Harnessing LLM Agents with Skill Programs

Anonymous ACL submission

Abstract

Equipping LLM agents with reusable skills derived from past experience has become a popular and successful approach for tackling complex and long-horizon tasks. However, such lessons are often encoded as textual guidance that remains largely advisory, lacking explicit mechanisms for when and how to intervene in the agent loop. To bridge the gap, we introduce HASP (**H**arnessing LLM **A**gents with **S**kill **P**rograms), a framework that upgrades skills into executable Program Functions (PFs). Rather than offering passive advice, PFs act as executable guardrails that activate on failure-prone states and modify the next action or inject corrective context. HASP is highly modular: it can be applied at inference time for direct agent-loop intervention, during post-training to provide structured supervision, or for self-improvement by evolving validated, teacher-reviewed PFs. Empirically, HASP drives substantial gains compared to both training-free and training-based methods on web-search, math reasoning, and coding tasks. For example, on web-search reasoning, inference-time PFs alone improve the average performance by 25% compared to (multi-loop) ReAct Agent, while post-training and controlled evolution achieve a 30.4% gain over Search-R1. To provide deeper insights into HASP, our mechanism analysis reveals how PFs trigger and intervene, how skills are internalized, and the requirement for stable skill library evolution.

1 Introduction

Recent advances in large language models (LLMs) enable increasingly capable agents (Schick et al., 2023; Wu et al., 2024; Li et al., 2025b) that plan, interact with environments, and solve complex tasks effectively (Jin et al., 2025; Jiang et al., 2025). However, as task distributions shift and feedback accumulates across episodes, many agent failures recur in recognizable forms. Multi-step agents may terminate before verification, commit to brittle intermediate conclusions, or repeat unproductive actions (Cemri et al., 2026). A central challenge, therefore, is to enable agents to recognize recurring failure patterns, abstract them into reusable knowledge or skills, and adapt future behavior accordingly (Shinn et al., 2023; Lu et al., 2026).

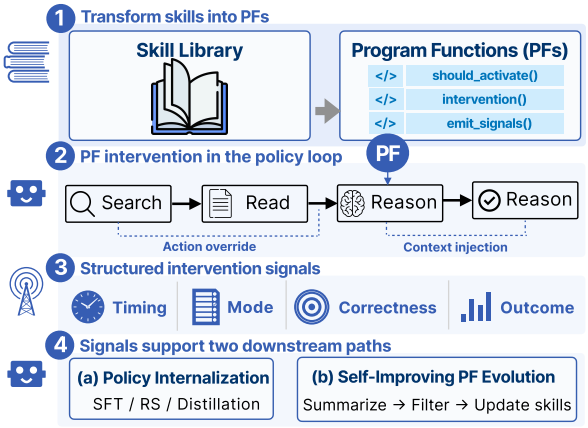

Figure 1: Overview of HASP. Unlike prompted skills, HASP represents skills as Program Functions (PFs) that activate on runtime states, intervene through action modification or context injection, and support training and validated skill evolution.

A natural response is to reuse past experience as *skills*, as behavioral knowledge abstracted from prior agent interactions. Existing agent systems already do so, but mostly in *textual* form (Shinn et al., 2023; Lu et al., 2026; Zhao et al., 2024): they are injected into prompts, retrieved as advice, or used indirectly to shape rewards during post-training. This makes them flexible, but also largely *advisory*. A textual skill can express what the agent should do in principle, but not precisely *when* it should activate inside the policy loop, *how* it should alter the next decision, and it is often ignored by the model in practice. As a result, there remains a gap between reusable experience expressed in language and reusable experience that can reliably and explicitly control agent behavior.

To bridge this gap, we present

HASP (**H**arnessing LLM **A**gents with **S**kill **P**rograms), a framework that reframes skills into executable *Program Functions* (PFs). As shown in Figure 1, PF is a reusable state–action intervention function: given the current agent state and a candidate next action, it decides whether intervention is needed and, if so, explicitly modifies or augments the policy. In this way, a skill is no longer a passive guideline the model may choose to follow; it becomes an executable object that can be triggered on demand and can intervene directly in the agent loop. HASP operates as an external *agent harness*, a control layer around the base agent: at each step, the base policy proposes an action, the harness retrieves relevant PFs, evaluates their activation predicates, executes valid interventions, and feeds the revised action or injected context back into the loop.

We initialize PFs from failure cases in the training pool, instantiate them with explicit activation and intervention interfaces, and admit them into the active library after syntax and mock-execution validation. The same state-action intervention interface makes HASP modular. At inference time, HASP can be plugged into existing agent loop to revise actions or inject corrective context without model updates. When post-training is available, each PF execution provides a structured record containing the original action, the PF repair, the activated skill, and the observed effect. HASP scores these events with structured PF-derived criteria, and uses the resulting PF-corrected traces to train the student base policy via SFT, rejection sampling, or on-policy distillation. Finally, when self-improvement is enabled, HASP revisits failures under the current checkpoint, summarizes recurring failure-repair patterns into candidate PFs, filters them through executable validation and teacher review, and updates the external skill library, thereby closing the loop between execution, learning, and library growth.

We evaluate HASP on web-search reasoning, mathematical reasoning, and coding. Even in the inference-time PF intervention, where PFs trigger based on agent's state, HASP achieves large gains over competitive baselines, underscoring the inherent value of PF design: on web-search reasoning, average accuracy rises to 51.0%. Adding auxiliary teacher for PF selection further improves inference-time performance to 56.2%. Beyond inference-time intervention, PF-derived supervision can be internalized without full reinforcement learning: under fixed skill library, rejection sampling reaches 59.3%, and on-policy distillation reaches 62.5% on web-search reasoning. Controlled evolution further improves the external skill library when paired with stable selection, with closed-loop rejection sampling reaching 60.3% on web-search reasoning and improving mathematical reasoning to 45.4%. On coding, teacher-augmented PF intervention reaches 68.7% average pass@1, while PF-based training improves it to 69.9%. Mechanism analysis further studies how PFs trigger and intervene, how skills are internalized, and the requirement for stable library evolution.

Our contributions are threefold: (1) **Skills as executable state-action intervention functions.** We transform reusable agent experience into executable Program Functions that can be triggered on demand and intervene explicitly in the agent loop, moving beyond passive textual skills. (2) **HASP: a highly-modular agent harness framework.** We propose HASP, an agent harness that supports controllable PF triggering and intervention, effective across inference-only, post-training, and self-improving paradigms. (3) **Strong empirical performance**. HASP achieves substantial gains over competitive baselines across a wide range of tasks.

2 Harnessing LLM Agents with Skill Programs

We present **HASP**, a framework that turns agent experience into executable state–action intervention functions through *Program Functions* (PFs). PFs can be inserted into an agent's reasoning process and triggered while the agent is solving a task, allowing them to correct intermediate decisions. Each PF execution leaves an intervention record of what the agent planned to do, how the PF changed it, and what happened afterward. These records can be used for post-training, while repeated failure patterns are turned into new candidate PFs and added to the skill library after validation. Figure 2 provides an overview of the framework and how PFs intervene in the reasoning process.

2.1 Inference-Time Agent Harness with Program Functions

We consider an agent that reasons step by step and uses external tools while solving a task. Its policy is denoted by π_θ. Given input x, the agent produces a sequence of steps $\tau = (s_0, a_0, o_1, \ldots, s_T, a_T)$, where s_t is agent's current state, a_t is the next ac-

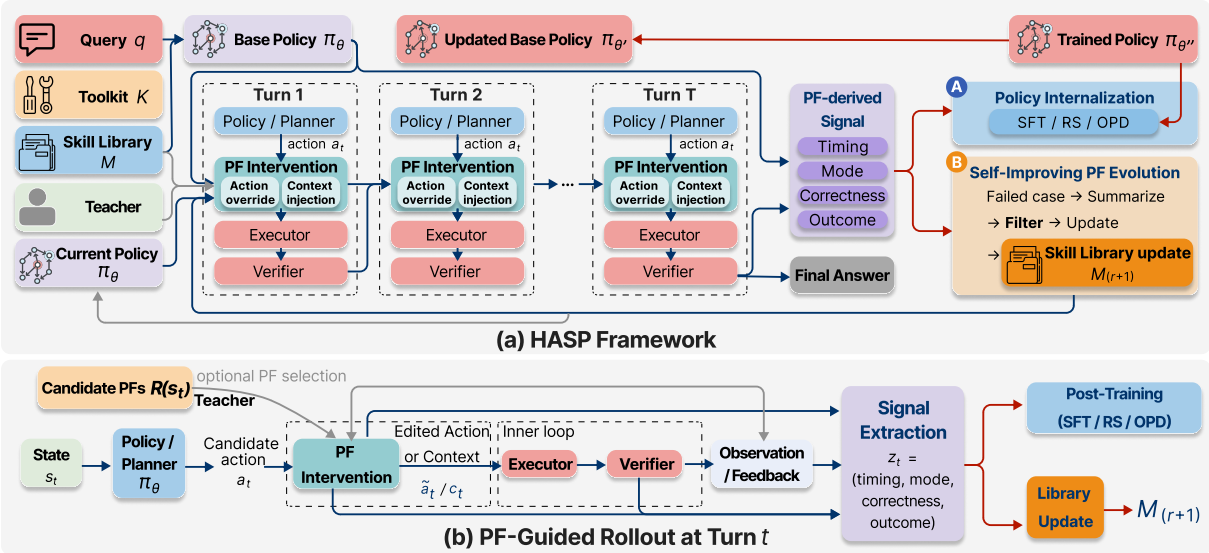

Figure 2: Overview of HASP. (a) At inference time, retrieved Program Functions (PFs) guide multi-turn agent rollouts by modifying actions or injecting context, while emitted signals support policy internalization and PF evolution. (b) A PF-guided turn converts the policy proposal, intervention, execution result, and feedback into signals for post-training and skill library update.

tion it proposes, and o_{t+1} is the result returned by the environment or external tools. At inference time, HASP wraps this base policy with an external *harness*, a control layer that retrieves relevant PFs from the skill library and lets them intervene before the next action is executed, enabling direct correction and improvement of intermediate decisions. The agent has access to an external toolkit K for tool execution and environment interaction.

Program Functions and Skill Library. Each skill in HASP is represented as a *Program Function* (PF), an executable module that decides whether to intervene under the current state and proposed action, and if so, how to repair the next decision. It has two parts: `should_activate`, which decides whether the skill should fire, and `intervene`, which returns the repair. Compared with natural-language reminders, PFs make skills explicit and executable: instead of merely stating a principle such as "avoid repeated searches," a PF specifies both when that principle applies and how the next decision should change. We maintain an external skill library $\mathcal{M}$. To initialize this library, we collect recovered failure cases from the training pool and summarize recurring failure–repair patterns into reusable candidate PFs (*e.g.,* entity confusion). Each candidate PF must specify both an activation condition and an intervention behavior, and is admitted into the library only after syntax, interface, and mock-execution validation. This ensures that the skill library contains executable and reusable interventions.

PF-guided intervention in the agent loop. At step t, the base policy first proposes an action $a_t^{\text{orig}} \sim \pi_\theta(\cdot \mid s_t)$. The harness then retrieves candidate PFs $\mathcal{R}(s_t) \subseteq \mathcal{M}$ and evaluates their activation functions on the current state and proposed action. In the PF-only setting, PFs are triggered solely by these activation functions. The harness then applies the activated PFs through an intervention operator Γ, producing $(\tilde{a}_t, c_t, \kappa_t) = \Gamma(s_t, a_t^{\text{orig}}, \mathcal{R}(s_t))$, where $\tilde{a}_t$ denotes the final action after PF intervention, c_t is optional corrective context injected into subsequent reasoning, and κ_t records the fired PFs and intervention mode. If no PF activates, then $\tilde{a}_t = a_t^{\text{orig}}$; otherwise, $\tilde{a}_t$ may be a modified or redirected action returned by the activated PFs.

PF intervention can operate in two main ways. First, a PF may directly modify the next action itself, yielding a repaired executable action $\tilde{a}_t$; for example, it may rewrite an over-constrained search query, redirect retrieval toward a more informative intermediate entity. Second, a PF may inject corrective context c_t back into the reasoning process, such as a warning about similar entities. This design separates choosing the next action from correcting it: the policy proposes what to do next, while PFs determine whether the proposed action should be executed as is, revised into a better action, or augmented with additional context. The pair $(a_t^{\text{orig}}, \tilde{a}_t)$ therefore records both what the policy would have done and how the harness repaired or redirected it, turning skill use into supervision over intermediate decisions rather than only final-answer feedback.

Auxiliary teacher for PF selection. HASP supports a minimal PF-only intervention setting, where candidate PFs are triggered solely by their activation functions. When available, an auxiliary teacher can further help select which PF to apply when more than one PF could reasonably be applied, improving intervention precision in ambiguous cases.

2.2 HASP for Post-Training: Internalizing PF-Guided Interventions

Beyond improving inference-time behavior, PF-guided intervention also provides supervision during post-training. Each PF activation produces record $e_t = (s_t, a_t^{\text{orig}}, \tilde{a}_t, c_t, \kappa_t, \Delta_t)$, which contains the triggering state, the action proposed by original policy, the corrected action, any injected context, metadata, and the resulting feedback. These events connect inference-time intervention to post-training: they capture when the intervention occurred, how it was applied, and whether it helped. HASP scores each record using four signals $\mathbf{z}_t = (t_t, m_t, q_t, o_t)$, corresponding to intervention timing, mode, correctness, and outcome, and aggregate them into a single per-step score A_t. We further compute trajectory-level PF score as $A(\tau) = T^{-1} \sum_{t=1}^{T} A_t$. Rather than training only on final answers, HASP uses PF-corrected actions and trajectories, scored by these PF-derived signals, to update the student base policy. The goal is to train the base policy toward PF-corrected actions and trajectories. Our main trained variant, **HASP-Evolve + RS**, combines evolving skill library with PF-guided rejection sampling; SFT and OPD are evaluated as alternative training paths.

Main recipe: PF-guided rejection sampling. Given sampled trajectories $\{\tau_i\}_{i=1}^{N}$, HASP scores each trajectory using both final task success and PF-guided intervention quality, with $\text{Score}(\tau_i) = \beta_1 \text{TaskSuccess}(\tau_i) + \beta_2 A(\tau_i)$. Only top-scoring trajectories are retained for training. Unlike standard rejection sampling, which only filters by final correctness, HASP also prefers trajectories whose intermediate decisions better match PF-guided correction: interventions should occur at appropriate states, use suitable modes, and lead to valid repairs with positive downstream effects. This recipe is well suited to the evolving skill library because it filters out noisy or harmful trajectories.

Variant 1: supervised fine-tuning on corrected actions. As a simpler baseline, each intervention record provides a corrected target $s_t \mapsto \tilde{a}_t$, weighted by intervention quality. We optimize $\mathcal{L}_{\text{SFT}} = -\sum_t w_t \log \pi_\theta(\tilde{a}_t \mid s_t)$, where w_t is a monotone function of A_t. This directly trains the student on PF-corrected local actions, such as rewriting over-constrained search queries, enforcing evidence verification before finalization, or using injected context to improve the next decision.

Variant 2: on-policy distillation (OPD). To test whether the scoring scheme also helps on the student's own states, OPD rolls out current policy, keeps PFs active on failure-prone steps, and trains the student on corrected behavior with $\mathcal{L}_{\text{OPD}} = -\sum_t \hat{w}_t \log \pi_\theta(\tilde{a}_t \mid s_t)$, where $\hat{w}_t$ reflects intervention quality under current policy. OPD aligns training with student's own states, but can be less stable when the skill library also changes.

2.3 HASP for Self-Improving Skill Library

The same PF interface also supports controlled growth of the skill library. After fixed training intervals, HASP revisits remaining failures under the current checkpoint and proposes candidate PFs $\mathcal{C}_r$ from recurring failure–repair patterns. Each candidate must specify both an activation condition and an intervention behavior so that it can be inserted into the same rollout harness. To prevent library pollution, candidates are admitted only after executable validation and teacher review: $Q_{\text{exec}}(c)$ checks syntax, mock execution, and legal return types, while $Q_{\text{teach}}(c)$ evaluates whether the candidate captures a reusable failure pattern, fires under appropriate conditions, and proposes a useful repair. A candidate is accepted only if $Q_{\text{exec}}(c) \geq \eta_{\text{exec}}$ and $Q_{\text{teach}}(c) \geq \eta_{\text{teach}}$, after which $\mathcal{M}_{r+1} = \text{Update}(\mathcal{M}_r, \{c \in \mathcal{C}_r : \text{Accept}(c)\})$. This filtering step removes overly specific, redundant, or noisy PFs that would make retrieval less accurate and weaken future interventions. Accepted PFs update the external skill library $\mathcal{M}_r \to \mathcal{M}_{r+1}$, while post-training updates the base policy $\pi_\theta \to \pi_{\theta'}$.

3 Experiment

We evaluate HASP along three axes: whether PFs improve inference-time decisions, whether PF-corrected traces help post-training, and whether remaining failures can be turned into new PFs through filtered evolution.

Tasks and metrics. We report accuracy for web-search and mathematical reasoning, and pass@1 for coding. For web-search reasoning, we use HotpotQA (Yang et al., 2018), 2Wiki (Ho et al., 2020), and MuSiQue (Trivedi et al., 2022). For

Method	Size	Web-search reasoning					Mathematical reasoning				
		HotpotQA	2Wiki	MuSiQue	Avg.	Δ to Ref.	AIME24	AMC23	GameOf24	Avg.	Δ to Ref.
Training-free / inference-time methods Ref. = **HASP-Intervention (Infer.)**											
GPT-4o (Hurst et al., 2024)	∼200B	54.0	49.5	24.0	42.5	↑ 13.7	13.3	60.0	32.0	35.1	↑ 3.7
GPT-4o-mini (Hurst et al., 2024)	∼8B	41.0	35.6	15.0	30.5	↑ 25.7	13.3	57.5	16.0	28.9	↑ 9.9
Qwen2.5-7B-Instruct (Qwen et al., 2025)	7B-Inst	21.0	23.0	6.0	16.7	↑ 39.5	6.7	47.5	33.0	29.1	↑ 9.7
RA-Agent (multi-loop)	7B-Inst	39.5	34.0	20.0	31.2	↑ 25.0	6.7	50.0	46.0	34.2	↑ 4.6
Prompt-Only Skills	7B-Inst	28.0	26.0	7.5	20.5	↑ 35.7	10.0	47.5	41.0	32.8	↑ 6.0
Iter-RetGen (Shao et al., 2023)	7B-Inst	22.0	25.9	6.6	18.2	↑ 38.0	–	–	–	–	–
AutoGen (Wu et al., 2024)	7B-Inst	50.0	44.0	15.9	36.6	↑ 19.6	13.3	57.5	24.0	31.6	↑ 7.2
TIR (Xue et al., 2025)	7B-Inst	–	–	–	–	–	10.0	50.0	33.3	31.1	↑ 7.7
HASP-Intervention (PF-only)	7B-Inst	59.0	67.0	27.0	51.0	↑ 5.2	6.7	55.0	46.0	35.9	↑ 2.9
HASP-Intervention (w. Teacher)	7B-Inst	**64.5**	**70.0**	**34.0**	**56.2**	–	**10.0**	**56.5**	**50.0**	**38.8**	–
Training-based methods Ref. = **HASP-Evolve + RS**											
SFT (vanilla)	7B-Inst	22.0	25.9	6.6	18.2	↑ 42.1	6.7	47.5	33.0	29.1	↑ 16.3
SKILL0 (Lu et al., 2026)	7B-Inst	40.0	38.3	16.4	31.6	↑ 28.7	–	–	–	–	–
Search-R1 (Jin et al., 2025)	7B-Inst	37.0	38.2	14.6	29.9	↑ 30.4	–	–	–	–	–
ZeroSearch (Sun et al., 2025)	7B-Base	34.6	35.2	18.0	29.3	↑ 31.0	–	–	–	–	–
ReSearch (Chen et al., 2025)	7B-Base	43.5	47.6	22.3	37.8	↑ 22.5	–	–	–	–	–
StepSearch (Wang et al., 2025)	7B-Base	38.6	36.6	22.6	32.6	↑ 27.7	–	–	–	–	–
VerlTool (Jiang et al., 2025)	7B-Base	44.8	45.3	19.3	36.5	↑ 23.8	–	–	–	–	–
SimpleRL-reason (Zeng et al., 2025b)	7B-Base	–	–	–	–	–	16.7	60.0	33.0	36.6	↑ 8.8
Open-Reasoner-Zero (Hu et al., 2025)	7B-Base	–	–	–	–	–	16.7	54.9	32.0	34.5	↑ 10.9
General-Reasoner (Ma et al., 2025)	7B-Base	–	–	–	–	–	13.3	55.0	33.0	33.8	↑ 11.6
Luffy (Yan et al., 2025)	7B-Inst	–	–	–	–	–	30.7	44.8	33.0	36.2	↑ 9.2
ToRL (Li et al., 2025a)	7B-Inst	–	–	–	–	–	20.0	60.0	31.0	37.0	↑ 8.4
AgentFlow (w/ Flow-GRPO) (Li et al., 2025b)	7B-Inst	57.0	77.2	25.3	53.2	↑ 7.1	40.0	61.5	53.0	51.5	↓ 6.1
HASP-Evolve + RS	7B-Inst	**69.0**	**74.0**	**38.0**	**60.3**	–	**16.7**	**57.5**	**62.0**	**45.4**	–

Note: For each domain, we report per-dataset accuracy, domain average, and Δ to the reference method in the same block. In the training-free block, the reference is HASP-Intervention (Infer.); in the training-based block, the reference is HASP-Evolve + RS. Methods without results on a given domain are marked as "–".

Table 1: Main results on web-search and mathematical reasoning under training-free and training-based settings.

Method	Size	Metric	HumanEval		MBPP		BigCodeBench		Avg.	Δ to Ref.
			Base	Plus	Base	Plus	Full	Hard		
Training-free / inference-time methods Ref. = **HASP-Intervention (Infer.)**										
GPT-4o (Hurst et al., 2024)	∼200B	pass@1	90.0	81.0	91.0	71.0	41.0	9.0	63.8	↑ 4.8
GPT-4o-mini (Hurst et al., 2024)	∼8B	pass@1	87.0	80.0	88.0	67.0	32.0	12.0	61.0	↑ 7.7
Qwen2.5-7B-Instruct (Qwen et al., 2025)	7B-Inst	pass@1	81.7	73.2	79.4	67.7	45.6	16.9	60.8	↑ 7.9
RA-Agent (multi-loop)	7B-Inst	pass@1	76.0	62.0	82.0	62.0	31.0	14.0	54.5	↑ 14.2
Prompt-Only Skills	7B-Inst	pass@1	86.0	75.0	86.0	66.0	40.0	14.0	61.2	↑ 7.5
HASP-Intervention (PF-only)	7B-Inst	pass@1	84.5	82.0	88.0	65.0	42.0	19.0	63.4	↑ 5.3
HASP-Intervention (w. Teacher)	7B-Inst	pass@1	88.0	85.0	91.0	73.0	49.0	26.0	**68.7**	–
Training-based methods Ref. = **HASP-Evolve + RS**										
SFT (vanilla)	7B-Inst	pass@1	82.0	77.0	76.0	63.0	34.0	13.0	57.5	↑ 12.4
AceCoderRM (Zeng et al., 2025a)	7B-Inst	pass@1	83.5	77.4	83.1	71.2	46.8	16.9	63.2	↑ 6.7
AceCoderRule (Zeng et al., 2025a)	7B-Inst	pass@1	84.1	77.4	80.2	68.3	46.8	15.5	62.1	↑ 7.8
GRPO (Code) (Fan et al., 2025)	7B-Inst	pass@1	85.9	81.1	86.7	75.1	52.0	29.7	68.4	↑ 1.5
P-GRPO (Code+RM) (Fan et al., 2025)	7B-Inst	pass@1	86.6	81.1	87.0	76.2	54.0	33.8	69.8	↑ 0.1
KodCode-RL Step 128 (Xu et al., 2025)	7B-Inst	pass@1	90.2	83.5	81.0	70.9	47.0	19.6	65.4	↑ 4.5
KodCode-RL Step 256 (Xu et al., 2025)	7B-Inst	pass@1	91.5	86.0	82.8	72.8	48.2	19.6	66.8	↑ 3.1
HASP-Evolve + RS	7B-Inst	pass@1	91.0	84.5	92.0	74.0	50.0	28.0	**69.9**	–

Note: For each block, Δ is computed against the reference method in that block: HASP-Intervention (Infer.) for training-free methods and HASP-Evolve + RS for training-based methods. All of our methods use Qwen2.5-7B-Instruct as the backbone unless otherwise specified.

Table 2: Main results on coding under training-free and training-based settings.

mathematical reasoning, we use AIME24 (Zhang and Math-AI, 2024), AMC23 (MAA, 2023), and GameOf24 (Lile, 2024), with answers judged by GPT-4o (Hurst et al., 2024). We evaluate Coding on HumanEval (BASE, PLUS) (Chen et al., 2021), MBPP (BASE, PLUS) (Austin et al., 2021), and BigCodeBench (FULL, HARD) (Zhuo et al., 2024).[1]

Backbone and agent setup. Unless otherwise specified, all methods use `Qwen/Qwen2.5-7B-Instruct` (Qwen et al., 2025) and follow the PF-augmented multi-step agent setup described in Section 2. The initial skill library is derived from recurring recovered failure–repair patterns in the training pool, and full PF families and examples are provided in Appendix C. When used, the teacher is restricted to PF selection or teacher trajectories for distillation.

Training setup and baselines. Our main trained variant is HASP-Evolve + RS, combining closed-loop PF evolution with PF-conditioned rejection sampling. For ablation, we evaluate six post-training settings: fixed-library HASP + SFT/RS/OPD and closed-loop HASP-Evolve + SFT/RS/OPD. In OPD, GPT-4o provides teacher trajectories with active PF interventions. Across all post-training variants of HASP, we use LoRA-based training setup; implementation details are

[1]Test-set and training-pool for web-search and math follow AgentFlow verbatim; for coding we adopt the same training-data protocol. Details are reported in Appendix C.1.

Setting	PF	Teacher	T	M	C	O	Exec	Teach	HotpotQA	2Wiki	MuSiQue	Avg.	Δ
Inference-time component ablation													
Full HASP-Intervention (Infer.)	✓	✓	✓	✓	✓	✓	N/A	N/A	**64.5**	**70.0**	**34.0**	**56.2**	–
RA-Agent (multi-loop)	–	–	✓	✓	✓	✓	N/A	N/A	39.5	34.0	20.0	31.2	↓ 25.0
PF only	✓	–	✓	✓	✓	✓	N/A	N/A	59.0	67.0	27.0	51.0	↓ 5.2
Teacher only	–	✓	✓	✓	✓	✓	N/A	N/A	59.0	69.0	24.0	50.7	↓ 5.5
Signal ablation (on E5)													
All four signals	✓	✓	✓	✓	✓	✓	✓	✓	**69.0**	**74.0**	**38.0**	**60.3**	–
w/o Timing	✓	✓	–	✓	✓	✓	✓	✓	68.0	60.0	29.5	52.5	↓ 7.8
w/o mode	✓	✓	✓	–	✓	✓	✓	✓	55.0	52.0	27.5	44.8	↓ 15.5
w/o Correctness	✓	✓	✓	✓	–	✓	✓	✓	62.0	53.0	29.5	48.2	↓ 12.1
w/o Outcome	✓	✓	✓	✓	✓	–	✓	✓	55.5	59.0	28.0	47.5	↓ 12.8
Filtering ablation (on E5)													
Evolution, full filtering	✓	✓	✓	✓	✓	✓	✓	✓	**69.0**	**74.0**	**38.0**	**60.3**	–
No evolution	✓	✓	✓	✓	✓	✓	–	–	69.5	72.5	36.0	59.3	↓ 1.0
Evolution, no filtering	✓	✓	✓	✓	✓	✓	–	–	35.5	58.0	15.5	36.3	↓ 24.0
Evolution, exec-only	✓	✓	✓	✓	✓	✓	✓	–	61.0	61.0	24.5	48.8	↓ 11.5
Evolution, teacher-only	✓	✓	✓	✓	✓	✓	–	✓	59.0	57.0	25.5	47.2	↓ 13.1

Note: T/M/C/O denote timing, mode, correctness, and outcome; Exec/Teach denote executable validation and teacher review.

Table 3: Ablations on web-search reasoning. Δ is computed relative to the first row of each block.

Web-search reasoning						
Method	**Size**	**HotpotQA**	**2Wiki**	**MuSiQue**	**Avg.**	**Δ vs PFs**
HASP-Intervention (Infer.)	7B-Inst	64.5	70.0	34.0	56.2	–
HASP + SFT	7B-Inst	66.0	69.0	35.5	56.8	+0.6
HASP + RS	7B-Inst	69.5	72.5	36.0	59.3	+3.1
HASP + OPD	7B-Inst	64.5	69.0	54.0	62.5	+6.3
HASP-Evolve + SFT	7B-Inst	68.5	70.0	37.0	58.5	+2.3
HASP-Evolve + RS	7B-Inst	69.0	74.0	38.0	60.3	+4.1
HASP-Evolve + OPD	7B-Inst	65.0	69.5	35.5	56.7	+0.5
Mathematical reasoning						
Method	**Size**	**AIME24**	**AMC23**	**GameOf24**	**Avg.**	**Δ vs PFs**
HASP-Intervention (Infer.)	7B-Inst	10.0	56.5	50.0	38.8	–
HASP + SFT	7B-Inst	13.3	58.5	51.0	40.9	+2.1
HASP + RS	7B-Inst	16.7	57.5	54.0	42.7	+3.9
HASP + OPD	7B-Inst	16.7	57.5	53.0	42.4	+3.6
HASP-Evolve + SFT	7B-Inst	13.3	59.0	52.0	41.4	+2.6
HASP-Evolve + RS	7B-Inst	16.7	57.5	62.0	45.4	+6.6
HASP-Evolve + OPD	7B-Inst	16.7	60.0	55.0	43.9	+5.1

Note: For each domain, we report per-dataset accuracy, domain average, and improvement over the inference-time PF system, HASP-Intervention (Infer.). For both reasoning domains, Δ is computed against the corresponding average of HASP-Intervention (Infer.).

Table 4: Ablation of HASP training and evolution strategies. We report results on six HASP variants: fixed-library SFT/RS/OPD and their evolved counterparts.

provided in Appendix C. Web-search and mathematical baselines follow AgentFlow (Li et al., 2025b), while coding baselines are taken from reported results or evaluated under our coding split.

3.1 Main Results: PF Intervention, Selection, and Internalization

We present the main results in four stages: whether executable PFs alone improve inference-time decisions, whether auxiliary teacher selection further improves PF dispatch, whether PF-derived events can be internalized through post-training, and whether closed-loop PF evolution further improves the external skill library.

Inference-time PF intervention outperforms strong baselines. The upper block of Table 1 and Table 2 compare PF-only intervention and PF intervention with auxiliary teacher selection. PF-only already outperforms the multi-loop ReAct agent and Prompt-Only Skills, showing that directly changing actions or adding corrective context is more effective than injecting skill text alone. On web-search reasoning, PF-only improves average accuracy to 51.0%, compared with 20.5% for Prompt-Only Skills. On coding, PF-only reaches 63.4% average pass@1. On mathematical reasoning, PF-only reaches 35.9%, compared with 32.8% for Prompt-Only Skills. Adding auxiliary teacher selection further improves performance to 56.2% on web-search reasoning, 38.8% on mathematical reasoning, and 68.7% average pass@1 on coding.

PF-derived scores provide effective supervision for post-training. Table 4 evaluates whether PF-corrected traces improve the trained student under the same PF-guided pipeline. Under a fixed skill library, all three post-training variants improve over inference-time intervention alone. On web-search reasoning, performance increases to 56.8%, 59.3%, and 62.5% for SFT, RS, and OPD, respectively, while on mathematical reasoning, the corresponding increases are from 38.8% to 40.9%, 42.7%, and 42.4%. Training dynamics shown in Figure 4 of App G are consistent with this pattern: SFT is stable but gives smaller gains, RS offers the best overall balance, and OPD reaches the strongest fixed-library web-search result. Relative to prior training-based baselines, PF-guided post-training is already competitive with strong training-based baselines, especially on web-search reasoning.

Closed-loop PF evolution further improves the external skill library. Table 4 evaluates closed-loop evolution, where failure cases are summarized into candidate PFs and filtered before updating the external library. Our main trained variant, HASP-Evolve + RS, further improves over fixed-library RS to 60.3% on web-search reasoning and 45.4% on mathematical reasoning, and reaches 69.9% average pass@1 on coding. In contrast, HASP-

Recovered failure family	Representative PF	Recovered proportion	Typical pattern	Avg. ΔEM / ΔAcc.
Reasoning drift / unsupported finalization	`reasoning_hallucination`	31.7%	`FINAL` → `SEARCH(question)`	+13.3 / +9.3
Premature finalization	`premature_final`	17.8%†	early `FINAL` → `SEARCH`	+11.1 / +4.4
Insufficient evidence before final	`no_read_before_final`	17.8%†	no-read `FINAL` → `SEARCH(verify)`	+0.4 / +0.8
Wrong entity focus	`wrong_entity_focus`	24.1%	wrong entity → `SEARCH(entity)`	+6.7 / +5.0
Harmful PF override	`pf_override_harmful`	8.7%	override → fallback `SEARCH`	−0.8 / −0.8

Generated skill review scores (scale 0–1)

Concept	Trigger	Intervention	Executability	Validation / Avg.
0.83	0.75	0.83	0.95	0.80 / 0.83

† joint proportion of the *exploration_control* cluster, which contains both PFs.

Table 5: Recovered failure structure and utility of representative evolved PF families on the web-search benchmark. The lower block reports average review scores of generated skills.

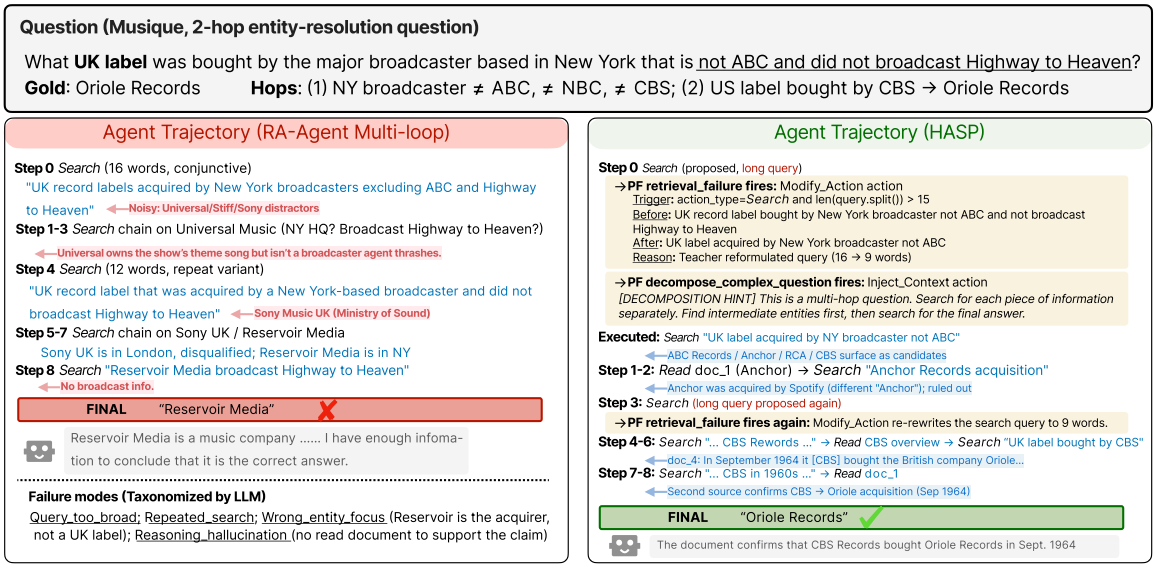

Figure 3: Case study on a MuSiQue two-hop entity-resolution question.

Evolve + OPD drops to 56.7% on web-search reasoning, showing that updating the model and the skill library together can be less stable. Training dynamics are consistent with this pattern: fixed-library runs are smoother, while closed-loop evolution introduces more variation, which works better with RS than with OPD. Overall, HASP-Evolve + RS gives the strongest overall results among our open-weight settings, while staying competitive with GRPO and KodCode-RL on coding.

3.2 Mechanism Analysis: How PFs Intervene and Why They Work

PFs intervene through two main mechanisms: directly revising the next action and injecting corrective context into the reasoning process. Across all fired PF events on web-search reasoning, 65.1% are action-level interventions and 34.9% are context-level interventions, showing that HASP frequently changes the next executable decision itself rather than only adding textual guidance.

Intervention patterns are concentrated in a few high-value skills. Trigger statistics over all PF activations in multi-step ReAct rollouts show that PF use is highly concentrated. The most frequently activated skills are `decompose_complex_question`, `insufficient_exploration`, and `answer_completeness`, and MuSiQue accounts for far more activations than HotpotQA or 2Wiki, indicating that harder multi-hop settings rely more heavily on PF intervention. The intervention patterns are also highly regular: most action-level PFs delay premature `FINAL` decisions by redirecting them into additional `SEARCH` steps, while most context-level PFs provide decomposition or completeness guidance.

Structured signals and strict filtering are both necessary. Table 3 shows that PF effectiveness depends on more than final-answer reward alone. Full HASP-Intervention reaches 56.2%, while removing timing, mode, correctness, and outcome causes drops of 7.8, 15.5, 12.1, and 12.8 points. The same table also shows that strict filtering is essential for stable skill-library evolution: full filtering reaches 60.3%, whereas unfiltered evolution drops sharply to 36.3%; using only executable filtering or only teacher-side filtering yields intermediate results of

48.8% and 47.2%.

Case-level evidence and failure structure support the PF design. Figure 3 shows how PFs recover a failure-prone MuSiQue trajectory by rewriting an over-constrained query and injecting a decomposition hint. Table 5 shows that recovered cases concentrate in a small number of recurring failure types, and that the most useful evolved PFs align closely with these high-frequency recoverable patterns, whereas overly specific or weakly reusable PFs contribute little or can even be harmful. We also study skills that cannot be converted into executable PFs and are therefore added to the skill library only as textual guidance. Their gains are limited and mixed, indicating that simply enlarging the library is not enough; the main benefit comes from skills that can be executed as PFs. Details are reported in Appendix A.5.

Skill internalization is strongest for behavior-correcting PFs. PFs that correct the model's own behavior, such as not reading enough or writing overly long queries, often become less necessary after training, whereas PFs tied to the question itself usually remain active. Quantitatively, `multi_hop_reasoning_failure` and `retrieval_failure` become silent on 100% of their previously triggered cases, while `decompose_complex_question` is rarely internalized. More broadly, only 8–38% of the accuracy gain can be directly attributed to cases where those PFs no longer need to fire; the larger effect comes from broader behavior changes, including more SEARCH and READ actions. This suggests that post-training mainly internalizes behavior-correcting PFs and a stronger overall reasoning style, while input-dependent PFs continue to provide useful online guidance.

4 Related Work

Post-training has been widely used to improve search, reasoning, tool use, and coding agents. Search-oriented methods such as Search-R1 (Jin et al., 2025), ReSearch (Chen et al., 2025), and VerlTool (Jiang et al., 2025) train models to interact more effectively with search or tool environments, while RL-based reasoning and coding methods further improve policy quality through reward-driven objectives (Zeng et al., 2025b; Hu et al., 2025; Ma et al., 2025; Li et al., 2025a; Fan et al., 2025). More broadly, LLM agents solve complex tasks by interleaving reasoning and acting (Yao et al., 2023), using external tools (Schick et al., 2023), and extending these loops through richer agentic workflows (Wu et al., 2024; Li et al., 2025b). Another prominent direction improves agent behavior by reusing past experience or skills. Methods such as Reflexion (Shinn et al., 2023), ExpeL (Zhao et al., 2024), and Voyager (Wang et al., 2023) store verbal lessons or memories, while more recent self-improving systems such as MemSkill (Zhang et al., 2026), SkillRL (Xia et al., 2026), and SAGE (Wang et al., 2026) study memory or skill evolution. Most prior methods reuse experience as prompt text, memory, or training signals. In contrast, HASP represents skills as executable state-action intervention functions that can be triggered inside the agent loop, while also supporting post-training and controlled skill evolution. Table 6 compares HASP with representative prior work.

Method	Skill / Memory Form	Runtime Control	Learning Signal	Policy Training	Skill Evolution
Skill-based / self-improving					
ExpeL (Zhao et al., 2024)	✓	○	○	✗	✗
MemSkill (Zhang et al., 2026)	✓	○	○	✗	✓
SKILL0 (Lu et al., 2026)	✓	✗	✓	✓	✗
SkillRL (Xia et al., 2026)	✓	○	✓	✓	✓
SAGE (Wang et al., 2026)	✓	○	✓	✓	✓
General RL-based					
AgentFlow (Li et al., 2025b)	○	○	✓	✓	○
Search-R1/ReSearch (Jin et al., 2025; Chen et al., 2025)	✗	○	✓	✓	✗
P-GRPO (Fan et al., 2025)	✗	✗	✓	✓	✗
HASP (Ours)	✓	✓	✓	✓	✓

Table 6: Qualitative comparison of HASP with representative prior methods. ✓: explicit support; ○: partial or indirect support; ✗: not directly supported.

5 Conclusion

We presented **HASP**, a framework that represents skills as reusable state-action intervention functions. Instantiated as Program Functions (PFs), skills become executable modular that can directly intervene on the agent's next decision, provide supervision for post-training, and grow through validated updates to an external skill library. This shared interface allows PFs to *act* during inference, *teach* during post-training, and *evolve* from recurring residual failures. Across web-search reasoning, mathematical reasoning, and coding tasks, HASP improves inference-time behavior, supports selective internalization without full reinforcement learning, and benefits from filtered skill evolution. Overall, our results suggest a complementary path for improving agents: instead of relying only on larger-scale policy optimization, skills can be made executable, explicitly triggered inside the agent loop, and gradually absorbed into both the policy and the external skill library.

Limitations

HASP has several limitations. First, while the core PF-only setting does not require a teacher, some stronger variants use external teachers for PF selection, teacher review, or on-policy distillation, which increases cost and may introduce teacher-specific bias. Second, PFs mainly help the model use and absorb existing strategies rather than discover fundamentally new ones; as suggested by harder benchmarks such as AIME24, tasks requiring new strategy discovery may benefit more from RL-style optimization. Third, our evaluation focuses on benchmark-style search, math, and coding tasks with relatively clear verification signals, leaving transfer to open-ended, long-horizon, interactive, or weakly verifiable environments unresolved. Fourth, the current PF interface supports only a limited set of intervention forms, mainly action modification and context injection. This improves executability and auditability, but also limits how much PFs can intervene. Finally, self-improving evolution may still degrade over time if generated PFs become too narrow, repetitive, or teacher-biased, and the current PF scoring interface may need to be adapted in domains where local repairs do not reliably translate into better final outcomes. Compute footprint and broader-impact analysis are provided in Appendix I.

References

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and 1 others. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*.

Mert Cemri, Melissa Z Pan, Shuyi Yang, Lakshya A Agrawal, Bhavya Chopra, Rishabh Tiwari, Kurt Keutzer, Aditya Parameswaran, Dan Klein, Kannan Ramchandran, Matei Zaharia, Joseph E. Gonzalez, and Ion Stoica. 2026. Why do multi-agent LLM systems fail? In *The Thirty-ninth Annual Conference on Neural Information Processing Systems Datasets and Benchmarks Track*.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, and 39 others. 2021. Evaluating large language models trained on code. *Preprint*, arXiv:2107.03374.

Mingyang Chen, Tianpeng Li, Haoze Sun, Yijie Zhou, Chenzheng Zhu, Haofen Wang, Jeff Z Pan, Wen Zhang, Huajun Chen, Fan Yang, and 1 others. 2025. ReSearch: Learning to reason with search for llms via reinforcement learning. *arXiv preprint arXiv:2503.19470*.

Lishui Fan, Yu Zhang, Mouxiang Chen, and Zhongxin Liu. 2025. Posterior-grpo: Rewarding reasoning processes in code generation. *Preprint*, arXiv:2508.05170.

Xanh Ho, Anh-Khoa Duong Nguyen, Saku Sugawara, and Akiko Aizawa. 2020. Constructing a multi-hop QA dataset for comprehensive evaluation of reasoning steps. In *Proceedings of the 28th International Conference on Computational Linguistics*, pages 6609–6625, Barcelona, Spain (Online). International Committee on Computational Linguistics.

Jingcheng Hu, Yinmin Zhang, Qi Han, Daxin Jiang, Xiangyu Zhang, and Heung-Yeung Shum. 2025. Open-reasoner-zero: An open source approach to scaling up reinforcement learning on the base model. *arXiv preprint arXiv:2503.24290*.

Aaron Hurst, Adam Lerer, Adam P Goucher, Adam Perelman, Aditya Ramesh, Aidan Clark, AJ Ostrow, Akila Welihinda, Alan Hayes, Alec Radford, and 1 others. 2024. GPT-4o system card. *arXiv preprint arXiv:2410.21276*.

Dongfu Jiang, Yi Lu, Zhuofeng Li, Zhiheng Lyu, Ping Nie, Haozhe Wang, Alex Su, Hui Chen, Kai Zou, Chao Du, and 1 others. 2025. VerlTool: Towards holistic agentic reinforcement learning with tool use. *arXiv preprint arXiv:2509.01055*.

Bowen Jin, Hansi Zeng, Zhenrui Yue, Jinsung Yoon, Sercan Arik, Dong Wang, Hamed Zamani, and Jiawei Han. 2025. Search-R1: Training llms to reason and leverage search engines with reinforcement learning. *arXiv preprint arXiv:2503.09516*.

Xuefeng Li, Haoyang Zou, and Pengfei Liu. 2025a. ToRL: Scaling tool-integrated rl. *arXiv preprint arXiv:2503.23383*.

Zhuofeng Li, Haoxiang Zhang, Seungju Han, Sheng Liu, Jianwen Xie, Yu Zhang, Yejin Choi, James Zou, and Pan Lu. 2025b. In-the-flow agentic system optimization for effective planning and tool use. *Preprint*, arXiv:2510.05592.

Nathan Lile. 2024. Math twenty four (24s game) dataset. https://huggingface.co/datasets/nlile/24-game.

Zhengxi Lu, Zhiyuan Yao, Jinyang Wu, Chengcheng Han, Qi Gu, Xunliang Cai, Weiming Lu, Jun Xiao, Yueting Zhuang, and Yongliang Shen. 2026. Skill0: In-context agentic reinforcement learning for skill internalization. *Preprint*, arXiv:2604.02268.

Xueguang Ma, Qian Liu, Dongfu Jiang, Ge Zhang, Zejun Ma, and Wenhu Chen. 2025. General-reasoner: Advancing llm reasoning across all domains. *arXiv preprint arXiv:2505.14652*.

MAA. 2023. American mathematics competitions. In *American Mathematics Competitions*.

Qwen, :, An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, and 25 others. 2025. Qwen2.5 technical report. *Preprint*, arXiv:2412.15115.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: Language models can teach themselves to use tools. *Preprint*, arXiv:2302.04761.

Zhihong Shao, Yeyun Gong, Yelong Shen, Minlie Huang, Nan Duan, and Weizhu Chen. 2023. Enhancing retrieval-augmented large language models with iterative retrieval-generation synergy. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 9248–9274.

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

Hao Sun, Zile Qiao, Jiayan Guo, Xuanbo Fan, Yingyan Hou, Yong Jiang, Pengjun Xie, Yan Zhang, Fei Huang, and Jingren Zhou. 2025. Zerosearch: Incentivize the search capability of llms without searching. *arXiv preprint arXiv:2505.04588*.

Harsh Trivedi, Niranjan Balasubramanian, Tushar Khot, and Ashish Sabharwal. 2022. Musique: Multi-hop questions via single-hop question composition. *Preprint*, arXiv:2108.00573.

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

Jiongxiao Wang, Qiaojing Yan, Yawei Wang, Yijun Tian, Soumya Smruti Mishra, Zhichao Xu, Megha Gandhi, Panpan Xu, and Lin Lee Cheong. 2026. Reinforcement learning for self-improving agent with skill library. *Preprint*, arXiv:2512.17102.

Ziliang Wang, Xuhui Zheng, Kang An, Cijun Ouyang, Jialu Cai, Yuhang Wang, and Yichao Wu. 2025. Stepsearch: Igniting llms search ability via stepwise proximal policy optimization. *arXiv preprint arXiv:2505.15107*.

Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, and 1 others. 2024. Autogen: Enabling next-gen llm applications via multi-agent conversations. In *First Conference on Language Modeling (COLM)*.

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. 2026. Skillrl: Evolving agents via recursive skill-augmented reinforcement learning. *Preprint*, arXiv:2602.08234.

Zhangchen Xu, Yang Liu, Yueqin Yin, Mingyuan Zhou, and Radha Poovendran. 2025. Kodcode: A diverse, challenging, and verifiable synthetic dataset for coding. *Preprint*, arXiv:2503.02951.

Zhenghai Xue, Longtao Zheng, Qian Liu, Yingru Li, Xiaosen Zheng, Zejun Ma, and Bo An. 2025. Simpletir: End-to-end reinforcement learning for multi-turn tool-integrated reasoning. *arXiv preprint arXiv:2509.02479*.

Jianhao Yan, Yafu Li, Zican Hu, Zhi Wang, Ganqu Cui, Xiaoye Qu, Yu Cheng, and Yue Zhang. 2025. Learning to reason under off-policy guidance. *arXiv preprint arXiv:2504.14945*.

Zhilin Yang, Peng Qi, Saizheng Zhang, Yoshua Bengio, William W. Cohen, Ruslan Salakhutdinov, and Christopher D. Manning. 2018. Hotpotqa: A dataset for diverse, explainable multi-hop question answering. *Preprint*, arXiv:1809.09600.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. React: Synergizing reasoning and acting in language models. *Preprint*, arXiv:2210.03629.

Huaye Zeng, Dongfu Jiang, Haozhe Wang, Ping Nie, Xiaotong Chen, and Wenhu Chen. 2025a. Acecoder: Acing coder rl via automated test-case synthesis. *Preprint*, arXiv:2502.01718.

Weihao Zeng, Yuzhen Huang, Qian Liu, Wei Liu, Keqing He, Zejun Ma, and Junxian He. 2025b. Simplerl-zoo: Investigating and taming zero reinforcement learning for open base models in the wild. *arXiv preprint arXiv:2503.18892*.

Haozhen Zhang, Quanyu Long, Jianzhu Bao, Tao Feng, Weizhi Zhang, Haodong Yue, and Wenya Wang. 2026. Memskill: Learning and evolving memory skills for self-evolving agents. *Preprint*, arXiv:2602.02474.

Yifan Zhang and Team Math-AI. 2024. American invitational mathematics examination (aime) 2024.

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

Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, and 1 others. 2024. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions. *arXiv preprint arXiv:2406.15877*.

A Additional Details of HASP Framework

A.1 PF Interface and Intervention Types

Each Program Function (PF) is a typed Python object exposing two methods:

- `should_activate(step_context, action_type, arg)` → `bool`
- `intervene(step_context, action_type, arg, teacher=None)` → `Intervention`

`should_activate` decides whether the skill is relevant at the current decision point; `intervene` returns a structured `Intervention` object whose `type` field is one of:

- `MODIFY_ACTION`: rewrite or refine the next action (action type and/or argument);
- `INJECT_CONTEXT`: append auxiliary text that the policy will see in the next observation;
- `NOOP`: abstain from changing the current decision (still emits an audit record).

PFs are by default deterministic and run on every step. A PF that sets `needs_teacher = True` additionally receives a teacher-model handle in `intervene`; such PFs must still degrade gracefully when `teacher` is `None`. The runtime enforces a per-PF rate limit (`_MAX_MODIFY_FIRES = 2` for `MODIFY_ACTION`) and per-skill caps (e.g. `RetrievalFailurePF` fires at most three times per episode) to prevent oscillation. The same dispatcher serves the web-search, math, and code domains without modification, so cross-domain transfer is mechanical.

A.2 Phase Instructions and Handlers

Beyond step-level activation, each PF may expose two extra surface areas. *Phase instructions*, declared in the YAML frontmatter of `SKILL.md`, attach prompt-level reminders to specific reasoning phases (`pre_final`, `post_search`, `post_read`); the runtime renders them only when the phase conditions match. *Handlers*, registered in `skill_handlers.py`, are FINAL-time verifiers that can flip a FINAL action back to SEARCH (capped at one such override per episode). The handler family is enabled only in web-search; math and code use single-step rollouts where there is no follow-up step for an override to land on.

A.3 An Example PF: Retrieval Failure

Skill A.1 shows the full `SKILL.md` for the `retrieval_failure` skill, and PF A.1 shows the corresponding PF class. The skill is one of three always-on web-search PFs, alongside `format_extraction_error` and `relevant_content_extractor`, because the rest of the library depends on the agent producing a well-formed search action and a clean observation.

Skill A.1: Adaptive Search Recovery (`retrieval_failure`)

```
---
skill_id: retrieval_failure
name: Adaptive Search Recovery
version: 2
priority: 0.8
error_category: retrieval_failure
applicable_modes: [all]
applicable_phases: [think, search]
system_summary: >
  When searches fail, reformulate queries with
     synonyms or
  decompose into sub-queries.
phases:
  post_search:
    conditions: [search_empty]
    priority_boost: 0.3
    action: reformulate_search
  pre_final:
    conditions: [no_read_yet]
    priority_boost: 0.2
    action: force_read_best_doc
---

# Adaptive Search Recovery

When initial searches fail, adapt by
     reformulating queries, using
synonyms, or decomposing complex questions into
     sub-queries.

## Detection Triggers
- Search returns no results or only irrelevant
     results
- Search query is too vague or too specific
- Multi-hop question requires intermediate
     entities not yet found

## Phase: post_search
No useful results. Try: (1) synonyms, (2)
     decompose multi-hop,
(3) broaden query. Do NOT proceed to FINAL
     without a relevant document.

## Phase: pre_final
STOP: You have not READ any document yet. Go
     back and READ at least
one relevant source before answering.
```

PF A.1: `RetrievalFailurePF`

```python
@register_pf("retrieval_failure")
class RetrievalFailurePF(ProgramFunction):
    needs_teacher = True
    _MAX_FIRES = 3
```

```
    def should_activate(self, ctx, action_type,
     arg):
        if action_type != "SEARCH":
            return False
        if len(arg.split()) <= 15:
            return False
        return
     ctx.get("_retrieval_failure_fires", 0) <
     self._MAX_FIRES

    def intervene(self, ctx, action_type, arg,
     model=None):
        ctx["_retrieval_failure_fires"] =
     ctx.get("_retrieval_failure_fires", 0) + 1
        if model is not None:
            clean = model.generate(
                messages=[{"role": "user",
     "content": (
                    f"A search agent is trying
     to answer this question:\n"
                    f"Question:
     {ctx.get('question','')}\n\n"
                    f"The agent generated this
     overly long search query:\n"
                    f"\"{arg[:500]}\"\n\n"
                    f"Extract a concise,
     effective search query "
                    f"(5-12 words) that
     captures the key search intent. "
                    f"Reply with ONLY the
     query.")}],
                max_tokens=50,
     temperature=0.0).strip().strip('"')
            if 1 < len(clean.split()) <= 20:
                return Intervention(

     type=InterventionType.MODIFY_ACTION,
                    new_action_type="SEARCH",
                    new_action_arg=clean,
                    reason="Teacher
     reformulated long query")
        # Deterministic fallback: extract
     quoted span / first short line
        return Intervention(
            type=InterventionType.MODIFY_ACTION,
            new_action_type="SEARCH",

     new_action_arg=self._smart_shorten(arg,
     ctx.get("question","")),
            reason="Smart-shortened query")
```

A math-domain example, `verification_missing`, is shown in Skill A.2; it is the highest-priority skill in the math library (priority 0.90, always active on `pre_final`).

Skill A.2: Solution Verification (`verification_missing`, math)

```
---
skill_id: verification_missing
name: Solution Verification
version: 1
priority: 0.90
error_category: verification_missing
applicable_modes: [all]
applicable_phases: [answer]
system_summary: >
  Before finalizing, plug the candidate answer
     back into the
  original problem to verify it satisfies all
     constraints.
phases:
  pre_final:
    conditions: []
    priority_boost: 0.5
    action: verify_solution
---

# Solution Verification

Substitute the candidate answer back into the
     original equation /
problem statement and confirm every constraint
     is satisfied.

## Phase: pre_final
Substitute your candidate answer back into each
     constraint of the
original problem. Does it satisfy every one? If
     not, recompute.

## Examples
**Scenario:** Solve sqrt(x+6) = x. Candidate: x
     in {-2, 3}.
**Wrong:** FINAL({-2, 3}) without checking.
**Correct:** Check x=-2: sqrt(4)=2 != -2
     (extraneous).
              Check x=3:  sqrt(9)=3  (valid).
     FINAL(3).
```

A.4 Skill Libraries Across Domains

The framework instantiates three skill libraries that share the same loader, registry, and intervention dispatcher; only the markdown content and `should_activate` predicates differ across domains.

Web-search library. Sixteen "PF skills" carry a deterministic activation rule plus a registered handler (eight LLM-assisted on FINAL, five pure-code, three legacy with inline handlers); eight "prompt-only" skills steer behavior through phase-instruction text without a handler. Four PFs use teacher assistance during the rollout: `retrieval_failure`, `format_extraction_error`, `reasoning_error`, and `answer_confidence_guard`. Three skills (`format_extraction_error`, `retrieval_failure`, `relevant_content_extractor`) are always included after PF selection because they normalize the action and observation streams that other skills consume.

Math library. All skills are PF-style, organized into five groups: algebraic/arithmetic correctness (`algebraic_sign_error`, `arithmetic_slip`, `simplification_incomplete`), domain/constraint (`boundary_violation`, `units_dimension_mismatch`), reasoning coverage (`case_incompleteness`, `overgeneralization`, `proof_step_gap`), substitution/verification (`substitution_invalid`, `verification_missing`), and output format (`final_format_error`).

Code library. The code library targets

single-shot code generation. Each PF runs a regex-based static analysis on the emitted code body and returns either an `INJECT_CONTEXT` hint or an audit `NOOP` (auto-patching student code is judged too risky). Skills cover complexity (`code_complexity_analysis`, `code_data_structure_choice`), bounds (`code_off_by_one`, `code_string_index_safety`, `code_edge_cases`), recursion/numerics, I/O, and verification. Behavioral steering comes mainly from injecting the corresponding `SKILL.md` text into the system prompt.

A.5 Prompt-Equivalent Skills

A registered PF is *prompt-equivalent* (criterion **A+B**) when (**A**) its `intervene` method emits only `INJECT_CONTEXT` (no `MODIFY_ACTION`), and (**B**) the injected `context_text` is a fixed literal string with no runtime substitution from `step_context`. Such a PF, when it fires, does nothing beyond pasting a constant reminder into the next observation, which is exactly what a phase-instruction prompt declared under `phases.{post_search,post_read,pre_final}.prompt` in `SKILL.md` would do. Their PF wrappers exist purely for interface uniformity (audit logging, signal scoring, unified dispatch). By contrast, PFs not classified as A+B either rewrite the action stream (`MODIFY_ACTION`, e.g. `retrieval_failure`, `insufficient_exploration`) or build the injected text dynamically from runtime state (e.g. `temporal_confusion` formats unsupported years into the warning, `citation_mismatch` formats the unsupported entity into the warning); both add capability that no prompt-only declaration can match. Note that A+B does not constrain how the PF *decides* to trigger.

Across the three domains, 10 of 26 web-search skills and 2 of 12 code skills satisfy A+B; no math skill satisfies A+B because all math PFs hook a `verify_X` action at `pre_final` rather than emitting a fixed `INJECT_CONTEXT`. Table 8 lists the 10 web-search prompt-equivalent skills and Table 9 lists the 2 code ones. Table 7 reports the marginal effect of adding these prompt-equivalent skills on top of the PF library, which is small and mixed across datasets and supports the claim that HASP's gains come from the non-A+B PFs.

A.6 Multi-Layer Skill Selection at Inference

The runtime never injects every library skill. Five concentric filters decide whether any given PF reaches the prompt or fires on an action.

(1) Master switches. Boolean flags including `skills_enabled`, `enable_program_functions`, `enable_skill_handlers`, `enable_prompt_only_skills`, `pf_only_mode`, and `enable_difficulty_gating` can short-circuit entire surfaces. The *difficulty gate* is the only switch that flips per-question: it asks the teacher for a 1–5 difficulty score, and bypasses skills below the threshold (heuristic fallback uses question length, multi-hop markers, and constraint phrases).

(2) Selection. Four cooperating selectors run in sequence. `SkillSelector.select()` ranks the library by mode_match $\cdot$ 0.5 + trigger_keyword_match $\cdot$ 0.5 at system-prompt build time. `PFSelector.select()` runs once per question and either delegates to the teacher (top-K by relevance, with three mandatory PFs always included) or applies a heuristic branching rule (computation $\rightarrow$ 3 mandatory; simple factoid $\rightarrow$ 4 light; multi-hop $\rightarrow$ safe + decomposition + entity-aware). Per-observation and per-step selectors then trim phase-gated reminders and handle high-priority overrides.

(3) Runtime trigger control. Even after a skill is selected, several conditions must align: the PF's own `should_activate` predicate, a JSON-condition evaluator over the phase-instruction conditions, per-PF rate limits, and a vote-based aggregator that counts how many handlers returned non-`None` on the same FINAL. When the count exceeds `handler_vote_threshold` (default 4 for web, 3 for math), the FINAL is overridden to a SEARCH; `_MAX_FINAL_OVERRIDES = 1` prevents a FINAL→SEARCH→FINAL→SEARCH loop.

(4) Budgets. Three classes of budget caps coexist: episode-level (`max_steps`, `max_search_calls`, `max_read_calls`, `timeout_seconds`), prompt-level (`max_skills_in_prompt`, `max_phase_instructions`, `pre_final_step_threshold`), and execution-level (`pf_top_k`, per-PF rate limits, `_MAX_FINAL_OVERRIDES`).

Table 7: Marginal effect of adding prompt-equivalent skills on top of the PF library, measured as Δ answer-MBE between the PF library augmented with prompt-equivalent skills and the PF library alone (`skills_top10_with_prompt` $-$ `skills_top10`). The effect is small and mixed in sign across the six datasets, indicating that prompt reminders alone do not consistently account for HASP's gains.

	HotpotQA	2Wiki	MuSiQue	AIME24	AMC23	GameOf24
Δ MBE	$+3.5$	-3.5	-1.0	0.0	$+2.5$	0.0

Table 8: Web-search PFs satisfying criterion A+B (prompt-equivalent), 10 of 26 skills. Each row gives the runtime activation trigger and the literal text the PF injects when it fires (the bracketed tag prefix is part of the verbatim string; long suffixes are abbreviated with "...").

Skill ID	**Trigger**	**Injected text (verbatim, abbreviated)**
`decompose_complex_question`	multi-hop pattern: ≥ 2 possessives, ≥ 2 "of the" chains, or relative-pronoun chain	"[DECOMPOSITION HINT] This is a multi-hop question. Search for each piece of information separately..."
`adversarial_distraction`	SEARCH result set with ≥ 3 conflict words	"[Note: Conflicting Sources] Search results contain contradictory claims. Cross-reference before answering."
`negation_oversight`	question contains "not / never / except" that the reasoning ignores	"[NEGATION REMINDER] The question contains a negation... Re-read the question carefully."
`wrong_entity_confusion`	SEARCH returns multiple similarly-named entities	"[ENTITY WARNING] Search results contain similarly-named entities. READ documents carefully..."
`reading_comprehension_error`	READ of dense content (≥ 10 numbers or ≥ 15 names)	"[READ CAREFULLY] This document contains many entities/numbers. Extract the specific information..."
`answer_completeness`	FINAL with single-token answer to multi-part question	"[COMPLETENESS WARNING] The question appears to have multiple parts, but your answer may be incomplete..."
`language_barrier`	READ doc with ≥ 50 non-ASCII chars (or FINAL with sources ≥ 5 non-ASCII)	"[LANGUAGE NOTE] This document contains non-English text. Be careful with transliteration..."
`iterative_refinement`	SEARCH with token-Jaccard > 0.5 vs. recent queries	"[SEARCH REFINEMENT] Your recent searches are very similar. Try a different approach..."
`claim_triangulation`	FINAL based on single read source with specific factual claims	"[VERIFICATION HINT] Your answer is based on a single source. Consider reading one more document..."
`misinformation_detector`	≥ 2 read docs with conflicting years or contradiction phrases	"[CROSS-CHECK WARNING] Your read documents may contain conflicting information..."

A.7 Skill Retrieval and PF Selection Prompt

When the framework retrieves PFs at the start of an episode, the teacher is asked to rank candidate skills against the current question. The selector prompt explicitly forbids over-selection ("fewer well-targeted PFs is better than many irrelevant ones") and pins three mandatory PFs that the teacher cannot drop. This second stage of filtering complements the lexical scoring in `SkillSelector` and reduces the chance that broad or off-topic skills affect the trajectory.

B Self-Improving Pipeline Details

The self-improving stage is the offline counterpart to inference: where inference *runs* the agent against a fixed library, self-improvement *grows* that library. Each epoch revisits residual failures from the current checkpoint and runs an eight-phase pipeline (Phases A–H). The phases produce per-epoch artifacts under `output_dir/epoch_{i}/`: trajectories, failure analyses, candidate skills, validation reports, teacher reviews, gradient streams, and training-data files. Algorithm 1 summarizes one epoch.

Algorithm 1: One self-improving epoch: from residual failures to library update and PF-derived training data.

Input: Student policy π_θ; teacher $\mathcal{T}$; current library $\mathcal{M}_r$; seed split $\mathcal{D}^{\text{seed}}$; validation split $\mathcal{D}^{\text{val}}$; budgets $K_{\text{cand}}, \eta_{\text{exec}}, \eta_{\text{teach}}$.

Output: Updated library $\mathcal{M}_{r+1}$; training datasets $\mathcal{D}^{\text{SFT}}, \mathcal{D}^{\text{DPO}}$.

```
// Phase A: PF-aware rollout on the seed split
Traj ← ∅;
for x ∈ D^seed do
    τ_x ← Rollout(π_θ, x, M_r, T);
    Traj ← Traj ∪ {τ_x};

// Phase B: heuristic + LLM failure analysis
F ← {τ ∈ Traj : EM(τ) = 0};
P ← HeuristicDetect(F) ∪ LLMSummarize(F, T);
C ← Cluster(P);              // Jaccard ≥ 0.5 dedup; drop buckets < 3

// Phase C: candidate proposal (rate-limited to K_cand)
S_cand ← ∅;
for c ∈ TopByFreq(C, K_cand) do
    (md, code) ← ProposePF(π_θ, c, M_r);
    S_cand ← S_cand ∪ {(md, code)};

// Phases D & E: executable validation and teacher review
S_ok ← ∅;
for s ∈ S_cand do
    Q_exec ← ExecValidate(s);      // syntax / interface / mock-exec /
     return-type
    (Q_concept, Q_trig, Q_int, Q'_exec, Q_val) ← TeacherReview(T, s);
    Q_skill ← 0.25Q_concept + 0.20Q_trig + 0.20Q_int + 0.20Q'_exec + 0.15Q_val;
    if Q_exec ≥ η_exec and Q_skill ≥ η_teach then
        S_ok ← S_ok ∪ {s};

// Phase F: library update with versioning + size cap
M_{r+1} ← Admit(M_r, S_ok);

// Phases G & H: signal scoring and training-data construction
for τ ∈ Traj, step t ∈ τ do
    z_t ← ScoreSignals(τ, t);
     // (timing, modality, correctness, outcome)
    A_t ← 0.15z_t^T + 0.10z_t^M + 0.25z_t^C + 0.50z_t^O;
(D^SFT, D^DPO) ← BuildTrainingData(Traj, {A_t}, S_ok);
return M_{r+1}, D^SFT, D^DPO;
```

B.1 Per-Epoch Phase Sequence

Phase A executes the current student against the seed split with PFs active; an optional two-stage prefilter cheaply identifies failing samples first, then concentrates the expensive PF-aware rollout on those. Phase B clusters failures (§B.2). Phase C asks the student to author candidate skills against each large/novel cluster, capped at 5 candidates per epoch (§B.3). Phase D runs four executable checks on each candidate (§B.4). Phase E asks the teacher for a five-dimensional review (§B.5). Phase F admits accepted candidates to the library with a versioned `__v{N}` suffix (§B.6). Phase G assigns PF-mediated credit to produce student and teacher gradient streams. Phase H emits SFT/DPO training data over both objectives (action-correction and

Table 9: Code PFs satisfying criterion A+B (prompt-equivalent), 2 of 12 skills. Both have empty `intervene` bodies that emit only `NOOP` audit records; their behavioral effect comes entirely from the corresponding `SKILL.md` text being injected into the system prompt at episode start.

Skill ID	Trigger	`intervene` behavior
`code_negative_handling`	weak signal on "negative" / "integer" in question; no static check	`NOOP` audit only
`code_test_walkthrough`	always fires on `class Solution` or stdin code; no static check	`NOOP` audit only

skill-authoring).

The validation pool is built once at setup time from the *tail* of each test dataset (samples after the test boundary), shuffled with a fixed seed, and split into 50 seed and 50 val samples per dataset, yielding $\approx 250+250$ samples across the five active web datasets. Cached splits are reused across runs unless the seed budget is set to -1 (use all).

B.2 Phase B — Failure Signal Detection

Phase B uses two detectors in parallel. The *heuristic* detector applies twelve hand-coded rules against every failed trajectory, including `premature_final` (trajectory shorter than three steps), `repeated_search` (queries with token-Jaccard > 0.8), `no_read_before_final`, `query_too_broad`/`too_narrow`, `wrong_entity_focus` (answer/-gold token overlap $< 30\%$), `reasoning_hallucination` (final whose content tokens are $< 30\%$ grounded in read content), `format_mismatch`, `partial_answer`, `contradictory_evidence_ignored`, `excessive_steps_no_progress`, and `pf_override_harmful` (a PF rewrote the action but the episode still failed).

The *LLM* detector asks the teacher to abstract each failed trajectory into a reusable category (Prompt B.1); per-trajectory generations run with `temperature=0.3` and `max_tokens=400`, and concurrency is bounded by a thread pool. Heuristic and LLM patterns are deduplicated independently (token-Jaccard ≥ 0.5 within the same category) and then bucketed into six broad existing-skill groups for novelty scoring; clusters with novelty ≥ 0.3 are flagged `is_new_category` and prioritized for proposal. Buckets below `min_cluster_size` ($= 3$) are dropped.

Prompt B.1: Phase B — LLM Failure Summarizer

```
SYSTEM:
You are an expert failure analyst for a ReAct
     web-search agent.
For each failed QA trajectory you must produce
     a concise, reusable
abstraction of WHY the agent failed so that a
     new Program Function
(PF) skill can be authored to fix this class of
     failure.

USER (per trajectory):
Question:
{question[:500]}

Gold answer(s): {gold}
Agent's final answer: {pred[:200]}

Trajectory (truncated, one line per step):
{trace[:1800]}

Return STRICT JSON with keys:
  category: short snake_case category id (4-30
     chars, e.g.
    'constraint_leak', 'premature_summary',
     'stale_evidence').
    Reuse an existing id if the failure mode
     matches.
  abstraction: 1-sentence description of the
     recurring failure mode.
  trigger: when in a ReAct step this failure
     manifests.
  intervention_hint: what a PF should do to
     prevent this.
Output ONLY the JSON object.
```

B.3 Phase C — Skill Proposal

For each surviving cluster, the student model authors a candidate skill consisting of (i) a `SKILL.md` document with YAML frontmatter, trigger conditions, intervention design, and concrete examples; and (ii) a `ProgramFunction` subclass implementing `should_activate`/`intervene`. The proposer prompt (Prompt B.2) constrains the keyword arguments of `Intervention` and forbids LLM calls inside `should_activate`, so the validator's mock execution remains feasible. Generation runs at `temperature=0.7`, capped at five candidates per epoch.

Prompt B.2: Phase C — Skill Proposer (system, excerpt)

```
You are a skill designer for a ReAct web search
     agent.

The agent operates in a SEARCH -> READ -> FINAL
     loop. At each step,
Program Functions (PFs) can check conditions
     and intervene by
modifying the agent's action or injecting
     context.

Your job: given a cluster of recurring
     failures, propose a NEW skill
```

```
that will prevent this failure pattern in the
     future. The skill
consists of:
  1. A SKILL.md specification (YAML frontmatter
     + markdown body).
  2. A ProgramFunction Python class with
     should_activate() and
     intervene() methods.

IMPORTANT RULES:
- The PF must be deterministic (NO LLM calls in
     should_activate).
- should_activate receives: step_context (dict),
  action_type (str: "SEARCH"/"READ"/"FINAL"),
     arg (str).
- intervene receives the same + optional
     teacher model, and returns
  an Intervention.
- The PF class must inherit from
     ProgramFunction and use the
  @register_pf decorator. should_activate must
     return bool;
  intervene must return an Intervention object.

EXACT Intervention interface (use ONLY these
     keyword arguments):
    type: InterventionType  (NOOP |
     MODIFY_ACTION | INJECT_CONTEXT)
    new_action_type: Optional[str]   # "SEARCH"
     | "READ" | "FINAL"
    new_action_arg:  Optional[str]
    context_text:    str
    reason:          str
    skill_id:        str
```

B.4 Phase D — Executable Validation

Each candidate is wrapped in a sandbox preamble (mocked `InterventionType`, `Intervention`, `ProgramFunction`, and `register_pf`) and subjected to four sequential checks: *syntax* (`ast.parse`), *interface* (a class inheriting from `ProgramFunction` with the two correctly-signed methods), *mock execution* (3 mock step contexts $\times$ {SEARCH, READ, FINAL} = 9 invocations), and *return type* (`intervene` constructs an `Intervention`). The three mock contexts cover the early-step/no-read regime (step 1, no reads), the mid-step/has-read regime (step 5, two reads), and the late-step/empty-results-with-contradictions regime (step 12). Any runtime exception fails check three; `Q_exec < 0.3` on review is later treated as a hard reject.

B.5 Phase E — Teacher Five-Dimensional Review

The teacher receives the candidate as four blocks (header, trigger conditions, intervention description, full `SKILL.md` and PF code) plus the source failure cluster's metadata, and is asked to score five dimensions on $[0, 1]$: *concept*, *trigger*, *intervention*, *executability*, and *validation utility*. The composite is

$$Q_{\text{skill}} = 0.25\, Q_{\text{concept}} + 0.20\, Q_{\text{trigger}} + 0.20\, Q_{\text{intervene}} + 0.20\, Q_{\text{exec}} + 0.15\, Q_{\text{val}}.$$

The decision is taken in priority order: a hard reject if $Q_{\text{exec}} < 0.3$; otherwise the teacher's literal `DECISION` line if present; otherwise a threshold fallback ($Q_{\text{skill}} \geq 0.60 \Rightarrow$ accept; $\geq 0.42 \Rightarrow$ revise; else reject). Two further thresholds, `new_group_threshold` $= 0.75$ and `same_group_threshold` $= 0.60$, gate library admission for entirely new categories versus refinement of an existing one. The system and user prompts are reproduced in Prompts B.3–B.4.

Prompt B.3: Phase E — Reviewer (system)

```
You are an expert skill quality reviewer for a
     ReAct web search
agent system.

The agent uses Program Functions (PFs) --
     deterministic Python hooks
that run every step. Each PF has:
  - should_activate(step_context, action_type,
     arg) -> bool
  - intervene(step_context, action_type, arg,
     teacher=None)
      -> Intervention

Intervention types: MODIFY_ACTION (change
     action),
INJECT_CONTEXT (add text to observation), NOOP.

step_context fields: question, step_count,
     has_read, search_count,
read_count, empty_results,
     contradictory_sources, max_steps,
action_history, last_search_results_text,
     all_read_contents, thought.

You must evaluate candidate skills rigorously
     across 5 dimensions.
```

Prompt B.4: Phase E — Reviewer (user, response template)

```
## Review Instructions
Score each dimension from 0.0 to 1.0 and
     provide brief feedback.

### Dimension 1: Concept (Q_concept)
- Is this a real, recurrent failure type?
- Is it generalizable, not a single-case patch?
- Is it clearly distinct from existing skills?

### Dimension 2: Trigger (Q_trigger)
- Is should_activate() condition specific and
     deterministic?
- Does it depend on available step_context
     fields?
- Is it neither too broad (many false
     positives) nor too narrow?

### Dimension 3: Intervention (Q_intervene)
- Does intervene() appropriately address the
     failure?
- Could it cause harmful side effects?
- Is the intervention proportional (not
     over-blocking)?

### Dimension 4: Executability (Q_exec)
- Does the code follow the ProgramFunction
     interface?
- Would it import without errors? Are return
     types correct?

### Dimension 5: Validation Utility (Q_val)
- Will this likely reduce the target failure
     type?
```

```
- Will it transfer to unseen questions?
- Will it interact well with existing PFs?

Respond in EXACTLY this format:
Q_concept: [0.0-1.0]
Concept feedback: [brief]
... (Q_trigger, Q_intervene, Q_exec, Q_val
     similarly) ...
DECISION: [ACCEPT / REVISE / REJECT]
OVERALL FEEDBACK: [1-3 sentences]
```

B.6 Phase F — Library Update and Versioning

Accepted candidates are appended to `generated/{skill_id}__v{N}/SKILL.md`, and their PF code is appended to `dynamic_program_functions.py`. When a candidate matches an existing base id, accept rewrites it as the next version (`__v{N+1}`); the old version remains on disk for trajectory replay. The runtime loads only the highest version per base id (`skill_ids_latest`), so older versions stay auditable but never fire. `max_library_size = 50` caps total active skill count, and `library_history.json` records every per-skill decision and review score so closed-loop runs can reconstruct the gated decision tree without rerunning rollouts.

B.7 Failure Clustering and Candidate PF Proposal

Self-improvement begins from the residual failures of the current checkpoint. Failures are grouped into recurring failure-repair patterns using the heuristic pass followed by the LLM consolidation pass (§B.2); to avoid near-duplicate skills, candidate groups are deduplicated using similarity-based clustering before proposal. Proposal is rate-limited so the library grows slowly and remains controllable over many rounds.

B.8 Why Strict Filtering Matters

The central empirical observation behind our framework is that self-improvement is useful only when library evolution is tightly controlled. Adding more skills is not automatically beneficial. When validation and filtering are weak, the library accumulates noisy or redundant skills, retrieval precision drops, and policy quality deteriorates. When filtering is strict, each accepted PF carries clearer intervention semantics and higher reuse value, and the evolving library continues to improve the agent over time. We therefore treat self-improving PF evolution as a *validated* and *rate-limited* process rather than unconstrained skill generation.

C Additional Experimental Details

C.1 Datasets and Train/Test Splits

For web-search reasoning and mathematical reasoning, our test sets and training pools follow the protocol of AgentFlow (Li et al., 2025b): we adopt the same source datasets, the same per-dataset test sizes, and the same convention of reserving the first N samples per file as the held-out test set with the remainder forming the training pool. This makes the two domains directly comparable to AgentFlow and to the prior search/reasoning baselines listed in §C. For coding, AgentFlow does not provide a precedent, so we adopt the same training-data construction protocol and additionally carve out per-dataset test/training splits ourselves, as detailed below. Across all domains, training pools are used both for the bootstrap PF-aware rollouts of V1 and as the source of seed/validation samples for the closed-loop evolution loop of V2 (§B).

Web-search reasoning. Each source file contains 1,000 randomly sampled questions; the first 200 rows of each file constitute the test set and the remaining 800 rows constitute the training pool. The closed-loop validation pool described in §B is drawn from the *tail* of the training pool (samples after the 200-row test boundary), shuffled with seed 42 and split into 50 seed and 50 val samples per dataset. Datasets whose total size equals their test boundary are auto-skipped because no tail remains; in our default web-search configuration, this leaves five active datasets feeding the seed/val splits ($\approx$ 250 seed + 250 val).

Table 10: Web-search dataset sizes. Test-set selection follows AgentFlow (Li et al., 2025b).

Dataset	Total	Test	Train pool
HotpotQA (Yang et al., 2018)	1,000	200	800
2WikiMultihopQA (Ho et al., 2020)	1,000	200	800
MuSiQue (Trivedi et al., 2022)	1,000	200	800

Mathematical reasoning. The math benchmarks are small, so we use the entire dataset as the test set, exactly as in AgentFlow (Li et al., 2025b): AIME24 (30 problems), AMC23 (40 problems), and GameOf24 (1,362 problems). The training data follows AgentFlow's training-pool composition. The closed-loop validation pool of §B draws from the same training-pool tail under the math configuration (`data_subdir = data_math`), with the difficulty gate disabled because all math problems are uniformly hard.

Table 11: Mathematical reasoning dataset sizes. The full set for evaluation follows AgentFlow (Li et al., 2025b).

Dataset	Total	Test
AIME24 (Zhang and Math-AI, 2024)	30	full
AMC23 (MAA, 2023)	40	full
GameOf24 (Lile, 2024)	1,362	full

Coding. Coding is the only domain not covered by AgentFlow (Li et al., 2025b), so we constructed the train/test splits explicitly. The training-data construction protocol (PF-aware rollout, four-signal scoring, V1/V2 topology) is identical to the other two domains. For HumanEval+ and MBPP+, we use the EvalPlus extended versions and reserve the first 50 unique problems (100 entries after EvalPlus row expansion) as the test set, with the remaining unique problems forming the training pool. For BigCodeBench, we reserve 100 entries as the test set and use the remaining 1,040 entries as the training pool. Full per-dataset statistics are reported in Table 12.

Table 12: Coding dataset sizes. AgentFlow does not cover coding, so we constructed the splits ourselves following the same training-pool layout convention. "Test" is the held-out evaluation set used in Table 2; "Train pool" is the remainder available for PF-aware rollouts. EvalPlus-based datasets count rows (a single problem may expand into multiple test entries).

Dataset	Total	Test	Train pool
HumanEval+	328 (164 unique)	100 (50 unique)	228 (114 unique)
MBPP+	756 (378 unique)	100 (50 unique)	656 (328 unique)
BigCodeBench/Full	1,140	100	1,040
BigCodeBench/Hard	148	full	—

Self-improving seed/val split (closed-loop only). Both V2 closed-loop training (E4–E6) and the offline self-improving runs of §B draw their evolve-time seed and validation samples from the *tail* of the training pool described above, shuffled deterministically with seed 42. The default budget is 50 seed + 50 val per dataset, capped at half the available tail for any dataset whose tail is smaller than 100. Datasets with no tail (i.e. test set already covers the entire file) are auto-skipped at setup time. Setting the seed budget to -1 uses the entire tail, with no validation split; this is reserved for full-coverage failure summarization in late evolve cycles.

C.2 Detailed Definition of E1–E6

The six post-training settings form a 2×3 grid over loop topology and training method.

V1: Open-loop training. V1 trains on a fixed PF-aware rollout pool collected from a frozen skill library. The library snapshot is fixed throughout training and no additional skill evolution occurs after data collection.

V2: Closed-loop training. V2 periodically evolves the skill library and refreshes the training data accordingly. This creates a closed loop in which newly accepted PFs can affect both subsequent rollouts and subsequent training data.

Training methods. The three training methods are supervised fine-tuning (SFT), rejection sampling (RS), and on-policy distillation (OPD). Their Cartesian product with V1/V2 gives: E1: V1 open-loop SFT; E2: V1 open-loop RS; E3: V1 open-loop OPD; E4: V2 closed-loop SFT; E5: V2 closed-loop RS; E6: V2 closed-loop OPD. In E3 and E6, GPT-4o serves as the teacher.

C.3 Shared Training Configuration

Table 13 summarizes the core training configuration shared across E1–E6. Across all settings, we use LoRA with rank 16 and $\alpha = 32$, bf16 precision, and gradient checkpointing. Effective tokens per optimizer step are ≈ 65K, so each "epoch" is a small number of optimizer steps even with the full SFT pool, which is why SFT runs for 10 epochs and RS / closed-loop runs use ≥ 8 epochs per phase.

C.4 Rollout and Teacher Configuration

For rejection sampling and on-policy distillation, the rollout configuration is shared unless otherwise noted. We use group size 2, sampling temperature 0.9, top-p 0.95, maximum reasoning steps 5, and maximum search calls 3. In OPD, GPT-4o serves as the teacher with maximum response length 512 and concurrency 64. Every rollout-using experiment runs in *teacher-snippet mode*: the teacher returns a ∼200-token snippet per SEARCH and READ is disabled (`max_read_calls` $= 0$); this decouples training rollouts from SerpAPI quotas and READ-cost variance, at the cost of trusting the teacher to summarize. Because of teacher-side 429 noise, the RS filter accepts `min_score` $= 0.15$ on a composite $0.7 \cdot \text{EM} + 0.2 \cdot \text{F1} + 0.1 \cdot \text{StepEconomy}$ rather than `require_exact_match=true`.

Table 13: Training configuration for E1–E6. All models use Qwen2.5-7B-Instruct with LoRA, bf16 training, and the same maximum sequence length.

Setting	Loop	Method	LR	Epochs	Rollout setup	Teacher
E1	Open-loop	SFT	1×10^{-5}	10	–	–
E2	Open-loop	RS	5×10^{-6}	8 per iteration	$k = 2$, temp 0.9, top-p 0.95, max steps 5	–
E3	Open-loop	OPD	5×10^{-6}	10	same as E2	GPT-4o
E4	Closed-loop	SFT	1×10^{-5}	refreshed per phase	refreshed after evolution	–
E5	Closed-loop	RS	5×10^{-6}	8 per phase	same as E2	–
E6	Closed-loop	OPD	5×10^{-6}	5 per phase	same as E2	GPT-4o

C.5 Closed-Loop Evolution Modes

Closed-loop training supports two evolution modes.

Full evolution. The full mode runs the complete self-improving pipeline (Phases A–H). One evolve cycle takes approximately one hour on the canonical setup; only E4 uses it, because SFT can amortize the cost over the slower 8-epoch phase.

Lite evolution. The lite mode samples 20 failed trajectories from the rollout pool, asks the current student vLLM checkpoint for at most three skill proposals, applies `ast.parse` compile-checks and skill-id deduplication, and writes the result to `generated/` — with no teacher API calls and no clustering, review, or gradient computation. It is significantly cheaper ($\sim$5% of the wall-clock) and yields most of the library-growth benefit, so E5 and E6 use it under their faster three-epoch phases.

C.6 Data Construction Pipeline

Training data is built in four layers. *Layer 0* produces raw trajectories: V1 runs a one-shot PF-aware ReAct rollout over the training pool with skills, PFs, and teacher all enabled (effectively Phase A run once); V2 re-runs Phase A every closed-loop cycle on the current student checkpoint with the just-evolved library. *Layer 1* scores each step with the four-signal aggregator and filters by a 0.25 reward floor; both the fine 15-dim breakdown and the coarse 4-scalar view are persisted on every sample so an offline rescorer can re-weight without rerunning rollouts. *Layer 2* renders steps into one of two objectives: **Objective A** ("use PFs") emits SFT, DPO, and prompt-only files where the SFT target is the PF-corrected action, the DPO chosen/rejected pair is restricted to steps with `was_modified=True`, and `sample_weight` carries the per-step reward; **Objective B** ("evolve") emits SFT and DPO files for the skill-author task, with target equal to the candidate's `SKILL.md + PF code` and reward $r_{\text{skill}} = Q_{\text{skill}} + 0.5 \cdot \Delta\text{EM}_{\text{val}}$. *Layer 3* is the per-experiment YAML `data:` block: each E points at one of the Layer-2 jsonls, and V2 experiments add a refresh hook that rebuilds the file in-place between phases.

C.7 Post-Training Recipes

We study three post-training settings.

SFT. PF-mediated corrections are converted into weighted supervised targets. This is the simplest way to internalize repaired actions.

Rejection sampling. Trajectories are sampled, scored by both task success and PF-conditioned intervention quality, and only the top subset is retained. This selects trajectories that are both correct and behaviorally clean.

On-policy distillation. The current model generates its own trajectories, PFs intervene on difficult states, and the corrected behaviors are distilled back into the model. This aligns the training distribution with the model's own inference-time state distribution.

The three recipes are deliberately defined over the same signal interface so that recipe choice and signal ablation remain orthogonal experimental axes.

D Fine-Grained Signal Composition

The main text uses four coarse signals (timing, modality, correctness, outcome) with weights $(0.15, 0.10, 0.25, 0.50)$. Internally, each is computed as a weighted sum of finer sub-signals; the coarse formulation is used in training because it is stable and easy to ablate, while the fine decomposition is used for diagnostics and analysis.

Timing (S1). A step is defined to be *risky* on the *proposed* action (before any PF intervention) if it is a FINAL with no prior READ, a FINAL at `step_count` < 3, or a SEARCH after `empty_results`. The four sub-signals

are `s1.tp` ($+0.25$, risky and any PF activated), `s1.fp` (-0.10, not risky but a PF still activated), `s1.fn` (-0.10, risky but no PF activated), and `s1.phase` ($+0.05$, normalized step index, rewarding earlier interventions).

Modality (S2). `s2.pre_action` ($+0.35$, a `MODIFY_ACTION` fired before the action ran) and `s2.post_obs` ($+0.35$, an `INJECT_CONTEXT` fired after the observation) carry all current signal; `s2.pre_reasoning` and `s2.post_action` are reserved slots that return 0 until corresponding intervention types are added, kept registered with non-zero weights so future PFs accrue reward without changing config.

Correctness (S3). `s3.syntactic` ($+0.20$, action type valid and argument non-empty); `s3.semantic` ($+0.50$, teacher-judged step score with a heuristic fallback that gives 0.7 to a FINAL→READ rescue, 0.5 to other modifications, 0.3 otherwise); `s3.domain` ($+0.30$, plugin slot averaged over domain-specific quality functions registered under `s3.domain.<name>`), default 0.

Outcome (S4). `s4.local` ($+0.40$, scored by transition: 0.8 for FINAL→READ, 0.7 for FINAL→SEARCH, 0.5 for SEARCH→SEARCH rewrite, 0.3 for any other modification, else 0); `s4.downstream` ($+0.40$, episode EM); `s4.cost` (-0.10, $\text{clip}((T-15)/10, 0, 1)$); `s4.side_effect` ($-0.10$, modified action but episode still failed). S4 is the heaviest family in coarse mode because it is the only one anchored directly to ground-truth EM.

Aggregation. In *fine mode*, the per-step score is $\sum_{\text{sub}} w_{\text{sub}} \cdot \text{value}_{\text{sub}}$, optionally normalized by $\sum |w_{\text{sub}}|$. In *coarse mode*, each family is first reduced to a scalar via family-internal normalization (preserving the sign of penalty sub-signals), then combined with the four coarse weights. The episode-level reward is a $50/50$ blend of the per-step mean and the terminal EM, so a perfect-EM episode with poor PF behavior does not get full credit and a near-miss episode with strong intermediate signals is not completely discarded.

Skill-level reward. For training the skill author (Phase H), there is one additional skill-level reward, $r_{\text{skill}} = Q_{\text{skill}} + \lambda \cdot \Delta\text{EM}_{\text{val}}$ with default $\lambda = 0.5$, where $\Delta\text{EM}_{\text{val}}$ is the change in validation EM after the skill is added. This is the only signal that is not step-level.

E Additional Main Results

This section provides three complementary views of the main results: (i) a controlled stage-by-stage breakdown that isolates inference-time intervention from post-training internalization and from closed-loop evolution; (ii) a deeper read of the cross-domain transfer to coding; and (iii) the dynamics of the skill library across self-improvement rounds.

E.1 Stage-by-Stage Contribution

Table 14 is constructed by selectively enabling each stage of the framework while keeping all other knobs fixed at the defaults from §C. Adding inference-time PFs to the multi-loop baseline already accounts for the bulk of the gain on web-search reasoning, since PFs convert recurring decision-level failures into structured repairs without modifying the policy. Post-training then internalizes those repairs into the model's own state distribution and captures the additional gain that PF-conditioned supervision provides *after* the policy has been queried with the library. Closed-loop evolution layers on top, growing the library when residual failures expose patterns that the seed library does not cover. The composition is roughly additive but sub-linear: the stages are not independent, and evolution alone contributes little when PFs are not also active during inference.

Table 14: Stage-by-stage contribution of the three mechanisms in our framework on web-search reasoning average accuracy. Each row turns on one additional component while keeping all others at the defaults from §C.

Setting	Avg. Acc.
Base multi-loop agent (RA-Agent)	31.2
+ Inference-time PF intervention	56.2
+ Post-training internalization (E3, OPD)	62.5
+ Closed-loop evolution (E5, RS)	60.3
Full system (best per-recipe combination)	–

E.2 Detailed Coding Results

In Section 3, we report pass@1 across HumanEval(Base/Plus), MBPP(Base/Plus), and BigCodeBench(Full/Hard). The coding setting is the strictest cross-domain transfer test for our framework: the action space collapses to a single FINAL emission, READ is unavailable, and the only PF surface is regex-based static analysis on the emitted code body that returns either an

INJECT_CONTEXT hint or a NOOP audit record (§A.4). Despite this constrained interface, two patterns are visible in the table.

Inference-time PFs help most on edge-case-heavy benchmarks. Static checks for missing length guards, off-by-one indexing, and stdin parsing are concentrated in the harder splits (BigCodeBench/Hard, MBPP/-Plus), so the inference-time PF rows tend to gain more there than on HumanEval/Base, where the Qwen2.5-7B-Instruct backbone already scores 81.7%. The phase-instruction surface (+code_test_walkthrough, +code_edge_cases) is the dominant lever in this domain — the PF code itself is auditing rather than rewriting.

Internalization complements RL-style coders. GRPO (Fan et al., 2025) and KodCode-RL (Xu et al., 2025) are stronger overall on HumanEval and MBPP because they expand the policy frontier through reward-shaped exploration. Our PF-derived supervision, in contrast, primarily targets capability elicitation: by training on PF-corrected steps, it lifts the rate at which existing strategies are reliably executed without performing exploration-style reward search. The two directions are largely orthogonal, and our coding gains are concentrated on edge-case correctness rather than on raw algorithmic difficulty.

E.3 Per-Dataset Robustness

The web-search and math averages reported in the main text aggregate over heterogeneous benchmarks, and the per-dataset numbers in Section 3 make a few qualitative points worth recording. On web-search, PFs help most on MuSiQue, where multi-hop chains create many opportunities for entity confusion and premature FINAL — exactly the failure modes that retrieval_failure, wrong_entity_confusion, and insufficient_exploration target. 2Wiki sits in the middle: AgentFlow remains stronger there, plausibly because its RL-style policy search compensates for the relatively shorter retrieval chains where PF-style local repair has less leverage. On math, PFs are most effective on AMC23 and GameOf24, where small algebraic and arithmetic slips dominate failure cases (addressed by algebraic_sign_error, arithmetic_slip, verification_missing). AIME24 remains the hardest: failures there are not local slips but missed strategy choices, which is why broader RL-style exploration outperforms our elicitation-oriented framework on that dataset.

E.4 Evolution Dynamics Across Rounds

We track three quantities across self-improvement rounds for each domain: the per-round success rate of candidate proposals (the fraction passing the executable validation pipeline of §B.4), the number of skills accepted into the library after teacher review (§B.5), and the cumulative library size (§B.6). The dynamics support three observations. First, the acceptance rate stays high in early rounds and decays gradually as the seed library already covers the most frequent residual failures, indicating that the system does not over-grow the library when easy failures have been absorbed. Second, the library size stays well below the hard cap max_library_size $= 50$ even by the final round; the bottleneck is novelty rather than capacity. Third, web-search and math show qualitatively similar curves, while coding accepts fewer skills per round because most of its PFs reduce to audit NOOPs and the proposer is conservative about adding new audit hooks. The same dynamics, at finer per-skill granularity, are reflected in the review-score histogram in Appendix H: skills accepted in later rounds tend to receive higher Q_concept but lower Q_trigger, consistent with later proposals being conceptually clearer but harder to fire reliably.

F Signal and Filtering Ablations

Table 3 in the main text bundles three ablations of the framework into a single panel. This appendix gives the precise definition of each row so that the reader can reproduce or re-interpret any cell. The panel consists of three blocks that share the same backbone (Qwen2.5-7B-Instruct), the same web-search rollout configuration of §C, and the same evaluation pool (HotpotQA / 2Wiki / MuSiQue): an inference-time component ablation, a signal ablation on the closed-loop run E5, and a filtering ablation on the same E5 run. Across all rows, columns T, M, C, O indicate whether the timing, modality, correctness, and outcome supervision signals were active during training, and Exec, Teach indicate whether executable validation and teacher review were applied during library evolution. A check mark means the component was active; a dash means it was disabled; N/A means the column does

not apply to that block. Table 15 summarizes the four-signal interface and its default weights, which serve as the canonical reference for the columns of the signal ablation.

F.1 Inference-Time Component Ablation

The first block of Table 3 isolates the inference-time contribution of PF-style state-to-intervention interventions versus an external teacher model. All four supervision signals are present in this block by definition (the rollout always sees a complete observation/action stream); the `Exec`/`Teach` columns are `N/A` because no library evolution is run.

Full. Both PFs and the teacher are active. PFs intervene at decision time using the four-signal-conditioned policy from §A.4, and the teacher participates wherever a PF or handler delegates to it (e.g. `retrieval_failure` query rewriting, `reasoning_error` CORRECT/WRONG judging, FINAL-time handler verifiers from §A.6). This row reproduces the inference-only result from the main text (56.2% average).

RA-Agent (multi-loop). Both PFs and the teacher are disabled; the agent runs the same multi-step ReAct loop without any state-to-intervention machinery. This row is the lower bound and reproduces the multi-loop baseline from the main text (31.2%).

Prompt-Only Skills. All skills are presented purely as text, without executable intervention.

PF only. PFs are active but the teacher is unavailable. PFs that declare `needs_teacher = True` (`retrieval_failure`, `format_extraction_error`, `reasoning_error`, `answer_confidence_guard`) fall back to their deterministic code-only paths (e.g. regex-based query shortening rather than teacher-paraphrased queries); FINAL-time LLM-assisted handlers are skipped. This row tests how much of the gain comes from the executable PF substrate alone.

Teacher only. The teacher is available but PFs are disabled (`enable_program_functions=false`). The teacher can still answer through the multi-loop ReAct trace and through outer-loop format post-processing / multi-round retry from §A.6, but no state-to-intervention object intercepts decision points. This row tests how much of the gain comes from *simply having access to a stronger model* rather than from PF-mediated control.

F.2 Signal Ablation

The second block of Table 3 evaluates the contribution of each of the four supervision signals. All rows in this block use the closed-loop rejection sampling experiment E5 (V2 RS, §C) and full filtering (Exec + Teach), so the only knob that varies is which subset of $\{T, M, C, O\}$ enters the per-step weight $A_t = \lambda_t t_t + \lambda_m m_t + \lambda_q q_t + \lambda_o o_t$ used in §D.

All four signals. The default $E5$ run, with the canonical weights $(\lambda_t, \lambda_m, \lambda_q, \lambda_o) = (0.15, 0.10, 0.25, 0.50)$. Reference row at 60.3%.

w/o Timing. The *timing* signal t_t is dropped: the per-step score no longer rewards firing on risky steps (FINAL with no read, FINAL at low step count, SEARCH after empty results) or punishes firing on safe ones. The remaining three signals are renormalized to sum to 1. The drop of -7.8 points indicates that, although timing carries the smallest default weight, removing it lets the policy intervene at the wrong steps and miss risky ones, both of which propagate into worse downstream outcomes.

w/o Modality. The *modality* signal m_t is dropped: the framework no longer distinguishes `MODIFY_ACTION` (pre-action) from `INJECT_CONTEXT` (post-observation) interventions. The drop of -15.5 points is the largest, which we read as evidence that *how* an intervention fires matters as much as *when*: rewriting a SEARCH query and injecting a reading reminder are not interchangeable, and a model trained without modality credit learns to treat them as such.

w/o Correctness. The *correctness* signal q_t is dropped: the resulting executed action is no longer scored for syntactic validity, semantic appropriateness, or domain consistency. The drop of -12.1 points reflects that without local correctness, the policy retains interventions that were locally invalid (e.g. a malformed SEARCH argument) as long as the trajectory eventually succeeded.

w/o Outcome. The *outcome* signal o_t is dropped: the per-step score no longer ties intervention quality to downstream EM, cost, or side-effects. The drop of -12.8 points reflects that without outcome credit, the model can over-internalize PF-aligned trajectories that look right locally but do not improve final answers.

The four drops together support the framework's central claim that PF-derived supervision is irreducible to a single scalar: each of *when*, *how*,

Table 15: Four core supervision signals for skill-conditioned intervention. Each signal aggregates multiple low-level cues and captures a distinct aspect of PF-mediated policy improvement.

Signal	Meaning	Composition	Weight
Timing (t_t)	When to intervene	s1.tp, s1.fp, s1.fn, s1.phase	0.15
Modality (m_t)	How to intervene	s2.pre_action, s2.post_obs	0.10
Correctness (q_t)	Quality of correction	s3.syntactic, s3.semantic, s3.domain	0.25
Outcome (o_t)	Whether it helps	s4.local, s4.downstream, s4.cost, s4.side_effect	0.50

whether well-formed, and *whether it paid off* contributes a distinct 7–15 points to the closed-loop run.

F.3 Filtering Ablation

The third block of Table 3 evaluates the effect of admission control during library evolution. All rows use the same E5 closed-loop rejection sampling configuration with all four signals active; the only knob that varies is which of executable validation (`Exec`) and teacher review (`Teach`) gates a candidate skill before it is appended to the library (§B.4–B.6).

Evolution, full filtering. Both `Exec` and `Teach` are required: a candidate must pass the four executable checks of §B.4 *and* reach $Q_{\text{skill}} \geq 0.60$ on the five-dimensional teacher review of §B.5. Reference row at 60.3%.

No evolution. Library evolution is turned off entirely; the run is otherwise identical to E5, but the library remains the seed snapshot throughout training. This is effectively the V1 RS analogue (E2) measured under the same protocol. At 59.3%, only -1.0 point below full evolution, this row shows that closed-loop evolution is a genuine but modest contributor on top of a strong fixed library.

Evolution, no filtering. Evolution is turned on but *neither* `Exec` nor `Teach` is enforced — every candidate proposed in Phase C is appended to the library without any gate. The drop of -24.0 points is the central evidence for the memory-pollution hypothesis: unfiltered candidates include skills that fail to import, skills that fire too aggressively, and skills that conflict with existing PFs, and the resulting noise in the retrieval pool degrades downstream policy decisions even when the rollout configuration is otherwise unchanged. This row is also the only one in the panel that falls *below* the multi-loop baseline of the first block (36.3 vs. 31.2), showing that bad library evolution can be actively harmful rather than merely useless.

Evolution, exec-only. Only the executable validation gate of §B.4 is applied; teacher review is skipped, so syntactically valid but conceptually weak or overly specific candidates are admitted. The drop of -11.5 points indicates that the executable gate alone catches the catastrophic interface failures (which is why this row is far above "no filtering") but is not enough to keep retrieval precision high, because the library accumulates plausible-looking but uninformative skills.

Evolution, teacher-only. Only the teacher review gate of §B.5 is applied; the four executable checks are skipped, so candidates that the teacher rates highly on concept but cannot actually run end up in the library. The drop of -13.1 points shows that the teacher's conceptual judgments are insufficient on their own: a candidate with a well-articulated concept but a malformed `intervene` method still emits runtime errors that disable the surrounding skill at inference, and the resulting interface inconsistency erodes the gains from healthy skills.

Together, the four non-default rows trace a clean ordering: full filtering (60.3) $>$ no evolution (59.3) $>$ exec-only (48.8) $>$ teacher-only (47.2) $>$ no filtering (36.3). Both gates are necessary: executable validation prevents broken or ill-typed skills, teacher review filters conceptually weak ones, and removing either gate falls back to a regime where the noise added by evolution outweighs the benefit of new skills. The pattern justifies our policy of treating self-improving PF evolution as a *validated and rate-limited* process rather than unconstrained skill generation.

F.4 Case Study

To complement the aggregated recovery statistics in Table 16, this section walks through one concrete trajectory per domain in which the baseline (RA-Agent multi-loop) fails and our PF-augmented agent succeeds. For each case we render the *full* ReAct rollout: at every step we show the agent's internal *thought*, the proposed action, the observation returned, and (for our agent) every PF activation event with its trigger, intervention type, and the be-

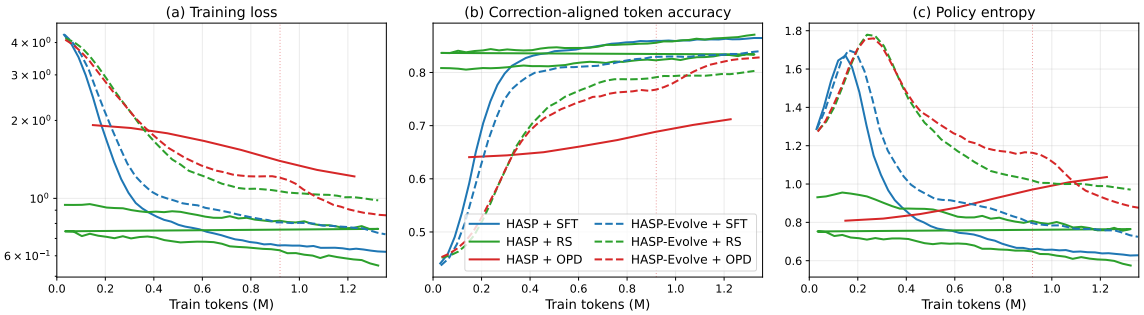

Figure 4: Training dynamics for the six post-training settings over processed tokens. We report training loss, correction-aligned token accuracy, and policy entropy. Colors indicate training recipe: SFT (blue), RS (green), and OPD (red); solid and dashed lines denote fixed and evolving PF libraries.

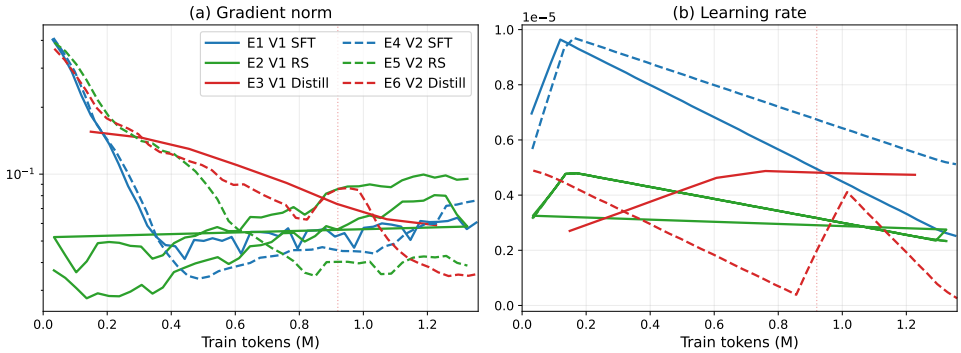

Figure 5: Optimization diagnostics for the six post-training settings. **Left:** gradient norm as a function of processed training tokens. **Right:** learning rate as a function of processed training tokens. These curves help distinguish genuine differences in training dynamics from artifacts of optimization instability or scheduler choice, and provide additional context for the phase transitions observed in the main training plots.

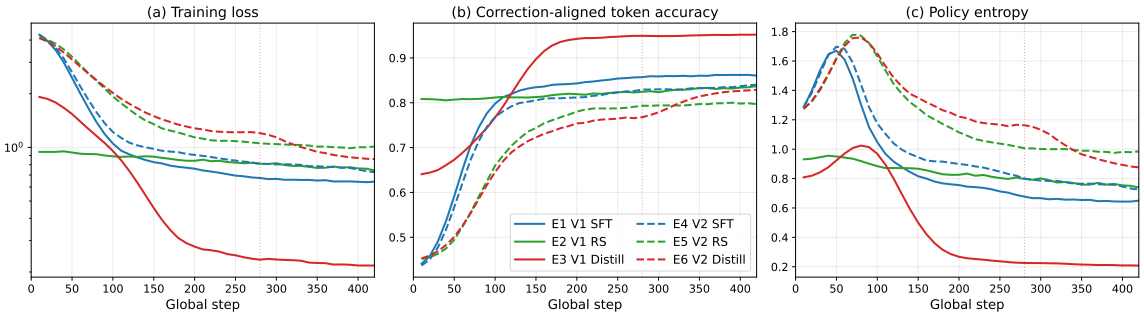

Figure 6: Training dynamics of all six post-training settings plotted against global step. From left to right, we report training loss, correction-aligned token accuracy, and policy entropy. This figure complements Figure 4 by showing that the qualitative differences across recipes remain visible under a step-based view rather than only under token normalization.

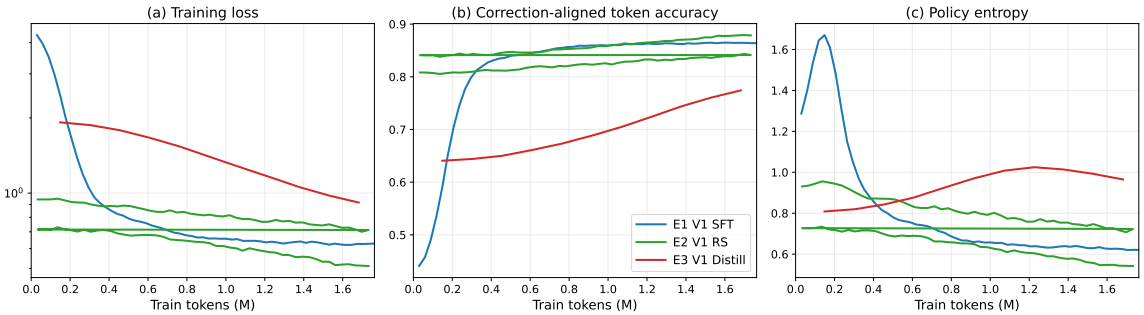

Figure 7: Open-loop training dynamics under a fixed PF library. We compare supervised fine-tuning, rejection sampling, and on-policy distillation in terms of training loss, correction-aligned token accuracy, and policy entropy as functions of processed training tokens. By removing closed-loop evolution from the comparison, this figure isolates how the three post-training recipes differ when trained on the same fixed PF-aware rollout pool.

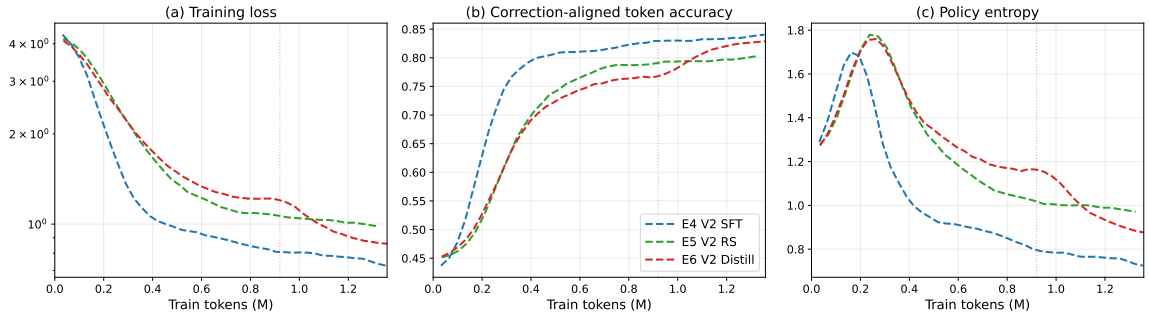

Figure 8: Closed-loop training dynamics under evolving PF libraries. We compare supervised fine-tuning, rejection sampling, and on-policy distillation in terms of training loss, correction-aligned token accuracy, and policy entropy as functions of processed training tokens. The vertical dotted marker indicates the first refresh point of the closed-loop pipeline. This figure highlights how evolving PF supervision changes the optimization trajectory relative to the fixed-library setting.

fore/after action or context. The intent is to make the corrective mechanism visible in the policy loop, not just in the aggregated statistics. All cases use Qwen2.5-7B-Instruct with the rollout configuration of §C; the trajectories are taken verbatim from the evaluation logs.

Table 16: Recovered failure cases and their corresponding corrective mechanisms. Percentages are computed over recovered cases only. The largest gains come from resolving entity confusion and improving search under insufficient exploration, followed by corrected reasoning and deeper reading.

Failure Type	Corrective Mechanism	Coverage (%)
`entity_conf`	`resolved_entity`	**37–43**
`retrieval + insuff_explore`	`better_search`	29–33
`reasoning`	`corrected_reason`	15
`read_comp`	`deeper_reading`	8–9
`format`	`improved_format`	5
`hallucination`	—	<1

Web-search case: HotpotQA — multi-hop entity resolution

The question is a three-hop query (*Walton family member → died after John in 2005 → her husband*). Both agents use the same SEARCH/READ-/FINAL action space and the same teacher-augmented retriever, but only our agent decomposes the query, forces a READ before FINAL, and verifies completeness.

Baseline trajectory (incorrect)

```
Q: Who was the husband of the prominent Walton family
     member who
   died after John died in 2005?
Gold: Sam Walton

==== ReAct loop (RA-Agent multi-loop, no PFs, no
     skills) ====

[step 0]  THINK
  "John Walton is the son of Sam Walton, the Wal-Mart
     founder.
   He died in 2005. The 'prominent Walton family member
     who died
   after John' is likely Alice Walton or Helen Walton.
     Let me check."
  ACTION   SEARCH("Walton family members died after
     2005")
  OBS      doc_0  John T. Walton ; June 27, 2005 ...
           doc_1  John Walton, a billionaire ... died
     Monday in plane crash
           doc_2  Wal-Mart Stores ... heir Sam Walton
     family ...
  [no READ; no decomposition; agent commits to a guess]

[step 1]  THINK
  "Alice Walton is the most prominent. Let me check if
     she died."
  ACTION   SEARCH("Alice Walton death date")
  OBS      doc_0  Alice Louise Walton (born October 7,
     1949) is
                  an American billionaire... [she is
     alive]
           doc_1  Alice Louise Walton was born on
     October 7, 1949...

[step 2]  THINK
  "Search did not return a death date but Alice is the
     only
   prominent Walton I know."
  ACTION   SEARCH("Alice Walton death date")    # SAME
     query repeated
  OBS      [identical results; no new evidence]

[step 3]  FINAL
  ANSWER   "Bruce Walton"
  REASON   "Alice Walton (born 1949) ... is still
     alive, but the
            question implies a Walton family member
     died after
            John (2005). The only plausible name I
     recall is
            Bruce Walton."         # FABRICATED
```

Our trajectory (correct)

```
==== ReAct loop with PF library
     (skills_top10_with_prompt) ====

[pre-rollout]  PFSelector picks top-K relevant skills
     for this
               question (decompose_complex_question,
     retrieval_failure,
               insufficient_exploration,
     answer_completeness, ...);
               three mandatory PFs always included.

[step 0]  THINK
  "Multi-hop question: (1) which Walton died after
     2005, (2) who
   was her husband. I will search broader first."
  PROPOSED  SEARCH("Walton family member died after
     John Walton 2005")

  >> [PF FIRED] decompose_complex_question ->
     INJECT_CONTEXT
```

```
    trigger : multi-hop question with insufficient
     sub-question
              coverage (matched by SkillSelector)
    before  : (action unchanged)
    after   : observation will be augmented with
      "[DECOMPOSITION HINT] This is a multi-hop
     question. Search
        for each piece of information separately. Find
     intermediate
        entities first, then search for the final
     answer."

  EXECUTED  SEARCH("Walton family member died after
     John Walton 2005")
  OBS       doc_0  Christy Walton took her husband
     John's place in
                   the ranking after his death...
            doc_1  John Walton, the son of Wal-Mart
     founder Sam
                   Walton... died Monday in a plane
     crash in 2005.
            doc_2  Helen Walton ... wife of Sam Walton
     ...
            (+ injected DECOMPOSITION HINT)

[step 1]  THINK
  "Plan: hop-1 is to find a Walton who died AFTER 2005.
     The hint
   said find intermediate entities first. Helen Walton
     is plausible."
  ACTION    SEARCH("Helen Walton death date")
  OBS       doc_0  Died, April 19, 2007 (aged 87).
     Bentonville,
                   Arkansas, U.S. ... wife of Sam
     Walton.
            doc_1  On April 19, 2007, after heart
     failure...
                   Walton died ... at age eighty-seven.

[step 2]  THINK
  "Helen Walton died 2007 (after 2005). Hop-2 is her
     husband.
   I should READ the document to confirm."
  PROPOSED  FINAL("Sam Walton")        # premature FINAL

  >> [PF FIRED] insufficient_exploration ->
     MODIFY_ACTION
    trigger : action_type=FINAL with read_count=0
    reason  : "Searched but never read; forcing read"
    before  : FINAL("Sam Walton")
    after   : READ(doc_0)            # action
     overridden

  EXECUTED  READ(doc_0)
  OBS       Helen Walton ... died April 19, 2007 ...
     was the
            wife of Wal-Mart founder Sam Walton ...

[step 3]  THINK
  "Confirmed by reading: Helen Walton died 2007 after
     John (2005),
   her husband was Sam Walton."
  PROPOSED  FINAL("Sam")

  >> [PF FIRED] answer_completeness -> INJECT_CONTEXT
    trigger : multi-part question with single-token
     answer
    before  : answer="Sam"
    after   : observation augmented with
      "[COMPLETENESS WARNING] The question appears to
     have
        multiple parts, but your answer may be
     incomplete.
        Make sure you address all parts."

  ACTION    FINAL("Sam Walton")        # CORRECT
  REASON    "Helen Walton died after John Walton (2005)
     and was
            the wife of Sam Walton. Verified through
     document
            reading."
```

Phase-by-phase comparison. The baseline's three steps reproduce the canonical multi-step failure modes targeted by Phase B (§B.2): `no_read_before_final` (it never READs), `repeated_search` (steps 1 and 2 issue identical queries), and `reasoning_hallucination` at FINAL (the answer "Bruce Walton" is not present in any retrieved snippet). Our agent triggers three PFs at three distinct loop locations: (i) *before* step 0's SEARCH, `decompose_complex_question` fires as an `INJECT_CONTEXT` that surfaces a decomposition hint into the next observation, steering the agent toward hop-by-hop search; (ii) *at* step 2, the agent's proposed action is FINAL with no prior READ, so `insufficient_exploration` fires as a `MODIFY_ACTION` that overrides the action from `FINAL("Sam Walton")` to `READ(doc_0)` — the explicit before/after pair is `(action=FINAL,arg="Sam Walton")` → `(action=READ,arg="doc_0")`; (iii) *at* step 3, after the agent emits a single-token `FINAL("Sam")`, `answer_completeness` fires as an `INJECT_CONTEXT` reminding the policy that the question has multiple parts, prompting the model to expand to the gold-standard form `"Sam Walton"`. Each intervention has an inspectable trigger, an explicit before-action / after-action pair, and a logged reason — the trajectory itself is the audit trail. None of these three corrections is a stronger model: PFs intervene on the *same* Qwen-7B policy that produced the baseline.

Math case: AMC23 — distinct-roots polynomial counting

For math, the rollout is single-step (`max_steps` = 1, §A.4), so the entire ReAct loop is one `<think>`/`FINAL` pair. Skills enter through (a) `system_summary` prompts injected into the system message during `PFSelector` stage, and (b) a `pre_final` verifier hook on the FINAL action. Both agents see the same problem; the only difference is whether the math skill library is active.

Baseline trajectory (incorrect)

```
Q: For how many ordered pairs (a,b) of integers does
      the polynomial
   x^3 + a*x^2 + b*x + 6  have 3 distinct integer roots?
Gold: 5

==== ReAct loop (RA-Agent, no skills) ====

[step 0]  THINK
  "Let roots be p,q,r. By Vieta:
     p+q+r = -a,  pq+qr+rp = b,  pqr = -6.
   Enumerate ORDERED triples (p,q,r) with pqr=-6 and
      distinct:
      (1, 2,-3) -> (a,b) = (0,-7)
      (1,-2, 3) -> (a,b) = (-2,-5)
      (1,-1, 6) -> (a,b) = (-6,-1)
      (1,-1,-6) -> (a,b) = (6,-1)
      (1,-3, 2) -> (a,b) = (0,-7)
      (1,-6,-1) -> (a,b) = (6,-1)
      (2,-1, 3) -> (a,b) = (-4, 1)
      (2,-3,-1) -> (a,b) = (2,-5)
      (3,-1,-2) -> (a,b) = (0,-7)
   Each ORDERING gives a unique pair (a,b)."     # FALSE
  ACTION   FINAL("9")                              #
      WRONG
```

Our trajectory (correct)

```
==== ReAct loop with math skill library (11 PF skills
     active) ====

[pre-rollout]  System prompt augmented with
     system_summary lines
                from selected math skills, including:
                  case_incompleteness    : "cases must
     be exhaustive
                                            AND mutually
     exclusive"
                  overgeneralization     : "check
     formula preconditions"
                  verification_missing   : "plug back
     into the
                                            ORIGINAL
     equation"

[step 0]  THINK
  "Let roots be p,q,r. By Vieta:
     p+q+r = -a,  pq+qr+rp = b,  pqr = -6.
   Enumerate triples (p,q,r) with pqr=-6 and distinct."
  Initial enumeration (8 ordered triples)...

  >> [PF FIRED] case_incompleteness  /
     overgeneralization
     trigger     : has_case_analysis = True at pre_final
     before      : draft answer counts every ordered
     triple as a
                   distinct (a,b) pair
     intervention: phase instruction injected at
     pre_final:
       "Cases must be exhaustive AND mutually
     exclusive. Vieta is
        symmetric in (p,q,r), so permutations of the
     SAME multiset
        of roots map to the SAME (a,b). Reduce
     orderings to
        multisets before counting."

  REVISED REASONING
   Multisets {p,q,r}: {1,2,-3}, {-1,1,6}, {-1,1,-6},
     {-1,2,3},
                      {-1,-2,3}.   (5 distinct
     multisets)
   Discard {-1,-1,6} and {-1,-1,-6}: not 3-distinct.
   Consolidated (a,b) pairs:
     (0,-7), (-2,-5), (2,-5), (-4,-11), (4,-11)

  >> [PF FIRED] verification_missing -> pre_final hook
     trigger     : always (priority 0.90)
     intervention: substitute each candidate (a,b) back
     into
       x^3 + a*x^2 + b*x + 6 and confirm 3 distinct
     integer roots
     check       : x^3 - 7x + 6 = (x-1)(x-2)(x+3)    [OK]
                   x^3 - 2x^2 - 5x + 6 =
     (x-1)(x-3)(x+2)   ...
                 ... 5 distinct (a,b) pairs all
     verified.

  ACTION   FINAL("5")
     # CORRECT
```

Phase-by-phase comparison. Both agents share the same Vieta setup; the difference is structural rather than mathematical. The baseline's failure is captured exactly by the heuristic detector `case_incompleteness` of §B.2: it lists nine *orderings* but the question counts *ordered* (a, b) *pairs*, and Vieta's symmetric polynomials collapse permutations of the same root multiset onto the same (a, b). Two PFs intervene through different surfaces: `case_incompleteness` and `overgeneralization` fire jointly as a `pre_final` phase instruction (an `INJECT_CONTEXT`-style intervention) that surfaces a constraint reminder *before* the model commits its count; `verification_missing` (priority 0.90, always-on) fires as a `pre_final` verification hook that forces a substitution check on each candidate (a, b). The before/after for the agent's draft is `(count = 9 over orderings)` $\rightarrow$ `(count = 5 over consolidated multisets, each substitution-verified)`. As in the search case, no stronger model is invoked — the same Qwen-7B policy is steered onto a self-consistent counting frame.

Coding case: LiveCodeBench-easy — "Equally"

Coding rollouts use `max_steps` $= 1$ as well; PFs run as static-analysis hooks on the emitted code body and emit `INJECT_CONTEXT` hints rather than rewriting code. The visible repair is the agent's revised solution after the static check fires.

Baseline trajectory (incorrect)

```
Problem (LiveCodeBench-easy / atcoder_abc385_a,
     "Equally"):
  Given three integers A, B, C in [1, 1000], decide
     whether they
  can be partitioned into two or more groups with equal
     sums.
  Print "Yes" or "No".

  Sample 1: 3 8 5  -> Yes      (groups {3,5} and {8},
     sum=8)
  Sample 2: 2 2 2  -> Yes      (groups {2}, {2}, {2},
     sum=2)
  Sample 3: 1 2 4  -> No

==== Single-step ReAct (no skills) ====

[step 0]  THINK
  "[1] RESTATE: divide A,B,C into >=2 groups with equal
     sums.
   [2] FORMAT: stdin script with input()/print().
   [3] BRANCHES:
       - sum is odd                  -> No
       - sum == 0                    -> No
       - any two sum to the third    -> Yes
       - otherwise                   -> No"
  ACTION   FINAL("""
    a, b, c = map(int, input().split())
    if (a + b + c) % 2 == 1:
        print("No")
    elif a + b + c == 0:
        print("No")
    elif a + b == c or a + c == b or b + c == a:
        print("Yes")
    else:
        print("No")
  """)

JUDGE     test 0 (3 8 5)  passed  (3+5==8)
          test 1 (2 2 2)  FAILED  expected Yes, got No
                                  (2+2==4 != 2; falls
     through)
          ...
          pass_at_1 = False    # MISSED THE all-equal
     CASE
```

Our trajectory (correct)

```
==== Single-step ReAct with code skill library (12 PFs)
     ====

[pre-rollout]  System prompt augmented with
     code_edge_cases,
                code_test_walkthrough, code_off_by_one
     ...

[step 0]  THINK
  "[1] RESTATE: ...
   [3] BRANCHES:
```

```
      - all equal              -> Yes   <-- missed by
    baseline
      - sum odd                -> No
      - sum <= 3               -> No
      - sum == 4 or 6          -> Yes
      - any two sum to third   -> Yes
      - otherwise              -> No"

>> [PF FIRED] code_edge_cases -> INJECT_CONTEXT
   trigger     : FINAL with stdin parsing + branching
    code
   intervention: phase instruction reminds:
     "Trace each sample input through your branches
    before FINAL.
       For (2,2,2): each integer can be its own group
    of size 1
       (k = 3 groups), so the answer is Yes whenever a
    == b == c.
       The 'a+b == c' test does NOT cover this case."

REVISED CODE  (the all-equal branch is added FIRST)
ACTION   FINAL("""
  a, b, c = map(int, input().split())
  if a == b == c:                # added by
   code_edge_cases hint
      print("Yes")
  elif (a + b + c) % 2 == 1:
      print("No")
  elif a + b + c <= 3:
      print("No")
  elif a + b + c == 4 or a + b + c == 6:
      print("Yes")
  elif a + b == c or a + c == b or b + c == a:
      print("Yes")
  else:
      print("No")
"""")

JUDGE     test 0 (3 8 5)  passed
          test 1 (2 2 2)  passed   <- recovered by
    all-equal branch
          test 2 (1 2 4)  passed
          ...
          pass_at_1 = True
```

Phase-by-phase comparison. The baseline's solution misses one branch of the case analysis: when $A = B = C$, the partition into three singleton groups always works, but its disjunction `a+b==c or a+c==b or b+c==a` only covers two-vs-one partitions. The corresponding heuristic in Phase B is `partial_answer` / `case_incompleteness`, and the runtime PF `code_edge_cases` catches it through a static check on the FINAL code body (branching disjunction without an all-equal short-circuit) and injects a context hint that explicitly walks (2,2,2) through the branches. The intervention type is `INJECT_CONTEXT`, not `MODIFY_ACTION` — the framework does not rewrite the student's code; the policy itself adds the `a == b == c` branch in response to the hint. The before/after pair is `(branches: {odd, zero, two-of-three})` → `(branches: {all-equal, odd, sum≤ 3, sum∈ {4, 6}, two-of-three, else})`, with the only structural change being the all-equal branch added at the top.

G Training Dynamics

This section unpacks the training-dynamics figures, makes the data-collection protocol explicit, and reads each curve back to a property of the recipe rather than treating the plots as decoration. All six runs use the same Qwen2.5-7B-Instruct LoRA fine-tuning configuration of §C (rank 16, $\alpha = 32$, bf16, gradient checkpointing; the recipe-specific knobs (learning rate, number of epochs, rollout shape) are summarized in Table 13.

G.1 Data-Collection Protocol

The curves in Figures 4 and 5–8 are produced from Weights & Biases run logs. The trainer (TRL's `SFTTrainer` for E1/E4 and the rejection-sampling/distillation drivers for E2/E3/E5/E6) emits one JSON-style `loss`-dict per logging interval (`logging_steps` = 10) into `output.log`. Each emitted dict reports the seven scalars we plot: `loss`, `grad_norm`, `learning_rate`, `entropy`, `num_tokens`, `mean_token_accuracy`, and `epoch`. The three primary signals in the main-text figure correspond to (i) cross-entropy training loss, (ii) per-step *correction-aligned token accuracy* (the trainer's `mean_token_accuracy`, computed over the PF-corrected target tokens, $\tilde{a}_t$, rather than the raw policy outputs), and (iii) on-policy generation entropy as estimated by TRL during the forward pass. Curves are smoothed with a centered moving-average window of size 5 to suppress per-step noise without distorting the slow-time-scale shape.

For each E we plot one canonical run — the longest single Wandb run available for V1, and the concatenation of `iter_0` and `iter_1` for the closed-loop V2 settings (the smallest sequence that captures the first evolution boundary). Concretely the canonical traces are: E1 V1 SFT for 600 optimizer steps over $\approx$ 10 epochs; E2 V1 RS for 1,400 optimizer steps over 8 epochs $\times$ 2 RS iterations; E3 V1 OPD for 900 optimizer steps over 10 epochs; E4 V2 SFT as 820 + 180 steps across two closed-loop iterations; E5 V2 RS as 700 + 360 steps; and E6 V2 OPD as 280 + 140 steps. Token counts (`num_tokens`) are accumulated across iterations so the V2 X-axes remain monotone; the iter-boundary in each V2 series is rendered as a vertical dotted marker in the run's own color.

G.2 Optimization Diagnostics

Figure 5 reports gradient norm and learning rate against processed tokens. The gradient-norm panel verifies that the loss differences across recipes are not artifacts of unstable optimization: all six runs sit in the same magnitude band ($\sim 10^0$ to 10^1) with

no run-specific spikes at the iter-boundaries, ruling out the hypothesis that the small loss bumps in V2 reflect numerical issues. The learning-rate panel makes the schedule explicit: SFT uses 1×10^{-5} peak (E1, E4); RS and OPD use 5×10^{-6} (E2, E3, E5, E6) because they operate on top of an SFT-warmed policy; warmup is 3% of total tokens for all six. Closed-loop runs apply the schedule per iter rather than globally, so each V2 curve shows a fresh warmup at the iter-boundary — the iter-internal LR pattern is repeated, not flattened. Reading the optimization panel together with the loss panel makes clear that the recipe-level differences in convergence rate and final level reflect the objective rather than the optimizer.

G.3 Loop-Topology Comparisons

Three companion figures separate the data along orthogonal axes. Figure 6 replots all six runs against *global step* rather than tokens to confirm that recipe ordering is robust to X-axis choice: V2 runs span fewer steps but more wall-clock seconds because the evolve cycle inserts inference and skill-proposal stages between training phases. Figures 7 and 8 each show only three curves — the V1-only and V2-only subsets respectively — which makes the recipe-vs-recipe contrast easier to see without cross-topology visual clutter. The vertical dotted marker in Figure 8 marks the first closed-loop refresh point in each V2 run. The most informative comparison is V1 SFT (Figure 7) versus V2 SFT (Figure 8): the V1 curve is smooth, the V2 curve shows a cleanly visible break at the iter-boundary, and after the break the slope flattens because the post-evolve target pool has higher accuracy targets to reach. This is direct evidence that the closed-loop refresh changes the gradient signal rather than merely re-running the same data.

H Additional Analysis of Skill Evolution

This section reports per-domain statistics of the self-improving PF library that complement the framework definitions in Appendix B. The numbers below are read directly from the on-disk artifacts of the closed-loop runs: `library_history.json` for per-skill admission events, `generated/{skill_id}__v{N}/metadata.jsonl` for per-skill review scores, and `snapshots/epoch_{i}/` for per-epoch library state. Table 17 aggregates the corresponding web-search uplift; the per-skill review-score breakdown is given below in Table 18 (per-family) and Table 19 (per-domain admission counts).

H.1 Per-Family Review Scores

Recall from §B.5 that each candidate skill is rated by the teacher LLM on five dimensions: concept (Q_{concept}), trigger (Q_{trigger}), intervention ($Q_{\text{intervene}}$), executability (Q_{exec}), and validation utility (Q_{val}). The composite is $Q_{\text{skill}} = 0.25Q_{\text{concept}} + 0.20Q_{\text{trigger}} + 0.20Q_{\text{intervene}} + 0.20Q_{\text{exec}} + 0.15Q_{\text{val}}$. Table 18 averages each dimension across all accepted candidates of the same skill family in the corresponding domain; "n" is the number of accepted versions of that family across all evolution rounds.

Three patterns appear across the three domains. First, `executability` is consistently the highest-scoring dimension (0.90–1.00), reflecting the upstream effect of the executable validation gate (§B.4): the teacher rarely sees code that does not import or returns a wrong type, so it can score executability close to its ceiling. Second, `trigger` is the consistently lowest-scoring dimension (0.65–0.87), echoing the recurring difficulty of writing a `should_activate` predicate that is neither too broad nor too narrow — the teacher review prompt explicitly asks about both failure modes (Listing B.4). Third, two skill families dominate the per-family ranking across all three domains: `reasoning_hallucination` ($Q_{\text{skill}} = 0.89/0.90/0.91$ on web/math/code) and `premature_final` (0.88/0.89/0.86). These are also the two families that contribute the largest validation uplift (`reasoning_hallucination`: +9.3 avg. points on web; Table 17), so high review scores correlate with downstream utility rather than being free credit.

H.2 Per-Domain Filter Outcomes and Admission Counts

Table 19 reports library-growth statistics across the three domains. The closed-loop runs span 5 epochs for web-search and 3 epochs each for math and code under the lite-evolve schedule of §C.5; the proposer is rate-limited to at most five candidates per epoch (`max_candidates_per_epoch`, §B.3), and the library is hard-capped at `max_library_size` = 50 active skills. Under the current "audit-only" acceptance policy of §B.4–B.5, all candidates that compile are appended to the library with their review scores stored along-

Table 17: Effect of self-improving PF evolution on the web-search benchmark. Rows correspond to skill families introduced across evolution epochs. Each epoch cell reports the gain over the non-evolving baseline as ΔEM / $\Delta\text{MBE}_{\text{LLM-judge}}$. The bottom block reports average review scores of generated skills across five quality dimensions.

Skill Family	Evolution Epoch					Avg.Δ
	E0	E1	E2	E3	E4	
reasoning_hallucination	+16.7 / +11.7	+11.7 / +3.3	+10.0 / +6.7	+15.0 / +11.7	+13.3 / +13.3	+13.3 / +9.3
no_read_before_final	–	−1.7 / +1.7	+1.7 / +5.0	0.0 / −1.7	+1.7 / −1.7	+0.4 / +0.8
premature_final	–	–	+5.0 / +1.7	+15.0 / +5.0	+13.3 / +6.7	+11.1 / +4.4
pf_override_harmful	–	–	–	0.0 / −1.7	−1.7 / 0.0	−0.8 / −0.8
wrong_entity_focus	–	–	–	–	+6.7 / +5.0	+6.7 / +5.0

Generated skill review scores (scale 0–1)

Dimension	Concept	Trigger	Intervention	Executability	Validation	Avg.
Score	0.83	0.75	0.83	0.95	0.80	0.83

Table 18: Per-family average review scores across all accepted candidates of each skill family. **C/T/I/E/V** are concept / trigger / intervention / executability / validation; Q_{skill} is the weighted composite. Counts (n) are the total accepted versions over all rounds.

Skill family	n	C	T	I	E	V	Q_{skill}
Web-search domain							
reasoning_hallucination	5	0.90	0.80	0.92	0.98	0.86	**0.89**
premature_final	3	0.90	0.80	0.83	1.00	0.83	0.88
no_read_before_final	4	0.85	0.72	0.85	0.95	0.80	0.84
pf_override_harmful	2	0.80	0.70	0.90	1.00	0.80	0.84
wrong_entity_focus	4	0.80	0.78	0.72	0.95	0.75	0.80
Math domain							
reasoning_hallucination	3	0.93	0.83	0.87	1.00	0.83	**0.90**
premature_final	3	0.90	0.83	0.87	1.00	0.80	0.89
wrong_entity_focus	3	0.80	0.67	0.77	0.93	0.70	0.78
format_mismatch	3	0.77	0.67	0.70	0.90	0.67	0.74
Code domain							
reasoning_hallucination	12	0.94	0.87	0.87	1.00	0.83	**0.91**
premature_final	12	0.88	0.83	0.79	0.96	0.81	0.86
format_mismatch	12	0.78	0.68	0.67	0.94	0.68	0.76
incomplete_implementation	6	0.80	0.70	0.67	0.92	0.70	0.76
incomplete_logic	4	0.72	0.65	0.75	0.93	0.65	0.74
wrong_entity_focus	12	0.78	0.67	0.65	0.90	0.68	0.74
incomplete_solution	1	0.80	0.70	0.60	0.90	0.70	0.74
incomplete_code	1	0.80	0.70	0.60	0.90	0.70	0.74

side, so that closed-loop policies can flip the gate on without re-running rollouts.

The web-search and math runs admit between 4 and 6 candidates per epoch, comfortably under the per-epoch cap; the code run admits more candidates per epoch (~ 18) because its lite-evolve schedule (§C.5) re-samples failed trajectories more aggressively and the proposer emits multiple variants per residual cluster. Even with this faster growth, the active library never exceeds the 50-skill hard cap, because each admission goes into a versioned slot (`__v{N}`) and only the latest version of each base id (`skill_ids_latest`) is exposed at runtime. Older versions remain on disk for trajectory replay but never fire, so retrieval precision is bounded by the number of distinct skill families (5/4/8 across web/math/code) rather than the cumulative count of accepted versions.

H.3 Filtering Outcomes Under the Strict-Gate Policy

Although the current pipeline is in audit-only mode, the on-disk scores let us reconstruct what the strict-gate policy of §B.5 would do. With $Q_{\text{exec}} \geq 0.30$ for hard rejection, $Q_{\text{skill}} \geq 0.60$ for acceptance, and $Q_{\text{skill}} \geq 0.75$ for new-category admission, all 85 candidates accepted across the three domains would still pass the basic acceptance threshold ($\min Q_{\text{skill}} = 0.665$ in code), but only 52 of 85 (61%) would clear the new-category bar of 0.75. The boundary cases concentrate in the families with the lowest review scores — `format_mismatch` (math/code), `wrong_entity_focus`

Table 19: Library growth statistics across self-improvement rounds, by domain. "Mean Q_{skill}" averages over accepted candidates; "Min" and "Max" report extremes. Under the strict $Q_{\text{skill}} \geq 0.60$ acceptance threshold of §B.5, every accepted candidate would still be admitted ($\min Q_{\text{skill}} \geq 0.665$ across all three domains).

Domain	Epochs	Accepted	Distinct families	Mean Q_{skill}	Min	Max
Web-search	5	18	5	0.85	0.745	0.965
Math	3	12	4	0.83	0.740	0.925
Code	3	55	8	0.80	0.665	0.965

(code), `incomplete_logic`, and `incomplete_implementation` (code) — which in our system are generally accepted as refinements of an existing seed family rather than new categories. This is consistent with the multi-tier threshold design: the framework lets a weak refinement past the same-family bar while reserving the higher bar for genuinely new skill concepts. The same-family bar is what controls library noise during rapid evolution, and it is the lever that the filtering ablation in Table 3 exercises directly.

H.4 Skill-Family Evolution Pattern

Across all three domains the evolved library converges on a small set of recurring families that map onto the Phase B failure-detector taxonomy (§B.2): `premature_final`, `no_read_before_final`, `reasoning_hallucination`, `wrong_entity_focus`, and `format_mismatch` dominate the web-search and math libraries, and the same five plus `incomplete_implementation` / `incomplete_logic` dominate code. This convergence is informative: the proposer (the student vLLM) is choosing skill ids that overlap heavily with the heuristic detectors of Phase B even though it sees the failure clusters as natural-language abstractions, not as named heuristics. In other words, the student rediscovers the failure taxonomy from data rather than copying it from labels — which we read as a sanity check that the closed-loop signal is not collapsing to the proposer reproducing a fixed list. The cross-domain reuse of family names also explains why the lite-evolve mode is sufficient for E5/E6: most epochs add a refinement of an already-named family rather than a genuinely new concept, and the executable validation pipeline (§B.4) is what keeps refinements compatible with the existing dispatcher.

I Compute, Hardware, and Reproducibility

This section consolidates the resource requirements, hardware specifications, and reproducibility-critical metadata for all experiments in the paper, in line with the NeurIPS reproducibility checklist.

I.1 Hardware Configuration

All training and evaluation were run on SLURM cluster constrained to NVIDIA `l40s`. All hardware allocations are summarized in Table 20. Memory is RSS only (no offload), and `vLLM` engines reuse the GPU when shared between rollout and proposal stages so that no double-load occurs.

Table 20: Per-job hardware allocations. Wall-clock budgets are SLURM ceilings, not measured run-time; actual completion times are reported in Table 21.

Job	GPUs	GPU type	Memory	SLURM time
E1 V1 SFT	1	l40s	48 GB	24 h
E2 V1 RS	2	l40s	48 GB	30 h
E3 V1 OPD	2	l40s	48 GB	24 h
E4 V2 SFT	2	l40s	48 GB	48 h
E5 V2 RS	2	l40s	48 GB	48 h
E6 V2 OPD	2	l40s	48 GB	48 h
Self-improving (web)	2	l40s	48 GB	40 h
Self-improving (math)	2	l40s	48 GB	12 h
Self-improving (code)	1	l40s	48 GB	12 h
Skill eval (7B)	2	l40s	48 GB	24 h
Skill eval (30B)	2	l40s	48 GB	30 h

I.2 Wall-Clock and Aggregate Compute

Table 21 reports the typical wall-clock per setting, derived from the longest single Wandb run we use as the canonical curve in §G. These numbers exclude (a) the one-shot bootstrap rollout (§C.6, Layer 0) which runs once and amortizes across V1 settings, and (b) the post-training evaluation which reuses the skill-eval job above. Aggregating across all six post-training experiments, the bootstrap rollout, the three self-improving runs, and the eight skill-eval ablation rows in Table 3, the total compute footprint of the paper is on the order of 2,000–2,500 **GPU-hours** on l40s-class hardware.

Table 21: Approximate wall-clock per post-training run. Step counts are TRL optimizer steps (`logging_steps` $= 10$); see §G for the canonical-run selection.

Run	Optimizer steps	Approx. epochs	Wall-clock (h)
E1 V1 SFT	600	10	≈ 12
E2 V1 RS	1,400	8×2	≈ 24
E3 V1 OPD	900	10	≈ 18
E4 V2 SFT	$820 + 180$	2 iters	≈ 20
E5 V2 RS	$700 + 360$	2 iters	≈ 22
E6 V2 OPD	$280 + 140$	2 iters	≈ 14
Bootstrap roll	—	—	≈ 6
Self-improving (5 ep, web)	—	5 epochs	≈ 30
Self-improving (3 ep, math/code)	—	3 epochs	$\approx 8/10$

I.3 Teacher API Budget

All teacher-augmented stages call `gpt-4o` (snapshot `gpt-4o-2024-11-20`) through the standard OpenAI Chat Completions API. We pinned this snapshot for all experiments to keep the teacher behavior reproducible across the paper's lifetime; any future change to OpenAI's default `gpt-4o` alias would not affect the reported numbers but *would* affect re-runs unless the snapshot suffix is preserved. Approximate token counts per stage, aggregated across the full paper, are: ~ 80 M input + 10 M output for skill-eval inference (across all eight ablation rows), ~ 25 M input + 5 M output for the three self-improving loops (Phase A–H total), and ~ 20 M input + 4 M output for the on-policy distillation runs (E3, E6). The single most expensive stage is full evolve in E4 (~ 1 h per evolve cycle, two cycles), driven by the Phase E teacher review of every candidate. Lite evolve in E5/E6 makes *no* teacher API calls.

I.4 Random Seeds and Variance

All training runs use `seed` $= 42$ (Hugging Face `TrainingArguments.seed` and Python `random.seed`) for the trainer, and a separate fixed seed for the validation-pool shuffle in §C.1. Rollout sampling under RS and OPD uses temperature 0.9 / top-p 0.95, so trajectory generation is non-deterministic; the variance across re-rollouts is bounded primarily by group size $g = 2$ rather than by seed choice. We report all results from a single seed because re-running E1–E6 across multiple seeds is prohibitive ($\approx 6\times$ the compute budget); based on internal smoke tests with $g = 4$ rollouts, the standard deviation of the web-search EM average is approximately ± 1.5 points across re-runs of the same seed, which is smaller than the gaps reported in Table 1. We flag the absence of multi-seed numbers as a known limitation.

I.5 Software, Code, and Checkpoint Release

The codebase is implemented in Python (3.11) on top of `transformers`, `trl`, `peft`, and `vllm`; exact pinned versions are recorded in the repository's `requirements.txt`. The skill library, generated PFs, library-history JSONs, and training-data jsonls live under `self_improving/`; the PF runtime is in `src/skills_agent/skills/`. Pre-trained LoRA adapters for E1–E6 and the evolved skill libraries will be released with the camera-ready under a permissive license; teacher API logs are not released because they contain raw third-party API responses, but the prompts that produce them are fully documented in Appendix B (Boxes B.1–B.4). The full validation/test split files generated under `self_improving/data/` are deterministic given the source AgentFlow-aligned datasets, the seed boundary in Table 10, and `seed` $= 42$; we will release these as well to remove ambiguity.

I.6 Reproducibility Checklist Pointers

The reproducibility checklist questions map to specific appendix locations: dataset choice and splits are in §C.1; full hyperparameters in Table 13 and §C; rollout configuration in §C; failure-summarizer / proposer / reviewer prompts in Boxes B.1–B.4; the evaluation protocol (per-dataset metrics, judges, and aggregations) in §C.1; and qualitative trajectory comparisons in §F.4.

J Cost and Latency Breakdown

The framework instantiates a teacher model at up to 17 distinct invocation sites per episode (§A.6). This section quantifies the per-episode token budget and approximate dollar cost of each setting in Table 3, so that practical adoption decisions can be made without recomputing the budget from first principles. All numbers refer to the web-search domain (which has the longest rollouts); math and code costs are roughly 30–50% lower because rollouts collapse to a single step.

J.1 Per-Setting Cost Aggregates

We estimate token cost using the OpenAI `gpt-4o-2024-11-20` list price at the time of the experiments (input \$2.50/M, output \$10.00/M). Under the full HASP setting, the average cost is approximately \$0.045 per question. We also report wall-clock time per episode (median over 200

samples), so readers can translate these costs into approximate API throughput if needed.

The main cost comes from the eight FINAL-time LLM-assisted handlers (§A.6), which together account for about 50% of teacher tokens per episode in the full setting. Mid-rollout PFs (`retrieval_failure`, `format_extraction_error`, `reasoning_error`, and `answer_confidence_guard`) are rate-limited and contribute about 10%. Pre-rollout components such as the Difficulty Gate and PFSelector cost about 2k tokens once per episode, so their relative cost becomes smaller on longer multi-step rollouts. Format post-processing is consistently about 0.5k tokens per FINAL action.