. PMLR, 2017. URL https://proceedings.mlr.press/v78/florensa17a.html.

Matteo Gallici, Mattie Fellows, Benjamin Ellis, Bartomeu Pou, Ivan Masmitja, Jakob Nicolaus Foerster, and Mario Martin. Simplifying deep temporal difference learning, 2025. URL https://arxiv.org/abs/2407.04811.

Dong Guo, Faming Wu, Feida Zhu, Fuxing Leng, Guang Shi, Haobin Chen, Haoqi Fan, Jian Wang, Jianyu Jiang, Jiawei Wang, et al. Seed1. 5-vl technical report. *arXiv preprint arXiv:2505.07062*, 2025.

William H Guss, Brandon Houghton, Nicholay Topin, Phillip Wang, Cayden Codel, Manuela Veloso, and Ruslan Salakhutdinov. Minerl: A large-scale dataset of minecraft demonstrations. *arXiv preprint arXiv:1907.13440*, 2019.

Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor, 2018. URL https://arxiv.org/abs/1801.01290.

Danijar Hafner, Jurgis Pasukonis, Jimmy Ba, and Timothy Lillicrap. Mastering diverse domains through world models. *arXiv preprint arXiv:2301.04104*, 2023.

Damion Harvey, Geraud Nangue Tasse, Benjamin Rosman, Branden Ingram, and Steven James. Unsupervised hierarchical skill discovery, 2026. URL https://arxiv.org/abs/2601.23156.

Kaichen He, Zihao Wang, Muyao Li, Anji Liu, and Yitao Liang. Training one model to master cross-level agentic actions via reinforcement learning, 2026. URL https://arxiv.org/abs/2512.09706.

Jiaheng Hu, Mohit Shridhar, Caden Lu, Dhruv Shah, Hao-Tien Lewis Chiang, Jie Tan, and Annie Xie. What matters in orchestrating robot policies: A systematic study of hierarchical vla agents, 2026. URL https://arxiv.org/abs/2606.10267.

De-An Huang, Suraj Nair, Danfei Xu, Yuke Zhu, Animesh Garg, Li Fei-Fei, Silvio Savarese, and Juan Carlos Niebles. Neural task graphs: Generalizing to unseen tasks from a single video demonstration, 2019. URL https://arxiv.org/abs/1807.03480.

Wenlong Huang, Pieter Abbeel, Deepak Pathak, and Igor Mordatch. Language models as zero-shot planners: Extracting actionable knowledge for embodied agents. *ICML*, 2022a.

Wenlong Huang, Fei Xia, Ted Xiao, Harris Chan, Jacky Liang, Pete Florence, Andy Zeng, Jonathan Tompson, Igor Mordatch, Yevgen Chebotar, et al. Inner monologue: Embodied reasoning through planning with language models. *arXiv preprint arXiv:2207.05608*, 2022b.

Matteo Iovino, Edvards Scukins, Jonathan Styrud, Petter Ögren, and Christian Smith. A survey of behavior trees in robotics and ai. *Robotics and Autonomous Systems*, 154:104096, 2022. doi: 10.1016/j.robot.2022.104096. URL https://doi.org/10.1016/j.robot.2022.104096.

Seungjoon Lee, Suhwan Kim, Minhyeon Oh, Youngsik Yoon, and Jungseul Ok. Experience-based knowledge correction for robust planning in minecraft. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=N22lDHYrXe.

Muyao Li, Zihao Wang, Kaichen He, Xiaojian Ma, and Yitao Liang. Jarvis-vla: Post-training large-scale vision language models to play visual games with keyboards and mouse. *arXiv preprint arXiv:2503.16365*, 2025.

Zaijing Li, Yuquan Xie, Rui Shao, Gongwei Chen, Dongmei Jiang, and Liqiang Nie. Optimus-1: Hybrid multimodal memory empowered agents excel in long-horizon tasks, 2024. URL https://arxiv.org/abs/2408.03615.

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. *arXiv preprint arXiv:2209.07753*, 2022.

Shalev Lifshitz, Keiran Paster, Harris Chan, Jimmy Ba, and Sheila McIlraith. Steve-1: A generative model for text-to-behavior in minecraft. *Advances in Neural Information Processing Systems*, 36, 2024.

Haowei Lin, Zihao Wang, Jianzhu Ma, and Yitao Liang. Mcu: A task-centric framework for open-ended agent evaluation in minecraft. *arXiv preprint arXiv:2310.08367*, 2023a.

Kevin Lin, Christopher Agia, Toki Migimatsu, Marco Pavone, and Jeannette Bohg. Text2motion: From natural language instructions to feasible plans. *arXiv preprint arXiv:2303.12153*, 2023b.

Corey Lynch, Ayzaan Wahid, Jonathan Tompson, Tianli Ding, James Betker, Robert Baruch, Travis Armstrong, and Pete Florence. Interactive language: Talking to robots in real time, 2022. URL https://arxiv.org/abs/2210.06407.

Yecheng Jason Ma, William Liang, Guanzhi Wang, De-An Huang, Osbert Bastani, Dinesh Jayaraman, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Eureka: Human-level reward design via coding large language models, 2024. URL https://arxiv.org/abs/2310.12931.

Michael Matthews, Michael Beukman, Benjamin Ellis, Mikayel Samvelyan, Matthew Jackson, Samuel Coward, and Jakob Foerster. Craftax: A lightning-fast benchmark for open-ended reinforcement learning, 2024. URL https://arxiv.org/abs/2402.16801.

Ofir Nachum, Shixiang Gu, Honglak Lee, and Sergey Levine. Data-efficient hierarchical reinforcement learning, 2018. URL https://arxiv.org/abs/1805.08296.

Soroush Nasiriany, Vitchyr Pong, Steven Lin, and Sergey Levine. Planning with goal-conditioned policies. *Advances in Neural Information Processing Systems*, 32, 2019.

OpenAI. Chatgpt: Optimizing language models for dialogue, 2023. URL https://openai.com/blog/chatgpt/.

OpenAI. Gpt-5.6 system card. https://deploymentsafety.openai.com/gpt-5-6, 2026. Published July 9, 2026.

Physical Intelligence, Kevin Black, Noah Brown, James Darpinian, Karan Dhabalia, Danny Driess, Adnan Esmail, Michael Equi, Chelsea Finn, Niccolo Fusai, Manuel Y. Galliker, Dibya Ghosh, Lachy Groom, Karol Hausman, Brian Ichter, Szymon Jakubczak, Tim Jones, Liyiming Ke, Devin LeBlanc, Sergey Levine, Adrian Li-Bell, Mohith Mothukuri, Suraj Nair, Karl Pertsch, Allen Z. Ren, Lucy Xiaoyang Shi, Laura Smith, Jost Tobias Springenberg, Kyle Stachowicz, James Tanner, Quan Vuong, Homer Walke, Anna Walling, Haohuan Wang, Lili Yu, and Ury Zhilinsky. $\pi_{0.5}$: A vision-language-action model with open-world generalization, 2025. URL https://arxiv.org/abs/2504.16054.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, YK Li, Yang Wu, et al. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024.

Archit Sharma, Shixiang Gu, Sergey Levine, Vikash Kumar, and Karol Hausman. Dynamics-aware unsupervised discovery of skills, 2020. URL https://arxiv.org/abs/1907.01657.

Noah Shinn, Beck Labash, and Ashwin Gopinath. Reflexion: an autonomous agent with dynamic memory and self-reflection. *arXiv preprint arXiv:2303.11366*, 2023.

Ishika Singh, Valts Blukis, Arsalan Mousavian, Ankit Goyal, Danfei Xu, Jonathan Tremblay, Dieter Fox, Jesse Thomason, and Animesh Garg. Progprompt: Generating situated robot task plans using large language models. *arXiv preprint arXiv:2209.11302*, 2022.

Gemini Team, Rohan Anil, Sebastian Borgeaud, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M Dai, Anja Hauth, Katie Millican, et al. Gemini: a family of highly capable multimodal models. *arXiv preprint arXiv:2312.11805*, 2023.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

Ikechukwu Uchendu, Ted Xiao, Yao Lu, Banghua Zhu, Mengyuan Yan, Joséphine Simon, Matthew Bennice, Chuyuan Fu, Cong Ma, Jiantao Jiao, Sergey Levine, and Karol Hausman. Jump-start reinforcement learning. In *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pp. 34556–34583. PMLR, 2023. URL https://proceedings.mlr.press/v202/uchendu23a.html.

Alexander Sasha Vezhnevets, Simon Osindero, Tom Schaul, Nicolas Heess, Max Jaderberg, David Silver, and Koray Kavukcuoglu. Feudal networks for hierarchical reinforcement learning, 2017. URL https://arxiv.org/abs/1703.01161.

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. *Transactions on Machine Learning Research*, 2024a.

Peng Wang, Shuai Bai, Sinan Tan, Shijie Wang, Zhihao Fan, Jinze Bai, Keqin Chen, Xuejing Liu, Jialin Wang, Wenbin Ge, et al. Qwen2-vl: Enhancing vision-language model's perception of the world at any resolution. *arXiv preprint arXiv:2409.12191*, 2024b.

Zihao Wang, Shaofei Cai, Guanzhou Chen, Anji Liu, Xiaojian Ma, Yitao Liang, and Team Craft-Jarvis. Describe, explain, plan and select: interactive planning with large language models enables open-world multi-task agents. In *Proceedings of the 37th International Conference on Neural Information Processing Systems*, pp. 34153–34189, 2023.

Zihao Wang, Shaofei Cai, Anji Liu, Yonggang Jin, Jinbing Hou, Bowei Zhang, Haowei Lin, Zhaofeng He, Zilong Zheng, Yaodong Yang, et al. Jarvis-1: Open-world multi-task agents with memory-augmented multimodal language models. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2024c.

Zihao Wang, Muyao Li, Kaichen He, Xiangyu Wang, Zhancun Mu, Anji Liu, and Yitao Liang. Openha: A series of open-source hierarchical agentic models in minecraft. *arXiv preprint arXiv:2509.13347*, 2025.

Zihao Wang, Muyao Li, Kaichen He, Haowei Lin, Xiaojian Ma, Anji Liu, and Yitao Liang. DeepHA: Scaling action chains elicits deep hierarchical agents. In *Proceedings of the 43rd International Conference on Machine Learning*, 2026. URL https://openreview.net/forum?id=5W0ezwAurK.

Peng Xia, Jianwen Chen, Hanyang Wang, Jiaqi Liu, Kaide Zeng, Yu Wang, Siwei Han, Yiyang Zhou, Xujiang Zhao, Haifeng Chen, Zeyu Zheng, Cihang Xie, and Huaxiu Yao. Skillrl: Evolving agents via recursive skill-augmented reinforcement learning, 2026. URL https://arxiv.org/abs/2602.08234.

Tianbao Xie, Siheng Zhao, Chen Henry Wu, Yitao Liu, Qian Luo, Victor Zhong, Yanchao Yang, and Tao Yu. Text2reward: Reward shaping with language models for reinforcement learning, 2024. URL https://arxiv.org/abs/2309.11489.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

Fangkai Yang, Daoming Lyu, Bo Liu, and Steven Gustafson. Peorl: Integrating symbolic planning and hierarchical reinforcement learning for robust decision-making, 2018. URL https://arxiv.org/abs/1804.07779.

Yifan Yang, Ziyang Gong, Weiquan Huang, Qihao Yang, Ziwei Zhou, Zisu Huang, Yan Li, Xuemei Gao, Qi Dai, Bei Liu, Kai Qiu, Yuqing Yang, Dongdong Chen, Xue Yang, and Chong Luo. Skillopt: Executive strategy for self-evolving agent skills, 2026. URL https://arxiv.org/abs/2605.23904.

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.

Wenhao Yu, Nimrod Gileadi, Chuyuan Fu, Sean Kirmani, Kuang-Huei Lee, Montse Gonzalez Arenas, Hao-Tien Lewis Chiang, Tom Erez, Leonard Hasenclever, Jan Humplik, Brian Ichter, Ted Xiao, Peng Xu, Andy Zeng, Tingnan Zhang, Nicolas Heess, Dorsa Sadigh, Jie Tan, Yuval Tassa, and Fei Xia. Language to rewards for robotic skill synthesis, 2023. URL https://arxiv.org/abs/2306.08647.

Haoqi Yuan, Chi Zhang, Hongcheng Wang, Feiyang Xie, Penglin Cai, Hao Dong, and Zongqing Lu. Plan4mc: Skill reinforcement learning and planning for open-world minecraft tasks. *arXiv preprint arXiv:2303.16563*, 2023.

Renos Zabounidis, Yue Wu, Simon Stepputtis, Woojun Kim, Yuanzhi Li, Tom Mitchell, and Katia Sycara. Scalar: Learning and composing skills through llm guided symbolic planning and deep rl grounding, 2026. URL https://arxiv.org/abs/2603.09036.

Jesse Zhang, Jiahui Zhang, Karl Pertsch, Ziyi Liu, Xiang Ren, Minsuk Chang, Shao-Hua Sun, and Joseph J. Lim. Bootstrap your own skills: Learning to solve new tasks with large language model guidance, 2023. URL https://arxiv.org/abs/2310.10021.

Siyang Zhang, Bin Li, Jingtao Qi, Xueying Wang, Fu Li, Jianan Wang, En Zhu, and Jinjing Sun. Code-BT: A code-driven approach to behavior tree generation for robot task planning with large language models. In *Proceedings of the Thirty-Fourth International Joint Conference on Artificial Intelligence*, pp. 8814–8822, 2025. doi: 10.24963/ijcai.2025/980. URL https://doi.org/10.24963/ijcai.2025/980.

Xufeng Zhao, Cornelius Weber, and Stefan Wermter. Agentic skill discovery, 2024. URL https://arxiv.org/abs/2405.15019.

Haotian Zhou, Yunhan Lin, Longwu Yan, Jihong Zhu, and Huasong Min. LLM-BT: Performing robotic adaptive tasks based on large language models and behavior trees. In *2024 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 16655–16661, 2024. doi: 10.1109/ICRA57147.2024.10610183. URL https://doi.org/10.1109/ICRA57147.2024.10610183.

A Problem Formulation

We formulate open-world, long-horizon tasks as partially observable sequential decision problems conditioned on a task instruction g. At each environment step t, the agent receives an observation o_t of the latent state s_t and executes a primitive action $a_t \in \mathcal{A}_{\text{env}}$. The environment then transitions and provides a task-dependent reward:

$$s_{t+1} \sim P(\cdot \mid s_t, a_t), \qquad r_t = R_g(s_t, a_t, s_{t+1}). \tag{4}$$

Because the state is not directly observable, decisions may depend on the agent's available observation and execution history, denoted by h_t.

Tasks such as obtaining a diamond from scratch in Minecraft require long sequences of interdependent behaviors, making learning from sparse task-level feedback difficult. We therefore consider a planner–controller representation in which a high-level executable structure coordinates a collection of low-level skill policies. Specifically, we represent an agent by $(\Pi, \mathcal{G})$, where $\Pi = \{\pi_k\}_{k=1}^{K}$ is a set of executable skill policies and $\mathcal{G}$ specifies their invocation, composition, and switching rules. Each policy implements a local behavior, such as collecting wood, crafting a tool, or mining a resource. Skill granularity describes the scope of behavior assigned to an individual policy, while skill organization describes how these policies are composed to solve the overall task.

Let $0 = t_0 < t_1 < \cdots$ denote the environment steps at which a skill is selected. The resulting hierarchical execution can be written as

$$\begin{aligned} &k_j \sim \mu_{\mathcal{G}}(\cdot \mid h_{t_j}, g), \\ &a_t \sim \pi_{k_j}(\cdot \mid h_t), \qquad t_j \le t < t_{j+1}, \end{aligned} \tag{5}$$

where $\mu_{\mathcal{G}}$ is the skill-selection rule induced by $\mathcal{G}$. A selected skill executes until it terminates or the structure switches control, so skill durations need not be fixed. This notation also accommodates deterministic routing and controllers that use only the current observation or a finite history.

The objective is to maximize the expected task return of the complete agent:

$$J(\Pi, \mathcal{G}; g) = \mathbb{E}_{\Pi, \mathcal{G}, P}\left[\sum_{t=0}^{T-1} \gamma^t r_t \,\middle|\, g\right], \tag{6}$$

where T is the episode horizon, $\gamma \in (0, 1]$ is the discount factor, and the expectation is over initial states, environment stochasticity, and agent decisions. Under this formulation, BEAT addresses two coupled learning problems: improving the execution capabilities of Π through online reinforcement learning, and revising the skill decomposition and executable structure $\mathcal{G}$ according to bottlenecks revealed by closed-loop execution. Structural updates may change both the number of skills and the behaviors assigned to them; consequently, Π is not assumed to remain fixed throughout training. Both forms of adaptation aim to improve the task-level objective in equation 6.

B Environment Details

We study BEAT in three complementary embodied-control settings spanning long-horizon planning, open-ended skill learning, and continuous manipulation. Minecraft combines visually grounded interaction with discrete graphical interfaces and is used to evaluate the organization of a pretrained controller. Full Craftax provides a fast, procedurally generated survival domain in which skill policies can be learned from random initialization. Language Table supplies a continuous-control setting with short-horizon motor actions but compositional, task-dependent objectives. Together, the three environments separate difficulties caused by visual embodiment, extended prerequisite structure, and continuous manipulation.

B.1 Minecraft

Minecraft is a three-dimensional sandbox in which an agent must navigate, gather resources, manipulate its inventory, craft and smelt items, and occasionally interact with other entities. These operations form long dependency chains: obtaining a target item may require locating raw materials, producing intermediate tools, opening a graphical interface, and recovering when a prerequisite



Figure 5: **Representative views of the three evaluation environments.** (a) A first-person Minecraft embodied-interaction scene from MineDojo (Fan et al., 2022); (b) mining and exploration in Full Craftax (Matthews et al., 2024); (c) the official Language Table simulator (Lynch et al., 2022); (d) an illustrative Minecraft crafting interface; (e) ranged combat in Full Craftax; and (f) the official real-robot Language Table setup shown for context. Our experiments use Minecraft 1.16.5 through MineStudio, symbolic Full Craftax, and the simulated state-only Language Table environment. Panels (d) and (f) illustrate the corresponding GUI and physical platform rather than frames from our experimental runs. Panel (d) is adapted from an Xbox México image distributed through Wikimedia Commons under CC BY 3.0.

is missing. In our mixed-start suite, the spawn region is a controlled flat 17×17 area. Cases vary the available inventory, completed prerequisites, resource and container placement, and, for selected tasks, the presence of live entities. An agent must therefore select a route from observable conditions rather than replay one fixed action sequence.

We use Minecraft 1.16.5 through MineStudio. Following the human-interface conventions of MineDojo and VPT (Fan et al., 2022; Baker et al., 2022), embodied interaction is ultimately realized through keyboard and mouse controls, while crafting, smelting, equipment management, and container interaction take place in graphical user interfaces. In the reported Minecraft skill-organization comparisons, physical behaviors are executed by the same frozen STEVE-1 controller across methods (Lifshitz et al., 2024), whereas shared scripted GUI skills implement interface operations. Both types of action consume the episode interaction budget.

Table 4 summarizes the corresponding human-equivalent controls. MineStudio represents each environment action with a factorized button branch and camera branch. Under the embodied `jarvis_low_level` profile, STEVE-1 selects among 864 compatible button combinations and 121 camera codes obtained from 11 bins on each camera axis. Inventory and hotbar operations are masked from this embodied policy and are instead issued by the shared GUI scripts through the same environment action interface.

The pretrained controller receives first-person visual observations together with a short textual skill instruction. At the planning level, BEAT uses only information available through the agent interface, including the observed inventory, detected nearby entities and visible blocks, and its own execution history. World seeds and hidden scenario identifiers are not exposed to the planner. The construction of the mixed-start task suite and its held-out evaluation protocol are described in Appendix E.1.

B.2 Full Craftax

Craftax is a JAX-based benchmark designed for high-throughput open-ended reinforcement learning (Matthews et al., 2024). We use the Full Craftax environment, which represents a two-dimensional,

Table 4: Human-equivalent low-level actions used in our Minecraft experiments. STEVE-1 generates embodied movement, interaction, and camera actions, while shared GUI scripts control inventory and hotbar operations and may also issue interaction or camera actions inside graphical interfaces. Compatible controls can be executed jointly.

Action type	Human input	Environment effect
Locomotion	W	Move forward.
	S	Move backward.
	A	Strafe left.
	D	Strafe right.
	Space	Jump or swim upward.
	Left Shift	Sneak.
	Left Ctrl	Sprint.
Interaction	Left mouse	Attack or break in-world; select a slot in a GUI.
	Right mouse	Use or place in-world; transfer items in a GUI.
	Q	Drop the selected item.
Inventory	E	Open or close the inventory interface.
	Keys 1–9	Select the corresponding hotbar slot.
Camera/cursor	Mouse X	Adjust horizontal view (yaw) or GUI cursor position.
	Mouse Y	Adjust vertical view (pitch) or GUI cursor position.
No-op / wait	No input	Issue no control while waiting for an interface or process.

procedurally generated world with partial observability and a discrete action space. Progress requires the agent to balance survival needs with exploration, resource collection, crafting, equipment upgrades, and combat. Later achievements depend on increasingly long chains of prerequisites and on transitions into additional regions of the world, making them substantially harder to discover from sparse feedback.

Our policies use the symbolic observation interface. It encodes the agent's local spatial surroundings together with non-spatial variables such as its inventory, equipment, and status, while leaving the remainder of the generated world unobserved. At every environment step, the policy selects one of 43 discrete native actions. Table 5 groups the complete action set by its environment-level function.

In the Full Craftax experiments reported in this paper, policies are initialized from random weights and receive neither demonstrations nor a pretrained controller. We evaluate the native achievements COLLECTDIAMOND, ENTERDUNGEON, UNLOCKGNOMISHLADDER, and ENTERGNOMISHMINES; the achievement flags also provide a common success criterion across training objectives. The optimization, reset, reward, and evaluation settings are given in Appendix E.2.

B.3 LANGUAGE TABLE

Language Table is a simulated tabletop manipulation environment in which a robot end effector pushes colored blocks to satisfy compositional spatial goals (Lynch et al., 2022). We retain the official physics and native task predicates but use a custom privileged state-only interface rather than the benchmark's original RGB-and-language behavioral-cloning protocol. This setting therefore studies online continuous control without visual-representation learning.

Each observation is a fixed-normalized 31-dimensional vector. It contains the positions and orientations of four blocks, the actual and commanded end-effector coordinates, the source identity, the target position and identity (or avoid-group mask), and the native goal radius. The policy outputs a two-dimensional continuous action $a_t = (a_x, a_y) \in [-1, 1]^2$, which is scaled to the planar end-effector command $0.1a_t \in [-0.1, 0.1]^2$ before being passed to the simulator. The environment runs at 10 Hz, and each episode contains at most 200 actions. The zero vector requests no planar

Table 5: Low-level action protocol in Full Craftax. The policy emits one categorical action ID at each environment step. Action names follow the native Full Craftax interface.

Action type	Discrete actions (ID)	Environment effect
No-op	NOOP (0)	Apply no intentional action.
Navigation	LEFT (1), RIGHT (2), UP (3), DOWN (4)	Move on the local grid.
Contextual interaction	DO (5)	Interact with, collect, mine, or attack the object in front of the agent.
Recovery	SLEEP (6), REST (17)	Invoke the corresponding recovery mechanic when its preconditions hold.
Placement	PLACE_STONE (7), PLACE_TABLE (8), PLACE_FURNACE (9), PLACE_PLANT (10), PLACE_TORCH (28)	Place a carried object in the world.
Pickaxe crafting	MAKE_WOOD_PICKAXE (11), MAKE_STONE_PICKAXE (12), MAKE_IRON_PICKAXE (13), MAKE_DIAMOND_PICKAXE (20)	Craft a pickaxe of the selected tier.
Sword crafting	MAKE_WOOD_SWORD (14), MAKE_STONE_SWORD (15), MAKE_IRON_SWORD (16), MAKE_DIAMOND_SWORD (21)	Craft a sword of the selected tier.
Other crafting	MAKE_IRON_ARMOUR (22), MAKE_DIAMOND_ARMOUR (23), MAKE_ARROW (25), MAKE_TORCH (38)	Craft armour, ammunition, or a utility item.
Floor traversal	DESCEND (18), ASCEND (19)	Move between adjacent dungeon floors when a ladder is available.
Ranged and magic	SHOOT_ARROW (24), CAST_FIREBALL (26), CAST_ICEBALL (27)	Execute a ranged or magical attack.
Potions and reading	DRINK_POTION_RED/GREEN/BLUE/PINK/CYAN/YELLOW (29–34), READ_BOOK (35)	Consume an item whose effect may depend on the current episode.
Enchantment and attributes	ENCHANT_SWORD (36), ENCHANT_ARMOUR (37), LEVEL_UP_DEXTERITY/STRENGTH/INTELLIGENCE (39–41), ENCHANT_BOW (42)	Improve equipment or an agent attribute.

motion, and blocks are manipulated by pushing rather than through a separate gripper action. Although these trajectories are much shorter than those in Minecraft or Craftax, a controller must still determine which object to manipulate, where it should be moved, and how to sequence intermediate positioning behaviors.

This interface focuses the study on whether an initially coarse control specification should remain intact or be divided into smaller manipulation phases. We consider BLOCKTOABSOLUTE, BLOCKTOBLOCK, BLOCKTORELOC (the official BlockToRelativeLocation task), and SEPARATEBLOCKS, using the environment's native task-success predicates in every comparison. Additional training and evaluation details appear in Appendix E.3.

C METHOD DETAILS

This appendix expands the execution and update rules of Algorithm 1. BEAT maintains an executable control tree $\mathcal{T}$, controller training state Θ, skill-entry buffers $\{\mathcal{D}_v\}$, and planner memory $\mathcal{M}$. The tree specifies what each skill should accomplish and how skills compose; reinforcement

learning improves the associated controllers. Execution feedback connects these two levels: it determines where to allocate policy learning and when to reconsider the skill organization.

C.1 Skill Specification and Tree Execution

A leaf v specifies a local objective, a controller π_v, a completion predicate c_v, a local training reward r_v, and an execution horizon H_v. The objective describes the behavior assigned to the controller, whereas c_v determines when control passes to the next stage. The reward r_v supplies the learning signal for that behavior and may include intermediate progress as well as completion. These local quantities are distinct from the task-level objective used to evaluate the complete agent. If a skill is implemented by an existing tool or script, it uses the same execution interface but has no policy parameters to optimize.

Let h_t denote the observation and execution history available to the executor at time t. On entering v, the executor checks $c_v(h_t)$. An already satisfied skill completes immediately; otherwise, π_v acts until completion, failure, or its horizon is reached. The executor records the entry context, actions taken, termination reason, and resulting handoff state.

A SeqAnd node executes its children in their specified order. It activates the first unfinished child and advances when that child completes; the parent completes after all children have completed. Completion is recorded in the execution history, so later stages can consume resources produced by earlier stages without reopening them. A XOR node evaluates its selector $f_v(h_{t_j})$ when entered at time t_j and executes only the selected child. That choice is held for the current invocation, and the selected child's completion completes the parent. These rules apply recursively when a child is itself a subtree. The executor maintains the active leaf, completed children, and selected branches for the current episode. This episode-level state is separate from planner memory $\mathcal{M}$, which persists across training and refinement attempts. Once a tree is specified, actions and control transfers follow its controllers, predicates, and selectors; the brain is invoked to revise the tree between learning phases, rather than to plan each action.

C.2 Execution Evidence and Bottleneck Identification

A fresh rollout starts from the task's initial-state distribution and executes the complete current agent. For each visited leaf, it supplies entry contexts, completion or failure outcomes, interaction lengths, and handoff states, together with the final task outcome. BEAT distinguishes *action-required attempts*, where the controller must act, from zero-action completions, where an upstream skill or the initial state has already satisfied the objective. Local controller success is measured on action-required attempts. A leaf that is never reached is not counted as a controller failure; its lack of visits is interpreted through upstream outcomes and routing.

The brain considers leaves with sufficient observed support and uses their local learning curves, failure trajectories, and downstream consequences to select a bottleneck b. This is more informative than selecting a leaf solely by its raw failure count. For example, a stage can frequently meet its own completion condition yet leave the next controller in states from which it rarely succeeds. Such evidence motivates revisiting the handoff or skill boundary, even when the upstream stage's local success rate is high.

C.3 Skill-Entry State Collection and Local Training

For each leaf v, a finite buffer $\mathcal{D}_v$ stores recent action-required entry states encountered by the complete agent, together with the context needed to initialize execution at that leaf. Already satisfied entries are excluded. The buffer therefore represents the situations in which the current organization actually calls upon π_v to act.

Given bottleneck b, local training samples an entry from $\mathcal{D}_b$, restores the environment to that entry, and collects a new rollout using the current controller and reward r_b. The stored entries are rollout initializations, not stored action trajectories reused as on-policy training data. The controller retains its ordinary observation interface after restoration. This concentrates interaction on the bottleneck without repeatedly executing an already reliable prefix.

Fresh closed-loop rollouts remain interleaved with local learning. Newly encountered entries refresh the finite buffers, while final task outcomes reveal whether local progress benefits the complete agent. When a controller or routing rule changes, these rollouts also expose the resulting changes in downstream entry distributions. Thus, local learning uses states reached by the agent, and the global stream continually updates what those states are.

C.4 Plateau Detection and Structural Proposals

BEAT first trains the bottleneck under its current skill definition. To detect a plateau, it averages local success rates over fixed-length blocks of training epochs. A block that strictly exceeds the best previous block mean resets the no-improvement counter; ties do not reset it. For bottleneck depth d_b, the patience in complete blocks is

$$p(d_b) = p_0 + 2d_b, \tag{7}$$

where p_0 is the base patience. After $p(d_b)$ blocks without improvement, persistent failures trigger structural reconsideration. The reference local-RL configuration uses $p_0 = 2$ and one training epoch per block; thus depths 0, 1, and 2 receive patience of 2, 4, and 6 complete blocks, respectively. An unsplit root skill or a direct child of the root has $d_b = 0$; each additional nested level increments d_b. Here a training epoch is a rollout-and-learning phase, not one PPO pass over a minibatch. Numerical settings and domain specializations are collected in Section C.6.

At this point, the brain uses the current tree, bottleneck evidence, and memory to propose a change (Equation 1). KEEP retains the organization for the next learning phase. A structural proposal replaces the selected leaf with a local subtree, specifying its child objectives, completion predicates, local rewards, and controller initializations, along with a selector for conditional composition. The surrounding tree is retained. The distinction between the two refinements is the source of difficulty: SEQAND separates successive behaviors within an invocation, whereas XOR separates alternative behaviors across entry contexts.

For a sequential refinement, the proposed intermediate boundaries should define useful handoffs: completing v_i should place the agent in a state from which v_{i+1} can pursue its objective. For a conditional refinement, f_v must select a child from information available at entry. For example, acquiring a resource and then using it is sequential composition; choosing an acquisition strategy according to the current inventory is conditional composition. Further refinements can be applied recursively to the resulting leaves.

Before adapting controllers for a conditional refinement, BEAT applies the proposed selector to the observed entry contexts. For an ℓ-way split, let

$$n_i = \sum_{h \in \mathcal{D}_b} \mathbf{1}\{f_b(h) = i\}, \qquad i = 1, \ldots, \ell, \tag{8}$$

where h denotes the context stored with an entry. Each entry must have a defined routing outcome, and every child must satisfy $n_i \geq m$, where m is the minimum branch-support threshold. With $m = 32$ in the restored-entry RL implementation, a binary split requires at least 64 entries, with at least 32 routed to each child. Only proposals with sufficient observed support advance to controller adaptation.

C.5 Candidate Adaptation, Comparison, and Memory

Before testing a structural change, BEAT retains the incumbent $(\mathcal{T}, \Theta)$ and creates a candidate training state for $\mathcal{T}'$. Here Θ includes policy and value-function parameters and their optimizer state. Unchanged skills preserve their existing training state. New or redefined skills are initialized from compatible existing controller weights or the base controller specified by the proposal; optimizer state is continued when resuming the same compatible training problem, and initialized anew for a new skill. The affected controllers then train on their assigned local objectives before the candidate is compared with the incumbent. Fresh execution of the candidate supplies entry states and handoff evidence for the revised subtree.

After candidate adaptation, BEAT samples N fresh task initial states $\{s_0^{(j)}\}_{j=1}^{N}$ from the task's initial-state distribution. For each j, both the incumbent and the candidate start from the same $s_0^{(j)}$

Table 6: **Local-RL reference settings.** A block aggregates training epochs; it is distinct from a PPO optimization epoch or a planner call.

Quantity	Setting
Base patience p_0	2 non-improving complete blocks
Block length B	1 training epoch
Short-block epoch budget	524,288 environment transitions
Depth-dependent patience	$p(d_b) = 2 + 2d_b$ blocks
XOR branch support m	32 observed entry contexts per child
Paired comparison size N	256 fresh starts per agent, using the same initial states
Local/global rollout allocation	Approximately 90% local and 10% fresh

and execute under the same interaction limit and task-level objective, producing paired trajectories (τ_j, τ'_j). The trees and controller parameters are held fixed during evaluation. The reference comparison uses $N = 256$ initial states, i.e., 256 episodes per agent rather than 256 episodes shared between the two. Let $R_{\text{task}}(\tau)$ be the return of a complete trajectory; for a success-rate objective, it is the indicator of final task completion. The two return estimates are

$$\widehat{J} = \frac{1}{N}\sum_{j=1}^{N} R_{\text{task}}(\tau_j), \qquad \widehat{J}' = \frac{1}{N}\sum_{j=1}^{N} R_{\text{task}}(\tau'_j). \tag{9}$$

The two matched closed-loop performance estimates are then passed to the acceptance rule together with the global planner memory M. In our implementation, ACCEPT is instantiated by GPT-5.6, which receives $(\widehat{J}, \widehat{J}')$ and the accumulated refinement memory and returns a binary PROMOTE/ROLLBACK decision.

After an attempt, $\mathcal{M}$ records the targeted skill, observed failure pattern, proposed change, controller-adaptation history, and task-level comparison, including whether the candidate was retained. Both accepted and rejected attempts inform subsequent calls, so the brain can distinguish unresolved bottlenecks from changes already tried.

C.6 Implementation Settings and Domain Specializations

Table 6 instantiates the local-learning quantities used above. These are settings of the reusable restored-entry RL implementation; the frozen-controller Minecraft specialization does not run local PPO or apply an RL-plateau stopping rule.

The longer-block Craftax Full configuration uses 9,961,472 transitions per training epoch with the same $B = 1$ and $p_0 = 2$. The Craftax Classic local-training configuration instead uses $B = 4$, $p_0 = 3$, 524,288 transitions per epoch, and $N = 512$ for its root comparison; the plateau statistic is unchanged.

Controller architectures. Craftax policies consume the symbolic observation vector, augmented with the configured observable resource/entity offsets where used, rather than rendered RGB frames. The actor and critic are separate feed-forward networks: each has three hidden layers of width 512 with $\tanh$ activations. The actor outputs categorical logits over the environment's discrete action set; the critic outputs one scalar value. They do not share weights with each other and have no recurrent state.

In the Craftax Full skill-conditioned learner, the first actor layer is shared across skills and the remaining two layers and action output are skill-specific. The critic independently follows the same shared-first-layer organization, with routing selecting the active skill parameters. The Classic local learner instead uses independently parameterized copies of this three-layer actor–critic. All begin from random initialization, with orthogonal hidden-layer weights (gain $\sqrt{2}$), zero biases, actor-output gain 0.01, and critic-output gain 1. Refinement can copy the parent controller into the new skill slots before learning their separate objectives. For scale, the Full PPO learner collects $1024 \times 64 = 65{,}536$ transitions per update batch and uses four optimization epochs with eight minibatches, i.e., 8192 transitions per minibatch.

In Minecraft, every neural leaf uses the same pretrained, frozen STEVE-1 controller (Lifshitz et al., 2024). The local checkpoint takes 128×128 RGB observations through an IMPALA-style convolutional encoder, followed by four recurrent Transformer blocks of width 2048 with 16 attention heads and a 256-step attention memory. Its convolutional channel multipliers are $(16, 32, 32)$ with width multiplier 8. The original action heads produce a categorical distribution over 8641 button combinations and 121 discretized camera actions.

A short leaf instruction is encoded by MineCLIP and the pretrained text-to-visual prior into a 512-dimensional condition, projected into the policy. We use classifier-free guidance with scale 6; in this implementation the logits are $7\ell_{\text{cond}} - 6\ell_{\text{uncond}}$. Conditions are sampled at skill invocation and actions are sampled stochastically. Distinct leaves can therefore request different behaviors without creating or training new neural heads. Crafting, smelting, equipment changes, and chest interactions use the shared GUI scripts. The pretrained value head is retained in the checkpoint but is not optimized or used to select high-level routes.

Because controller optimization is disabled, persistent execution failures directly motivate structural refinement without an RL plateau. The planner uses inventory, observable nearby entities/blocks, and execution history; selectors cannot read the world seed or hidden start identifier. Candidate organizations are exercised closed-loop under the same task budget; the local-RL branch-data threshold m does not apply to these frozen-controller development batches.

Planner configuration and budget. The reference brain configuration is GPT-5.6 (model identifier `gpt-5.6-sol`) with reasoning effort `xhigh`.[1] The model identifier is not pinned to a date-stamped snapshot. Temperature and top-p are not explicitly overridden, and no LLM sampling seed is set. Calls return a JSON specification and a short rationale; environment actions are handled by the executor.

A planner call denotes one top-level specification-generation or revision request, not a low-level controller invocation. In the Minecraft mixed-start suite, the standard budget is three calls per task: initial construction (r0), followed by two feedback-conditioned revisions (r1 and r2). Lantern uses one additional revision (r3). This amounts to a nominal budget of $29 \times 3 + 4 = 91$ specification requests for the thirty listed task artifacts, excluding transport retries and earlier development pilots. Evaluation freezes these artifacts and makes *zero* high-level LLM calls.

The long-running Craftax Full skill learner schedules planner reviews no more frequently than every 50M environment transitions. This gives at most 16 scheduled review opportunities within an 800M-step budget (20 within 1B), in addition to initial specification construction. Busy review slots are skipped, and completed calls may return KEEP, so these budgets do not count accepted changes. The review interval is distinct from the local-RL block length.

LLM token cost is incurred during development; frozen evaluation incurs only environment and controller compute. A complete dollar total was not retained. Token-metered cost is $C = (p_{\text{in}}T_{\text{in}} + p_{\text{cache}}T_{\text{cache}} + p_{\text{out}}T_{\text{out}})/10^6$, where the three token counts separate uncached input, cached input, and billed output (including reasoning), and the prices are per million tokens for the service tier used.

D ADDITIONAL EXPERIMENTS

D.1 LANGUAGE TABLE

Setup. We additionally evaluate BEAT in the official Language Table simulator (Lynch et al., 2022) to study whether adaptive organization transfers beyond discrete Craftax and Minecraft environments. Policies receive a 31-dimensional geometric state containing block poses, end-effector coordinates, source and target identities, and target geometry. We consider four tasks: BlockToAbsolute, BlockToBlock, BlockToReloc, and SeparateBlocks.

BEAT and the fixed-specification baseline start from the same single-leaf task specification and randomly initialized PPO policy. BEAT may reorganize the task into at most three skills and adapt

[1] Model naming and reasoning-effort terminology follow the OpenAI model documentation.

Table 7: Language Table success rates (%) with geometric state inputs. Results aggregate 192 fresh evaluation episodes (3 seeds $\times$ 64 episodes) at the common 9.961M-step checkpoint. Bold denotes the best result on each task.

Task	Fixed spec.	BEAT	SAC
BlockToAbsolute	0.00	**2.60**	0.52
BlockToBlock	0.00	**3.65**	0.00
BlockToReloc	2.60	**6.77**	0.52
SeparateBlocks	**85.42**	81.77	27.60

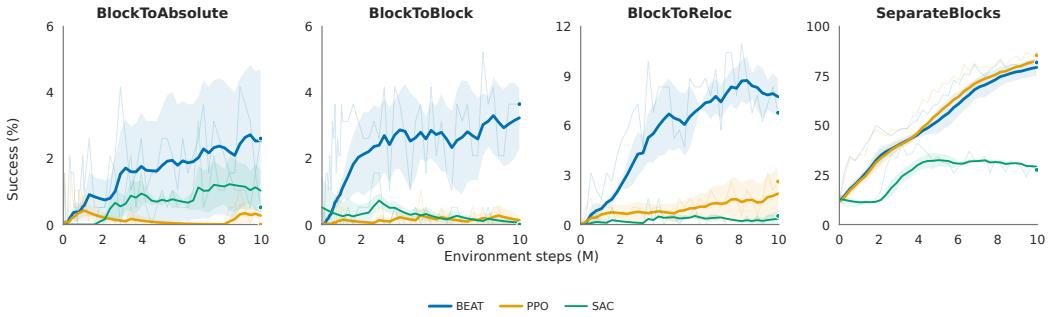


Figure 6: Language Table learning curves over three training seeds. Curves show the mean of seed-wise causal EWMAs with a 1M-step half-life; shading denotes one sample standard deviation across seeds.

the associated shaping, whereas the fixed variant retains its initial specification throughout training. We additionally include SAC (Haarnoja et al., 2018) as a standard off-policy RL reference.

Each method uses three training seeds and a common 10M-transition budget. At the common 9.961M-step checkpoint, each seed is evaluated on 64 fresh episodes, yielding 192 evaluation episodes per method and task.

Results. As shown in Table 7, BEAT improves over the fixed-specification baseline on three of the four tasks, including BlockToReloc, where success increases from 2.60% to 6.77%. The fixed specification remains slightly stronger on SeparateBlocks (85.42% vs. 81.77%). Overall, these results provide additional evidence that adaptive task organization can improve policy learning beyond the discrete long-horizon domains considered in the main experiments.

E Experimental Details

This appendix provides implementation details for the Minecraft and Craftax experiments in the main paper, followed by the protocol for the additional Language Table study. Unless otherwise stated, methods within each comparison share the same task definitions, interaction budgets, evaluation conditions, and success criteria.

E.1 Minecraft

Environment and task construction. We use Minecraft 1.16.5 through MineStudio and construct a 30-task mixed-start acquisition suite, divided evenly into Simple, Medium, and Hard tiers. Simple tasks contain short dependency chains and limited resource choice; Medium tasks span multiple observable resource or progress states; Hard tasks additionally involve longer dependencies, alternative routes, and occasionally combat. Each task is considered successful when the target item appears in the agent's inventory.

Initial states vary the available ingredients, completed prerequisites, nearby resources, entities, and useful chest contents. Task-appropriate tools, crafting stations, and fuel are supplied where required.

Environments use a flat 17×17 spawn region populated with task-specific resources. Compared methods use matched initial states, resource placements, and interaction budgets.

Controller and skill interface. All Minecraft methods interact with the environment through the same STEVE-1 controller (Lifshitz et al., 2024), using guidance scale 6 and stochastic text conditioning. Physical behaviors are invoked through short textual skill instructions, while crafting, smelting, equipment changes, and chest interaction use a shared set of GUI skills. All controller and GUI actions count toward the task interaction budget.

The pretrained STEVE-1 controller is kept frozen in the main skill-organization comparison so that methods differ only in high-level planning and organization. Goal completion is determined directly from the observed inventory. The shared executor may skip already satisfied subgoals and invoke prerequisite producers already represented in the current plan. High-level selectors use only observable information, including inventory, nearby entities and blocks, execution history, and skill outcomes.

Baseline planners. We adapt Plan4MC, JARVIS-1, XENON, DEPS, Optimus-1, Reflexion, and a GPT-5.6 goal-only planner to the same MineStudio environment and STEVE-1 skill interface. High-level language-model planning uses GPT-5.6 at xhigh reasoning effort.

Plan4MC leverages its precomputed task graph for goal-conditioned subgoal decomposition. JARVIS-1 uses an adapted recipe and skill-chain prior. XENON initializes planning from its released dependency knowledge. DEPS follows the Describe–Explain–Plan–Select organization using the disclosed task context. Optimus-1 uses its hierarchical dependency prior with initially empty experience memory. Reflexion converts execution feedback into verbal reflections stored in episodic memory to guide subsequent planning attempts. The GPT-5.6 goal-only baseline receives only the target item and the available skill interfaces, without an explicit task decomposition.

These adaptations preserve the characteristic high-level organization of each method while sharing the same low-level controller and execution interface. Planning artifacts may be constructed or refined during development according to each method's update mechanism; the resulting artifact is then used for evaluation rollouts without modification within the rollout batch.

BEAT refinement. BEAT begins from a coarse goal specification and iteratively reorganizes it using rollout feedback. Development feedback includes observed inventories, skill outcomes, execution events, and failure information. We perform at most three refinement rounds in the Minecraft experiments.

The Chain and Code variants in the representation ablation use the same rollout feedback and revision budget as BEAT. Chain restricts organization to SEQAND-connected skill sequences. Code instead implements a Python `next_skill(state)` controller that selects skills from observable state. BEAT uses the SEQAND/XOR representation described in Section 3, supporting both sequential dependencies and state-dependent alternatives.

Execution and evaluation. Direct skill execution uses the timeout associated with the active skill. Missing-material recovery receives an attempt budget of at least 1,800 environment steps, increased to 3,200 steps for iron ore when sufficient episode budget remains.

Development and final testing use disjoint environment-seed sets. Planning artifacts are constructed and refined during development, then frozen for final evaluation on the test seeds. All Minecraft results in the main paper use 32 test episodes per task. Task variants and test initializations are matched across compared methods, and success rates are computed directly from task completion over these episodes. Tier averages assign equal weight to all ten tasks in each tier.

For the low-level RL ablation, BEAT w/o RL and BEAT w/ RL use the same high-level organization and evaluation protocol. BEAT w/ RL additionally performs targeted PPO optimization on bottleneck skills identified from rollout feedback. The corresponding optimization settings are reported in Table 8.

Table 8: Optimization settings for the PPO-based learners in Full Craftax, the Minecraft controller-adaptation ablation, and Language Table. Minecraft PPO is used only in the low-level adaptation experiment; the controller remains frozen in the main skill-organization comparison.

Setting	Full Craftax	Minecraft RL ablation	Language Table
Parallel environments	1,024	32	32
Rollout batch	65,536 transitions	128 episodes per expert	8,192 transitions
PPO epochs	4	2	10
Minibatching	8192 steps	up to 5,120 steps (40×128)	4,096 steps
Gradient accumulation	–	16 minibatches	–
Learning rate	2×10^{-4}	2×10^{-5}	3×10^{-4}
Discount γ	0.99	0.999	0.99
GAE λ	0.8	0.95	0.95
PPO clipping	0.2	0.2	0.2
Entropy coefficient	0.01	0	0
Pretrained-policy KL coefficient	0	0.2	0

E.2 Full Craftax

Environment and training budget. We evaluate four long-horizon goals in Full Craftax (Matthews et al., 2024): CollectDiamond, EnterDungeon, UnlockGnomishLadder, and EnterGnomishMines. All methods are trained for at most 800M environment steps, and the low-level policies are initialized from random weights. Final performance is evaluated from fresh task initializations using the native success criterion of each goal.

BEAT, Weak-prior PPO, and Strong-prior PPO use the same PPO backbone. Their principal optimization settings are summarized in Table 8.

Task specifications and adaptation. The fixed-prior ablation compares BEAT against two PPO variants whose task organization is specified before training and then kept fixed.

Weak-prior PPO uses a manually designed stage decomposition. Strong-prior PPO follows the Scalar-style construction (Zabounidis et al., 2026), combining richer prior world knowledge with a VLM to produce a detailed task-specific stage decomposition. Neither fixed-prior baseline revises its organization during training.

BEAT instead starts from a single broad leaf and progressively constructs and revises its skill organization from rollout feedback. The planner uses execution evidence to identify bottlenecks and may refine skill decomposition, routing, and the associated local shaping as training proceeds.

CollectDiamond primarily uses specification-based shaping during training. For the remaining goals, training combines environment reward with task-specific specification reward. Regardless of the training objective or task organization, all reported results use the same native task-success criterion for final evaluation.

Comparison methods. We compare against general RL baselines PPO (Schulman et al., 2017), PQN (Gallici et al., 2025), GRPO (Shao et al., 2024), and SAC (Haarnoja et al., 2018). We additionally construct Craftax adaptations of ARM-FM (Castanyer et al., 2026), Voyager (Wang et al., 2024a), and SkillRL (Xia et al., 2026). The fixed-prior analysis additionally includes Weak-prior PPO and Strong-prior PPO as described above.

ARM-FM organizes learning through a language-generated reward machine, Voyager through an incrementally constructed skill library, and SkillRL through structured skill memory. Each method is adapted to the common Craftax observation and action interface while retaining its characteristic organization mechanism.

Checkpoint selection and evaluation. All methods use the same 800M-step interaction budget. During training, the success statistics used for learning curves and checkpoint selection are computed exclusively from fresh-start closed-loop rollouts. These rollouts begin from the task's initial-

state distribution and execute the complete agent, with success determined by the native criterion for the target task. Rollouts restored from skill-entry buffers are excluded from these statistics; local skill-completion rates are not used as substitutes for complete-task success.

For each method–task pair, we select the checkpoint with the highest observed fresh-start batch success rate within the interaction budget. The selected checkpoint is then evaluated independently on 256 new episodes from fresh task initializations. Table 2 reports these independent evaluation success rates, and the four-task macro-average is their unweighted mean.

Learning curves use the same fresh-start closed-loop success statistics recorded throughout training and are smoothed with a 20M-step causal EWMA for visualization. All tabulated values are computed from the independent 256-episode evaluation of the selected checkpoint rather than from the smoothed learning curves.

E.3 Language Table

Environment and observations. The additional Language Table study uses the official simulator and native task-success predicates. Policies receive a fixed normalized 31-dimensional geometric observation containing block poses, end-effector coordinates, source and target identities, and target geometry. Policies are initialized from random weights, and the phase-conditioned policy has width 128.

Training uses the native success reward together with distance-based shaping. PPO uses the Language Table settings in Table 8, with 256 rollout steps per environment.

Adaptive organization. The fixed-specification PPO baseline and BEAT begin from the same single-leaf task specification and randomly initialized PPO policy. The fixed baseline retains this specification throughout training. BEAT may instead split an active skill, revise its associated shaping, and reorganize the task into at most three leaves. Newly created child policy heads inherit the parent parameters and Adam optimizer state.

Evaluation. Each method uses three training seeds and a common 10M-transition training budget. At the common 9.961M-step checkpoint, each seed is evaluated on 64 fresh episodes, giving 192 evaluation episodes per method and task. The SAC reference uses the same state representation, transition budget, and evaluation protocol.

Learning curves are smoothed independently for each seed using a causal EWMA with a 1M-step half-life. Solid curves report the mean across seeds, shading shows one sample standard deviation, and tabulated results are computed directly from raw evaluation outcomes.

F Case Studies of Local Control-Tree Refinement

We provide three case studies illustrating how BEAT performs local control-tree refinement in Minecraft, Craftax, and Language Table. Rather than reproducing planner prompts or implementation schemas, we focus on the resulting structural changes: what execution evidence motivates a refinement, which local control problem is replaced, and what useful state the new handoff exposes to downstream behavior.

Across domains, a refinement changes only a selected leaf and preserves the surrounding tree, the final task objective, and the overall interaction budget. A selected leaf may be retained or replaced by a local SeqAnd or XOR subtree when the supported operators permit it. Thus organizational evolution changes which behaviors are assigned to controllers and how those behaviors compose; it does not require the underlying controller architecture itself to change.

F.1 Prior Conditions and Common Refinement Interface

Prior conditions. BEAT starts from a coarse executable specification. In the Craftax prior ablation, this initialization consists of a single broad leaf covering the complete task rather than a pre-specified sequence of intermediate stages. The leaf has the task-level completion condition and,

Table 9: **Prior conditions in the main-text ablation.** Both fixed-prior controls train their policies while retaining their initial specification. BEAT instead starts from a coarse executable specification and may refine its control tree from execution evidence.

Setting	Initial specification	Online change
Weak-prior PPO	Manually designed, fixed subgoal specification	Policy only
Strong-prior PPO	Detailed fixed stage decomposition generated by a VLM following Scalar, using rich world knowledge	Policy only
BEAT	Coarse executable specification; a single broad leaf in the Craftax prior ablation	Control tree; policy when RL is enabled

when RL is enabled, a local reward that may include supported shaping. Intermediate stages are introduced only through subsequent bottleneck-guided refinement.

This distinguishes BEAT from the two fixed-prior controls used in the main-text ablation. Weak-prior PPO supplies a manually specified set of useful intermediate subgoals, while Strong-prior PPO supplies a more detailed VLM-generated stage decomposition informed by world knowledge. Both fixed priors train their policies while retaining the supplied specification; BEAT can additionally revise its control tree.

Local executable boundaries. For a leaf v, the completion predicate determines when control is handed to the next skill,

$$c_v : h_t \mapsto \{0, 1\}, \tag{10}$$

where h_t contains the observations and execution state exposed by the task adapter. When the controller is trainable, a leaf may additionally define a local reward

$$r_v : (h_t, a_t, h_{t+1}) \mapsto \mathbb{R}. \tag{11}$$

Completion and reward have different roles: the former defines an executable handoff, whereas the latter provides a learning signal within the active control problem.

A sequential refinement is useful when one behavior should produce a specific state from which another behavior can proceed. A conditional refinement is useful when distinct observed entry contexts require different routes. For an XOR node entered at time t_j, the selector has the form

$$f_v : h_{t_j} \mapsto \{0, \ldots, \ell - 1\}. \tag{12}$$

The selector is evaluated when the subtree is entered rather than at every primitive action.

The organizer bases these changes on execution evidence associated with the current leaves, including whether a leaf is reached, whether it completes or times out, the state observed at entry and handoff, and the final task outcome. In RL-enabled domains, local learning history additionally indicates whether further controller optimization continues to improve the selected control problem. In the organization-only Minecraft setting, the controllers remain frozen, so repeated execution failures directly provide the relevant structural evidence.

The following cases illustrate the resulting refinements rather than the internal representation used to request them.

F.2 Minecraft: Conditional Resource Acquisition

Minecraft provides the organization-only setting: STEVE-1 controller weights remain frozen, while the control tree determines which behavior is invoked and when. GUI operations such as crafting, equipment changes, and inventory interaction are handled separately by existing scripts. Structural refinement therefore changes the organization of existing capabilities without gradient updates to the underlying controller.

Refining one acquisition leaf. Figure 7 shows a local refinement from the Jukebox task. The selected leaf is responsible for obtaining two oak logs. Execution evidence motivates splitting this broad acquisition behavior at a state where one log has already been obtained. The replacement first

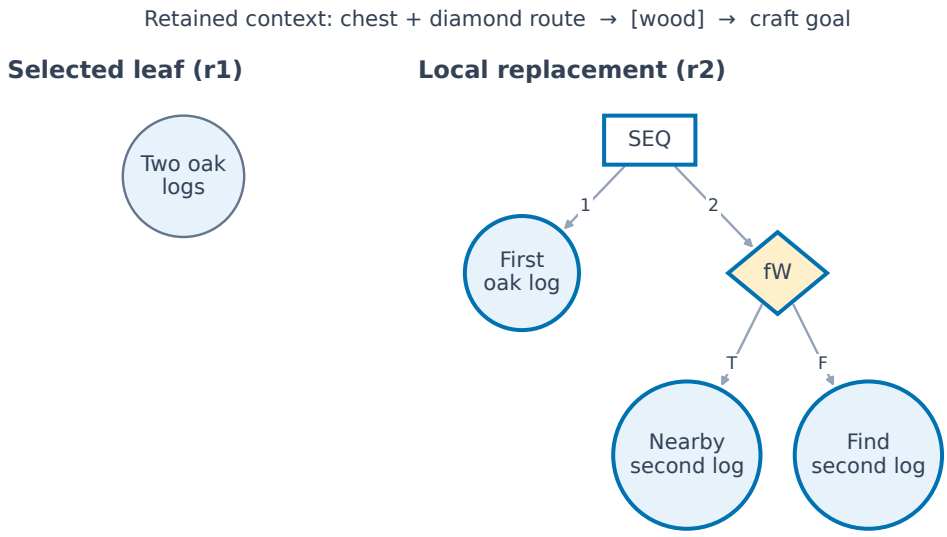


Figure 7: **Minecraft case study: conditional resource acquisition.** A single two-log acquisition leaf is replaced by a sequential subtree. After obtaining the first log, an XOR node selects direct acquisition when another oak log is detected nearby and search otherwise. The surrounding procedure is retained, and STEVE-1 weights remain frozen.

obtains one oak log and then chooses how to obtain the second: if another oak-log block is detected nearby, the agent directly acquires it; otherwise, it executes a search behavior before gathering.

The surrounding procedure is unchanged. In particular, the preceding chest and diamond-acquisition stages and the following plank and Jukebox crafting stages retain their original roles. The change is therefore a local SEQAND refinement containing an XOR route, rather than a replanning or reordering of the full task.

Let $n_{\text{oak}}(h)$ denote the number of oak logs in inventory, and let $B_{32}(h, \text{oak})$ indicate that the executor detects an oak-log block within distance 32. The relevant local completion predicates are

$$\begin{aligned} c_{\text{first}}(h) &= [n_{\text{oak}}(h) \geq 1], \\ c_{\text{near}}(h) = c_{\text{search}}(h) &= [n_{\text{oak}}(h) \geq 2]. \end{aligned} \tag{13}$$

The selector for the second-log subtree is

$$f_W(h) = \begin{cases} 0, & B_{32}(h, \text{oak}), \\ 1, & \text{otherwise.} \end{cases} \tag{14}$$

The useful distinction is not merely the numerical split between one and two logs. Obtaining the first log creates an explicit handoff at which the agent can inspect its acquisition context. When another source is already visible, the next controller can focus on direct interaction; otherwise, the assigned behavior includes searching for a new source. A negative nearby-block test therefore selects a search route rather than asserting that no oak log exists in the environment.

The same boundaries also handle mixed initial states naturally. If an episode already contains one oak log, the first child is immediately satisfied and execution proceeds to the conditional second-log route. If two logs are already held, the acquisition stage can complete without unnecessary interaction. In all cases, the downstream crafting stages receive the same required inventory as before the refinement.

F.3 CRAFTAX: SEPARATING COMBAT FROM DESCENT

Craftax illustrates refinement when the selected controller is trainable. Consider the task of entering the Gnomish mines. Let L denote the current absolute floor and K_1 the number of floor-1 hostile

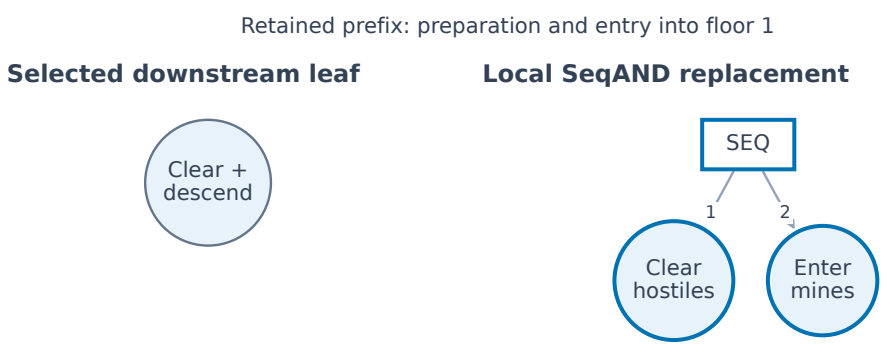


Figure 8: **Craftax case study: separating combat from descent.** A broad Mines-entry control problem is replaced by two sequential behaviors. The first clears the floor-1 hostile requirement; the second inherits the original objective of reaching floor 2. Upstream preparation and the overall task budget are unchanged.

kills. The final task criterion is

$$L \geq 2. \tag{15}$$

A broad downstream control problem responsible for reaching the mines must both clear the hostile requirement on floor 1 and subsequently locate and use the unlocked downward route. These behaviors have different control requirements. Combat requires engagement with nearby hostiles, whereas descent requires navigation after the exit has become available.

BEAT therefore introduces the intermediate boundary

$$L = 1 \ \wedge \ K_1 \geq 8. \tag{16}$$

This state has a concrete downstream consequence: the combat requirement has been satisfied and the next controller can focus on reaching and using the unlocked exit. The second child retains the original terminal condition $L \geq 2$, so defeating the required hostiles alone does not count as task completion.

Learning signals follow the new responsibilities. The structural boundary also permits the two children to receive learning signals matched to their roles. The combat child can use positive progress in the same-floor kill counter, capped at the required eight kills. The descent child instead uses progress toward the absolute target floor together with supported navigation feedback associated with the unlocked down-ladder.

These rewards do not redefine the task criterion. They optimize the local controllers created by the decomposition, while complete-agent success is still determined by entering floor 2. Likewise, the handoff occurs in the same live simulator state: completing the combat child does not reset the environment or allocate a new episode budget.

This case therefore illustrates why a useful refinement is not simply a longer task description. The intermediate predicate identifies a state at which the controller's responsibility changes qualitatively: before the boundary the agent must resolve combat, and after it the relevant problem is navigation and descent.

F.4 Language Table: Recursive Geometric Refinement

Language Table provides a complementary example in which local sequential refinement exposes geometric handoffs between manipulation behaviors. Figure 9 shows the refinement history for BlockToReloc.

A broad relocation leaf is first replaced by two control problems: establishing an appropriate pushing configuration and completing relocation to the final target. BEAT then refines only the alignment

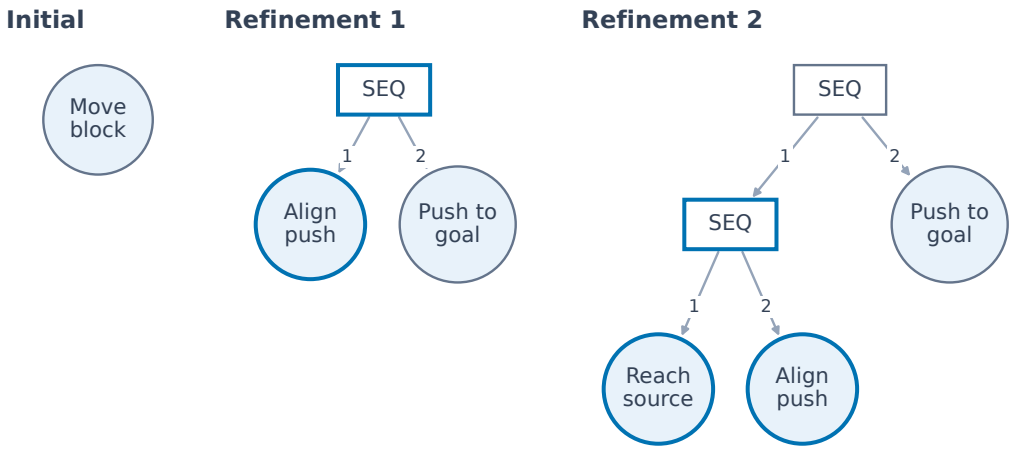


Figure 9: **Language Table case study: recursive geometric refinement.** The first local replacement separates geometric alignment from terminal relocation. A second replacement acts only on the alignment leaf, separating source approach from waypoint acquisition. The final relocation behavior and the surrounding sequence are retained.

leaf, separating approach to the source object from acquisition of the desired pushing waypoint. The terminal relocation behavior is unchanged by this second refinement.

Let e, s, and g denote the XY positions of the end effector, source block, and relocation target. The desired pushing waypoint is placed behind the source relative to the target direction:

$$p = \mathrm{clip}_{\mathrm{workspace}}\left(s - 0.05\frac{g-s}{\|g-s\|_2+\epsilon}\right). \tag{17}$$

The two intermediate completion predicates can then be written as

$$\|e-s\|_2 < \tau_s, \qquad \|e-p\|_2 < \tau_p, \tag{18}$$

for source-approach and waypoint-alignment thresholds τ_s and τ_p, respectively. The final leaf continues to use the environment's native relocation success predicate.

These boundaries define control handoffs rather than substitute success criteria. Reaching the source gives the alignment controller an appropriate entry condition; reaching the pushing waypoint gives the terminal controller a useful configuration from which to establish contact, correct residual alignment error, and move the block to its target. The downstream controller therefore remains responsible for the actual manipulation needed to satisfy the native task objective.

The refinement is recursive but local. During the second split, only the alignment leaf changes; the terminal relocation leaf remains untouched. More generally, this illustrates how BEAT can expose additional control structure without repeatedly regenerating a complete procedure: a broad behavior is retained where it is adequate and subdivided only where execution evidence supports a more useful boundary.

F.5 Summary

The three cases expose the same structural principle in different settings. In Minecraft, a conditional branch separates direct acquisition from search according to an observed resource context while keeping the controller weights fixed. In Craftax, an intermediate predicate separates combat from navigation and allows the corresponding trainable controllers to optimize different local objectives. In Language Table, recursive geometric handoffs separate approach, alignment, and terminal manipulation.

In each case, the purpose of refinement is therefore not to introduce a longer pre-specified plan. The new structure identifies an executable boundary at which either the required behavior or the relevant

entry context changes. Because the replacement is local, unaffected parts of the control tree retain their previous roles, while closed-loop task success remains the criterion for evaluating the resulting agent.

G MINECRAFT TASK SUITE

We evaluate on a 30-task Minecraft item-acquisition suite consisting of ten Simple, ten Medium, and ten Hard tasks. Each task specifies a target item and a maximum interaction budget. Episodes terminate successfully once the target item appears in the agent's inventory.

The tasks are designed to cover progressively longer acquisition and crafting chains. Simple tasks typically require a short crafting or resource-acquisition sequence; Medium tasks introduce multiple missing ingredients, tool use, or physical resource collection; Hard tasks combine several such requirements into longer multi-stage objectives.

Tables 10–12 summarize the task budgets and high-level objectives.

Table 10: **Simple Minecraft tasks.** Task length denotes the maximum number of environment frames.

Task	Length	Brief definition
S01 Crafting Table	2000	Acquire a crafting table from wood, prepared materials, or a nearby container.
S02 Red Wool	1200	Acquire white wool when necessary and combine it with red dye.
S03 Torch	1000	Obtain coal and a stick, then craft a torch.
S04 Lever	1000	Obtain cobblestone and a stick, then craft a lever.
S05 Pumpkin Pie	1400	Collect the required food ingredients and craft a pumpkin pie.
S06 Painting	1800	Obtain wool and sufficient wood-derived sticks, then craft a painting.
S07 Bread	1800	Acquire wheat from available sources and craft bread.
S08 Bucket	2000	Obtain three iron ingots, including mining and smelting when required, and craft a bucket.
S09 Iron Nugget	1600	Acquire iron and convert it into at least one iron nugget.
S10 Red Carpet	1600	Acquire wool, dye it red, and craft red carpet.

Table 11: **Medium Minecraft tasks.** Task length denotes the maximum number of environment frames.

Task	Length	Brief definition
M01 White Bed	2000	Acquire wool and wood as needed and craft a white bed.
M02 Redstone Torch	2200	Acquire redstone and wood-derived materials, including tool preparation when necessary, and craft a redstone torch.
M03 Item Frame	3000	Acquire leather and wood, convert the wood into sticks, and craft an item frame.
M04 Book	3000	Acquire leather and sugar cane, convert the cane into paper, and craft a book.
M05 Shield	3600	Acquire wood and iron, including mining or smelting when needed, and craft a shield.
M06 White Bed: Wool + Wood	3000	Acquire both wool and wood through physical interaction before crafting a white bed.
M07 Cartography Table	3800	Acquire paper and wood from the environment and craft a cartography table.
M08 Tripwire Hook	3800	Acquire iron and wood, preserving the required plank and stick forms, and craft a tripwire hook.
M09 Loom	3400	Acquire string and wood through gathering, containers, cobwebs, or combat, and craft a loom.
M10 White Banner	3800	Acquire multiple units of wool together with wood and craft a white banner.

Table 12: **Hard Minecraft tasks.** Task length denotes the maximum number of environment frames.

Task	Length	Brief definition
H01 Lantern	2600	Acquire iron-derived nuggets and a torch, then craft a lantern.
H02 Bookshelf	4200	Acquire the resources needed for books and wooden planks, then craft a bookshelf through a multi-stage recipe.
H03 Piston	4200	Acquire wood, cobblestone, iron, and redstone, then craft a piston.
H04 Lead	4200	Acquire string and slime, including combat or block conversion when necessary, and craft a lead.
H05 Powered Rail	6000	Acquire gold, redstone, and wood through a long mining and smelting chain, then craft powered rails.
H06 Bow	3600	Acquire string and wood, convert the wood into sticks, and craft a bow.
H07 Fishing Rod	3600	Acquire string and wood and craft a fishing rod.
H08 Jukebox	3800	Acquire a diamond and sufficient wood, then craft a jukebox.
H09 Dispenser	5600	Acquire or craft a bow, collect cobblestone and redstone, and assemble a dispenser through a multi-stage acquisition chain.
H10 Crossbow	5600	Acquire string, iron, and wood, craft the required tripwire hook, and then craft a crossbow.