# KNOWLEDGE-CENTRIC SELF-IMPROVEMENT

**Anonymous authors**
Paper under double-blind review

## ABSTRACT

Self-improving AI systems typically treat the agent as the object that improves, by optimizing prompts, workflows, harnesses, or even the agent's own code. This agent-centric view can make improvements expensive to maintain and difficult to transfer, because gains become tied to a particular agent design, task distribution, or adaptation run. We study a complementary paradigm: knowledge-centric self-improvement, in which agents remain generic and disposable while the persistent object is a curated knowledge base that agents can leverage for future tasks. We conduct controlled case studies to operationalize this idea via a simple protocol. Agents attempt one task, then contribute evidence-grounded insights to a shared knowledge base via task-level and cross-task forums, followed by knowledge distillation. Because self-improvement is contained in the knowledge rather than the agent, improvement can be more inspectable, transferable, and portable. Across abstract reasoning, coding, and terminal benchmarks, this protocol improves solve rates while reducing dollar cost relative to agent-centric baselines. The resulting distilled knowledge also transfers to held-out tasks and across LLM families, indicating that the improvement is not merely an LLM- or run-specific behavior. These results support a new view of self-improving agentic systems: progress can be driven primarily by the curated persistent knowledge.

## 1 INTRODUCTION

The design of self-improving AI systems is rooted in a simple question: when the system gains experience through solving tasks, what should persist and improve over time? Most approaches implicitly answer this question by making the agent the persistent object of improvement, through prompt and workflow optimization (Fernando et al., 2024; Zhang et al., 2025d; Zhuge et al., 2024), harness search (Lee et al., 2026), or self-modification of the agent's underlying code (Zhang et al., 2025c; Wang et al., 2025b; Hu et al., 2025; Yin et al., 2025; Xia et al., 2025). This agent-centric view has produced strong systems, but it also makes improvement expensive to maintain. A single persistent agent must absorb many local lessons, some of which are task-specific, redundant, or mutually inconsistent; as the agent grows, useful behavior can be diluted by conflicting updates, and each new adaptation might degrade performance on the previous tasks.

We study a complementary paradigm in which the persistent object is a curated knowledge base rather than the agent itself (Figure 1). In this *knowledge-centric self-improvement* paradigm, agents are transient workers that read from and write to the shared knowledge. The knowledge base uses forum discussion to compare and reconcile seemingly conflicting opinions, and provides relevant guidance to future agents. Compared to agent-centric approaches, this shift can make self-improvement cheaper to maintain and easier to reuse: the system no longer has to preserve an increasingly specialized agent, but instead maintains an external knowledge base that can be inspected, condensed, and shared across fresh agents. Rather than coupling each improvement to a particular agent design or adaptation run, the system extracts reusable knowledge that can transfer across tasks and LLM families.

To isolate persistent curated knowledge as the improvement mechanism, we keep agents generic, stateless, and disposable. Each agent starts with a fresh context, receives the relevant distilled knowledge bundle, attempts one task with standard tool use, and contributes its experience back to the shared knowledge base. Agents do not carry private memory, role specialization, task-specific architecture, or custom orchestration. The only object that changes is the curated knowledge.

We instantiate this design with a deliberately simple three-stage curation protocol, organized as threaded discussions in which agents post claims, cite one another, and reply with supporting or



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.

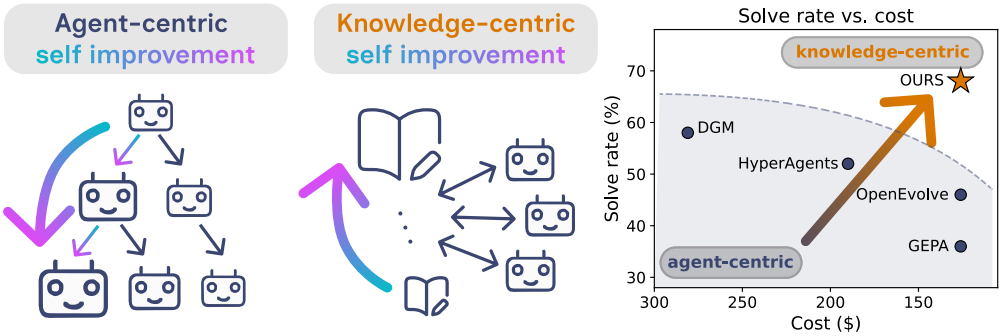



Figure 1: **From agent-centric to knowledge-centric self-improvement.** *(Left)* Conventional self-improving agentic systems treat the agent as the persistent substrate, evolving prompts, workflows, or agent code. *(Middle)* We invert this design. Agents are kept generic and disposable, while the persistent improving substrate is a shared knowledge base, curated through a simple protocol to generate and utilize distilled insights. *(Right)* Across our benchmarks, shifting from agent-centric to knowledge-centric self-improvement pushes the Pareto frontier forward.

challenging evidence. After task execution, each agent posts to a task-level forum, turning its attempt into evidence-grounded local claims about what worked, what failed, and which constraints or hypotheses mattered, which peer agents then support or challenge. A cross-task forum then brings these claims into open discussion across tasks: agents respond to peers' posts, and recurring principles, disagreements, and failure modes surface through this exchange. Finally, a distillation stage consolidates the surviving local and global claims into compact bundles that future fresh agents consume. We do not claim that this protocol is optimal; rather, it provides a controlled way to test whether curated knowledge alone can drive self-improvement when agent design is held fixed.

This paradigm goes beyond standard experience reuse or memory architectures (Packer et al., 2024; Zhong et al., 2023; Zhang et al., 2025a; Zhao et al., 2024; Wang et al., 2023; 2024; Tang et al., 2025) in what is stored and retrieved for future agents before they act. Those systems accumulate the agents' own successes and failures, whereas our knowledge curation protocol converts experience into deeper, more scoped, evidence-grounded guidance. We also treat disagreement as useful evidence rather than instability. Task-level forums preserve competing local hypotheses, cross-task forums test which ones recur or fail elsewhere, and distillation keeps only the claims that remain actionable and supported. Moreover, prior systems often still rely on persistent or specialized solving agents. In contrast, our agents are generic and disposable — the curated knowledge base is the only object that changes across tasks.

We evaluate on three categories of tasks: abstract reasoning (ARC-AGI-1 (Chollet, 2019), ARC-AGI-2 (de Oliveira et al., 2026)), coding (Polyglot (Gauthier, 2024), SWE-bench Pro (Deng et al., 2025)), and terminal skills (Terminal-Bench 2 (Merrill et al., 2026)). Our contributions are as follows:

- **We develop a controlled framework for studying knowledge-centric self-improvement.** We introduce a protocol in which the only improving object is a shared knowledge base, while agents are fixed as generic and disposable. This setup isolates curated knowledge as the source of improvement, and provides a testbed for studying knowledge-driven gains independently of agent design.
- **Knowledge curation outperforms agent-centric optimization with higher cost efficiency.** Compared against strong agent-centric self-improvement baselines, including Darwin Gödel Machine (DGM) (Zhang et al., 2025c), HyperAgents (Zhang et al., 2026a), and Meta-Harness (Lee et al., 2026), generic agents paired with a curated knowledge base achieve higher solve rates across the evaluated benchmarks, often at substantially lower dollar cost.
- **Curated knowledge transfers across tasks and across LLM families.** Distilled bundles curated on a training set improve zero-shot performance on a held-out test set, and bundles curated under one LLM family remain effective when consumed by another. This is evidence that curation yields an artifact whose utility extends beyond the run that generated it.

Taken together, these results highlight the advantage of curating an efficiently reusable knowledge artifact, rather than having improvements be coupled to the agent design.

| Evolving object | Representative approaches | Update mechanism | Agent-agnostic |
|---|---|---|---|
| Prompt | GEPA, OpenEvolve (Agrawal et al., 2025; Sharma, 2025) | Prompt evolution | ✗ |
| Harness | Meta-Harness (Lee et al., 2026) | Harness search | ✗ |
| Agent program | DGM, HyperAgents (Zhang et al., 2025c; 2026a) | Self-modification | ✗ |
| Experience† | ExpeL, ACE (Zhao et al., 2024; Zhang et al., 2025d) | Centralized curator | ✓, ✗ |
| Knowledge base | **Ours** | Peer discussion, distillation | ✓ |

Table 1: Self-improvement methods that differ in their "evolving object" and that we compare against in Section 4. †ExpeL and ACE reuse past experience but do not study self-improvement over generations; ExpeL is agent-agnostic, while ACE is not. Appendix D compares them with our method in detail. *Agent-agnostic*: task-solving instances can be discarded after each task without losing accumulated experience, which is retained externally and supplied to fresh instances; improvement needs no agent engineering, so generic and disposable agents are sufficient.

## 2 Related Work

We distinguish prior work along two axes: whether the persistent object is the agent, a memory store, or an external knowledge base; and whether past experience is stored, retrieved, optimized, or curated into transferable guidance. Table 1 places representative methods on both axes.

**Self-improving systems.** A large body of work on self-improving AI systems treats the agent as the object that improves over time. Iterative gains are achieved by modifying the agent's prompts (Assumpção et al., 2025; Fernando et al., 2024; Wang et al., 2025a; Ye et al., 2026; Zelikman et al., 2022; Zhang et al., 2025d), workflows (Zhuge et al., 2024), learned skill and tool libraries (Wu et al., 2024b), model weights (Akiba et al., 2025; Zhang et al., 2025e), harness search (Lee et al., 2026), or even the agent's underlying codebase (Hu et al., 2025; Wang et al., 2025b; Zhang et al., 2025c; 2026a; Xia et al., 2025; Yin et al., 2025). Despite their differences, these methods share a common premise: self-improvement is achieved by making the agent more capable through better self-feedback (Shinn et al., 2023; Madaan et al., 2023), stronger internal structure, or more sophisticated self-modification. Our work studies a different design dimension: we hold the agent fixed and ask whether improvement can compound when the only changing object is a curated knowledge base.

**Agent memory architectures and benchmarks.** A parallel line of work develops memory architectures that let agents persist and organize information across interactions. Building on retrieval augmentation (Lewis et al., 2020; Borgeaud et al., 2022), many systems study how to store, retrieve, link, and maintain memories so that agents can better leverage prior context (Park et al., 2023; Packer et al., 2024; Zhong et al., 2023; Xu et al., 2025; Fang et al., 2025). Recent work further improves the memory substrate itself, moving from flat retrieval toward structured, production-ready, or self-evolving memory systems (Chhikara et al., 2025; Zhang et al., 2026b; 2025b). Complementary benchmarks such as LoCoMo and LongMemEval evaluate whether agents can retain and reason over information across long conversational histories (Maharana et al., 2024; Wu et al., 2024a). These works are largely organized around the continuity problem: how an agent should retain, update, and retrieve useful context over long interactions, episodes, or task histories. We instead treat memory as storage infrastructure rather than the object of study, and make evidence-grounded knowledge curation the mechanism of improvement. Our central question is not how to remember more, but how raw attempts become scoped, transferable guidance that improves over generations while agents remain disposable.

**Experience reuse and knowledge transfer.** Prior work uses past experience to guide later decisions by learning reusable skills or workflows (Wang et al., 2023; 2024; Zheng et al., 2025), extracting lessons and guidelines (Zhao et al., 2024; Fu et al., 2024; Majumder et al., 2023), or developing reasoning templates and strategic principles (Yang et al., 2024b; Suzgun et al., 2025; Wu et al., 2026). ACE emphasizes incremental curation of a strategy playbook (Zhang et al., 2025d); G-Memory organizes and retrieves multi-agent interaction histories and insights (Zhang et al., 2025a); and Agent KB shares experience across agent frameworks (Tang et al., 2025). In scientific discovery, PiFlow uses principles to guide hypothesis generation and evidence gathering (Pu et al., 2025). Our distinction lies in how guidance is curated before reuse: agents review one another's claims against evidence. Supporting and conflicting evidence informs which claims are retained, revised, or limited to particular conditions. We hold agent configurations fixed to study improvement through this

curation process. We then freeze the resulting knowledge base and test whether its guidance transfers to unseen tasks and different LLM families.

## 3 KNOWLEDGE CURATION PROTOCOL

Our protocol curates evidence from task attempts into guidance for future agents. Each agent starts with a fresh context, and experience is carried forward through a shared knowledge base. We keep the model, tools, and task-solving instructions fixed across generations to study improvement through knowledge curation.

### 3.1 SYSTEM OVERVIEW

The framework is designed to isolate knowledge curation as the only mechanism of self-improvement, to enable controlled experimentation. Agents are transient instances: each receives a task description, standard tool access, and a seed bundle drawn from the shared knowledge base, attempts the task in a single solving session, and records its outcome. Benchmark-specific in-session submission budgets (e.g., ARC's two blind trials per test input) follow the official benchmark protocols and are listed in Appendix J. The agents do not inherit private memory, identity, specialization, or modified prompts from previous agents. All persistent state lives in the external knowledge base.

Once agents complete their attempts, the system curates the knowledge base through three stages, as depicted in Figure 2. The three stages form an abstraction ladder. Task-level forums preserve local evidence and discussion from individual attempts, cross-task forums test which claims survive debate across tasks, and distillation combines the surviving claims into knowledge bundles for the next generation.

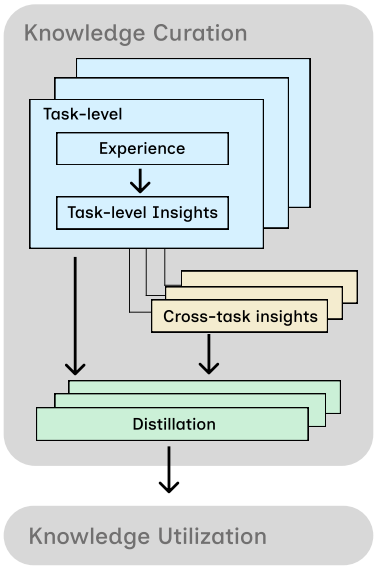



Figure 2: Knowledge curation protocol consisting of three stages. **Task-level forum** (agents propose task-level posts summarizing local outcomes and evidence on the current task), **Cross-task forum** (agents discuss transferable patterns and review peer posts with supporting or challenging evidence), and **Distillation** (forum posts are consolidated into typed task-level and cross-task bundles written back to the shared knowledge base). The schemas of the task-level posts, cross-task posts, and distilled bundles are listed in Appendix F.

**Stage 1: Task-level forum.** The task-level forum converts execution traces into task-specific insights through discussion on a per-task thread. After an attempt, the agent posts an assumption underlying its approach, evidence from the attempt, and a proposed change for the next attempt. Agents read earlier posts and cite evidence that supports or challenges their interpretation. This discussion records what was tried, what failed, and what remains to be checked. For ARC tasks, evidence may include a rejected transformation or a mismatch with a training example; for coding and terminal tasks, it may include an API constraint, a dependency issue, or a failed test (Figure 3). The resulting guidance is specific to the task.

**Stage 2: Cross-task forum.** The cross-task forum decides which local observations should survive beyond the task that produced them. Agents discuss task-level findings in a shared thread to assess their applicability beyond the source task. Claims must refer to a concrete operation, error type, invariant, or other task detail and cite evidence from task attempts. In later posts, agents are instructed to agree with, disagree with, or synthesize a cited claim. Supporting and conflicting evidence helps agents identify recurring observations and specify where guidance applies. The output is guidance proposed for reuse across tasks; its transfer to held-out tasks is evaluated in Section 4.4. Appendix C shows how disagreements led to revised guidance and informed later attempts.

**Stage 3: Distillation.** Distillation simply organizes the forum guidance into bundles for the next generation. An LLM retains actionable claims, their supporting evidence, and the conditions under which they apply. It produces a bundle for each unsolved task from that task's forum posts, and

a shared bundle from the cross-task posts. Both use the same format and are written back to the knowledge base. Each fresh agent receives its task's bundle and the shared bundle before attempting the task (Figure 3). This design enables controlled evaluation of two questions: first, whether curated knowledge increases task-solving performance, and second, whether the resulting artifact transfers beyond the tasks and LLM family that produced it. We describe the experimental setup and findings in Section 4.

# 4 EXPERIMENTS

We evaluate whether curated knowledge can drive self-improvement across real-world coding, terminal-execution, and abstract-reasoning tasks. We compare against the two predominant families of LLM-guided self-improvement: agent-centric systems (Zhang et al., 2026a; 2025c; Lee et al., 2026) and prompt optimization frameworks (Sharma, 2025; Agrawal et al., 2025).

Concretely, our experiments address four questions:

- **Does knowledge curation outperform agent-centric self-improvement?** (§ 4.1)
- **Does it outperform prompt optimization and experience reuse?** (§ 4.2)
- **Does the same protocol remain effective across different LLM families?** (§ 4.3)
- **Does the curated knowledge transfer to held-out tasks and different LLMs?** (§ 4.4)

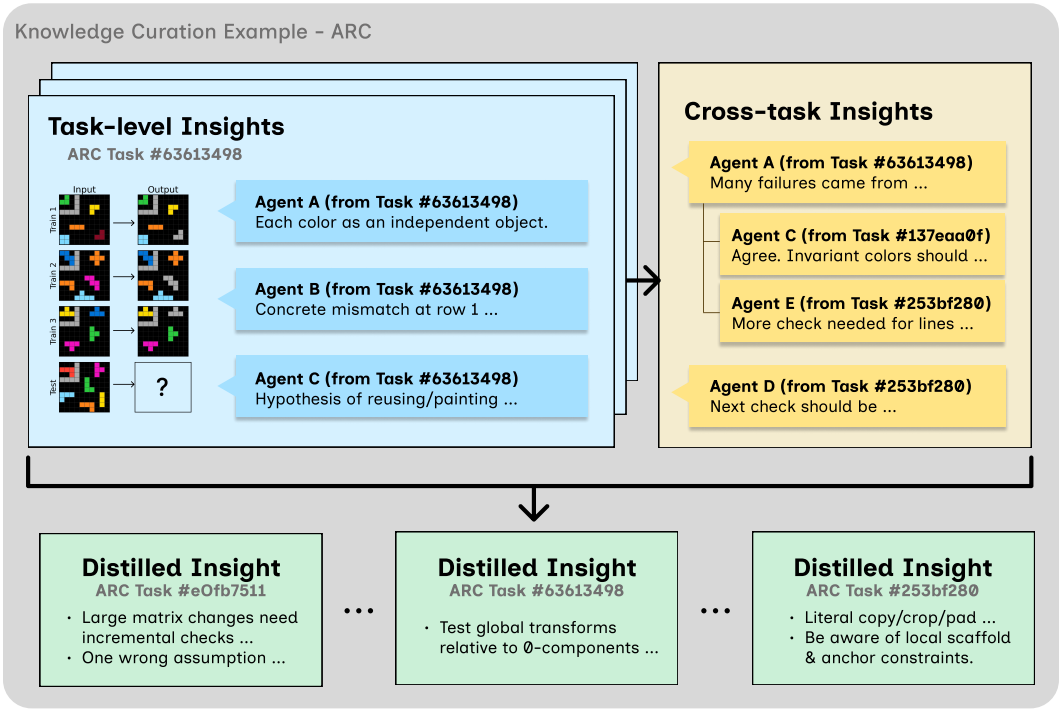
## 4.1 COMPARISON WITH AGENT-CENTRIC APPROACHES

**Benchmarks and protocol.** We evaluate on five benchmarks spanning coding, terminal skills, and abstract reasoning. A more in-depth discussion on benchmarks is presented in Appendix L. Polyglot (Gauthier, 2024) evaluates code generation and editing across C++, Go, Java, JavaScript, Python, and Rust, and we use the 50-task subset adopted by Darwin Gödel Machine (DGM) (Zhang et al., 2025c) and HyperAgents (Zhang et al., 2026a). SWE-bench Pro (Deng et al., 2025) evaluates realistic software-engineering agents on harder, less contaminated, and more diverse repository tasks, and from the 731 public instances we sample a 50-task subset uniformly at random with a fixed seed. ARC-AGI-1 and ARC-AGI-2 (Chollet, 2019; de Oliveira et al., 2026) evaluate abstract visual reasoning by requiring exact-grid reconstruction from a few input-output examples, and we sample 50 tasks from each benchmark and score them with the official exact-match protocol. Terminal-Bench 2 (Merrill et al., 2026) evaluates agents in 89 real terminal environments with containerized tasks, human-written reference solutions, and verification tests.

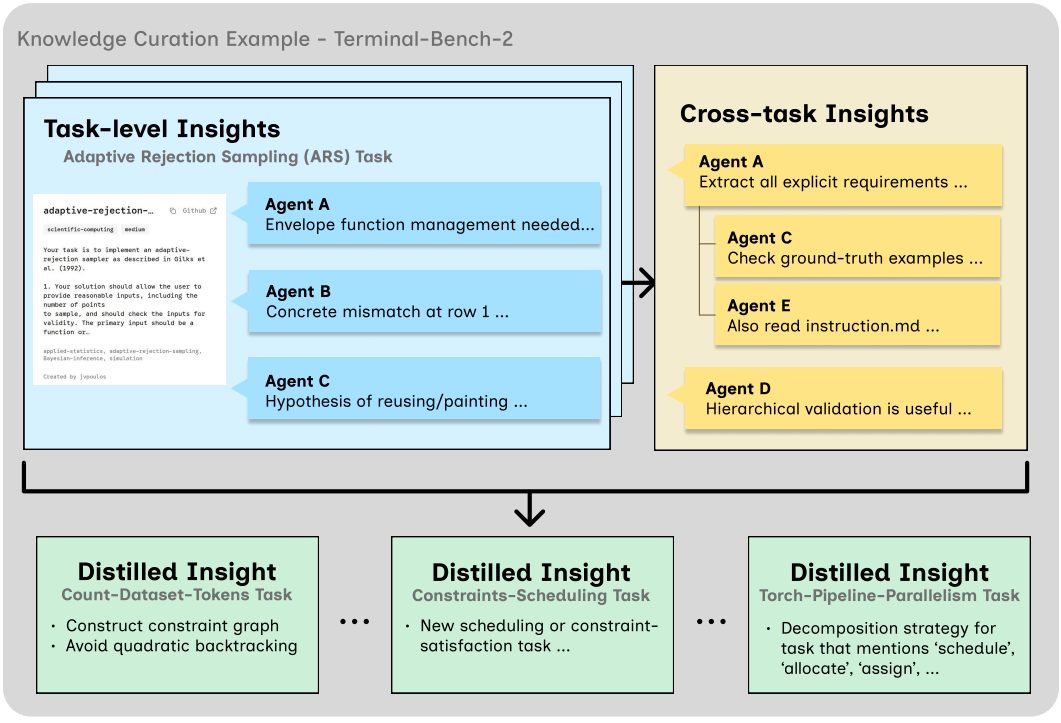
While our setup allows for asynchronous and decentralized systems, for practicality and direct comparability with agent-centric baselines, we implement a generation-based system. We begin with a pool of 50 tasks for each coding and abstract-reasoning benchmark. At each generation, we spawn one agent per unsolved task. Each agent receives the typed attempt table for its assigned task, the per-task distilled insights for that task, and the cross-task distilled insights from the global knowledge base. After execution, agents contribute back through the three-stage curation protocol of Section 3. Solved tasks are removed from the active pool, so the system does not spend tokens revisiting tasks already passed; in preliminary runs, a solved task was reliably solved again in later generations given its per-task bundle. We run 10 generations per benchmark and run our approach with three seeds.

**Agents and baselines.** Task-solving agents are implemented as off-the-shelf calls to the Anthropic Claude Agent SDK with Haiku 4.5 and the OpenAI Agents SDK with GPT-5.4-mini using medium reasoning; the forum and distillation phases of the curation protocol instead call the same LLMs through direct API loops without agent tooling. For coding and terminal benchmarks, agents receive the standard SDK tool set of file read/write, shell execution, and test invocation. Our system and the agent-centric baselines draw from this same tool set.

We compare against two representative agent-centric self-improving baselines, Darwin Gödel Machine (DGM) (Zhang et al., 2025c) and HyperAgents (Zhang et al., 2026a). Both frameworks improve performance through iterative refinement of agent behavior, prompts, strategies, or reasoning processes. For Terminal-Bench 2, we compare directly against strong reported systems including OpenHands (Wang et al., 2025c), Terminus 2 (Merrill et al., 2026), Mini-SWE-Agent (Yang et al., 2024a), Terminus-KIRA (KRAFTON AI & Ludo Robotics, 2026), Goose (Block, Inc., 2025), and Meta-Harness (Lee et al., 2026). These scores are the Haiku 4.5 results reported in Table 7 of




(a) Example of the knowledge curation pipeline applied to an Abstract Reasoning Corpus (ARC) task.




(b) Example of the knowledge curation pipeline applied to a Terminal-Bench 2 task.

Figure 3: Examples of knowledge curation on two task families. Local task evidence is first organized into task-specific guidance, then compared against cross-task patterns, and finally distilled into seed bundles that future agents can use. Detailed renderings of both examples are in Appendix B.

(a) Abstract Reasoning *(left)* and Coding *(right)* Benchmarks.

| | Abstract Reasoning | | | | Coding | | | |
|---|---|---|---|---|---|---|---|---|
| | ARC-AGI-1 | | ARC-AGI-2 | | Polyglot | | SWE-bench Pro | |
| Method | Solve ↑ | Cost ↓ | Solve ↑ | Cost ↓ | Solve ↑ | Cost ↓ | Solve ↑ | Cost ↓ |
| $\text{OURS}_{\text{haiku 4.5}}$ | **86.7%** ±4.2 | **$76** ±16 | **82.7%** ±6.1 | **$80** ±1 | **68.0%** ±2.0 | **$126** ±6 | **64.0%** ±2.0 | **$208** ±19 |
| $\text{HyperAgents}_{\text{haiku 4.5}}$ | 70% | $234 | 60% | $188 | 52% | $190 | 42% | $431 |
| $\text{DGM}_{\text{haiku 4.5}}$ | n/a | n/a | n/a | n/a | 58% | $281 | 54% | $713 |

(b) Terminal Skill Benchmark

| Method | Terminal-Bench 2 |
|---|---|
| OpenHands | 13.9% |
| Terminus 2 | 28.3% |
| Mini-SWE-Agent | 29.8% |
| Terminus-KIRA | 33.7% |
| Goose | 35.5% |
| Meta-Harness | 37.6% |
| OURS | **43.8%** ±3.4 |

Table 2: Haiku 4.5 comparisons. **(a)** Solve rates and costs over 10 generations. OURS entries are mean±standard deviation over three seeds; baselines are single runs. All baselines are rerun under our evaluation protocol. DGM is evaluated only on coding benchmarks (SWE-bench and Polyglot) in the original work and self-modifies its coding capabilities. **(b)** Terminal-Bench 2 baselines are reported Haiku 4.5 scores from Table 7 of Lee et al. (2026); Meta-Harness also uses an Opus 4.6 proposer.

the Meta-Harness paper (Lee et al., 2026); we did not rerun these systems. Meta-Harness score combines Claude Opus 4.6 as proposer with Haiku 4.5 as the base agent. These baselines improve the agent, execution scaffold, or orchestration loop, while our method instead keeps agents simple and externalizes improvement into a shared curated knowledge base.

**Results.** Table 2 shows that agents using our knowledge curation protocol achieve the highest solve rates among the listed Haiku-based methods on ARC-AGI-1, ARC-AGI-2, Polyglot, and SWE-bench Pro, while also using the lowest reported costs. Terminal-Bench 2 evaluates multistep workflows in terminal environments, requiring agents to coordinate command execution, respond to intermediate results, and produce a final state that passes task-specific tests (Merrill et al., 2026). On this benchmark, our framework reaches 43.8%, 6.2 points above Meta-Harness, the strongest listed system (Table 2b).

The qualitative Terminal-Bench examples in Figures 3b and 6 (see Figures 3a and 5 for ARC-AGI counterparts) illustrate the mechanism behind these gains: later attempts use recorded guidance about environment assumptions, missing artifacts, and incorrect workflows to avoid repeating earlier mistakes. Overall, these results suggest that knowledge-centric approaches can be highly cost-effective for self-improvement in agentic systems.

### 4.2 Comparison with Prompt Optimization and Experience Reuse

Other related approaches include those that rely on prompt optimization or more basic forms of experience reuse. The former optimizes a reusable solver prompt and can be viewed as a lightweight agent-centric variant, while the latter is similar to our setting but without peer-discussion curation.

**Setup.** For prompt optimization, we compare against GEPA (Agrawal et al., 2025) and OpenEvolve (Sharma, 2025) on ARC-AGI-1 and Polyglot, which provide representative abstract-reasoning and code-reasoning settings. GEPA and OpenEvolve optimize a reusable solver or coding-policy prompt over the same fixed train-50 task pool. We match the improvement budget to the realized cost of our main experiments on each benchmark (Table 2). Under these budgets, GEPA runs for 15 iterations on ARC-AGI-1 and 7 iterations on Polyglot, while OpenEvolve runs for 81 iterations on ARC-AGI-1 and 46 iterations on Polyglot.

For experience reuse, we compare against two representative baselines: ExpeL (Zhao et al., 2024) and ACE (Zhang et al., 2025d) (Appendix I). Because neither originally supports generational self-improvement, we adapt both into our iterative framework. Within this setup, the fundamental distinction lies in knowledge curation. First, neither baseline natively maintains per-task knowledge, so we provide both with the same per-task knowledge our agents receive. Second, their curation protocols rely on centralized curation rather than peer discussion, which is the exact mechanism we

Table 3: Prompt optimization and experience reuse baselines. GEPA and OpenEvolve: best score under a budget matched to our main runs. ExpeL and ACE: our 10-generation loop, with per-task knowledge added (Appendix I). OURS: mean±std over three seeds; baselines are single runs.

| Method | ARC-AGI-1 | Polyglot |
|---|---|---|
| OURS | **86.7%** ±4.2 | **68.0%** ±2.0 |
| *Prompt optimization* | | |
| GEPA | 44% | 36% |
| OpenEvolve | 54% | 46% |
| *Experience reuse* | | |
| ACE + per-task knowledge | 72% | 50% |
| ExpeL + per-task knowledge | 78% | 56% |

isolate. ExpeL uses a global curator to maintain an editable rule list and retrieve trajectories from similar tasks, while ACE reflects per attempt and relies on a global curator to maintain a playbook.

**Results.** Table 3 shows that knowledge curation outperforms both prompt-optimization and experience-reuse baselines. The comparison with prompt optimization further reinforces the value of knowledge-centric self-improvement. The prompt optimizer is essentially refining a reusable agent-centric prompt, whereas our method refines a reusable knowledge artifact that records distilled insights, environment assumptions, and recurring failure modes. The comparison with experience reuse highlights the value of our knowledge curation mechanism. Experience reuse methods use a single centralized curator, whereas our protocol curates knowledge through peer forum discussion. Our ablations support this: removing the per-task forum, the cross-task forum, or distillation each lowers the solve rate(Appendix N). As these baselines showed limited gains on ARC-AGI-1 and Polyglot, we did not extend them to the more resource-intensive benchmarks because of cost.

### 4.3 LLM Generalization

The previous experiments fix the LLM and vary the comparison method. We now reverse this and ask whether the same procedure remains effective when the LLM family changes.

**Setup.** To test whether the protocol is tied to a single LLM family, we run the same knowledge curation procedure with Haiku 4.5 and GPT-5.4-mini (medium reasoning) across the coding and abstract-reasoning benchmarks. Each run uses the same 50-task pools and 10-generation budget described above. Hyperparameters and task selection follow Appendix J and Appendix K.

**Results.** Table 4 shows that the protocol remains effective under both LLMs. GPT-5.4-mini attains the higher mean solve rate and lower cost on all four benchmarks, while the Haiku 4.5 runs still exceed every agent-centric baseline in Table 2 at a fraction of the cost. These results suggest that the curated knowledge protocol is not tied to a single LLM family.

Table 4: Cross-LLM Generalization. Solve rates and costs of our framework across different LLM families. GPT-5.4-mini runs use medium reasoning effort (Section 4.1).

| | **Abstract Reasoning** | | | | **Coding** | | | |
|---|---|---|---|---|---|---|---|---|
| | ARC-AGI-1 | | ARC-AGI-2 | | Polyglot | | SWE-bench Pro | |
| Method | Solve ↑ | Cost ↓ | Solve ↑ | Cost ↓ | Solve ↑ | Cost ↓ | Solve ↑ | Cost ↓ |
| $\text{OURS}_{\text{haiku 4.5}}$ | 86.7%±4.2 | $76±16 | 82.7%±6.1 | $80±1 | 68.0%±2.0 | $126±6 | 64.0%±2.0 | $208±19 |
| $\text{OURS}_{\text{gpt 5.4-mini}}$ | **93.3%** ±7.0 | **$16**±3 | **90.0%** ±5.3 | **$20**±1 | **72.7%** ±2.3 | **$56**±1 | **70.7%** ±2.3 | **$157**±11 |

### 4.4 Held-Out Knowledge Transfer

We now ask whether the final knowledge base is useful on its own: once frozen, does it help fresh agents on held-out tasks, including agents that use a different LLM?

**Setup.** The experiment has three steps.

1. **Curate.** A *donor* LLM (Haiku 4.5 or GPT-5.4-mini) runs the full 10-generation protocol on the 50-task self-improvement split (§ 4.1). We freeze the resulting generation-10 knowledge base, which holds cross-task heuristics, constraints, pitfalls, and validation strategies.

2. **Hold out.** For Polyglot and ARC-AGI-1, we build a held-out evaluation split that is disjoint from the self-improvement split. To keep this split difficult, we sample it only from tasks that both recipient LLMs fail without any knowledge (Appendix K).
3. **Solve.** A *recipient* LLM attempts each held-out task zero-shot. It first makes a single LLM call to explore the frozen knowledge base and select the entries relevant to the current task, then solves the task with that context in a single attempt. The recipient runs no forum, performs no distillation, and never updates the knowledge base.

For each recipient we compare three conditions: no knowledge (baseline), Haiku-curated knowledge, and GPT-curated knowledge. Every condition is run over three seeds, and we report mean±standard deviation. Because sampling is stochastic, the no-knowledge baseline re-solves a few of the selected tasks on reruns; this rerun score is the reference we compare against.

**Results.** *Curated knowledge helps on held-out tasks.* As shown in Table 5, the frozen knowledge base outperforms the no-knowledge baseline in all eight benchmark–donor–recipient combinations. With the stronger donor, solve rates rise by 1.9–3.5×: on Polyglot from 8.3% to 20.0% (GPT recipient) and from 3.3% to 11.7% (Haiku recipient); on ARC-AGI-1 from 23.3% to 43.3% (GPT) and from 13.3% to 28.3% (Haiku).

*The knowledge transfers across LLM families.* GPT-curated knowledge raises Haiku from 3.3% to 11.7% on Polyglot and from 13.3% to 28.3% on ARC-AGI-1; Haiku-curated knowledge raises GPT from 8.3% to 11.7% and from 23.3% to 38.3%. This evidence suggests that the knowledge base captures reusable task structure rather than habits specific to the LLM family.

Table 5: Knowledge Transfer Result. Zero-shot solve rate (%) on 20 held-out tasks when the curated knowledge artifact (i.e., the knowledge assets from our framework collected over the 10 generations shown in Tables 2a and 4) is reused on the test set. Darker shading indicates higher solve rate.

**Coding (Polyglot) Task**

| | Knowledge Donor | | |
|---|---|---|---|
| Recipient | None | GPT | Haiku |
| GPT | 8.3%±2.9 | **20.0%±5.0** | 11.7%±2.9 |
| Haiku | 3.3%±2.9 | 11.7%±2.9 | 11.7%±2.9 |

**Abstract Reasoning (ARC-AGI-1) Task**

| | Knowledge Donor | | |
|---|---|---|---|
| Recipient | None | GPT | Haiku |
| GPT | 23.3%±2.9 | **43.3%±2.9** | 38.3%±12.6 |
| Haiku | 13.3%±2.9 | 28.3%±2.9 | 23.3%±2.9 |

## 5 Conclusion

This work proposes a shift from agent-centric to knowledge-centric self-improvement, where the persistent substrate of AI progress is a shared, curated knowledge base rather than the agents themselves. With a three-stage protocol of task-level forums, cross-task forums, and distillation, stateless and intentionally simple agents outperform agent-centric self-improvement baselines. This suggests that the bottleneck may lie less in the complexity of the agent architecture than in the quality and structure of the knowledge it can access and refine. Because the knowledge lives outside any single LLM, it also points toward "plug-and-play" systems in which different LLMs and agent variants contribute to one inspectable body of reusable understanding; our transfer results across LLM families and held-out tasks are a first step in this direction.

Our experiments held agent architectures simple to isolate the effects of knowledge curation. Combining the protocol with recursive search or planning modules, scaling it to long-horizon tasks that need task decomposition, and including contributions from human experts are natural next steps toward reliable discovery systems. Our study also has limitations: we evaluate on fixed task subsets of moderate size with proprietary LLMs, and cost limited the baselines to single runs. The protocol also assumes that agents contribute in good faith, so a wrong but confident claim that no peer contests could persist in the knowledge base. Testing robustness to such harmful knowledge, for example by injecting false claims and tracking whether later generations prune them, is an interesting extension.

### AI USE STATEMENT

In this work, we used generative AI tools for limited assistance with Python code implementation and manuscript editing. We reviewed all AI-assisted work, checked the resulting code for correctness, and reviewed suggested wording for accuracy. We take responsibility for the final content of this work, including all text, claims, and artifacts produced with the aid of generative AI.

### ETHICS STATEMENT

We have reviewed the Code of Ethics and considered its applicability to this work. We have identified no ethical concerns or violations of the code arising from the methods or findings presented.

### REPRODUCIBILITY STATEMENT

We support reproducibility through detailed documentation of the method and experimental procedures. Section 3 describes the knowledge curation protocol, complemented by benchmark-specific prompts and artifact schemas in Appendices E and F. The appendix documents the execution lifecycle and tool access (Appendices A and G), benchmark definitions and task subsets (Appendices L), and hyperparameters and execution budgets (Appendix J). We also describe baseline adaptations and cost accounting (Appendices I and H), together with the frozen-knowledge transfer procedure and adapter prompts (Appendix M). These descriptions support independent implementation of the protocol and examination of the evaluation settings. We will release our code upon publication.

## REFERENCES

Lakshya A Agrawal, Shangyin Tan, Dilara Soylu, Noah Ziems, Rishi Khare, Krista Opsahl-Ong, Arnav Singhvi, Herumb Shandilya, Michael J Ryan, Meng Jiang, Christopher Potts, Koushik Sen, Alexandros G. Dimakis, Ion Stoica, Dan Klein, Matei Zaharia, and Omar Khattab. Gepa: Reflective prompt evolution can outperform reinforcement learning. *arXiv preprint arXiv:2507.19457*, 2025.

Takuya Akiba, Makoto Shing, Yujin Tang, Qi Sun, and David Ha. Evolutionary Optimization of Model Merging Recipes. *Nature Machine Intelligence*, 7(2):195–204, 2025. ISSN 2522-5839. doi: 10.1038/s42256-024-00975-8.

Henrique Assumpção, Diego Ferreira, Leandro Campos, and Fabricio Murai. CodeEvolve: An open source evolutionary coding agent for algorithmic discovery and optimization. *arXiv preprint arXiv:2510.14150*, 2025.

Block, Inc. goose: An open source, extensible ai agent, 2025. URL https://github.com/block/goose.

Sebastian Borgeaud, Arthur Mensch, Jordan Hoffmann, Trevor Cai, Eliza Rutherford, Katie Millican, George van den Driessche, Jean-Baptiste Lespiau, Bogdan Damoc, Aidan Clark, Diego de Las Casas, Aurelia Guy, Jacob Menick, Roman Ring, Tom Hennigan, Saffron Huang, Loren Maggiore, Chris Jones, Albin Cassirer, Andy Brock, Michela Paganini, Geoffrey Irving, Oriol Vinyals, Simon Osindero, Karen Simonyan, Jack W. Rae, Erich Elsen, and Laurent Sifre. Improving language models by retrieving from trillions of tokens. *arXiv preprint arXiv:2112.04426*, 2022.

Prateek Chhikara, Dev Khant, Saket Aryan, Taranjeet Singh, and Deshraj Yadav. Mem0: Building production-ready ai agents with scalable long-term memory. *arXiv preprint arXiv:2504.19413*, 2025.

François Chollet. On the measure of intelligence. *arXiv preprint arXiv:1911.01547*, 2019.

Wallyson Lemes de Oliveira, Mekhron Bobokhonov, Matteo Caorsi, Aldo Podestà, Gabriele Beltramo, Luca Crosato, Matteo Bonotto, Federica Cecchetto, Hadrien Espic, Dan Titus Salajan, Stefan Taga, Luca Pana, and Joe Carthy. Arc-agi-2 technical report. *arXiv preprint arXiv:2603.06590*, 2026.

Xiang Deng, Jeff Da, Edwin Pan, Yannis Yiming He, Charles Ide, Kanak Garg, Niklas Lauffer, Andrew Park, Nitin Pasari, Chetan Rane, Karmini Sampath, Maya Krishnan, Srivatsa Kundurthy, Sean Hendryx, Zifan Wang, Vijay Bharadwaj, Jeff Holm, Raja Aluri, Chen Bo Calvin Zhang, Noah Jacobson, Bing Liu, and Brad Kenstler. Swe-bench pro: Can ai agents solve long-horizon software engineering tasks? *arXiv preprint arXiv:2509.16941*, 2025.

Runnan Fang, Yuan Liang, Xiaobin Wang, Jialong Wu, Shuofei Qiao, Pengjun Xie, Fei Huang, Huajun Chen, and Ningyu Zhang. Memp: Exploring agent procedural memory. *arXiv preprint arXiv:2508.06433*, 2025.

Chrisantha Fernando, Dylan Banarse, Henryk Michalewski, Simon Osindero, and Tim Rocktäschel. Promptbreeder: Self-referential self-improvement via prompt evolution. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *ICML'24*, pp. 13481–13544. JMLR.org, 2024.

Yao Fu, Dong-Ki Kim, Jaekyeom Kim, Sungryull Sohn, Lajanugen Logeswaran, Kyunghoon Bae, and Honglak Lee. Autoguide: Automated generation and selection of context-aware guidelines for large language model agents. *arXiv preprint arXiv:2403.08978*, 2024.

Paul Gauthier. o1 tops aider's new polyglot leaderboard. https://aider.chat/2024/12/21/polyglot.html, December 2024. Accessed: 2026-01-28.

Shengran Hu, Cong Lu, and Jeff Clune. Automated design of agentic systems. *arXiv preprint arXiv:2408.08435*, 2025.

KRAFTON AI and Ludo Robotics. Terminus-KIRA: Boosting frontier model performance on terminal-bench with minimal harness, 2026. URL https://github.com/krafton-ai/KIRA.

Yoonho Lee, Roshen Nair, Qizheng Zhang, Kangwook Lee, Omar Khattab, and Chelsea Finn. Meta-harness: End-to-end optimization of model harnesses. *arXiv preprint arXiv:2603.28052*, 2026.

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. Retrieval-augmented generation for knowledge-intensive nlp tasks. *arXiv preprint arXiv:2005.11401*, 2020.

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. *arXiv preprint arXiv:2303.17651*, 2023.

Adyasha Maharana, Dong-Ho Lee, Sergey Tulyakov, Mohit Bansal, Francesco Barbieri, and Yuwei Fang. Evaluating very long-term conversational memory of llm agents. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 13851–13870. Association for Computational Linguistics, 2024.

Bodhisattwa Prasad Majumder, Bhavana Dalvi Mishra, Peter Jansen, Oyvind Tafjord, Niket Tandon, Li Zhang, Chris Callison-Burch, and Peter Clark. Clin: A continually learning language agent for rapid task adaptation and generalization. *arXiv preprint arXiv:2310.10134*, 2023.

Mike A. Merrill, Alexander G. Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Yeon Shin, Thomas Walshe, E. Kelly Buchanan, Junhong Shen, Guanghao Ye, Haowei Lin, Jason Poulos, Maoyu Wang, Marianna Nezhurina, Jenia Jitsev, Di Lu, Orfeas Menis Mastromichalakis, Zhiwei Xu, Zizhao Chen, Yue Liu, Robert Zhang, Leon Liangyu Chen, Anurag Kashyap, Jan-Lucas Uslu, Jeffrey Li, Jianbo Wu, Minghao Yan, Song Bian, Vedang Sharma, Ke Sun, Steven Dillmann, Akshay Anand, Andrew Lanpouthakoun, Bardia Koopah, Changran Hu, Etash Guha, Gabriel H. S. Dreiman, Jiacheng Zhu, Karl Krauth, Li Zhong, Niklas Muennighoff, Robert Amanfu, Shangyin Tan, Shreyas Pimpalgaonkar, Tushar Aggarwal, Xiangning Lin, Xin Lan, Xuandong Zhao, Yiqing Liang, Yuanli Wang, Zilong Wang, Changzhi Zhou, David Heineman, Hange Liu, Harsh Trivedi, John Yang, Junhong Lin, Manish Shetty, Michael Yang, Nabil Omi,

Negin Raoof, Shanda Li, Terry Yue Zhuo, Wuwei Lin, Yiwei Dai, Yuxin Wang, Wenhao Chai, Shang Zhou, Dariush Wahdany, Ziyu She, Jiaming Hu, Zhikang Dong, Yuxuan Zhu, Sasha Cui, Ahson Saiyed, Arinbjörn Kolbeinsson, Jesse Hu, Christopher Michael Rytting, Ryan Marten, Yixin Wang, Alex Dimakis, Andy Konwinski, and Ludwig Schmidt. Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces. *arXiv preprint arXiv:2601.11868*, 2026.

Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco J. R. Ruiz, Abbas Mehrabian, M. Pawan Kumar, Abigail See, Swarat Chaudhuri, George Holland, Alex Davies, Sebastian Nowozin, Pushmeet Kohli, and Matej Balog. Alphaevolve: A coding agent for scientific and algorithmic discovery. *arXiv preprint arXiv:2506.13131*, 2025.

Charles Packer, Sarah Wooders, Kevin Lin, Vivian Fang, Shishir G. Patil, Ion Stoica, and Joseph E. Gonzalez. Memgpt: Towards llms as operating systems. *arXiv preprint arXiv:2310.08560*, 2024.

Joon Sung Park, Joseph C. O'Brien, Carrie J. Cai, Meredith Ringel Morris, Percy Liang, and Michael S. Bernstein. Generative agents: Interactive simulacra of human behavior. *arXiv preprint arXiv:2304.03442*, 2023.

Yingming Pu, Tao Lin, and Hongyu Chen. PiFlow: Principle-Aware Scientific Discovery with Multi-Agent Collaboration. *arXiv preprint arXiv:2505.15047*, May 2025.

Asankhaya Sharma. Openevolve: an open-source evolutionary coding agent, 2025. URL https://github.com/algorithmicsuperintelligence/openevolve.

Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. *arXiv preprint arXiv:2303.11366*, 2023.

Mirac Suzgun, Mert Yuksekgonul, Federico Bianchi, Dan Jurafsky, and James Zou. Dynamic cheatsheet: Test-time learning with adaptive memory. *arXiv preprint arXiv:2504.07952*, 2025.

Xiangru Tang, Tianrui Qin, Tianhao Peng, Ziyang Zhou, Daniel Shao, Tingting Du, Xinming Wei, Peng Xia, Fang Wu, He Zhu, Ge Zhang, Jiaheng Liu, Xingyao Wang, Sirui Hong, Chenglin Wu, Hao Cheng, Chi Wang, and Wangchunshu Zhou. Agent kb: Leveraging cross-domain experience for agentic problem solving. *arXiv preprint arXiv:2507.06229*, 2025.

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*, October 2023.

Jianyu Wang, Zhiqiang Hu, and Lidong Bing. Evolving Prompts In-Context: An Open-ended, Self-replicating Perspective. In *Forty-Second International Conference on Machine Learning*, June 2025a.

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. *arXiv preprint arXiv:2510.21614*, 2025b.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. Openhands: An open platform for ai software developers as generalist agents. *arXiv preprint arXiv:2407.16741*, 2025c.

Zora Zhiruo Wang, Jiayuan Mao, Daniel Fried, and Graham Neubig. Agent workflow memory. *arXiv preprint arXiv:2409.07429*, 2024.

Di Wu, Hongwei Wang, Wenhao Yu, Yuwei Zhang, Kai-Wei Chang, and Dong Yu. Longmemeval: Benchmarking chat assistants on long-term interactive memory. *arXiv preprint arXiv:2410.10813*, 2024a.

Rong Wu, Xiaoman Wang, Jianbiao Mei, Pinlong Cai, Daocheng Fu, Cheng Yang, Licheng Wen, Xuemeng Yang, Yufan Shen, Yuxin Wang, and Botian Shi. Evolver: Self-evolving llm agents through an experience-driven lifecycle. *arXiv preprint arXiv:2510.16079*, 2026.

Zhiyong Wu, Chengcheng Han, Zichen Ding, Zhenmin Weng, Zhoumianze Liu, Shunyu Yao, Tao Yu, and Lingpeng Kong. OS-Copilot: Towards Generalist Computer Agents with Self-Improvement, February 2024b.

Chunqiu Steven Xia, Zhe Wang, Yan Yang, Yuxiang Wei, and Lingming Zhang. Live-SWE-agent: Can Software Engineering Agents Self-Evolve on the Fly?, November 2025.

Wujiang Xu, Zujie Liang, Kai Mei, Hang Gao, Juntao Tan, and Yongfeng Zhang. A-mem: Agentic memory for llm agents. *arXiv preprint arXiv:2502.12110*, 2025.

John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik R Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024a.

Ling Yang, Zhaochen Yu, Tianjun Zhang, Shiyi Cao, Minkai Xu, Wentao Zhang, Joseph E. Gonzalez, and Bin Cui. Buffer of thoughts: Thought-augmented reasoning with large language models. *arXiv preprint arXiv:2406.04271*, 2024b.

Haoran Ye, Xuning He, Vincent Arak, Haonan Dong, and Guojie Song. Meta Context Engineering via Agentic Skill Evolution. *arXiv preprint arXiv:2601.21557*, February 2026.

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 Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar (eds.), *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 27890–27913, Vienna, Austria, July 2025. Association for Computational Linguistics. doi: 10.18653/v1/2025.acl-long.1354.

Eric Zelikman, Yuhuai Wu, Jesse Mu, and Noah Goodman. STaR: Bootstrapping Reasoning With Reasoning. In *Advances in Neural Information Processing Systems*, volume 35, pp. 15476–15488, October 2022.

Guibin Zhang, Muxin Fu, Guancheng Wan, Miao Yu, Kun Wang, and Shuicheng Yan. G-memory: Tracing hierarchical memory for multi-agent systems. *arXiv preprint arXiv:2506.07398*, 2025a.

Guibin Zhang, Haotian Ren, Chong Zhan, Zhenhong Zhou, Junhao Wang, He Zhu, Wangchunshu Zhou, and Shuicheng Yan. MemEvolve: Meta-Evolution of Agent Memory Systems. *arXiv preprint arXiv:2512.18746*, December 2025b.

Jenny Zhang, Shengran Hu, Cong Lu, Robert Lange, and Jeff Clune. Darwin godel machine: Open-ended evolution of self-improving agents. *arXiv preprint arXiv:2505.22954*, 2025c.

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

Qizheng Zhang, Changran Hu, Shubhangi Upasani, Boyuan Ma, Fenglu Hong, Vamsidhar Kamanuru, Jay Rainton, Chen Wu, Mengmeng Ji, Hanchen Li, Urmish Thakker, James Zou, and Kunle Olukotun. Agentic context engineering: Evolving contexts for self-improving language models. *arXiv preprint arXiv:2510.04618*, 2025d.

Shengtao Zhang, Jiaqian Wang, Ruiwen Zhou, Junwei Liao, Yuchen Feng, Zhuo Li, Yujie Zheng, Weinan Zhang, Ying Wen, Zhiyu Li, Feiyu Xiong, Yutao Qi, Bo Tang, and Muning Wen. Memrl: Self-evolving agents via runtime reinforcement learning on episodic memory. *arXiv preprint arXiv:2601.03192*, 2026b.

Yiqun Zhang, Peng Ye, Xiaocui Yang, Shi Feng, Shufei Zhang, Lei Bai, Wanli Ouyang, and Shuyue Hu. Nature-Inspired Population-Based Evolution of Large Language Models. *arXiv preprint arXiv:2503.01155*, March 2025e.

Andrew Zhao, Daniel Huang, Quentin Xu, Matthieu Lin, Yong-Jin Liu, and Gao Huang. Expel: Llm agents are experiential learners. *arXiv preprint arXiv:2308.10144*, 2024.

Boyuan Zheng, Michael Y. Fatemi, Xiaolong Jin, Zora Zhiruo Wang, Apurva Gandhi, Yueqi Song, Yu Gu, Jayanth Srinivasa, Gaowen Liu, Graham Neubig, and Yu Su. Skillweaver: Web agents can self-improve by discovering and honing skills. *arXiv preprint arXiv:2504.07079*, 2025.

Wanjun Zhong, Lianghong Guo, Qiqi Gao, He Ye, and Yanlin Wang. Memorybank: Enhancing large language models with long-term memory. *arXiv preprint arXiv:2305.10250*, 2023.

Mingchen Zhuge, Wenyi Wang, Louis Kirsch, Francesco Faccio, Dmitrii Khizbullin, and Jürgen Schmidhuber. GPTSwarm: Language Agents as Optimizable Graphs. In *Forty-First International Conference on Machine Learning*, June 2024.

## APPENDIX CONTENTS

A. Attempt Runtime Lifecycle . . . 15
B. Curated Knowledge Examples . . . 16
C. Disagreement as Evidence: Worked Examples . . . 18
D. Comparison with Experience-Reuse Methods . . . 20
E. Prompts by Benchmark . . . 20
F. Protocol Schemas . . . 25
G. Tool Surfaces by Benchmark . . . 27
H. Cost Accounting . . . 28
I. Baseline Setup . . . 28
J. Hyperparameters . . . 29
K. Task Selection . . . 29
L. Benchmark Details . . . 29
M. Knowledge Transfer . . . 30
N. Protocol Ablations . . . 32

## A ATTEMPT RUNTIME LIFECYCLE

Figure 4 shows a simplified view of how a single task attempt flows through the runtime. The Engine dispatches the task to a Host launcher, which spawns a transient Worker container; the agent-runner inside the container drives a provider query loop with MCP-served tools, and the JSON status envelope returned through the Host is normalized before the Engine scores the output and persists results.

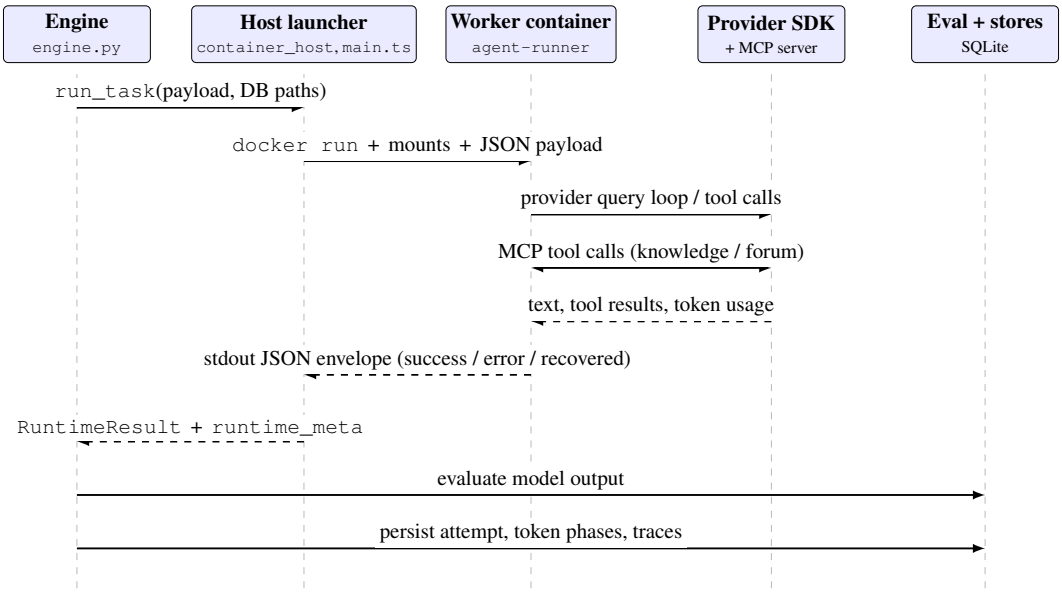



Figure 4: Simplified attempt runtime lifecycle. Solid arrows are forward calls, dashed arrows are returns, and the bidirectional arrow denotes in-loop MCP tool exchange. Between the container envelope and the `RuntimeResult`, the Host parses the envelope and normalizes outcomes (e.g., reclassifying empty success-like envelopes as `silent_failure`) so that they are not scored as false zeros.

## B CURATED KNOWLEDGE EXAMPLES

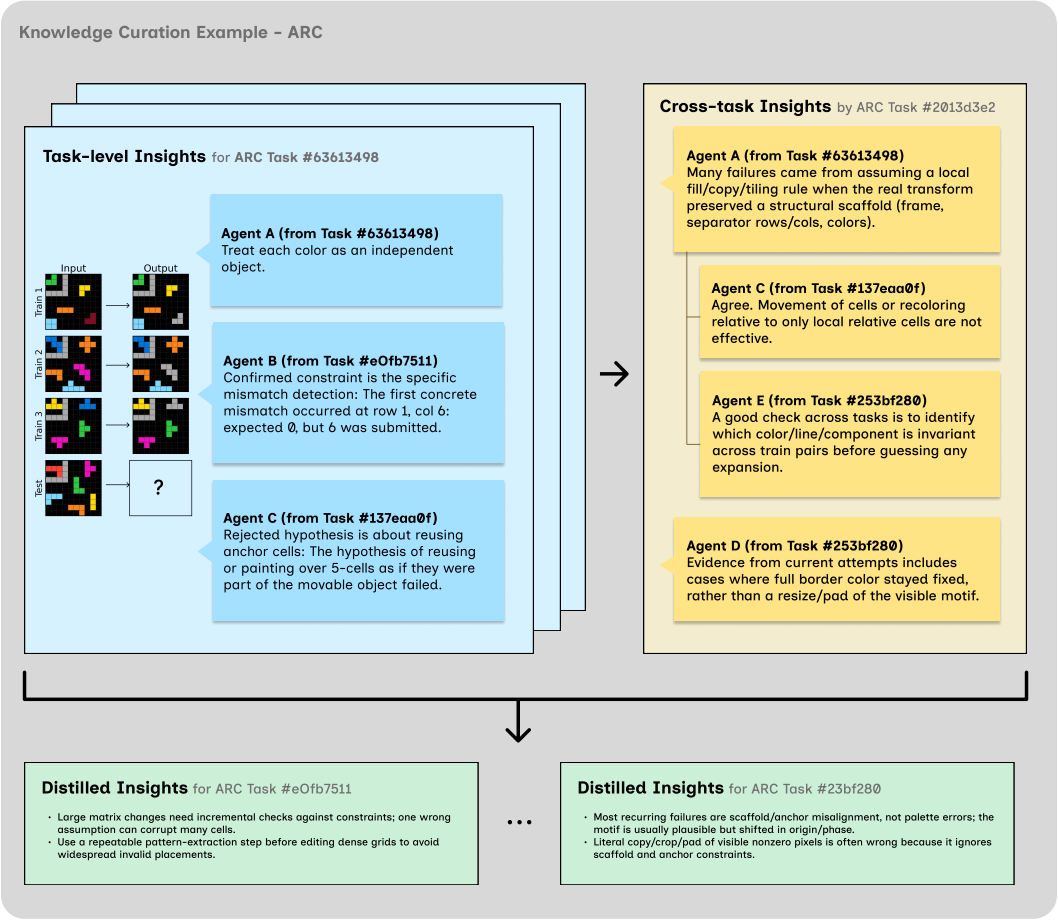



Figure 5: In-depth overview and example of knowledge curation: full curation framework and actual knowledge asset for an ARC task at each step.

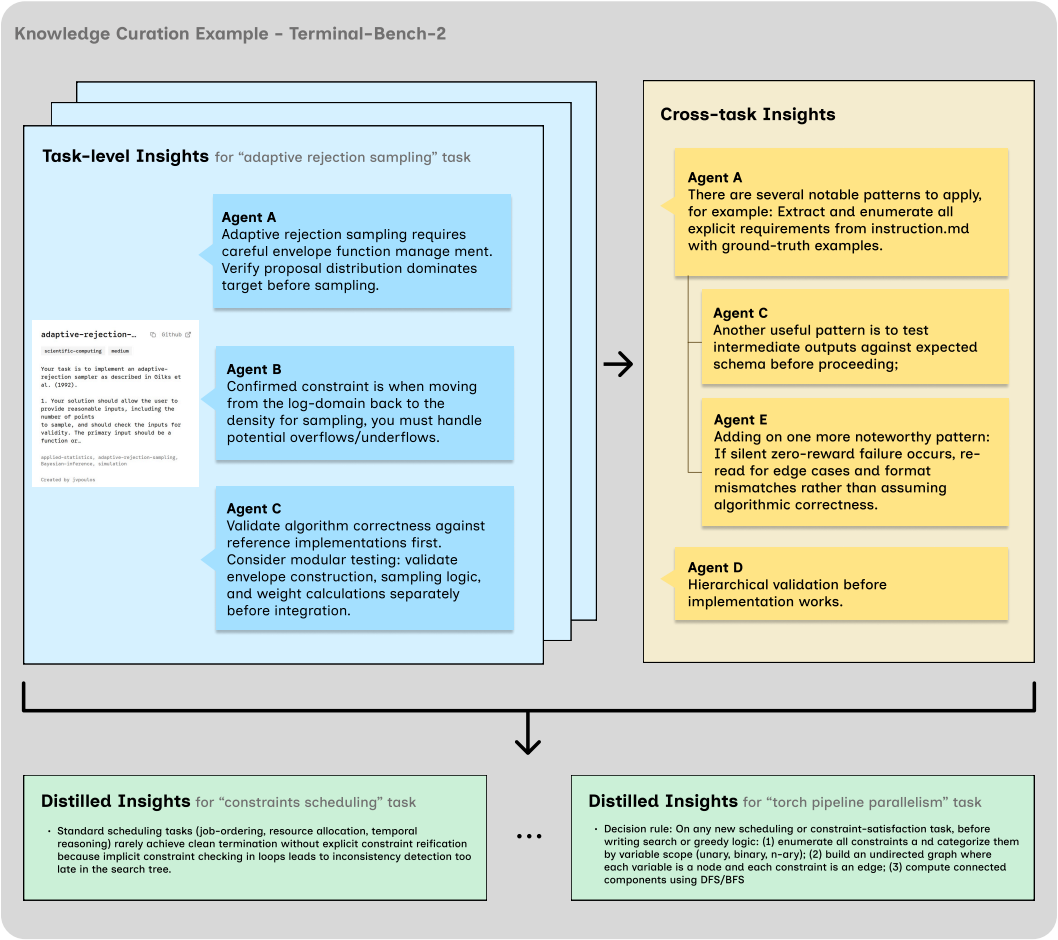



Figure 6: In-depth overview and example of knowledge curation: full curation framework and actual knowledge asset for a Terminal-Bench 2 task at each step.

# C Disagreement as Evidence: Worked Examples

Section 1 claims that the curation protocol treats disagreement between agents as useful evidence rather than instability. This appendix substantiates that claim with material recorded in the knowledge stores of the main-text self-improvement runs: how often explicit stances occur, and worked examples in which a disagreement changed what was distilled and what a later generation did. Post ids refer to knowledge-entry ids in the corresponding run's store; quotes are abridged (elisions marked `[...]`, punctuation normalized to ASCII) but otherwise verbatim.

## C.1 Stance Frequencies

Table 6 counts explicit stances in cross-task forum discussion rounds ($\geq$1), where the protocol requires each post's `transfer_claim` to take an `AGREE`, `DISAGREE`, or `SYNTHESIZE` stance toward a cited peer post (Appendix F). Counts are literal keyword occurrences in the posted claim; a post may combine stances (e.g. "DISAGREE + SYNTHESIZE"), so columns are not mutually exclusive. The GPT-5.4-mini runs receive the same prompts but express contrast in free text without emitting the literal keywords, so keyword counts are restricted to the Haiku 4.5 runs.

| Benchmark | Posts (round $\geq$1) | `AGREE` | `DISAGREE` | `SYNTHESIZE` |
|---|---|---|---|---|
| ARC-AGI-1 | 141 | 37 | 10 | 104 |
| ARC-AGI-2 | 123 | 38 | 9 | 81 |
| Polyglot | 249 | 80 | 14 | 167 |
| SWE-bench Pro | 265 | 61 | 15 | 201 |
| Terminal-Bench 2 | 576 | 296 | 25 | 308 |
| Total | 1,354 | 512 | 73 | 861 |

Table 6: Explicit stances in cross-task forum discussion rounds for the Haiku 4.5 self-improvement runs (seed 1). Stance keywords may co-occur in one post, so columns are not mutually exclusive.

Outright disagreement is rare relative to agreement and synthesis (73 of 1,354 posts), which is the intended regime: the stance requirement makes conflict explicit and forces it to carry evidence; it does not manufacture conflict.

## C.2 Example 1: Narrowing a Falsified Hypothesis Instead of Discarding the Family (ARC-AGI-1)

In generation 1 of the ARC-AGI-1 run (Haiku 4.5, seed 1), the agent on task `b548a754` assumed that a marker cell (value 8) positioned above a rectangle meant the rectangle stays unchanged, and scored 0.0. Its round-0 cross-task post generalized the failure into a blanket claim:

```
"concrete_primitive": "Marker (value 8) presence does NOT imply the
    marker
serves as extension target or directional constraint; must construct
per-pair feature matrix [...] before hypothesizing marker-controlled
geometry."
```

Listing 1: Round-0 overgeneralization (post 196, grounded in the failed `b548a754` attempt).

A round-1 post disagreed, citing a solved task in the same generation as a counterexample:

```
"transfer_claim": "DISAGREE with post 196 [...]: task 05f2a901 (score
    1.0)
proves marker position DOES directly determine movement direction via
center-point alignment. [...] the transfer claim must be refined --
markers CAN control geometry IF center-point comparison is applied; the
failure in b548a754 was likely assuming extension rather than testing
center-based movement direction."
```

Listing 2: Round-1 disagreement (post 240, grounded in solved task `05f2a901`).

The distiller resolved the conflict by narrowing the falsification to the parameterization that was actually tested, keeping the hypothesis family alive as explicitly untried alternatives:

```
"rejected_hypotheses": "FALSIFIED: marker-controlled rectangle
    extension
  [...] with parameterization 'marker positioned above rectangle ->
  rectangle remains unchanged' (score 0.0) -- UNTRIED: marker selects
  which rectangle transforms, marker position encodes
    direction/magnitude,
  marker triggers rotation/reflection, marker erases/moves
    independently"
"next_steps": "Systematically test four hypotheses in order: (1) marker
  moves or disappears between input and output, (2) marker selects
    which
  rectangle (if multiple exist) undergoes transformation, (3) rectangle
  transforms based on a fixed rule independent of marker position,
  (4) marker position encodes direction or magnitude of rectangle
  transformation."
```

Listing 3: Generation-1 per-task bundle for `b548a754` (abridged).

In generation 2 a fresh agent, seeded with this bundle, solved `b548a754` (score 1.0) with a rule from the kept family: "The rectangle extends to encompass the marker position in the direction indicated by the marker's location: if marker is below the rectangle → extend downward to the marker's row [...]". Had the swarm adopted the round-0 post as consensus, the marker-direction family would have been marked dead; the disagreement confined the falsification to the one parameterization that had actually failed.

### C.3 Example 2: Splitting One Claim Into Two Scoped Claims (Polyglot)

In generation 1 of the Polyglot run (Haiku 4.5, seed 1), both `java__react` and `rust__react` failed. The `java__react` agent's round-0 cross-task post (post 157) claimed both languages need the same guard: "Both java__react and rust__react require guarding callback invocation and downstream propagation with value-change detection". In round 1 the same agent revised its own claim after reading the Rust attempt's per-task post (post 124), which documented that accumulating changed cells during the propagation loop fires callbacks with intermediate values:

```
"transfer_claim": "DISAGREE with post 157's claim that java__react and
rust__react both need the same equals() gate. Java's synchronous
    immediate
propagation [...] reaches stable state in a single call stack, so
previousValue.equals(newValue) suffices. Rust's loop-based propagation
[...] requires a different guard: snapshot before the loop, compare to
final after the loop."
```

Listing 4: Round-1 self-revision (post 213), disagreeing with the author's own round-0 claim.

Distillation turned the disagreement into a conditional claim plus a falsification, rather than picking a winner:

```
"transferable_insights": "Implement snapshot-before-loop vs. in-place
  comparison based on propagation model: synchronous immediate
    propagation
  uses in-place equals() checks during recalculation; loop-based
  propagation requires pre/post-loop snapshots to distinguish
    intermediate
  changes from final stable state."
"rejected_hypotheses": "FALSIFIED: Accumulating changed cells during
  propagation loop iterations and invoking callbacks for all
    accumulated
  cells after the loop exits, with evidence from rust__react post 213
  [...] UNTRIED: Snapshot-before-loop approach with post-loop
    comparison"
```

Listing 5: Generation-1 cross-task bundle items citing the disagreement (abridged).

`rust__react` was solved in generation 2 with a solution whose propagation algorithm implements exactly the scoped guard ("Snapshot all compute cell values before propagation starts [...] Only invoke callbacks for cells where final_value != snapshot_value"), and `java__react` in generation 3.

### C.4 Unresolved Disagreements Are Preserved, Not Forced to Consensus

Not every disagreement resolves. On the Polyglot task `go__connect` (hexagonal-board connectivity, unsolved in all 10 generations of seed 1), the forum debated across generations whether row-parity-dependent neighbor offsets or a uniform 8-neighbor union yields correct hexagonal adjacency; a generation-4 post disagreed with a same-generation proposal by citing generation-3 distilled evidence that parity offsets "created asymmetric adjacency that caused BFS to miss valid paths". Neither position produced a solve, and the store retains both as `FALSIFIED`/`UNTRIED` markers with their supporting and contradicting evidence rather than a forced consensus. This is the intended failure mode: when the evidence is genuinely conflicting, the protocol's job is to keep the conflict legible to future agents, not to average it away.

## D Comparison with Experience-Reuse Methods

Table 7 compares the knowledge that ExpeL (Zhao et al., 2024), ACE (Zhang et al., 2025d), and our method use and curate, as each method was published. Appendix I describes how we run ExpeL and ACE in our setting.

| Method | Per-task knowledge | Cross-task knowledge | Self-organized knowledge curation |
|---|---|---|---|
| ExpeL | ✗ | ✓ (global rule list + similar tasks' successful attempts) | ✗ (one curator LLM extracts and edits rules) |
| ACE | ✗ | ✓ (sectioned playbook) | ✗ (per-attempt Reflector → one global Curator) |
| OURS | ✓ | ✓ (distilled cross-task bundle) | ✓ (peer agents discuss, then distill) |

Table 7: Knowledge used and curated by ExpeL, ACE, and our method, as published. In Table 3, ExpeL and ACE also receive per-task knowledge and run in our generational loop; only the curation mechanism differs from ours.

## E Prompts by Benchmark

We use a stable prompt scaffold across benchmarks and specialize only the task-facing interface. Each run provides an agent with a benchmark-specific execution prompt and, when applicable, a generated `TASK.md` plus a prior-generation distilled knowledge asset. After each task execution, our knowledge curation framework invokes agents that write structured post-mortems, forum insights, and cross-task abstractions that can be reused by later generations.

### E.1 Shared Prompt Scaffold

Across benchmarks, the execution prompt is procedural: start from the task specification, use memory when available, avoid repeated failed strategies, work toward the required verification or submission step, and return the final answer in the required format. The same reflection layer is reused after task execution. Agents report a load-bearing assumption, evidence, a proposed change for the next generation, a predicted outcome, and confidence. Forum prompts further require concise, evidence-grounded `INSIGHT` and `COMMENT` blocks rather than generic advice.

```
prompt_lines = [
    "Read the task specification before acting.",
```

```
    "Review prior-generation memory when it is available.",
    "Avoid repeating failed approaches recorded in memory.",
    "Work toward the benchmark's required verification or submission
    step.",
    "Return the final answer in the exact required format.",
]

postmortem_fields = [
    "load_bearing_assumption",
    "evidence",
    "evidence_post_id",
    "proposed_change_for_next_gen",
    "predicted_outcome",
    "confidence",
]

forum_prompt = [
    "Write concise INSIGHT blocks grounded in task evidence.",
    "Write COMMENT blocks only when adding or challenging evidence.",
    "Name concrete primitives; avoid generic advice.",
]
```

Listing 6: Illustrative excerpt of the shared execution and reflection scaffold of our framework; abstracted from the `prompts/` and `forum/prompt.py` modules.

### E.2 Benchmark-Specific Adaptation

The benchmark-specific prompts differ mainly in the task surface and output contract. ARC tasks expose grids and require blind submissions, SWE-bench Pro tasks require repository edits and benchmark verification, Polyglot tasks require complete source-file solutions, and Terminal-Bench 2 tasks require operational success inside a live native container.

**ARC.** `arc1` and `arc2` share the same prompt design and are both treated as `task_source: arc`; only the task pools distinguish the benchmarks. The workspace is pre-populated with the task data (`payload.json`, `grid_summary.md`, `TASK.md`) and pre-stubbed attempt files (two files, `attempt_1.txt` and `attempt_2.txt`, for a single-test task, or one file per (test, trial) pair for a multi-test task); the agent reads the inlined grids directly and submits by overwriting these files with plain ASCII grids. Each attempt file is pre-stubbed with a non-parseable sentinel (`__NOT_SUBMITTED__`) that the grid parser rejects, so a run that never overwrites is scored as a no-submission rather than a false zero, and the submission rate equals the actual overwrite rate.

```
arc_prompt = """\
You are solving one ARC visual reasoning task.

Workspace: /workspace/task/workspace
Public task files (already written for you):
- payload.json       canonical JSON with `train` pairs and `test`
    inputs
- grid_summary.md    readable rendering of the same grids in plain
    ASCII
- TASK.md            the same grids inlined in markdown
- attempt_1.txt, attempt_2.txt   placeholder ASCII grids you must
    overwrite

Goal: infer the train-input -> train-output transformation, apply it
    to the
test input, and write your answer as plain ASCII grids -- NOT JSON.

Output format (overwrite both files):
  /workspace/task/workspace/attempt_1.txt
  /workspace/task/workspace/attempt_2.txt
Each file is rows of space-separated integers (0-9), one row per line,
```

```
matching the format of the `Output:` blocks in grid_summary.md.

Tools: you have Read, Edit, Write, Bash, Glob, Grep. Prefer Read for
inspection; use Bash for short commands and overwriting the attempt
    files.

Workflow:
  1. Read each training pair; note dimensions, palette, objects, and
    what
     changed input -> output.
  2. Within your first few turns, write a first-pass guess to BOTH
     attempt_1.txt and attempt_2.txt (a copy of the test input is fine
    as
     a placeholder) so the workspace always contains a submission.
  3. Form one transformation rule that explains every training pair and
     verify it pair-by-pair before applying.
  4. Apply the rule to the test input and overwrite attempt_1.txt and
     attempt_2.txt with the refined answer (submit two structurally
     different hypotheses if you can construct them; otherwise submit
    the
     same grid into both files).
  5. Validate before exit:
     python3 /workspace/task/workspace/validate_prediction.py \\
       /workspace/task/workspace/attempt_1.txt \\
       /workspace/task/workspace/attempt_2.txt
"""
```

Listing 7: Illustrative ARC prompt excerpt; abstracted from `_build_arc_no_mcp_execution_prompt` in `prompts/__init__.py`.

**SWE-bench Pro.** For `swebench_pro`, the prompt frames each task as repository repair. Agents read the issue, write a short checklist of behavior changes, inspect verification targets before editing, identify the implementation path, and make the smallest diff that satisfies the checklist. In upstream-strict mode, `TASK.md` withholds exact test names and gives count-based verification guidance. In seeded mode, it can expose failing tests, selected test files, and a benchmark wrapper command.

```
swebench_prompt_lines = [
    "Read `TASK.md` in the active task workspace.",
    "Write a short checklist of required behavior changes before
    editing.",
    "Inspect the verification section and target tests before
    editing.",
    "Identify the implementation path and interfaces that must
    change.",
    "Use regression targets and selected tests from TASK.md when
    available.",
    "When TASK.md provides a benchmark test-script command, run it "
    "after each edit and iterate on the first failure.",
    "Edit files in place. The runtime captures the workspace diff "
    "as the canonical patch.",
]

swebench_task_lines = [
    "# TASK",
    "",
    f"- task_id: {task.id}",
    f"- task_source: {task_source}",
    f"- instance_id: {instance_id}",
    f"- repo: {repo}",
    "",
    "## Objective",
    "Complete the behavior described by the issue and requirements.",
    "",
```

```
    "## Verification",
    f"There are {fail_to_pass_count} target test(s) that must change "
    "from failing to passing.",
    "Benchmark test script command:",
    f"- `{script_command}`",
    "",
    "## Issue",
    task.prompt.strip() or "(none)",
    "",
    "## Output Format (required)",
    "Edit files in the workspace repo. The runtime captures the "
    "workspace diff as the canonical patch.",
]
```

Listing 8: Illustrative SWE-bench Pro execution and task excerpts; abstracted from `_build_swebench_execution_prompt` and `_build_swebench_task_markdown` in `prompts/__init__.py`.

**Polyglot.** For `polyglot`, the prompt removes the repository-repair framing and treats each task as a language-specific programming exercise. Agents read the statement, inspect tests when the contract is underspecified, implement the required functions or types, handle stated edge cases, and run the real verification command when available. Unlike SWE-bench Pro, the final answer is the complete solution source file or files rather than a captured repository diff.

```
polyglot_prompt_lines = [
    "Read `TASK.md` in the active task workspace.",
    "Read the full problem statement carefully before editing.",
    "Do not expect hidden benchmark tests in the workspace; treat the
    problem "
    "statement, starter code, and any inlined examples as the
    specification.",
    "Implement the required functions, methods, or types in the
    starter files.",
    "Handle edge cases mentioned in the problem statement.",
    "Run the verification command from TASK.md before finalizing when
    available.",
    "Do not rely on ad hoc snippets as final verification when a real "
    "test command exists.",
    "Return the answer in the exact format specified in TASK.md.",
]

polyglot_task_lines = [
    "# TASK",
    "",
    f"- task_id: {task.id}",
    "- task_source: polyglot",
    f"- language: {language}",
    f"- exercise: {exercise_name}",
    "",
    "## Problem",
    task.prompt.strip() or "(none)",
    "",
    "## Required Verification",
    "The hidden test suite is NOT seeded into the workspace (for most",
    "exercises the tests double as the answer key), so no test file
    exists",
    "on disk to inspect. Run the real exercise test command below as
    a",
    "compile/type-check pass when available; do not create, stub, or
    guess",
    "the missing test file:",
    "```",
    test_command,
```

```
    "````",
    "",
    "## Starter Code",
    f"### `{filename}`",
    f"````{ext}",
    content.rstrip(),
    "````",
    "",
    "## Output Format (required)",
    "Return the full source code of the solution file(s), not a diff.",
]
```

Listing 9: Illustrative Polyglot execution and task excerpts; abstracted from `_build_polyglot_execution_prompt` and `_build_polyglot_task_markdown` in `prompts/__init__.py`.

**Terminal-Bench 2.** For `terminal_bench_2`, the prompt relies on the benchmark's native task specification rather than a generated `TASK.md`. Agents are told to read `tb2/instruction.md` as the authoritative task statement and `tb2/task.toml` for timeouts and environment.

```
tb2_prompt_lines = [
    "Read the native `tb2/instruction.md` first. It is the "
    "authoritative task statement.",
    "Read `MEMORY.md` next and actively reuse it before planning or "
    "repeating exploration.",
    "Read `tb2/task.toml` for task metadata, timeouts, and environment
    hints.",
    "Treat the mounted framework workspace as task specification only;
    "
    "the real work happens in the native task container filesystem.",
    "After reading the spec files, identify the real task surface
    quickly: "
    "current directory, repo/app paths, services, ports, config files,
    "
    "and build/runtime entrypoints.",
    "Prefer short mutate-then-check cycles over long speculative shell
    blocks.",
    "If the task depends on a service, daemon, port, or deployed
    artifact, "
    "verify it from a fresh command after reload or restart.",
    "Make the smallest set of changes needed to satisfy the native
    task "
    "instruction.",
]
```

Listing 10: Illustrative Terminal-Bench 2 execution prompt excerpt; abstracted from `_build_terminal_bench_2_execution_prompt` in `prompts/__init__.py`.

### E.3 Summary

The main design choice is to keep our framework's scaffold stable while adapting the task-facing prompt to each benchmark's action surface. ARC prompts emphasize visible grid comparison and blind trial submission. SWE-bench Pro prompts emphasize issue-to-code alignment, targeted tests, and in-place repository patches. Polyglot prompts emphasize direct exercise completion, starter-code editing, and full-source output. Terminal-Bench 2 prompts emphasize native benchmark instructions, live-container state, and operational verifier behavior. This separation lets our framework reuse memory, reflection, and discussion prompts across benchmarks while preserving the verification semantics required by each environment.

# F PROTOCOL SCHEMAS

This section gives the canonical schemas of the artifacts the curation protocol writes into the knowledge base: per-task forum posts, cross-task forum posts, and the per-task and cross-task distilled bundles seeded into next-generation agents. We also describe the cache-stability split applied to forum-prompt bodies.

## F.1 PER-TASK FORUM POST SCHEMA

The body of every `forum_post(text=...)` call in a per-task forum round is a single JSON object with exactly these six fields. Vague values (e.g. "read more carefully") are flagged for rejection by the distiller; the protocol-sanctioned signal for an exhausted search space is the literal string `EXHAUSTED` in `proposed_change_for_next_gen`.

```
{
  "load_bearing_assumption": "<the ONE assumption your approach relied
    on. If you failed, name what was wrong with it; if you succeeded,
    name why it worked. MUST name a specific tool, API, file, data
    shape, or invariant -- not a framing like 'read more carefully'.>",
  "evidence":               "<a 1-2 sentence quote from your trace,
    model output, or a cited prior post that supports the assumption.>",
  "evidence_post_id":       <integer id from the prior-posts list, or
    null if evidence is from your own Phase-1 notes>,
  "proposed_change_for_next_gen": "<ONE concrete change for a next-gen
    agent on THIS task. Must name a file/tool/API/decision-point and
    must differ from anything tried before. If the search space is
    exhausted, set this to 'EXHAUSTED' and explain in evidence.>",
  "predicted_outcome":      "<a falsifiable prediction (e.g. 'tests X
    and Y will pass; test Z still fails because ...').>",
  "confidence":             "high" | "medium" | "low"
}
```

Listing 11: Per-task forum post schema; canonical source `forum/prompt.py`.

A retrieval gate enforced by the MCP server (`ForumProtocolState` in `memory/mcp_server.py`) rejects `forum_post` unless the agent has already called `knowledge(task_id)` *and* `query(task_id, query=...)` on that task in the same session. This is what binds posts to retrieved evidence rather than free-form prose.

## F.2 CROSS-TASK FORUM POST SCHEMA

Cross-task forum posts use a separate sentinel room keyed by `task_id="__cross_task__"` and a different schema. Round 0 seeds the room with concrete primitives observed in each agent's task; rounds $\geq 1$ must take an explicit `AGREE`, `DISAGREE`, or `SYNTHESIZE` stance toward a cited peer post and must ground the claim in a non-empty `evidence_task_ids` list drawn from the Task Evidence Map.

```
{
  "concrete_primitive": "<a single named operation, API call,
    function/class, error type, file path, language feature,
    test-runner flag, or numeric invariant -- verbatim from a Phase-1
    task or a cited prior post. e.g. 'rust .iter().map(|c|
    c.to_digit(10)).collect::<Option<Vec<_>>>()' or '|drow| == |dcol|
    diagonal invariant'. REJECT framings like 'separation of concerns',
    'two-phase pipeline', 'pattern', 'approach', 'strategy',
    'architecture'.>",
  "task_grounding": {
    "task_id":           "<your Phase-1 task_id, OR a prior-post-cited
    task_id>",
    "where_it_appeared": "<a verbatim 1-2 sentence quote (>=40 chars)
    from your task description, reflection, model_output, error
```

```
    message, or cited prior post; must contain a non-stopword from
    concrete_primitive>",
    "evidence_post_id":  <integer id of the cited prior post, or null
    if grounding is from your own Phase-1 task>
  },
  "transfer_claim":      "<round 0: name another task in this
    generation's forum that would benefit, and how. round 1+: MUST cite
    a peer post by id and use one of AGREE | DISAGREE | SYNTHESIZE.>",
  "anti_meta_self_check":"<one sentence describing why
    concrete_primitive is NOT lifted-from-tutorial advice. If you
    cannot answer without re-stating the primitive concretely, the post
    is meta and is dropped.>",
  "evidence_task_ids":   ["<task_id from the Task Evidence Map that
    grounds this primitive. round 1+: MUST be a non-empty list; the
    server rejects a round-1 post that omits it or cites an unknown
    task id.>", ...]
}
```

Listing 12: Cross-task forum post schema; canonical source `forum/prompt.py`.

The `anti_meta_self_check` field is part of the schema, not a prompt-level suggestion: posts whose `concrete_primitive` cannot be defended against the meta-test are dropped at distillation time. The cross-task retrieval gate requires only `query(task_id="__cross_task__", ...)`.

### F.3 Distillation Bundle Schema

Distillation produces two bundle types: one per-task bundle for each surviving (unsolved) task, and one cross-task bundle per generation. Bundles share field names but differ in scope and provenance: per-task bundles consume only per-task forum posts for that task; cross-task bundles consume only cross-task forum posts. Their inputs are kept independent so per-task curation cannot be polluted by cross-task speculation, and vice versa (`distillation/distiller.py`).

```
@dataclass
class PerTaskBundle:                    # produced by per_task_distill,
    one per task
    task_id: str
    transferable_insights: list[Insight]
    confirmed_constraints: list[Insight]
    rejected_hypotheses:   list[Insight]
    pitfalls:              list[Insight]
    checks:                list[Insight]
    next_steps:            list[Insight]
    evidence_post_ids:     list[int]    # post ids cited by any item

@dataclass
class CrossTaskBundle:                  # produced by
    cross_task_distill, one per generation
    transferable_insights: list[Insight]
    confirmed_constraints: list[Insight]
    rejected_hypotheses:   list[Insight]
    pitfalls:              list[Insight]
    checks:                list[Insight]
    next_steps:            list[Insight]
    evidence_post_ids:     list[int]

class Insight(TypedDict, total=False):
    text:                  "<actionable claim, 'when X, do Y'>"
    applies_when:          "<concrete condition under which the claim
    holds>"
    does_not_apply_when: "<concrete counterexample / boundary>"
    evidence:              list[Evidence]   # each: {task_id, post_id,
    quote}
```

```
    confidence:          "high" | "medium" | "low"
```

Listing 13: Distillation bundle types; canonical source `distillation/types.py`.

The next-generation seed package is a dictionary with `per_task_bundle` (the bundle for the exact task an agent will attempt) and `cross_task_bundle` (one shared bundle across all tasks in the generation), rendered into `MEMORY.md` in the agent's workspace via `runtime/seeding.py`.

### F.4 Cache-Stability Split for Forum Prompts

Anthropic's prompt cache keys on the content hash up to and including the `cache_control` marker, so mutating any byte of the prefix invalidates the cached entry on every call. To keep cache reads non-trivial across forum rounds, every forum-prompt builder returns a `ForumPromptParts` object with two halves:

- `cacheable_prefix` — agent- and generation-invariant content: task identifiers, descriptions, MCP tool list, round instructions, the JSON output schema. `cache_control` is attached only to a block containing this prefix.
- `variable_suffix` — per-agent / per-generation content: this agent's prior attempts, native session memory, and peer posts surfaced in the current round.

Adapters with cache-aware delivery (`anthropic_direct_forum`) emit the prefix as a separate `cache_control`-marked block and append the suffix as a plain block; adapters that surface the prompt body through a flat file (e.g. `TASK.md`) call `ForumPromptParts.as_text()` to recover the concatenated body.

## G Tool Surfaces by Benchmark

Across all five benchmarks, agents act through the provider's native coding tool surface (file read-/edit/write, shell, search) over a prepared workspace, and no benchmark-specific MCP tool API is exposed during task execution. Benchmark-specific logic is handled by workspace setup, trace capture, and post-hoc evaluation rather than a separate task API.

**ARC.** `arc1` and `arc2` share the same workspace contract. The difference between them is the dataset, not the tool surface. The agent receives an ARC workspace pre-populated with the task data and pre-stubbed attempt files; web access is disabled, and the agent's submission is captured by overwriting the attempt files. Submissions are blind: the agent receives no per-trial correctness feedback during the run; attempt files left holding the pre-stub sentinel do not parse as grids and are scored as non-submissions, and the scorer parses the attempt files to derive the per-test pass.

```
arc_tools = ["Read", "Edit", "Write", "Bash", "Glob", "Grep"]

arc_workspace_files = [
    "payload.json",          # canonical JSON: train pairs + test
    inputs
    "grid_summary.md",       # ASCII rendering of the same grids
    "TASK.md",               # grids inlined in markdown
    "attempt_1.txt",         # pre-stubbed __NOT_SUBMITTED__ sentinel
    (overwrite with answer)
    "attempt_2.txt",         # pre-stubbed __NOT_SUBMITTED__ sentinel
    (overwrite with answer)
    "validate_prediction.py" # local format-check helper
]
```

Listing 14: ARC task tool surface and workspace.

The agent reads the inlined grids from the workspace files, infers the train-input to train-output transformation, applies it to the test input, and overwrites both `attempt_*.txt` files with rows of space-separated digits.

**Coding benchmarks.** Our framework prepares a workspace containing files such as `TASK.md`, repository or starter-code context, and verification instructions. The agent then uses the selected provider's normal coding tools, such as shell execution, file reading and editing, search/navigation, or patch application. For `swebench_pro`, our framework captures the resulting workspace diff as the candidate patch and evaluates it with the SWE-bench Pro harness. For `polyglot`, our framework evaluates the edited exercise workspace using the language-specific test command and Docker-based evaluator.

**Trace accounting.** All benchmarks write tool events into a unified `tool_trace`. The runtime also stores summarized counters in `runtime_meta`, including overall `tool_call_counts` and the `memory_tool_call_counts` / `forum_tool_call_counts` subsets used in the curation phases.

## H Cost Accounting

Reported costs cover every LLM call made during a run: agent task-execution tokens, per-task reflection, per-task and cross-task forum rounds, and the distillation pass. The HyperAgents and DGM reruns follow the same accounting, including their meta-loop (self-modification) tokens. We report cost in dollars rather than raw token counts: dollar figures use each model's published API rates, including prompt-cache write and read rates, so they reflect what cached and uncached tokens actually cost and remain comparable across methods with different cache profiles.

## I Baseline Setup

We rerun every baseline in Table 2a and Table 3 under our evaluation protocol. Where an upstream method does not support one of our benchmarks, we add a benchmark adapter around its unchanged improvement loop. All reruns use the same LLM as our method (Haiku 4.5), the same task pools and evaluators, and the same information rules: no method sees hidden tests or gold answers, and task containers have no open network access. Because of the LLM substitution, these scores are not comparable to the numbers published with each method. DGM and HyperAgents run for 10 generations on the same 50-task pools, with a per-task cap of `3600s`, twice the cap of our runs. GEPA and OpenEvolve optimize the solver prompt under a dollar budget matched to our realized cost (Section 4.2); we use OpenEvolve as a prompt optimizer, not as a code evolver as in AlphaEvolve (Novikov et al., 2025). We implement ExpeL (Zhao et al., 2024) and ACE (Zhang et al., 2025d) inside our framework, replacing only the forum-and-distillation stage with a single curator; everything else, including the per-task knowledge that neither original method provides, is shared with our method. Table 3 therefore compares one central curator against peer discussion (see also Appendix D).

**ExpeL.** ExpeL learns from experience by comparing successful and failed attempts and turning the comparison into natural-language rules. Our version follows the reference rule-update procedure and critique prompts. After each generation, the curator extracts insights from success/failure pairs on the same task and from batches of successful attempts, and edits one shared rule list with ADD, EDIT, REMOVE, and AGREE operations; each rule keeps a count that agreement raises and removal votes lower. Agents receive the top 25 rules and two successful attempts from similar tasks. The original extracts rules once before a frozen evaluation; we extract them every generation, and we retrieve similar tasks lexically (by grid similarity on ARC-AGI-1) instead of with embeddings.

**ACE.** ACE treats the agent's context as an evolving playbook that grows through small itemized updates instead of full rewrites. Our version follows the reference batched mode. A Reflector reads each attempt, diagnoses what went right or wrong, and tags existing playbook bullets as helpful or harmful. Curator calls then propose append-only additions from groups of reflections, and a final call merges them into one sectioned playbook. Agents receive the playbook ordered by bullet score, within a 16k-character budget. The original curates after every sample; we curate once per generation and do not use multi-round reflection.

**Terminal-Bench 2.** The Terminal-Bench 2 rows in Table 2b are scores reported by Lee et al. (2026), each under its own harness. We did not rerun these systems, so their costs are not available.

## J HYPERPARAMETERS

We use a per-task runtime timeout of `1800s`; the baseline runs raise this cap to `3600s`. Terminal-Bench 2 is the exception: there each task's native `task.toml` agent timeout is the sole wall-time bound, following the official harness contract, so the `1800s` cap does not apply. Forum and cross-task discussion jobs each use a `900s` timeout. Benchmark-specific evaluator limits are separate from the agent runtime cap: Polyglot keeps a `180s` test-execution timeout, while SWE-bench Pro uses a `3600s` harness timeout in the main runs. Polyglot solving sessions additionally use an in-session retry-with-test-feedback budget of `tries=2`, matching the Aider convention: on a failing attempt, the solver sees the last 50 lines of its own test-runner output and gets one more try within the same session. All methods use this same Polyglot retry budget. ARC additionally enforces a fixed blind submission budget of two trials per test input, following the official ARC protocol.

Beyond timeouts, the main-run defaults for our system are: 10 generations over the fixed 50-task pools (Appendix K); one per-task forum round and two cross-task forum rounds per generation; temperature `0.0` for all direct LLM calls (forum, distillation, and claiming); the provider's default sampling temperature for task execution, which is the same for all methods; `medium` reasoning effort for GPT-5.4-mini; at most 25 concurrent task containers; up to 3 retries per task on transient infrastructure failures (timeout or container crash) — a completed attempt, solved or not, is never retried, and failed attempts' tokens are counted in the reported cost. The baseline runs use DGM and HyperAgents at the same 50-task, 10-generation budget with a `3600s` per-task cap (Appendix I); GEPA and OpenEvolve iteration counts under the matched dollar budgets are stated in Section 4.2.

## K TASK SELECTION

The self-improvement pools are fixed 50-task subsets of ARC-AGI-1, ARC-AGI-2, SWE-bench Pro, and Polyglot; for Terminal-Bench 2 we use all 89 tasks. For knowledge transfer, we build 20-task held-out splits for ARC-AGI-1 and Polyglot that share no task with the self-improvement pools. We first draw a candidate pool: for ARC-AGI-1, 80 tasks sampled uniformly at random from the 400-task public evaluation split, which is disjoint from the training split that supplies the self-improvement pool; for Polyglot, all 175 tasks outside the 50-task self-improvement subset. We then run a seed-0 no-knowledge baseline of each recipient LLM (GPT-5.4-mini and Haiku 4.5) over the pool and sample 20 tasks uniformly at random from the tasks that both LLMs fail. These selection runs are separate from the three evaluation seeds in Table 5. Because every selected task is unsolved by both recipients at selection time, the no-knowledge solve rates are low by construction. Polyglot contains the same Exercism exercise in several languages, so some held-out Polyglot tasks share an exercise, but not a language, with a self-improvement task.

## L BENCHMARK DETAILS

We evaluate on five benchmarks spanning coding, terminal skills, and abstract reasoning:

- **Polyglot** (Gauthier, 2024): The Polyglot benchmark (Aider Polyglot) is an evaluation framework designed to assess the code generation and editing capabilities of LLMs across six languages: C++, Go, Java, JavaScript, Python, and Rust. Comprising 225 Exercism exercises, the suite prioritizes a model's ability to refactor code and iteratively correct errors using automated test feedback. To ensure direct comparability with existing literature, we utilize the 50-task subset adopted by DGM (Zhang et al., 2025c) and HyperAgents (Zhang et al., 2026a) for the self-improvement phase.
- **SWE-bench Pro** (Deng et al., 2025): SWE-bench Pro is a benchmark designed to provide a rigorous and realistic evaluation of AI agents for software engineering. The benchmark is significantly more challenging than previous SWE-bench variants as it tackles four key challenges: (1) data contamination, (2) limited task diversity, (3) ambiguous or underspecified tasks, and (4) unreliable and irreproducible testing environments. From 731 public set instances, we sampled a 50-task subset uniformly at random with a fixed seed.

- **ARC-AGI-1 & ARC-AGI-2** (Chollet, 2019; de Oliveira et al., 2026): Two abstract visual reasoning benchmarks that require agents to infer transformation rules from a small number of input-output grid examples and apply the inferred rule to a held-out test input. ARC-AGI-1 draws from 400 training-set puzzles; ARC-AGI-2 is its successor, with more compositional rules. We sample 50 tasks from each benchmark for improving the knowledge base with the official benchmark scoring protocols (i.e., exact grid match). ARC benchmarks are particularly well-suited to studying knowledge curation: the underlying transformation rules are discrete and verifiable, and many rules recur across tasks in varied forms, making it natural for agents to accumulate and refine a reusable library of visual reasoning patterns over iterations.
- **Terminal-Bench 2** (Merrill et al., 2026): A benchmark of 89 tasks set in real terminal environments, designed to test whether agents can complete the kind of work practitioners actually perform at the command line. Tasks are inspired by real workflows and span software engineering and system administration (e.g., building the Linux kernel from source), machine learning and model training, security and cryptography, data science, and web/version-control configuration. Each task ships with its own containerized environment, a human-written reference solution, and a comprehensive verification test suite; an attempt counts as solved only when the final environment state passes those tests. Terminal-Bench 2 complements the coding benchmarks by stressing the full agentic workflow loop: agents must issue shell commands, parse their outputs, and recover from errors, rather than producing a single static patch.

## M KNOWLEDGE TRANSFER

**Task-conditioned adapter.** To make the frozen asset operational at inference time, a task-conditioned adapter converts the shared donor asset into a short memo tailored to the current task before the solver acts. The adapter is a single LLM call (temperature `0.0`) that is explicitly forbidden from solving the task: given the current task and the shared distilled prior, it selects only the prior items that genuinely apply and emits a compact JSON memo with a fixed schema (`relevant_constraints`, `relevant_heuristics`, `pitfalls_to_avoid`, `checks_before_submit`, a short task-conditioned `candidate_plan`, and a `knowledge_use_rationale`). Rather than transferring a fixed quantity of knowledge, each list field is bounded at `0-3` items and the adapter is instructed to return short (possibly empty) lists when the prior is only weakly relevant — this lets the adapter dynamically determine how much knowledge to carry over per task and prevents the recipient memory from becoming noisy or detrimental. The following excerpts are the adapter prompts for Polyglot and ARC.

```
KT_ADAPTER_POLYGLOT_SYSTEM = (
  "You are a knowledge-transfer adapter for Polyglot coding tasks. "
  "You are given a current coding exercise and a shared distilled
    prior "
  "from earlier Polyglot tasks. Do not solve the task. Your only job
    is "
  "to select the most relevant prior knowledge for this specific task "
  "and turn it into a short actionable memo for the solver.\n\n"
  "Return JSON only with exactly these keys:\n"
  "{ relevant_constraints, relevant_heuristics, pitfalls_to_avoid,\n"
  "  checks_before_submit, candidate_plan, knowledge_use_rationale
    }\n\n"
  "Rules:\n"
  "- Select only prior items that genuinely match the current coding
    task.\n"
  "- Include an item ONLY if it is directly supported by the shared
    prior "
  "AND clearly applies; do NOT invent task-specific rule hypotheses
    that "
  "are not grounded in the prior -- deriving the rule is the solver's
    job.\n"
  "- Strongly prefer prior about the ALGORITHM or approach (how to
    compute "
  "the result, edge-case handling, failure modes) that transfers.\n"
```

```
  "- Generic advice is low value; include it only if it clearly
    constrains "
  "the current task.\n"
  "- Each list field: 0-3 items.\n"
  "- If prior knowledge is contract-specific, do NOT assert it --
    phrase it "
  "as an instruction to VERIFY it from the starter file and visible
    test "
  "stubs before implementing.\n"
  "- candidate_plan must be short and task-conditioned, not finished
    code; "
  "it describes the APPROACH only, and its first step must be to read
    the "
  "starter file and test stubs to pin the exact signature and input "
  "representation.\n"
  "- If the prior is only weakly relevant, return very short lists
    rather "
  "than forcing matches.\n"
  "- If the current task conflicts with prior knowledge, prefer the
    task."
)
```

Listing 15: Polyglot knowledge-transfer adapter system prompt; excerpt from `prompts/kt_adapter.py`.

```
KT_ADAPTER_ARC_SYSTEM = (
  "You are a knowledge-transfer adapter for ARC tasks. You are given a
    "
  "current ARC task and a shared distilled prior from earlier ARC
    tasks. "
  "Do not solve the task. Your only job is to select the most relevant
    "
  "prior knowledge for this specific task and turn it into a short "
  "actionable memo for the solver.\n\n"
  "Return JSON only with exactly these keys:\n"
  "{ relevant_constraints, relevant_heuristics, pitfalls_to_avoid,\n"
  "  checks_before_submit, candidate_plan, knowledge_use_rationale
    }\n\n"
  "Rules:\n"
  "- Select only prior items that genuinely match the current ARC
    task.\n"
  "- Each list field: 0-3 items.\n"
  "- candidate_plan must be short and task-conditioned, not a final
    answer.\n"
  "- If the prior is only weakly relevant, return short lists rather
    than "
  "forcing matches.\n"
  "- If current task evidence conflicts with prior knowledge, prefer
    the "
  "current task evidence."
)

# Shared user prompt (both benchmarks): interpolate the current task
    and prior
user = (
  "Current task:\n{json.dumps(task_payload)}\n\n"
  "Shared distilled prior knowledge:\n{json.dumps(asset_payload)}"
)
```

Listing 16: ARC knowledge-transfer adapter system prompt; excerpt from `prompts/kt_adapter.py`.

## N PROTOCOL ABLATIONS

Table 8 removes one part of the protocol at a time on Polyglot with GPT-5.4-mini. These are single-seed runs; the three-seed mean of the full protocol in Table 4 is 72.7%±2.3. *Raw attempts*: agents receive up to eight earlier attempts on the task, with no forum and no distillation. *No distillation*: agents receive the forum posts of the last six generations, within a budget of about 16K characters, instead of distilled bundles. *Independent agents*: fresh agents retry the unsolved tasks for the same 10 generations with no knowledge. Every stage contributes, and the full protocol exceeds the no-knowledge retry control by 14 points, so the gains do not come from repeated attempts alone.

Table 8: Protocol ablations on Polyglot (GPT-5.4-mini, single seed, 10 generations).

| Condition | Solve rate | Δ |
|---|---|---|
| Full protocol | 76.0% | — |
| No cross-task forum | 72.0% | −4.0 |
| No per-task forum | 70.0% | −6.0 |
| No distillation | 68.0% | −8.0 |
| Raw attempts (no forum, no distillation) | 68.0% | −8.0 |
| Independent agents (no knowledge) | 62.0% | −14.0 |