
Procedural Memory Distillation: Online Reflection for Self-Improving Language Models

Ye Liu, Srijan Bansal, Bo Pang, Yang Li,
Zeyu Leo Liu, Yifei Ming, Zixuan Ke, Shafiq Joty, Semih Yavuz
Salesforce AI Research
yeliu, srijanbansal, sjoty, syavuz@salesforce.com

Abstract

Reinforcement learning with verifiable rewards (RLVR), along with recent self-distillation variants such as SDPO, evaluates each rollout against a verifier and updates the policy from that episode-level signal. However, the richer procedural information in the rollout is rarely retained or reused. Across episodes and epochs, the model repeatedly encounters related problems under a changing policy, producing cross-episode signals that episode-local updates cannot capture: which strategies consistently pass verification, which failure modes persist, which patterns recur. We propose **Procedural Memory Distillation (PMD)**, which converts these cross-episode signals into *reusable procedural memory* and distills it into the policy’s weights during training. This memory functions as a training scaffold, absorbed into the policy itself, yielding a memory-free model at inference. PMD organizes the memory at three levels of abstraction: raw trajectories, self-reflected strategies and lessons, and higher-level behavioral patterns that recur across problems, all extracted online from the model’s own trajectories. A memory-conditioned self-teacher draws on the accumulated experience to supervise the student on its own rollouts, enabling student to progressively internalize procedural knowledge within its parameters. The central design principle is co-evolution: the policy generates rollouts that update the memory, and memory shapes the supervision that updates the policy. Empirically, across Qwen3-8B and OLMo3-Instruct-7B, PMD improves over SDPO by 3.8–5.5% on SCIKNOWEVAL and 7.9–13.6% on LIVECODEBENCH. Co-evolution powers these gains: freezing either the memory or the policy trails PMD by more than 10% across SCIKNOWEVAL domains.

1 Introduction

The prevailing paradigm for preference optimization and reinforcement learning with verifiable rewards (RLVR) operates at the level of individual episodes; methods such as PPO [25], DPO [22] and GRPO [26, 6] convert per-rollout preferences or outcome-based checks into learning signals. Each rollout receives a reward, feedback or hindsight correction; the policy gets updated accordingly; and the experience is discarded. This design is natural when episodes are independent. However, in practice, models repeatedly encounter the same or related problems across epochs under a continually evolving policy. These repeated interactions carry cross-episode signals that isolated, one-step updates cannot capture: which strategies pass verification, which failure modes persist, which patterns recur.

Recent work suggests that the learning signal available during training can be richer than a scalar reward alone. For instance, in Self-Distillation Policy Optimization (SDPO) [10], the current policy can act as self-teacher when conditioned on training-time context: textual feedback when available, or a successful sibling rollout from the same group when it is not. This reflects a broader shift from offline distillation to on-policy distillation, where the student is trained on states it actually visits rather than static teacher demonstrations [8, 23, 1, 5, 30].

While SDPO and related on-policy distillation approaches [1, 53] help alleviate the sparse reward limitations of standard RLVR and the distributional mismatch of offline distillation, their updates remain episode-local: they do not systematically preserve what the model has discovered across earlier attempts. We propose **Procedural Memory Distillation (PMD)**, which converts these repeated attempts into reusable procedural memory and distills it into the policy’s weights during training. PMD builds the memory online from the model’s own rollouts, organized into three levels: experience (raw trajectories), insight (per-problem strategies and lessons), and behavior (cross-problem reasoning patterns). This memory conditions an on-policy self-teacher: the student continues to learn from trajectories sampled from the current policy, as in SDPO, but the teacher is additionally informed by what the model has discovered in prior epochs. Memory thus functions as a training scaffold: it enriches the teacher during learning and is gradually internalized into the student’s parameters, so the resulting model reasons natively at inference.

A central design principle of PMD is the *co-evolution of policy and memory*. At each stage of training, the current policy generates rollouts, receives feedback, and writes new experience into memory, and the updated memory then conditions the self-teacher that trains the next version of the policy. As shown on Table 1, this tight, online coupling is what distinguishes PMD from static or offline memory banks extracted once by a fixed model: as the learner evolves, memory evolves with it, keeping the teacher signal aligned with the policy’s current strengths and failure modes throughout training.

PMD organizes procedural memory into three levels. *Experience memory* (Level-0) stores rollouts, rewards, and feedback for each problem. *Insight memory* (Level-1) reflects on these attempts to extract strategies and lessons. *Behavior memory* (Level-2) groups semantically related problems and distills their experiences and insights into reusable reasoning patterns. This hierarchy exposes a trade-off between concreteness and abstraction. Experience memory preserves the faithful evidence but remains highly local, while behavior memory transfers broadly but can become too coarse. Insight memory strikes a middle ground by retaining problem-grounded lessons in a compact and reusable form. During training, PMD draws on problem-specific experience and insight memories, while retrieving cross-problem behavioral patterns from a global memory.

Unlike work on reflection [29], refinement [17], skill learning [39, 33, 16], and memory [3], which externalize feedback, trajectories, or skills and rely on them during inference, PMD uses memory only during teacher-path training. This design enables the student to internalize procedural knowledge, making memory a mechanism for learning rather than a dependency at deployment.

Our contributions are summarized as follows:

1. **Procedural Memory Distillation (PMD)**. We propose a self-distillation framework that turns episode-local supervision into cross-episode procedural memory, built online from the model’s own rollouts and distilled into the policy’s weights. The resulting model reasons natively at inference, with no external memory dependency.
2. **Co-evolution as the design principle**. PMD updates policy and memory jointly: rollouts from the current policy refresh memory, and the refreshed memory shapes the supervision that trains the next policy. This online coupling distinguishes PMD from static or offline memory banks.
3. **A three-level procedural memory hierarchy**. We propose experience, insight, and behavior memory, and characterize the fidelity-transfer trade-off across them. Empirically, distilling Level-1 insights together with Level-2 behaviors yields the strongest internalized policy.
4. **Empirical validation on two verifiable domains**. PMD improves over GRPO and SDPO with both Qwen3-8B and OLMo3-Instruct-7B by 3.8–5.5% on SCIKNOWEVAL and 7.9–13.6% on LIVECODEBENCH. Freezing either the memory or the policy alone costs more than 10% drop on SCIKNOWEVAL, isolating co-evolution as the source of PMD’s gains.
5. **Better test-time compute scaling**. We show that PMD continues to gain from additional rollouts where SDPO saturates, widening the gap with the rollout budget and opening 2-4× wider verifier headroom on SCIKNOWEVAL.

Table 1: Comparison of learning and memory paradigms. Inference-time memory agent methods include Memory-Bank, MemGPT, Memento and A-MEM [54, 20, 55, 41].

Method	Evolving policy	Memory-free inference	Persistent memory	Evolving memory	Policy-memory co-evolution
Base model	✗	✓	✗	✗	✗
GRPO / RLVR [26, 6]	✓	✓	✗	✗	✗
SDPO / OPD [10, 53, 24]	✓	✓	✗	✗	✗
OPCD [47]	✓	✓	✗	✗	✗
OEL [46]	✓	✓	✗	✗	✗
Inference-time memory agents	✗	✗	✓	✓	✗
PMD	✓	✓	✓	✓	✓

2 Related Work

On-policy and self-distillation. Knowledge distillation trains a student to imitate a teacher distribution [8], but language-model training also suffers from exposure bias when supervision is detached from the learner’s own states. This motivates online imitation and on-policy distillation, where supervision is generated on trajectories visited by the current policy [23, 1, 5, 15, 30]. Recent reasoning-oriented methods use privileged feedback, traces, or revisions for denser supervision: SDPO learns from feedback-conditioned hindsight [10], OPSD and OPSDC distill privileged reasoning context [53, 24], SD-Zero converts binary rewards into self-revision targets [7], and RLSD studies self-distilled RLVR [44]. However, self-distillation can fail when teacher and student distributions drift, privileged information leaks, or reasoning verbalization is suppressed [12, 14, 27, 50]. PMD adopts the on-policy distillation view but changes the privileged signal: beyond the current answer, feedback, or trace, the teacher also receives procedural memory accumulated from prior attempts.

Experiential learning and internalizing context. Related work studies how models turn interaction history into reusable knowledge. OPCD distills transient context into model parameters via on-policy reverse-KL training [47], while OEL extracts experiential knowledge from deployment trajectories before consolidation [46]. ERL, X-KD, MR-Search, and R-Zero likewise use reflection, environments, or generated tasks to improve behavior across episodes [28, 2, 40, 9]. Our setting instead resembles training-time self-improvement: the policy revisits related problems across epochs, updates memory online, and distills from a memory-conditioned teacher into a memory-free student.

Memory, skills, and self-improvement. Memory-based agents maintain explicit stores of reflections, trajectories, or skills retrieved at inference time, including retrieval-augmented generation and long-term memory systems [13, 21, 20], reflection and skill-library agents [45, 29, 17, 32], and memory-management methods that learn to add, update, compress, or route memory [42, 48, 35, 38, 52]. Related work such as ReasoningBank, SkillRL, Skill-SD, SKILLO, MemGen, Mem2Evolve, EvolveR, and SkillOrchestra distills experience into reusable memories or skills for later reasoning or agent behavior [19, 39, 33, 16, 49, 3, 37, 34]. Unlike these approaches, PMD treats memory as a training scaffold rather than an inference-time dependency, studying which memory granularity can be distilled into the policy and transfer to held-out problems.

3 Method

Our goal is to turn repeated training-time experience into a form of procedural memory that improves the policy beyond the current episode. We build on SDPO and variants [10, 53], which uses the current policy in two roles: a *student* that produces the rollout and a *self-teacher* that re-evaluates the same rollout after seeing additional training-time context such as environment feedback or a successful solution. More broadly, this follows the on-policy distillation principle that the teacher should supervise the learner on states the learner actually visits, rather than providing off-policy demonstrations [23, 30]. In standard SDPO, this extra context is largely episode-specific – it is constructed from the current rollout group and is discarded after the current policy update. In this work, we take a broader perspective and hypothesize that such information across episodes captures information about the dynamics of how the model solves the same or similar problems over times, accumulating a rich source of self-learning signals.

We extend this setup by introducing *online procedural memory distillation*. As shown in Figure 1, this corresponds to a four-stage loop: (1) the student makes attempts to a problem and receives verifier feedback; (2) self-reflection summarizes these successes and failures into different levels of procedural memory in the form of strategies, lessons, and behaviors; (3) the teacher leverages these problem-specific and global memory items to produce memory-conditioned supervision; and (4) this guidance is distilled back into the student, through self-distillation, for the next epoch.

3.1 Procedural Memory-conditioned Self-Distillation

We formalize PMD at the level of training update steps. Let π_{θ_t} denote the policy before update step t , and let $M_t = \{M_t^{\text{exp}}, M_t^{\text{ins}}, M_t^{\text{beh}}\}$ denote the procedural memory available at that step. Experience and insight memories are problem-specific, while behavior memory is shared across problems. We assume each epoch contains K update steps.

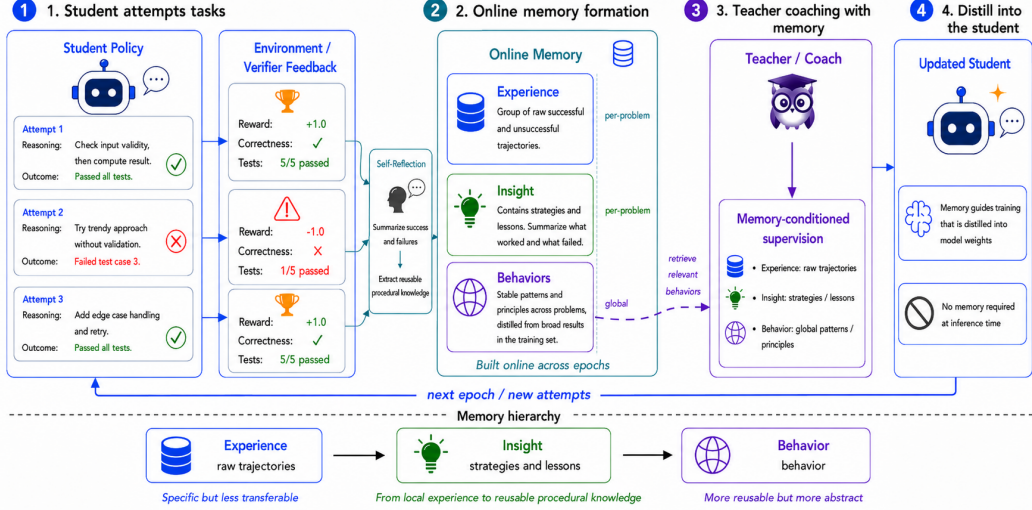


Figure 1: **Overview of Procedural Memory Distillation (PMD).** (1) The student makes repeated attempts and receives verifier feedback. (2) Self-reflection summarizes successes and failures into online memory. (3) The teacher retrieves relevant memory in the form of experience, insight, and behaviors to provide memory-conditioned supervision. (4) This guidance is distilled into the updated student for the next epoch. The bottom row shows the memory hierarchy, from raw trajectories to increasingly abstract and reusable procedural knowledge.

Given a mini-batch B_t , the current policy produces rollout groups

$$\mathcal{T}_{i,t} = \left(x_i, \{y_{i,t}^{(j)}\}_{j=1}^J, \{r_{i,t}^{(j)}, f_{i,t}^{(j)}\}_{j=1}^J \right), \quad x_i \in B_t, \quad (1)$$

where $r_{i,t}^{(j)}$ and $f_{i,t}^{(j)}$ denote the reward and feedback for rollout j . The same rollout group is used to update memory and to provide current-batch context for the teacher reprompt.

Memory update. Experience memory is updated online as new rollouts arrive. After the rollout group for problem x_i is available, PMD updates the problem-specific experience and insight memories as

$$M_{t+1}^{\text{exp}}[i] = \mathcal{U}^{\text{exp}}(M_t^{\text{exp}}[i], \mathcal{T}_{i,t}), \quad M_{t+1}^{\text{ins}}[i] = \mathcal{U}^{\text{ins}}(M_t^{\text{ins}}[i], \text{Reflect}(\mathcal{T}_{i,t})), \quad x_i \in B_t. \quad (2)$$

For problems not visited in the current mini-batch, their experience and insight memories remain unchanged. The reflection step extracts strategies from successful rollouts and lessons from failed rollouts; when both are available, it contrasts successes and failures to identify what distinguishes correct reasoning from incorrect reasoning.

Behavior memory is updated on a slower time scale. Within an epoch, it is kept fixed. After every K update steps, PMD clusters semantically related training questions, aggregates their latest insight memories, and abstracts them into reusable behavioral patterns:

$$M_{t+K}^{\text{beh}} = \mathcal{U}^{\text{beh}}(M_t^{\text{beh}}, \{\text{Abstract}(\{M_{t+K}^{\text{ins}}[i] : i \in c\}) : c \in \text{Cluster}(\mathcal{D}_{\text{train}})\}). \quad (3)$$

Thus, experience memory is updated as rollouts arrive, insight memory is updated after each problem-level rollout group, and behavior memory is consolidated periodically across related problems.

Memory-conditioned teacher. The teacher is conditioned on the latest available memory. For each problem x_i , PMD directly accesses its problem-specific memories and retrieves relevant behaviors from the global behavior bank:

$$m_{i,t}^{\text{exp}} = M_{t+1}^{\text{exp}}[i], \quad m_{i,t}^{\text{ins}} = M_{t+1}^{\text{ins}}[i], \quad m_{i,t}^{\text{beh}} = \mathcal{R}(M_t^{\text{beh}}, x_i). \quad (4)$$

The teacher memory context is then

$$m_{i,t} = \text{Compose}(m_{i,t}^{\text{exp}}, m_{i,t}^{\text{ins}}, m_{i,t}^{\text{beh}}), \quad (5)$$

where \mathcal{R} retrieves relevant behavior memories conditioned on the current problem.

Let $g_{i,t}$ denote the current-batch context from $\mathcal{T}_{i,t}$, such as verifier feedback or a successful sibling response. The memory-conditioned self-teacher is

$$q_{\theta_t}(\cdot \mid x_i, m_{i,t}, g_{i,t}) := \pi_{\theta_t}(\cdot \mid \text{reprompt}(x_i, m_{i,t}, g_{i,t})). \quad (6)$$

As in SDPO, the teacher is fixed through stopgrad. PMD differs by conditioning the teacher on procedural memory updated from the current policy’s attempts, rather than relying only on episode-local feedback.

Policy update. Following SDPO, the student is trained on trajectories sampled from the current policy and matches the memory-conditioned teacher on the same prefix states:

$$\mathcal{L}_{\text{PMD}}(\theta; \theta_t, M_t) = \mathbb{E}_{x_i, y_i} \left[\sum_{s=1}^{|y_i|} \text{KL}(\pi_{\theta}(\cdot \mid x_i, y_{i,<s}) \parallel \text{stopgrad}(q_{\theta_t}(\cdot \mid x_i, m_{i,t}, g_{i,t}, y_{i,<s}))) \right], \quad (7)$$

where $x_i \in B_t$ and y_i is a rollout from $\mathcal{T}_{i,t}$. The policy is updated as

$$\theta_{t+1} \leftarrow \theta_t - \eta \nabla_{\theta} \mathcal{L}_{\text{PMD}}(\theta; \theta_t, M_t). \quad (8)$$

Feedback from the current rollout explains a local error or success, while procedural memory summarizes what the learner has discovered across repeated attempts: recurring mistakes, reusable strategies, and higher-level behaviors that may not be visible from a single batch. Because the policy changes during training, the memory that guides the teacher must also evolve. A memory bank written once by an earlier policy can drift out of sync with the learner’s current policy. This motivates building memory online, so that the teacher remains compatible with the policy it supervises [14] and the knowledge source stays consistent with the on-policy learner that consumes it [46].

3.2 Online Construction of Procedural Memory

PMD organizes procedural memory into three levels: *experience*, *insight*, and *behavior* memory. During training, the model interacts with the environment through trial and error. Each rollout provides a training signal, such as reward, feedback, or a successful sibling response, and PMD records these signals as experience memory for each problem. The agent then reflects on the accumulated experience for the same problem to extract insights, including strategies that led to success and lessons that explain failure. This is similar in spirit to off-policy learning, where an agent can learn from past experience rather than only from the latest on-policy action [36]. Finally, PMD abstracts across insights from different problems to produce behavior memory: reusable skills or reasoning patterns that recur across problems. Thus, the memory hierarchy moves from concrete attempts, to problem-level insights, to cross-problem behaviors.

Experience memory. As rollouts are produced by the current policy π_{θ_k} , PMD stores raw experience on a per-problem basis. For each training example x_i , the experience memory $M_k^{\text{exp}}[i]$ records concrete attempts, including successful rollouts, failed rollouts, rewards, and environment feedback if available. This is the most faithful memory level: it preserves model’s detailed reasoning. Because experience memory is updated online after each batch, it tracks the evolving policy rather than freezing the perspective of an earlier model.

Insight memory. Experience memory is faithful but often too local and verbose to guide future learning directly. PMD therefore converts stored experience into problem-level insights. For each problem x_i , the insight memory $M_k^{\text{ins}}[i]$ summarizes successful and unsuccessful rollouts into strategies and lessons. Strategies capture reasoning patterns that consistently produce correct answers, while lessons identify recurring mistakes and explain their failures. When both successful and failed attempts are available, extraction is contrastive: the model compares the two groups to identify what differentiates successful reasoning from unsuccessful reasoning, rather than summarizing each rollout independently. PMD also uses the fraction of successful rollouts as a confidence signal, making insight memory reusable, confidence-aware training guidance across epochs.

Behavior memory. The final level abstracts across problems. PMD encodes each question with an embedding model, clusters questions by similarity, and uses an LLM-based abstraction module to

distill the experiences and insights within each cluster into behavior memory M_k^{beh} . Each behavior is a short reusable instruction that captures recurring reasoning patterns, mistakes, or skills across related questions, providing guidance beyond a single training example. This completes the memory hierarchy from experience to insight to behavior.

The hierarchy reflects a trade-off between concrete and abstract memories. Experience memory preserves faithful evidence but remains local; behavior memory transfers broadly but can become coarse; insight memory balances both by preserving compact lessons. PMD studies which memory level best supports learning and generalization. See Appendix Sections B.3, B.4, and B.6 for details.

Memory access. PMD accesses memory according to its scope. Experience and insight memories are problem-specific and are read for the current training example. Behavior memory is global and is retrieved by \mathcal{R} , a dense retrieval function that encodes the current question x_i with an embedding model [51] and returns the top- K most similar behavior entries from the global bank. This keeps lower-level memory grounded in the current problem while allowing behavior memory to transfer reusable procedural guidance across related problems. Memory access occurs only on the teacher path. Rather than creating a memory-augmented inference model, PMD uses memory to let the teacher convert accumulated experience into a stronger target distribution for the student. This distinguishes PMD from retrieval-based memory systems such as RAG [13], MemGPT [20], and Memento [55], which rely on memory at inference time. In PMD, the student requires no inference-time memory, as procedural knowledge is gradually internalized through self-distillation.

4 Experiments

4.1 Experimental setup

Data and Metrics We evaluate PMD in two verifiable domains: **SCIKNOWEVAL**, a science multiple-choice reasoning benchmark [4] covering biology, chemistry, physics, and materials science, and **LiveCodeBench**, a contamination-aware code-generation benchmark with execution-based unit-test feedback [11]. Following SDPO [10], we report **avg@16** on SCIKNOWEVAL, where each rollout is scored as 1.0 if the extracted answer letter matches the ground truth and 0.0 otherwise, and **score@4** on LiveCodeBench, where sparse reward assigns 1.0 only when all unit tests pass. Both settings produce repeated successes, failures, and feedback across epochs, allowing us to study whether episode experience can be compressed into memory and distilled into the policy.

Model and Training We evaluate PMD with two open-source instruction-tuned policies, Qwen3-8B [43] and OLMo3-Instruct-7B [18]. Unless otherwise stated, all experiments are conducted with thinking mode disabled (*think-off*) following the setup in SDPO [10] for fair comparison. During training, we sample $n = 8$ on-policy rollouts per prompt and use the same SDPO-style reverse-KL self-distillation objective. Appendix C provides additional details on data splits, infrastructure, hyperparameters, memory construction, and evaluation protocols.

Baselines We compare PMD against the base policy, GRPO [26, 6], and SDPO [10]. GRPO isolates sparse reward-based optimization, while SDPO isolates feedback-conditioned self-distillation without explicit procedural memory. PMD uses the same self-distillation backbone as SDPO, but conditions the teacher on online procedural memory built from the evolving policy’s own trajectories.

4.2 Procedural memory provides strong self-learning signal

Main results. In Table 2, we show that PMD consistently outperforms both GRPO and SDPO across model families and benchmarks. Relative to SDPO, PMD improves SCIKNOWEVAL AVG from 74.4 to 77.2 for Qwen3-8B and from 69.5 to 73.3 for OLMo3-Instruct-7B, corresponding to relative gains of 3.8% and 5.5%, respectively. On LIVECODEBENCH, PMD improves over SDPO from 47.9 to 51.7 for Qwen3-8B and from 45.0 to 51.1 for OLMo3-Instruct-7B, yielding relative gains of 7.9% and 13.6%. Since the key difference between PMD and SDPO is whether model rollouts are accumulated as memory during training, these results suggest that training-time memories provide a valuable self-learning signal that SDPO does not exploit. In the rest of this section, we present analytical studies to better understand the source of these gains.

Table 2: **Main results on SCIKNOWEVAL and LiveCodeBench.** We compare PMD with the base policy, GRPO, and SDPO. All experiments use thinking-mode disabled (*think-off*). SCIKNOWEVAL reports avg@16, where each rollout is scored as correct if the extracted answer letter matches the ground truth. LiveCodeBench reports score@4 with sparse unit-test rewards, where a rollout receives score 1.0 only if all unit tests pass. SCIKNOWEVAL AVG is the mean over biology, chemistry, physics, and materials. Best results are shown in bold.

Model	Method	SciKnowEval					LiveCodeBench
		Biology	Chemistry	Physics	Materials	AVG	v6
Qwen3-8B	Base Policy [43]	32.4	41.6	58.5	59.2	47.9	27.1
	GRPO [26, 6]	63.2	73.2	70.6	70.7	69.4	41.2
	SDPO [10]	63.6	80.6	72.8	80.4	74.4	47.9
	PMD	68.5	82.8	74.7	82.9	77.2	51.7
OLMo3-Instruct-7B	Base Policy [18]	15.4	22.6	35.9	36.7	27.7	27.7
	GRPO [26, 6]	45.8	70.1	63.3	76.5	63.9	36.1
	SDPO [10]	54.1	79.0	66.1	76.8	69.5	45.0
	PMD	66.1	81.3	68.2	77.4	73.3	51.1

Table 3: **Decomposing PMD’s gain into interacting mechanisms.** *PMD-Transient* is identical to *PMD* but discards memory after every step, isolating within-step reflection from cross-step persistence. *Evolving Memory + Frozen Policy* updates memory while keeping the policy fixed. *Frozen Memory + SDPO* distills from a fixed memory bank, built once and held fixed. SCIKNOWEVAL reports avg@16 and LiveCodeBench reports score@4.

Model	Method	SciKnowEval					LiveCodeBench
		Biology	Chemistry	Physics	Materials	AVG	v6
Qwen3-8B	Base Policy [43]	32.4	41.6	58.5	59.2	47.9	27.1
	SDPO [10]	63.6	80.6	72.8	80.4	74.4	47.9
	Evolving Memory + Frozen Policy	34.3	49.8	60.9	70.8	54.0	35.9
	Frozen Memory + Evolving Policy	44.6	77.7	63.1	74.5	65.0	47.5
	PMD-Transient	67.4	81.8	73.8	79.8	75.7	48.1
	PMD	68.5	82.8	74.7	82.9	77.2	51.7

Decomposing the gain: reflection, persistence and co-evolution. PMD improves learning through three interacting mechanisms: (i) reflective abstraction from rollouts into reusable strategies and lessons, (ii) persistence of these abstractions across optimization steps, and (iii) co-evolution of memory and policy during training. To disentangle these effects, we introduce a set of controlled variants. **PMD-Transient** constructs memory from the current batch, used to condition the teacher for a single optimization step, and then discarded. It retains the full reflection pipeline—rollouts → experience → insights → behaviors → teacher conditioning—but removes cross-step accumulation. Comparing PMD-Transient with SDPO and PMD isolates the contribution of reflective abstractions and persistent memory beyond transient reflection. To further study the role of co-evolution between memory and policy, we consider two additional variants. **Evolving Memory + Frozen Policy** keeps the base policy fixed while continuously updating the memory across epochs, testing whether memory accumulation alone can improve performance in the absence of parameter updates. In contrast, **Frozen Memory + Evolving Policy** first constructs a memory bank using the frozen-policy setup, and subsequently freezes this memory while training the policy using SDPO-style updates. This isolates whether a static repository of distilled strategies is sufficient, or whether gains arise specifically from the iterative co-adaptation between the evolving policy and evolving memory.

Table 3 decomposes PMD’s improvement over SDPO. **Reflection helps even without persistence.** PMD-Transient discards memory after every step, yet still outperforms SDPO (+1.3pp on SCIKNOWEVAL AVG and +0.2pp on LIVECODEBENCH). Structured insights extracted from a single batch already provide a stronger signal than raw context. **Persistence drives most of the gain.** Allowing memory to persist across steps (PMD) yields an additional +1.5pp on SCIKNOWEVAL and +3.6pp on LIVECODEBENCH. The two mechanisms are complementary, but their relative importance depends on task structure: (1) in science MCQ, where a single batch often suffices to uncover the right strategy, reflection and persistence contribute roughly equally; (2) in code generation, where successful patterns are rarer and must be consolidated across many attempts, persistence accounts

for nearly all the improvement. **Evolving memory alone is insufficient.** Updating memory while keeping the policy fixed barely improves performance (54.0 / 35.9): no distillation occurs, only a richer prompt that the model never internalizes. **Static memory loses alignment with an evolving policy.** Training against a memory bank constructed once at initialization performs substantially better (65.0 / 47.5), but still falls well short of PMD because the policy evolves while the memory remains fixed, causing the teacher signal to become stale. Together, these results identify co-evolution as the main driver of PMD’s gains, with the value of persistence increasing alongside task difficulty.

4.3 Transferability and Utility of Learned Memory

We evaluate cross-scale transfer of learned memory (insight+behavior) on SCIKNOWEVAL to understand if it provides additive gains beyond policy quality. The memory is constructed from Qwen3-8B rollouts under both PMD (co-evolving memory) and frozen-policy settings, then evaluated against target models ranging from Qwen3-1.7B to Qwen3-32B. As shown in Figure 2, across all sizes, memory-augmented inference outperforms the no-memory baseline, indicating that retrieved memory encodes reusable task signal rather than model-specific artifacts. Finally, PMD co-evolved memory outperforms the memory evolved with frozen policy.

Two patterns are robust. First, co-evolved memory (PMD) outperforms memory used with a frozen policy, showing that jointly adapting policy and memory improves downstream transfer quality. Second, increasing retrieval depth yields monotonic improvements (top-5 > top-3 > top-1), suggesting that additional retrieved memories are beneficial rather than noisy. Importantly, memory gains are large enough to offset model scale in multiple regimes: a smaller model with deeper retrieval can surpass a larger no-memory model (e.g., 4B with top-5 retrieval > no-memory 8B), and this scaling trade-off continues at higher sizes (e.g., 8B with retrieval exceeding larger no-memory counterparts). Overall, these results establish that memory is transferable across scales and practically useful, and that co-evolving memory with policy is the strongest variant.

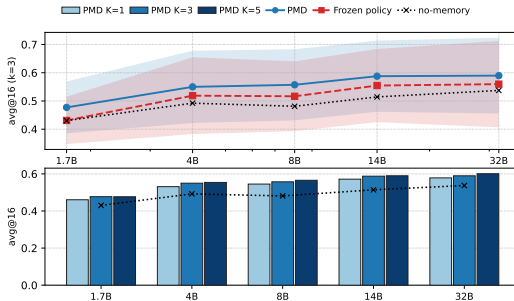


Figure 2: **Memory transfer on SCIKNOWEVAL** Memories are learned from Qwen3-8B under both PMD (co-evolving policy) and frozen-policy settings, then transferred across model scales. Top: PMD vs. frozen-policy memory transfer (shaded bands: cross-domain variability). Bottom: PMD retrieved memories vs. performance ($K \in 1, 3, 5$); dotted black denotes no-memory.

4.4 Test-Time Scaling: PMD preserves the coverage that SDPO trades away

We sample $n=16$ rollouts per validation question and report $\text{maj}@k$ (majority vote, random tie-break) and $\text{best}@k$ (oracle pass-any). In Fig. 3, the shaded band between $\text{maj}@k$ and $\text{best}@k$ is *verifier headroom*: the additional accuracy a perfect verifier could recover beyond self-consistency voting (annotated at $k=16$ as $+\Delta$). Across all SCIKNOWEVAL subjects, PMD preserves answer-space coverage that SDPO collapses. PMD is already better at $k=1$ by 2–5 pp, and the gap grows to 7–10 pp at $k=16$, because PMD continues improving with k while SDPO saturates early. The $\text{maj}@k \rightarrow \text{best}@k$ band is consistently wider for PMD (about 2–4 \times), indicating more recoverable accuracy for downstream reranking; on Material, SDPO’s headroom collapses to zero ($\text{best}@16(=)\text{maj}@16$), a direct signature of mode collapse. Consistently, Fig.4 shows that PMD solves a strictly larger set of problems (coverage (+9)–(14%) vs. SDPO, with only (2)–(4%) SDPO-only solves), confirming that PMD expands useful answer-space coverage rather than merely shifting solved instances. Together, these results show that test-time scaling methods (self-consistency, verifier reranking, best-of-(N)) remain effective under PMD but are limited under SDPO collapse.

4.5 Additional Analyses

The appendix provides supporting analyses. Appendix A.1 compares experience-only memory, experience plus insight, and the full hierarchy with behavior memory, showing that procedural memory works best when distilled into the policy and that behavior memory gives the best average

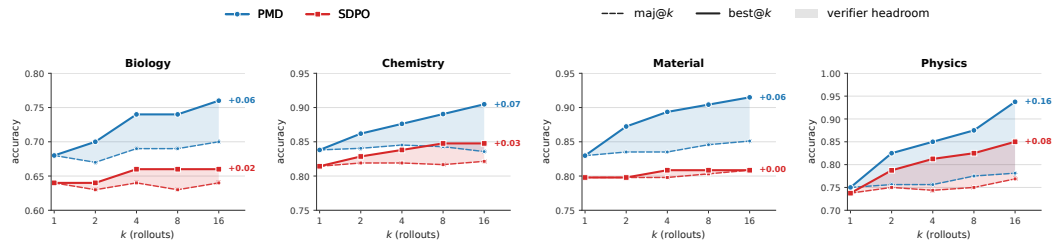


Figure 3: **PMD preserves answer-space coverage that SDPO collapses on SCIKNOWEVAL.** Using 16 rollouts/problem, lines show maj@k and best@k as rollout budget k increases. The shaded band (maj@k \rightarrow best@k) is verifier headroom. PMD’s band is 2–4 \times wider than SDPO across all subjects, indicating greater retained candidate diversity.

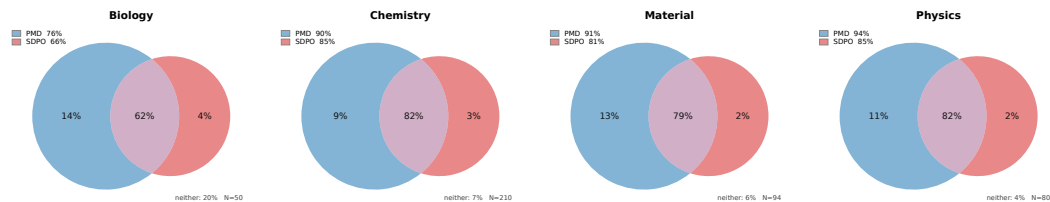


Figure 4: Per-subject Venn diagrams of problems with at least one of 16 rollouts correct (N problems per panel; *neither* = problems neither method solves). PMD’s solved set covers 9–14% more problems than SDPO’s in every domain, while only 2–4% are solved exclusively by SDPO. Together with verifier headroom, this shows that **memory-augmented training preserves coverage of the answer space, whereas SDPO collapses it onto a narrower subset**, limiting inference-time scaling.

PMD performance. Appendix A.2 first probes memory internalization by decoding the student without memory and tracking the increased use of terms such as “strategy”, “lesson”, and “behavior” during training. Appendix A.3 further analyzes internalization through response length and discourse structure, showing that PMD adopts teacher-side procedural reasoning patterns even when decoded without any memory prompt. Finally, Appendix A.4 studies memory bank evolution, showing that lower-level memories collect problem-specific evidence while behavior memory acts as a higher-level consolidation mechanism.

5 Conclusion

We presented PMD, a procedural memory distillation framework that turns repeated training-time experience into reusable supervision for self-improving language models. PMD moves beyond episode-local hindsight by building memory online from the learner’s own trajectories, abstracting it into experience, insight, and behavior memory, and using this evolving memory to condition an on-policy self-teacher. Because memory is used only during training, the final student does not depend on external retrieval at inference time; recurring procedural knowledge is instead gradually internalized into the policy. Our experiments on SCIKNOWEVAL and LIVECODEBENCH show that this co-evolution of memory and policy improves over SDPO and over controls with frozen policies or fixed memories. These results suggest that self-improving models benefit not only from immediate feedback, but also from structured memory that records what they have tried, reflects on what worked or failed, and distills reusable behaviors into the model itself.

Limitations. Our current evaluation focuses on repeated training over fixed task distributions, where problem-local experience can be accumulated across epochs. Although we evaluate two different verifiable domains, science reasoning and code generation, this setting is still narrower than long-horizon agentic environments in which memory, tools, and skills must evolve across heterogeneous tasks [32, 3, 37]. We therefore do not claim that PMD fully solves online agent self-improvement across domains. Rather, our results isolate a more specific question: whether procedural knowledge extracted from repeated attempts can co-evolve with the policy and be distilled into memory-free inference behavior. Extending PMD from problem-local memory to broader behavior retrieval across tasks, and validating it on long-horizon agent benchmarks, are important directions for future work.

References

- [1] Rishabh Agarwal, Nino Vieillard, Yongchao Zhou, Piotr Stanczyk, Sergio Ramos Garea, Matthieu Geist, and Olivier Bachem. “On-Policy Distillation of Language Models: Learning from Self-Generated Mistakes”. In: *International Conference on Learning Representations*. 2024.
- [2] Yuang Cai and Yuyu Yuan. “X-KD: General Experiential Knowledge Distillation for Large Language Models”. In: *arXiv preprint arXiv:2602.12674* (2026).
- [3] Zihao Cheng, Zeming Liu, Yingyu Shan, Xinyi Wang, Xiangrong Zhu, Yunpu Ma, Hongru Wang, Yuhang Guo, Wei Lin, and Yunhong Wang. “Mem2Evolve: Towards Self-Evolving Agents via Co-Evolutionary Capability Expansion and Experience Distillation”. In: *arXiv preprint arXiv:2604.10923* (2026).
- [4] Kehua Feng, Keyan Ding, Weijie Wang, Xiang Zhuang, Zeyuan Wang, Ming Qin, Yu Zhao, Jianhua Yao, Qiang Zhang, and Huajun Chen. “SciKnowEval: Evaluating Multi-level Scientific Knowledge of Large Language Models”. In: *arXiv preprint arXiv:2406.09098* (2024).
- [5] Yuxian Gu, Li Dong, Furu Wei, and Minlie Huang. “MiniLLM: Knowledge Distillation of Large Language Models”. In: *International Conference on Learning Representations*. 2024.
- [6] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. “DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning”. In: *arXiv preprint arXiv:2501.12948* (2025).
- [7] Yinghui He, Simran Kaur, Adithya Bhaskar, Yongjin Yang, Jiarui Liu, Narutatsu Ri, Liam Fowl, Abhishek Panigrahi, Danqi Chen, and Sanjeev Arora. “Self-Distillation Zero: Self-Revision Turns Binary Rewards into Dense Supervision”. In: *arXiv preprint arXiv:2604.12002* (2026).
- [8] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. “Distilling the Knowledge in a Neural Network”. In: *NeurIPS Deep Learning and Representation Learning Workshop*. 2015.
- [9] Chengsong Huang, Wenhao Yu, Xiaoyang Wang, Hongming Zhang, Zongxia Li, Ruosen Li, Jiabin Huang, Haitao Mi, and Dong Yu. “R-Zero: Self-Evolving Reasoning LLM from Zero Data”. In: *arXiv preprint arXiv:2508.05004* (2025).
- [10] Jonas Hübner, Frederike Lübeck, Lejs Behric, Anton Baumann, Marco Bagatella, Daniel Marta, Ido Hakimi, Idan Shenfeld, Thomas Kleine Büning, Carlos Guestrin, and Andreas Krause. “Reinforcement learning via self-distillation”. In: *arXiv preprint arXiv:2601.20802* (2026).
- [11] Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. “LiveCodeBench: Holistic and Contamination Free Evaluation of Large Language Models for Code”. In: *arXiv preprint arXiv:2403.07974* (2024).
- [12] Jeonghye Kim, Xufang Luo, Minbeom Kim, Sangmook Lee, Dohyung Kim, Jiwon Jeon, Dongsheng Li, and Yuqing Yang. “Why Does Self-Distillation (Sometimes) Degrade the Reasoning Capability of LLMs?” In: *arXiv preprint arXiv:2603.24472* (2026).
- [13] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Kuttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. “Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks”. In: *Advances in Neural Information Processing Systems*. 2020.
- [14] Yaxuan Li, Yuxin Zuo, Bingxiang He, Jinqian Zhang, Chaojun Xiao, Cheng Qian, Tianyu Yu, Huan-ang Gao, Wenkai Yang, Zhiyuan Liu, and Ning Ding. “Rethinking On-Policy Distillation of Large Language Models: Phenomenology, Mechanism, and Recipe”. In: *arXiv preprint arXiv:2604.13016* (2026).
- [15] Kevin Lu and Thinking Machines Lab. *On-Policy Distillation*. Thinking Machines Lab: Connectionism. 2025. URL: <https://thinkingmachines.ai/blog/on-policy-distillation/>.
- [16] Zhengxi Lu, Zhiyuan Yao, Jinyang Wu, Chengcheng Han, Qi Gu, Xunliang Cai, Weiming Lu, Jun Xiao, Yueting Zhuang, and Yongliang Shen. “SKILL0: In-Context Agentic Reinforcement Learning for Skill Internalization”. In: *arXiv preprint arXiv:2604.02268* (2026).

- [17] Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegrefe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Sean Welleck, Bodhisattwa Prasad Majumder, Shashank Gupta, Amir Yazdanbakhsh, and Peter Clark. “Self-Refine: Iterative Refinement with Self-Feedback”. In: *Advances in Neural Information Processing Systems*. 2023.
- [18] Team Olmo, Allyson Ettinger, Amanda Bertsch, Bailey Kuehl, David Graham, David Heine-man, Dirk Groeneveld, Faeze Brahman, Finbarr Timbers, Hamish Ivison, et al. “Olmo 3”. In: *arXiv preprint arXiv:2512.13961* (2025).
- [19] Siru Ouyang, Jun Yan, I Hsu, Yanfei Chen, Ke Jiang, Zifeng Wang, Rujun Han, Long T Le, Samira Daruki, Xiangru Tang, et al. “Reasoningbank: Scaling agent self-evolving with reasoning memory”. In: *arXiv preprint arXiv:2509.25140* (2025).
- [20] Charles Packer, Vivian Fang, Shishir G. Patil, Kevin Lin, Sarah Wooders, and Joseph E. Gonzalez. “MemGPT: Towards LLMs as Operating Systems”. In: *arXiv preprint arXiv:2310.08560* (2023).
- [21] Joon Sung Park, Joseph C. O’Brien, Carrie Jun Cai, Meredith Ringel Morris, Percy Liang, and Michael S. Bernstein. “Generative Agents: Interactive Simulacra of Human Behavior”. In: *ACM Symposium on User Interface Software and Technology*. 2023.
- [22] Rafael Rafailov, Archit Sharma, Eric Mitchell, Stefano Ermon, Christopher D. Manning, and Chelsea Finn. “Direct Preference Optimization: Your Language Model is Secretly a Reward Model”. In: *Advances in Neural Information Processing Systems*. 2023.
- [23] Stephane Ross, Geoffrey Gordon, and Drew Bagnell. “A Reduction of Imitation Learning and Structured Prediction to No-Regret Online Learning”. In: *International Conference on Artificial Intelligence and Statistics*. 2011.
- [24] Hejian Sang, Yuanda Xu, Zhengze Zhou, Ran He, Zhipeng Wang, and Jiachen Sun. “On-policy self-distillation for reasoning compression”. In: *arXiv preprint arXiv:2603.05433* (2026).
- [25] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. “Proximal Policy Optimization Algorithms”. In: *arXiv preprint arXiv:1707.06347* (2017).
- [26] Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. “DeepSeekMath: Pushing the Limits of Mathematical Reasoning in Open Language Models”. In: *arXiv preprint arXiv:2402.03300* (2024).
- [27] Idan Shenfeld, Mehul Damani, Jonas Hübotter, and Pulkit Agrawal. “Self-distillation enables continual learning”. In: *arXiv preprint arXiv:2601.19897* (2026).
- [28] Taiwei Shi, Sihao Chen, Bowen Jiang, Linxin Song, Longqi Yang, and Jieyu Zhao. “Experiential reinforcement learning”. In: *arXiv preprint arXiv:2602.13949* (2026).
- [29] Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. “Reflexion: Language Agents with Verbal Reinforcement Learning”. In: *Advances in Neural Information Processing Systems*. 2023.
- [30] Mingyang Song and Mao Zheng. “A Survey of On-Policy Distillation for Large Language Models”. In: *arXiv preprint arXiv:2604.00626* (2026).
- [31] Saksham Sahai Srivastava and Haoyu He. “MemoryGraft: Persistent Compromise of LLM Agents via Poisoned Experience Retrieval”. In: *arXiv preprint arXiv:2512.16962* (2025).
- [32] 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”. In: *arXiv preprint arXiv:2305.16291* (2023).
- [33] Hao Wang, Guozhi Wang, Han Xiao, Yufeng Zhou, Yue Pan, Jichao Wang, Ke Xu, Yafei Wen, Xiaohu Ruan, Xiaoxin Chen, and Honggang Qi. “Skill-SD: Skill-Conditioned Self-Distillation for Multi-turn LLM Agents”. In: *arXiv preprint arXiv:2604.10674* (2026).
- [34] Jiayu Wang, Yifei Ming, Zixuan Ke, Shafiq Joty, Aws Albarghouthi, and Frederic Sala. “SkillOrchestra: Learning to route agents via skill transfer”. In: *arXiv preprint arXiv:2602.19672* (2026).
- [35] Yu Wang, Ryuichi Takano, Zhiqi Liang, Yuzhen Mao, Yuanzhe Hu, Julian McAuley, and Xiaojian Wu. “Mem- α : Learning memory construction via reinforcement learning”. In: *arXiv preprint arXiv:2509.25911* (2025).
- [36] Christopher JCH Watkins and Peter Dayan. “Q-learning”. In: *Machine learning* 8.3 (1992), pp. 279–292.

- [37] Rui Wu, Yifei Li, Yuchen Zhang, Yiming Wang, Xiaodong Li, Lichang Chen, Jinyang Chen, Lei Li, and Xipeng Qiu. “Self-Evolving LLM Agents through an Experience-Driven Lifecycle”. In: *arXiv preprint arXiv:2510.16079* (2025).
- [38] Zijun Wu, Yongchang Hao, and Lili Mou. “TokMem: One-token procedural memory for large language models”. In: *arXiv preprint arXiv:2510.00444* (2025).
- [39] Peng Xia, Jianwen Chen, Hanyang Wang, Jiaqi Liu, Kaide Zeng, Yu Wang, Siwei Han, Yiyang Zhou, Xujiang Zhao, Haifeng Chen, Zeyu Zheng, Cihang Xie, and Huaxiu Yao. “SkillRL: Evolving agents via recursive skill-augmented reinforcement learning”. In: *arXiv preprint arXiv:2602.08234* (2026).
- [40] Teng Xiao, Yige Yuan, Hamish Ivison, Huaisheng Zhu, Faeze Brahman, Nathan Lambert, Pradeep Dasigi, Noah A. Smith, and Hannaneh Hajishirzi. “Meta-Reinforcement Learning with Self-Reflection for Agentic Search”. In: *arXiv preprint arXiv:2603.11327* (2026).
- [41] Wujiang Xu, Zujie Liang, Kai Mei, Hang Gao, Juntao Tan, and Yongfeng Zhang. “A-mem: Agentic memory for llm agents”. In: *Advances in Neural Information Processing Systems* 38 (2026), pp. 17577–17604.
- [42] Sikuan Yan, Xiufeng Yang, Zuchao Huang, Ercong Nie, Zifeng Ding, Zonggen Li, Xiaowen Ma, Jinhe Bi, Kristian Kersting, Jeff Z. Pan, Hinrich Schütze, Volker Tresp, and Yunpu Ma. “Memory-R1: Enhancing large language model agents to manage and utilize memories via reinforcement learning”. In: *arXiv preprint arXiv:2508.19828* (2025).
- [43] An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. “Qwen3 technical report”. In: *arXiv preprint arXiv:2505.09388* (2025).
- [44] Chenxu Yang, Chuanyu Qin, Qingyi Si, Minghui Chen, Naibin Gu, Dingyu Yao, Zheng Lin, Weiping Wang, Jiaqi Wang, and Nan Duan. “Self-Distilled RLVR”. In: *arXiv preprint arXiv:2604.03128* (2026).
- [45] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. “ReAct: Synergizing Reasoning and Acting in Language Models”. In: *International Conference on Learning Representations*. 2023.
- [46] Tianzhu Ye, Li Dong, Qingxiu Dong, Xun Wu, Shaohan Huang, and Furu Wei. “Online experiential learning for language models”. In: *arXiv preprint arXiv:2603.16856* (2026).
- [47] Tianzhu Ye, Li Dong, Xun Wu, Shaohan Huang, and Furu Wei. “On-policy context distillation for language models”. In: *arXiv preprint arXiv:2602.12275* (2026).
- [48] Yi Yu, Liuyi Yao, Yuexiang Xie, Qingquan Tan, Jiaqi Feng, Yaliang Li, and Libing Wu. “Agentic memory: Learning unified long-term and short-term memory management for large language model agents”. In: *arXiv preprint arXiv:2601.01885* (2026).
- [49] Guibin Zhang, Muxin Fu, and Shuicheng Yan. “MemGen: Weaving generative latent memory for self-evolving agents”. In: *arXiv preprint arXiv:2509.24704* (2025).
- [50] Ruixiang Zhang, Richard He Bai, Huangjie Zheng, Navdeep Jaitly, Ronan Collobert, and Yizhe Zhang. “Embarrassingly Simple Self-Distillation Improves Code Generation”. In: *arXiv preprint arXiv:2604.01193* (2026).
- [51] Yanzhao Zhang, Mingxin Li, Dingkun Long, Xin Zhang, Huan Lin, Baosong Yang, Pengjun Xie, An Yang, Dayiheng Liu, Junyang Lin, et al. “Qwen3 embedding: Advancing text embedding and reranking through foundation models”. In: *arXiv preprint arXiv:2506.05176* (2025).
- [52] Zhenyuan Zhang, Xianzhang Jia, Zhiqin Yang, Zhenbo Song, Wei Xue, Sirui Han, and Yike Guo. “MemFly: On-the-Fly Memory Optimization via Information Bottleneck”. In: *arXiv preprint arXiv:2602.07885* (2026).
- [53] Siyan Zhao, Zhihui Xie, Mengchen Liu, Jing Huang, Guan Pang, Feiyu Chen, and Aditya Grover. “Self-distilled reasoner: On-policy self-distillation for large language models”. In: *arXiv preprint arXiv:2601.18734* (2026).
- [54] Wanjun Zhong, Lianghong Guo, Qiqi Gao, He Ye, and Yanlin Wang. “Memorybank: Enhancing large language models with long-term memory”. In: *Proceedings of the AAAI conference on artificial intelligence*. Vol. 38. 17. 2024, pp. 19724–19731.
- [55] Huichi Zhou, Yihang Chen, Siyuan Guo, Xue Yan, Kin Hei Lee, Zihan Wang, Ka Yiu Lee, Guchun Zhang, Kun Shao, Linyi Yang, and Jun Wang. “Memento: Fine-tuning LLM agents without fine-tuning LLMs”. In: *arXiv preprint arXiv:2508.16153* (2025).

Table 4: Generalization of different procedural-memory levels on SCIKNOWEVAL. The frozen-policy setting tests whether memory helps as external context, while PMD tests whether each memory level can be distilled into the policy. AVG is the mean over the four subjects. Best result within each block is shown in bold.

Setting	Memory Level	Biology	Chemistry	Physics	Materials	AVG
Evolving Memory + Frozen Policy (Top-3)	Experience Only	42.3	53.5	59.1	68.8	55.9
	Experience + Insight	50.4	52.5	62.6	67.8	58.3
	Experience + Insight + Behavior	45.6	54.4	59.5	69.4	57.2
PMD	Experience Only	63.0	80.6	73.3	80.4	74.3
	Experience + Insight	67.1	82.1	75.2	78.2	75.7
	Experience + Insight + Behavior	68.5	82.7	74.1	83.6	77.2

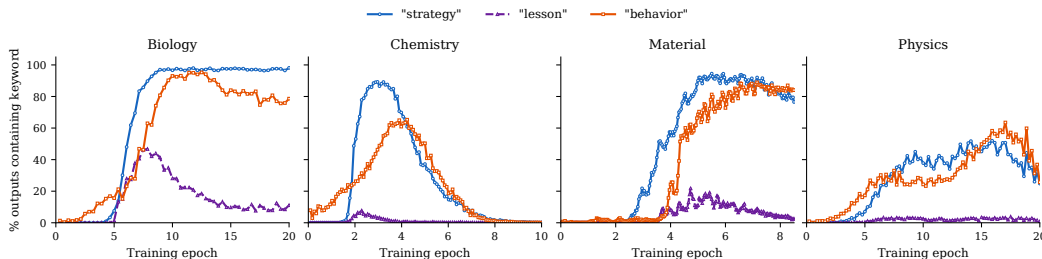


Figure 5: Surface-level probe of procedural-memory internalization. We collect rollouts from student and track the percentage of rollouts containing keywords – “strategy”, “lesson”, and “behavior” – across training steps. The increasing frequency of memory-related terms, together with validation accuracy trends, suggests that PMD gradually transfers teacher-side procedural guidance into the student policy.

A More Experiment Analysis

A.1 Ablation on Memory Granularity

To study the effect of memory abstraction, we compare three memory configurations: *experience only*, *experience + insight*, and the full hierarchy of *experience + insight + behavior*. This tests which level of procedural memory is most useful for reuse and internalization: concrete experience, problem-level insight, or cross-problem behavior [39, 33, 16, 46]. We evaluate these configurations in two settings: *Evolving Memory + Frozen Policy*, where memory can only help as external prompt context, and *PMD*, where memory is used to guide self-distillation and must be internalized into the policy to improve test performance.

Table 4 shows that procedural memory is most effective when it is distilled into the policy rather than used only as external context. In the frozen-policy setting, adding insight improves Biology and Physics, while adding behavior helps Chemistry and Materials; however, the gains are inconsistent because the underlying policy is unchanged. Under PMD, adding insight improves over experience-only memory on Biology, Chemistry, and Physics, suggesting that problem-level strategies and lessons provide a stronger training signal than concrete experience alone. Adding behavior gives the best average performance and achieves the strongest results on Chemistry and Materials, while remaining above experience-only memory on all four domains. These results support our fidelity-transfer trade-off hypothesis: experience preserves faithful local evidence, insight provides compact problem-level guidance, and behavior adds complementary cross-problem knowledge when distilled into the evolving policy.

A.2 Internalization of Procedural Memory

We examine whether PMD can internalize procedural memory into the policy parameters. Prior work on self-distilled RLVR shows that privileged teacher-side information can be pathologically into model weights through distillation [44], making model hallucinate references to ground truth solution.

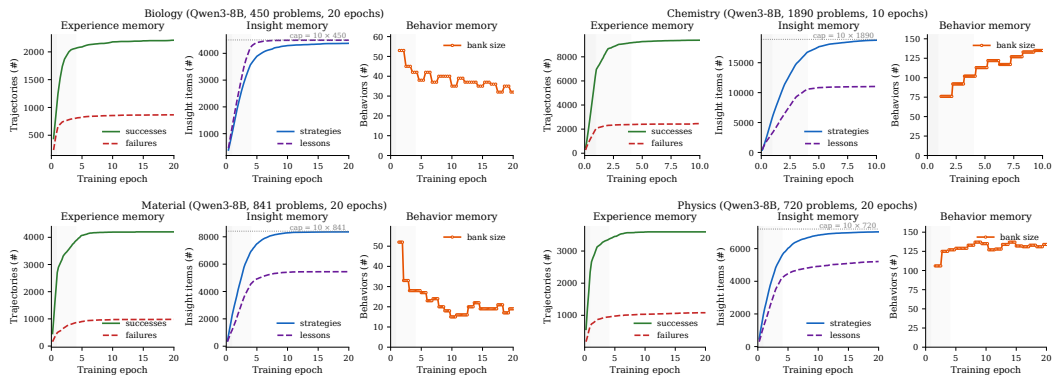


Figure 6: Memory dynamics across SCIKNOWEVAL subjects. We track experience memory, insight memory, and behavior memory during PMD training. Per-problem memories tend to accumulate and saturate, while the global behavior bank shows subject-dependent consolidation dynamics.

Table 5: Response length analysis on SCIKNOWEVAL-Biology. Accuracy follows the same evaluation protocol as Table 2. Token statistics are computed over decoded evaluation responses. PMD produces longer responses than SDPO, but remains well below the length regime of GRPO.

Method	Acc.	Mean	Median	Max	Min
GRPO	63.2	2457	1306	8192	492
SDPO	63.6	186	185	251	120
PMD	68.5	481	474	711	296

PMD use this mechanism for a different purpose: the teacher is not only given the raw solution as feedback, but is also conditioned on abstract memory built from previous rollouts.

At inference time, the student is decoded without any memory prompt. Therefore, if performance improves after training, the gain cannot come from reading external memory at test time. Instead, the useful information from procedural memory must have been absorbed into the model weights during distillation. Figure 5 provides a surface-level probe of this effect. As training proceeds, keyword used by different memory components – “strategy”, “lesson”, and “behavior” – appear more often in the student rollouts, especially in Biology, Material, and Physics. The increase is biggest for “strategy” and “behavior”, suggesting that the trained policy increasingly expresses the procedural concepts that were only available to the teacher during training. Together with improved validation performance, this supports our view of PMD as a training scaffold that converts abstract procedural guidance into persistent model behavior, rather than relying on solution leakage or inference-time memory access.

A.3 Length and discourse-level evidence of internalization.

We further probe internalization by analyzing the length and discourse structure of decoded responses on SCIKNOWEVAL-Biology. At evaluation time, SDPO and PMD are decoded with the same no-memory prompt: neither model can retrieve strategies, lessons, or behaviors. Nevertheless, PMD produces moderately longer responses than SDPO while remaining far shorter than GRPO. As shown in Table 5, PMD improves accuracy from 63.6 to 68.5 over SDPO, while increasing median response length from 185 to 474 tokens. This increase is controlled rather than simply verbose: GRPO generates much longer responses, with a median length of 1306 tokens and generations often reaching the 8192-token cap, yet performs below PMD.

To understand what accounts for the additional tokens, we manually annotate matched SDPO and PMD rollouts with the same inference prompt. Table 6 shows that most of the extra PMD length comes from reusable reasoning machinery rather than generic elaboration. SDPO responses are dominated by a compact qualitative explanation and a final answer block. PMD responses add discourse patterns that mirror the training-time teacher context: per-option comparison and rejection, meta-level justification grounded in previously successful reasoning, and strategy-like phrasing that

Table 6: Discourse-level decomposition of matched SDPO and PMD responses. Counts are approximate token contributions from a hand annotation. The additional PMD tokens mainly correspond to procedural reasoning patterns exposed only to the teacher during training.

Discourse function in response	SDPO	PMD	Likely source in training-time teacher
Qualitative reasoning over the concept or sequence	150	130	Successful sibling rollout
Per-option comparison and rejection	0	~120	Contrastive strategies and lessons
Meta-level justification of the reasoning path	0	~60	Accumulated insight memory
Strategy phrasing resembling named behavior skills	0	~80	Behavior memory
Wrap-up paragraph restating the selected choice	0	~40	Successful-solution template
Final <answer> block	10	10	Output format
Total	~160	~440	–

Table 7: Training overhead relative to SDPO. We report median wall-clock time per training step and peak GPU memory under the same batch size (32), rollout count ($n = 8$), and maximum response length (8192). Results are pooled across the four SCIKNOWEVAL subjects using Qwen3-8B on $8 \times H200$ GPUs. PMD adds teacher-side memory retrieval and asynchronous memory extraction, but does not require additional teacher rollout generation.

Method	Step Time	Relative Time	Peak GPU Memory
SDPO	37.0 s	1.00×	54.6 GB
PMD	53.6 s	1.45×	54.9 GB

resembles behavior-memory entries. These components are not present in the inference-time prompt. They therefore provide qualitative evidence that PMD has absorbed teacher-side procedural structure into the policy weights.

This analysis complements the keyword probe in Figure 5. The keyword probe shows that PMD increasingly emits memory-related terms such as “strategy”, “lesson”, and “behavior” during training. The length and discourse analysis shows a more fine-grained version of the same phenomenon: the trained policy does not merely mention memory terms, but also adopts the procedural form of the teacher distribution. In this sense, PMD internalizes the memory bank as a reasoning style. The memory does not need to be available at test time because its reusable structure has been distilled into the student, at the cost of a moderate, sub-cap increase in response length.

A.4 Memory Bank Size Dynamics.

We track all three-level PMD memory during training on SCIKNOWEVAL 4 subjects with Qwen3-8B. As shown in Figure 6, across all SCIKNOWEVAL subjects, PMD exhibits a consistent separation between per-problem accumulation and cross-problem consolidation. Experience and insight memories grow rapidly during the early epochs as new problems are covered and repeated rollouts add successful trajectories, failed trajectories, strategies, and lessons. Their growth plateaus as the fixed per-problem capacity is approached, especially for lesson memory. In contrast, the behavior bank does not simply accumulate entries. Its dynamics depend on the diversity of the subject: biology and material quickly saturate and then shrink as redundant behaviors are removed, while chemistry and physics continue to admit new behaviors or maintain a larger bank. This suggests that lower-level memories mainly preserve problem-specific evidence, whereas the behavior bank acts as a higher-level consolidation mechanism that retains reusable skills only when they remain distinct and broadly useful.

A.5 Compute Overhead

As shown in Table 7, PMD increases step time by $1.45 \times$ while leaving peak GPU memory nearly unchanged (+0.3 GB). The additional cost mainly comes from longer teacher prompts after memory injection, which increases generation and actor-update time. The memory bank itself is stored off-GPU as text and embeddings, and insight/behavior extraction runs asynchronously. Thus, PMD adds training-time overhead over SDPO, but preserves the same memory-free inference setting and does not introduce extra teacher rollout generation.

B Additional Method Details

This appendix gives implementation details for the memory construction and teacher reprompting used by PMD. The main text describes PMD as an online lifecycle: the model records experience from its own attempts, reflects on that experience into insight, abstracts recurring patterns into behavior, uses the resulting memory to condition the teacher, and distills the teacher signal back into the policy. Here we make this lifecycle concrete and provide the prompt templates used by the memory modules.

B.1 Memory store and update rules

The online memory is organized into three levels: *experience*, *insight*, and *behavior*. These levels differ in scope and abstraction. Experience memory is the most concrete level and stores what the model actually tried on each problem. Insight memory reflects on the accumulated experience for the same problem and extracts compact problem-level guidance. Behavior memory abstracts across semantically related problems and stores reusable reasoning patterns in a global bank. During teacher reprompting, experience and insight are accessed directly through the current training example, while behavior is retrieved from the global behavior bank.

Experience memory. Each problem maintains a bounded set of successful and failed responses, together with rewards and environment feedback when available. In our current implementation, each memory entry stores up to five successful responses and up to three failed response–feedback pairs. Experience memory is updated synchronously after every training batch. Because naive storage quickly collapses into near-duplicate attempts, new responses pass through a diversity-aware novelty gate based on cosine similarity between response embeddings. Near-duplicates are rejected, and when storage is full, the most redundant stored item is evicted. This keeps experience memory faithful to the model’s actual attempts while preserving a diverse set of evidence for later reflection.

Insight memory. After experience memory is updated, an asynchronous extractor reads the accumulated successes and failures for each problem and produces insight memory. Insight memory contains two types of problem-level items: *strategies*, which describe reasoning patterns that led to correct solutions, and *lessons*, which describe recurring mistakes and why they failed. When both successful and failed attempts are available, the extraction is contrastive: the extractor compares the two groups side by side and identifies what separates successful reasoning from unsuccessful reasoning. When only one side is available, the system falls back to a single-sided prompt. Insight items are accumulated across epochs rather than overwritten. New items are embedded, checked for novelty against existing insight items, and added only if they are sufficiently distinct. This allows the per-problem insight to become richer over time without collapsing into paraphrases.

Anti-shortcut filtering. For multiple-choice domains, the insight extractor includes explicit anti-shortcut rules. These rules forbid meta-language about successful or failed attempts, option-letter references, generic test-taking heuristics, and overly problem-specific phrasing. This matters in practice. In early biology runs on SciKnowEval, roughly 39% of extracted insight items were shortcut-like. After adding the anti-shortcut rules to all extraction variants, the contamination rate dropped to roughly 3%. We measure contamination by scanning insight items with pattern-based filters for meta-language, option references, test-taking heuristics, and problem-specific phrasing.

Behavior memory. Behavior memory is a global bank of reusable instructions that transfer across problems. Each behavior contains a name, an instruction, its source, and an embedding used for retrieval. The bank stores up to 500 behaviors in the current implementation.

To construct behavior memory, PMD first encodes each question using Qwen3-Embedding-0.6B and clusters questions by semantic similarity in the embedding space. The LLM abstraction module is then applied to the experience and insight associated with each cluster. Thus, clustering is embedding-based rather than LLM-based; the LLM is used to distill each semantic cluster into compact behaviors. Each behavior captures a recurring reasoning pattern, mistake to avoid, or reusable skill that appears across related problems.

Behavior extraction runs after insight extraction and therefore builds primarily on problem-level insight rather than directly on noisy experience alone. During cold start, behaviors are extracted

from semantic clusters built from the current available summaries. Once the behavior bank contains enough entries, we switch to a retrieve-then-decide update: the extractor first retrieves relevant existing behaviors, then decides whether to add new behaviors, revise existing ones, or remove outdated ones. This keeps the behavior bank curated and stable rather than allowing it to grow as an unstructured list of summaries.

B.2 Teacher reprompt construction

PMD constructs the teacher reprompt by wrapping the original problem with the procedural memory accumulated so far. The memory context contains three types of information: problem-specific strategies extracted from successful rollouts, lessons extracted from failed rollouts, and retrieved behavior memory distilled from related problems. When available, the prompt also includes a previous successful solution and environment feedback from an unsuccessful attempt. The teacher is then asked to solve the original problem under this memory-conditioned context, and the student matches the resulting teacher distribution using the same reverse-KL self-distillation objective as in the main method.

```

Memory-Conditioned Teacher Reprompt

{problem_text}

Relevant strategies from correct solutions to this problem:
{strategies}

Lessons from past failed attempts on this problem:
{lessons}

Reasoning skills --- reusable reasoning patterns learned across all
problems so far. Apply any that are relevant:

1. {behavior_1_name}: {behavior_1_instruction}
2. {behavior_2_name}: {behavior_2_instruction}
...
K. {behavior_K_name}: {behavior_K_instruction}

Correct solution:
{successful_previous_attempt}

The following is feedback from your unsuccessful earlier attempt:
{feedback_raw}

Correctly solve the original question.

```

Here, `{problem_text}` is the original question. The fields `{strategies}` and `{lessons}` are problem-level insight memories extracted from accumulated successful and failed rollouts for the same problem. The behavior entries `{behavior_k_name}` and `{behavior_k_instruction}` are retrieved from the global behavior memory bank and represent reusable reasoning patterns learned across related problems.

The final two optional fields expose raw experience to the teacher. When a previous rollout solved the same problem, `{successful_previous_attempt}` provides that solution as a reference. When a failed rollout has executable or verifier-derived feedback, `{feedback_raw}` provides the corresponding environment feedback. If either field is unavailable, the corresponding block is omitted from the reprompt.

The memory ablations differ only in which parts of this template are exposed to the teacher. In behavior mode, the teacher receives the original problem and retrieved behavior memory. In insight+behavior mode, it additionally receives the problem-specific strategies and lessons. In full mode, it receives the complete context, including strategies, lessons, retrieved behaviors, successful prior attempts, and available feedback. The memory extraction pipeline is unchanged across modes; only the visible memory context differs.

B.3 Prompt templates for insight extraction

Insight extraction uses the student model itself, served separately, as a memory extractor. The prompts are designed to turn accumulated experience into reusable problem-level insight rather than problem-specific summaries. Each insight item is stored as either a strategy or a lesson.

Contrastive prompt. When both successful and failed attempts are available, we use a contrastive prompt. The prompt contains four ingredients: a confidence signal derived from rollout statistics, the original problem statement, a bounded set of successful attempts, and a bounded set of failed attempts with feedback. It then asks the extractor to produce strategies and lessons as the two fields of insight memory.

A simplified version of the prompt is shown below.

Contrastive Insight Extraction Prompt

```
You are a science tutor extracting GENERAL scientific knowledge from a student model's attempts at a multiple-choice question. The goal is to produce reusable knowledge that helps solve ANY similar problem in this domain, not just this specific question.

Confidence signal: {n_success} out of {n_total} attempts were correct
({confidence_label}).

Question and Choices:
{problem_statement}

SUCCESSFUL ATTEMPTS -- {n_success}/{n_total} correct:
{successes_text}

FAILED ATTEMPTS -- {n_failure}/{n_total} wrong:
{failures_text}

Your task: extract GENERAL, TRANSFERABLE scientific principles revealed by comparing the successful and failed attempts. The output must be useful for solving other problems in the same domain, not just this specific question.

Rules:
- strategies: 2--3 general scientific principles or reasoning frameworks.
- lessons: 2--3 common misconceptions or reasoning errors.
- Be domain-specific.
- Each item: short title (<=10 words) + 2--4 sentences.
- Do not reference this question, option letters, attempts, the model, or test-taking shortcuts.

Respond ONLY with valid JSON:
{"strategies": [{"title": "...", "content": "..."}, ...], "lessons": [{"title": "...", "content": "..."}, ...]}
```

The confidence signal calibrates how strongly the extractor should trust the observed pattern. The anti-shortcut rules prevent the extractor from producing answer-choice heuristics instead of reusable domain knowledge.

Single-sided fallbacks. When only successful attempts are available, the extractor produces strategy-style insight. When only failed attempts are available, it produces lesson-style insight. Both fallbacks preserve the same principle: outputs should be domain-specific, phrased as standalone general knowledge, and free of option references, attempt references, and shortcut heuristics.

Strategies-Only Insight Extraction Prompt

```
Strategies-only fallback:
```

You are extracting GENERAL scientific knowledge from correct solutions to a multiple-choice question. The goal is to produce reusable principles that help solve ANY similar problem in this domain, not just this specific question.

Question and Choices:
{problem_statement}

Correct Solution(s):
{solutions}

Respond ONLY with valid JSON:
{ "strategies": [{"title": "...", "content": "..."}, ...] }

Lessons-Only Insight Extraction Prompt

Lessons-only fallback:

You are extracting GENERAL scientific misconceptions from wrong answers to a multiple-choice question. The goal is to identify common reasoning errors that recur across problems in this domain, not just errors specific to this question.

Question and Choices:
{problem_statement}

Wrong Solution(s) with Feedback:
{failures_text}

Respond ONLY with valid JSON:
{ "lessons": [{"title": "...", "content": "..."}, ...] }

Shortcut contamination analysis. To quantify shortcut contamination, we scan extracted insight items with regex patterns corresponding to four categories: meta-language about successful or failed attempts, option-letter or option-value references, generic test-taking heuristics, and problem-specific phrasing such as “this question” or “the given sequence.” A memory item is flagged if it matches any of these categories, and contamination is reported as the fraction of flagged items in the store.

B.4 Prompt templates for behavior extraction

Behavior extraction operates across related problems rather than within a single problem. We first encode each question with Qwen3-Embedding-0.6B and group questions by semantic similarity. For each semantic cluster, we build a short summary consisting of the problem text and the available memory: primarily insight memory, with experience memory used as fallback evidence when insight is not yet available.

Cold-start extraction. When the behavior bank is empty or still small, the extractor infers an initial set of reusable behaviors from the current semantic clusters.

Cold-Start Behavior Extraction Prompt

You are a metacognitive strategist analyzing a cluster of semantically related problems and the model's accumulated memory on them. Your goal is to distill reusable behaviors -- general-purpose reasoning strategies, common pitfalls, and actionable rules -- that transfer across related problems.

A behavior is NOT a solution to a specific problem. It is a reusable skill, a pattern to recognize, a common mistake to avoid, or a strategy to apply. Behaviors should be general enough to help on future unseen problems.

Below are summaries from {n_problems} related problems:

{problem_summaries}

```

Instructions:
1. Identify recurring patterns across these related problems.
2. Extract 3--8 reusable behaviors, each with a name and instruction.
3. Focus on high-level, transferable guidance.
4. Include both positive behaviors and mistakes to avoid.
5. Avoid problem-specific wording, option-letter shortcuts, and references to individual attempts.

Respond ONLY with valid JSON:
{"behaviors": [{"name": "behavior_...", "instruction": "..."}, ...]}

```

Evolution prompt. Once the bank contains existing behaviors, the extractor switches to an evolution prompt. It is shown the current semantic-cluster summaries and the most relevant existing behaviors retrieved by embedding similarity. It then decides whether to create, update, or remove behavior entries.

Behavior Bank Evolution Prompt

```

You are a metacognitive strategist maintaining a behavior bank -- a collection of reusable reasoning patterns, common pitfalls, and actionable rules that transfer across problems.

Below are summaries from {n_problems} semantically related problems:

{problem_summaries}

Here are EXISTING behaviors from your bank that are relevant to these problems:

{existing_behaviors}

Your task is to EVOLVE the behavior bank based on the new evidence:
1. Review existing behaviors against the new cluster evidence.
2. Identify gaps not covered by existing behaviors.
3. Decide on actions: new, update, or remove.
4. Keep behaviors reusable, concise, and independent of any single problem.

Respond ONLY with valid JSON:
{"actions": [
{"action": "new", "name": "behavior_...", "instruction": "..."},
{"action": "update", "name": "behavior_existing_name", "instruction": "..."},
{"action": "remove", "name": "behavior_bad_one"}]}

```

This retrieve-then-decide pattern prevents the behavior bank from becoming an uncurated list of summaries. Instead, the bank is maintained as a compact set of reusable behaviors that remain useful across training.

B.5 Behavior retrieval and persistence

PMD accesses each memory level according to its scope. Experience and insight memories are problem-specific, so during training they are read directly for the current example. Behavior memory is cross-problem, so it is retrieved from the global behavior bank.

For behavior retrieval, we encode the current question with Qwen3-Embedding-0.6B and compare it with the embeddings of stored behaviors using cosine similarity. The top- K most similar behaviors are returned and inserted into the teacher prompt. When feedback is available, we optionally append it to the retrieval query so that the retrieved behaviors can reflect both the problem and the current failure mode. In the main experiments, we retrieve the top- $K = 3$ behavior entries.

All memory levels are periodically persisted to disk, including the experience store, insight store, global behavior bank, and validation outputs. The latest behavior bank is saved after each extraction step and automatically reloaded on restart. The final student is still evaluated without memory retrieval; behavior retrieval is used only on the teacher path during training.

Table 8: A SCIKNOWEVAL chemistry example showing the three PMD memory levels. Experience is problem-specific evidence; insight is a compact problem-level abstraction; behavior is a cross-problem rule distilled from a cluster of related SMILES molar-mass questions.

Memory Level	Example
Experience	<p>Failure: a rollout misreads the ring token <code>C1CC1</code>, overcounts carbon atoms, and selects A. The feedback records that the chosen answer is A while the correct answer is B.</p> <p>Success: another rollout recognizes <code>C1CC1</code> as cyclopropane, accounts for the carbonyl, amide, and aldehyde groups, and selects B.</p>
Insight	<p>Strategies: count all atoms in the molecular formula, including implicit hydrogens omitted by SMILES; use precise atomic weights when answer options are close; decompose complex structures into functional groups before counting.</p> <p>Lessons: avoid overcounting or undercounting atoms in rings; do not ignore implicit hydrogens in SMILES; verify ring tokens such as <code>C1CC1</code> rather than treating them as ordinary linear fragments.</p>
Behavior	<p>The behavior evolver clusters this problem with related SMILES molar-mass questions and produces reusable rules such as:</p> <p>behavior_account_for_implicit_hydrogens: hydrogen atoms are often implicit in SMILES, especially in rings and functional groups, and must still be included in molar-mass calculations.</p> <p>behavior_verify_ring_atom_counts: cyclic structures should be checked explicitly to avoid miscounting ring atoms.</p>

B.6 Qualitative Examples of Procedural Memory

To make the memory hierarchy concrete, we show one example from SCIKNOWEVAL and one from LIVECODEBENCH. In both cases, experience preserves concrete attempts and feedback, insight compresses those attempts into problem-level strategies and lessons, and behavior abstracts recurring patterns across related problems.

SciKnowEval example. Table 8 shows a chemistry question asking for the molar weight of the SMILES string `C1CC1C(=O)NCCC=O`. The correct answer is B, corresponding to a molar weight of 141.17 g/mol. The example illustrates how a concrete atom-counting failure becomes a reusable SMILES-parsing behavior.

LiveCodeBench example. Table 9 shows a code-generation problem asking for the capped geometric sum $X = \sum_{i=0}^M N^i$, printing X if $X \leq 10^9$ and `inf` otherwise. The main traps are the degenerate case $N = 1$, early termination that changes the sum, and modular exponentiation that destroys the exact value needed for a threshold comparison.

These examples show the fidelity-transfer trade-off studied in the main paper. Experience is maximally faithful but tied to a single problem. Insight remains problem-specific but compresses repeated attempts into named strategies and lessons. Behavior removes most problem-specific details and can be retrieved for semantically related unseen problems, but it is necessarily coarser than the underlying experience.

B.7 Algorithm

Algorithm 1 summarizes the PMD training loop.

The algorithm highlights three design choices. First, memory is constructed online from the model’s own attempts under a changing policy, keeping it aligned with the learner. Second, memory is organized as a hierarchy of experience, insight, and behavior, allowing us to study the trade-off between fidelity and transfer. Third, memory is used only through the teacher path. The student does not rely on memory at inference time; retrieval strengthens the training target rather than creating a permanent serving dependency.

Table 9: A LIVECODEBENCH example showing the three PMD memory levels. Experience records concrete code failures and unit-test feedback; insight summarizes the local algorithmic mistakes; behavior abstracts them into reusable coding principles for capped arithmetic and closed-form formulas.

Memory Level	Example
Experience	The rollout group contains five successful and three failed attempts. One failed attempt uses an iterative loop with an off-by-one update and breaks early; the feedback includes cases such as ($N = 21, M = 5$) where the output is too small. Another failed attempt uses modular exponentiation, e.g., <code>pow(N, M+1, 1e9+1)</code> , which changes the value and fails unit tests. A successful attempt handles $N = 1$ separately, computes the closed form exactly without a modulus, and prints <code>inf</code> only after comparing the exact result to 10^9 .
Insight	Strategies: use the geometric-series formula when applicable; handle $N = 1$ as a special case because the formula divides by $N - 1$; use exact arithmetic for the final threshold comparison. Lessons: modular arithmetic should not be used unless the problem asks for a value modulo some number; early termination is unsafe unless the maintained state proves the final capped comparison is unchanged; off-by-one additions can double-count the N^0 term.
Behavior	The behavior evolver clusters this problem with other capped-sum and closed-form arithmetic problems and produces reusable behaviors such as: behavior_handle_formula_degenerate_cases_separately: branch on algebraic degeneracies before applying a closed form, and reject modular arithmetic for overflow or cap decisions. behavior_cap_growth_during_iterative_arithmetic: when a numeric process is capped at a threshold, maintain the running value and stop only when the cap comparison is already determined.

C Experimental Details

This section gives the implementation and evaluation details behind the experiments in Section 4. We keep these details in the appendix so that the main experimental section can focus on the comparison among standard policy optimization, self-distillation, memory-only adaptation, fixed-memory distillation, and the full online PMD training loop.

C.1 Backbone and infrastructure

We train and evaluate two open-source instruction-tuned policies: Qwen3-8B [43] and OLMo3-Instruct-7B [18]. Unless otherwise stated, both model families use the same PMD training recipe, reward functions, memory construction pipeline, and evaluation protocol. All runs use a single node with $8 \times$ H200-143GB GPUs. The policy is FSDP-sharded across all eight GPUs, and rollouts are served with SGLang v0.4 using the same shard. We colocate a second model server on GPU 7 in bf16 mode without CUDA graph to serve the student-contrastive memory extractor; this extractor uses the same backbone family as the policy being trained and shares the rollout/FSDP GPU through `mem-fraction-static=0.2`. Optimization uses AdamW in fp32, and rollout serving uses `gpu_memory_utilization=0.5`. For experiments with an evolving global behavior bank, the behavior-bank evolver is GPT-5.4 and is called at most once per training step after the warm-up period. Behavior clustering and behavior retrieval use Qwen3-Embedding-0.6B for both policy families.

C.2 Datasets and rewards

Table 10 summarizes the datasets. SciKnowEval uses held-out science multiple-choice questions and a rule-based reward that extracts the final answer letter [4]. LiveCodeBench uses half unit test of LiveCodeBench v6 is used for training, and the full v6 unit test set is reserved for evaluation [10]. The code reward scorer executes generated solutions against problem-specific unit tests and uses sparse rewards.

Algorithm 1 Procedural Memory Distillation

```
1: Input: dataset  $\mathcal{D}$ , student policy  $\pi_\theta$ , memory store  $\mathcal{M} = \{M^{\text{exp}}, M^{\text{ins}}, M^{\text{beh}}\}$ 
2: for epoch = 1, ...,  $E$  do
3:   for batch  $B \subset \mathcal{D}$  do
4:     Sample rollout groups  $\{y_i\}_{i \in B}$  from  $\pi_\theta(\cdot | x_i)$ 
5:     Obtain rewards and environment feedback
6:     for each problem  $i \in B$  do
7:       Update experience memory  $M^{\text{exp}}[i]$  with new successes, failures, rewards, and
       feedback
8:       Update insight memory  $M^{\text{ins}}[i]$  by reflecting on accumulated experience
9:     end for
10:    Encode questions with embedding model and cluster related problems by semantic
    similarity
11:    Update behavior memory  $M^{\text{beh}}$  by abstracting cluster-level experience and insight
12:    for each problem  $i \in B$  do
13:      Access  $m_i^{\text{exp}} = M^{\text{exp}}[i]$  and  $m_i^{\text{ins}} = M^{\text{ins}}[i]$ 
14:      Retrieve  $m_i^{\text{beh}} = \mathcal{R}(M^{\text{beh}}, x_i)$  using embedding model
15:      Compose teacher memory  $m_i = \text{Compose}(m_i^{\text{exp}}, m_i^{\text{ins}}, m_i^{\text{beh}})$ 
16:      Construct teacher context reprompt( $x_i, f_i, m_i$ )
17:    end for
18:    Compute  $\mathcal{L}_{\text{PMD}}$  and update  $\theta$ 
19:  end for
20: end for
21: Return: distilled policy  $\pi_\theta$ 
```

Table 10: Dataset splits and reward used in our experiments.

Benchmark	Domain	#Train	#Test	Reward
SciKnowEval—Biology	MCQ	450	50	Rule-based final choice A–D
SciKnowEval—Chemistry	MCQ	1,890	210	Rule-based final choice A–D
SciKnowEval—Materials	MCQ	841	94	Rule-based final choice A–D
SciKnowEval—Physics	MCQ	720	80	Rule-based final choice A–D
LiveCodeBench v6 train / test	Code	924	131	Per-unit-test pass rate

C.3 Training hyperparameters

We use the SDPO objective with reverse KL between the rollout-conditioned student and a teacher reprompt enriched with retrieved memory [10]. Table 11 lists the main hyperparameters. Entries that differ between SciKnowEval and LiveCodeBench are shown separately.

C.4 Memory module and bank operating modes

The teacher reprompt receives three memory levels in order: experience, insight, and behavior. Experience memory contains concrete student attempts and is always included in the full-memory setting, with each problem capped at five successful rollouts and three failed rollouts. Insight memory contains problem-level strategies and lessons distilled by the student-contrastive extractor; we use the MCQ extractor for SciKnowEval and the code extractor for LiveCodeBench. For each policy family, the extractor uses the same backbone family as the policy being trained. Extraction runs concurrently with `llm_concurrency=4`, uses the combined conditioning mode, and accesses problem-keyed experience and insight memory exactly when available.

Behavior memory is a global bank of cross-problem behaviors. It is retrieved by dense similarity with Qwen3-Embedding-0.6B over the current problem text for both Qwen3-8B and OLMo3-Instruct-7B experiments. Because semantically similar tasks may still require different procedures, we use behavior retrieval only on the teacher path and keep the final student memory-free at inference time,

Table 11: Training hyperparameters for SciKnowEval and LiveCodeBench.

Setting	SciKnowEval	LiveCodeBench
Backbone	Qwen3-8B / OLMo3-Instruct-7B	Qwen3-8B / OLMo3-Instruct-7B
Total epochs	20	20
Optimizer	AdamW (fp32)	AdamW (fp32)
Learning rate	1×10^{-5}	1×10^{-6}
LR warmup steps	10	0
Train batch size	32	32
PPO mini-batch size	32	1
Rollouts per prompt n	8	8
max prompt length	2,048	2,048
max response length	8,192	8,192
max model len	18,944	18,944
Rollout sampling	$T = 1.0, \text{top-}p = 1, \text{top-}k = -1$	$T = 1.0, \text{top-}p = 1, \text{top-}k = -1$
Validation samples	16 samples	4 samples
Validation interval	Every 5 steps	Every 5 steps
SDPO α	0.5	1.0
distillation topk	100	20
EMA update rate	–	0.01
Advantage estimator	GRPO, no critic	GRPO, no critic
Rollout-IS correction	Per-token	Per-token
Reasoning template	enable_thinking=False	enable_thinking=False
Reward	Rule-based MCQ	Sparse per-unit-test reward

partly to avoid the persistent imitation risks observed in experience-retrieval systems [31]. We retrieve the top- $K = 3$ behaviors per problem and inject them as a behavior-memory block.