SCALAR: Self-Supervised Composition and Learning of Skills with LLM Planning and RL

Anonymous Author(s)
Affiliation
Address
email

Abstract

A core challenge in reinforcement learning (RL) is effective exploration, particularly for long-horizon tasks. Recent approaches have explored the utility of large language models (LLMs), combining capabilities to 1) decompose objectives into skills and 2) generate code such as rewards and verifiers. However, ad hoc prompt and program designs, as well as their reliance on single proxy rewards, can lead to reward hacking and hallucinations. Furthermore, synthesizing the correct functions remains challenging without actual environment interactions. To address these challenges, we propose **Self-Supervised Composition and Learning of Skills** (SCALAR), an iterative, bi-directional framework that couples an LLM planner and low-level RL controllers through a **skill library**. The skill library is a set of skills that, when composed, define a set of furthest reachable states by the current agent. In SCALAR, the library is iteratively expanded by a high-level LLM planner in conjunction with low-level RL agents. In one direction, an LLM planner uses information in the skill library to propose new skills with (1) preconditions reachable through existing skill compositions and (2) termination conditions unachievable by current skills. Reusing existing skill compositions narrows the task of the RL agent to exploring (2) rather than returning to known states (1). In the other direction, the LLM planner refines its world knowledge *concurrently* with RL training by analyzing successful RL trajectories. We call this process *Pivotal Trajectory Analysis*. We evaluate SCALAR on the Crafter benchmark, a challenging long-horizon task, in which SCALAR achieves **86.3%** diamond-collection success, surpassing the previous state-of-the-art methods in overall performance and convergence speed. These results show that frontier-guided skill composition, together with verifier-based learning and bi-directional refinement, yields substantially more reliable long-horizon control under sparse rewards.

1 Introduction

Recent progress in large language models (LLMs) [1, 2, 3, 4, 5, 6, 7] and inference-time scaling [8, 9] has led to rapid advancements in LLM-based AI agents [10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]. However, large language models suffer from extended inference times that may not be suitable for real-time control. By contrast, Reinforcement Learning (RL) can produce strong low-level control policies given sufficient trials and supervision [21, 22, 23, 24, 25], but RL agents lack the extensive prior knowledge and explicit reasoning capabilities available to LLMs.

Integrating LLMs and RL promises to combine complementary strengths: the structured reasoning and common-sense knowledge of language models with the sample-efficient low-level control of RL. Early work has demonstrated promising results by using LLMs for reward shaping [26, 27, 28, 29, 30] and policy guidance [17, 30]. In particular, LLMs are effective at generating reward functions and

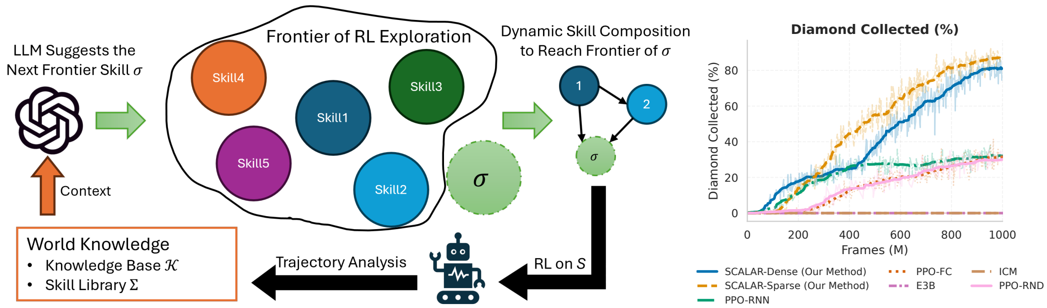

Figure 1: *Left:* SCALAR maintains 1) a frontier skill library, which is used to inform RL of next task outside of the exploration frontier, and 2) a knowledge base, which stores current knowledge and assumptions the LLM has about the environment. *Right:* SCALAR has 2x higher success rate for collecting diamond, the hardest achievement in the Crafter benchmark.

assisting with task decomposition due to their coding and problem-solving capabilities—features we seek to leverage for RL integration. However, real-world tasks often involve domain-specific knowledge and long-horizon planning requirements, which increase compositional complexity and make reasoning more challenging for LLMs [16, 31].

In this work, we propose Self-Supervised Composition and Learning of Skills (SCALAR), an iterative, bi-directional framework designed to address these challenges by tightly coupling symbolic high-level LLM planning with low-level RL. The core of SCALAR is a **skill library**, a set of skills that, when composed, define the set of furthest reachable states by the current agent. This definition formally connects the dots between RL exploration [32] with few-shot LLM planning [12]. Under SCALAR, the LLM planner proposes new candidate skills with symbolic preconditions and gains, along with reward/verifier templates; candidates are admitted only if they are both *feasible* from and *novel* with respect to the current frontier of reachable symbols. During RL training, SCALAR exploits the *feasibility* by composing existing skills from the skill library to effectively return to the starting states of new skills before initiating RL training. This design focuses RL training on *novel* scenarios and reduces the burden of long horizon explorations [32].

Under SCALAR, the LLM planner and RL controller could be viewed as a single agent tasked to expand the set of reachable symbols/states defined by the skill library. Therefore, trajectories from low-level RL controllers should not only benefit RL training, but also improve the LLM planner. Motivated by [33] we take the first few successful RL trajectories (**pivotal trajectories**) and feed them back to the planner via Pivotal Trajectory Analysis. The LLM planner reasons with the pivotal trajectories, (i) refines the proposed skill's preconditions and gains to match the environment's affordances, and (ii) expands the symbolic knowledge used for future proposals. This iterative feedback loop improves the planner's priors, enabling progressively richer and more compositional behaviors.

We evaluate SCALAR on the Crafter benchmark [34], a long-horizon, sparse-reward survival and crafting environment where purely scalar rewards and standard exploration techniques struggle. Across environments, SCALAR consistently expands the frontier of reachable states and converts symbolic proposals into executable skills with high success rates. Empirically, this produces substantially higher diamond-collection rates and shorter training episodes compared to SOTA baselines (Fig. 1), demonstrating that combining skill composition with verifier-based training enables agents to solve tasks that are otherwise difficult to reach using scalar rewards alone. Our contributions are as follows:

- Combining LLM-guided planning and RL-based skill grounding within bi-directional loop
- Formalization of frontier skill discovery, connecting RL exploration [32] and LLM planning [12]
- Pivotal Trajectory Analysis for concurrently refining skill specifications using successful rollouts
- Substantial performance and efficiency improvements on the Crafter vs PPO baselines

2 Related Work

Reward Shaping with LLMs Reinforcement learning for long-horizon tasks faces significant challenges in defining precise reward functions that effectively guide learning without introducing unintended behaviors [35, 36, 37]. To address these limitations, intrinsic motivation and reward shaping techniques have been developed to provide additional unsupervised learning signals [38]. Recent work has investigated leveraging the code generation and reasoning capabilities of language models to automatically construct reward functions from task descriptions. Early explorations of LLMs for reward shaping [26, 27, 28] began with applications where the LLM generates a reward function for the whole task without direct involvement in agent interactions. [39] extends this by allowing LLMs to provide dynamic reward adjustments based on agent interactions, assigning positive feedback to beneficial actions and negative feedback to detrimental ones. [40] integrates LLM-generated subgoal hints into model rollouts, providing intrinsic rewards for goal completion and guiding agents toward meaningful exploration in challenging tasks.

Skill Decomposition and Learning An alternative approach to handling long-horizon tasks involves learning a collection of skills rather than a single monolithic policy. Traditional skill discovery methods [41, 32] focus on identifying useful behavioral primitives through exploration and graph-based representations. [42] develops multi-level skill hierarchies for navigation in maze-like domains, while a large body of work frames skill emergence as maximizing dependence between states and skill labels via mutual information (MI) [43, 44, 45, 46, 47, 48]. However, discriminator-based MI objectives can saturate once a classifier perfectly separates skills, often yielding behaviors that differ only in subtle, non-salient ways [44]. To promote more behaviorally distinct skills, recent methods replace MI with Wasserstein dependency measures [49, 50, 51, 52, 53], pairing the objective with a task-relevant metric (e.g., Euclidean distance in state space [51] or controllability-aware distances that favor rare transitions [52]). The advent of LLMs has further enabled skill decomposition by turning high-level goals into skill definitions with dense rewards and termination conditions [54, 55, 56], or by generating subgoal sequences before training [57]; yet these typically follow a one-shot, feedforward plan that is not refined from interaction.

Learning from Environment Interactions A critical limitation of current LLM-based skill decomposition methods is their reliance on static, one-shot planning that cannot adapt when the initial context provided to the LLM is insufficient for the task at hand. Learning about the environment from interactions becomes essential when the LLM's initial understanding is incomplete or incorrect. LLM self-improvement through environment feedback has shown success in coding [58, 59] and planning agents [17, 60], with structured prompting techniques enabling generate-evaluate-reflect cycles. [60] demonstrates that LLMs can gather and store information about environment dynamics from low-level interactions, though prior work primarily considers LLM-as-agent settings. Our approach, SCALAR, addresses the gap in existing LLM-based skill decomposition methods by introducing bi-directional learning from environmental interactions. Unlike previous approaches that generate static skill decompositions, SCALAR continuously refines its understanding of both skill specifications and environment dynamics through Pivotal Trajectory Analysis, enabling adaptive skill learning that improves with experience.

3 Preliminaries

Our goal is to learn a library of temporally extended *skills* that can be composed to solve long-horizon, partially observable tasks. The agent interacts with an environment while an external proposer (e.g., an LLM) suggests candidate skills and reward specifications; the algorithm must decide *when* a skill may start, *when* it ends, and *what* it achieves. To support these decisions, we introduce a symbolic abstraction of observations into Boolean fluents and define options over this symbolic state. This lets us compute which fluents are currently reachable with the available skills and identify the boundary—the *exploration frontier*—where learning the next skill is most useful. The definitions below establish this formal footing used by our method in later sections.

Tasks as POMDPs. We model tasks as POMDPs $(S, A, T, R, \Omega, O, \gamma)$, where S are states, A actions, T the transition kernel, $R : S \times A \to \mathbb{R}$ rewards, Ω observations, O the observation kernel, and $\gamma \in [0, 1)$ the discount.

Symbolic abstraction. We encode observation histories with $\Phi : \Omega \to \mathcal{Z}$ so that $z_t = \Phi(o_{0:t})$, mapping into symbolic states in alphabet $\mathcal{Z}$. Let $\mathcal{F}$ be the universe of atomic symbols ("fluents"); a

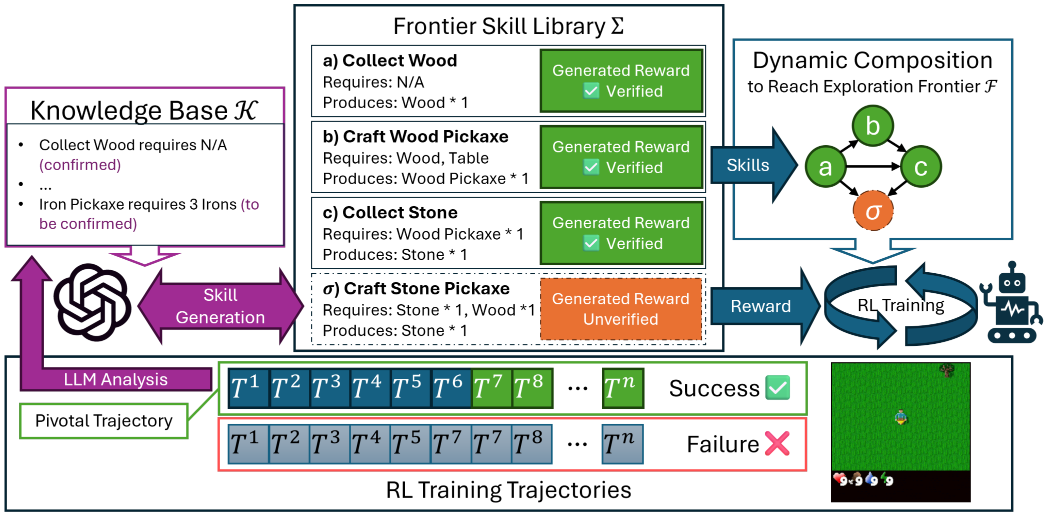

Figure 2: **Method: frontier-guided RL with skill generation and verifier-decoupled rewards.** The LLM proposes candidate skills with preconditions and outputs; rewards for each skill are generated and verified. Skills are dynamically composed to reach the exploration frontier, and the agent performs RL using these rewards, yielding success/failure trajectories. Successful (pivotal) trajectories update the knowledge base and extend the frontier.

labeling map $L : \mathcal{Z} \to 2^{\mathcal{F}}$ returns the fluents true in z, i.e., $L(z) \subseteq \mathcal{F}$. These fluents specify skill initiation, termination, and achievements.

Skills. A *skill* σ is an option $\sigma = (I_\sigma, \pi_\sigma, \beta_\sigma)$ [61] with initiation set $I_\sigma \subseteq S$, policy π_σ, and termination $\beta_\sigma : S \to [0, 1]$; equivalently it terminates in $B_\sigma = \{\, s : \beta_\sigma(s) = 1 \,\}$.

Skills over symbols. We lift initiation and termination to the symbolic layer via fluent predicates $\iota_\sigma, \tau_\sigma : 2^{\mathcal{F}} \to \{0, 1\}$. The *symbolic initiation* and *symbolic termination* regions are

$$I_\sigma^{\mathcal{Z}} = \{\, z \in \mathcal{Z} : \iota_\sigma(L(z)) = 1 \,\}, \qquad B_\sigma^{\mathcal{Z}} = \{\, z \in \mathcal{Z} : \tau_\sigma(L(z)) = 1 \,\}.$$

Execution of σ may start at any $z \in I_\sigma^{\mathcal{Z}}$ and terminates upon first entry into $B_\sigma^{\mathcal{Z}}$.

Reachability and the frontier. Let Σ be the skill library. For each $\sigma \in \Sigma$, fix a conservative *achievement set* $A_\sigma \subseteq \mathcal{F}$ of fluents that hold upon termination, and given known fluents $P \subseteq \mathcal{F}$, define the one-step fluent closure:

$$A_\sigma \subseteq \bigcap_{z \in B_\sigma^{\mathcal{Z}}} L(z), \qquad \mathsf{Close}_\Sigma(P) = P \cup \bigcup_{\sigma \in \Sigma:\ \iota_\sigma(P)=1} A_\sigma.$$

Starting from base fluents $P_0 = L(z)$ of the current symbolic state, the *reachable fluents* form the least fixed point

$$\mathcal{F}_\Sigma^*(P_0) = \lim_{k \to \infty} \mathsf{Close}_\Sigma^{(k)}(P_0).$$

We call $\mathcal{F}_\Sigma^*(P_0)$ the *frontier*. The induced set of frontier symbolic states

$$\mathcal{Z}_\Sigma^*(P_0) = \{\, z \in \mathcal{Z} : L(z) \subseteq \mathcal{F}_\Sigma^*(P_0) \,\}$$

identifies starting points for learning new skills that extend reach beyond the current frontier.

4 SCALAR: Self-Supervised Composition and Learning of Skills

This section introduces **SCALAR**, an iterative procedure that augments the skill library Σ with frontier-expanding skills: the *initiation region* is reachable from $P_0 = L(z)$, and the *termination region* contributes new fluents beyond $\mathcal{F}_\Sigma^*(P_0)$. Concretely, a proposed skill σ_{new} comes with initiation and termination predicates over fluents $(\iota_{\sigma_{\text{new}}}, \tau_{\sigma_{\text{new}}})$. SCALAR forms a closed loop for

LLM-guided skill discovery: an agent explores a game-like world while an LLM analyzes trajectories, hypothesizes new skills with reward functions, verifies those rewards from execution data, and stores confirmed facts in a growing knowledge base $\mathcal{K}$. The LLM then dynamically composes verified skills to push the agent farther to the exploration frontier, creating bi-directional feedback between symbolic planning and RL execution. An overview is given in Algorithm 2 and Figure 2.

Knowledge base and initialization. We initialize $\mathcal{K}$ from plain-text specs of the agent's world: a list of state symbols (e.g., block names, inventory item names and counts) and the discrete actions the agent can take. From these definitions we instantiate a small predicate set over $\mathcal{Z}$ and mechanically compose actions with state predicates to generate *hypothesized* initiation/termination pairs

$$\big(\iota_\sigma,\ \tau_\sigma\big) : 2^{\mathcal{F}} \to \{0,1\},$$

All such predicates/sets are explicitly marked as hypotheses. For example, the knowledge base might initially contain "Iron Pickaxe requires 3 Irons (to be confirmed)" based on general world knowledge, even though the actual requirement in this environment may differ.

Proposing new skills with LLM priors. Guided by $\mathcal{K}$, a language model proposes a new skill $\sigma_{\text{new}} = (\iota_{\sigma_{\text{new}}}, \tau_{\sigma_{\text{new}}}, r_{\mathcal{Z}}, \kappa_{\mathcal{Z}})$ by specifying symbolic initiation and termination along with learning signals, where $r_{\mathcal{Z}} : \mathcal{Z} \times A \times \mathcal{Z} \to \mathbb{R}$ is a proposed shaping/reward and $\kappa_{\mathcal{Z}} : \mathcal{Z} \to \{0,1\}$ is a completion indicator *consistent with* termination, typically $\kappa_{\mathcal{Z}}(z) = \tau_{\sigma_{\text{new}}}(L(z))$. The knowledge base induces priors $(\iota_{\sigma_{\text{new}}}, \tau_{\sigma_{\text{new}}}) \sim \Pi_{\text{pred}}(\cdot \mid \text{prompt}, \mathcal{K})$ and $(r_{\mathcal{Z}}, \kappa_{\mathcal{Z}}) \sim \Pi_{\text{rl}}(\cdot \mid \text{prompt}, \mathcal{K})$. For instance, the LLM might propose "craft iron pickaxe" with initiation requiring 2 wood, 3 iron, and a crafting table (based on Minecraft priors) and termination achieving +1 iron pickaxe.

Pre-policy verification. Before any learning, we test the proposed skill against the current frontier $\mathcal{F}^*_\Sigma(P_0)$:

$$\textbf{(Novelty)}\ \exists\, P \subseteq \mathcal{F} \text{ s.t. } \tau_{\sigma_{\text{new}}}(P) = 1 \text{ and } P \not\subseteq \mathcal{F}^*_\Sigma(P_0), \quad \textbf{(Feasibility)}\ \iota_{\sigma_{\text{new}}}\big(\mathcal{F}^*_\Sigma(P_0)\big) = 1.$$

The first check rules out skills whose termination condition does not imply any fluent beyond what is already reachable by composing existing skills; the second ensures the proposed initiation condition can be satisfied with the current skill library. For instance, if crafting an iron pickaxe is already the gain of another skill, the novelty check would reject the proposal, while if we lack skills to collect wood or iron, the feasibility check would fail.

Training Only proposals passing the above criteria proceed to learning. Let the admissible start set for σ_{new} be

$$\mathcal{S}_{\text{start}}(\sigma_{\text{new}}) \ = \ \big\{\, z \in \mathcal{Z} \ : \ \iota_{\sigma_{\text{new}}}\big(L(z)\big) = 1 \ \text{ and } \ L(z) \subseteq \mathcal{F}^*_\Sigma(P_0) \,\big\}.$$

We sample a start state $z^\star$ uniformly from $\mathcal{S}_{\text{start}}(\sigma_{\text{new}})$ and *first return* to that state using only existing capabilities: if the environment supports resets we initialize at an observation o_0 with $\Phi(o_0) = z^\star$; otherwise we execute a feasible composition of skills from Σ until the encoded state satisfies $\Phi(o_t) = z^\star$. Once $z^\star$ is reached, we *then explore* by collecting experience to learn $\pi_{\sigma_{\text{new}}}$ under the proposed reward $r_{\mathcal{Z}}$ (or members of the ensemble $\mathcal{R}$), while success is determined exclusively by the termination predicate $\tau_{\sigma_{\text{new}}}\big(L(z_t)\big) = 1$ (equivalently, $\kappa_{\mathcal{Z}}(z_t) = 1$). Episodes terminate when $\tau_{\sigma_{\text{new}}}(L(z_t)) = 1$ or after a fixed horizon H. For the iron pickaxe example, we would first navigate to a state with 2 wood, 3 irons, and a crafting table, then explore actions to learn the crafting behavior. This first-return-then-explore regimen isolates exploration in neighborhoods where the initiation condition holds and stabilizes credit assignment by isolating the new behavior from the roll-in. An instantiation with PPO is given in Algorithm 1.

Pivotal Trajectory Analysis From successful rollouts $\sigma_k = (z_0^{(k)}, \ldots, z_{T_k}^{(k)})$ we form fluent trajectories $\mathbf{s}^{(k)} = (L(z_0^{(k)}), \ldots, L(z_{T_k}^{(k)}))$. An LLM analyzes these trajectories to refine the symbolic specification of the skill by tightening its initiation and termination predicates and updating the knowledge base. Concretely, we obtain

$$\big(\widehat{\iota}_{\sigma_{\text{new}}},\, \widehat{\tau}_{\sigma_{\text{new}}},\, \mathcal{K}'\big) \ = \ U_{\text{LLM}}\Big(\mathcal{K};\ \{\mathbf{s}^{(k)}\}_k\Big),$$

where $\widehat{\iota}_{\sigma_{\text{new}}}, \widehat{\tau}_{\sigma_{\text{new}}} : 2^{\mathcal{F}} \to \{0,1\}$ are revised predicates consistent with observed starts and terminations. For instance, successful iron pickaxe trajectories might reveal that only 1 iron (not 3) is actually consumed, leading to updated knowledge base entries and revised initiation conditions. We add the refined skill to Σ with $(\widehat{\iota}_{\sigma_{\text{new}}}, \widehat{\tau}_{\sigma_{\text{new}}})$, update $\mathcal{K} \leftarrow \mathcal{K}'$, and recompute the frontier $\mathcal{F}^*_\Sigma(P_0)$.

Method	Setup (%)		Pickaxes (%)			Goal (%)	Survival			Episodes
Method	Table	Furnace	Wood	Stone	Iron	Diamond	Energy	Food	Drink	Ep. Len.
Baselines										
PPO-RNN	**100.0**	**99.9**	**100.0**	**99.7**	97.8	40.8	7.4	9.1	10.5	284.1
PPO-FC	**100.0**	99.8	**99.9**	99.1	93.2	35.4	12.8	16.0	18.4	515.9
PPO-RND	**99.9**	99.7	**99.9**	98.8	96.7	38.1	**14.5**	**18.4**	**20.9**	**591.7**
ICM	0.0	0.0	0.0	0.0	0.0	0.0	8.6	8.9	9.2	306.7
E3B	4.0	0.0	0.2	0.0	0.0	0.0	2.8	4.3	5.3	140.2
Our Method (Ablations)										
No Trajectory Analysis	99.6	94.2	99.6	99.3	93.7	74.1	13.5	17.1	19.5	558.2
Shared Networks	**99.9**	98.9	**99.9**	**99.8**	**98.5**	57.4	6.8	8.0	9.6	258.8
Our Method										
SCALAR-Dense	**99.9**	98.1	**99.9**	**99.7**	97.8	81.8	**14.5**	**18.7**	**21.2**	**610.3**
SCALAR-Sparse	**99.9**	98.1	**99.9**	**99.7**	97.8	**86.3**	13.5	17.2	20.1	564.1

Table 1: Final performance comparison on Craftax-Classic diamond collection task. Achievement rates reported as percentages; survival metrics (Energy/Food/Drink) show mean intrinsic recovery per episode; episode lengths as raw values.

Mitigating reward hacking. Instead of committing to a single predicted shaping reward for a candidate skill, we *sample* an ensemble

$$\mathcal{R} = \{r_{\mathcal{Z}}^{(j)}\}_{j=1}^{M} \sim \Pi_{\text{rl}}(\cdot \mid \text{prompt}, \mathcal{K})$$

and train a policy $\pi^{(j)}$ under each $r_{\mathcal{Z}}^{(j)}$. Treating $\kappa_{\mathcal{Z}}$ purely as a task-level verifier, we estimate the verified success rate $\hat{s}^{(j)} = \Pr[\kappa_{\mathcal{Z}}(z_T) = 1 \mid \pi^{(j)}]$ from rollouts. We then select $j^\star \in \arg\max_{j \in \{1,\ldots,M\}} \hat{s}^{(j)}$ (optionally restricting to j with $\hat{s}^{(j)} \geq \eta$) and use $\pi^{(j^\star)}$ as the learned controller for the skill. For example, one reward function might incentivize approaching crafting tables, while another rewards inventory changes; however, regardless of reward, only the policy that actually produces an iron pickaxe receives high verified success. By decoupling learning from evaluation, policies that exploit proxy rewards without accomplishing the task receive low verified success and are not selected.

5 Experiments

We evaluate **SCALAR** on Craftax-Classic [34, 62] against strong model-free baselines and targeted ablations, using a JAX implementation of PPO for policy learning and GPT-4.1 as the LLM prior. Across the suite, **SCALAR** matches or exceeds strong model-free baselines on achievements that standard policies already master (e.g., *Table*, *Furnace*, *Wood/Stone/Iron pickaxes*); see Table 1. Consequently, we focus our analysis on the hardest benchmark, *Diamond*, where sparse progress signals and long survival horizons are essential. On this task, SCALAR attains substantially higher success than PPO variants while maintaining competitive upstream achievements (Table 1; learning curves in Fig. 3, left). We report achievement success, survival behaviors (sleep/drink/eat), and episode length.

Defining a Symbolic Encoder The observation space of Craftax-Classic is symbolic: an egocentric $7{\times}9$ local map (blocks/mobs), nearest-block offsets, inventory, and vitals. We take this parsed state as our encoder and treat the current observation as sufficient for the symbol, so that $z_t = \Phi(o_t)$ and $L(z_t) \subseteq \mathcal{F}$.

Skills In SCALAR, each skill σ consists of reward, completion, *requirements*, *consumption*, and *gain* functions, written as $\sigma = \big(r_{\mathcal{Z}},\, \kappa_{\mathcal{Z}},\, \mathsf{req}(\sigma),\, \mathsf{cons}(\sigma),\, \mathsf{gain}(\sigma)\big)$. These are consistent with the formal predicates from Sec. 4: the requirement set encodes *initiation*, while gains/consumption summarize the *effects at termination*:

$$\iota_\sigma(L(z)) = 1 \iff \mathsf{req}(\sigma) \subseteq L(z), \qquad \kappa_{\mathcal{Z}}(z) = \tau_\sigma(L(z)) = 1.$$

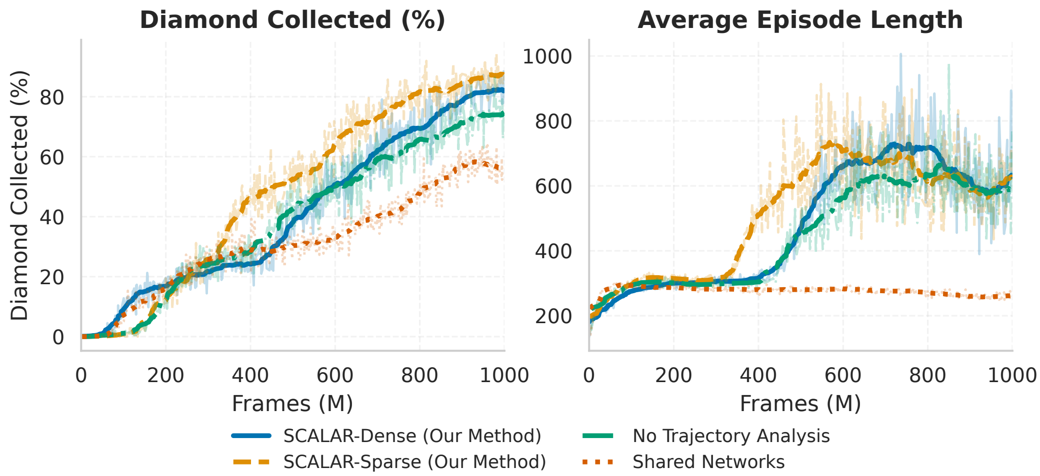

Figure 3: Diamond collection success rate and episode length during SCALAR training. *Left:* percentage of episodes collecting diamond (higher is better). *Right:* mean episode length during training. Bold curves show moving window averaged performance; shaded regions/faint traces show raw per epoch performance. Methods: SCALAR-Dense (blue), SCALAR-Sparse (green), No Trajectory Analysis (orange dashed), Shared Networks (red dotted).

Intuitively, $\mathsf{gain}(\sigma) \subseteq \mathcal{F}$ collects fluents that (typically) hold upon termination, and $\mathsf{cons}(\sigma) \subseteq \mathcal{F}$ captures fluents that are consumed or no longer hold after termination (e.g., spent resources). For instance, craft iron pickaxe might have $\mathsf{req}(\sigma) = \{2\text{ wood}, 1\text{ iron}, \textsc{Near}(\textsc{CraftingTable})\}$, $\mathsf{cons}(\sigma) = \{2\text{ wood}, 1\text{ iron}\}$, and $\mathsf{gain}(\sigma) = \{1\text{ iron pickaxe}\}$. For count-valued fluents (inventory quantities), we parameterize required/consumed/gained *amounts* with simple linear forms $f(n) = an + b$ in the number of executions n. The induced sets are

$$I_\sigma^{\mathcal{Z}} = \{\, z : \mathsf{req}(\sigma) \subseteq L(z) \,\}, \qquad B_\sigma^{\mathcal{Z}} = \{\, z : \tau_\sigma(L(z)) = 1 \,\}.$$

Ephemeral skills. We mark a skill σ as *ephemeral* when its gains are not persistent across time or position (e.g., $\textsc{Near}(\textsc{CraftingTable})$ after walking away). The LLM decides ephemerality from symbolic rollouts. During planning, if an ephemeral skill appears as a prerequisite, we substitute it by its *requirements*: if $\textsc{PlaceCraftingTable}$ requires 4 wood, then $\textsc{CraftPickaxe}$ requires $\{6\text{ wood}, 1\text{ iron}\}$ instead of $\{2\text{ wood}, 1\text{ iron}, \textsc{Near}(\textsc{CraftingTable})\}$. This ensures the frontier reflects persistent capabilities rather than transient intermediates.

Skill Composition and Frontier Approximation Given a library Σ, we compose skills to reach frontier states that satisfy a proposed skill's requirements. Computing the full frontier $\mathcal{F}^*_\Sigma(P_0)$ online is not compatible with JAX compilation, so we approximate it by tracing backwards from the proposed skill's requirements through the dependency graph induced by enumerating each skill's requirements, consumption, and gains. A level-order (BFS) traversal of this graph yields an execution order in which all prerequisites of any skill appear in earlier layers; thus, when the proposed skill is reached, its requirements are met. This produces a tractable subset of reachable fluents sufficient for feasibility/novelty checks and, in practice, matches the states visited by the executed plan.

Training and Evaluation Protocol We train policies with PPO. Episodes initialize the environment state randomly, as in standard RL, but execution follows the layer order from the dependency graph so that, by the time the trajectory reaches the proposed skill, the encountered state distribution satisfies its preconditions. We consider a skill successfully trained when its success rate matches at least some α which in practice we set to 0.8. For a Goal Task, such as collecting diamond, we set $\alpha = 0.99$ and use all remaining budgeted frames once collect diamond skill is reached.

Baselines include PPO-FC/RND/RNN and intrinsic-motivation methods. Ablations remove trajectory analysis, replace sparse rewards with dense shaping, or replace per-skill heads with a single shared network while keeping the same execution order. We report *Diamond* success, survival proxies, and episode length; early sample efficiency is summarized at 100M frames and final performance at the training horizon, with matched budgets for fairness (cf. Fig.3, Fig.4, Table 1).

Knowledge Base / Skill Updates via Pivotal Trajectory Analysis After each skill returns a successful trajectory we run pivotal trajectory analysis that compares the skills inferred prerequisites to what was actually required for the successful trajectory. This information is used to update the prerequisites of the skill and update the knowledge base. For example, if our iron pickaxe skill was initially learned with requirements $\{2\ \text{wood}, 3\ \text{iron}, \textsc{Near}(\textsc{CraftingTable})\}$ but successful trajectories show it only consumes 1 iron, the analysis updates the knowledge base to "Iron Pickaxe requires 1 Iron (confirmed)" and revises future skill proposals accordingly. Prompts for this process are detailed in Appendix E.

5.1 Focused Rewards Enable Survival Learning

A notable qualitative difference is that SCALAR learns to "live forever" in the sense of sustaining long episodes before and during diamond search. Because the environment's native reward does *not* directly incentivize sleeping, eating, or drinking, early learning is dominated by other reward signals. Once the *Diamond* objective is introduced, its sparsity makes the per-timestep health penalty comparatively larger; the agent is thereby driven to master survival subskills so that exploration for diamond can proceed without premature termination. Empirically, mean episode length grows markedly during training (Fig. 3, right), coinciding with the onset of reliable diamond collection.

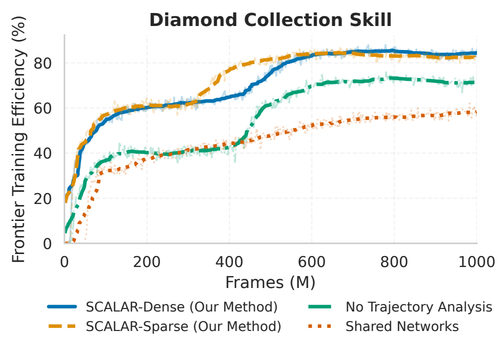

Figure 4: Fraction of training time spent on diamond collection vs. prerequisite skills. Y-axis shows percent of total training frames allocated to the target diamond skill after reaching the iron-pickaxe frontier state. Higher values indicate more efficient utilization of training budget.

This advantage becomes evident when examining survival behaviors conditional on episode outcomes. In failed episodes, SCALAR agents achieve dramatically better survival metrics—20.9 energy, 26.8 food, and 30.2 drink recovery compared to the best baseline (PPO-RND) at only 13.9, 17.6, and 20.1 respectively (Table 5). This difference stems from reward focus: SCALAR's diamond skill receives reward only for diamond collection, making the health penalty relatively significant and driving survival behavior learning. Baseline policies receive rewards for every achievement, diluting the importance of the health penalty and preventing effective survival learning.

5.2 Dense vs. Sparse Reward Trade-offs

We compare training with LLM proposed dense shaping signals against strictly sparse objectives. Dense rewards provide clear advantages for *sample efficiency*—SCALAR-Dense reaches 5.3% diamond collection by 100M frames compared to only 0.3% for SCALAR-Sparse (Table 4), with the learning-curve advantage evident in Fig.3. However, this early advantage comes at a cost: beyond $\sim$300M frames, the sparse-only variant learns survival behaviors more aggressively, achieving higher intrinsic recovery (17.2 energy, 20.1 food, 20.1 drink vs. 14.5, 18.7, 21.2 for dense), and ultimately surpassing dense shaping in final diamond performance (86.3% vs. 81.8%; Table 1).

This trade-off reveals a fundamental tension: dense shaping accelerates initial skill acquisition by providing intermediate feedback signals, but can inadvertently compete with the sparse health penalty that drives long-horizon survival learning. Once diamond collection becomes the primary objective, its sparsity amplifies the relative importance of health maintenance, driving agents to master sleeping, eating, and drinking. Dense rewards may dilute this crucial signal, leading to shorter episodes and reduced final performance despite faster initial progress.

Computational Cost. A key advantage of SCALAR's architecture is that the LLM is not invoked in the RL training loop. LLM calls are only made between training runs for high-level planning: proposing new skills, generating reward/verifier code, and performing pivotal trajectory analysis. This amortizes the cost of LLM inference over millions of environment steps. For all experiments reported, including all baselines and ablations, the total cost of LLM queries was $2.83, corresponding to 1.429M input tokens and 125k output tokens.

5.3 Ablations

Pivotal Trajectory Analysis. We ablate the *trajectory analysis* step that updates domain knowledge from successful rollouts. Without analysis, the LLM prior overestimates material requirements (e.g., predicting that additional wood/stone/iron are needed before each pickaxe), leading to systematic over-collection before attempting diamond. This effect is visible in the resources accumulated *before* success (Table 2)—without trajectory analysis, agents collect 19 wood vs. 9 wood, 11 stone vs. 5 stone, and 3 iron vs. 1 iron before achieving diamond—and manifests as worse *frontier training efficiency*, where the agent spends more time reaching the iron-pickaxe frontier and accrues fewer training frames on the target skill (Fig.4). Quantitatively, at 100M frames the no-analysis variant collects diamonds only 0.8% of the time versus 5.3% for full SCALAR-Dense (Table 4). Over the full budget, final diamond success is also lower (74.1% vs. 81.8% for SCALAR-Dense; Table 1).

Resource	No Traj	Traj
Wood	19	9
Stone	11	5
Coal	1	1
Iron	3	1
Diamond	1	1
Wood pickaxe	1	1
Stone pickaxe	1	1
Iron pickaxe	1	1

Table 2: Resources collected before collecting Diamond with and without trajectory analysis.

Shared networks vs. per-skill networks. Finally, we replace SCALAR's per-skill networks with a single shared network trained on the same curriculum (execution order) discovered by SCALAR. While this removes modularity and thus the ability to reorder skills at test time, it also degrades learning. The shared model exhibits worse frontier efficiency than even the no-analysis ablation (Fig.4), substantially shorter episodes (258.8 average steps), and fails to acquire key survival behaviors (e.g., only 6.8 energy, 8.0 food, and 9.6 drink recovery per episode compared to 14.5, 18.7, and 21.2 for SCALAR-Dense), all while achieving a lower final diamond rate (57.4%; Table 1). These trends are consistent with interference and credit-assignment challenges in the shared representation: coupling all skills into one network entangles execution order with control, hindering both sample efficiency and the emergence of long-horizon survival. *Nevertheless*, even this non-modular variant outperforms hand-engineered reward baselines on *Diamond* (cf. Table 1), indicating that the SCALAR-derived curriculum provides a stronger learning signal than human-designed shaping alone.

6 Conclusion and Future Work

With SCALAR, we propose a novel formal perspective of understanding the synergy between LLM planners and RL agents. SCALAR advances long-horizon control by learning a library of composable skills that can be recombined to meet user-specified goals. Empirically, it (1) achieves state-of-the-art performance on the most challenging Craftax-Classic task, *Diamond*, while matching strong baselines on easier achievements; (2) induces survival behaviors as a *prerequisite* to sparse diamond reward, resulting in substantially longer episodes; (3) yields *controllable* policies—skill compositions execute desired goals with fewer unrelated actions; and (4) *achieves* these results by performing **pivotal trajectory analysis** that writes back experience to the knowledge base, sharpening the learned world model *beyond* the default LLM prior; coupled with **per-skill modularization** for composability, this yields superior sample efficiency, robust survival behavior, and higher final performance.

Limitations and future work. SCALAR has several limitations that point toward future research directions. First, our experiments assume LLM priors with meaningful domain knowledge. Future work should stress-test SCALAR in systematically incorrect prior settings: long-horizon, procedurally generated worlds that violate common-sense rules, augmenting trajectory analysis with counterfactual querying and active knowledge refinement. Second, SCALAR requires high-quality symbolic encoders, whose construction affects proposal filtering and verifier accuracy. Third, per-skill networks improve modularity but increase memory/compute costs; future work should develop composability with shared backbones (e.g., goal-conditioned policies with skill heads) to retain explicit contracts while reducing compute. Additional directions include improving data efficiency in low-sample regimes (<1M frames), scaling to full Craftax with richer hazards, and applying SCALAR to broader domains where symbolic structure is less obvious (e.g., robotics, UI automation).

References

[1] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. Palm: Scaling language modeling with pathways. *arXiv preprint arXiv:2204.02311*, 2022.

[2] OpenAI. Gpt-4 technical report, 2023.

[3] James Manyika. An overview of bard: an early experiment with generative ai. `https://ai.google/static/documents/google-about-bard.pdf`. Accessed: May 27, 2023.

[4] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023.

[5] Albert Q Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, et al. Mistral 7b. *arXiv preprint arXiv:2310.06825*, 2023.

[6] Yuanzhi Li, Sébastien Bubeck, Ronen Eldan, Allie Del Giorno, Suriya Gunasekar, and Yin Tat Lee. Textbooks are all you need ii: phi-1.5 technical report. *arXiv preprint arXiv:2309.05463*, 2023.

[7] Jupinder Parmar, Shrimai Prabhumoye, Joseph Jennings, Mostofa Patwary, Sandeep Subramanian, Dan Su, Chen Zhu, Deepak Narayanan, Aastha Jhunjhunwala, Ayush Dattagupta, Vibhu Jawa, Jiwei Liu, Ameya Mahabaleshwarkar, Osvald Nitski, Annika Brundyn, James Maki, Miguel Martinez, Jiaxuan You, John Kamalu, Patrick LeGresley, Denys Fridman, Jared Casper, Ashwath Aithal, Oleksii Kuchaiev, Mohammad Shoeybi, Jonathan Cohen, and Bryan Catanzaro. Nemotron-4 15b technical report, 2024.

[8] OpenAI. Introducing openai `o1`. `https://openai.com/o1/`. [Accessed 17-04-2025].

[9] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

[10] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *The Eleventh International Conference on Learning Representations*, 2022.

[11] Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems*, volume 36, 2023.

[12] Michael Ahn, Anthony Brohan, Noah Brown, Yevgen Chebotar, Omar Cortes, Byron David, Chelsea Finn, Chuyuan Fu, Keerthana Gopalakrishnan, Karol Hausman, Alex Herzog, Daniel Ho, Jasmine Hsu, Julian Ibarz, Brian Ichter, Alex Irpan, Eric Jang, Rosario Jauregui Ruano, Kyle Jeffrey, Sally Jesmonth, Nikhil J Joshi, Ryan Julian, Dmitry Kalashnikov, Yuheng Kuang, Kuang-Huei Lee, Sergey Levine, Yao Lu, Linda Luu, Carolina Parada, Peter Pastor, Jornell Quiambao, Kanishka Rao, Jarek Rettinghouse, Diego Reyes, Pierre Sermanet, Nicolas Sievers, Clayton Tan, Alexander Toshev, Vincent Vanhoucke, Fei Xia, Ted Xiao, Peng Xu, Sichun Xu, Mengyuan Yan, and Andy Zeng. Do as i can, not as i say: Grounding language in robotic affordances, 2022.

[13] Yuqing Du, Olivia Watkins, Zihan Wang, Cédric Colas, Trevor Darrell, Pieter Abbeel, Abhishek Gupta, and Jacob Andreas. Guiding pretraining in reinforcement learning with large language models. *arXiv preprint arXiv:2302.06692*, 2023.

[14] Zihao Wang, Shaofei Cai, Anji Liu, Xiaojian Ma, and Yitao Liang. Describe, explain, plan and select: Interactive planning with large language models enables open-world multi-task agents. *arXiv preprint arXiv:2302.01560*, 2023.

[15] Yue Wu, So Yeon Min, Yonatan Bisk, Ruslan Salakhutdinov, Amos Azaria, Yuanzhi Li, Tom Mitchell, and Shrimai Prabhumoye. Plan, eliminate, and track–language models are good teachers for embodied agents. *arXiv preprint arXiv:2305.02412*, 2023.

[16] Yue Wu, So Yeon Min, Shrimai Prabhumoye, Yonatan Bisk, Russ R Salakhutdinov, Amos Azaria, Tom M Mitchell, and Yuanzhi Li. Spring: Studying papers and reasoning to play games. In *Advances in Neural Information Processing Systems*, volume 36, 2023.

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

[18] OpenAI. Introducing deep research. `https://openai.com/index/introducing-deep-research/`. [Accessed 17-04-2025].

[19] 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.

[20] Jessy Lin, Yuqing Du, Olivia Watkins, Danijar Hafner, Pieter Abbeel, Dan Klein, and Anca Dragan. Learning to model the world with language. *arXiv preprint arXiv:2308.01399*, 2023.

[21] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pages 1928–1937. PMLR, 2016.

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

[23] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *nature*, 550(7676):354–359, 2017.

[24] Danijar Hafner, Timothy Lillicrap, Mohammad Norouzi, and Jimmy Ba. Mastering atari with discrete world models. *arXiv preprint arXiv:2010.02193*, 2020.

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

[26] Yue Wu, Yewen Fan, Paul Pu Liang, Amos Azaria, Yuanzhi Li, and Tom M Mitchell. Read and reap the rewards: Learning to play atari with the help of instruction manuals. *Advances in Neural Information Processing Systems*, 36, 2024.

[27] 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. *arXiv preprint arXiv:2309.11489*, 2023.

[28] 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. *arXiv preprint arXiv:2310.12931*, 2023.

[29] Shengjie Sun, Runze Liu, Jiafei Lyu, Jing-Wen Yang, Liangpeng Zhang, and Xiu Li. A large language model-driven reward design framework via dynamic feedback for reinforcement learning. *arXiv preprint arXiv:2410.14660*, 2024.

[30] Shaoteng Liu, Haoqi Yuan, Minda Hu, Yanwei Li, Yukang Chen, Shu Liu, Zongqing Lu, and Jiaya Jia. Rl-gpt: Integrating reinforcement learning and code-as-policy. *Advances in Neural Information Processing Systems*, 37:28430–28459, 2024.

[31] Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan, Saiful Haq, Ashutosh Sharma, Thomas T Joshi, Hanna Moazam, et al. Dspy: Compiling declarative language model calls into self-improving pipelines. *arXiv preprint arXiv:2310.03714*, 2023.

[32] Adrien Ecoffet, Joost Huizinga, Joel Lehman, Kenneth O Stanley, and Jeff Clune. Go-explore: a new approach for hard-exploration problems. *arXiv preprint arXiv:1901.10995*, 2019.

[33] Eric Zelikman, Yuhuai Wu, Jesse Mu, and Noah Goodman. Star: Bootstrapping reasoning with reasoning. *Advances in Neural Information Processing Systems*, 35:15476–15488, 2022.

[34] Danijar Hafner. Benchmarking the spectrum of agent capabilities. *arXiv preprint arXiv:2109.06780*, 2021.

[35] Dylan Hadfield-Menell, Smitha Milli, Pieter Abbeel, Stuart J Russell, and Anca Dragan. Inverse reward design. *Advances in neural information processing systems*, 30, 2017.

[36] Victoria Krakovna, Laurent Orseau, Richard Ngo, Miljan Martic, and Shane Legg. Avoiding side effects by considering future tasks, 2020.

[37] Tom Everitt, Marcus Hutter, Ramana Kumar, and Victoria Krakovna. Reward tampering problems and solutions in reinforcement learning: A causal influence diagram perspective, 2021.

[38] Deepak Pathak, Pulkit Agrawal, Alexei A Efros, and Trevor Darrell. Curiosity-driven exploration by self-supervised prediction. In *International conference on machine learning*, pages 2778–2787. PMLR, 2017.

[39] Yongxin Deng, Xihe Qiu, Jue Chen, and Xiaoyu Tan. Reward guidance for reinforcement learning tasks based on large language models: The lmgt framework. *Knowledge-Based Systems*, page 113689, 2025.

[40] Zeyuan Liu, Ziyu Huan, Xiyao Wang, Jiafei Lyu, Jian Tao, Xiu Li, Furong Huang, and Huazhe Xu. World models with hints of large language models for goal achieving, 2024.

[41] Akhil Bagaria, Jason K Senthil, and George Konidaris. Skill discovery for exploration and planning using deep skill graphs. In *International conference on machine learning*, pages 521–531. PMLR, 2021.

[42] Joshua Benjamin Evans and Özgür Şimşek. Creating multi-level skill hierarchies in reinforcement learning. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023.

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

[44] 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*, 2018.

[45] Archit Sharma, Shixiang Gu, Sergey Levine, Vikash Kumar, and Karol Hausman. Dynamics-aware unsupervised discovery of skills. In *International Conference on Learning Representations*, 2019.

[46] Steven Hansen, Will Dabney, Andre Barreto, David Warde-Farley, Tom Van de Wiele, and Volodymyr Mnih. Fast task inference with variational intrinsic successor features. In *International Conference on Learning Representations*, 2019.

[47] Hao Liu and Pieter Abbeel. Aps: Active pretraining with successor features. In *International Conference on Machine Learning*, pages 6736–6747. PMLR, 2021.

[48] Michael Laskin, Hao Liu, Xue Bin Peng, Denis Yarats, Aravind Rajeswaran, and Pieter Abbeel. Unsupervised reinforcement learning with contrastive intrinsic control. In *Advances in Neural Information Processing Systems*, 2022.

[49] Sherjil Ozair, Corey Lynch, Yoshua Bengio, Aaron Van den Oord, Sergey Levine, and Pierre Sermanet. Wasserstein dependency measure for representation learning. *Advances in Neural Information Processing Systems*, 32, 2019.

[50] Shuncheng He, Yuhang Jiang, Hongchang Zhang, Jianzhun Shao, and Xiangyang Ji. Wasserstein unsupervised reinforcement learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 36, pages 6884–6892, 2022.

[51] Seohong Park, Jongwook Choi, Jaekyeom Kim, Honglak Lee, and Gunhee Kim. Lipschitz-constrained unsupervised skill discovery. In *International Conference on Learning Representations*, 2021.

[52] Seohong Park, Kimin Lee, Youngwoon Lee, and Pieter Abbeel. Controllability-aware unsupervised skill discovery. In *International Conference on Machine Learning*, pages 27225–27245. PMLR, 2023.

[53] Seohong Park, Oleh Rybkin, and Sergey Levine. Metra: Scalable unsupervised rl with metric-aware abstraction. In *The Twelfth International Conference on Learning Representations*, 2023.

[54] Shuo Cheng and Danfei Xu. League: Guided skill learning and abstraction for long-horizon manipulation. *IEEE Robotics and Automation Letters*, 8(10):6451–6458, 2023.

[55] Zhaoyi Li, Kelin Yu, Shuo Cheng, and Danfei Xu. League++: Empowering continual robot learning through guided skill acquisition with large language models. In *ICLR 2024 Workshop on Large Language Model (LLM) Agents*, 2024.

[56] Wensen Mao, Wenjie Qiu, Yuanlin Duan, and He Zhu. Skill discovery using language models, 2025.

[57] Chak Lam Shek and Pratap Tokekar. Option discovery using llm-guided semantic hierarchical reinforcement learning, 2025.

[58] Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. Self-refine: Iterative refinement with self-feedback. In A. Oh, T. Neumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine, editors, *Advances in Neural Information Processing Systems*, volume 36, pages 46534–46594. Curran Associates, Inc., 2023.

[59] Yiyang Jin, Kunzhao Xu, Hang Li, Xueting Han, Yanmin Zhou, Cheng Li, and Jing Bai. Reveal: Self-evolving code agents via iterative generation-verification. *arXiv preprint arXiv:2506.11442*, 2025.

[60] Yue Wu, Yewen Fan, So Yeon Min, Shrimai Prabhumoye, Stephen McAleer, Yonatan Bisk, Ruslan Salakhutdinov, Yuanzhi Li, and Tom Mitchell. Agentkit: structured llm reasoning with dynamic graphs. *arXiv preprint arXiv:2404.11483*, 2024.

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

[62] 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. In *International Conference on Machine Learning (ICML)*, 2024.

A Implementation Details

This section provides implementation details for SCALAR (Self-Supervised Composition and Learning of Skills), covering reinforcement learning hyperparameters, skill composition mechanisms, LLM integration, and system architecture decisions.

A.1 Reinforcement Learning Configuration

PPO Hyperparameters We use Proximal Policy Optimization (PPO) as the base RL algorithm with the hyperparameters specified in Table 3.

Hyperparameter	Value
Learning rate	2×10^{-4} (with linear annealing)
Discount factor (γ)	0.99
GAE lambda (λ)	0.8
Clip coefficient	0.2
Entropy coefficient	0.01
Value function coefficient	0.5
Maximum gradient norm	1.0
Update epochs per iteration	4
Number of minibatches	8
Steps per environment rollout	64
Parallel environments	1024
Total training timesteps	10^9
Network layer size	512
Activation function	Tanh

Table 3: PPO hyperparameters used in SCALAR training.

Mixture-of-Experts Architecture SCALAR employs a mixture-of-experts (MoE) approach where each skill maintains separate actor and critic networks rather than sharing parameters. The implementation creates dedicated network instances for each task, with the active network selected based on the current player state through a mapping function `map_player_state_to_skill`. This architecture enables independent learning of different skills without interference, as each expert can specialize in its assigned task without catastrophic forgetting from other skills. The modular design further facilitates compositional reuse of learned behaviors, allowing trained skills to be invoked as building blocks for more complex tasks. Additionally, this separation provides task-specific credit assignment, ensuring that learning signals for each skill are not diluted by conflicting objectives from other tasks.

A.2 Skill Composition and Frontier Implementation

Dependency Graph Construction Skill composition is implemented through a dependency resolution system that constructs a directed acyclic graph (DAG) where nodes represent skills and edges represent prerequisite relationships. The system first preprocesses skills to inline ephemeral requirements—those that do not persist across time or location—directly into their dependent skills. The graph construction process recursively builds dependency chains while tracking visited nodes to prevent cycles. Once constructed, the system prunes unnecessary tool productions by identifying and removing redundant intermediate steps, such as avoiding multiple crafting table placements when one suffices. A level-order traversal determines the execution sequence for just-in-time resource collection while respecting dependency constraints.

Frontier Approximation Computing the exact frontier $\mathcal{F}^*_\Sigma(P_0)$ is computationally expensive and incompatible with JAX compilation due to dynamic graph operations. Instead, we approximate the reachable fluent set by tracing backwards from proposed skill requirements through the dependency graph using breadth-first search to identify achievable fluent combinations. The system employs caching mechanisms to store computation results and avoid redundant graph traversals when evaluating multiple candidate skills. During skill admission, the approximated frontier enables validation of both feasibility constraints—ensuring that proposed skill preconditions can be satisfied with current capabilities—and novelty constraints—verifying that the skill's termination conditions extend the agent's reachable state space.

Inventory Constraints The system enforces Craftax's inventory limits (maximum 9 items per type) through a resource management strategy that tracks current inventory levels throughout skill execution. When a collection skill would exceed the capacity limit, the system splits the task into smaller chunks that respect the constraint while maintaining task completion. The algorithm schedules intermediate consumption of excess resources by identifying skills that consume specific items and inserting them at appropriate points in the execution sequence. The system also adjusts the execution order to minimize inventory pressure by deferring collections until items are needed and inserting deferred collection nodes when capacity becomes available after consumption events.

A.3 LLM Integration and Prompt Engineering

Model Configuration We use GPT-4.1 as the planning LLM with a maximum generation limit of 4096 tokens per query. The model operates with default temperature settings that vary by prompt type to balance creativity and consistency depending on the reasoning task. Each query context includes environment specifications (BlockType, Action, and Achievement enumerations), the current knowledge base state, and descriptions of existing skills. The system employs structured output formats using JSON schemas for skill specifications to ensure consistent parsing and integration with the RL training pipeline. Multi-step reasoning is facilitated through chain-of-thought prompting that guides the LLM through problem decomposition and solution construction.

Prompt Structure Our prompting system employs specialized templates for different phases of the skill discovery process. The **Skill Proposal** template provides context including environment constants (BlockType, Action, Achievement enumerations), the current knowledge base with confirmed and hypothetical facts, and specifications of existing skills to inform the LLM's decision-making process. The **Code Generation** template focuses on producing syntactically correct reward and completion functions with proper JAX compatibility, including necessary imports and function signatures that integrate with the training pipeline. **Trajectory Analysis** prompts guide the LLM through examination of successful rollouts to identify discrepancies between predicted and actual skill requirements, enabling refinement of skill specifications. **Knowledge Base Update** templates facilitate the incorporation of verified facts and the removal of outdated hypotheses based on empirical evidence from RL execution.

Code Validation Generated code undergoes multiple validation steps to ensure correctness and compatibility. The system first performs syntax verification using Python AST parsing to catch basic syntactic errors before execution. JAX compatibility checking follows, testing the generated functions under JIT compilation to identify any operations incompatible with JAX's functional programming constraints. Function signature validation ensures that generated reward and completion functions conform to expected interfaces, including proper parameter names and return types required by the training pipeline. The system also conducts logical consistency checks by executing the functions with sample environment states to verify that they produce reasonable outputs and handle edge cases appropriately, with failed validations triggering code regeneration.

A.4 Training Protocol and Evaluation

Skill Training Process Each skill follows a standardized training protocol that begins with **pre-training validation** to check feasibility and novelty constraints before committing computational resources. The system then performs **frontier navigation** by executing the dependency chain to reach the skill's preconditions, ensuring that the agent starts training from appropriate initial states. **Policy learning** follows, training the skill-specific policy head using the proposed reward functions while maintaining separation from other skills through the MoE architecture. **Success evaluation** measures performance using verifier functions that assess task completion independent of the reward signal used for training. Finally, **trajectory analysis** examines successful rollouts to identify discrepancies between predicted and actual requirements, enabling refinement of skill specifications and knowledge base updates.

Success Thresholds We use different success criteria for different skill types based on their role in the overall task hierarchy. Intermediate skills require an 80% success rate threshold to be considered adequately learned, providing a balance between training efficiency and reliability when these skills are composed into longer chains. Goal tasks such as diamond collection are set to a higher 99% success rate threshold to consume the remaining training budget. Survival skills emerge through health penalty optimization rather than explicit thresholds, as the agent naturally learns these behaviors when the health penalty becomes significant relative to sparse task rewards during long-horizon episodes.

Reward Ensemble Strategy To mitigate reward hacking, we employ an ensemble approach that generates multiple reward functions for each proposed skill, capturing different aspects of the desired behavior or alternative formulations of the same objective. The system trains separate policies under each reward variant, allowing different learning dynamics and exploration strategies to emerge from the various reward signals. All trained policies are then evaluated using task-specific verifiers that measure actual task completion independent of the reward functions used during training. Finally, we select the policy with the highest verified success rate, ensuring that the chosen behavior achieves the intended objective rather than exploiting weaknesses in any individual reward formulation.

A.5 System Architecture

Modular Design The codebase is organized into several key modules that separate concerns and enable maintainable development. The `flowrl.ppo_flow` module contains the extended PPO implementation with MoE support, handling the core reinforcement learning training loop and skill-specific policy management. `flowrl.llm.flow` serves as the main orchestration class that manages the interaction between the LLM planning system and RL training components, coordinating skill proposal, validation, and learning cycles. The `flowrl.skill_dependency_resolver` module implements graph-based skill composition, handling dependency resolution, execution ordering, and inventory constraint management. Finally, `flowrl.llm.craftax_classic` contains environment-specific prompts and code generation utilities tailored to the Craftax domain.

Checkpointing System SCALAR includes a checkpointing mechanism that saves skill library state after each successful skill acquisition, preserving both the learned policies and their symbolic specifications. The system stores the refined knowledge base with confirmed facts and updated hypotheses, enabling the LLM to build upon verified domain knowledge in future iterations. This design enables resumption from arbitrary training stages without loss of accumulated learning, supporting long-running experiments that may span multiple days or weeks. The implementation maintains backward compatibility with previous checkpoint formats to ensure robustness against system updates and facilitate reproducibility of earlier experiments.

JAX Optimization Performance optimizations leverage JAX's compilation and parallelization capabilities to achieve efficient large-scale training. JIT compilation is applied to all training loops and environment steps, eliminating Python interpreter overhead and enabling optimized execution on both CPU and GPU hardware. Persistent compilation caching reduces startup overhead by storing compiled functions across runs, particularly beneficial for iterative experiments that restart frequently during skill discovery. Vectorized environment execution operates across 1024 parallel instances, maximizing throughput and sample collection efficiency. The system employs optimized memory layouts for large-scale trajectory storage, minimizing memory allocation overhead and enabling efficient batch processing of experience data.

A.6 Environment Integration

Craftax-Classic Interface We interface with Craftax-Classic through several key mechanisms that enable SCALAR's skill-based approach. Symbolic observation encoding directly maps the environment's structured observations to fluent sets, providing the symbolic representations required for LLM reasoning and frontier computation. Custom reward function injection allows skill-specific training by dynamically replacing the environment's default reward with LLM-generated reward functions tailored to individual skills. Episode termination control enables skill completion detection by monitoring verifier functions and terminating episodes when skills are successfully completed or predetermined time limits are reached. Achievement tracking provides progress monitoring by maintaining records of completed tasks and environmental milestones that inform both the LLM's planning decisions and the evaluation of skill success rates.

State Encoding The symbolic encoder maps Craftax observations to fluent sets through a structured transformation process. Inventory quantities are extracted as count-valued fluents, representing the number of each item type currently possessed by the agent and forming the basis for skill prerequisite checking. Spatial information including nearby blocks and distances are encoded as positional fluents that capture the agent's local environment and proximity to relevant resources or structures. Achievements are represented as binary fluents that track completed milestones and unlock access to new skills or capabilities. Player vitals such as health, food, drink, and energy levels are maintained as continuous fluents that influence survival behavior and long-term planning considerations.

Method	Setup (%)		Pickaxes (%)			Goal (%)
	Table	Furnace	Wood	Stone	Iron	Diamond
Baselines						
PPO-RNN	100.0	99.0	99.5	95.2	54.5	0.9
PPO-FC	100.0	97.7	100.0	93.0	49.1	1.2
E3B	5.1	0.0	0.0	0.0	0.0	0.0
ICM	0.0	0.0	0.0	0.0	0.0	0.0
PPO-RND	100.0	98.4	99.7	90.6	21.8	0.3
Our Method						
SCALAR-Dense	99.8	84.6	99.8	97.6	76.8	5.3
SCALAR-Sparse	99.8	85.2	99.8	97.8	76.1	0.3
Ablations						
No Trajectory Analysis	99.1	73.4	99.1	96.7	69.4	0.8
Shared Networks	99.9	79.5	99.4	96.6	53.8	2.0

Table 4: Baselines (top); Our Method (mid); and ablations (bottom), all evaluated at 100M frames. Baselines are aggregated using the same timesteps as the reference ablation run (within the 100M cutoff) to ensure an equal training horizon. Values are means over 1000 trajectories; achievements reported as percentages.

Method	Success Episodes				Failure Episodes			
Method	Energy	Food	Drink	Length	Energy	Food	Drink	Length
Baselines								
PPO-RNN	7.4	9.1	10.4	285.0	7.4	9.1	10.5	283.4
PPO-FC	13.1	16.3	18.6	526.1	12.7	15.9	18.3	510.3
PPO-RND	**15.4**	**19.6**	**22.1**	**632.8**	13.9	17.6	20.1	566.4
ICM	—	—	—	—	8.6	8.9	9.2	306.7
E3B	—	—	—	—	2.8	4.3	5.3	140.2
Our Methods								
SCALAR-Dense (Our Method)	13.0	16.9	19.2	543.6	**20.9**	**26.8**	**30.2**	**909.4**
SCALAR-Sparse (Our Method)	12.4	15.9	18.6	513.7	**20.4**	**25.7**	**29.7**	**881.7**
No Trajectory Analysis	13.2	16.9	19.3	542.9	14.4	17.7	20.2	601.9
Shared Networks	5.3	6.2	7.6	185.4	8.9	10.4	12.4	357.8

Table 5: Conditional survival metrics from local experiments. Values show mean metrics for episodes that succeeded (collected diamond) vs failed.

B Craftax-100M Ablation

C Conditional Distribution on Success

D Algorithm Definitions

Algorithm 1: TrainSkill (on-policy PPO; first-return then explore)

Input: Skill predicates $(\iota_\sigma, \tau_\sigma)$, reward $r_{\mathcal{Z}}$, encoder Φ, labeler L, library Σ, frontier $F{=}\mathcal{F}^*_\Sigma(P_0)$, horizon H, total step budget B, PPO hyperparameters $(\gamma, \lambda, \epsilon, K, M, c_{\text{vf}}, c_{\text{ent}})$, policy π_θ, value V_ψ

Output: Trained π_θ, success rate $\hat{s}$

$\mathcal{S}_{\text{start}} \leftarrow \{\, z \in \mathcal{Z} \mid \iota_\sigma(L(z)){=}1 \wedge L(z) \subseteq F \,\}$;
if $\mathcal{S}_{start} = \emptyset$ **then**
    return $(\pi_\theta, 0)$
$S \leftarrow 0,\ E \leftarrow 0,\ J \leftarrow 0$;
while $J < B$ **do**
    Sample $z^\star \sim \mathrm{Unif}(\mathcal{S}_{\text{start}})$;
    `// First return to` $z^\star$ `using only existing skills`
    if *env can reset to* $z^\star$ **then**
        reset to o_0 with $\Phi(o_0) = z^\star$
    else
        compose skills in Σ until current $z = z^\star$
    $\mathcal{D} \leftarrow \emptyset$;;
    $z \leftarrow z^\star$;;
    $t \leftarrow 0$;;
    done $\leftarrow$ **false**;
    `// On-policy rollout (no replay reuse)`
    while $t < H$ ***and*** $\neg$*done* ***and*** $J < B$ **do**
        sample $a \sim \pi_\theta(\cdot \mid z)$; execute a; observe o'; $z' \leftarrow \Phi(o')$;
        $r \leftarrow r_{\mathcal{Z}}(z, a, z')$;;
        done $\leftarrow [\, \tau_\sigma(L(z')){=}1 \,]$;;
        store $(z, a, r, z', \textit{done}, \log \pi_\theta(a|z))$ in $\mathcal{D}$;
        $z \leftarrow z'$;;
        $t \leftarrow t{+}1$;;
        $J \leftarrow J{+}1$;
    $E \leftarrow E{+}1$;;
    if *done* **then**
        $S \leftarrow S{+}1$
    `// PPO advantage/return computation (GAE)`
    compute $\hat{V}_i \leftarrow V_\psi(z_i)$ for all $(z_i, \cdot) \in \mathcal{D}$;;
    $\delta_i \leftarrow r_i + \gamma(1{-}\textit{done}_i)\hat{V}_{i+1} - \hat{V}_i$;;
    $\hat{A}_i \leftarrow \sum_{j \geq i} (\gamma\lambda)^{j-i}\, \delta_j$;;
    $\hat{R}_i \leftarrow \hat{A}_i + \hat{V}_i$;
    `// PPO updates with clipping`
    for $k = 1$ **to** K **do**
        split $\mathcal{D}$ into minibatches of size M;
        foreach *minibatch* $\mathcal{B} \subset \mathcal{D}$ **do**
            compute $\rho_i \leftarrow \exp(\log \pi_\theta(a_i|z_i) - \log \pi_\theta^{\text{old}}(a_i|z_i))$ for $i \in \mathcal{B}$;
            $L^{\text{clip}} \leftarrow \frac{1}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \min\big(\rho_i \hat{A}_i,\ \mathrm{clip}(\rho_i, 1{-}\epsilon, 1{+}\epsilon)\hat{A}_i\big)$;
            $L^{\text{vf}} \leftarrow \frac{1}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} (\hat{R}_i - V_\psi(z_i))^2, \quad L^{\text{ent}} \leftarrow \frac{1}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \mathcal{H}\big(\pi_\theta(\cdot|z_i)\big)$;
            maximize $L^{\text{clip}} + c_{\text{ent}} L^{\text{ent}} - c_{\text{vf}} L^{\text{vf}}$ w.r.t. (θ, ψ);
    set $\log \pi_\theta^{\text{old}}(\cdot|\cdot) \leftarrow \log \pi_\theta(\cdot|\cdot)$; `// refresh on-policy baseline`
$\hat{s} \leftarrow S / \max(1, E)$;;
return $(\pi_\theta, \hat{s})$;

Algorithm 2: SCALAR

Input: $\mathcal{K}$ (KB), Σ (skills), P_0, priors $(\pi_{\text{req}}, \pi_{\text{gain}}, \Pi_{\text{rl}})$, verifier $\kappa_{\mathcal{Z}}$, rewards $\mathcal{R}$, threshold η

repeat

  Propose: Sample $(\mathsf{req}, \mathsf{gain})$ and $(r_{\mathcal{Z}}, \kappa_{\mathcal{Z}})$ from priors.

  Verify: Reject if not novel (**??**) or not feasible (**??**).

  Start state: Pick $z^\star$ where req holds within $\mathcal{F}^*_\Sigma(P_0)$.

  First return: Reset/compose with Σ until $\phi(o) = z^\star$.

  Train & prune: For each $r \in \mathcal{R}$ learn π, estimate verified success $\hat{s}$ with $\kappa_{\mathcal{Z}}$; keep $\{r : \hat{s} \geq \eta\}$ and select $r^\star$.

  Analyze: From successful traces infer $(\widehat{\mathsf{req}}, \widehat{\mathsf{gain}}, \mathcal{K}')$.

  Update: Add $(r^\star, \kappa_{\mathcal{Z}}, \widehat{\mathsf{req}}, \widehat{\mathsf{gain}})$ to Σ; set $\mathcal{K} \leftarrow \mathcal{K}' \cup \{\widehat{\mathsf{req}} \Rightarrow \widehat{\mathsf{gain}}\}$; recompute $\mathcal{F}^*_\Sigma(P_0)$.

until *budget exhausted or converged*

E Prompt Details

E.1 Skill Proposal and Instantiation (Prompts & Procedure)

We use a four-stage, LLM-guided pipeline to propose and instantiate new skills that expand the reachable frontier while staying compatible with existing skills and the knowledge base.

Overview. Given the current knowledge base (KB) and library of learned skills, we (i) select a *new* next skill whose prerequisites are already satisfiable by existing skills, (ii) formalize its requirements/consumption/gain using lambda forms in terms of existing skill gains, (iii) design anti-gaming (sparse) and shaped (dense) reward signals, and (iv) generate executable code for success checking and rewards. Each stage is driven by a dedicated prompt; we include all raw prompts below.

Procedure.

1. **Choose the next skill (`next_task`).** Review the KB and existing skills; list candidate objectives and select one that is new (not in the library) and feasible with current abilities. Output a compact JSON summary (name, one-line description, target gain).
2. **Specify requirements/consumption/gain (`next_subtask`).** Express requirements and consumption as Python lambda strings of the form `"lambda n:  a*n + b"` keyed strictly by names that appear as gains of existing skills; mark whether the skill is *ephemeral*.
3. **Design rewards with hacking analysis (`create_skill_densify_reward_reasoning`).** Analyze per-timestep factors, rule out reward-cycling exploits, and produce a minimal sparse reward plus (optional) dense terms with small coefficients that cannot dominate the sparse signal.
4. **Emit code stubs (`create_skill_coding`).** Generate JAX-compilable implementations of `task_is_done`, `task_reward`, and `task_network_number` that follow the environment contracts and avoid stateful assumptions.

Raw prompt: `next_task`.

```

Environment Details:
@struct.dataclass
class Inventory:
    wood: int = 0
    stone: int = 0
    coal: int = 0
    iron: int = 0
    diamond: int = 0
    sapling: int = 0
    wood_pickaxe: int = 0
    stone_pickaxe: int = 0
    iron_pickaxe: int = 0
```

```
    wood_sword: int = 0
    stone_sword: int = 0
    iron_sword: int = 0
#max inventory size is 9 for each item

# ENUMS
class BlockType(Enum):
    INVALID = 0
    OUT_OF_BOUNDS = 1
    GRASS = 2
    WATER = 3
    STONE = 4
    TREE = 5
    WOOD = 6
    PATH = 7
    COAL = 8
    IRON = 9
    DIAMOND = 10
    CRAFTING_TABLE = 11
    FURNACE = 12
    SAND = 13
    LAVA = 14
    PLANT = 15
    RIPE_PLANT = 16

class Action(Enum):
    NOOP = 0  #
    LEFT = 1  # a
    RIGHT = 2  # d
    UP = 3  # w
    DOWN = 4  # s
    DO = 5  # space
    SLEEP = 6  # tab
    PLACE_STONE = 7  # r
    PLACE_TABLE = 8  # t
    PLACE_FURNACE = 9  # f
    PLACE_PLANT = 10  # p
    MAKE_WOOD_PICKAXE = 11  # 1
    MAKE_STONE_PICKAXE = 12  # 2
    MAKE_IRON_PICKAXE = 13  # 3
    MAKE_WOOD_SWORD = 14  # 4
    MAKE_STONE_SWORD = 15  # 5
    MAKE_IRON_SWORD = 16  # 6

class Achievement(Enum):
    COLLECT_WOOD = 0
    PLACE_TABLE = 1
    EAT_COW = 2
    COLLECT_SAPLING = 3
    COLLECT_DRINK = 4
    MAKE_WOOD_PICKAXE = 5
    MAKE_WOOD_SWORD = 6
    PLACE_PLANT = 7
    DEFEAT_ZOMBIE = 8
    COLLECT_STONE = 9
    PLACE_STONE = 10
    EAT_PLANT = 11
    DEFEAT_SKELETON = 12
    MAKE_STONE_PICKAXE = 13
```

```
    MAKE_STONE_SWORD = 14
    WAKE_UP = 15
    PLACE_FURNACE = 16
    COLLECT_COAL = 17
    COLLECT_IRON = 18
    COLLECT_DIAMOND = 19
    MAKE_IRON_PICKAXE = 20
    MAKE_IRON_SWORD = 21

Knowledgebase:

$db.knowledge_base$

Existing Skills:

$db.skills_without_code$

# Instruction
Consider the knowledgebase, and existing skills. Identify the next skill that should be learned.
Fill out the following sections explicitly before arriving at the final formatted output.

## Review Existing Skills
In a few sentences, review existing skills.

## Future Objectives
List up to 3 potential future objectives that the player could work toward next. For each objecti

## Immediate Objective
Identify the next skill the player should learn based on your analysis. CRITICAL: Do NOT propose


# Formatting
Finally, complete the following Json dictionary as your output.
```json
{
"skill_name": # name of the objective
"description": # (string) 1-line description of the objective
"gain": # (str) what the player will gain after applying the skill.
}
```

Raw prompt: next$_s ubtask$.

```
Consider the Knowledgebase and existing skills.

Knowledgebase:

$db.knowledge_base$

Existing Skills

$db.skills$

Skill to Learn

$db.current.skill$

Instruction
```
```

Analyze Knowledgebase

Identify and draw connections between the skill to learn and any relevant existing knowledge.

Task Analysis

Explicitly analyze the current skill:
- What is the core objective?
- What are the specific requirements?
- What resources are consumed when applying the skill.

Previous Skill Analysis

In a bulleted list, write what each skill gains. The requirements and consumption dictionaries fo

Ephemeral Analysis

Determine if this skill is ephemeral. A skill is ephemeral if the gain itself is not observable i

Note
- Distance/adjaceny CANNOT be directly verified or quantified, but you can use the closest block
- Skills should be explicit and complete on their own, and should convey a clear quantifiable go
- Requirements are a SUPERSET of consumption: requirements include everything needed (both consu
- Each value in requirements/consumption should be written as a Python lambda function string th
- a = amount of resource consumed PER unit of gain (scales with n)
- b = amount of resource required but NOT consumed (fixed amount regardless of n)
- Ask yourself: "Does this requirement scale with the number of times I apply the skill?"
- If YES (scales with n): use "lambda n: a*n + 0" format
- If NO (fixed amount): use "lambda n: 0*n + b" format
- Requirements do not support 'or'
- Each key in requirements/consumption must be a key in the gain of an existing skill.

Formatting

Finally, complete the following Json dictionary as your output.

```
{
"skill_name": , # name of the current skill
"requirements": , # (dict) total amount needed available using "lambda n: a*n + b" format. Each k
"consumption": , # (dict) amount consumed using "lambda n: a*n + b" format. Each key must exactly
"gain": , # (dict) a dictionary of what is gained by applying the skill. The gain for the skill g
"ephemeral": , # (bool) true if the gain itself is not observable in the inventory, false if the
}
```

Raw prompt: create$_{s}kill_{d}ensify_{r}eward_{r}easoning.$

All factors

Environment definitions:

```
class BlockType(Enum):
    INVALID = 0
    OUT_OF_BOUNDS = 1
    GRASS = 2
    WATER = 3
    STONE = 4
    TREE = 5
```

```
    WOOD = 6
    PATH = 7
    COAL = 8
    IRON = 9
    DIAMOND = 10
    CRAFTING_TABLE = 11
    FURNACE = 12
    SAND = 13
    LAVA = 14
    PLANT = 15
    RIPE_PLANT = 16
# Max inventory value is 9, max player intrinsics values are also 9
@struct.dataclass
class Inventory:
    wood: int = 0
    stone: int = 0
    coal: int = 0
    iron: int = 0
    diamond: int = 0
    sapling: int = 0
    wood_pickaxe: int = 0
    stone_pickaxe: int = 0
    iron_pickaxe: int = 0
    wood_sword: int = 0
    stone_sword: int = 0
    iron_sword: int = 0

class Achievement(Enum):
    COLLECT_WOOD = 0
    PLACE_TABLE = 1
    EAT_COW = 2
    COLLECT_SAPLING = 3
    COLLECT_DRINK = 4
    MAKE_WOOD_PICKAXE = 5
    MAKE_WOOD_SWORD = 6
    PLACE_PLANT = 7
    DEFEAT_ZOMBIE = 8
    COLLECT_STONE = 9
    PLACE_STONE = 10
    EAT_PLANT = 11
    DEFEAT_SKELETON = 12
    MAKE_STONE_PICKAXE = 13
    MAKE_STONE_SWORD = 14
    WAKE_UP = 15
    PLACE_FURNACE = 16
    COLLECT_COAL = 17
    COLLECT_IRON = 18
    COLLECT_DIAMOND = 19
    MAKE_IRON_PICKAXE = 20
    MAKE_IRON_SWORD = 21
```

The reward function is calculated independently at each timestep using these available factors:

- inventory_diff (Inventory): The change in the player's inventory between the current and previ
- closest_bocks_changes (numpy.ndarray): The changes in distance to closest blocks of each type
- player_intrinsics (jnp.ndarray): The intrinsic values
- player_intrinsics_diff (jnp.ndarray): The changes in current intrinsic values from the last ti

Other Information

- This reward function is called independently at each timestep

```
• Each timestep's reward is calculated using only information from the current and previous time
• The reward at timestep t cannot access information from timestep t-2 or earlier
• The completion criteria is a separate function; do not worry about implementing it
• No state can be stored between timesteps - each reward calculation must be independent

Skill

Given the following skill, design the reward function for the Skill $db.current.skill_name$

$db.current.skill_with_consumption$

Steps

Explicitly complete the following steps before arriving at your final formatted output
0. Analyze Skill Gains and identify appropriate reward factors:
 • What is the core objective of this subtask?
 • What specific behaviors or outcomes need to be rewarded?
 • For each available factor, determine if it can provide meaningful feedback for the required be
 • Remove any factors that are irrelevant to the subtask objectives or should not be used.
 • Assume all requirements for the skill has been met before the skill is applied.
List the remaining factors that will be analyzed in subsequent steps.

 1. Analyze each factor's per-timestep behavior, responding to each question explicitly:
 • How does the raw factor behave at each individual timestep?
 • What does a positive vs negative value mean at a single timestep?
 • What is measured when we use this raw factor as a direct reward?
 • Write out a sequence of timestep values for a potential reward hacking attempt. Sum these valu
 • Based on the sequence sum: Does the reward cycling result in positive net reward? If so, state
 • Write the exact transformation: If we concluded the raw factor naturally prevents reward hacki
 2. Filter out factors with no obvious non-hackable reward functions or those that are not releva
 3. Classify the remaining factors in to dense and sparse rewards. The chosen sparse reward shoul
 4. Design a minimalistic sparse reward formula:
 • Use the raw factor directly if it was shown to naturally prevent reward hacking
 • Include only the minimum operations needed for the reward signal
 • Verify the formula matches your timestep sequence analysis from step 1
 5. Design a dense reward formula:
 • For each factor proven safe in step 1, include it directly
 • If multiple factors are valid, combine them through simple addition
 • No additional transformations beyond what was proven necessary in step 1
 • For each factor included, include a coefficient between 0.0 and 1.0 such that the the magnitud
 • The sum of the sparse reward across timesteps should be greater then the sum of the dense rewa
 • Write ''NA'' if no dense reward is needed
 6. Write both rewards into mathematical formula, and double-check for redundancy

Note
 • The optimization stops when completion critiera is met, so no more rewards will be provided af

If no dense reward function is possible or needed for this task, simply state NA.

{
"sparse_reward_only_function": # (str) Minimal reward pseudocode
"dense_reward_function": # (str) Dense reward pseudocode, "NA" if not available
}
```

Raw prompt: create$_{s}kill_{c}oding.$

```
class BlockType(Enum):
```

```
    INVALID = 0
    OUT_OF_BOUNDS = 1
    GRASS = 2
    WATER = 3
    STONE = 4
    TREE = 5
    WOOD = 6
    PATH = 7
    COAL = 8
    IRON = 9
    DIAMOND = 10
    CRAFTING_TABLE = 11
    FURNACE = 12
    SAND = 13
    LAVA = 14
    PLANT = 15
    RIPE_PLANT = 16
# Max inventory value is 9, max player intrinsics values are also 9
@struct.dataclass
class Inventory:
    wood: int = 0
    stone: int = 0
    coal: int = 0
    iron: int = 0
    diamond: int = 0
    sapling: int = 0
    wood_pickaxe: int = 0
    stone_pickaxe: int = 0
    iron_pickaxe: int = 0
    wood_sword: int = 0
    stone_sword: int = 0
    iron_sword: int = 0

class Achievement(Enum):
    COLLECT_WOOD = 0
    PLACE_TABLE = 1
    EAT_COW = 2
    COLLECT_SAPLING = 3
    COLLECT_DRINK = 4
    MAKE_WOOD_PICKAXE = 5
    MAKE_WOOD_SWORD = 6
    PLACE_PLANT = 7
    DEFEAT_ZOMBIE = 8
    COLLECT_STONE = 9
    PLACE_STONE = 10
    EAT_PLANT = 11
    DEFEAT_SKELETON = 12
    MAKE_STONE_PICKAXE = 13
    MAKE_STONE_SWORD = 14
    WAKE_UP = 15
    PLACE_FURNACE = 16
    COLLECT_COAL = 17
    COLLECT_IRON = 18
    COLLECT_DIAMOND = 19
    MAKE_IRON_PICKAXE = 20
    MAKE_IRON_SWORD = 21

#when indexing an enum make sure to use .value
```

```
#Here are example docstrings:

def task_is_done(inventory, inventory_diff, closest_blocks, closest_blocks_prev, player_intrinsic
    \"\"\"
    Determines whether Task ‘$db.current.skill_name$‘ is complete.
    Do not call external functions or make any assumptions beyond the information given to you.

    Args:
        inventory (Inventory): The player’s current inventory, defined in the above struct
        inventory_diff (Inventory): The change in the player’s inventory between the current and
        closest_blocks (numpy.ndarray): A 3D tensor of shape (len(BlockType), 2, K) representing
        #default of 30,30 if less then k seen, ordered by distance (so :,:,0 would be the closest
        # to get the l2 distance of the agent from the closest diamond for example would be jnp.l
        closest_blocks_prev (numpy.ndarray): A 3D array of shape (len(BlockType), 2, K) represent
        #default of 30,30 if less then k seen, ordered by distance (so :,:,0 would be the closest
        player_intrinsics (jnp.ndarray): An len 4 array representing the player’s health, food, d
        player_intrinsics_diff (jnp.ndarray): An len 4 array representing the change in the playe
        achievements (jnp.ndarray): A 1D array (22,) of achievements, where each element is an bo
        n (int): The target amount to reach in inventory for the main gain item.

    Returns:
        bool: True if the main gain item in inventory has reached the target amount n, False othe
    \"\"\"
    return TODO


def task_reward(inventory_diff, closest_blocks, closest_blocks_prev, player_intrinsics_diff, achi
    \"\"\"
    Calculates the reward for Task ‘$db.current.skill_name$‘ based on changes in inventory and ot
    Do not call external functions or make any assumptions beyond the information given to you.

    Args:
        inventory_diff (Inventory): The change in the player’s inventory between the current and
        closest_blocks (numpy.ndarray): A 3D array of shape (len(BlockType), 2, K) representing t
        #default of 30,30 if less then k seen, ordered by distance (so :,:,0 would be the closest
        #Since the environment is a 2d gridworld, an object next to the player will have a distan
        closest_blocks_prev (numpy.ndarray): A 3D array of shape (len(BlockType), 2, K) represent
        #default of 30,30 if less then k seen, ordered by distance (so :,:,0 would be the closest
        health_penalty (float): The penalty for losing health. Negative when loosing health and p
        player_intrinsics_diff (jnp.ndarray): An len 4 array representing the change in the playe
        achievements_diff (jnp.ndarray): A 1D array (22,) of achievements, where each element is

    Returns:
        float: Reward for RL agent

    Note:
        The task reward should be two parts:
          1. Sparse reward
          2. Dense reward
        Make sure to disable (2) if (1) is triggered, e.g. sparse_reward +  (sparse_reward == 0.0

    \"\"\"
    return TODO + health_penalty

def task_network_number():
"""
Returns the network index corresponding to the nodes associated skill
Returns:
int: Network index
```

```
\"\"\"
return TODO

Given the above documentations, implement the task_is_done, task_reward, and task_network_number

$db.current.skill_with_consumption$
$db.current.reward$

The dense reward to include is:

$db.current.dense_reward_factor$

The current number of skills is:

$db.current.num_skills$

Implementation Guidelines:

For task_is_done:
 • Identify the main gain item from the skill's "gain" dictionary (the item with the highest gai
 • Check if the current inventory amount of that main gain item is >= n (the target amount)
 • Return True when the target amount is reached, False otherwise
 • Use inventory.{item_name} to access inventory amounts (e.g., inventory.wood, inventory.stone)
 • If a skill is ephemeral, the inventory does not suffice, so for completion criteria you can us

For task_reward and task_network_number:
 • Follow the existing reward structure and network numbering as before

The task network number should be num_skills since we're creating a new skill and the networks ar
Do not change the function signature or the docstrings. Do not make any assumptions beyond the ir
The code you write should be able to be jax compiled, no if statements.
No need to retype BlockType, Inventory, and Achievement they will be provided in the environment.
No need to add coefficents to rewards, for example, no need for 10 * inventory_diff.*, just use t
Return all three functions in a single code block, don't seperate it into 3.
No need to return the docstrings.
Your code will be pasted into a file that already has the following imports. Do not add any addit
from craftax.craftax_classic.constants import *
from craftax.craftax_classic.envs.craftax_state import Inventory
import jax
```

E.2 Knowledge Base / Skill Updates via Pivotal Trajectory Analysis

After each verified success, we run *pivotal trajectory analysis* to reconcile what the skill *actually* needed and produced with what it *claimed* to need and produce. Concretely, we compare the inferred preconditions/effects against the successful rollout and (i) update the skill's requirements, consumption, and gain, and (ii) propose edits to the knowledge base (KB) where assumptions can be marked verified, removed, or left unchanged.

Procedure. Let $\sigma = (z_0, \dots, z_T)$ be a successful rollout and let $L(z_t)$ be the set of facts at time t.

1. **Summarize the trajectory.** Convert σ to a sequence of fact-sets $\mathbf{s} = (L(z_0), \dots, L(z_T))$ and attach the current skill definition and KB snapshot.
2. **LLM pass #1: Update the skill.** Using the "update_skill_from_trajectory" prompt (below), the LLM infers requirement, consumption, and gain functions written as lambda strings in the form `"lambda n:  a*n + b"`, where a captures per-application usage and b captures fixed setup needs.
3. **LLM pass #2: Propose KB edits.** Using the "propose_knowledge_base_updates" prompt (below), the LLM proposes targeted KB changes: change `ASSUMPTION` $\rightarrow$ `VERIFIED`

when supported by the trajectory, remove disproven assumptions, and keep unsupported assumptions unchanged.

4. **Post-processing.** We validate JSON structure and key alignment (each requirement/consumption key must match an existing skill gain), apply the updates, and recompute the frontier.

Raw prompt: `update_skill_from_trajectory`**.**

```
    You need to update a skill based on its execution trajectory.

Current Skill:
‘‘‘
$db.current.skill_with_consumption$
‘‘‘

Existing Skills (for requirements validation):
‘‘‘
$db.skills$
‘‘‘

Trajectory Data:
‘‘‘
$db.example_trajectory$
‘‘‘

## Task

Analyze the trajectory to determine what the skill actually required, consumed, and gained, then

The trajectory shows a specific instance (e.g. n=1), but you need to infer the general pattern.

**IMPORTANT CONSTRAINTS:**
- Requirements are a SUPERSET of consumption: requirements include everything needed (both consum
- Each value in requirements/consumption should be written as a Python lambda function string tha
  - a = amount of resource consumed PER unit of gain (scales with n)
  - b = amount of resource required but NOT consumed (fixed amount regardless of n)
  - Ask yourself: "Does this requirement scale with the number of times I apply the skill?"
    - If YES (scales with n): use "lambda n: a*n + 0" format
    - If NO (fixed amount): use "lambda n: 0*n + b" format
- Requirements do not support 'or'
- Each key in requirements/consumption must be a key in the gain of an existing skill.

Update the skill's requirements and gain as lambda functions based on what the trajectory reveale

# Formatting
‘‘‘json
{
"skill_name": "", # name of the skill
"updated_requirements": {},  # total amount needed available using "lambda n: a*n + b" format. Ea
"updated_consumption": {}, # amount consumed using "lambda n: a*n + b" format. Each key must exac
"updated_gain": {} # a dictionary of what is gained by applying the skill. The gain for the skill
}
```

Raw prompt: `propose`$_{k}nowledge_{b}ase_{u}pdates.$

```
    You need to propose which parts of the knowledge base should be updated based on trajectory a

Knowledge Base:
‘‘‘
```

```
$db.knowledge_base$
‘‘‘

Trajectory Data:
‘‘‘
$db.example_trajectory$
‘‘‘

Current Skill:
‘‘‘
$db.current.skill_with_consumption$
‘‘‘

## Task

Look at the knowledge base structure and propose which specific entries/fields should be updated

The knowledge base contains requirement lists with items marked as "ASSUMPTION" or "VERIFIED". Ba

1. ASSUMPTIONS that were confirmed by the trajectory become "VERIFIED: condition"
2. ASSUMPTIONS that were proven FALSE by the trajectory should be REMOVED
3. ASSUMPTIONS that cannot be verified from this trajectory MUST remain "ASSUMPTION: condition"
4. New requirements discovered from the trajectory are added as "VERIFIED: condition"

**CRITICAL**: Only make changes when you have clear evidence from the trajectory:
- Change ASSUMPTION to VERIFIED if trajectory confirms it’s true
- REMOVE assumptions if trajectory proves they’re false
- KEEP assumptions unchanged if trajectory provides no evidence either way

Requirements should be in the format: "VERIFIED: condition" or "ASSUMPTION: condition"

# Formatting
‘‘‘json
{
"proposed_updates": [
    {
        "path": ["key1", "subkey", "field"], # Path to the field in the knowledge base
        "updated_requirements": [], # Complete updated list of requirements (verified + remaining
        "reason_for_update": "" # What the trajectory showed that confirms, disproves, or leaves
    }
]
}
‘‘‘
```