LOOPED-RSI: WHEN THE CONSOLIDATED WHOLE IS GREATER THAN THE SUM OF ITS PARTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Recursive self-improvement (RSI) increasingly relies on branching search over populations of self-modified agents. We identify two structural limitations of this paradigm. First, useful progress can become fragmented across lineages: a population may accumulate complementary discoveries that are not jointly realized by any individual agent. Second, task capability and self-improvement capability are inherited together even though they require different forms of feedback. We introduce **Looped-RSI**, which augments branching self-improvement with periodic population-level consolidation. Rather than selecting only among existing agents, Looped-RSI reasons over the evidence accumulated across the population to construct a successor, while separately consolidating the improver according to the quality of the descendants it produces. Across interactive environments, Looped-RSI produces stronger self-improvement trajectories. We further show that consolidation can recover substantial progress from populations in which every individual agent receives zero reward, and that the resulting improvers transfer effectively to previously unseen environments. These results suggest that recursive progress need not reside in a single lineage or a single agent state, but can emerge across the search process and be made heritable through consolidation.

1 INTRODUCTION

Recursive self-improvement (RSI) rests on the possibility that improvements made by an intelligent system can enable further improvements, allowing progress to compound over time (Good, 1966; Schmidhuber, 1987; 2003). Recent approaches pursue this goal through open-ended search over populations of self-modified agents (Zhang et al., 2026a; Wang et al., 2026; Liu et al., 2026; Iacob et al., 2026; Zhang et al., 2026b). Rather than committing to a single trajectory, these methods branch repeatedly, preserving alternative modifications as potential stepping stones for future improvement. This is a natural strategy for open-ended search, where the value of a modification may become apparent only through its descendants (Wright, 1932; Lehman & Stanley, 2011).

Yet, the same branching mechanism that protects search from premature commitment creates an inheritance bottleneck. As exploration diverges, useful progress can become *fragmented across lineages*. One branch may discover a promising strategy but fail to implement it correctly; another may develop a complementary mechanism; a third may reveal through failure that an apparently plausible direction should be avoided. In other words, these discoveries remain present somewhere in the search tree, but they are embedded in different nodes. Moreover, the value of a partial discovery need not be reflected in the reward of the node that contains it. A population can therefore accumulate meaningful progress even when no individual agent jointly realizes that progress, and even when candidate-level evaluation provides little indication of where it resides.

We observe a striking instance of this phenomenon in a sparse-reward interactive BALROG (Paglieri et al., 2025) environments. Open-ended RSI (Zhang et al., 2026b) produces a search tree in which *every generated agent receives zero task reward*. Candidate-level evaluation therefore indicates no progress and provides no basis for identifying a useful lineage. Nevertheless, as illustrated in Figure 1 consolidating exactly the same population, **without further branching,** can produce an agent with up to 80% accuracy [1]. Hence this shows, that the search had accumulated actionable

[1]The analysis of this consolidation is presented in the Appendix A

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

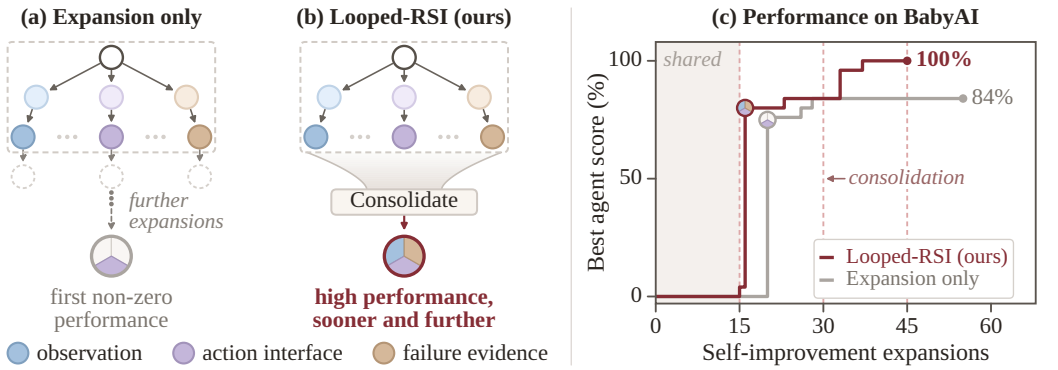


Figure 1: The consolidated whole is greater than the sum of its parts. Color marks useful population finding. (a) Expansion-only RSI scatters abilities across separate lineages; the first non-zero agent appears only after further expansions. (b) Looped-RSI consolidates them, each **worth zero** alone, into one successor that reaches **high performance earlier** and **a higher ceiling**. (c) Comparison with Hyperagents Zhang et al. (2026b). Dashed lines mark consolidation firing every 15 expansions.

progress that was expressed by none of its individual agents, revealing a critical limitation of lineage-based inheritance. Existing RSI methods may use archive-wide information to guide selection or future modification (Wang et al., 2026; Zhang et al., 2026b; Liu et al., 2026), but each realized descendant is still produced by modifying an existing agent state. When complementary progress is distributed across different trajectories, selecting another parent cannot by itself make that progress jointly heritable. The relevant question is therefore not only *which agent should the search continue from?*, but also *what has the search collectively learned?*

Furthermore, branching RSI introduces a credit-assignment problem *within* individual nodes. In systems such as HyperAgent (Zhang et al., 2026b), a node contains both a task agent and the mechanism responsible for improving it. These two components are inherited together, yet node-level reward primarily evaluates only the first: how well the current task agent performs. This creates a mismatch between the signal used for selection and the state that is propagated, akin to the performance-metaproductivity mismatch identified by HGM (Wang et al., 2026), but located inside the node rather than between nodes. A node may achieve high task reward because it contains a strong task agent while simultaneously carrying an ineffective improver; selecting that node propagates both. Conversely, a weak task agent may contain an improver that would generate substantially better descendants, but its value can remain hidden behind the node's poor current reward. Current *capability* and future *improvability* are therefore coupled inside the same unit of inheritance even though they answer different questions and require different feedback.

We introduce **Looped-RSI**, a recursive self-improvement paradigm designed around these two observations. Across nodes, it introduces *population-level consolidation*: a many-to-one operation that reasons over modifications, evaluation outcomes, behaviors, failures, and self-improvement traces accumulated across multiple trajectories to construct a new shared successor. Consolidation is not a mechanical union of agents; it can preserve, discard, or complete discoveries according to the evidence produced during search. The resulting successor becomes the starting point for subsequent exploration, allowing progress to compound across lineages rather than only along them. Within each node, Looped-RSI separates the consolidation of *task capability* from the consolidation of *self-improvement capability*. Task agents can be evaluated directly by what they achieve. Improvers cannot: their quality lies in the agents they produce. We therefore consolidate alternative improvers using evidence from their descendants and compare them from the same task-agent state. Hence, task capability is credited by current performance, while self-improvement is credited by its effect on future agents. Together, these two operations change the role of a branching RSI population. The search tree is not only a collection of candidate states from which another parent must be selected; it is also a record of what the self-improvement process has discovered. Looped-RSI periodically converts this accumulated *search state* back into the *agent state* inherited by future search.

Our main contributions are:

- We identify two structural limitations of population-based RSI: *fragmented progress across nodes*, where complementary discoveries remain distributed across separate lineages, and *credit coupling within nodes*, where task agents and meta-agents are inherited together despite requiring different feedback.
- We introduce *Looped-RSI*, which periodically consolidates population-level search evidence into a shared successor. Across nodes, it synthesizes task capability from discoveries distributed across trajectories; within nodes, it consolidates the improver, i.e., meta-agent, separately and evaluates it through the descendants it produces.
- We show that consolidation changes both current capability and future self-improvement. Looped-RSI reaches stronger final agents than HyperAgent on BabyAI and Crafter with fewer evolutionary expansions, preserves its advantage under larger and held-out evaluation, and produces meta-agents that transfer substantially better from BabyAI to unseen BabaIsAI and Crafter environments.

2 LOOPED-RSI

Our proposed framework Looped-RSI augments standard RSI self-improvement branching with a periodic consolidation step designed around the two limitations identified above. *Across nodes*, useful progress may be fragmented across different branches and never appear jointly in a single agent. *Within nodes*, the task agent and its improver are inherited together even though they require different forms of feedback. Looped-RSI addresses these two problems separately.

Let the agent at the beginning of round r be

$$S_r = (\pi_r, \mu_r),$$

where π_r is the task agent and μ_r is the improver responsible for generating future modifications. Starting from S_r, the underlying branching RSI procedure performs N self-improvement expansions, which we call $\mathrm{Explore}$,

$$\mathcal{A}_r = \mathrm{Explore}(S_r, N),$$

producing a set of descendant nodes $\mathcal{A}_r$ including the round root S_r together with their modifications, evaluations, behaviors, and self-improvement traces.

Looped-RSI then consolidates information from this round with $\mathrm{Consolidate}$ operation into a new single agent

$$S_{r+1} = (\pi_r, \mu_r) = \mathrm{Consolidate}(\mathcal{A}_r)$$

which becomes the starting point for the next round.

2.1 ACROSS NODES: CONSOLIDATING FRAGMENTED PROGRESS

Branching exploration may discover complementary capabilities along different trajectories without ever combining them in a single node. Looped-RSI therefore reasons over the full population produced during the round before deciding what should be carried forward.

We first choose a high-performing state as the *reference state*,

$$S_r^\star = (\pi_{r+1}^\star, \mu_{r+1}^\star) = \operatorname*{arg\,max}_{\pi \in \{\pi_r\} \cup \{\pi_v : v \in \mathcal{A}_r\}} \widehat{J}(\pi),$$

where $\widehat{J}$ denotes measured task performance. The reference agent provides a strong starting point for consolidation, but consolidation is not restricted to the capabilities already present in that agent.

For every node $v \in \mathcal{A}_r$, we collect evidence describing both what changed and what happened after that change. We write

$$e_v = (\Delta_v, y_v, b_v, h_v),$$

where Δ_v contains the node's ancestry and modifications, y_v contains aggregate and fine-grained evaluation results, b_v summarizes its behavior during evaluation, and h_v contains the self-improvement trace that motivated the modification. The evidence available from round r is $E_r = \{e_v \mid v \in \mathcal{A}_r\}$.

Crucially, E_r includes unsuccessful nodes. A low or zero task score does not imply that every mechanism introduced by that node is useless. A node may discover a valid action interface but fail to use observations correctly, implement a useful capability but never invoke it, or reveal through failure that a particular strategy is invalid. Such information may only become useful when considered together with discoveries from other branches. Using E_r, the task consolidator proposes

$$\widetilde{\pi}_{r+1} = C_{\text{task}}\left(\pi_r^\star, E_r\right).$$

The consolidator may retain useful mechanisms, discard unsupported changes, repair partially correct implementations, complete unfinished capabilities, or combine complementary discoveries from different nodes. The goal is therefore not to select one existing node or mechanically merge patches, but to construct a single coherent task agent from evidence distributed across the population.

The proposed task agent is then evaluated using the same task protocol used during exploration. If it satisfies the required validity checks and does not reduce measured task performance relative to the reference agent, it becomes π_{r+1}; otherwise, the reference task agent is retained. The exact acceptance rule is given in Appendix C. This stage directly addresses *fragmented progress across nodes*: capabilities discovered in separate branches can become combined in the task agent used to start the next round.

2.2 Within Nodes: Decoupling Capability from Improvability

A separate credit-assignment problem arises within each node. A node contains both a task agent π and an improver μ, yet its task reward primarily measures the current performance of π. Selecting the node then propagates both components using a signal that directly evaluates only one of them.

This is problematic because task capability and self-improvement capability answer different questions. The task agent should be evaluated by how well it performs now. The improver should be evaluated by whether it can generate better agents in the future.

Looped-RSI therefore consolidates the improver separately and addresses this by reasoning separately over the improvers encountered during the round and the descendants they produced. In this sense, the improver consolidator acts as a *meta-meta-improver*: it improves the mechanism responsible for future self-improvement. Different branches may contain different versions of μ, each of which has generated its own descendants. For each distinct improver, we collect evidence about the modifications it produced, whether those modifications resulted in valid descendants, and how those descendants performed. We denote this collection by E_r^{imp}.

Using this descendant-level evidence, the improver consolidator proposes

$$\widetilde{\mu}_{r+1} = C_{\text{imp}}\left(\mu_r^\star, E_r^{\text{imp}}\right).$$

The proposed improver and the current improver are then evaluated from the *same task-agent starting point*, π_{r+1}. This isolates the quality of the improver from the performance of the task agent with which it happened to co-evolve. For n independent self-improvement attempts, let $v_i^{(\mu)}$ denote the descendant produced by improver μ. For an improver μ, we measure *offspring validity* as the fraction of valid probes,

$$\widehat{V}_n(\mu) = \frac{1}{n}\sum_{i=1}^{n} \mathbf{1}\left[v_i^{(\mu)} \text{ is valid}\right],$$

and *offspring quality* is the mean probe-child score,

$$\widehat{Q}_n(\mu) = \frac{1}{n}\sum_{i=1}^{n} \widehat{J}\left(v_i^{(\mu)}\right).$$

A candidate improver is accepted only when $\widehat{V}_n(\mu_{\text{cand}}) \geq \widehat{V}_n(\mu_{\text{cur}})$ and $\widehat{Q}_n(\mu_{\text{cand}}) \geq \widehat{Q}_n(\mu_{\text{cur}})$. That is, the new improver must be at least as likely to produce a usable change, and at least as good at producing high-scoring children, as the improver it replaces. The full probing, acceptance, and revocation procedure is described in Appendix C.

The two consolidation stages therefore follow different credit-assignment principles:

task agents are evaluated by what they achieve;	improvers are evaluated by what they produce.

After both stages, the resulting task agent and improver form the agent used to start the next exploration round,

$$S_{r+1} = (\pi_{r+1}, \mu_{r+1}).$$

The overall procedure alternates between branching exploration and consolidation,

$$S_r \xrightarrow{\text{Explore}} \mathcal{A}_r \xrightarrow{\text{Consolidate}} S_{r+1} \xrightarrow{\text{Explore}} \mathcal{A}_{r+1} \to \cdots .$$

Looped-RSI thus addresses the two current limitations of standard RSI through two corresponding operations. *Across nodes*, task consolidation combines progress that would otherwise remain fragmented across branches. *Within nodes*, improver consolidation separates current task capability from future improvability and evaluates the improver through the descendants it produces. Together, these operations allow discoveries made during branching search to be carried forward into subsequent rounds of self-improvement.

3 Related work

RSI denotes a regime in which the object being improved includes the improvement procedure itself, so that each accepted modification can raise the quality of subsequent modifications. The idea traces to Good (1966), who argued that a machine able to design a better machine would trigger an "intelligence explosion", and was first given algorithmic form by Schmidhuber (1987), whose self-referential learner proposes edits to its own program and keeps those that test well. The Success-Story Algorithm (Schmidhuber & Zhao, 1996; Schmidhuber et al., 1997) judges such edits in hindsight, rolling back any modification, together with everything built on it, whose subsequent reward rate does not exceed the rate before it. The Gödel Machine replaces this empirical test with a proof: it rewrites its own code, including its proof searcher, only when it can prove an expected-utility improvement relative to its encoded axioms and utility function (Schmidhuber, 2003).

Recent LLM-based systems return to empirical feedback, operationalizing the principle through code modification validated on tasks. STOP applies a program improver to itself (Zelikman et al., 2024), Gödel Agent and SICA let agents revise their own implementations (Yin et al., 2025; Robeyns et al., 2025), and DGM pairs self-modification with an expanding archive of agents that preserves diverse evolutionary branches (Zhang et al., 2026a). Since the evolving coding agent also produces the next modifications, coding gains can serve both task-solving and further self-modification, but the two need not coincide; DGM and SICA nonetheless select parents by benchmark score, treating a stronger task solver as a more promising source of future modifications. HGM names this the metaproductivity–performance mismatch and instead estimates clade-metaproductivity from descendant outcomes to guide the search over self-modifications (Wang et al., 2026). Later work refines other components of the same archive search: the Mendel Gödel Machine changes the evidence a modification is conditioned on, drawing on an agent's behavior across several tasks or on a differently descended agent's handling of the same task rather than on one failed run (Liu et al., 2026), while the Red Queen Gödel Machine questions the fixed benchmark itself, searching evaluators jointly with task agents and letting the utility change between epochs while holding it fixed within each (Iacob et al., 2026).

From a design perspective, HyperAgents (Zhang et al., 2026b) explicitly separates the node into a *task agent*, which solves the task, and an editable *meta agent*, which modifies both the task agent and itself. This removes the assumption that task-solving ability and self-modification ability must coincide, and makes the improvement procedure itself subject to recursive improvement. However, the two components are still inherited together inside a single node, and each new descendant is generated by modifying one existing node at a time. Two limitations therefore remain. First, useful discoveries made along different branches can remain fragmented across the archive rather than becoming jointly inherited by future agents. Second, node-level task performance directly measures the current task agent but does not directly measure the quality of the meta agent that generates future descendants; a strong task agent may therefore carry a weak improver, while a useful improver may remain attached to a weak task agent. Looped-RSI targets these two limitations directly: it consolidates complementary progress across branches into a shared successor, and separately consolidates task and self-improvement capability using task performance and descendant quality, respectively.

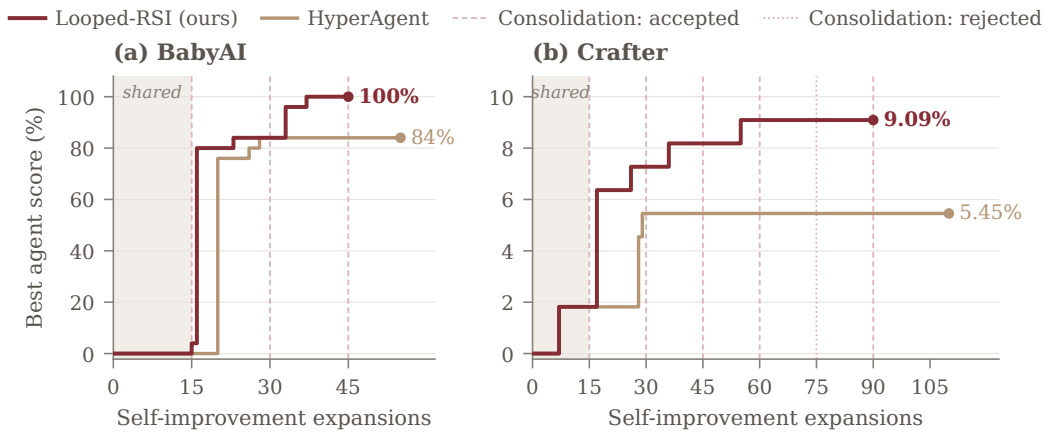


Figure 2: Best agent found over self-improvement expansions on (a) BabyAI and (b) Crafter. The first 15 expansions are shared by both methods; vertical lines mark Looped-RSI consolidation boundaries. HyperAgent receives more evolutionary expansions in both environments: 55 vs. 45 on BabyAI and 110 vs. 90 on Crafter.

4 Experiments

In this section, we conduct several experiments that evaluate Looped-RSI in different settings.

4.1 Experimental Setup

We perform full self-improvement runs on two interactive environments from BALROG (Paglieri et al., 2025): **BabyAI**, a language-conditioned grid world with compositional navigation and object-manipulation tasks (Chevalier-Boisvert et al., 2019), and **Crafter**, an open-world survival environment with procedurally generated worlds and 22 achievement-based objectives (Hafner, 2021). We use a sparse *discovery setting*: agents receive the task objective but no action vocabulary, demonstrations, environment mechanics, or task-specific hints. We compare against HyperAgent (Zhang et al., 2026b) using the same branching self-improvement procedure, task/meta-agent architecture, evaluation protocol, and Qwen3.6-35B-A3B backbone. To control for early search stochasticity, Looped-RSI and HyperAgent share exactly the same first 15-node population and diverge only at the first consolidation boundary. Looped-RSI performs 45 evolutionary expansions on BabyAI and 90 on Crafter, while HyperAgent is allowed 55 and 110, respectively. HyperAgent therefore receives more ordinary branching opportunities in both comparisons. Consolidation requires additional inference and evaluation and is not counted as an evolutionary expansion; these numbers should therefore be interpreted as evolutionary-search budgets rather than matched total inference compute. Full environment, evaluation, consolidation, and implementation details are provided in Appendix C.

4.2 Consolidation Changes the Course of Self-Improvement

We first ask whether consolidating discoveries distributed across different branches changes what subsequent self-improvement can achieve. Figure 2 compares the full Looped-RSI and HyperAgent trajectories on BabyAI and Crafter. In both environments, the methods share exactly the same first 15 expansions and differ only after the first consolidation boundary.

BabyAI. The shared first-round population provides an extreme example of fragmented progress: all (15) generated nodes receive zero task reward. Candidate-level evaluation therefore identifies no successful lineage to continue. After consolidation, however, the very first new Looped-RSI expansion, generation (16), produces an agent with (80%) success across the five BabyAI task families. HyperAgent, continuing ordinary branching from the identical population, first reaches comparable performance at generation (20), with (76%) success. The advantage persists throughout evolution: Looped-RSI reaches (100%) within (45) expansions, whereas HyperAgent reaches (84%) after (55).

Crafter. Crafter shows the same pattern over a longer horizon. During the shared first 15 expansions, the strongest agent remains near 2% achievement rate. After the first consolidation, Looped-RSI rises above 6%, while HyperAgent does not reach comparable performance until approximately

Table 1: Final evaluation of the final agents. BabyAI uses 50 fresh episodes per split; Crafter uses 20 fresh procedurally generated worlds. The BabyAI held-out split is never used during self-improvement.

Environment	Evaluation	Looped-RSI	HyperAgent
BabyAI	Fresh episodes	**98.0%**	82.0%
BabyAI	Held-out split	**96.0%**	80.0%
Crafter	Fresh worlds	**9.32%**	6.59%

expansion 28. The trajectories remain separated thereafter: Looped-RSI reaches 9.09% within 90 expansions, compared with 5.45% for HyperAgent after 110. Across both environments, consolidation changes more than which node is selected next. By bringing progress fragmented across branches into the agent from which search continues, Looped-RSI allows later generations to build directly on discoveries that were previously separated. HyperAgent can recover some of these gains through additional branching, but Looped-RSI makes them available earlier and reaches stronger agents with fewer evolutionary expansions.

The advantage persists under independent final evaluation. Search-time scores are measured with relatively small evaluation budgets and repeatedly consulted during self-improvement. We therefore independently re-evaluate the final agents using substantially larger episode budgets. On BabyAI, Looped-RSI retains a 16-percentage-point advantage under both evaluations: 98% versus 82% on fresh episodes and 96% versus 80% on the held-out OOD `MixedTestLocal` split. The Crafter results are consistent with this finding, showing the superiority of Looped-RSI. As a complementary analysis, Appendix A freezes this all-zero population and shows how consolidation combines complementary mechanisms that occur separately in unsuccessful nodes, while synthesizing additional behavior from population-wide failure evidence.

Consolidation improves the machinery for subsequent self-improvement. The best-agent trajectories capture the strongest solution discovered, but not whether the inherited agent remains effective at generating further self-improvements. We therefore examine the progression of the fraction of Crafter expansions that produce a node with non-zero staged reward. After the first consolidation, approximately 60% of Looped-RSI expansions produce a scoring descendant across most subsequent windows. HyperAgent remains below 20% through expansion 60 and rises only later, reaching roughly 30% in its final windows. This difference is particularly relevant to the within-node credit-assignment problem. HyperAgent selects nodes primarily according to current task performance, but inherits both the task agent and the meta-agent attached to that node. A strong task agent therefore does not imply that the inherited meta-agent is effective at producing the next generation of improvements. Our proposed approach Looped-RSI instead consolidates the meta-agent separately, using evidence from the descendants produced by different improvers. In this sense, consolidation acts at a second level of self-improvement: it improves not only the current agent, but also the mechanism responsible for generating future improvements.

The productive-descendant analysis is consistent with improving the machinery of self-improvement, but does not by itself isolate the contribution of the task agent from that of the improver, since both may change during consolidation. Crafter provides a more direct test. Across the full run, all six task-consolidation proposals were rejected, so the accepted consolidation updates changed only the improver. We can therefore compare the consolidated and current improvers from exactly the same task-agent state. In round 2, the children generated by the consolidated improver achieved a mean score of 6.4, compared with 2.7 for the current improver; in round 3, the corresponding scores were 5.5 and 0.9. Because the task-agent starting point is held fixed, these comparisons directly attribute the improvement in descendant generation to the improver consolidation itself. Appendix B traces the population evidence and modifications underlying these accepted updates accross both BabyAI and Crafter. Section 4.3 asks a complementary question: whether the resulting improver remains effective after the source-domain task agent is removed altogether. We therefore transfer the improvers evolved on BabyAI to previously unseen environments and evaluate whether they can drive recursive self-improvement from the same target-domain starting point.

4.3 Within Nodes: Evolved Meta-Agents Transfer Across Environments

We next evaluate the component responsible for future self-improvement more directly. We take the final meta-agents evolved on BabyAI by Looped-RSI and HyperAgent and use them to initialize

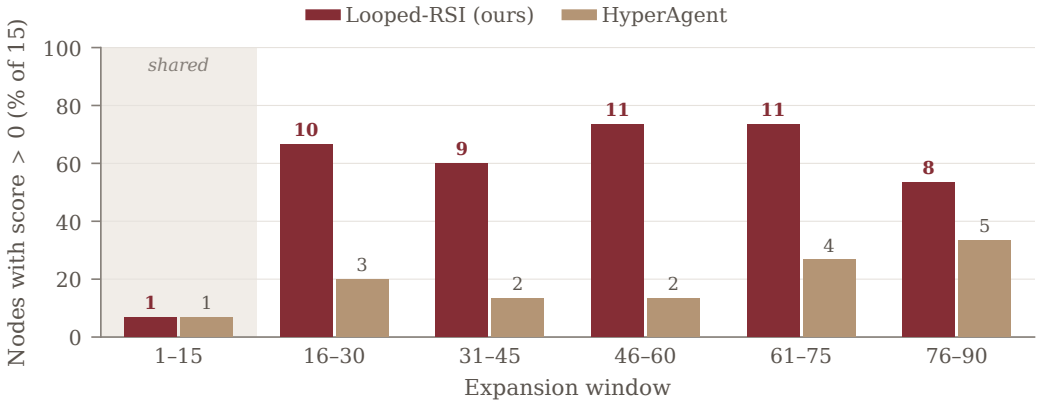


Figure 3: Fraction of newly generated Crafter nodes obtaining non-zero staged reward in successive 15-expansion windows. Numbers above the bars indicate the number of scoring descendants out of 15. The first window is shared by both methods.

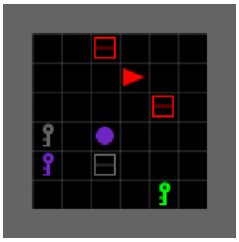
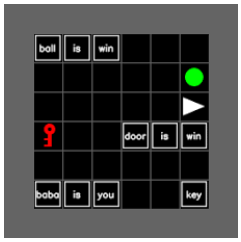
BabyAI (RSI Domain)	BabyAI → BabaIsAI	BabyAI → Crafter
		
Hyperagents	2.5%	0.0%
Looped-RSI	**47.50%**	**19.09 %**

Table 2: Best target-domain performance obtained by transferring the final improvers learned on BabyAI to BabaIsAI and Crafter. In each target domain, the transferred improver starts from the same target-domain task agent and is allowed to perform the same sequence of self-improvement steps. Both methods receive the same self-improvement budget, and no population-level consolidation is performed during transfer.

self-improvement on two previously unseen environments, BabaIsAI and Crafter. Both methods start from the same target-domain task agent and follow the same self-improvement protocol. No population-level consolidation is performed during transfer. Importantly, the transferred meta-agent retains its full recursive role: it may modify the task agent, other parts of the codebase, and its own self-improvement procedure. The transfer experiment therefore evaluates more than whether a fixed editing strategy generalizes. It asks whether the meta-agent acquired during BabyAI provides a stronger starting point for recursive self-improvement when the environment changes.

Table 2 summarizes the best target-domain performance reached during transfer. The Looped-RSI meta-agent reaches 47.5% on BabaIsAI and 19.09% on Crafter, compared with only 2.5% and 0.0% for HyperAgent. The transfer trajectories in Figure 4 reveal that this separation begins immediately. On Crafter target, the meta-agent transferred from Looped-RSI produces a positive-scoring agent in the first generation, whereas the HyperAgent meta-agent produces zero reward. At this point, the child is generated by the meta-agent inherited directly from BabyAI; any target-domain modification of the meta-agent can influence only subsequent generations. The first-generation gap therefore provides a particularly direct indication that Looped-RSI has produced a stronger meta-agent before target-domain recursive adaptation has had an opportunity to compound. For BabaIsAI target, one 1-shot expansion was enough to get a strong scoring agent.

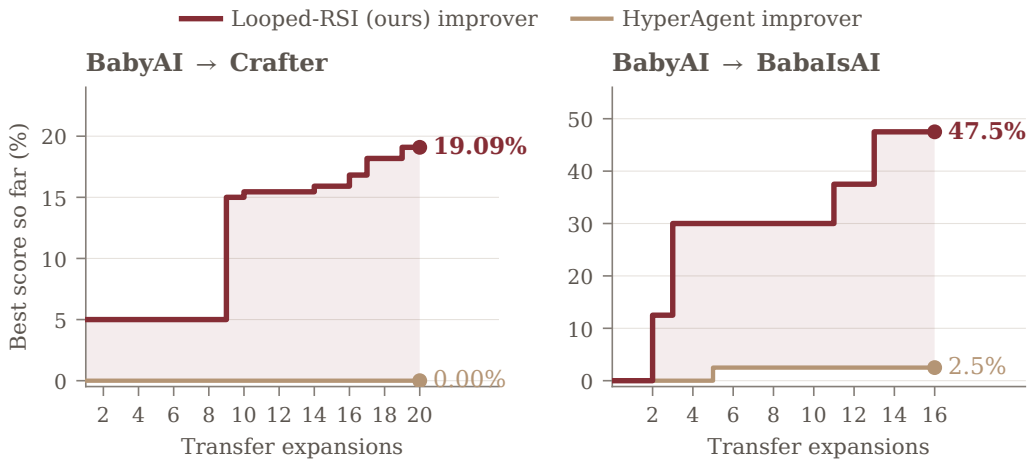


Figure 4: Cross-environment transfer of meta-agents evolved on BabyAI to BabaIsAI and Crafter. Both methods start from the same target-domain task agent and receive the same self-improvement budget, with no population-level consolidation during transfer. The Looped-RSI meta-agent already produces positive task performance in the first generations on both target environments, while HyperAgent remains at zero. Subsequent generations allow the meta-agent itself to evolve, revealing how effectively each transferred self-improvement process continues to improve.

The later expansions test a second, distinct property. As modifications to the meta-agent are inherited, continued transfer measures not only whether the initial improver can improve the task agent, but also whether it provides a strong basis for improving its own self-improvement procedure. Here the trajectories continue to diverge rather than converge: Looped-RSI keeps producing stronger descendants and ultimately reaches the performance reported in Table 2, while HyperAgent makes little or no progress on either target. This behavior directly reflects the within-node credit-assignment problem. HyperAgent selects nodes according to current task performance, yet inherits both the task agent and the meta-agent attached to that node. A strong task agent therefore need not carry a strong mechanism for producing future improvements. Looped-RSI instead consolidates the meta-agent separately using evidence from the descendants generated by different improvers. In this sense, the improver consolidator acts as a *meta-meta-improver*: it improves the mechanism responsible for future self-improvement. The transfer results expose both consequences of this distinction. The Looped-RSI improver is already more effective when first moved to an unseen environment, and, once recursive adaptation begins, it provides a substantially stronger basis for continued self-improvement. Thus, the advantage learned on BabyAI is not confined to the task agent or to the source environment; it is carried by the mechanism that generates and recursively refines future improvements.

5 Conclusion

We introduced Looped-RSI, a recursive self-improvement framework that augments branching search with periodic population-level consolidation. The method is motivated by two limitations of current population-based RSI. *Across nodes*, useful progress can become fragmented across separate lineages and remain invisible to individual-agent reward. *Within nodes*, current task performance does not directly measure the quality of the meta-agent responsible for producing future improvements. Looped-RSI addresses these problems separately. It consolidates task capabilities from evidence distributed across branches and independently consolidates the meta-agent using the descendants it produces. Across BabyAI and Crafter, this leads to stronger self-improvement trajectories and stronger final agents than HyperAgent, even with fewer evolutionary expansions. The resulting meta-agent also transfers substantially better to unseen environments, showing that the benefit is not confined to the source-domain task agent. More broadly, our results suggest that recursive progress should not be identified solely with the best agent currently present in the search tree. Progress can be distributed across unsuccessful branches, and the mechanism that generates future improvements can itself require separate credit and consolidation. Effective RSI therefore requires not only searching for better agents, but also determining what the search has discovered and which improvement mechanisms should be inherited next.

AI USE STATEMENT

Generative AI tools were used to assist with code implementation and debugging, as well as language polishing of the manuscript. The core research idea was developed by the authors. AI-assisted code was reviewed and tested by the authors before use in the experiments, and all AI-assisted manuscript parts were manually reviewed and revised for technical correctness and consistency with the reported results. The authors take full responsibility for the final content of the paper.

REFERENCES

Maxime Chevalier-Boisvert, Dzmitry Bahdanau, Salem Lahlou, Lucas Willems, Chitwan Saharia, Thien Huu Nguyen, and Yoshua Bengio. Babyai: First steps towards grounded language learning with a human in the loop. In *International conference on learning representations*, volume 105, pp. 1–14. New Orleans, LA, 2019.

Irving John Good. Speculations concerning the first ultraintelligent machine. In *Advances in computers*, volume 6, pp. 31–88. Elsevier, 1966.

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

Alex Iacob, Andrej Jovanović, William F. Shen, Daniel Burkhardt, Meghdad Kurmanji, Nurbek Tastan, Lorenzo Sani, Niccolò Alberto Elia Venanzi, Ambroise Odonnat, Zeyu Cao, Bill Marino, Xinchi Qiu, and Nicholas D. Lane. The red queen gödel machine: Co-evolving agents and their evaluators. *arXiv preprint arXiv:2606.26294*, 2026.

Joel Lehman and Kenneth O. Stanley. Abandoning objectives: Evolution through the search for novelty alone. *Evolutionary Computation*, 19(2):189–223, 2011.

Changzhi Liu, Yilun Liu, Sikuan Yan, Volker Tresp, and Yunpu Ma. Mendel gödel machine: Recursive self-improving coding agents via comparative evolution. *arXiv preprint arXiv:2608.07645*, 2026.

Davide Paglieri, Bartłomiej Cupiał, Samuel Coward, Ulyana Piterbarg, Maciej Wołczyk, Akbir Khan, Eduardo Pignatelli, Łukasz Kuciński, Lerrel Pinto, Rob Fergus, et al. Balrog: Benchmarking agentic llm and vlm reasoning on games. In *International Conference on Learning Representations*, volume 2025, pp. 96666–96702, 2025.

Maxime Robeyns, Martin Szummer, and Laurence Aitchison. A self-improving coding agent. In *ICLR Workshop on Scaling Self-Improving Foundation Models (SSI-FM)*, 2025. URL https://openreview.net/forum?id=rShJCyLsOr.

J. Schmidhuber. Gödel machines: self-referential universal problem solvers making provably optimal self-improvements. Technical Report IDSIA-19-03, arXiv:cs.LO/0309048, IDSIA, Manno-Lugano, Switzerland, 2003. Revised 2006.

Jürgen Schmidhuber. Evolutionary principles in self-referential learning, or on learning how to learn: the meta-meta-... hook. Diploma thesis, Institut für Informatik, Technische Universität München, 1987.

Jürgen Schmidhuber and Jieyu Zhao. Multi-agent learning with the success-story algorithm. In *Workshop on Learning in Distributed Artificial Intelligence Systems*, pp. 82–93. Springer, 1996.

Jürgen Schmidhuber, Jieyu Zhao, and Marco Wiering. Shifting inductive bias with success-story algorithm, adaptive levin search, and incremental self-improvement. *Machine Learning*, 28(1): 105–130, 1997.

Wenyi Wang, Piotr Piękos, Li Nanbo, Firas Laakom, Yimeng Chen, Mateusz Ostaszewski, Mingchen Zhuge, and Jürgen Schmidhuber. Huxley-Gödel machine: Human-level coding agent development by an approximation of the optimal self-improving machine. In *International Conference on Learning Representations (ICLR)*, 2026. URL https://openreview.net/forum?id=T0EiEuhOOL.

Sewall Wright. The roles of mutation, inbreeding, crossbreeding and selection in evolution. In *Proceedings of the Sixth International Congress of Genetics*, 1932.

Xunjian Yin, Xinyi Wang, Liangming Pan, Li Lin, Xiaojun Wan, and William Yang Wang. Gödel agent: A self-referential agent framework for recursively self-improvement. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 27890–27913, 2025.

Eric Zelikman, Eliana Lorch, Lester Mackey, and Adam Tauman Kalai. Self-taught optimizer (STOP): Recursively self-improving code generation. In *Conference on Language Modeling (COLM)*, 2024. URL https://openreview.net/forum?id=46Zgqo4QIU. Spotlight.

Jenny Zhang, Shengran Hu, Cong Lu, Robert Lange, and Jeff Clune. Darwin Gödel machine: Open-ended evolution of self-improving agents. In *International Conference on Learning Representations (ICLR)*, 2026a. URL https://openreview.net/forum?id=pUpzQZTvGY.

Jenny Zhang, Bingchen Zhao, Wannan Yang, Jakob Foerster, Jeff Clune, Minqi Jiang, Sam Devlin, and Tatiana Shavrina. Hyperagents. *arXiv preprint arXiv:2603.19461*, 2026b.

A CONSOLIDATION ANALYSIS OF FIGURE 1

Figure 1 illustrates the case in which all 15 agents generated during an exploration round receive zero task reward, yet consolidation of the same population produces a strong BabyAI agent. Here we analyze this case in detail.

A.1 REPEATED CONSOLIDATION FROM THE SAME POPULATION

To test whether the result depends on a single favorable consolidation sample and study the robustness of consolidation sampling, we repeat the task-consolidation step 10 times using exactly the same 15-node population evidence and the same starting state. The resulting BabyAI scores are

$$0.00,\ 0.04,\ 0.04,\ 0.76,\ 0.76,\ 0.80,\ 0.80,\ 0.80,\ 0.80,\ 0.80.$$

Thus, $7/10$ independent consolidations reach at least 76% success, $5/10$ reach 80%. This occurs despite every individual node in the population having task score 0.

The high-performing consolidations converge to the same qualitative solution. The key was not to identify a hidden high-performing node, none existed, but to combine complementary information expressed separately across unsuccessful nodes.

A.2 WHAT WAS PRESENT IN THE POPULATION?

The most informative evidence came from the behavioral summaries stored for each node.

Node 4 contained the most important evidence about the *action interface*. It was the only node whose outputs were consistently accepted by the environment. Its modification introduced the six valid action strings

$$\{\texttt{turn left}, \texttt{turn right}, \texttt{go forward}, \texttt{pick up}, \texttt{drop}, \texttt{toggle}\},$$

together with an extraction procedure that mapped free-form model responses to one of these actions.

However, Node 4 still scored zero because it was effectively blind to the environment state: it failed to incorporate the current observation into the task-agent prompt. Consequently, 178 of its 200 actions were simply `go forward`.

Node 12 exhibited the complementary failure. Its task agent correctly incorporated the current observation, previous action, and long-term textual context into the model prompt, but emitted an incompatible action vocabulary such as `north`, `south`, `east`, and `west`. Every one of its 200 outputs was therefore rejected.

The scalar reward consequently assigns the same value—zero—to two qualitatively different agents:

$$\underbrace{\text{valid actions but insufficient state information}}_{\text{Node 4}} \quad \text{and} \quad \underbrace{\text{state-aware reasoning but invalid actions}}_{\text{Node 12}}.$$

Neither capability is sufficient by itself, but together they contain most of what is required for successful behavior.

A.3 WHAT SUCCESSFUL CONSOLIDATION CONSTRUCTED

The successful consolidation sessions explicitly identified this complementarity and synthesized a task agent from several pieces of population evidence. Table 3 traces the main components of the resulting agent to their sources.

The crucial operation is therefore not selection but *completion*. Consolidation places Node 4's valid vocabulary and action parser underneath Node 12's observation-aware prompting pipeline. It then

Table 3: Main components synthesized by successful consolidation. The resulting agent is not copied from any single node: it combines complementary mechanisms and introduces additional logic inferred from population-wide failures.

Merged component	Function	Source
Valid BabyAI action set	Provides the six action strings accepted by the environment.	Node 4
Action extraction and normalization	Extracts an `ACTION: <name>` pattern, removes punctuation, and matches the result against the valid action set.	Node 4
Action-constrained system prompt	Instructs the model to output only a next action and suppresses conversational responses.	Primarily Node 4, informed by rules appearing in Node 12
Observation pipeline	Adds the current observation, previous action, and available long-term textual context to the task-agent prompt.	Node 12 and related nodes
Synonym translation	Maps rejected vocabulary observed elsewhere in the population—e.g., `north`, `pickup`, and `open`—onto valid environment actions.	Synthesized during consolidation from cross-node failure evidence
Additional parsing and tests	Adds fallback parsing, examples, and explicit tests of the normalization procedure.	Synthesized during consolidation

uses rejected actions observed elsewhere in the population to construct a translation layer between the vocabulary the language model naturally emits and the vocabulary accepted by the environment.

In compact form,

$$\underbrace{\text{Node 4}}_{\substack{\text{knows valid}\\ \text{action strings}}} + \underbrace{\text{Node 12}}_{\substack{\text{uses the}\\ \text{observation}}} + \underbrace{\text{population failures}}_{\substack{\text{reveal common}\\ \text{invalid synonyms}}} \longrightarrow \underbrace{\text{working agent}}_{\text{76–80\%}}.$$

The 80% consolidated agent succeeds on four of the five BabyAI task families (Table 4).

Table 4: Per-task success of the 80% consolidated agent.

Task	Success (%)
`goto`	100
`pickup`	100
`open`	100
`pick_up_seq_go_to`	100
`putnext`	0
Mean	80

This agent therefore contains a capability combination that was not instantiated by any member of the population from which it was produced.

A.4 Why Some Consolidations Still Fail

The unsuccessful consolidation samples are also informative. They typically identify Node 4 as the strongest source of evidence because it is the only node with consistently valid actions, but then treat the remaining zero-scoring nodes as globally uninformative. In particular, they discard Node 12's observation-processing logic together with its incorrect action vocabulary.

The resulting agent is consequently close to Node 4 itself: it knows which actions are valid, but does not reliably condition them on the current environment state. These consolidations obtain only 0–4% performance, with the weakest variants remaining near the original zero-reward behavior.

Across the broader consolidation analysis, the presence of the observation-processing block sharply separates successful and unsuccessful solutions: all consolidations that incorporated this capability reached 76–80%, whereas those that omitted it remained at or below 4%. The decisive step was therefore not simply identifying the highest-quality node, but separating a useful capability from the failure mode of the node containing it.

This example illustrates the distinction underlying population-level consolidation. Individual-node reward answers *'How well did this particular combination of mechanisms work?"* It does not directly answer *'Which mechanisms discovered across the population become useful when combined?"* In this case, the latter information was already present in the search state even though no individual agent received nonzero reward.

B Qualitative Analysis of Consolidation

Appendix A.1–A.4 showed how task consolidation built an 80% BabyAI agent out of a population in which every node scored zero. This appendix does the same for the other half of consolidation, the improver. We go through every accepted improver consolidation, two on BabyAI and two on Crafter, and show what the consolidator saw, what it changed, and why the change was accepted.

This section uses two conventions. First, "improver k" denotes the improver version first written by node k of the round. Other nodes may inherit this version unchanged, so a single improver version can have children under several nodes. Second, the *current* improver is the one carried by the round's best node. The consolidator's candidate replaces it only if it passes the test of Appendix C.5: both improvers are run five times from the same task agent, and we refer to these runs as *probes*. The candidate must match or exceed the current improver both in the fraction of probes that produce a valid change and in the mean score of the five resulting children.

B.1 BabyAI, Round 1: Letting the Improver See the Evaluation Results

Round 1 is the all-zero round of Figure 1: all 15 nodes scored zero. Three improver versions appeared during the round, written by nodes 2, 6 and 8. All three produced children with real patches, but since every child scored zero, child scores could not tell them apart. The consolidator therefore compared what each improver's code actually did (Table 5).

Table 5: The three improver versions available at the end of BabyAI round 1. With all child scores at zero, the consolidator chose by what each improver did and by how many children it produced.

Improver	Children	What it added to `meta_agent.py`	Decision
Improver 2	3 (nodes 7, 9, 15), all with non-empty patches; node 15 also inherited the improver itself	Before editing, reads `archive.jsonl` and `progress_info*` from the evaluation folder and pastes the result into its own instruction	**Adopted**
Improver 6	2	An instruction that names BabyAI and MiniHack and dictates an `ACTION: MOVE: NORTH` output format	Rejected: "domain-specific references would be rejected"
Improver 8	2	Tells the improver to edit only `task_agent.py`	Rejected: it "narrows" the improver, which must be able to modify any file, including itself

The consolidator's own summary of the decision:

> *"Adopt Improver 2's approach as the base. It has the most children (3), highest fecundity [i.e., patch rate], and its key innovation—reading actual evaluation results and feeding them back as context—is a procedural improvement that works across domains."*

What changed. The seed improver's entire instruction is one sentence, `Modify any part of the codebase at {repo_path}` (Appendix C.10). The new improver adds one step before every editing session: it opens the run's `progress_info` file and pastes its contents into its own instruction, for example

> *"Best scores: [0.0, 0.84, 0.84], Best lineage genids: [...]"*

followed by

> *"Analyze these results to identify failures and improve the agent codebase."*

It also states a goal for the first time: "to improve performance on the benchmark tasks." (The new code additionally tries to read a summary field from `archive.jsonl`, but that field does not exist, so this part always yields "No summary" and has no effect.)

Why it was accepted. The new improver and the seed improver were each run five times from the same task agent. The five children of the new improver scored 0.8, 0.4, 0.8, 0.6, 0.8, a mean of 0.68. The five children of the seed improver scored 0.0, 0.0, 0.2, 0.0, 0.0, a mean of 0.04. The only difference between the two improvers is whether the improver gets to see how the agent it is about to edit actually performed.

B.2 BabyAI, Round 2: Keeping the Mechanism, Removing the Benchmark from It

By the end of round 2, the best node (node 8) carried an improver that its own lineage had rewritten twice, first at node 1, then at node 4, and finally at node 8. This current improver already did something useful: it opened the evaluation report of the latest generation, read the BabyAI progression score and the list of rejected actions, and instructed the child to focus on improving the system prompt. However, every part of it was written for BabyAI. The report path, the field names, and the instructed focus all named the benchmark.

Unlike in round 1, the children of round 2 had non-zero scores, so the five improver versions that appeared during the round could be compared by what their children achieved (Table 6). Improver 1 stood out on every measure available to the consolidator: it had the most children, all five of them received a non-empty change, their mean score was the highest among the versions with more than one child, and four of the five had edited both the task agent and the improver, which indicates that the improver was steering its children toward substantive changes. Improver 2 produced children with comparable best scores but one clear failure, and improvers 6, 7 and 9 had only one child each, too few to judge.

Table 6: Improver versions with scored children at the end of BabyAI round 2. Each row is one improver version, named after the node that wrote it. Improver 1 was written by node 1 and inherited unchanged by node 4, so its children are children of either node.

Improver	Children	Child scores	Mean	Files the children edited
Improver 1	5	0.76, 0.76, 0.72, 0.80, 0.84	0.78	`meta_agent.py` and `task_agent.py` in 4 of 5
Improver 2	4	0.80, 0.80, 0.24, 0.80	0.66	`task_agent.py` only
Improver 6	1	0.80	0.80	—
Improver 7	1	0.80	0.80	—
Improver 9	1	0.72	0.72	—

The consolidator selected improver 1 as the base and then applied a strategy it described as domain generalization. It kept the mechanism, reading the evaluation report and placing the observed failures in front of the child, and removed everything in it that named the benchmark. Hard-coded evaluation paths were replaced by a generic search for the report file, benchmark-specific field names were replaced by generic ones, and the fixed instruction about what to focus on was replaced by an open-ended request to analyze the results. Table 7 lists the three substitutions.

Table 7: What the round-2 BabyAI improver consolidation changed. Every benchmark-specific path, field name, and focus area is replaced by a generic equivalent.

Removed	Replaced by
`os.path.join(eval_path, latest_gen, "balrog_babyai_eval", "report.json")`	`os.walk(eval_dir)` to find any `report.json`
`report['environments']['babyai']['progression_percentage']` and `failed_action_candidates`	`average_progress`, `standard_error`, plus any of `failed_tasks`, `missing_actions`, `error_summary`, `lowest_scores` if present
"Focus on: (1) improving the system prompt with . . . "	"Analyze these results to identify failures and improve the agent codebase."

Why it was accepted. Run five times from the same task agent, the new improver produced a valid change in 5 of 5 runs, and its children had a mean score of 0.88. The current improver produced a valid change in only 2 of 5 runs, and its children had a mean score of 0.28. Removing the benchmark from the improver therefore did not cost anything on the benchmark itself; the generalized version was both more reliable and produced better children.

Consequences. This is the improver that is later transferred to Crafter in Section 4.3. It contains no BabyAI-specific words at all, which is a direct result of this consolidation rather than of any processing at transfer time. Within BabyAI, the following round produced 9 of 15 children that scored above their 84% parent, three of them at 100%. HyperAgent's children of its own 84% parents never exceeded them.

B.3 Crafter, Rounds 2 and 3: Turning Observed Failures into Instructions

On Crafter, none of the six task consolidations changed the task agent. Each candidate either contained no real change or behaved identically to the current task agent on replayed evaluation steps, and was rejected by the checks of Appendix C.4. Consolidation therefore acted only through the improver. Two rounds show the pattern clearly.[2]

Table 8: Improver versions with scored children at the Crafter improver consolidations of rounds 2 and 3. Each row is one improver version, named after the node that wrote it; numbering restarts every round, so improver 3 of round 2 and improver 3 of round 3 are different improvers. In round 2, improver 3 produced a change in only half of its editing sessions; in round 3, every version produced a change in every session. For the round-3 versions other than improver 3, only the best child score was recorded.

Round	Improver	Children	Child scores	Mean	What the consolidator did with it
2	Improver 2	4	5.5, 4.5, 3.2, 7.3	5.1	Rejected
2	**Improver 3**	1	6.4	6.4	Adopted its procedure for reading the evaluation report
2	**Improver 7**	2	3.6, 3.2	3.4	Adopted its list of warnings about earlier mistakes
3	**Improver 3**	1	6.8	6.8	Adopted its whole instruction
3	Improver 13	—	best child 3.6	—	Rejected
3	Six other versions	—	all children ≤ 3.6	—	Rejected

Round 2. Three improver versions had scored children (Table 8). Improver 2 had the most children and the best single child, but its results were uneven. Improver 3 had produced only one child, and only half of its editing sessions had produced a change at all, but that one child scored second best.

[2]Crafter scores below are achievement-progression percentages, on the same scale as in the main text.

The two children of improver 7 were the weakest. Rather than adopt one improver, the consolidator took one section from each of the two weaker ones. From improver 3 it took a procedure for reading the evaluation report: open `report.json`, look at the list of rejected actions, identify patterns in them, track progression, and check for regressions. From improver 7 it took a list of explicit warnings, written from the mistakes earlier children had made, against synonym or fuzzy action matching and against verbose instructions. It added one principle of its own, that the task agent should validate its outputs and correct invalid ones. Run five times from the same task agent, the new improver's children had a mean score of 6.4, against 2.7 for those of the current improver.

Round 3. Eight improver versions appeared, and every one of them produced a change in every editing session, so reliability could not separate them and child scores were the only discriminator (Table 8). The child of improver 3 (node 9) scored 6.8, nearly double the next best, a child of improver 13 at 3.6. The consolidator adopted improver 3's instruction as a whole. It names the dominant failure, verbose conversational output instead of a single valid action, and gives an example of it; it prescribes a six-step structure for the task agent's prompt; and it lists a few principles for parsing actions, among them to sort action names by length before matching and to return `Noop` as the safe default. Run five times from the same task agent, the new improver's children had a mean score of 5.5, against 0.9 for those of the current improver.

The last of those principles is where a Crafter action name entered the improver (`Noop` appears three times in the improver by round 6). The consolidation prompt forbids benchmark-specific content, and the static check of Appendix C.5 records a fixed list of domain words for diagnosis, but action names are not on that list. The leak is small and did not affect acceptance, but it shows that the improver's domain generality is enforced by instruction and diagnostics, not by construction.

B.4 What the Cases Have in Common

- **The one task consolidation that produced a new agent (Appendix A.3) combined complementary halves of two nodes and wrote the glue between them.** What made this possible was the per-node record of rejected actions (`usage.txt`): rejection counts singled out the one node with a valid action vocabulary, and the verbatim rejected outputs supplied the keys of the synonym map.
- **The improver consolidations did not copy children's code; they turned children's outcomes into instructions**—which evaluation files to open, which field lists the failures, what to warn the next child against. In BabyAI round 2 the consolidator also removed the benchmark from a working mechanism so that it would generalize, and that generalized mechanism is what later transferred to Crafter.
- **Acceptance was decided by how the children of each improver performed, not by the score of the consolidated code itself.** In both accepted BabyAI consolidations the margin was large (0.68 vs 0.04; 0.88 vs 0.28). On Crafter it was roughly one achievement wide.

C Experimental Details

This appendix describes the implementation, the evaluation protocol, the acceptance tests used by consolidation, and the integrity controls used in all experiments. Looped-RSI and HyperAgent share the same evolutionary loop, the same task-agent and improver architecture, the same model backbone, the same evaluation harness, and the same integrity constraints. The only algorithmic difference is the periodic population-level consolidation introduced in Section 2.

C.1 Evolutionary Search

Each node contains a task agent and an improver. The task agent maps the current instruction, observation, and previous action to an environment action. The improver has shell and editor access to a copy of the agent codebase and is invoked once per expansion to propose a self-modification.

An *expansion* is one self-modification attempt. Given the current archive, a parent is sampled, its improver is run once, and the resulting child is checked and evaluated before being added to the archive. An attempt counts as one expansion even if it produces no change, generates invalid code,

violates the admissibility policy, crashes, or fails staged evaluation. We therefore use expansions as the unit of evolutionary budget.

For a valid node i with score s_i and c_i existing children, HyperAgent assigns parent-selection weight

$$w_i = \sigma(10(s_i - m)) \exp\left[-\left(\frac{c_i}{8}\right)^3\right],$$

where m is the mean score of the three highest-scoring nodes. Parents are sampled proportionally to w_i. The second factor discourages repeatedly expanding nodes that already have many children.

C.2 Looped-RSI and HyperAgent Budgets

Looped-RSI groups expansions into rounds of 15. Each round is followed by task-agent consolidation and improver consolidation. The accepted consolidated state is the starting point of the next round.

HyperAgent uses the identical evolutionary procedure but keeps expanding the search tree without consolidation.

To minimize variance from early exploration, the two methods share exactly the same first-round population of 15 nodes. The HyperAgent arm is initialized from this common archive while the underlying working copy is reset to the original seed snapshot. The methods therefore share the same initial agents and search history and diverge only after the first consolidation boundary.

The evolutionary budgets are:

	Looped-RSI	HyperAgent
BabyAI	$3 \times 15 = 45$	55
Crafter	$6 \times 15 = 90$	110

HyperAgent therefore receives more ordinary self-improvement expansions in both environments. Consolidation requires additional inference and evaluation compute and is not included in the expansion count.

C.3 Population Evidence

At each consolidation boundary, the consolidator receives evidence from all nodes generated during the current round, including unsuccessful ones. For each node v, we provide its ancestry and its modification, aggregate and per-episode task outcomes, behavioral evidence collected during evaluation, and the self-improvement trace that produced it.

Concretely, the evidence includes the change relative to the node's parent, the cumulative change relative to the round's starting point, complete copies of newly introduced files, response-validity statistics, frequently rejected outputs, and whether newly introduced functionality was actually invoked during evaluation. The self-improvement trace additionally records the failure mode targeted by the improver and its stated motivation for the change.

Following Section 2.1, we summarize the evidence associated with node v as

$$e_v = (\Delta_v, y_v, b_v, h_v),$$

and the population evidence for round r as

$$E_r = \{e_v \mid v \in \mathcal{A}_r\}.$$

Importantly, zero-scoring nodes remain part of E_r. Their changes and behavioral traces can therefore contribute to consolidation even though they are not eligible to be selected as parents.

For improver consolidation, nodes are additionally grouped by which improver version they carry. For each distinct improver version, the consolidator receives its code change, the children it generated, its patch rate, and the evaluation outcomes of those children. Improver quality is therefore represented through its children rather than through the task score of the node the improver lives in.

C.4 TASK CONSOLIDATION

Task consolidation starts from the reference agent, the round's best-scoring node, and may edit only task-side code. The consolidator may inspect the complete round evidence and may preserve, repair, complete, combine, or reject mechanisms discovered across different branches. Its proposal, the candidate, must pass the following three checks in order to replace reference agent's current task agent.

Validity. A candidate is rejected if it modifies protected improver-side code, produces no effective task-side change, or fails the integrity checks described in Appendix C.11. In particular, modifications to `meta_agent.py` or `run_meta_agent.py` during task consolidation are not permitted.

Behavioral change. Before running a full task evaluation, we test whether the candidate behaves any differently from the current task agent. At least ten stored evaluation exchanges from the current task agent are replayed through both agents while holding the language-model response fixed. This gives a deterministic behavioral fingerprint without additional language-model calls.

If both agents produce identical extracted actions on all replayed exchanges, the candidate changes nothing that matters and is rejected without further evaluation.

Task performance. All remaining candidates are evaluated using the same full evaluation protocol as ordinary evolutionary nodes. Let s_{cand} and s_{cur} denote the scores of the candidate and of the current task agent. The candidate is accepted when

$$s_{\text{cand}} > s_{\text{cur}},$$

or when

$$s_{\text{cand}} = s_{\text{cur}} > 0.$$

Otherwise, the current task agent is kept. A tie at a positive score is accepted because the synthesized agent may contain additional population-derived capabilities without any measured regression, whereas a tie at zero gives no positive evidence for replacement.

C.5 IMPROVER CONSOLIDATION

Improver consolidation is evaluated independently of task consolidation. The candidate improver and the current improver are compared from the same task-agent state, so that the current task capability does not confound the estimate of improver quality.

Static validity. A candidate improver is rejected if its source does not parse, exceeds 200 lines, loses unrestricted shell and editor access, removes the underlying language-model call, or removes the instruction-level framing required for continued self-improvement.

The resulting improver must retain `tools_available='all'` and a valid call to `chat_with_agent`. It must also preserve the self-improvement interface used during evolution. Domain-specific terms introduced into the improver are recorded for diagnosis but do not themselves determine acceptance.

Probe children. The candidate improver and the current improver are each run five times, under the same conditions used during evolution and starting from the same task agent. We call these runs *probes* and the agents they produce *probe children*.

A probe is *valid* if the run produces a non-empty change and the resulting codebase compiles. Each probe child then receives staged task evaluation. A failed or unevaluable probe contributes a child score of zero.

For an improver μ, *offspring validity* is the fraction of valid probes,

$$\widehat{V}_5(\mu) = \frac{1}{5}\sum_{i=1}^{5} \mathbf{1}\left[v_i^{(\mu)} \text{ is valid}\right],$$

and *offspring quality* is the mean probe-child score,

$$\widehat{Q}_5(\mu) = \frac{1}{5}\sum_{i=1}^{5} \widehat{J}\left(v_i^{(\mu)}\right).$$

A candidate improver is accepted only when

$$\widehat{V}_5(\mu_{\text{cand}}) \geq \widehat{V}_5(\mu_{\text{cur}}) \qquad \text{and} \qquad \widehat{Q}_5(\mu_{\text{cand}}) \geq \widehat{Q}_5(\mu_{\text{cur}}).$$

That is, the new improver must be at least as likely to produce a usable change, and at least as good at producing high-scoring children, as the improver it replaces.

Undoing an accepted improver later. Five probes may not reveal slower degradation, so an accepted improver remains reversible. The previously validated improver is restored if either the patch rate in the following round falls below 60% of the patch rate observed before the consolidation, or no child modifies the improver for two consecutive rounds. The first condition detects an improver that has stopped producing usable changes; the second detects an improver that has stopped modifying itself.

C.6 Discovery Environments

We evaluate in a deliberately sparse discovery setting. Agents receive the task objective but no action vocabulary, environment mechanics, demonstrations, or task-specific hints. Invalid outputs consume an environment step and leave the environment unchanged. The following observation only informs the agent that its previous action was invalid.

BabyAI. We use five task families from `BabyAI-MixedTrainLocal-v0`: `goto`, `pickup`, `open`, `putnext`, and `pick_up_seq_go_to`.

Each episode samples a fresh task instance and layout. We use `num_dists=0`. Consequently, evaluation varies both across the five task families and across independently sampled instances within each family.

The task agent receives only

```
You are an agent playing a simple navigation game.
Your goal is to {mission}.
```

where `{mission}` is the goal supplied by the environment.

Crafter. Crafter uses the single `default` environment. Every episode samples a fresh procedurally generated world with area 64×64 and a 9×9 local observation.

The task agent receives only

```
You are an agent playing a survival game.  Your goal
is to unlock as many achievements as possible.
```

Performance is the percentage of the 22 Crafter achievements unlocked, averaged over episodes.

C.7 Evaluation During Evolution

Candidate nodes use the same two-stage evaluation procedure under both Looped-RSI and HyperAgent.

	BabyAI	Crafter
Staged evaluation	1 episode/task (5 total)	1 episode
Promotion criterion	score > 0	score > 0
Full evaluation	5 episodes/task (25 total)	10 episodes
Maximum steps	40	50

For BabyAI, staged evaluation therefore executes one episode from each of the five task families. A positive staged score triggers full evaluation using five episodes per task family.

For Crafter, staged evaluation uses one freshly generated world, while full evaluation uses ten fresh worlds.

Nodes that fail staged evaluation remain in the archive but do not receive a full task evaluation and cannot be selected as parents. Their changes and traces nevertheless remain available to Looped-RSI as population evidence.

A node is considered fully evaluated only when its evaluation report contains at least 20 completed BabyAI episodes or at least 8 completed Crafter episodes.

C.8 Final Evaluation

Final agents are independently re-evaluated using larger episode and interaction budgets than those used during evolution.

For BabyAI, we evaluate on both fresh episodes from `BabyAI-MixedTrainLocal-v0` and the held-out `BabyAI-MixedTestLocal-v0` split. Both contain the same five task families. The held-out split is never used during evolutionary search.

Each BabyAI final evaluation uses ten episodes per task family, giving

$$5 \times 10 = 50$$

episodes per split.

Crafter has no separate held-out split. The benchmark consists of a single procedurally generated environment, so independently sampled episodes already correspond to fresh worlds. Final evaluation uses 20 fresh Crafter episodes.

	BabyAI	Crafter
In-distribution	`MixedTrainLocal`	Fresh worlds
OOD	`MixedTestLocal`	–
Episodes	10/task, 50/split	20
Step cap	128	128
Workers	10	10

The local BabyAI environment wrapper originally instantiated only `BabyAI-MixedTrainLocal-v0`. For the final held-out evaluation, we additionally allow it to instantiate `BabyAI-MixedTestLocal-v0`. This compatibility change is made only in the final-evaluation copy and does not modify observations, actions, rewards, or any part of the evolutionary runs.

C.9 Model and Inference Configuration

All task-agent, improver, consolidation, and probe calls use the same Qwen3.6-35B-A3B backbone served through a local vLLM OpenAI-compatible endpoint. Task-agent evaluation uses reasoning enabled with temperature 1.0, `top_p` $= 0.95$, `top_k` $= 20$, `min_p` $= 0.0$, presence penalty 0.0, repetition penalty 1.0, and a maximum output length of 16,384 tokens.

Improver sessions, consolidation sessions, and improver probes use the same configuration except that the temperature is 0.6.

Each environment step has a wall-clock limit of 900 seconds. On timeout, the step is recorded as failed and the task-agent process is restarted before the following step. Improver sessions and improver probes are capped at 90 minutes, while consolidation sessions are capped at 2 hours.

BabyAI experiments run in the `hyperagents` Docker image. Crafter is built on Python 3.10 with `crafter==1.8.3`.

C.10 PROMPT DESIGN

The task, improver, and consolidation prompts are fixed by the experimental protocol. Importantly, the consolidation prompts are host-side and excluded from the evolvable agent state. They describe how population evidence should be inspected and synthesized, but contain no domain-specific action names, environment mechanics, strategies, or task solutions. Any environment-specific knowledge available during consolidation must therefore originate from the population itself.

Task-agent prompt. The task-agent prompts are the bare goal descriptions given in Appendix C.6. No action vocabulary or environment hint is appended.

Seed improver. Both methods begin from the same minimal improver instruction:

```
Modify any part of the codebase at {repo_path}.  You
are the meta agent, implemented in `meta_agent.py`;
you are invoked once per generation, and your
modifications to this codebase persist to future
generations.  The repository also contains a task
agent (`task_agent.py`) that is evaluated on a
benchmark task.  Evaluation results of previously
generated agents can be found at {scores_path}.
```

The improver retains unrestricted shell and editor access through `tools_available='all'` and may modify either task-side or improver-side code.

Task consolidation. The task consolidator is told that it receives nodes generated from a common round starting point and is asked to construct a single coherent task agent from their evidence. It is instructed to inspect changes, per-episode outcomes, behavioral traces, newly introduced files, and self-improvement traces before editing the reference agent.

The prompt emphasizes three principles. First, measured outcomes should be preferred over plausibility when determining which changes helped. Second, capability should be distinguished from exploitation: a useful mechanism may appear in an unsuccessful node because that node failed to invoke or integrate it correctly. Third, consolidation should synthesize a coherent successor rather than mechanically merging all changes.

The consolidator is explicitly permitted to repair small implementation failures and complete partially realized mechanisms when supported by population evidence. It is prohibited from modifying improver-side or evaluation code.

Improver consolidation. The improver consolidator is explicitly told that an improver should not be judged by the task score of the node containing it, but by the children it produces. It receives child counts, patch rates, child scores, improver code changes, and summaries of the changes attempted by those children.

The consolidator may keep the current improver, adopt a supported mechanism from another improver, or combine compatible mechanisms. It is instructed to prefer minimal, evidence-supported changes rather than concatenating multiple improver instructions.

The resulting improver must remain domain-general. It may encode procedures for inspecting evaluation evidence, identifying failure modes, generating changes, and checking them, but may not encode benchmark-specific actions or environment knowledge. The improver is restricted to at most 200 lines.

Exact prompt templates are included in the released implementation.

C.11 EVALUATION INTEGRITY

Because the evolving system can modify its own code, task execution is isolated from evaluator and environment internals.

Evaluation harness code, environment implementations, consolidation logic, evolutionary drivers, run metadata, and previous-run outputs are excluded from the inherited agent state. The `domains/` directory is restored after every improver session and cannot become part of an evolved snapshot.

Before evaluation, newly introduced task-side code is checked for prohibited access to environment packages, evaluator internals, unsafe introspection, process-spawning interfaces, or unrestricted tool access. The task agent itself cannot enable `tools_available='all'`.

During task evaluation, the evolved agent executes in a separate process and communicates with the evaluator only through the observation/action interface. Imports exposing BALROG internals, the underlying environment implementations, simulator packages, and relevant process-control functionality are blocked at runtime.

These integrity controls are applied identically to Looped-RSI and HyperAgent.

D Full Prompts

This appendix reproduces the final prompts exactly as received by the model in the reported experiments. The task-agent instructions are taken from episode transcripts, the improver instruction from the first-generation session transcript, and the two consolidation prompts from the corresponding run artifacts. For the consolidation prompts, only the round index, number of nodes, champion identity, and measured scores vary across rounds. The instances below correspond to BabyAI run b2, round 2.

D.1 Task-Agent Prompts

BabyAI. The placeholder `{mission}` is replaced by the episode-specific goal sentence, for example, `put the blue ball next to the grey box`.

```
You are an agent playing a simple navigation game. Your goal is to
  ↪ {mission}.
```

Crafter. The task agent receives the following prompt:

```
You are an agent playing a survival game. Your goal is to unlock as
  ↪ many achievements as possible.
```

D.2 Improver Prompt

The improver is invoked once per generation. The following is the complete instruction received by the model at the first generation:

```
Modify any part of the codebase at `/hyperagents/`. You are the meta
  ↪ agent, implemented in `meta_agent.py`; you are invoked once per
  ↪ generation, and your modifications to this codebase persist to
  ↪ future generations. The repository also contains a task agent
  ↪ (`task_agent.py`) that is evaluated on a benchmark task.
  ↪ Evaluation results of previously generated agents can be found at
  ↪ `/tmp/generate_v3_lmrsi_b1_r1`.
```

The archive path is run-specific: `/tmp/generate_<run_id>` is the corresponding run archive, mounted read-only into the improver session.

D.3 Task Consolidation Prompt

The following is the complete Stage-T consolidation prompt for BabyAI run b2, round 2:

```
You are consolidating a population of self-improving agents.
```

```
Round 2 of evolution has finished. It produced 15 agent variants
("nodes"). Each node is composed of a meta_agent and a task_agent and
  ↪ is a
mutation of the same starting agent, and each was evaluated on the same
benchmark. Your job is to produce ONE task agent that combines what
  ↪ actually
worked, so the next round can build on it.

# What you are given

`/hyperagents/lmrsi/nodes/` - one directory per node, named
  ↪ `node_<gen>_<score>`:

```
info.json      generation id, parent, train score, per-unit score
  ↪ breakdown

own_step.diff  what THIS node changed relative to its parent

patch.diff     the node's cumulative change relative to the round's
  ↪ root.
               In these diffs, `task_agent.py` is the node's TASK
  ↪ AGENT (the
               agent that plays the benchmark) and `meta_agent.py` /
               `run_meta_agent.py` are its META AGENT (the improver
  ↪ that
               edits the codebase between generations).

new_files/     whole files this node CREATED, ready to read and to run
  ↪ -
               not just diff hunks. A newly invented capability lives
  ↪ here.

usage.txt      what this node's TASK AGENT actually DID when evaluated:
               response validity rates, a histogram of its outputs,
  ↪ and its
               most frequent REJECTED responses verbatim. Also which
  ↪ files
               its patch added (new capability) vs merely modified,
  ↪ and any
               code it added but never invokes.

self_evo.md    the tail of the META AGENT session that created this
  ↪ node,
               including its reasoning about which weakness it was
  ↪ trying
               to fix
```

`/hyperagents/lmrsi/root_info.json`   describes your working copy

`/hyperagents/`   YOUR WORKING COPY - edit this.

What to do

1. Explore the evidence before changing anything. Read several nodes'
  ↪ diffs,
   info.json and usage.txt. The rejected-responses lists show what the
  ↪ agent
   TRIED to do and what the environment refused - across the whole
  ↪ population
   this is a map of what has been learned about the environment so
  ↪ far, even by
   nodes that scored zero.
```

```
2. Work out which changes actually helped, using evidence, not
   ↪ plausibility:

   * scores here are measured on FEW episodes; a small score
   ↪ difference is noise

   * a change is convincing if a node succeeded on units YOUR WORKING
   ↪ COPY and
     other nodes failed (compare per-unit breakdowns in info.json) -
   ↪ importing
     such a capability is the single most valuable merge you can make

   * a zero-scoring node may still contain the round's most valuable
   ↪ idea,
     wrapped in a mistake - separate the two.

3. Separate CAPABILITY from EXPLOITATION. A node's score measures how
   ↪ well that
   node used a change - NOT whether the change is good. A node may
   ↪ have built
   something genuinely useful (a new input it reads, a new processing
   ↪ step, a
   new file) and still scored zero because it never wired the
   ↪ capability into
   its behaviour. usage.txt and `code added but never invoked` are how
   ↪ you tell.
   OPEN new_files entries AND READ THEM before deciding.

4. VERIFY rather than guess. You have a shell: run `python -c "..."`
   ↪ probes,
   parse a node's file, replay its extraction on an example response
   ↪ from
   usage.txt. Three outcomes, three responses:

   * it works and helped         -> carry it over

   * it works but was never used -> carry it over AND wire it in: make
   ↪ the
     agent actually read it / call it / include it where it helps. A
     capability the agent never invokes is dead code.

   * it is buggy                 -> fix it if the bug is small (a
   ↪ wrong index,
     a bad string split). If the idea is unsound or unverifiable,
   ↪ leave it
     out and say why.

   You are allowed - encouraged - to FINISH work rather than only
   ↪ select among
   finished work. You see the whole population at once, which no
   ↪ single node did.

5. Edit the working copy so `task_agent.py` (plus any helpers you
   ↪ keep) is a
   single coherent agent. Two nodes may implement conflicting versions
   ↪ of the
   same idea: pick the better one. Do not staple every patch together.

6. Leave out changes that are incoherent or that special-case
   ↪ individual test
   items. An untested change may simply be unmeasured; an
   ↪ item-specific hack is
   worthless whatever it scored.
```

```
The result is NOT the union of the population. Expect to reject a
  ↪ share of what
you are shown. The judgment on each change is yours - this document
  ↪ tells you
what the evidence can and cannot support, never what to conclude.

# Hard requirements

* The result MUST be a working agent: `TaskAgent.forward(inputs)` in
  `task_agent.py` returning `(prediction, message_history)`, same
  ↪ signature.

* Only modify task-side files (task_agent.py, utils/, files you
  ↪ create). Do NOT
  modify meta_agent.py, run_meta_agent.py, `domains/`, `lmrsi/`, or
  ↪ `outputs/`.

* No new dependencies; assume no network access.

* `python -c "import ast; ast.parse(open('task_agent.py').read())"`
  ↪ must pass.

* Do not weaken or bypass evaluation, and do not try to read benchmark
  ↪ internals.

Finish when the working copy holds the single best agent you can
  ↪ justify from
the evidence. Write a short summary of what you kept, finished, fixed
  ↪ and
rejected - and why - to `lmrsi/MERGE_REPORT.md`.
```

When no evolved node outperforms the round seed, only the description of the working copy changes. It is replaced by:

```
`/hyperagents/`                          YOUR WORKING COPY - edit this.
  ↪ It IS the round's starting
agent (score <seed score>): no node beat it this round.
```

D.4 Improver Consolidation Prompt

The following is the complete Stage-M consolidation prompt for BabyAI run b2, round 2:

```
You are consolidating the SELF-IMPROVEMENT side of an evolutionary run.

This codebase contains `meta_agent.py`: the improver that is invoked
  ↪ once per
generation to modify the codebase, and whose own code is inherited by
  ↪ the next
round. Round 2 ran several DISTINCT improver variants (each node's
meta code, inherited or self-modified). An improver cannot be judged
  ↪ by its own
task score - it must be judged by WHAT ITS CHILDREN ACHIEVED.

# What you are given

`/hyperagents/lmrsi/improvers/improver_<gen>/`:

```
info.json   whose meta code this is, how many children it produced,
               how many children carried real patches (patch rate), the
               children's scores, and probe results where available
```
```

```
meta_step.diff its meta-side change relative to the round root's improver

children.md    one line per child: what files it changed, its score, and a
               one-line summary of what it attempted
```

`/hyperagents/` - YOUR WORKING COPY. Its meta_agent.py is currently the round
champion's improver.

What to do

1. Read every improver's info.json and children.md. High child scores from ONE
   child are weak evidence; a consistent pattern (several children with real,
   scoring patches) is strong evidence. An improver whose sessions rarely
   produce a patch at all is defective regardless of its ideas.

2. Decide what the next round's improver should be: keep the champion's, adopt one
   variant's, or combine specific mechanisms (an instruction improvement from
   one, a context-gathering step from another). Combine only what you can see
   working in the children evidence - merged instructions have historically
   broken improvers, so prefer minimal coherent edits over concatenation.

3. Where the evidence shows which strategy produced improving children (see the
   nodes' self_evo traces), you may want to investigate what led to this
   improvement and encode that guidance into the improver's instruction.

4. Only modify meta-side files (meta_agent.py, run_meta_agent.py). You may
   create new helper files, but do not modify existing shared files.

5. The result must keep the improver runnable and able to modify any part of
   the codebase. `python -c "import ast; ast.parse(open('meta_agent.py').read())"`
   must pass.

6. Keep the improver GENERAL and SELF-IMPROVING (these are checked mechanically;
   violations are rejected outright):

   * it must keep `tools_available='all'` and the META_AGENT_TASK_CONTEXT sentence
     (the improver is invoked once per generation and may modify ANY file,
     including meta_agent.py itself -- do not narrow it to task_agent.py only);

   * do not hard-code the benchmark or domain (no task names, environment names,
     or domain-specific evaluation file names in the instruction): refer to the

```
    evaluation results through 'eval_path' and let the improver
  ↪ discover the
    domain from the evidence, so the same improver works on a new
  ↪ game;

  * encode PROCEDURE (how to read the evidence, how to find failure
  ↪ modes, how to
    make and check changes), not domain facts; keep meta_agent.py
  ↪ under 200 lines.

 Improvers are accepted only if their probe children are at least as
  ↪ fecund AND
 score at least as well as the current improver's, so prefer one
  ↪ well-supported
 change over many.
```