TEST TIME COMPUTING FOR EVOLVING AGENT SKILL OPTIMIZATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Large language model (LLM) agents increasingly rely on reusable *skills* that encode procedural knowledge for specialized tasks. Recent work has begun to optimize such skills automatically from execution feedback, but existing methods primarily focus on *how to revise an individual skill*, while the outer search over alternative refinement paths remains comparatively underexplored. Early revision choices can therefore narrow subsequent exploration, and simply spending more optimization computation along the same trajectory may not recover alternative directions. We introduce `EvoBranch`, a plug-in evolution framework that separates the native skill-revision mechanism from the search strategy around it. `EvoBranch` reuses an existing skill optimizer without modifying its revision procedure, while broadening exploration through two complementary mechanisms. Branch-local optimization uses execution trajectories and branch-specific history to produce multiple targeted revisions from each parent skill, while global evolution generates additional candidates across branches through mutation, crossover, and regeneration. Validation-based Top-M beam selection then determines which refinement paths receive further optimization. We evaluate `EvoBranch` with 3 skill optimizers, 3 LLM backbones, and 4 benchmarks. `EvoBranch` improves the average performance of all nine model-optimizer combinations, with gains of up to 51.4 percentage points .Compute-controlled comparisons further show that longer single trajectories and independent restarts do not reproduce these gains.Anonymous code is available at: `https://anonymous.4open.science/r/EvoBranch-for-skill-optimization-DA3F/`

1 INTRODUCTION

Large language model (LLM) agents increasingly rely on reusable *skills*, which are structured instructions and executable scripts that guide agents through specialized tasks Xu & Yan (2026). A coding skill, for example, may instruct an agent how to inspect a repository, modify relevant files, and validate the resulting patch. Because a skill can be reused across many future tasks, improving its effectiveness offers a scalable way to enhance agent behavior without updating the underlying model Ma et al. (2026). This observation has motivated growing interest in *skill optimization* Yang et al. (2026), where an optimizer automatically revises an initial skill based on feedback from downstream executions Ni et al. (2026); Alzubi et al. (2026); Liu et al. (2026).

Existing skill-optimization methods have developed increasingly effective mechanisms for proposing revisions Ding et al. (2026). They may use execution feedback Yang et al. (2026); Li et al. (2026a), model-generated critiques Peng et al. (2026), or successful trajectories to identify weaknesses in the current skill and produce an improved version Ma et al. (2026). However, most attention has focused on *how to revise a given skill*, while the outer search process over alternative refinement paths remains comparatively underexplored. In particular, an optimizer must also determine which candidate directions to generate, which alternatives to retain, and which revisions should receive further optimization computation. Existing methods often organize this search around a single incumbent, a dominant optimization trajectory, or a limited set of alternatives Yang et al. (2026); Ma et al. (2026); Li et al. (2026a); Peng et al. (2026).

This overlook matters because skill optimization is inherently path dependent. An early revision determines the starting point for later improvements and may prematurely steer the optimizer to-

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

ward a narrow region of the search space Wang et al. (2024). Simply extending one optimization trajectory can therefore reinforce early decisions while leaving qualitatively different skill designs unexplored Guo et al. (2024); Fernando et al. (2023). The final skill depends not only on the underlying revision mechanism, but also on the search strategy that determines which candidates to generate, evaluate, and refine Pryzant et al. (2023); Alzubi et al. (2026). This raises a fundamental question:

> *Can existing skill optimizers be augmented with an outer search process that explores multiple refinement paths and allocates subsequent optimization effort among them?*

Several straightforward strategies can spend more optimization computation without changing the underlying search structure. Extending a single optimization trajectory explores more revisions, but remains conditioned on its earlier choices. Running independent restarts broadens exploration, but does not allow information discovered along one trajectory to guide the subsequent optimization of another. These limitations suggest that additional computation is most useful when it is organized across multiple evolving candidates rather than spent independently or along a single refinement path.

Realizing such an outer search process presents three challenges. First, how can a single parent skill be expanded into multiple meaningful local revision directions while preserving the experience of what has already been observed and attempted? Second, once multiple branches are optimized independently, how can the search move beyond branch-local refinement and exploit information across branches to generate new directions? Third, as the search space expands, how should a limited optimization budget be allocated among competing refinement paths, rather than simply spending more computation?

Our key insight is that *a skill optimizer's revision mechanism and the search strategy around it can be treated as separate design choices*. An existing optimizer can provide the mechanism for revising an individual skill, while an outer search process determines how alternative refinement paths are generated, maintained, and allocated further optimization effort. This separation makes the search over candidate skills an explicit component of skill optimization, while preserving the underlying revision mechanism of the native optimizer.

Based on this insight, we introduce `EvoBranch`, a plug-in evolution framework that wraps existing skill optimizers without changing their native revision mechanisms. `EvoBranch` maintains multiple evolving skill branches and expands the search space at two levels. First, *branch-local optimization* uses execution trajectories and branch-specific history to construct multiple targeted inputs to the native optimizer, allowing a single parent skill to produce multiple local revision directions. Second, *global evolution* operates across branches through mutation, crossover, and regeneration to create additional candidates beyond independent local refinement. Finally, validation-based Top-M beam selection determines which refinement paths receive further optimization in the next iteration.

We evaluate `EvoBranch` across three skill optimizers, three backbone models, and four benchmarks. `EvoBranch` improves the average performance of all nine model–optimizer combinations.Importantly, compute-controlled comparisons show that spending comparable or greater optimization computation on either a longer single trajectory or independent restarts does not reproduce these gains. These results indicate that performance depends not only on the amount of optimization computation, but also on how it is organized across competing refinement paths. Ablation studies further examine the contributions of Branch History, Trajectory Clustering, and Global Evolution.

Our contributions are summarized as follows.

Problem formulation. We separate the mechanism for revising an individual skill from the outer search process over alternative refinement paths, highlighting the allocation of subsequent optimization computation as an underexplored dimension of skill optimization.

Method. We introduce `EvoBranch`, a plug-in evolution framework that reuses existing skill optimizers while broadening search through branch-local optimization and global evolution, followed by validation-based Top-M beam selection.

Empirical evaluation. We evaluate `EvoBranch` across 3 skill optimizers, 3 models, and 4 benchmarks. We further compare against alternative ways of spending optimization computation and ablate the major components of the framework.

2 Background & Related Work

Agent Skills. The emerging Agent Skills format provides a concrete implementation of this idea. Each skill package is organized around a `SKILL.md` file that stores metadata and procedural guidance, with optional scripts or reference resources Agent Skills (2025). Skills are portable and reusable across compatible agent settings, providing a lightweight external layer of procedural knowledge for LLM agents Li et al. (2026b). Variants of the format are already supported by several agent harnesses, including Claude Code and Codex OpenAI; Anthropic (2025).

However, current skill creation has largely relied on manual authoring, which requires time and domain expertise Alzubi et al. (2026). Consequently, recent work has begun to automate the construction and refinement of skills using execution experience and feedback Ding et al. (2026). These efforts have given rise to a growing line of work on *skill optimization*, where skills themselves become explicit targets of iterative improvement.

Skill Optimization. Recent work has increasingly explored automated construction and optimization of agent skills. One line of work, like Trace2Skill Ni et al. (2026)and SkillX Wang et al. (2026), focuses on extracting and constructing skill artifacts from agent trajectories, rather than repeatedly optimizing a single persistent skill artifact. SkillOpt Yang et al. (2026) takes a further step by treating a reusable skill as an independently optimizable external artifact. Starting from the current skill, it generates a candidate revision from execution feedback and accepts the update only when it improves validation performance. Related methods, including SSO Peng et al. (2026), HDSO Shang & Yang (2026), CoEvoSkills Zhang et al. (2026), and SkillGen Ma et al. (2026), differ in how candidates are proposed, but their outer-loop optimization is still largely organized around a single incumbent or a dominant optimization trajectory. EvoSkill Alzubi et al. (2026) maintains a fixed-capacity frontier and expands its members in a round-robin manner. However, each expansion produces a single candidate from one selected parent, while the accumulated feedback history is shared across frontier members rather than maintained as branch-specific optimization state.

In contrast to introducing another skill-revision rule, our focus is the outer search process around existing skill optimizers: maintaining branch-specific optimization histories, expanding each parent into multiple targeted revision directions, evolving candidates across branches, and allocating subsequent optimization effort among competing refinement paths.

Test-time Computing.. A growing body of work studies how additional computation at inference time can improve the performance of a fixed language model. Early approaches such as self-consistency increase test-time computation by sampling multiple reasoning trajectories and aggregating their predictions Wang et al. (2022). More recent work has examined a broader range of inference-time strategies, including best-of-N sampling, verifier-guided search, adaptive revision, and tree search Snell et al. (2025). These studies show that the benefit of additional computation depends not only on the total compute budget, but also on how that budget is allocated across alternative reasoning paths or candidate outputs. In particular, adaptive allocation can concentrate computation on more promising or difficult instances rather than spending it uniformly Snell et al. (2025). We extend test-time computation to reusable skill optimization, allocating additional computation to generate, evaluate, and refine candidate skills rather than improve individual query responses, while keeping model parameters fixed.

3 Approach

3.1 Problem Formulation

We consider skill optimization for an LLM agent $\mathcal{M}$ whose model parameters remain fixed. Given a task x and a reusable skill $\mathcal{S}$, the agent executes x conditioned on $\mathcal{S}$, producing a trajectory $\tau_{\mathcal{M}}(x; \mathcal{S})$

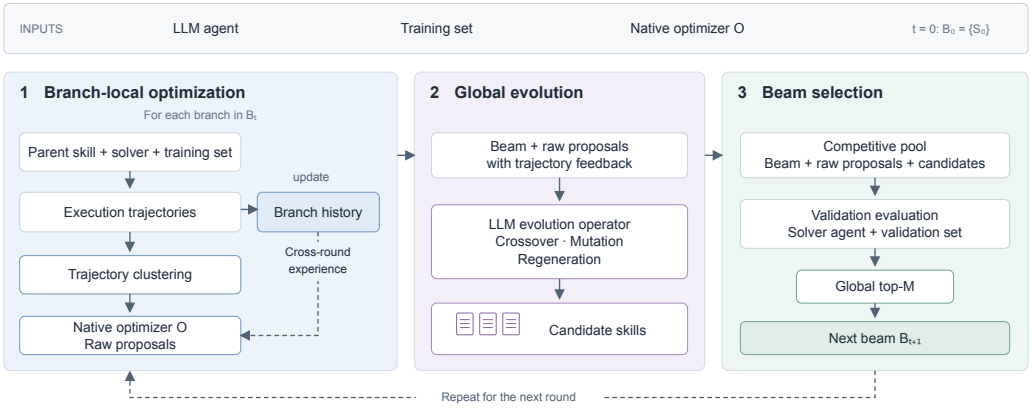


Figure 1: Overview of EvoBranch.

and a task score $r_{\mathcal{M}}(x;\mathcal{S})$. We define skill quality as the average validation score:

$$V(\mathcal{S}) = \frac{1}{|\mathcal{D}_{\text{val}}|} \sum_{x \in \mathcal{D}_{\text{val}}} r_{\mathcal{M}}(x;\mathcal{S}). \tag{1}$$

Starting from a seed skill $\mathcal{S}_0$, a native optimizer $\mathcal{O}$ uses execution feedback from $\mathcal{D}_{\text{train}}$ to generate and refine candidates. Under an optimization budget B, it selects

$$\mathcal{S}^*_{\mathcal{O}} \in \arg \max_{\mathcal{S} \in \mathcal{C}_{\mathcal{O}}(\mathcal{S}_0, \mathcal{D}_{\text{train}}; B)} V(\mathcal{S}), \tag{2}$$

where $\mathcal{C}_{\mathcal{O}}$ denotes the explored candidate set. The validation set guides candidate selection, while a held-out test set is reserved for final evaluation.

This formulation exposes two distinct determinants of optimization quality: *how individual skills are revised* and *which revisions receive further computation*. Existing optimizers often couple sophisticated revision mechanisms with a single trajectory or limited alternatives, allowing early decisions to narrow subsequent exploration. Improving the revision mechanism alone therefore leaves the allocation of optimization computation underexplored. We introduce a general plug-in evolution framework $\mathcal{E}$ that makes this allocation explicit. It reuses $\mathcal{O}$ as a native revision operator while maintaining diverse candidates and directing further optimization across competing trajectories. The resulting optimizer $\mathcal{E}(\mathcal{O})$ returns

$$\mathcal{S}^*_{\mathcal{E}(\mathcal{O})} \in \arg \max_{\mathcal{S} \in \mathcal{C}_{\mathcal{E}(\mathcal{O})}(\mathcal{S}_0, \mathcal{D}_{\text{train}}; B)} V(\mathcal{S}) \quad \bigwedge \quad ||\mathcal{C}_{\mathcal{E}(\mathcal{O})}(\mathcal{S}_0, \mathcal{D}_{\text{train}}; B)|| \gg ||\mathcal{C}_{\mathcal{O}}(\mathcal{S}_0, \mathcal{D}_{\text{train}}; B)|| \tag{3}$$

Our contribution is thus an optimizer-level evolution mechanism that broadens exploration and allocates a finite budget across alternative refinement paths. This design enables existing revision methods to benefit from structured search without changing their underlying mechanisms.

3.2 Design Overview

Figure 1 illustrates the overall workflow of EvoBranch. EvoBranch takes as input a skill optimizer $\mathcal{O}$, a seed skill $\mathcal{S}_0$, and training and validation datasets. At iteration 0, the seed skill $\mathcal{S}_0$ initializes the skill candidate set. In each subsequent iteration, $\mathcal{O}$ optimizes the current candidates, while EvoBranch expands the search space through two complementary mechanisms: (1) branch-local optimization and (2) global evolution. After generating new skill candidates, EvoBranch evaluates them on the validation set and selects the top-M candidates to serve as the starting population for the next iteration.

3.3 Detailed Design

Branch-local Optimization. A native skill optimizer typically explores only a limited number of revisions from the current skill in each iteration. To explore more possible optimization directions,

EvoBranch expands each parent skill into multiple targeted proposals based on its actual execution trajectories. Meanwhile, EvoBranch maintains a branch-specific history so that optimization experience accumulated across iterations can be reused in subsequent revisions.

At iteration t, EvoBranch maintains a beam

$$\mathcal{B}_t = \{b_i^t\}_{i=1}^{m_t}, \qquad b_i^t = (\mathcal{S}_i^t, H_i^t), \qquad m_t \le M, \tag{4}$$

where M is the beam width, $\mathcal{S}_i^t$ is the current skill of branch i, and H_i^t is its Branch History. Specifically, $H_i^t = (\mathcal{I}_i^t, \mathcal{A}_i^t, \rho_i^t)$, where $\mathcal{I}_i^t$ records previously observed issues and their persistence, $\mathcal{A}_i^t$ records previous attempts to address these issues, and ρ_i^t records regeneration events. At iteration 0, the beam contains only the seed skill $\mathcal{S}_0$ with an empty history.

For each branch b_i^t, the LLM agent uses $\mathcal{S}_i^t$ to execute training tasks selected according to the native optimizer's original protocol. The resulting execution trajectories and associated feedback are denoted by $\mathcal{T}_i^t$. Given $(\mathcal{S}_i^t, H_i^t, \mathcal{T}_i^t)$, branch-local optimization produces two outputs: an updated Branch History $\widetilde{H}_i^t$ and a set of raw proposals $\mathcal{R}_i^t$.

We first update the Branch History using the newly observed trajectories and feedback. Newly observed issues are added to the history, while previously observed but unresolved issues are marked as recurring. Previous repair attempts are also retained, allowing later iterations to distinguish a new optimization direction from one that has already been repeatedly attempted.

Next, EvoBranch clusters the trajectories to separate different revision directions. If recurring unresolved issues are observed, their associated trajectories are placed into one dedicated cluster, while the remaining trajectories are grouped by K-means into the other $K-1$ clusters. If no recurring unresolved issue is observed, all trajectories are directly grouped into K clusters by K-means.

Each cluster is then used to generate one revision of the same parent skill. For cluster $G_{i,k}^t$, EvoBranch provides the parent skill $\mathcal{S}_i^t$, the trajectories in $G_{i,k}^t$, and the relevant part of the updated history $\widetilde{H}_{i,k}^t$ to the native optimizer:

$$R_{i,k}^t = \mathcal{O}\big(\mathcal{S}_i^t, G_{i,k}^t; \widetilde{H}_{i,k}^t\big). \tag{5}$$

The native optimizer follows its original revision procedure without modification. Applying it independently to all trajectory clusters therefore allows one parent skill to generate multiple targeted raw proposals. We collect the proposals from branch i and from the entire beam as

$$\mathcal{R}_i^t = \{R_{i,k}^t\}_{k=1}^{K}, \qquad \mathcal{R}_t = \bigcup_{i=1}^{m_t} \mathcal{R}_i^t. \tag{6}$$

Global Evolution. Branch-local optimization expands the neighborhood of each parent skill, but its raw proposals are still generated independently within individual branches. They therefore cannot directly combine complementary directions discovered by different branches or deliberately leave a refinement path that has repeatedly failed. EvoBranch addresses this limitation through global evolution over the current search state.

After branch-local optimization, we denote the current branches with their updated histories by $\widetilde{\mathcal{B}}_t = \{(\mathcal{S}_i^t, \widetilde{H}_i^t)\}_{i=1}^{m_t}$, and collect all execution trajectories as $\mathcal{T}_t = \bigcup_{i=1}^{m_t} \mathcal{T}_i^t$. We further deduplicate the raw proposals in Eq. 6 to obtain $\overline{\mathcal{R}}_t$. Global evolution therefore takes $(\widetilde{\mathcal{B}}_t, \mathcal{T}_t, \overline{\mathcal{R}}_t)$ as input and outputs a set of evolution candidates $\mathcal{C}_t$.

An LLM-based evolution operator jointly considers these inputs and generates new candidates through three complementary operations. **Mutation** further revises an existing direction by targeting an unresolved issue, with priority given to issues that have received fewer previous repair attempts. **Crossover** combines complementary revision directions discovered across different branches, allowing useful parts of different proposals to be integrated into a new skill. **Regeneration** starts from a new optimization direction when local revision has repeatedly failed; it is prioritized when a recurring unresolved issue has persisted for at least two iterations and has already received at least two repair attempts.

To keep the additional search budget bounded, EvoBranch generates at most three evolution candidates in each iteration: A deterministic routing rule decides which of the three operations occupy

the available candidate slots, with crossover using at most one slot. A single LLM call then jointly generates the requested candidates. The operator does not observe validation scores during candidate generation.

Each generated candidate also carries forward the optimization experience from which it originates. Mutation and regeneration inherit the history of their source branch and append the current operation as a new attempt, while regeneration additionally records a restart. Crossover merges the histories of its source branches before recording the crossover operation. We therefore treat each element of $\mathcal{C}_t$ as a candidate branch containing both a generated skill and its corresponding history. Raw proposals $\overline{\mathcal{R}}_t$ serve only as intermediate inputs to global evolution and are not directly inserted into the next beam.

Beam Selection. After global evolution, `EvoBranch` uses the frozen validation set to determine which refinement paths receive further optimization. The competitive pool consists of the current parent branches and the evolution candidates:

$$\mathcal{P}_t = \widetilde{\mathcal{B}}_t \cup \mathcal{C}_t, \qquad \mathcal{B}_{t+1} = \mathrm{TopM}_{b=(\mathcal{S},H)\in\mathcal{P}_t} V(\mathcal{S}). \tag{7}$$

New candidate skills are evaluated by the fixed LLM agent on the validation set, while parent skills reuse their cached validation scores. The top-M branches form the beam for the next iteration, together with their corresponding histories. In this way, subsequent optimization computation is allocated to promising refinement paths while multiple alternatives remain active.

After the final iteration, `EvoBranch` generates one additional crossover-style candidate from the surviving beam. This candidate competes with the surviving skills under the same validation-based ranking rule, and the best skill is selected for final deployment. The held-out test set is never used for candidate generation or selection.

4 Evaluation

4.1 Evaluation Setup

Datasets. We evaluate `EvoBranch` on four benchmarks spanning research-level mathematical reasoning, programming, biomedical question answering, and chemistry. **LiveMathematicianBench** He et al. evaluates research-level mathematical reasoning through dynamically constructed multiple-choice questions derived from recent arXiv papers.**LiveCodeBench** Jain et al. (2025) evaluates coding capabilities on continuously updated programming problems collected from competitive-programming platforms, with an emphasis on reducing benchmark contamination. **PubMedQA** Jin et al. (2019) evaluates biomedical research question answering using questions grounded in PubMed abstracts with yes/no/maybe answers. For **ChemLLMBench** Guo et al. (2023), we use the reaction-yield prediction task, which formulates yield prediction as binary classification of whether a reaction is high-yielding. We report each benchmark's native task metric as a percentage on the held-out test split. Within every matched comparison, Native and EvoBranch use the same training, validation, and test partitions; validation data are used for skill selection, while the test split is reserved for final reporting.

Skill Optimizers Under Evolution. We evaluate EvoBranch with three representative skill optimizers: **SkillOpt** Yang et al. (2026), **SSO** Peng et al. (2026), and **SkillGen** Ma et al. (2026). SkillOpt treats a reusable skill document as the trainable external state of a frozen agent. It converts scored execution trajectories into bounded textual updates and accepts a candidate only when it improves performance on a held-out validation split. SSO instead performs self-supervised skill optimization without ground-truth feedback. It compares executions using an LLM judge, aggregates behavioral evidence across tasks, and renders a revised skill whose update is gated by an SkillGen derives reusable behaviors from successful and failed trajectories and iteratively generates, verifies, and refines candidate skills according to their empirical effect on agent performance. For all three optimizers, we preserve the native skill-revision mechanism and use it as the inner optimizer within EvoBranch. We compare the original *Native* optimizer against *+EvoBranch* under the same model, benchmark, and data split, so that the comparison isolates the effect of the outer candidate-search process.

Table 1: Performance across models, skill optimizers, and benchmarks.

Model	Optimizer	Method	LiveMath	LiveCode	PubMedQA	ChemLLMBench	Avg.
DeepSeek-V4.1-Flash	SkillOpt	Native	42.1	**72.0**	**81.3**	**80.4**	69.0
		+ OURS	**47.7** +5.6	68.2 -3.8	**81.3**	**80.4**	**69.4**
	SSO	Native	**27.1**	**64.5**	77.6	32.7	50.5
		+ OURS	**27.1**	63.6 -0.9	**80.4** +2.8	**61.7** +29.0	**58.2**
	SkillGen	Native	30.8	62.6	71.0	30.8	48.8
		+ OURS	**39.3** +8.5	**64.5** +1.9	**76.6** +5.6	**71.0** +40.2	**62.9**
GPT-5.6 Luna	SkillOpt	Native	41.1	80.4	68.2	**80.4**	67.5
		+ OURS	**67.3** +26.2	**81.3** +0.9	**76.6** +8.4	79.4 -1.0	**76.2**
	SSO	Native	29.9	71.0	56.1	29.0	46.5
		+ OURS	**33.6** +3.7	**75.7** +4.7	**78.5** +22.4	**80.4** +51.4	**67.1**
	SkillGen	Native	33.6	**77.6**	74.8	69.2	63.8
		+ OURS	**41.1** +7.5	**77.6**	**75.7** +0.9	**70.1** +0.9	**66.1**
Qwen3.5-4B	SkillOpt	Native	**55.1**	43.0	**73.8**	**74.8**	61.7
		+ OURS	**55.1**	**45.8** +2.8	**73.8**	**74.8**	**62.4**
	SSO	Native	26.2	42.1	67.3	32.7	42.1
		+ OURS	**34.6** +8.4	**45.8** +3.7	**73.8** +6.5	**68.2** +35.5	**55.6**
	SkillGen	Native	21.5	43.0	**75.7**	77.6	54.5
		+ OURS	**38.3** +16.8	**46.7** +3.7	**75.7**	**79.4** +1.8	**60.0**

Large Language Models. We evaluate three backbones: **DeepSeek-V4.1-Flash**, **GPT-5.6 Luna**, and **Qwen3.5-4B**. The backbones is the task-execution model whose behavior is adapted through the skill; its model parameters remain fixed throughout skill optimization. To control for optimizer-side model variation, we use **DeepSeek-V4.1-Flash** for all optimizer-side LLM calls across SkillOpt, SSO, and SkillGen.

Implementation and Process. All methods share one direct-chat harness and benchmark scorer; the backbone decodes greedily. Each benchmark is frozen to 177 tasks split $35/35/107$ into train, validation, and test with a fixed seed; the test split is never used during optimization or selection. Each optimizer keeps its native S_0: SkillOpt starts from its initial skill document, SSO and SkillGen from no skill. Native optimizers use their papers' defaults , and +`EvoBranch` inherits the same per-step settings, replacing only the outer loop. po`EvoBranch` uses $T = 4$ rounds, beam width $M = 3$, and $K = 3$ evidence subsets per branch. Candidates are ranked on the validation split, and after the final round the top-1 candidate by validation score is deployed on the test split. Optimization cost counts every optimizer-side call, including clustering, history tracking, the operator, and validation scoring, and is reported separately from deployment cost.

4.2 Main Results

Overall Performance. Table 1 compares each native optimizer with its `EvoBranch`-augmented counterpart across three models and four benchmarks. `EvoBranch` improves the average performance of all nine model-optimizer combinations. Among the 36 individual settings, it improves 25, matches 8, and decreases performance in only 3. The gains are particularly pronounced for SSO and SkillGen: with GPT-5.6 Luna, the average performance of SSO increases from 46.5 to 67.1; with DeepSeek-V4.1-Flash, SkillGen improves from 48.8 to 62.9; and with Qwen3.5-4B, SSO improves from 42.1 to 55.6. These improvements span different native optimizers, backbone models, and task domains, suggesting that the outer search process can complement heterogeneous skill-revision mechanisms rather than relying on a particular optimizer design.

Deployment Cost. Table 2 reports the token cost of deploying the final selected skill on the held-out test set. Deployment cost varies across model–optimizer–benchmark combinations, reflecting the behavior of the final selected skill rather than the breadth of the optimization search. This separates the one-time optimization overhead of `EvoBranch` from its downstream deployment cost. Detailed optimization-time token usage and LLM-call counts are reported in the Appendix.

4.3 Other Evaluation

Is More Compute Alone Sufficient. Table 3 compares `EvoBranch` with two straightforward ways of spending additional optimization compute on SkillOpt: extending a single optimization trajectory and running independent restarts.

Table 2: Deployment token usage across models, skill optimizers, and benchmarks. Token counts are reported in millions (M).

Model	Optimizer	Method	LiveMath	LiveCode	PubMedQA	ChemLLMBench
DeepSeek-V4.1-Flash	SkillOpt	Native	0.23M	0.46M	0.28M	0.30M
		+ OURS	0.49M	0.30M	0.14M	0.17M
	SSO	Native	0.17M	0.22M	0.11M	0.11M
		+ OURS	0.17M	0.31M	0.11M	0.18M
	SkillGen	Native	0.15M	0.20M	0.12M	0.13M
		+ OURS	0.37M	0.36M	0.15M	0.20M
GPT-5.6 Luna	SkillOpt	Native	0.40M	0.30M	0.14M	0.21M
		+ OURS	0.45M	0.57M	0.36M	0.16M
	SSO	Native	0.23M	0.36M	0.11M	0.14M
		+ OURS	0.38M	0.40M	0.16M	0.34M
	SkillGen	Native	0.25M	0.29M	0.18M	0.20M
		+ OURS	0.33M	0.39M	0.22M	0.28M
Qwen3.5-4B	SkillOpt	Native	0.83M	1.09M	0.11M	0.26M
		+ OURS	0.77M	0.96M	0.22M	0.26M
	SSO	Native	0.59M	0.88M	0.10M	0.26M
		+ OURS	0.87M	0.88M	0.13M	0.11M
	SkillGen	Native	0.63M	0.88M	0.13M	0.33M
		+ OURS	0.79M	0.63M	0.22M	0.18M

Table 3: Compute-controlled comparison of alternative optimization-time search strategies.

Search Strategy	TEST (%)	VAL	Opt. Calls	Opt. Tokens (M)
Full OURS	**67.3**	**28/35**	1047	5.36
Native SkillOpt	41.1	18/35	478	2.04
Longer Single Trajectory	43.9	20/35	963	6.22
Independent Restarts	41.1	18/35	1094	4.45

The longer single trajectory consumes 6.22M tokens, exceeding the 5.36M tokens used by `EvoBranch`, but improves test performance only from 41.1 to 43.9, compared with 67.3 for `EvoBranch`. Similarly, independent restarts use 1,094 optimization calls, slightly more than the 1,047 calls of `EvoBranch`, yet obtain the same 41.1 test performance as native SkillOpt. The validation results show the same pattern, with `EvoBranch` reaching 28/35 compared with 20/35 for the longer trajectory and 18/35 for independent restarts. Thus, the gain cannot be explained by additional optimization budget alone. How computation is allocated across competing and evolving refinement paths is critical to making the extra compute effective.

Effect of Search Breadth. Table 4 studies the effect of beam width M. A single-branch search ($M = 1$) achieves 45.8% test performance, whereas the default $M = 3$ reaches 67.3%, and increasing the beam to $M = 4$ further improves performance to 72.9%. This supports the motivation for maintaining multiple candidate trajectories rather than committing to a single refinement path. The trend is not strictly monotonic—$M = 2$ obtains 43.0%—indicating that simply retaining more candidates does not guarantee improvement; which candidates survive and receive subsequent optimization also matters. Meanwhile, broader search increases cost from 762 calls and 3.81M tokens at $M = 1$ to 1,339 calls and 7.43M tokens at $M = 4$. We therefore use $M = 3$ in the main experiments as a balance between search breadth, performance, and optimization cost.

Table 4: Effect of beam width M on final performance

Beam Width M	TEST (%)	Opt. Calls	Opt. Tokens (M)
1	45.8	762	3.81
2	43.0	852	3.73
3	67.3	1047	5.36
4	**72.9**	1339	7.43

4.4 Ablation Study

Table 5 ablates the three main components of EvoBranch and shows that each contributes meaningfully to the final performance. Removing any component reduces the average score from 72.0 achieved by the full method, indicating that the gains of EvoBranch do not stem from a single dominant mechanism.

Table 5: Ablation study of the main components of EvoBranch for GPT-5.6 Luna. Avg. denotes the average score across LiveMath and PubMedQA.

Method	LiveMath	PubMedQA	Avg.
Native SkillOpt	41.1	68.2	54.7
Full OURS	**67.3**	**76.6**	**72.0**
w/o Branch History	45.8	72.9	59.4
w/o Clustering	62.6	70.1	66.4
w/o Global Evolution Operator	63.6	75.7	69.7

Branch History has the largest effect: removing it reduces the average score to 59.4 and LiveMath performance from 67.3 to 45.8. This substantial degradation highlights the importance of retaining cross-round experience, which allows the optimizer to identify persistent failure modes, avoid repeatedly revisiting ineffective directions, and steer subsequent search toward refinement paths that remain underexplored.

Removing *Trajectory Clustering* lowers the average score to 66.4. This result is consistent with the benefit of separating heterogeneous trajectories into more coherent groups before revision. By organizing distinct failure patterns and solution behaviors into targeted refinement directions, trajectory clustering provides cleaner optimization signals than directly aggregating mixed evidence.

Finally, removing the *Global Evolution Operator* reduces the average score to 69.7. Although this drop is smaller than those of the other two components, the degradation is consistent across both benchmarks, supporting the role of global mutation, crossover, and regeneration in introducing candidate-level variation beyond branch-local revisions. This broader exploration helps prevent the search from becoming overly concentrated around a small set of local refinement trajectories.

Together, these ablations show that the three components are complementary: *Branch History* preserves accumulated optimization experience, *Trajectory Clustering* structures that experience into targeted local refinement directions, and the *Global Evolution Operator* expands exploration beyond those local branches. Their combination enables EvoBranch to balance experience reuse, structured refinement, and global diversity throughout the optimization process.

5 Conclusion

We introduce EvoBranch, a plug-in evolution framework that decouples native skill revision from the outer search over alternative refinement trajectories. Instead of committing to a single optimization path, EvoBranch explicitly maintains and evolves multiple promising skill candidates, enabling broader exploration of the skill space. Its key design combines branch-local optimization, global evolution, and validation-based Top-M selection to preserve strong intermediate solutions while dynamically reallocating optimization effort toward promising directions. This formulation turns skill optimization from a largely sequential refinement process into a structured population-based search that can better avoid premature convergence to suboptimal skills. Across multiple native optimizers, benchmarks, and LLM backbones, EvoBranch consistently improves average performance over the underlying optimization procedures. Importantly, compute-controlled experiments show that these improvements persist even under comparable optimization budgets, indicating that the gains arise from the proposed search strategy rather than simply from using more computation. Overall, our results demonstrate that structuring how optimization-time computation explores and selects reusable skills is a critical dimension for improving skill optimization.

STATEMENT

AI USE STATEMENT

Generative AI tools, including ChatGPT and Claude, were used throughout this project to assist with literature exploration and summarization, implementation and debugging, experimental workflow development, and drafting and editing portions of the manuscript. AI-assisted code was inspected and tested through the experimental pipeline. All reported experimental results, claims, analyses, and citations were reviewed and verified by the authors. The authors take full responsibility for the final content of this work, including all text, code, claims, and artifacts produced with the assistance of generative AI.

ETHICS STATEMENT

This work studies optimization methods for reusable LLM-agent skills and does not involve human subjects, personally identifiable information, or the collection of sensitive user data. Our experiments use existing benchmark datasets and fixed language models for research evaluation. The proposed method is intended as a general research framework for improving reusable agent skills; as with other techniques that improve agent capabilities, downstream deployment may require application-specific safeguards and evaluation. We are not aware of additional ethical concerns specific to the experiments conducted in this work.

REPRODUCIBILITY STATEMENT

We provide the methodological details of `EvoBranch` in Section 3 and the experimental setup, data splits, optimization settings, and evaluation protocol in Section 4. Additional experimental results, including optimization-time LLM-call statistics, are reported in the Appendix. We also provide an anonymous repository containing the implementation and supporting experimental materials at `https://anonymous.4open.science/r/EvoBranch-for-skill-optimization-DA3F/`.

REFERENCES

Agent Skills. Agent skills specification. `https://agentskills.io/specification`, 2025. Accessed: 2026-09-25.

Salaheddin Alzubi, Noah Provenzano, Jaydon Bingham, Weiyuan Chen, and Tu Vu. Evoskill: Automated skill discovery for multi-agent systems. *arXiv preprint arXiv:2603.02766*, 2026.

Anthropic. Anthropic skills documentation. `https://platform.claude.com/docs/en/agents-and-tools/agent-skills/overview`, 2025. Accessed: 2026-09-25.

Kexin Ding, Yang Zhou, Can Jin, Feng Tong, Mu Zhou, and Dimitris N. Metaxas. Agent skill evaluation and evolution: Frameworks and benchmarks, 2026. URL `https://arxiv.org/abs/2606.11435`.

Chrisantha Fernando, Dylan Banarse, Henryk Michalewski, Simon Osindero, and Tim Rocktäschel. Promptbreeder: Self-referential self-improvement via prompt evolution. *arXiv preprint arXiv:2309.16797*, 2023.

Qingyan Guo, Rui Wang, Junliang Guo, Bei Li, Kaitao Song, Xu Tan, Guoqing Liu, Jiang Bian, and Yujiu Yang. Connecting large language models with evolutionary algorithms yields powerful prompt optimizers. In *International Conference on Learning Representations*, volume 2024, pp. 34133–34156, 2024.

Taicheng Guo, Bozhao Nan, Zhenwen Liang, Zhichun Guo, Nitesh Chawla, Olaf Wiest, Xiangliang Zhang, et al. What can large language models do in chemistry? a comprehensive benchmark on eight tasks. *Advances in neural information processing systems*, 36:59662–59688, 2023.

Linyang He, Qiyao Yu, Hanze Dong, Baohao Liao, Xinxing Xu, Micah Goldblum, Jiang Bian, and Nima Mesgarani. Livemathematicianbench: A live benchmark for research-level mathematical reasoning with proof sketches. In *Third Conference on Language Modeling*.

Naman Jain, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. Livecodebench: Holistic and contamination free evaluation of large language models for code. In *International Conference on Learning Representations*, volume 2025, pp. 58791–58831, 2025.

Qiao Jin, Bhuwan Dhingra, Zhengping Liu, William Cohen, and Xinghua Lu. Pubmedqa: A dataset for biomedical research question answering. In *Proceedings of the 2019 conference on empirical methods in natural language processing and the 9th international joint conference on natural language processing (EMNLP-IJCNLP)*, pp. 2567–2577, 2019.

Gaoyuan Li, Meihao Fan, Yizhe Liu, Shaolei Zhang, Ju Fan, Siyi Wang, Jiaheng Hou, Xudong Weng, Honghan Tian, and Zang Li. Skilladam: Stable and efficient skill evolution for agents. *arXiv preprint arXiv:2609.08944*, 2026a.

Xiangyi Li, Yimin Liu, Wenbo Chen, Bingran You, Zonglin Di, Yifeng He, Shenghan Zheng, Kyoung Whan Choe, Jiankai Sun, Shuyi Wang, et al. Skillsbench: Benchmarking how well agent skills work across diverse tasks. *arXiv preprint arXiv:2602.12670*, 2026b.

Xingyan Liu, Xiyue Luo, Linyu Li, Ganghong Huang, Jianfeng Liu, and Honglin Qiao. Skillforge: Forging domain-specific, self-evolving agent skills in cloud technical support. *arXiv preprint arXiv:2604.08618*, 2026.

Yuchen Ma, Yue Huang, Han Bao, Haomin Zhuang, Swadheen Shukla, Michel Galley, Xiangliang Zhang, and Stefan Feuerriegel. Skillgen: Verified inference-time agent skill synthesis. *arXiv preprint arXiv:2605.10999*, 2026.

Jingwei Ni, Yihao Liu, Xinpeng Liu, Yutao Sun, Mengyu Zhou, Pengyu Cheng, Dexin Wang, Erchao Zhao, Xiaoxi Jiang, and Guanjun Jiang. Trace2skill: Distill trajectory-local lessons into transferable agent skills. *arXiv preprint arXiv:2603.25158*, 2026.

OpenAI. Build skills. `https://learn.chatgpt.com/docs/build-skills`. Accessed: 2026-09-25.

Siran Peng, Cuiyu Yang, Tianyu Fu, Tianshuo Zhang, Haoyuan Zhang, Weisong Zhao, Anyang Su, Minghui Wu, Huiying Li, Xiangyu Zhu, et al. Self-supervised skill optimization. *arXiv preprint arXiv:2607.28777*, 2026.

Reid Pryzant, Dan Iter, Jerry Li, Yin Lee, Chenguang Zhu, and Michael Zeng. Automatic prompt optimization with "gradient descent" and beam search. In *Proceedings of the 2023 conference on empirical methods in natural language processing*, pp. 7957–7968, 2023.

Fangxin Shang and Yehui Yang. Hypothesis-driven skill optimization for llm agents. *arXiv preprint arXiv:2606.22330*, 2026.

Charlie Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. Scaling llm test-time compute optimally can be more effective than scaling parameters for reasoning. In *International Conference on Learning Representations*, volume 2025, pp. 10131–10165, 2025.

Chenxi Wang, Zhuoyun Yu, Xin Xie, Wuguannan Yao, Runnan Fang, Shuofei Qiao, Kexin Cao, Guozhou Zheng, Xiang Qi, Peng Zhang, et al. Skillx: Automatically constructing skill knowledge bases for agents. *arXiv preprint arXiv:2604.04804*, 2026.

Xinyuan Wang, Chenxi Li, Zhen Wang, Fan Bai, Haotian Luo, Jiayou Zhang, Nebojsa Jojic, Eric Xing, and Zhiting Hu. Promptagent: Strategic planning with language models enables expert-level prompt optimization. In *International Conference on Learning Representations*, volume 2024, pp. 23967–24001, 2024.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. *arXiv preprint arXiv:2203.11171*, 2022.

Renjun Xu and Yang Yan. Agent skills for large language models: Architecture, acquisition, security, and the path forward. *arXiv preprint arXiv:2602.12430*, 2026.

Yifan Yang, Ziyang Gong, Weiquan Huang, Qihao Yang, Ziwei Zhou, Zisu Huang, Yan Li, Xuemei Gao, Qi Dai, Bei Liu, et al. Skillopt: Executive strategy for self-evolving agent skills. *arXiv preprint arXiv:2605.23904*, 2026.

Hanrong Zhang, Shicheng Fan, Henry Peng Zou, Yankai Chen, Zhenting Wang, Jiayu Zhou, Chengze Li, Wei-Chieh Huang, Yifei Yao, Kening Zheng, et al. Coevoskills: Self-evolving agent skills via co-evolutionary verification. *arXiv preprint arXiv:2604.01687*, 2026.

A Appendix

A Optimization cost

Table 6: Optimization token usage across models, skill optimizers, and benchmarks. Token counts are reported in millions (M).

Model	Optimizer	Method	LiveMath	LiveCode	PubMedQA	ChemLLMBench
DeepSeek-V4.1-Flash	SkillOpt	Native	1.84M	2.51M	1.17M	1.30M
		+ OURS	4.58M	4.84M	3.49M	3.04M
	SSO	Native	2.20M	3.10M	1.10M	1.10M
		+ OURS	5.16M	5.80M	2.41M	2.86M
	SkillGen	Native	3.08M	2.40M	0.18M	0.25M
		+ OURS	5.23M	4.93M	2.63M	4.49M
GPT-5.6 Luna	SkillOpt	Native	2.04M	2.06M	0.93M	1.34M
		+ OURS	5.36M	6.78M	3.51M	3.70M
	SSO	Native	1.72M	2.61M	0.89M	1.03M
		+ OURS	4.38M	6.02M	2.49M	2.74M
	SkillGen	Native	0.37M	1.87M	0.30M	0.44M
		+ OURS	4.99M	2.47M	2.86M	4.63M
Qwen3.5-4B	SkillOpt	Native	5.03M	9.98M	1.03M	1.96M
		+ OURS	10.44M	15.92M	2.97M	4.51M
	SSO	Native	6.43M	11.56M	1.07M	2.51M
		+ OURS	14.20M	21.01M	2.65M	4.62M
	SkillGen	Native	6.34M	9.42M	0.18M	1.39M
		+ OURS	8.56M	7.27M	3.09M	2.80M

Tables 6 and 7 report the optimization-phase cost of every cell in Table 1. They are taken from the same run that produced each accuracy number. Optimization cost counts every LLM call made before deployment. That includes the optimizer-side calls (native proposal, the OURS issue tracker and evolution operator, SkillOpt's slow update and meta skill), the solver rollouts on TRAIN, and every VAL evaluation used for selection. Token counts are the provider-reported usage. The final TEST evaluation is not included; it is accounted separately as deployment cost.

Over all 36 (solver, optimizer, benchmark) cells, OURS uses $2.19\times$ the LLM calls (40,055 vs. 18,264 in total) and $2.13\times$ the tokens (197.4M vs. 92.7M) of the native optimizers. The per-cell median ratios are $2.04\times$ calls and $2.41\times$ tokens. By optimizer, the mean per-cell ratio is $2.18\times$ calls and $2.59\times$ tokens for SkillOpt, and $1.76\times$ calls and $2.30\times$ tokens for SSO. For SkillGen it is $4.53\times$ calls and $7.71\times$ tokens. The SkillGen ratio is large because native SkillGen often terminates early. For example, native SkillGen on DeepSeek/PubMedQA makes only 141 calls. OURS runs a fixed $T{=}4$, $M{=}3$, $K{=}3$ search, so its call count is nearly independent of the solver: on average 1,130 (DeepSeek), 1,103 (Luna), and 1,104 (Qwen) calls per cell. Its token count is not solver-independent. It doubles on Qwen (8.2M vs. 4.1–4.2M tokens per cell) because Qwen produces much longer rollouts on TRAIN and VAL. OURS is cheaper than the native optimizer in one cell (SkillGen, Qwen, LiveCode: $0.77\times$ tokens). Appendix B.3 gives a compute-matched comparison.

Table 7: Optimization-time LLM calls across models, skill optimizers, and benchmarks.

Model	Optimizer	Method	LiveMath	LiveCode	PubMedQA	ChemLLMBench
DeepSeek-V4.1-Flash	SkillOpt	Native	404	476	476	475
		+ OURS	995	998	1029	958
	SSO	Native	703	728	696	692
		+ OURS	1120	1229	1247	1237
	SkillGen	Native	651	523	141	162
		+ OURS	1294	1017	1129	1312
GPT-5.6 Luna	SkillOpt	Native	478	475	436	474
		+ OURS	1047	1077	1067	1000
	SSO	Native	695	716	584	689
		+ OURS	1177	1264	1201	1155
	SkillGen	Native	166	478	183	233
		+ OURS	1268	599	1134	1250
Qwen3.5-4B	SkillOpt	Native	475	477	475	441
		+ OURS	985	934	1036	988
	SSO	Native	705	731	696	712
		+ OURS	1208	1260	1248	1295
	SkillGen	Native	663	613	138	404
		+ OURS	1267	941	1113	976

B ADDITIONAL EXPERIMENTS

All experiments in this section use the same frozen splits (TRAIN/VAL/TEST $= 35/35/107$), the same solver contract, and the same optimizer LLM as the main experiment. Unless stated otherwise, they use SkillOpt as the native optimizer and GPT-5.6 Luna as the solver. TEST accuracy is in %, and VAL is the number of correct VAL tasks out of 35.

B.1 BEAM WIDTH

We vary only the beam width $M \in \{1, 2, 3, 4\}$ and keep every other setting of the full method fixed (T=4, K=3, seed 42, same operator, tracker and deployment rule). M=1 is a single-incumbent loop: each round, the operator's offspring challenge one surviving skill. Larger M also means more raw proposals per round ($M \times K$) and more VAL evaluations, so cost grows with M. Table 8 reports the best VAL score in the beam after each round, together with the deployed skill's VAL and TEST scores and the optimization cost. The M=3 column is the run reported in Table 1.

Table 8: Beam-width sensitivity (SkillOpt + OURS, Luna, LiveMath). Rows 1–4: best VAL score (/35) in the beam after each round. M=3 is the run in Table 1; a second run with the same configuration reached VAL 17/17/21/21 and TEST 50.5 (1,035 calls, 5.14M tokens).

Round	M=1	M=2	M=3	M=4
1	14	21	18	16
2	17	21	18	17
3	19	21	26	22
4	19	21	28	28
Deployed skill VAL (/35)	19	21	28	28
TEST acc. (%)	45.8	43.0	67.3	72.9
Optimization LLM calls	762	852	1,047	1,339
Optimization tokens	3.81M	3.73M	5.36M	7.43M

TEST accuracy follows the VAL level that the search reaches. Runs that finish at VAL 19–21 score 43–50 on TEST, and runs that reach VAL 28 score 67–73. With M=1 and M=2 the search never leaves the 19–21 plateau. For M=2, from round 3 onward both survivors are parents carried over from the previous round, so no new offspring beats them. With M=1 the incumbent is replaced

every round, but the replacement is never better than the plateau. $M{=}3$ and $M{=}4$ both escape it, to 26 then 28 and to 22 then 28. With $M{=}4$, new offspring enter the beam in every round. $M{=}4$ costs 28% more calls and 39% more tokens than $M{=}3$. Its TEST gain (72.9 vs. 67.3) is within the run-to-run noise of this cell: two $M{=}3$ runs with identical configuration reached 50.5 and 67.3 (Table 8). Each $M \neq 3$ point is a single run. We therefore read the trend only qualitatively: $M \geq 3$ is needed to leave the plateau, and $M{=}3$ is the cost/quality compromise used in the main experiment.

B.2 Cross-solver transfer of optimized skills

We test whether a skill optimized for one solver helps another. We take the final skills of SkillOpt + OURS optimized with Luna (the Luna cells of Table 1) and deploy them unchanged, without re-optimization, on DeepSeek and Qwen. Each deployment uses the target cell's solver contract, harness, scorer, and TEST split. Each DeepSeek transfer was deployed twice to measure deployment-only noise; each Qwen transfer was deployed once.

Table 9: Transfer of Luna-optimized SkillOpt + OURS skills to other solvers (TEST acc., %). "Own" is SkillOpt + OURS optimized with the target solver itself (Table 1). DeepSeek transfers show the mean of two deployments, with both values in parentheses.

Target solver	Benchmark	Native	OURS (own)	Luna skill → target	Luna skill on Luna
DeepSeek	LiveMath	42.1	47.7	48.6 (50.5 / 46.7)	67.3
DeepSeek	PubMedQA	81.3	81.3	81.8 (81.3 / 82.2)	76.6
DeepSeek	LiveCode	72.0	68.2	63.1 (64.5 / 61.7)	81.3
Qwen	LiveMath	55.1	55.1	61.7	67.3
Qwen	PubMedQA	73.8	73.8	77.6	76.6

Transfer is benchmark dependent. On LiveMath, the Luna skill is about as good on DeepSeek as DeepSeek's own optimized skill (48.6 vs. 47.7), and it improves Qwen from 55.1 to 61.7. On PubMedQA, the transferred skill matches DeepSeek's native and own-optimized skills (81.8 vs. 81.3) and improves Qwen from 73.8 to 77.6. On LiveCode, transfer is negative (63.1 vs. 72.0 native); we did not analyse this case further. The LiveMath gains mostly come from a single benchmark-specific rule in the Luna skill, the "meta-option" rule analysed in Appendix B.5. On Qwen, the transferred skill selects the meta-option on 45 of the 48 TEST items where it is present and correct, compared with 31 for Qwen's own skill. On the other 59 items it is worse (21 vs. 28 correct).

B.3 Compute-matched native baselines

We ask whether native SkillOpt catches up with OURS when it is given the same optimization budget. The budget B is the full optimization cost of the OURS run in Table 1 for this cell: 1,047 calls and 5.36M tokens, including all VAL evaluations. We spend more compute on native SkillOpt in two ways: a longer single trajectory (8 or 10 epochs instead of 4) and independent restarts (seeds 42/43/44, with the final skill chosen by VAL; ties go to the lowest seed, a rule fixed before looking at TEST). Token cost per epoch cannot be fixed in advance because it grows as the skill gets longer. We therefore also truncate every trajectory at the last epoch that fits within B and deploy the skill SkillOpt holds at that point. That skill is byte-identical to what the run would have deployed had it stopped there. Over-budget rows are reported as well.

Within budget, three of the four native strategies stay near the native 4-epoch level (41–42). One strategy, the 10-epoch trajectory stopped at B, reaches 69.2 and is 1.9 points (two TEST items) above OURS. Two over-budget strategies also exceed OURS. So this experiment does *not* show that OURS beats native SkillOpt at equal compute. It shows that native SkillOpt reaches this level only for some ways of spending the budget, and that its outcome varies strongly with how the budget is spent. The three restarts reach 41.1, 45.8 and 70.1 on TEST. When truncated to three epochs, all three hold skills tied at VAL 18/35 whose TEST scores are 41.1, 45.8 and 57.0. With 35 VAL tasks (one task $= 2.9$ points) these skills cannot be told apart.

Table 10: Compute-matched comparison (SkillOpt, Luna, LiveMath). B = optimization cost of OURS. Selection uses VAL only.

Method	TEST	VAL	Calls ($/B$)	Tokens ($/B$)
OURS (T=4, M=3, K=3)	67.3	28	1,047 (1.00)	5.36M (1.00)
Native, 4 epochs (Table 1)	41.1	18	478 (0.46)	2.04M (0.38)
Within budget				
8 epochs, stopped at B (ep. 7; incumbent ep. 5)	42.1	19	846 (0.81)	5.33M (1.00)
10 epochs, stopped at B (ep. 8; incumbent ep. 8)	69.2	28	963 (0.92)	5.32M (0.99)
2 restarts $\times$ 4 epochs (seeds 42, 43)	41.1	18	953 (0.91)	4.06M (0.76)
3 restarts $\times$ 3 epochs (seeds 42–44)	41.1	18	1,094 (1.04)	4.45M (0.83)
Over budget				
8 epochs, full	43.9	20	963 (0.92)	6.22M (1.16)
10 epochs, full	72.9	30	1,194 (1.14)	6.98M (1.30)
3 restarts $\times$ 4 epochs (seeds 42–44)	70.1	26	1,428 (1.36)	6.21M (1.16)

B.4 ABLATION: ALL RUNS AND SEARCH TRAJECTORIES

Table 11 lists every run behind the ablation study (SkillOpt, Luna), including same-configuration repeats. All OURS variants use the matched epoch protocol (Appendix C).

Table 11: All ablation runs, in chronological order (TEST acc. %, deployed skill's VAL score out of 35 in parentheses). Where a cell has several runs, the last one is the run reported in the ablation table.

Variant	LiveMath	PubMedQA
Native SkillOpt	41.1 (18)	68.2 (23)
Full OURS	50.5 (21); 67.3 (28)	75.7 (24); 75.7 (24); 76.6 (25)
w/o History State	45.8 (18)	72.9 (25)
w/o History-aware Partition	62.6 (23)	79.4 (24); 70.1 (25)
w/o Attempt-aware Proposal	56.1 (22)	78.5 (25); 77.6 (24); 76.6 (26); 74.8 (25)
w/o Global Evolution	70.1 (24); 63.6 (24)	76.6 (24); 75.7 (25); 75.7 (25)

On LiveMath, the best VAL score in the beam after rounds 1–4 is 18/18/26/28 for Full OURS (17/17/21/21 in the repeat) and 16/18/18/18 without the history state. Native SkillOpt's VAL score over its four epochs is 15/15/18/18. Without history-aware partition the trajectory is 17/17/17/23, without attempt-aware proposal 14/18/22/22, and without global evolution 18/18/18/24 (18/18/20/24 in the repeat). On PubMedQA, the deployed skills of all variants score 24–26/35 on VAL. We also measured skills with *identical* VAL scores that differ by up to 4.7 TEST points (72.9–77.6), so the PubMedQA differences between variants are within noise.

B.5 BENCHMARK-SPECIFIC EFFECTS

The meta-option in LiveMath. In 48 of the 107 LiveMath TEST items, one option reads "One of the remaining options is correct, but a stronger result can be proven", and in all 48 it is the correct answer. On this benchmark, the rate at which a skill makes the solver pick this option explains most of the accuracy differences between skills (Table 12). The largest LiveMath gain in Table 1 (SkillOpt, Luna: $41.1 \rightarrow 67.3$) comes entirely from these items (10→41 of 48). On the other 59 items the OURS skill is slightly worse (34→31). The SkillGen gain on Luna is instead spread over both groups (+4 each). We report this so that the LiveMath results are read as the optimizer learning a strong regularity of this benchmark from TRAIN evidence, not as a general improvement in mathematical reasoning.

ChemLLMBench label imbalance. 86 of the 107 ChemLLMBench TEST items are labelled "No", so a solver that always answers "No" scores 80.4. Every 80.4 entry in the ChemLLMBench column corresponds to such a constant-"No" behaviour, and no cell exceeds this majority-class

Table 12: LiveMath TEST items answered correctly, split by whether the meta-option is present and correct (48 items) or not (59 items).

Solver	Skill	Meta-option items (/48)	Other items (/59)	TEST acc.
Luna	SkillOpt native	10	34	41.1
Luna	SkillOpt + OURS	41	31	67.3
Luna	SkillGen native	3	33	33.6
Luna	SkillGen + OURS	7	37	41.1
DeepSeek	SkillOpt native	14	31	42.1
DeepSeek	SkillOpt + OURS	28	23	47.7
Qwen	SkillOpt native	33	26	55.1
Qwen	SkillOpt + OURS	31	28	55.1

baseline. Differences in this column, in either direction, should therefore be read as differences in how far a skill moves the solver away from, or towards, the majority answer.

C IMPLEMENTATION DETAILS

C.1 SHARED PROTOCOL

All methods in a cell share the frozen TRAIN/VAL/TEST split ($35/35/107$, split seed 42), the solver, its decoding contract (temperature 0, a fixed per-benchmark/solver output limit), the benchmark harness, and the official scorer. The optimizer-side LLM is DeepSeek-V4.1-Flash for every method, and OURS embeds text with bge-small-en-v1.5. The three native optimizers run with the default hyperparameters of their papers or repositories. SkillOpt uses 4 epochs with batch 40. SSO uses $K{=}3$ probes, $|H|{=}8$, $U{=}4$, and $T{=}4$ steps. SkillGen uses its repository's configuration, with the task pool restricted to TRAIN. Any cost-saving change was applied to native and OURS runs alike. Selection in OURS only ever uses the fixed VAL split. TEST is evaluated once, on the single deployed skill.

C.2 ONE OURS ROUND

Each round t has a beam $\mathcal{B}_t$ of at most M skills. $\mathcal{B}_0$ is the native optimizer's own starting point: SkillOpt's initial skill, and the empty skill for SSO and SkillGen. A round runs seven steps:

1. **Evidence.** Each parent in $\mathcal{B}_t$ is run with the optimizer's *native* evidence collection on TRAIN.
2. **Issue tracking.** An LLM issue tracker updates the parent's per-branch issue table (prompt in Appendix D). If the reply cannot be parsed after one retry, a deterministic fallback groups failures by (task type, error signal).
3. **Partition.** The parent's evidence is split into at most K subsets. Tasks belonging to a recurring, unresolved issue form their own subset. The remaining tasks are clustered with seeded K-means (seed 42, 10 initialisations) on embeddings of the exact text the native optimizer would show its own model for that task.
4. **Native proposal.** For each (parent, subset) pair, the native optimizer makes one proposal step conditioned only on that subset. We also pass it an attempt-aware context listing the relevant issues and the fixes already tried. These raw proposals are only material for the operator: they are never scored and cannot enter the beam.
5. **Evolution operator.** One LLM call receives all parents, all raw proposals, the branch histories, the evidence (never any VAL score), and a routing plan computed by code. It returns up to $\min(3, \#\text{unique raw proposals})$ offspring. The routing plan assigns each slot one of three operators. *Regeneration* goes to a branch holding an issue that has recurred for at least 2 rounds after at least 2 attempts. *Crossover* goes to one slot when at least two sources target different issue sets. *Mutation* gets every remaining slot and targets the least-attempted unresolved issue.

6. **Scoring.** New offspring are scored once on the fixed VAL split. Carried-over parents keep their score.
7. **Selection.** Parents and offspring compete in a single global Top-M that forms $\mathcal{B}_{t+1}$. Exact VAL ties are broken by the SHA-256 hash of the skill text.

After T rounds, the operator merges all survivors into one crossover candidate, which is scored on VAL. Only the Top-1 of {survivors, merged candidate} is deployed.

Table 13: Frozen OURS hyperparameters, identical for all optimizers, solvers and benchmarks.

Rounds T / beam width M / evidence subsets K	4 / 3 / 3
Offspring per round	$\min(3, \#\text{unique raw proposals})$
Regeneration trigger	issue recurring ≥ 2 rounds and ≥ 2 attempts
Operator: max output tokens / prompt budget	16,384 / 400k characters
Issue tracker: prompt budget	60k characters
Parse retries (operator, tracker)	1, then deterministic fallback
Partition	history-aware; K-means, seed 42, n_{init}=10
Tie-break / deployment	skill-text hash / final merge + VAL Top-1

C.3 PLUGGING IN SKILLOPT, SSO AND SKILLGEN

OURS only replaces the *search policy*: the accept/reject gate, the single chain of edits, and the step count. Everything an optimizer uses to *propose* an edit stays native. Each optimizer sits behind an adapter with three calls: `collect_evidence` (native evidence for one parent), `evidence_features` (the per-task text the native optimizer shows its model, used for partitioning), and `propose` (one native proposal step from one parent on one evidence subset). The adapters call the unmodified code of the original repositories; no optimizer source was edited. The search framework never imports an optimizer. A fourth, shared component scores any skill on VAL with the benchmark's official scorer. None of the native gates (SkillOpt's selection gate, SSO's pairwise judge, SkillGen's net-gain verification) takes part in selection. Table 14 summarises the mapping.

Table 14: How each native optimizer is mapped onto the OURS roles.

	SkillOpt	SSO	SkillGen
Start $\mathcal{B}_0$	native initial skill	empty skill	no skill (native start)
Evidence per parent	rollouts on batch t of SkillOpt's own TRAIN data plan	the parent's anchor executions on the native batch X_t (label-free)	round 0: no-skill TRAIN trajectories; later: native verification feedback (repairs, regressions, still-failing cases, case analyses) against the fixed no-skill baseline
Raw proposal	one native reflect-and-edit step with a fresh one-step edit budget	one unmodified SSO step on the subset (probes, pairwise decisions, behaviours, clusters, rendering) with an empty validation batch, so the native gate is vacuous	round 0: native induction + skill generation; later: native `refine_skill` on a copy of the parent
Attempt-aware context	native `step_buffer_context` argument	extra `branch_history` field in the probe generator's input	appended to the native revision guidance
Bypassed	selection gate; LR schedule across rounds	pairwise current-vs-candidate gate	sequential loop, net-gain gate, best-net-gain choice
Epoch-level state	slow update + meta skill per new survivor (matched epoch protocol)	round t uses the native batch X_t (matched protocol)	run-level analysis induced once from the no-skill baseline

Siblings, not chains. The K proposals of a parent are independent siblings: each starts from the parent's own text. SSO's skills are immutable. For SkillGen, whose `refine_skill` edits its input in place, each proposal receives a deep copy of the parent.

Matched epoch protocol for SkillOpt. SkillOpt has two epoch-level proposal mechanisms. The *slow update* writes a protected guidance block by comparing two generations of a skill; the *meta skill* is optimizer-side memory. To keep proposal capability native, OURS round t corresponds to native epoch $t{+}1$. After each round's Top-M, except the last, every *new* survivor receives SkillOpt's own slow update against its anchor parent and its own meta skill. Both use SkillOpt's native code, prompts, sample count and seed rule. Offspring inherit the protected block verbatim: the code

enforces this, and the operator prompt states it. Survivors keep their VAL score, matching SkillOpt's default of accepting the slow update without re-validation.

D PROMPTS

OURS adds two LLM roles on top of the native optimizers: the issue tracker and the evolution-operator family. Both run on the same optimizer-side LLM as the native optimizer. Their prompts are reproduced verbatim below. {...} marks a field filled by code. The native optimizers' own prompts are unchanged and are those of the original papers.

D.1 ISSUE TRACKER

Called once per round per parent, after evidence collection.

System prompt.

```
You are the issue tracker of a skill-optimization search.

A "branch" is one candidate skill being optimized. Each round it is run on some training tasks and you are shown, per task, whether it was solved (when available), the task type, an error signal and a short rendering of what happened. You maintain a SMALL table of the distinct underlying ISSUES this branch keeps running into -- recurring failure modes, not one row per task.

You are given the branch's issue table from the PREVIOUS round and this round's per-task evidence. Return the UPDATED table. Rules:
- If a problem in the new evidence is the SAME underlying issue as one in the prior table, REUSE its issue_id and set status to "recurring"; increment rounds_seen and set last_seen_round to this round.
- If a problem is genuinely NEW, add a row with an EMPTY issue_id (the system assigns a stable id) and status "new".
- If a prior issue no longer appears in this round's evidence, keep its row but set status to "resolved".
- Keep the table small: merge tasks that share one underlying cause into one issue.

Return ONLY a JSON object of this exact shape, nothing else:
{"issues": [{"issue_id": "<id from the prior table, or empty for a new issue>", "description": "<one sentence>", "status": "new|recurring|resolved", "task_ids": ["<task id>", ...], "first_seen_round": <int>, "last_seen_round": <int>, "rounds_seen": <int>}]}
```

User prompt template. Each evidence record is headed by `--- task <id> {"solved": ..., "task_type": ..., "error_signal": ...} ---`. SSO has no `solved` flag, so its records omit it. The record body is the native evidence text, truncated head+tail when the character budget requires it.

```
Round {round_id}, branch {branch_id}.

=== PRIOR ISSUE TABLE (previous round) ===
{prior}

=== THIS ROUND'S PER-TASK EVIDENCE ===
{evidence}
```

D.2 EVOLUTION-OPERATOR FAMILY

One call per round. The final merge uses the same prompt with a single crossover slot over the last beam's survivors.

System prompt.

```
You are the evolution operator family of a skill-optimization search.

A "skill" is a natural-language document prepended to a frozen model's instructions to make it solve a benchmark's tasks better. Several optimizers each proposed edited skills ("raw proposals"), each from ONE subset of the evidence gathered when a parent skill was run on training tasks. You are also shown, per branch, its HISTORY: the issues it has run into, which of them recur, and which fixes were already tried.

```

```
Your job is to produce new offspring skills by applying, per offspring, exactly ONE of three evolution operators. A separate held-out validation decides which candidates survive; you NEVER see those scores.

The three operators:
1. mutation: make a LOCAL change to the parent skill aimed at a persistent, still-unresolved issue. Do not rewrite the whole skill.
2. crossover: COMBINE the complementary, evidence-supported strategies of two or more different branches / raw proposals into one skill; resolve their conflicts.
3. regeneration: when a branch has repeatedly failed to fix an issue, STOP patching its old skill and write a NEW skill from scratch grounded in the accumulated evidence for that issue.

You are given a ROUTING PLAN that assigns each offspring an operator, a source branch and the target issue(s). Follow it: produce exactly one offspring per plan entry, in order, using the assigned operator.

Constraints:
- Every change you keep must be supported by the evidence shown; do not invent task facts.
- Do NOT return a parent skill unchanged; parents are already in the pool.
- The offspring must differ from each other in substance.
- Each offspring is a COMPLETE skill document, not an edit or a diff.

Return ONLY a JSON object of this exact shape, nothing else:
{"candidates": [{"operator_type": "mutation|crossover|regeneration", "target_issue_ids": ["<issue id>", ...], "source_proposal_ids": ["<raw proposal id>", ...], "rationale": "<one or two sentences>", "skill": "<the complete skill document>"}]}
```

User prompt template.

```
Round {round_id}. Produce exactly N = {n_op} offspring, one per ROUTING PLAN entry, in order.

=== PARENTS (already in the pool; do not return unchanged) ===
{parents}

=== RAW PROPOSALS (operator material; each came from ONE evidence subset of its parent) ===
{proposals}

=== BRANCH HISTORIES (issues each branch ran into and fixes already tried) ===
{histories}

=== ROUTING PLAN (which operator produces each offspring, its source branch, its target issues) ===
{routing_plan}

=== EVIDENCE (the native evidence each subset was conditioned on{truncation_note}) ===
{evidence}
```

Each branch history ({histories}) is rendered as follows, and each routing-plan entry as <i>. operator=<type>; source branch: <id>; target issues: <ids>:

```
--- branch <id> (restarts: <n>) ---
issues:
- [<issue id>] status=<new|recurring|resolved>, seen in <n> round(s), tasks: <task ids>
    <one-sentence description>
attempts already tried:
    r<round> [<mutation|crossover|regeneration|raw_proposal>] -> <issue ids>: <summary>
```

Protected-block note. Prepended to the parents section only when a SkillOpt slow-update block is present:

```
PROTECTED BLOCK RULE: a skill may contain a block between <!-- SLOW_UPDATE_START --> and <!-- SLOW_UPDATE_END -->. It is written by the native optimizer's epoch-level update and is PROTECTED: reproduce the source branch's block VERBATIM in every offspring, never edit, merge, shorten or drop its contents, and make all of your changes outside it.
```

Repair note. Appended to the user prompt on the single retry, for both the operator and the tracker:

```
Your previous reply could not be parsed. Return ONLY the JSON object described in the instructions, with no prose before or after it.
```

D.3 ATTEMPT-AWARE CONTEXT FOR THE NATIVE PROPOSER

Passed through the native channel listed in Table 14. The context is empty when the branch has no issue relevant to the subset, so in that case the native proposer receives exactly its original input.

```
This branch has already run into these issues on the related tasks:
- [<issue id>] status=<status>, seen in <n> round(s), tasks: <task ids>
    <one-sentence description>
    already tried [<kind>]: <summary>
Do NOT regenerate a fix that was already tried above for these issues; propose a genuinely
    different change.
```

E FINAL SKILLS: NATIVE VS. OURS

We show the deployed skills of three cells where OURS improves most over the native optimizer, one per optimizer and all with the Luna solver. We do not use ChemLLMBench cells because of its majority-class effect (Appendix B.5). Skills are reproduced verbatim, including SkillOpt's protected slow-update block (`<!-- SLOW_UPDATE_START/END -->`).

In 6 of the 36 native cells of Table 1, the native search never accepts any candidate over its starting point, so it deploys *no* skill: SkillGen on DeepSeek/LiveMath, DeepSeek/LiveCode, Luna/LiveCode, Qwen/LiveMath and Qwen/LiveCode, and SSO on Qwen/LiveMath. OURS improves on 5 of these cells and ties on one. For example, SkillGen on Qwen/LiveMath goes from 21.5 to 38.3. Conversely, in two OURS cells the empty starting skill wins on VAL and is deployed: SSO on DeepSeek/LiveMath and Qwen/ChemLLMBench.

E.1 SKILLOPT, LUNA, LIVEMATH (41.1 → 67.3)

Both skills contain the key idea for this benchmark: take the meta-option seriously. They differ in whether they turn it into a consistent decision rule. The native skill states it only in its slow-update block ("be willing to select it often"). Its main body contains the opposite instruction as well ("stop condition against always-meta overshoot", "choose the meta-option only if no listed option captures the true strength"). With these conflicting instructions, the solver picks the meta-option on only 10 of the 48 items where it is correct. The OURS skill turns the idea into one procedure. It computes the strongest provable claim, lists concrete "weakening markers", makes the meta-option the default whenever it is present, and names the two recurring errors seen during search. The solver then picks it on 41 of 48 items. On the other 59 items the OURS skill is slightly worse (31 vs. 34; Table 12). The search trajectory shows how OURS got there: its best VAL went 18/18/26/28, while the native chain stayed at 15/15/18/18.

Native SkillOpt (TEST 41.1).

```
# Live Mathematical MCQ Heuristics

## Option Comparison
- Compare all options before committing. The correct choice is often the strongest statement
    justified by the question, while nearby distractors are weaker, overstrong, or miss an
    equality case.
- Treat the option "One of the remaining options is correct, but a stronger result can be
    proven" as a positive meta-claim, not a throwaway distractor: it asserts some concrete
    option is true AND that an even stronger result exists. Prefer it when all concrete
    options are weaker than a known stronger result or miss an extra clause (larger exponent
     range, wider parameter range, local/global scope extension, additional qualitative
    property).
- Conversely, choose a concrete option over the meta-option only when that concrete option
    really is the strongest provable statement and no stronger version is justified by the
    hypotheses.
- Track exact quantifiers such as "there exists", "for every", "if and only if", and "exactly
     when".

## Theorem-Level Precision
- Check whether an option weakens the conclusion by dropping a characterization, equality
    clause, or full equivalence.
- Check whether an option overstates the theorem by upgrading regularity, removing scale
    restrictions, or changing an existential statement into a universal one.

- Scan for hierarchical strength relations among concrete options: if one option properly
    contains another as a special case (e.g., valid for all q in a range vs a single q, or
```

```
     with extra clauses vs without), neither alone is strongest --- which often makes the
     catch-all meta-option correct.
- Compare exponent/parameter ranges exactly: an option valid for every q in [p, sp] is
     stronger than one valid only at q=p; i>0 is weaker than i>=0. If such gaps exist,
     suspect the catch-all meta-option.

## Hypotheses
- Verify the hypotheses and domain carefully. Distractors often keep the theorem shape but
     alter the required assumptions.

- Test each candidate against the exact quantifiers: a universal 'for every' statement on a
     property is stronger than the corresponding 'there exists' statement; watch for options
     that silently drop a quantifier or switch existence/universality --- these are usually
     distractors.
- For 'which statement is valid?' or 'what is the strongest statement?' items, compare
     options by logical strength (a statement implying another is stronger). Verify which
     implications are actually available from the given conditions and test counterexamples
     that separate options differing by a single logical step (e.g., 1-dimensional vs
     arbitrary dimension, non-strict vs strict monotonicity).
- Pay close attention to equality cases, extremal conditions, and whether a result applies to
      the full family or only a restricted subfamily.

## Final Answer
- Output the final answer as the single option label only.

## Strength Ordering

**Counter the strongest-concrete bias (meta-first override next to the strength-ordering rule
     ):** A single concrete option can look like the most precise theorem while still
     omitting a terminal clause, endpoint exponent, equality case (e.g. requiring equality
     only for i>0 rather than all i>=0), quantifier gap, or universal/local scope extension.
     If any such gap exists relative to the full strength of the hypotheses, the meta-option
     is correct --- do not pick the near-miss. In particular, an option that adds an
     unnecessary condition (e.g. an extra 'same underlying set' assumption) or proves the
     conclusion only for a restricted index/exponent range is a distractor. If the concrete
     options are all strictly weaker variants of one another and exactly one concrete option
     is the maximal element and it truly matches the full strength of the hypotheses with no
     missing clause, pick that concrete option (stop condition against always-meta overshoot)
     .
- Before comparing individual options, rank them by logical strength along the dimensions the
      question cares about (regularity, integrability, quantifier scope, equality cases). The
      correct answer is normally the strongest statement actually supported by the hypotheses
     .
- If an option claims a stronger result can be proven, check whether another option already
     states that stronger result. Choose the meta-option only if no listed option captures
     the true strength, or if the listed stronger options are invalid.
- Distinguish 'what is the strongest statement' from 'which statement is true': the former
     requires eliminating weaker true statements.
- Do not stop at the first true-sounding statement; identify the strongest theorem supported
     by the hypotheses and verify no listed option captures an even stronger result.

## Boundary and Exponent Precision
- Check global vs local scope: a statement of the form "there exists C with ||...||_{X} <= C"
      is stronger than the corresponding statement only for compactly contained subsets.
- Verify equality-case boundaries (e.g., whether the theorem includes i=0, n=0, or the
     endpoint exponent), since dropping or adding such cases often makes a concrete option
     weaker or stronger than the true theorem.

<!-- SLOW_UPDATE_START -->
## Meta-Option Priority (READ FIRST)

Across a large class of advanced 'strongest statement' problems, the correct answer is the
     catch-all option 'One of the remaining options is correct, but a stronger result can be
     proven.' When this meta-option is among the choices, you must test it BEFORE settling on
      any concrete option and you should be willing to select it often. The default failure
     mode is choosing a plausible, literally-true concrete theorem that is merely a weaker or
      special case of what the hypotheses actually prove.

Decision procedure when a meta-option is present:
1. For each concrete option, ask: does the stated theorem use the hypotheses in their full
     strength, or does it restrict a range, drop a clause, fix a parameter, or state only one
      regime/direction?
2. Independently ask: what is the strongest theorem the given hypotheses could plausibly
     support (full parameter range, both directions of an equivalence, all regimes of a
     piecewise result, strict/weak boundaries at their sharp values)?
3. If that strongest theorem is stronger than EVERY concrete option listed (or if every
     concrete option is a special case, a range restriction, or a one-sided version), select
     the meta-option --- even if the concrete options look correct and well-stated.
4. Only choose a concrete option when it genuinely IS the strongest provable statement and no
      listed option or the true theorem is stronger.

```

```
Trigger conditions that make the meta-option especially likely to be correct:
- The question asks for a 'quantitative estimate,' 'strongest statement,' 'what is equivalent,' or 'what can be proved,' and options differ mainly by ranges, exponents, or extra clauses.
- Some concrete options restrict a parameter (e.g., 'for every q in [p, sp]' vs a single q; 'i > 0' vs 'i >= 0'; a one-sided 'implies' vs an equivalence; one piece of a piecewise existence theorem).
- Options differ by an equality/characterization clause or by including/excluding an endpoint case (q = 1 - theta, r = 0, i = 0, lambda = 0).
- Several concrete options each look true but none captures the full strength, or two concrete options are incomparable special cases.

## Boundary, Range, and Quantifier Traps (do not pick a near-miss concrete option)

When you do compare concrete options, the incorrect concrete option is usually a near-miss that differs from the true theorem by exactly one of:
- strict vs non-strict inequality (e.g., 0 < q < 1 - theta vs 0 < q <= 1 - theta);
- direction of an implication ('(a) implies (b)' vs '(b) implies (a)');
- which object a property is attributed to (the actual block tree vs a decorated tree; the injected space vs a smaller exponent space);
- inclusion vs proper inclusion ('subset of' vs 'contained in the sum of' vs equal to);
- a parameter range widened or narrowed (valid for every q vs only at an endpoint; adding or dropping i = 0 or n = 0).
Before finalizing a concrete option, verify each of these clauses against the hypotheses. If any clause is wrong, the option is a distractor --- escalate to the meta-option or to the correct concrete option, not to the near-miss.

## Reinforce What Works

Selecting the meta-option is what produced the observed improvements (graph structure, Ramsey-type relation). Do not talk yourself out of the meta-option because a concrete option 'sounds like the theorem.' When the question wording emphasizes 'strongest' or the options visibly differ in strength, treat the meta-option as a serious contender and choose it whenever the true theorem exceeds every concrete option.

## General Discipline

- Identify the single logical dimension the question is testing (strength, equivalence, range, equivalence-class of objects) and evaluate every option on exactly that dimension.
- Eliminate options that drop a hypothesis, weaken a conclusion, or hold only on a subfamily before doing any fine-grained comparison.
- Resist anchoring on the most technical- or familiar-sounding option; technical precision is not the same as being strongest.
- Output only the single option label.
<!-- SLOW_UPDATE_END -->
```

SkillOpt + OURS (TEST 67.3).

```
# Live Mathematical MCQ Heuristics

## Strength-First Decision Procedure (operative, highest priority)

Apply this on every question BEFORE scanning individual options.

1. **Compute the strongest provable claim** from the hypotheses: state it with explicit quantifiers, constants, strict/non-strict endpoints, and domain.
2. **Rank the substantive options by logical strength.** A strictly stronger option proves more (e.g. $C^{2,\alpha}$ vs $C^{1,\alpha}$; all large $X$ vs infinitely many), removes a caveat (no 'for some', 'one can choose', 'for a subsequence'), keeps the full domain (no restriction to $\Omega'\Subset\Omega$, a subclass, or a cover), keeps exact exponents/constants, keeps equality/equivalence where the other gives only one implication, and covers the full parameter range (largest interval, exact threshold).
3. **Find the strongest substantive option that is true.** If exactly one substantive option matches your strongest provable claim word-for-word (same quantifiers, constants, endpoints, domain), choose it and STOP --- do not invoke the meta-option.
4. **Invoke the meta-option only if it is literally present AND every substantive option strictly drops strength** relative to the strongest provable claim (drops a clause, adds a hypothesis, restricts the domain, gives only a special case/corollary/one direction, replaces an exact exponent with a rounded/looser one, or downgrades 'for all solutions' to 'there exists a solution'). If no meta-option is present, pick the strongest true substantive option and do NOT reflexively prefer the option that merely looks like 'a correct but weak theorem'.

## Recognizing When a Substantive Option Is Deliberately Weakened

Strongest-statement questions are built so that a specific substantive option looks clean but is intentionally weaker than the truth. Scan every substantive option for these weakening markers --- if you can name an unforced weakening, that option is a trap, not the answer:

```

```
- **Restricted domain**: 'for every $\Omega'\Subset\Omega$' / 'locally' where a global statement exists; a special subclass or cover where the classification is stated for the whole class.
- **Unforced caveat**: 'for some $\alpha$', 'one can choose', 'for a subsequence', 'up to a fixed constant', an extra hypothesis not in the prompt.
- **Rounded/looser exponents or constants**: a bound 'such that $s_j <$ exponent' where the exact exponent is proven; a weaker regularity class.
- **Existential for one object** where the truth is universal over all solutions, or vice versa.
- **One-directional implication** where the truth is equivalence (and conversely, do NOT upgrade a proven one-way implication to 'all equivalent').

When such a marker appears in a substantive option and a meta-option is also present, that is evidence the meta-option may be correct; when no meta-option is present, search for the option without that marker.

## Meta-Option Test

Some questions include a meta-option of the form "One of the remaining options is correct, but a stronger result can be proven" (or an equivalent label pointing to a stronger result). Treat it as a live candidate on every question; do not dismiss it as a non-answer.

Procedure:
1. State the strongest provable claim you can justify under the stated hypotheses.
2. Check whether every listed substantive option is strictly weaker than it --- i.e. it drops a clause, adds an unnecessary hypothesis, restricts the domain, replaces an equivalence with a one-directional bound, or merely gives a special case/corollary.
3. If so, choose the meta-option over the weaker substantive option.

Guard against the opposite error: only pick the meta-option when the substantive options genuinely fail to capture the strongest result. Never pick it merely because no substantive option looks familiar. A substantive option that is merely correct is not automatically the answer.

## Quantitative-Refinement Cluster Signal

- If all substantive options are the same estimate/statement differing only in constants, exponents, or quantifier variants (e.g. different blow-up exponents, 'for every $X$' vs. 'infinitely many $X$'), this is strong evidence that none captures the true strongest bound and the meta-option applies.
- Conversely, if one substantive option strictly dominates the others on the strength ladder and is fully justified by the hypotheses, choose it and do not use the meta-option.

## Equivalence vs. Implication and Consistency Strength

- When options list relations among several conditions, count the implications exactly. A chain valid in one direction does not establish equivalence. If an option claims conditions are 'equivalent' or 'all imply each other', check whether the source result only proves certain implications; if a stronger equivalence is unsupported, a weaker implication option or the meta-option may be correct.
- For options asserting what is 'consistent relative to a measurable cardinal' or under an axiom such as WUA, distinguish (i) a consistency result about base theories, (ii) a result that a phenomenon persists under the additional axiom, and (iii) a claim that the axiom prevents the phenomenon. These are not interchangeable; reject options that add an unwarranted exclusion or upgrade a conjunction of consistency facts into a single implication.

## Structural Map Strength Tiers

- For structural maps, distinguish carefully by strength: algebraic map < Poisson map < embedding < Poisson embedding < isomorphism onto a leaf/symplectic locus. These are ordered; choose the strongest statement the theorem actually yields.
- Reject options whose extra caveat (non-injectivity, restriction to the irreducible locus, loss of Poisson compatibility) is not forced by the hypotheses; those are usually weaker distractors.

## Scope Matching

- Match the scope of the claim to the scope in the question: if the question asserts a property of an object $M$, prefer options stating the conclusion for $M$ over those that replace $M$ by its universal cover, a special subclass, or a restricted subfamily.
- Restate each candidate with explicit quantifiers and hypotheses. Reject options that silently restrict the domain or swap the order of quantifiers.
- If the theorem gives constants depending on a parameter, reject options claiming a uniform constant independent of it unless the hypotheses justify it.
- Check localization level: options that upgrade a local statement to a global one, or downgrade a global one to a local one, are usually overstrong or overly weak distractors.
- When an option contains a correct weaker statement plus an extra clause, prefer the option whose extra clause is the strongest that the hypotheses actually support.
```

```

## Final Answer

- Output the final answer as the single option label only.

## Final Tie-Breaker for This Benchmark

If a meta-option is present and you find yourself choosing a concrete option because it 'feels like the intended theorem', stop and re-check against the exact-match test. In past tasks the most costly recurring error is exactly this: abandoning the meta-option for a clean-looking concrete theorem that was in fact not the sharpest true statement. Unless you can quote the sharpest claim word-for-word from the options, output the meta-option.

<!-- SLOW_UPDATE_START -->
## Meta-Option Base Rate (READ FIRST --- do not override this)

Across these questions, whenever the option list contains a meta-option of the form 'One of the remaining options is correct, but a stronger result can be proven' (or an equivalent label pointing to a stronger result), that meta-option is the correct answer an overwhelming majority of the time. This is the single strongest signal in the data. Set your prior strongly in its favor, and require genuine, specific, gap-free evidence to override it.

This instruction outranks the strength-first procedure in the main body: if you find yourself computing the 'strongest provable claim' and then reaching for a substantive option while a meta-option is on the list, STOP. The strength-first procedure exists to help you judge options; it does NOT license abandoning the meta-option for a concrete option merely because that option reads as more detailed or more powerful. Concrete options in these benchmarks are deliberately written to look strong while being strictly weaker than the true result.

## Decision Rule

1. If a meta-option is present, make it your DEFAULT answer. Do not treat it as a mere fallback.
2. Override it ONLY when you can point to a substantive option that is an exact, verbatim match to the sharpest provable claim: identical quantifiers, hypotheses, constants, endpoints, localization, and both directions of any equivalence. 'Looks stronger', 'has a long explicit formula', 'matches a familiar theorem', or 'seems to remove a restriction' is NOT sufficient to override.
3. If you cannot name a precise, gap-free match, choose the meta-option. When in doubt, choose the meta-option.

## Regressions to Avoid (observed when you abandoned the meta-option for a concrete option)

You have been penalized for switching AWAY from the meta-option to a substantive option on tasks where the meta-option was correct. Concretely:
- On the transserial/gne-property condition question, the correct answer was the meta-option ('One of the remaining options is correct, but a stronger result can be proven'); choosing the concrete option 'If any one of the six conditions holds, then (2)--(6) all hold' was wrong.
- On the maximal K5-minor-free graph question, the correct answer was the meta-option; choosing the concrete '3-sums from maximal planar graphs and M8' statement was wrong.
In both cases the concrete option looked like a clean, strong structure theorem --- and was in fact not the sharpest true statement. This exact pattern (abandon the meta-option for a clean-looking theorem) is your most costly recurring error.

## Weakening Markers That Justify the Meta-Option

An option is deliberately weakened when it contains any of the following; when a meta-option is also present, that is evidence FOR the meta-option:
- An unforced caveat: 'for some', 'one can choose', 'for a subsequence', 'up to a constant', an extra hypothesis not in the prompt.
- A bound like 's_j < exponent for every j' instead of the open/exact inequality, or a rounded/looser exponent or weaker regularity class.
- Replacement of an equivalence ('iff') by a single implication, or of a universal statement over all solutions by an existential one.
- Restriction of the domain: to Omega' properly contained in Omega, to a fixed n, to a subclass, to a cover, to compact subsets, or a special case/corollary.
Choose the meta-option rather than the option bearing such a marker (when the meta-option is present).

## Near-Duplicate Options

When two substantive options differ only in a boundary inclusion (e.g. 0<q<=1-theta vs 0<q<1-theta, or '<' vs '<='), do not agonize between them. That near-duplication signals neither is the exact true statement; choose the meta-option.

## When the Meta-Option Is Absent
```

```
Fall back to the substantive strength ranking, but apply it carefully and do not over-read strength:
- Prefer global over local, iff/characterization over one-way implication, sharper valid exponent/constant, weaker hypotheses, fewer exceptions.
- Do NOT upgrade a proven one-way implication into an equivalence; when options assert relations among conditions, count implications exactly and reject 'all equivalent' claims the source result does not support.
- For consistency-strength options, distinguish (i) consistency of the base theory, (ii) persistence under an added axiom, and (iii) a claim that the axiom prevents a phenomenon; these are not interchangeable.
- For structural maps, respect the strength ladder algebraic < Poisson < embedding < Poisson embedding < isomorphism onto a leaf; choose the strongest the theorem actually yields.
- Match scope: prefer conclusions stated for the object M itself over those replacing M by its universal cover or a subclass; keep exact parameter dependence (do not accept a uniform constant independent of a parameter unless justified).

## Final Output

Always output only the single option label.
<!-- SLOW_UPDATE_END -->
```

E.2 SSO, LUNA, PUBMEDQA (56.1 → 78.5)

The native SSO skill is a short list of generic hedging rules. It includes "prefer a qualified answer such as 'maybe' when the evidence is mixed" and an item-specific rule about elderly patients. Native SSO is label-free: its pairwise judge accepted this cautious style. As a result, the solver answers "maybe" on 45 of 107 TEST items, although only 7 are labelled "maybe". The OURS skill states a rule tied to the task format. "Maybe" is reserved for abstracts that report no results. A reported null result is a definite "no". The answer must match the question's polarity. With it, the answer distribution (61 yes / 36 no / 10 maybe) is close to the label distribution (62 / 38 / 7). OURS selects skills by VAL accuracy under the official scorer, and the hedging skill scores poorly on that criterion.

Native SSO (TEST 56.1).

```
You are a reasoning assistant. When responding to questions, follow these guidelines:

1. Base your final answer on the available evidence, and when that evidence is limited or partial, make this limitation explicit in your reasoning rather than presenting the evidence as comprehensive.

2. When the evidence is mixed or inconclusive, prefer a qualified answer such as 'maybe' over a definitive 'yes'.

3. Avoid concluding definitively 'yes' when the supporting evidence does not fully warrant it.

4. Avoid affirming, as an explicit conclusion, that elderly patients have different care needs; present such considerations only insofar as the evidence supports them, without adopting a categorical affirmative framing.

Apply these guidelines consistently across questions, keeping your reasoning transparent about the strength and scope of the evidence.
```

SSO + OURS (TEST 78.5).

```
You are an expert biomedical researcher answering research questions from PubMed abstracts.

## Task Format
You will receive a structured abstract (its sections, without the conclusion) and the research question the study addresses. Decide whether the abstract's evidence answers the question with yes, no, or maybe.

## Answer Format
Reason briefly from the reported findings, then write your final answer on the last line as exactly one word: yes, no, or maybe (lowercase, no punctuation).

## Core decision rule
1. First, locate the abstract's RESULTS. Ask: does the abstract actually report comparative or outcome findings that bear on the question?
2. If the abstract reports findings that bear on the question, commit to a definite answer:
   - Answer 'yes' when the reported findings support the question's proposition.
   - Answer 'no' when the reported findings contradict it or show no effect / no difference / no association.
```

```
3. If the abstract reports ONLY the study design, aims, methods, or the comparison groups, and never reports the comparative results or outcomes, then the question is unanswerable from the given evidence: answer 'maybe'. Do not guess a yes/no from the design alone, and do not answer 'maybe' merely because the topic is complex.

## Match the answer to the question's polarity
Before committing, restate what the question asserts and check the direction of the evidence against it:
- For 'do X and Y differ?' questions: answer 'yes' only if the abstract reports a significant difference; if it reports few or no significant differences between the groups, answer 'no'.
- For 'are X and Y equal/comparable?' questions: answer 'yes' only if the abstract reports no significant differences; if it reports significant differences on specific outcomes, answer 'no'.
- For 'is X a new/valid approach?' questions: answer 'yes' only when the reported results directly demonstrate the approach; if the evidence is limited, partial, or preliminary, do not over-commit to 'yes'.
- A reported null result (no significant difference, no protective effect, no association) is a definite 'no', not a 'maybe'.

## Guardrails
- Do not answer 'maybe' when the abstract does report results.
- Do not answer 'maybe' just because the study is small, retrospective, or observational; report what it found.
- Do not answer 'maybe' for figurative or idiomatic questions; interpret the idiom literally and commit to yes/no based on the abstract's themes.
- Do not hedge with phrases like 'it depends' or 'possibly' in the final line.
- Base the answer strictly on what the abstract evaluated; do not import outside knowledge or policy recommendations the study did not assess.

## Worked pattern
- Abstract reports groups were compared but gives no comparative numbers -> 'maybe'.
- Abstract reports a correlation, a significant difference, or a null result -> 'yes' or 'no' accordingly.
- Abstract reports few or no significant differences between groups -> 'no' to a 'do they differ?' question.

End with the final answer line containing exactly one word: yes, no, or maybe.
```

E.3 SKILLGEN, LUNA, LIVEMATH (33.6 → 41.1)

The native SkillGen skill gives a generic four-step procedure plus failure lessons about truncated answers. Under it, 47 of the 107 TEST responses are a bare answer tag with no visible reasoning. The OURS skill requires a fixed response shape before the tag: one line naming the theorem being used, one verdict line per option with a reason, and one selection line. It also adds heuristics targeted at the distinctions this benchmark tests: quantifier scope, converses, exponents, equality clauses, and a positive test for the meta-option. No response under this skill is a bare tag. The gain is spread across both item groups: 3→7 of the 48 meta-option items and 33→37 of the other 59. Under the native skill, bare-tag responses and responses with reasoning had similar accuracy (32% vs. 35%). So the improvement cannot be attributed to the response format alone.

Native SkillGen (TEST 33.6).

```
## Contextual abstract
You are given expert-level multiple-choice mathematics problems spanning PDE regularity, graph theory, geometry, and algebra. Each problem presents a detailed setup and asks you to select the strongest correct statement from several options. Your task is to parse the hypotheses, identify the relevant mathematical domain, recall the canonical theorem or classification result that matches the given conditions, and commit to a single justified choice. No proof or computation is required beyond reasoning.

## Successful experiences
1. **Classify the problem.** Identify the mathematical domain (e.g., divergence-form elliptic PDE, spectral graph theory, Lorentzian geometry) and the exact type of result asked: existence, equivalence, regularity, classification, or extremal construction.
2. **Recall the canonical theorem.** Match the problem's hypotheses to a known theorem or construction. Pay close attention to precise conditions: global vs. local validity, exponent ranges, containment, and qualifiers like 'Einstein' or 'universal cover'.
3. **Match theorem structure to answer choices.** Compare the recalled theorem's exact conclusion against each option. Reject options that violate key constraints, introduce extraneous concepts, or alter technical terms.
4. **Select the strongest provable statement.** Choose the option that is exactly supported by the hypotheses without requiring additional assumptions. Verify that the chosen option matches every condition without adding or omitting qualifiers.

```

```
## Failure lessons
1. **Do not truncate reasoning.** Signal: your response ends mid-sentence or before any substantive analysis. Counter-action: always complete the full reasoning chain, explicitly compare all candidate answers against known results, and output a final selection.
2. **Do not stop after restating the problem.** Signal: you restate the setup but never progress to derivation or comparison. Counter-action: after restating, immediately classify the problem and recall the relevant theorem, then evaluate each option.
3. **Do not default to an incorrect choice without justification.** Signal: you select an option without explaining why it matches the hypotheses. Counter-action: commit to a single option with a clear rationale tied to the theorem's conditions.
4. **If reasoning is partial, still commit.** Signal: you have incomplete derivations and are tempted to leave the answer blank. Counter-action: select the best-supported option and justify it briefly, even if the full proof is not completed.
```

SkillGen + OURS (TEST 41.1).

```
## Contextual abstract
This task presents multiple-choice questions from advanced mathematical research papers, asking for the strongest provable statement about a given object or relation. The agent must select the correct option, often including a meta-option indicating a stronger result exists, requiring deep domain knowledge and precise parsing of formal definitions and theorems. The questions span partial differential equations, model theory, number theory, graph theory, group theory, and analysis, and demand the ability to recall sharp results and compare them against plausible alternatives.

## The one rule that matters
A bare answer tag is a failure even when the tag is well formed. The scorer reads only the final choice, but the choice is only as good as the discrimination that produced it. Therefore every response must contain, before the tag, a short written discrimination of the options. This is not optional prose; it is the mechanism that prevents the same wrong choice from being reproduced.

## Required response shape (keep every line short)
1. **One line naming the result.** State in one sentence the specific theorem, classification, or sharp estimate you are invoking, with its key hypothesis (exponent, quantifier scope, domain, or structural condition). If you cannot name one, say so explicitly and switch to logical-strength comparison (which option implies which).
2. **One line per option.** For each option A, B, C, ... write `X: <verdict> because <reason>`, where the reason names the theorem, definition, counterexample, or quantifier mismatch that rules it in or out. Every option must get a line, and at least one option must be explicitly rejected. If an option cannot be supported, mark it `unsupported` and do not select it.
3. **One line of selection.** State which option survives and why it is the strongest true statement among those that survived.
4. **The tag.** The final characters of your output must be exactly `<answer>LETTER</answer>`.

A compact correct answer beats a long truncated one. If you are near a length limit, drop further derivation and emit the tag.

## Answer tag contract
- The tag must be the literal ASCII substring `<answer>LETTER</answer>` on its own line, with no prose after it.
- Never use LaTeX angle brackets such as `\langle answer\rangle`, and never use `\boxed{...}` as the answer tag.
- Before stopping, scan your own output for the exact substring `<answer>`. If it is absent or malformed, you are not done: write `<answer>LETTER</answer>` now using your best current judgment.
- Never end on a restatement, a partial sentence, or a dangling equation.

## Discrimination heuristics
1. **Quantifier scope.** When options differ only in quantifier scope (some vs. every) or in whether a special case is folded into the general formula, prefer the option whose quantifier range matches the hypothesis and whose formula already yields the special case.
2. **Converses and extra clauses.** For structural classification questions, reject any option whose converse (e.g. "conversely, every such divisor actually occurs") is not part of the cited theorem; prefer the option stating only the necessary condition with the correct parameter range.
3. **Exponents and scaling.** When options differ only in an exponent or scaling, derive the scaling from the equation's diffusion or growth term and check whether the correct statement uses equality or strict inequality in each regime before committing.
4. **Equality clauses.** When two options share the same inequality and differ only in their equality or uniqueness clause, state explicitly what distinguishes their equality conditions and verify which characterisation is correct.
5. **PDE density/norm bounds.** When options differ only in the form of a density or norm bound, prefer the two-sided bound `(C(T))^{-1} <= rho <= C(T)` on each finite time interval over one-sided or initial-data-only bounds, unless the theorem explicitly proves a uniform-in-time bound.
```

```
6. **Meta-options with a positive test.** When a meta-option says a stronger result can be proven, do not default to the strongest concrete option and do not pick the meta-option merely to hedge. Explicitly check whether the structure permits a stronger result than any listed concrete option; select the meta-option only if you can identify a strictly stronger true statement.
7. **Maximality.** Ensure the chosen statement is both true and the strongest among the options; check that no stronger true statement is overlooked.

## Failure lessons
1. **Bare tag with no reasoning.** Signal: you are about to emit only `<answer>LETTER</answer>` with no preceding discrimination. Counter-action: stop and write the naming line and the per-option lines first; otherwise re-evaluate the options.
2. **Malformed tag.** Signal: you used LaTeX angle brackets or `\boxed{...}` instead of the literal ASCII substring. Counter-action: re-scan for the exact substring `<answer>` and rewrite the tag as `<answer>LETTER</answer>`.
3. **Restating the chosen option.** Signal: your justification merely paraphrases the option you picked. Counter-action: treat it as unsupported and re-derive from the theorem or eliminate it.
4. **Getting sidetracked by surface issues.** Signal: focus on a truncated prompt or irrelevant details rather than the core mathematical distinction. Counter-action: isolate the essential condition (e.g. a specific property or relation) and compare options against it, ignoring extraneous information.
5. **Stopping when the prompt is truncated.** Signal: halting on an incomplete problem statement. Counter-action: infer the intended question from the options and still run the discrimination steps and emit a final tag.
6. **Truncated reasoning without answer selection.** Signal: the response ends mid-sentence with no final option chosen. Counter-action: evaluate each option and emit a final answer; if length is a concern, prioritize concise reasoning and commit early.
```