Evolutionary System Prompt Learning for Reinforcement Learning in LLMs

Lunjun Zhang [1] **Ryan Chen** [2 3] **Bradly C. Stadie** [2 3]

Abstract

Building agentic systems that can autonomously self-improve from experience is a longstanding goal of AI. Large language models (LLMs) today primarily self-improve via two mechanisms: self-reflection for context updates, and reinforcement learning (RL) for weight updates. In this work, we propose **Evolutionary System Prompt Learning (E-SPL)**, a method for jointly improving model contexts and model weights. In each RL iteration, E-SPL samples trajectories under multiple system prompts in parallel, then jointly applies RL updates to LLM weights and evolutionary updates to system prompts. System prompts evolve via mutation and crossover, two genetic operators driven by LLM self-reflection; selection is based on relative performance ratings updated across RL iterations. E-SPL encourages a natural division between declarative knowledge encoded in prompts and procedural knowledge encoded in weights, resulting in improved performance across reasoning and agentic tasks. For instance, in an easy-to-hard (AIME → BeyondAIME) generalization setting, E-SPL improves RL success rate from $38.8\% \rightarrow 45.1\%$ while also outperforming reflective prompt evolution (40.0%). Overall, our results demonstrate that RL and system prompt evolution are deeply synergistic, and combining the two yields consistent gains in sample efficiency and generalization.

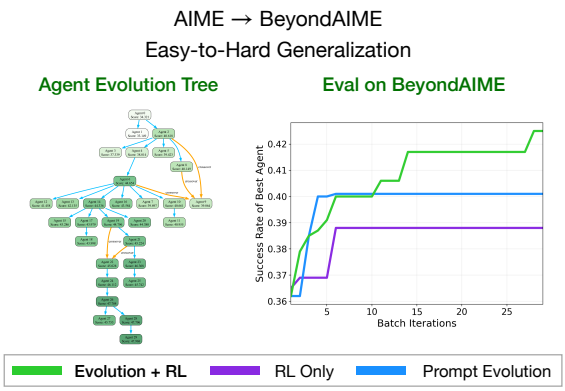

Figure 1. **Evolving System Prompts helps RL generalize better**.

1. Introduction

Evolutionary algorithms (EA) and reinforcement learning (RL) have long been studied as complementary paradigms for optimizing intelligent behaviors. Evolutionary methods excel at population-level exploration over discrete, high-level structures (Koza, 1994; Bäck et al., 1997; Lehman & Stanley, 2011), while reinforcement learning is well suited for fine-grained optimization of policy parameters through trial and error (Williams, 1992; Sutton et al., 1999; Silver et al., 2016). Historically, however, these two approaches have largely operated on different representational levels: evolution over programs (Langdon & Poli, 2013) and architectures (Stanley & Miikkulainen, 2002) on the one hand, and gradient-based learning over a fixed set of parameters (Schulman et al., 2015; Mnih et al., 2015) on the other. Bridging these representational levels in a unified and scalable way remains an open challenge, particularly in the context of large language models (LLMs) (Brown et al., 2020; Achiam et al., 2023), where both the model weights (Jaech et al., 2024) and the model contexts (Wei et al., 2022; Zhou et al., 2022) critically shape behavior.

Recent work on prompt evolution and LLM agents demonstrates that prompt (or context) engineering is a powerful mechanism for improving test-time performance. Techniques such as self-reflection (Shinn et al., 2023; Madaan et al., 2023; Cai et al., 2025) and population-based search (Guo et al., 2023; Fernando et al., 2023; Romera-Paredes et al., 2024; Agrawal et al., 2025) show that prompts can encode substantial declarative knowledge and behavioral priors without modifying model weights. At the same time, RL has emerged as the predominant post-training approach to fundamentally improve reasoning capabilities (Jaech et al.,

[1]Department of Computer Science, University of Toronto [2]Department of Statistics and Data Science, Northwestern University [3]Bridgewater AIA Labs. Correspondence to: Lunjun Zhang <lunjun@cs.toronto.edu>.

The 3rd *AI for Math Workshop* at the 43rd International Conference on Machine Learning (ICML), Seoul, South Korea, 2026.
Copyright 2026 by the author(s).

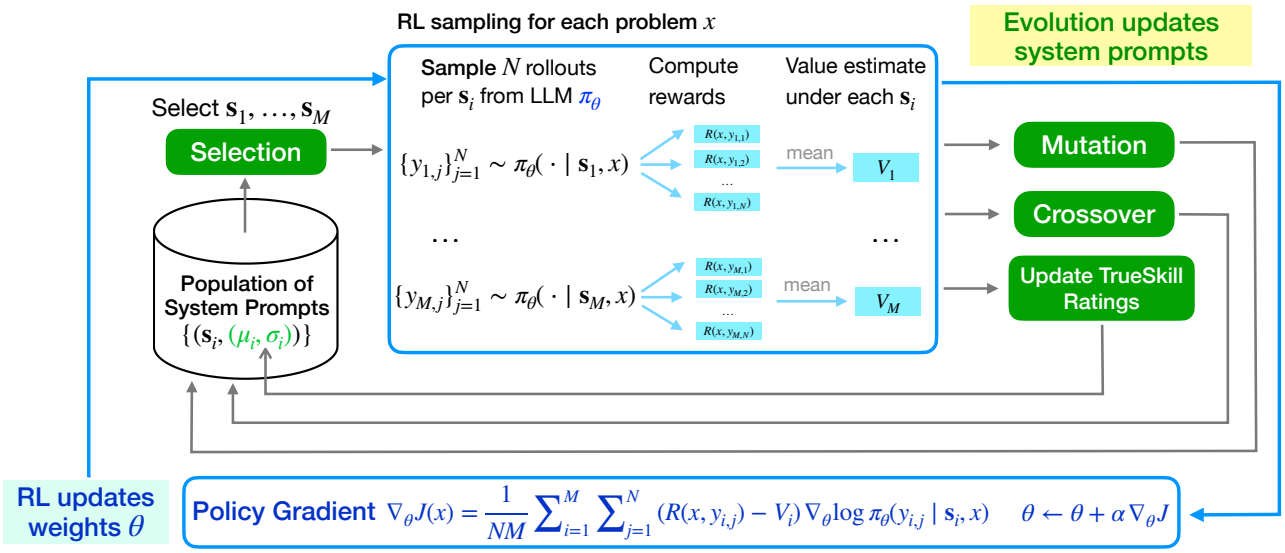

Figure 2. **Evolutionary System Prompt Learning (E-SPL)** jointly optimizes model contexts and model weights to enhance LLM self-improvement. Evolution updates system prompts; RL updates weights. The learned system prompts can encode *declarative knowledge* via articulated principles and strategies, while RL gradients can further hone the model's *procedural knowledge* for reliable execution.

2024; Guo et al., 2025; Chen et al., 2025; Comanici et al., 2025). Despite their complementary strengths, existing approaches typically optimize model contexts and weights in isolation, leaving their potential synergies unexplored.

In this work, we argue that effective self-improving agents should jointly evolve their high-level behavioral specifications and their low-level parametric policies. This perspective motivates **Evolutionary System Prompt Learning (E-SPL)**, a framework that tightly couples reinforcement learning over model weights with evolutionary search over system prompts. By maintaining a population of system prompts that are evaluated, rated, and evolved alongside RL training, E-SPL enables structured exploration over declarative knowledge while encoding procedural knowledge into weights. This joint optimization yields a synergistic dynamics in which improved prompts accelerate learning, and improved weights amplify the benefits of prompt evolution.

Experimentally, E-SPL enhances RL performance of from $56.3\% \rightarrow 60.6\%$ on AIME 2025, $50.0\% \rightarrow 52.7\%$ on HMMT 2025 November test, and notably, $38.8\% \rightarrow 45.1\%$ on AIME $\rightarrow$ BeyondAIME in an easy-to-hard generalization setting, using a base model of DeepSeek v3.1 (DeepSeek-AI, 2024). E-SPL significantly outperforms prompt evolution as well, showing that weight updates are often necessary for improving the model's tacit knowledge and intuition. On an agentic search task (Jin et al., 2025), we improved RL performance from $44.2\% \rightarrow 48.6\%$ on OpenAI's gpt-oss-120b (Agarwal et al., 2025) with limited data. Together, these results show that combining RL and system prompt evolution unlocks a synergistic form of self-improvement that neither approach achieves in isolation.

2. Related Work

Prompt Evolution and Self-Reflection Evolutionary and search-based approaches such as EvoPrompt (Guo et al., 2023), PromptBreeder (Fernando et al., 2023), Automatic Prompt Engineer (Zhou et al., 2022), and FunSearch (Romera-Paredes et al., 2024) demonstrate that population-based search over prompts can substantially improve test-time performance. Iterative self-reflection methods, including Reflexion (Shinn et al., 2023), Self-Refine (Madaan et al., 2023), and Training-free GRPO (Cai et al., 2025), further show that LLMs can self-improve through context refinement alone. However, these approaches operate with frozen model parameters, limiting their ability to improve implicit knowledge acquired through experience.

Scaling RL for LLMs RL has become the primary mechanism for improving LLM reasoning, from early preference-based alignment (Stiennon et al., 2020; Ouyang et al., 2022) to reasoning-focused methods such as o1 (Jaech et al., 2024), R1 (Shao et al., 2024; Guo et al., 2025), and further improvements to policy gradients (Yu et al., 2025; Chen et al., 2025; Williams, 1992; Schulman et al., 2017). However, these methods assume a fixed system prompt during training, treating contexts as exogenous. While Ye et al. (2025) discover prompts within RL loops, those serve as synthetic training data rather than system-level declarative knowledge. Recent observations (Tan et al., 2025; Azarbal et al., 2025; MacDiarmid et al., 2025) suggest that system-level contexts can improve RL generalization by reducing optimization pressure to globally update model weights.

Evolutionary Algorithms and Genetic Programming Hybrid Evolution+RL approaches (Khadka & Tumer, 2018;

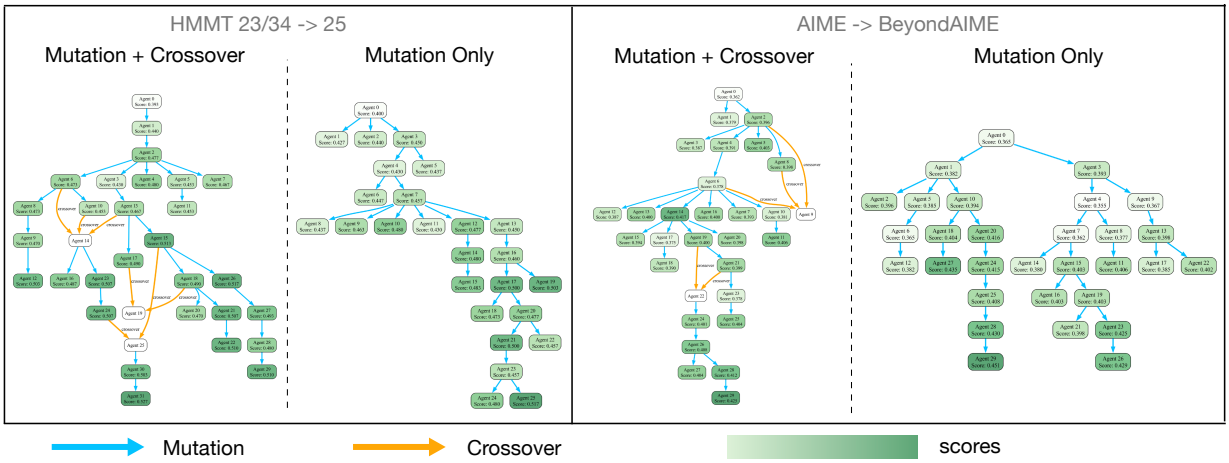

Figure 3. **Evolutionary trees of E-SPL**. During RL, E-SPL creates an evolutionary tree of system prompts, by re-suing the same data already generated by RL with minimal additional computational overhead. Each genetic operator (mutation or crossover) only requires a sampling server for LLM self-reflection with different context construction strategies, which can be concurrent with RL gradient updates.

Discovered System Prompts for Solving Math Problems (deepseek v3.1)

[1] For generating functions with a numerator $(1 - x^M)^k$ and a denominator as a product of geometric series, the coefficient of x^N is a sum over j of binomial coefficients times the number of solutions to a Diophantine equation. If $N < M$, only the $j = 0$ term contributes. For $N \geq M$, multiple j may contribute, and the signs from the binomial coefficients must be considered.

[2] After computations, **verify results for consistency across domains: check denominators, symmetry, and physical plausibility. Use alternative methods to recompute critical quantities and detect patterns such as periodicity.**

[3] Avoid detours and unjustified guesses, including contest heuristics without validation. **Trust sound derivations; if inconsistencies persist, explore alternative configurations or identities.**

[4] To verify modular injectivity, factor differences and check non-vanishing of the cofactor. For complex cases, test small primes; derivative condition is sufficient but not necessary.

[5] For sums over integer pairs with conditions like $\gcd(a, b) = 1$, consider using Möbius inversion or changing the order of summation to simplify the expression.

[6] When free parameters appear but a fixed answer is required, **look for cancellations forced by constraints or boundary conditions. Verify that the conclusion is consistent with all given conditions.**

Figure 4. **Discovered strategies in learned System Prompts for solving math problems.** Those explicit behavior specifications include: useful heuristics and tips for various categories of problems, self-verification strategies such as checking for consistency and plausibility, a list of common failure modes to avoid, etc. Note that RL is done under diverse system prompts, and does not overfit to any particular one.

Jaderberg et al., 2017; Pourchot & Sigaud, 2018) leverage population dynamics to complement gradient-based optimization. More recently, FunSearch (Romera-Paredes et al., 2024) and AlphaEvolve (Novikov et al., 2025) demonstrate that LLM-guided evolutionary search over programs can yield novel discoveries. The Darwin Gödel Machine (Zhang et al., 2025) applies self-referential modification to agentic prompts but does not update model weights. Classical genetic programming (Koza, 1994; Langdon & Poli, 2013) refines candidate programs via mutation and crossover; an LLM under a system prompt can be viewed as such a program (Agrawal et al., 2025; Novikov et al., 2025), with LLM-designed genetic operators (Hemberg et al., 2024). Unlike prior work that combines LLM evolution with DPO (Rafailov et al., 2023) for combinatorial tasks, E-SPL tackles the more general setting of LLM post-training, jointly optimizing system prompts and model weights in a single loop without additional agentic scaffold (Gottweis et al., 2025).

3. Method

E-SPL conditions RL weight updates on a population of system prompts, and reuse the resulting RL data to jointly evolve this population via two genetic operators, mutation and crossover, both of which are based on LLM self-

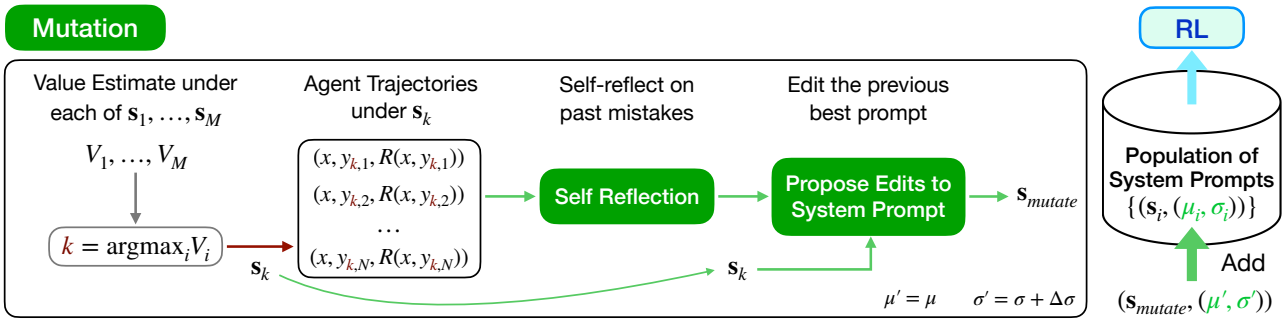

Figure 5. **Mutation operator** in E-SPL. The highest-performing prompt in each iteration undergoes **LLM self-reflection** on group-wise agent trajectories and their outcomes. An LLM-generated diff edits the parent into a child system prompt, removing ineffective rules and converting observed mistakes into improved declarative instructions, yielding a new prompt that enters the evolutionary population.

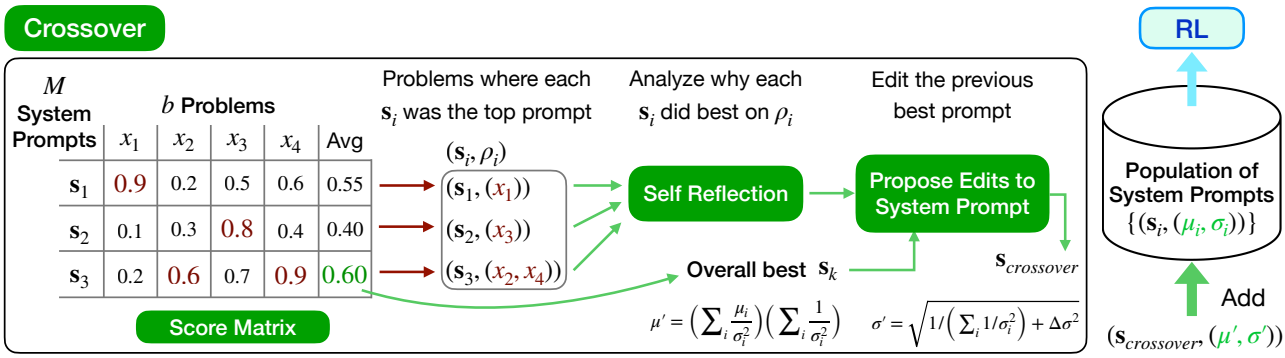

Figure 6. **Crossover operator** in E-SPL. System prompts are compared based on their problem-wise performance within the current RL batch. Guided by these differential strengths and weaknesses, an **LLM self-reflection** process selectively recombines the most effective complementary segments from multiple parent prompts, yielding a new child prompt that enters the evolutionary population.

reflection. **We design the evolutionary algorithm to be directly compatible with RL post-training pipelines, allowing RL to benefit from an evolving repertoire of declarative knowledge** while incurring minimal additional overhead.

3.1. Reinforcement Learning for Weight Update

The RL component of E-SPL optimizes the LLM policy π_θ to maximize rewards on dataset $\mathcal{D}$ under a variety of system prompts. Each system prompt, denoted as $\mathbf{s}$, is a list of strategic instructions that contain declarative knowledge on how to solve various categories of problems. The system prompts are not given *a priori*, but are instead discovered by the LLM through self-reflection on its own experience during the RL training process. Each RL iteration uses M system prompts $\{\mathbf{s}_i\}_{i=1}^M$ in parallel, so that the model weights do not overfit to any particular system prompt; this also induces direct evolutionary competition among system prompts. For each problem $x \sim \mathcal{D}$ and each system prompt $\mathbf{s}_i$, we sample N trajectories $y_{i,j} \sim \pi_\theta(\cdot \mid \mathbf{s}_i, x)$ for $j = 1, \ldots, N$, compute outcome rewards $r_{i,j} = R(x, y_{i,j})$, and value estimates $V_i = \frac{1}{N}\sum_{j=1}^N r_{i,j}$. The policy gradient objective for each problem x is $\mathcal{J}(\theta, x) = \frac{1}{NM}\sum_{i=1}^M \sum_{j=1}^N (r_{i,j} - V_i) \log \pi_\theta(y_{i,j} | \mathbf{s}_i, x)$. We update θ on a batch $\mathcal{B} \sim \mathcal{D}$, regularized toward π_{ref}: $\theta \leftarrow \theta + \alpha \frac{1}{|\mathcal{B}|}\sum_{x \in \mathcal{B}} \nabla_\theta(\mathcal{J}(\theta, x) - \beta\, \text{KL}(\pi_\theta, \pi_{\text{ref}}))$. When using low-rank adaptation (Hu et al., 2022), the KL term is implicit and often ignored ($\beta = 0$).

3.2. Evolutionary Algorithm for Context Update

Integrating evolutionary search with online RL poses two challenges: prior genetic operators assume a stationary fitness landscape, whereas RL shifts model weights every iteration; and they require separate evaluation on held-out data, whereas RL generates rollouts on-policy. E-SPL addresses both by reusing RL rollouts for fitness evaluation and employing TrueSkill ratings to aggregate relative rankings across non-stationary iterations.

Our evolutionary algorithm (**EA**) optimizes the LLM system prompt to best elicit model capabilities and encode explicit knowledge learned from experience. Like in classical EA, we keep a population of candidates, and: (1) **evaluate the fitness** of different system prompts continuously during training; (2) **select** the successful ones from the population; (3) **reproduce** new system prompts from selected parents to add to the population via mutation and crossover – both grounded in LLM self-reflection but differing in their context construction strategies. The population $\mathcal{S}$ starts with a single root, and gradually grows into an evolutionary tree.

Algorithm 1 Evolutionary System Prompt Learning for RL

Input: Problem set $\mathcal{D}(\cdot)$, reward function $R(\cdot,\cdot)$.
Input: RL policy $\pi_\theta(\cdot)$; reference policy $\pi_{\text{ref}}(\cdot)$.
Input: RL hyper-parameters: N, α, β.
Input: Initial system prompt $\mathbf{s}_{\text{root}}$; M, $p_{\text{crossover}}$.
Input: Initial (μ_0, σ_0) for TrueSkill; $\Delta\sigma$; λ.
Initialize $\mathcal{S} = \{(\mathbf{s}_{\text{root}}, \mu_0, \sigma_0)\}$. {**Population to Evolve**}
while not converged **do**
  Sample a problem $x \sim \mathcal{D}$
  Sample M system prompts $\{(\mathbf{s}_i, \mu_i, \sigma_i)\}_{i=1}^{M} \sim \mathcal{S}$, each according to probability $p_i \propto \exp(\mu_i + \lambda\sigma_i)$
  for $i = 1$ **to** M **do**
    Sample N solutions $\{y_{i,j}\}_{j=1}^{N}$: $y_{i,j} \sim \pi_\theta(\cdot \mid \mathbf{s}_i, x)$
    Reward $r_{i,j} = R(x, y_{i,j})$, value $V_i = \frac{1}{N}\sum_{j=1}^{N} r_{i,j}$
  end for
  $\mathcal{J}(\theta) = \frac{1}{NM}\sum_i \sum_j (r_{i,j} - V_i) \log \pi_\theta(y_{i,j}|\mathbf{s}_i, x)$
  $\theta \leftarrow \theta + \alpha \nabla_\theta(\mathcal{J}(\theta) - \beta \mathrm{KL}(\pi_\theta, \pi_{\text{ref}}))$ {**RL update**}

  $\mathbf{v} \leftarrow \mathrm{argsort}([V_1, \ldots, V_M])$ {Rank the list of $\mathbf{s}_i$}
  $\{(\mu_i, \sigma_i)\}_{i=1}^{M} \leftarrow \mathrm{TrueSkill.update}(\{(\mu_i, \sigma_i)\}_{i=1}^{M}, \mathbf{v})$ {Change the ratings in the population $\mathcal{S}$}

  $k \leftarrow \mathrm{argmax}_i V_i$ {Only mutate the best prompt $\mathbf{s}_k$}
  Create **mutation self-reflection** prompt Ψ_{reflect} using $[(x, y_{k,1}, r_{k,1}), \ldots, (x, y_{k,N}, r_{k,N})]$.
  $\ell \sim \pi_{\text{ref}}(\cdot \mid \Psi_{\text{reflect}})$ diff $\sim \pi_{\text{ref}}(\cdot \mid \mathbf{s}_k, \ell)$
  $\mathbf{s}_{\text{mutate}} \leftarrow \mathrm{git.apply}(\mathbf{s}_k, \mathrm{diff})$ {**Mutation step**}
  $\mathcal{S}.\mathrm{append}((\mathbf{s}_{\text{mutate}}, \mu_k, \sigma_k + \Delta\sigma))$

  if $p_{\text{crossover}} > \mathrm{random.rand}()$ **then**
    Let $\mathcal{B} = \{x_b\}_{b=1}^{B}$ be a batch of B past problems
    Let $\Omega = \{(\mathbf{s}_i, \mu_i, \sigma_i)\}_{i=1}^{M}$
    $(\mathbf{s}', \mu', \sigma') = \mathrm{Crossover}(\mathcal{B}, R, \Omega, \pi_\theta, \pi_{\text{ref}})$ {**Alg 2**}
    $\mathcal{S}.\mathrm{append}((\mathbf{s}', \mu', \sigma'))$
  end if
end while

Fitness Evaluation Our EA reuses RL rollout statistics directly, eliminating the held-out evaluation typical of prompt evolution methods. Since absolute returns are incomparable across iterations (different batches and weights), we treat each iteration as a **tournament**: within an iteration all prompts share the same batch and weights, so only their relative ordering matters. We rank prompts by V_i, compute $\mathbf{v} \leftarrow \mathrm{argsort}([V_1, \ldots, V_M])$, and update persistent TrueSkill (Herbrich et al., 2006) ratings (μ_i, σ_i) via Bayesian message passing (see §B).

Selection We sample M prompts per iteration using an optimistic UCB-style score: $\mathrm{score}_i = \mu_i + \lambda\sigma_i$, $p_i \propto \exp(\mathrm{score}_i/T)$, where λ and T trade off exploitation and exploration. We restrict sampling to the latest K prompts (a sliding window) to prevent stale prompts from being selected.

Mutation We mutate only the best-performing prompt $\mathbf{s}_k$ ($k \leftarrow \mathrm{argmax}_i V_i$), creating direct evolutionary pressure: **only the prompt that outperforms all competitively selected peers reproduces**. The reference policy π_{ref} first self-reflects on the rollouts under $\mathbf{s}_k$ to produce lessons $\ell \sim \pi_{\text{ref}}(\cdot \mid \Psi_{\text{reflect}}, \{(x, y_{k,j}, r_{k,j})\}_{j=1}^{N})$, then proposes edits diff $\sim \pi_{\text{ref}}(\cdot \mid \mathbf{s}_k, \ell)$ applied as $\mathbf{s}_{\text{mutate}} \leftarrow \mathrm{git.apply}(\mathbf{s}_k, \mathrm{diff})$. The child is added to $\mathcal{S}$ with rating $(\mu_k, \sigma_k + \Delta\sigma)$; inheriting the parent's rating with increased uncertainty favors exploration.

Crossover While mutation evolves individual lineages, crossover recombines knowledge across parents. We evaluate M prompts on a batch $\mathcal{B}$ of B problems, yielding a score matrix $\Phi \in \mathbb{R}^{M \times B}$ where $\Phi_{i,b}$ captures the per-problem advantage of each prompt. The problem-wise strengths identified by Φ are incorporated into the crossover reflection prompt Ψ_{cr}, grounding self-reflection in concrete differential advantages and enabling targeted recombination. The resulting $\mathbf{s}_{\text{crossover}}$ is added to $\mathcal{S}$ with a precision-weighted average of the parents' ratings (Algorithm 2).

Summary E-SPL augments RL post-training with an evolutionary algorithm based on self-reflection from experience. In parallel with RL updates, E-SPL uses relative performance rankings of system prompts to inform evolutionary selection. Mutation and crossover operators, both driven by LLM self-reflection, produce new system prompts with minimal computational overhead.

4. Experiments

We evaluate E-SPL on mathematical reasoning and agentic search benchmarks, asking: **(1)** Does jointly evolving system prompts and model weights outperform optimizing either alone? **(2)** How does E-SPL affect the learning speed and generalization of RL? **(3)** What emergent patterns arise in discovered system prompts?

4.1. Experimental Setup

Tasks and datasets. Our main evaluations include: **(1) DAPO100→AIME25**, which follows the protocols from (Cai et al., 2025) that trains on 100 selected problems from the DAPO dataset (Yu et al., 2025) and tests on AIME 2025 competition; **(2) HMMT 23/24→25**, which uses all of HMMT 2023, 2024, and February 2025 problems as training set, and November 2025 problems as test set; **(3) AIME→BeyondAIME**, which uses past years of AIME (AIME 22, AIME 23, and AIME 24) as the training set, and BeyondAIME (ByteDance-Seed, 2025) as test set, a set of problems at or above AIME difficulty. In addition, we also evaluate on **(4) Agentic Search** on Natural Questions (NQ) (Kwiatkowski et al., 2019) and HotpotQA (Yang et al., 2018), where we subsample 196 problems for training

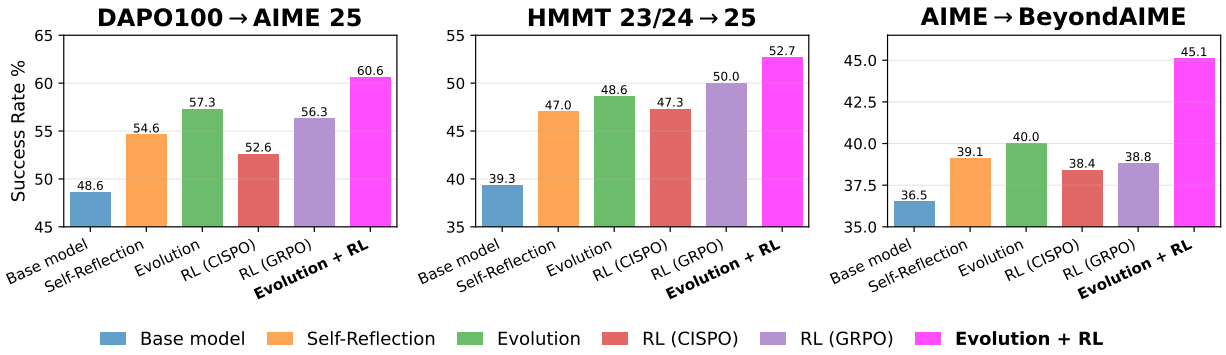

Figure 7. Results on mathematical reasoning datasets using DeepSeek-v3.1. **Evolution + RL** (our E-SPL method) performs best compared to self-reflection (prompting only), evolution (prompting only), and RL (weight update only). The results show that coupling RL with evolution on system prompts unlocks a synergistic form of self-improvement that neither approach achieves in isolation.

and 64 problems for test; the task is multi-hop question answering with search engine API calls within the chain-of-thoughts (Jin et al., 2025).

Models On math tasks, we use DeepSeek v3.1 (DeepSeek-AI, 2024), a 671B MoE (Shazeer et al., 2017) model with 37B active parameters. On agentic search tasks, we use OpenAI's gpt-oss-120b (Agarwal et al., 2025), which is a 120B MoE model with 5B active parameters. All finetuning is done via low-rank adaptation (Hu et al., 2022) with a rank of 32.

Methods compared We compare the following variants:

- **RL only**: standard RL post-training with a fixed system prompt; we benchmark the popular **GRPO** (Shao et al., 2024) and **CISPO** (Chen et al., 2025) algorithms;
- **Self-reflection**: iterative prompt refinement via reflection (Shinn et al., 2023; Cai et al., 2025; Ouyang et al., 2025), without weight updates. Equivalent to mutation-only with frozen weights and population size 1.
- **Evolution**: which runs our evolutionary algorithm (EA) for system prompt search with frozen model weights. Our EA is similar to GEPA (Agrawal et al., 2025) but uses only train-time rollout statistics rather than a separate feedback set.
- **Evolution + RL (E-SPL)**: our full method, jointly evolving contexts and updating weights.

4.2. Main Results

Figure 7 summarizes performance on math reasoning benchmarks using DeepSeek-v3.1 model. Figure 10 summarizes our results on agentic search tasks using OpenAI gpt-oss-120b. Across all datasets, **Evolution + RL** (E-SPL) consistently outperforms baselines that optimize either prompts (evolution-only) or weights (RL-only) in isolation.

- On **AIME 2025**, E-SPL improves success rate from $56.3\% \rightarrow 60.6\%$ over RL-only.
- On **HMMT 2025**, performance improves from $50.0\% \rightarrow 52.7\%$ after applying E-SPL.
- On an easy-to-hard generalization (Sun et al., 2024) setting, **AIME** $\rightarrow$ **BeyondAIME**, the gains are most pronounced, where E-SPL improves RL from $38.8\% \rightarrow 45.1\%$ in success rate, capable of learning more generalizable strategies from easier training problems.
- On **agentic search** tasks, E-SPL improves RL from $44.2\% \rightarrow 48.6\%$, which demonstrates that E-SPL is not limited to mathematical reasoning, but is also effective in agentic reasoning with tool use, where evolved prompts discover novel workflows (Figure 16).
- On all tasks above, E-SPL outperforms evolution-only as well, with relative improvements over evolution-only ranging from $+5.8\%$ to $+12.8\%$ on math reasoning tasks, and $+42.9\%$ on the agentic search task.

Evolution-only consistently outperforms self-reflection, and can sometimes outperform RL-only. On both DAPO100 $\rightarrow$ AIME25 and AIME $\rightarrow$ BeyondAIME, evolution outperforms RL; on HMMT 23/24 $\rightarrow$ 25, GRPO slightly outperforms evolution. This is consistent with prior findings in GEPA (Agrawal et al., 2025).

Figure 8 shows that evolution-only learns fastest early on but plateaus, while E-SPL achieves both faster learning and higher asymptotic performance.

Notably, the relative strength of RL versus evolution varies by domain — RL dominates in agentic search (44.2% vs 34.0%) but is on-par with evolution in math reasoning. E-SPL combines the strengths of both regardless of domain.

4.3. Emergent Patterns in Learned System Prompts

Generative Self-Verification A recurring theme is generative self-verification (Zhang et al., 2024). In agentic search,

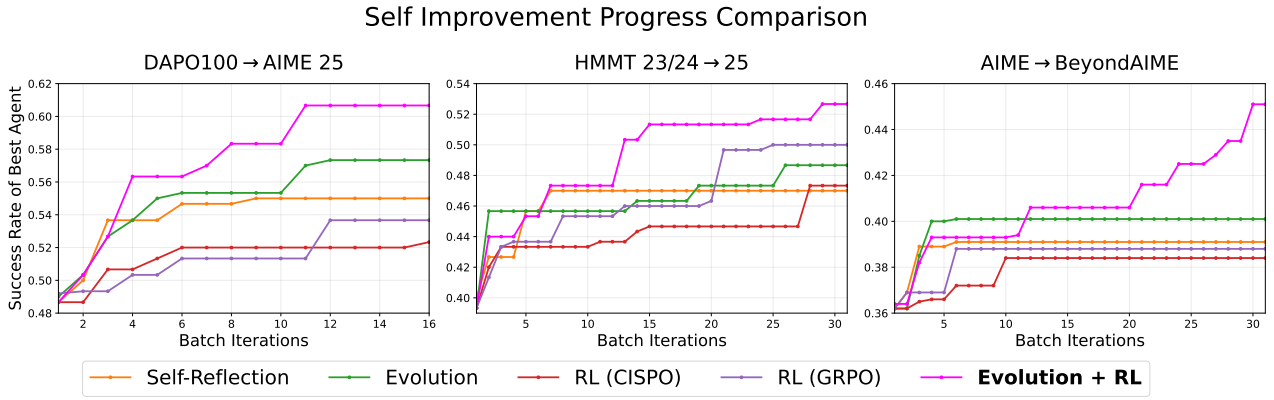

Figure 8. **E-SPL learns faster, and achieves better asymptotic performance that neither RL nor prompt evolution alone could achieve.** By studying the learned system prompts (e.g. Figure 4 and Figure 17), we hypothesize that E-SPL achieves this by encouraging a natural division between declarative knowledge (factual information or global heuristics that can be verbally explained) in prompts and procedural knowledge (practical, instinctive "know-how" that relies on intuition developed through repetition) in weights.

	Mistakes in Learned E-SPL System Prompts
Discovered	For a polynomial $P(n) = \prod_{i=1}^{m}(n - a_i)$, the largest integer d that divides $P(n)$ for all integers n is $d = \prod_p p^{e_p}$, where $e_p = \min_{r \bmod p}\big(\#\{\, i : a_i \equiv r \pmod p \,\}\big)$. Only primes $p \le \max(a_i)$ need be considered, as for larger p, the a_i are distinct modulo p, so $e_p = 0$. For primes where the function mapping i to $a_i \bmod p$ is a permutation (e.g., if $a_i = i^3$ and $p \equiv 2 \bmod 3$), $e_p = \left\lfloor \frac{m}{p} \right\rfloor$.
Corrected	For a polynomial $P(n) = \prod_{i=1}^{m}(n - a_i)$, the largest integer d that divides $P(n)$ for all integers n is $d = \prod_p p^{e_p}$, where $e_p = \min_{n \in \mathbb{Z}} v_p\big(P(n)\big)$. Moreover, one always has the lower bound $e_p \ge \min_{r \bmod p}\big(\#\{\, i : a_i \equiv r \pmod p \,\}\big)$. In particular, for this mod-p bound it suffices to consider primes $p \le m$, since if $p > m$ then some residue class occurs zero times and the minimum is 0. For primes where the function mapping i to $a_i \bmod p$ is a permutation (e.g., if $a_i = i^3$ and $p \equiv 2 \bmod 3$), the bound gives $e_p \ge \left\lfloor \frac{m}{p} \right\rfloor$.
Discovered	For graphs with odd-degree vertices, find a minimum T-join by pairing odd vertices along efficient paths. In symmetric graphs such as grids, pair boundary vertices optimally to minimize total length.
Corrected	For graphs with odd-degree vertices, find a minimum T-join by pairing odd vertices via a minimum-weight perfect matching on shortest-path distances. In symmetric graphs such as grids, restrict the matching to symmetry-respecting pairings to minimize total length.

Table 1. The discovered system prompts sometimes include **heuristics that are not always true but still helpful to problem-solving**. Unlike RL updates to model weights, the learned system prompts are interpretable and thus can be monitored and corrected. Here we showcase two subtle mistakes in two discovered principles, and what their **corrected** versions should be (see §C.1 for explanations).

the learned system prompt (Figure 16) specifies a workflow that labels each search result as VERIFIED or NOT VERIFIED based on source reliability and verbatim phrase matching, looping until verification succeeds. In math, a similar strategy emerged: *'perform sanity checks: test small instances, ensure bounds, and verify extreme cases'* (Figure 18). RL then refines execution of these workflows rather than discovering them from scratch.

Codified Domain Expertise E-SPL also codifies domain-specific heuristics — e.g., when to apply the Chinese Remainder Theorem or Möbius inversion (Figure 17, Figure 9). These declarative strategies let RL focus on honing core reasoning skills rather than memorizing domain knowledge.

Failure Mode: Missing Caveats Not all discovered heuristics are strictly correct — some over-generalize or miss caveats (Table 1). However, unlike opaque RL weight updates, E-SPL's learned knowledge is explicit and editable, making it amenable to human monitoring, correction, and scalable oversight (Leike et al., 2018; Bowman et al., 2022).

4.4. Ablation Studies

Sampling Policy for Evolution Figure 11 shows that using a fixed reference policy π_{ref} for self-reflection consistently outperforms using the continually changing RL policy π_θ. A fixed reference provides a stable basis for mutation and crossover, whereas allowing the editor to drift alongside the policy introduces instability.

Effect of Crossover Adding crossover to mutation accelerates early learning by propagating useful substructures across lineages, with comparable final performance (Fig-

Figure 9. **An example of how the RL model utilizes learned system prompts** (also see Figure 22).

Example of How E-SPL Shapes Model Behavior

Learned System Prompt

When solving math problems, you MUST first carefully read and understand the helpful instructions and principles below:
··· **[G2]. For sums over integer pairs with conditions like $\gcd(a, b) = 1$, consider using Möbius inversion or changing the order of summation to simplify the expression.** ···

Problem

Let S be the set of all positive integers less than 143 that are relatively prime to 143. Compute the number of ordered triples (a, b, c) of elements of S such that $a + b = c$.

Model Output

Let's solve the problem step by step:
Step 1: Factorize $143 = 11 \times 13$. So, S consists of positive integers < 143 that are coprime to both 11 and 13.
···
Step 5: **Use Möbius inversion (Principle [G2]).** Let $f(n)$ be the number of pairs $(a, b) \in S \times S$ with $a + b = n$. We want $\sum_{n=1}^{142} f(n) \cdot \mathbf{1}_{\gcd(n,143)=1}$. ···
···
Step 11: Apply inclusion-exclusion. ··· Therefore, the number of ordered triples (a, b, c) is 5940.
Final Answer:

$$\boxed{5940}$$

Ground truth answer: 5940

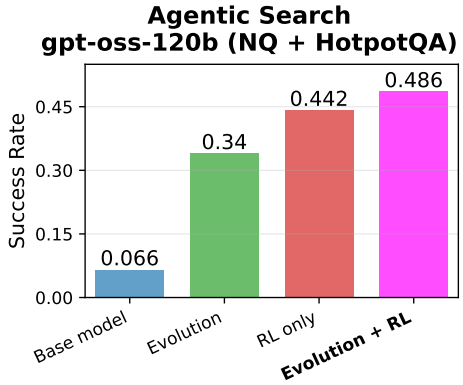

Figure 10. E-SPL outperforms both Evolution and RL on agentic search tasks with gpt-oss-120b.

ure 15). Both mutation-only and mutation+crossover consistently outperform RL-only, confirming that E-SPL's gains are attributable to evolutionary search over system prompts as a whole, rather than any specific operator.

Hyperparameter Sensitivity TrueSkill is used off-the-shelf with default parameters (see §B). The remaining hyperparameters (see §A.6) are either standard RL hyperparameters or EA hyperparameters shared identically with baselines, so E-SPL introduces no additional tuning burden beyond the baselines of RL and EA.

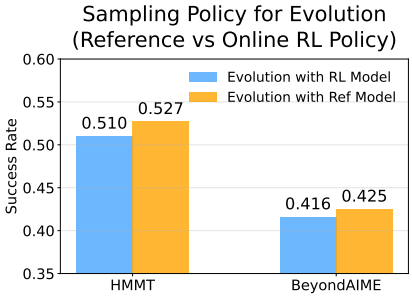

Figure 11. Using a fixed reference policy rather than the online RL policy for prompt evolution is important.

5. Conclusion

We have shown that RL and evolutionary prompt search are deeply synergistic: E-SPL jointly optimizes model weights and system prompts, yielding consistent gains in sample efficiency, asymptotic performance, and generalization across reasoning and agentic tasks. A key insight is that system prompts provide a natural substrate for declarative knowledge that complements the procedural knowledge acquired through RL, and that the two forms of self-improvement reinforce each other. We hope E-SPL motivates further work on unifying evolutionary self-rewrite with gradient-based learning for increasingly autonomous self-improving agents.

References

Achiam, J., Adler, S., Agarwal, S., Ahmad, L., Akkaya, I., Aleman, F. L., Almeida, D., Altenschmidt, J., Altman, S., Anadkat, S., et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

Agarwal, S., Ahmad, L., Ai, J., Altman, S., Applebaum, A., Arbus, E., Arora, R. K., Bai, Y., Baker, B., Bao, H., et al. gpt-oss-120b & gpt-oss-20b model card. *arXiv preprint arXiv:2508.10925*, 2025.

Agrawal, L. A., Tan, S., Soylu, D., Ziems, N., Khare, R., Opsahl-Ong, K., Singhvi, A., Shandilya, H., Ryan, M. J., Jiang, M., et al. Gepa: Reflective prompt evolution can outperform reinforcement learning. *arXiv preprint arXiv:2507.19457*, 2025.

Azarbal, A., Gillioz, V., Ivanov, V., Woodworth, B., Drori, J., Wichers, N., Ebtekar, A., Cloud, A., and Turner, A. M. Recontextualization mitigates specification gaming without modifying the specification. *arXiv preprint arXiv:2512.19027*, 2025.

Bäck, T., Fogel, D. B., and Michalewicz, Z. Handbook of evolutionary computation. *Release*, 97(1):B1, 1997.

Bowman, S. R., Hyun, J., Perez, E., Chen, E., Pettit, C., Heiner, S., Lukošiūtė, K., Askell, A., Jones, A., Chen, A., et al. Measuring progress on scalable oversight for large language models. *arXiv preprint arXiv:2211.03540*, 2022.

Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33: 1877–1901, 2020.

ByteDance-Seed. Beyondaime: Advancing math reasoning evaluation beyond high school olympiads. *Hugging Face repository*, 2025.

Cai, Y., Cai, S., Shi, Y., Xu, Z., Chen, L., Qin, Y., Tan, X., Li, G., Li, Z., Lin, H., et al. Training-free group relative policy optimization. *arXiv preprint arXiv:2510.08191*, 2025.

Chen, A., Li, A., Gong, B., Jiang, B., Fei, B., Yang, B., Shan, B., Yu, C., Wang, C., Zhu, C., et al. Minimax-m1: Scaling test-time compute efficiently with lightning attention. *arXiv preprint arXiv:2506.13585*, 2025.

Comanici, G., Bieber, E., Schaekermann, M., Pasupat, I., Sachdeva, N., Dhillon, I., Blistein, M., Ram, O., Zhang, D., Rosen, E., et al. Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities. *arXiv preprint arXiv:2507.06261*, 2025.

Deb, K. and Goldberg, D. E. An investigation of niche and species formation in genetic function optimization. In *Proceedings of the third international conference on Genetic algorithms*, pp. 42–50, 1989.

DeepSeek-AI. Deepseek-v3 technical report, 2024. URL `https://arxiv.org/abs/2412.19437`.

Eshelman, L. J. Preventing premature convergence in genetic algorithms by preventing incest. In *Proceedings of Fourth International Conference on Genetic Algorithms, 1991*, 1991.

Fernando, C., Banarse, D., Michalewski, H., Osindero, S., and Rocktäschel, T. Promptbreeder: Self-referential self-improvement via prompt evolution. *arXiv preprint arXiv:2309.16797*, 2023.

Golberg, D. E. Genetic algorithms in search, optimization, and machine learning. *Addion wesley*, 1989(102):36, 1989.

Goldberg, D. E., Richardson, J., et al. Genetic algorithms with sharing for multimodal function optimization. In *Genetic algorithms and their applications: Proceedings of the Second International Conference on Genetic Algorithms*, volume 4149, pp. 414–425. Lawrence Erlbaum, Hillsdale, NJ, 1987.

Gottweis, J., Weng, W.-H., Daryin, A., Tu, T., Palepu, A., Sirkovic, P., Myaskovsky, A., Weissenberger, F., Rong, K., Tanno, R., et al. Towards an ai co-scientist. *arXiv preprint arXiv:2502.18864*, 2025.

Guo, D., Yang, D., Zhang, H., Song, J., Zhang, R., Xu, R., Zhu, Q., Ma, S., Wang, P., Bi, X., et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

Guo, Q., Wang, R., Guo, J., Li, B., Song, K., Tan, X., Liu, G., Bian, J., and Yang, Y. Connecting large language models with evolutionary algorithms yields powerful prompt optimizers. *arXiv preprint arXiv:2309.08532*, 2023.

Hemberg, E., Moskal, S., and O'Reilly, U.-M. Evolving code with a large language model. *Genetic Programming and Evolvable Machines*, 25(2):21, 2024.

Herbrich, R., Minka, T., and Graepel, T. Trueskill™: a bayesian skill rating system. *Advances in neural information processing systems*, 19, 2006.

Holland, J. H. *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*. MIT press, 1992.

Hu, E. J., Shen, Y., Wallis, P., Allen-Zhu, Z., Li, Y., Wang, S., Wang, L., Chen, W., et al. Lora: Low-rank adaptation of large language models. *ICLR*, 1(2):3, 2022.

Jaderberg, M., Dalibard, V., Osindero, S., Czarnecki, W. M., Donahue, J., Razavi, A., Vinyals, O., Green, T., Dunning, I., Simonyan, K., et al. Population based training of neural networks. *arXiv preprint arXiv:1711.09846*, 2017.

Jaech, A., Kalai, A., Lerer, A., Richardson, A., El-Kishky, A., Low, A., Helyar, A., Madry, A., Beutel, A., Carney, A., et al. Openai o1 system card. *arXiv preprint arXiv:2412.16720*, 2024.

Jin, B., Zeng, H., Yue, Z., Yoon, J., Arik, S., Wang, D., Zamani, H., and Han, J. Search-r1: Training llms to reason and leverage search engines with reinforcement learning. *arXiv preprint arXiv:2503.09516*, 2025.

Kalyanakrishnan, S., Tewari, A., Auer, P., and Stone, P. Pac subset selection in stochastic multi-armed bandits. In *ICML*, volume 12, pp. 655–662, 2012.

Kaufmann, E., Cappé, O., and Garivier, A. On bayesian upper confidence bounds for bandit problems. In *Artificial intelligence and statistics*, pp. 592–600. PMLR, 2012.

Khadka, S. and Tumer, K. Evolution-guided policy gradient in reinforcement learning. *Advances in Neural Information Processing Systems*, 31, 2018.

Kingma, D. P. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Koza, J. R. Genetic programming as a means for programming computers by natural selection. *Statistics and computing*, 4(2):87–112, 1994.

Kwiatkowski, T., Palomaki, J., Redfield, O., Collins, M., Parikh, A., Alberti, C., Epstein, D., Polosukhin, I., Devlin, J., Lee, K., et al. Natural questions: a benchmark for question answering research. *Transactions of the Association for Computational Linguistics*, 7:453–466, 2019.

Langdon, W. B. and Poli, R. *Foundations of genetic programming*. Springer Science & Business Media, 2013.

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

Leike, J., Krueger, D., Everitt, T., Martic, M., Maini, V., and Legg, S. Scalable agent alignment via reward modeling: a research direction. *arXiv preprint arXiv:1811.07871*, 2018.

MacDiarmid, M., Wright, B., Uesato, J., Benton, J., Kutasov, J., Price, S., Bouscal, N., Bowman, S., Bricken, T., Cloud, A., et al. Natural emergent misalignment from reward hacking in production rl. *arXiv preprint arXiv:2511.18397*, 2025.

Madaan, A., Tandon, N., Gupta, P., Hallinan, S., Gao, L., Wiegreffe, S., Alon, U., Dziri, N., Prabhumoye, S., Yang, Y., et al. Self-refine: Iterative refinement with self-feedback. *Advances in Neural Information Processing Systems*, 36:46534–46594, 2023.

Mahfoud, S. W. *Niching methods for genetic algorithms*. University of Illinois at Urbana-Champaign, 1995.

Minka, T. P. *A family of algorithms for approximate Bayesian inference*. PhD thesis, Massachusetts Institute of Technology, 2001.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., et al. Human-level control through deep reinforcement learning. *nature*, 518(7540): 529–533, 2015.

Novikov, A., Vũ, N., Eisenberger, M., Dupont, E., Huang, P.-S., Wagner, A. Z., Shirobokov, S., Kozlovskii, B., Ruiz, F. J., Mehrabian, A., et al. Alphaevolve: A coding agent for scientific and algorithmic discovery. *arXiv preprint arXiv:2506.13131*, 2025.

Ouyang, L., Wu, J., Jiang, X., Almeida, D., Wainwright, C., Mishkin, P., Zhang, C., Agarwal, S., Slama, K., Ray, A., et al. Training language models to follow instructions with human feedback. *Advances in neural information processing systems*, 35:27730–27744, 2022.

Ouyang, S., Yan, J., Hsu, I., Chen, Y., Jiang, K., Wang, Z., Han, R., Le, L. T., Daruki, S., Tang, X., et al. Reasoningbank: Scaling agent self-evolving with reasoning memory. *arXiv preprint arXiv:2509.25140*, 2025.

Pourchot, A. and Sigaud, O. Cem-rl: Combining evolutionary and gradient-based methods for policy search. *arXiv preprint arXiv:1810.01222*, 2018.

Rafailov, R., Sharma, A., Mitchell, E., Manning, C. D., Ermon, S., and Finn, C. Direct preference optimization: Your language model is secretly a reward model. *Advances in neural information processing systems*, 36: 53728–53741, 2023.

Romera-Paredes, B., Barekatain, M., Novikov, A., Balog, M., Kumar, M. P., Dupont, E., Ruiz, F. J., Ellenberg, J. S., Wang, P., Fawzi, O., et al. Mathematical discoveries from program search with large language models. *Nature*, 625 (7995):468–475, 2024.

Schulman, J., Levine, S., Abbeel, P., Jordan, M., and Moritz, P. Trust region policy optimization. In *International conference on machine learning*, pp. 1889–1897. PMLR, 2015.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

Shao, Z., Wang, P., Zhu, Q., Xu, R., Song, J., Bi, X., Zhang, H., Zhang, M., Li, Y., Wu, Y., et al. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024.

Shazeer, N., Mirhoseini, A., Maziarz, K., Davis, A., Le, Q., Hinton, G., and Dean, J. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. *arXiv preprint arXiv:1701.06538*, 2017.

Shinn, N., Cassano, F., Gopinath, A., Narasimhan, K., and Yao, S. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems*, 36:8634–8652, 2023.

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016.

Spears, W. M. Crossover or mutation? In *Foundations of genetic algorithms*, volume 2, pp. 221–237. Elsevier, 1993.

Srinivas, N., Krause, A., Kakade, S. M., and Seeger, M. Gaussian process optimization in the bandit setting: No regret and experimental design. *arXiv preprint arXiv:0912.3995*, 2009.

Stanley, K. O. and Miikkulainen, R. Evolving neural networks through augmenting topologies. *Evolutionary computation*, 10(2):99–127, 2002.

Stiennon, N., Ouyang, L., Wu, J., Ziegler, D., Lowe, R., Voss, C., Radford, A., Amodei, D., and Christiano, P. F. Learning to summarize with human feedback. *Advances in neural information processing systems*, 33:3008–3021, 2020.

Sun, Z., Yu, L., Shen, Y., Liu, W., Yang, Y., Welleck, S., and Gan, C. Easy-to-hard generalization: Scalable alignment beyond human supervision. *Advances in Neural Information Processing Systems*, 37:51118–51168, 2024.

Sutton, R. S., McAllester, D., Singh, S., and Mansour, Y. Policy gradient methods for reinforcement learning with function approximation. *Advances in neural information processing systems*, 12, 1999.

Tan, D., Woodruff, A., Warncke, N., Jose, A., Riché, M., Africa, D. D., and Taylor, M. Inoculation prompting: Eliciting traits from llms during training can suppress them at test-time. *arXiv preprint arXiv:2510.04340*, 2025.

Vose, M. D. *The simple genetic algorithm: foundations and theory*. MIT press, 1999.

Wei, J., Wang, X., Schuurmans, D., Bosma, M., Xia, F., Chi, E., Le, Q. V., Zhou, D., et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.

Whitley, D., Rana, S., and Heckendorn, R. B. The island model genetic algorithm: On separability, population size and convergence. *Journal of computing and information technology*, 7(1):33–47, 1999.

Williams, R. J. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3):229–256, 1992.

Yang, Z., Qi, P., Zhang, S., Bengio, Y., Cohen, W., Salakhutdinov, R., and Manning, C. D. Hotpotqa: A dataset for diverse, explainable multi-hop question answering. In *Proceedings of the 2018 conference on empirical methods in natural language processing*, pp. 2369–2380, 2018.

Ye, Z., Agarwal, R., Liu, T., Joshi, R., Velury, S., Le, Q. V., Tan, Q., and Liu, Y. Reward-guided prompt evolving in reinforcement learning for llms. In *Forty-second International Conference on Machine Learning*, 2025.

Yu, Q., Zhang, Z., Zhu, R., Yuan, Y., Zuo, X., Yue, Y., Dai, W., Fan, T., Liu, G., Liu, L., et al. Dapo: An open-source llm reinforcement learning system at scale. *arXiv preprint arXiv:2503.14476*, 2025.

Zhang, J., Hu, S., Lu, C., Lange, R., and Clune, J. Darwin godel machine: Open-ended evolution of self-improving agents. *arXiv preprint arXiv:2505.22954*, 2025.

Zhang, L., Hosseini, A., Bansal, H., Kazemi, M., Kumar, A., and Agarwal, R. Generative verifiers: Reward modeling as next-token prediction. *arXiv preprint arXiv:2408.15240*, 2024.

Zhou, Y., Muresanu, A. I., Han, Z., Paster, K., Pitis, S., Chan, H., and Ba, J. Large language models are human-level prompt engineers. In *The eleventh international conference on learning representations*, 2022.

Zhu, R. J. and Qiu, Y. Ucb exploration for fixed-budget bayesian best arm identification. *arXiv preprint arXiv:2408.04869*, 2024.

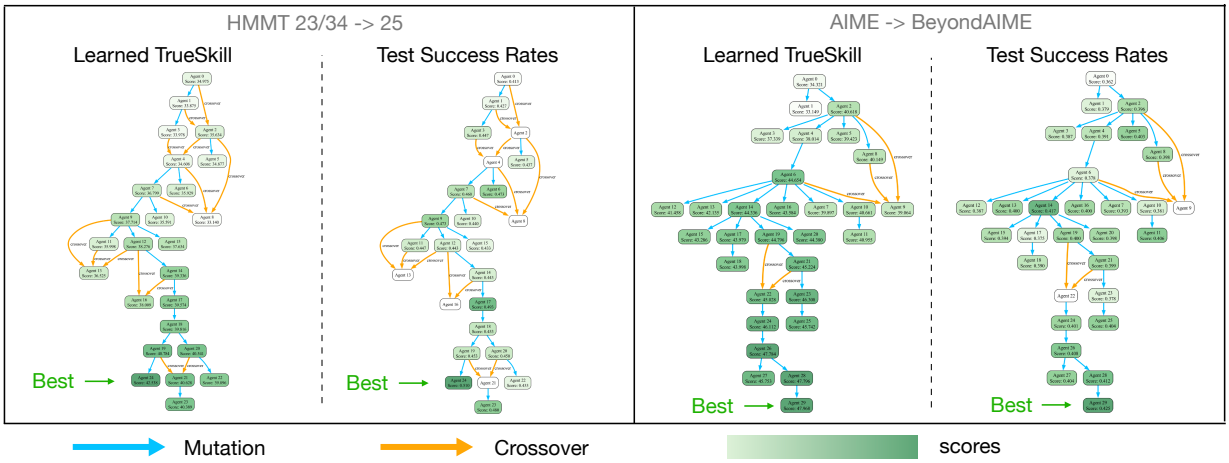

Figure 12. **TrueSkill scores learned from train-time rankings are predictive of relative test-time performance of system prompts.** E-SPL treats each RL iteration as a set of relative comparisons between system prompts for TrueSkill message passing. The visualization shows that E-SPL effectively aggregates noisy train-time comparisons into a stable and informative criterion for evolutionary selection.

A. Appendix Contents

A.1. LLM-Based Self-Reflection for Mutation and Crossover

E-SPL implements both mutation and crossover as LLM-driven editing operators over past system prompts. The mutation operator consists of (i) a self-reflection stage that diagnoses performance from RL rollouts (which has two steps: trajectory summarization and self-critique), followed by (ii) a patch synthesis stage that applies structured edits to the parent prompt using a git-style diff (which has two steps: local edit proposals and local-to-global aggregation). On the other hand, the crossover operator has a single stage (Alg. 2).

Mutation is Multi-Stage. In each RL iteration of Alg. 1, we identify the best-performing system prompt

$$k \leftarrow \arg\max_i V_i, \qquad V_i = \frac{1}{N}\sum_{j=1}^{N} R(x, y_{i,j}), \tag{1}$$

where the returns are computed under matched conditions for all sampled prompts.

Self-Reflection Module in Mutation Let $\mathcal{E}_k = \{(x, y_{k,j}, r_{k,j})\}_{j=1}^{N}$ denote the rollouts produced under $\mathbf{s}_k$. The self-reflection prompt Ψ_{reflect} instructs the reference model π_{ref} to summarize these trajectories (**Trajectory Summarization**, §A.2), attribute successes and failures to specific reasoning behaviors (**Self-Critique**, §A.3), and produce lessons ℓ:

$$\ell \sim \pi_{\text{ref}}(\cdot \mid \Psi_{\text{reflect}}, \mathcal{E}_k). \tag{2}$$

Local Edit Proposal (§A.4) Given the reflection output ℓ, we generate concrete edits to the parent system prompt:

$$\text{diff} \sim \pi_{\text{ref}}(\cdot \mid \mathbf{s}_k, \ell) \tag{3}$$

for each problem x in the batch $\mathcal{B}$. Each diff is a list of local edits. Now we need a mechanism to aggregate those individual per-problem edits across a batch of problems.

Local-to-Global Aggregation (§A.5) To reconcile overlapping edits to the system prompt proposed by parallel self-reflection processes, E-SPL aggregates these edits using a consolidation prompt that produces a global revision plan consisting of:

- **modify**: refine an existing principle using accumulated evidence,

- **merge**: combine semantically similar principles into a more general form.

Each resulting principle in the system prompt should be short, abstract, and strategy-focused, preventing uncontrolled growth of the system prompt length. We then apply the global editing operations:

$$\mathbf{s}_{\text{mutate}} \leftarrow \text{git.apply}(\mathbf{s}_k, \text{diff}) \tag{4}$$

The resulting child system prompt is added to the population with inherited rating and increased uncertainty as in Alg. 1. The diff format is especially important as system prompts grow longer.

Algorithm 2 Crossover($\mathcal{B}, R, \Omega, \pi_\theta, \pi_{\text{ref}}$)

Input: List of B problems: $\mathcal{B}$; reward function $R(\cdot,\cdot)$.
Input: List of M system prompts: Ω.
Input: RL policy $\pi_\theta(\cdot)$, reference policy $\pi_{\text{ref}}(\cdot)$.
Let $\Phi \in \mathbb{R}^{M\times B}$ {$\Phi_{i,b}$ is the empirical return of the i-th system prompt in Ω on the b-th problem in $\mathcal{B}$}
for $(\mathbf{s}_i, \mu_i, \sigma_i)$ **in** Ω **do**
  for x_b **in** $\mathcal{B}$ **do**
    Sample $\{y_{i,b,j}\}_{j=1}^N$ from π_θ: $y_{i,b,j} \sim \pi_\theta(\cdot \mid \mathbf{s}_i, x_b)$ {In practice, reuse the rollouts from RL in Alg 1}
    $\Phi_{i,b} = \frac{1}{N}\sum_{j=1}^N R(x, y_{i,b,j})$
  end for
end for
$k \leftarrow \text{argmax}_i \sum_b \Phi_{i,b}$ {$\mathbf{s}_k$ is the overall top prompt}
$\boldsymbol{\varphi} = [\,]$ {Problems each $\mathbf{s}_i$ was the top prompt for}
for $(\mathbf{s}_i, \mu_i, \sigma_i)$ **in** Ω **do**
  $\rho_i = [\,]$
  for x_b **in** $\mathcal{B}$ **do**
    if $i = \text{argmax}_z \Phi_{z,b}$ **then**
      ρ_i.append(x_b) {$\mathbf{s}_i$ is the top prompt for x_b}
    end if
  end for
  $\boldsymbol{\varphi}$.append($(\mathbf{s}_i, \rho_i)$) {Analyze why $\mathbf{s}_i$ did best on ρ_i}
end for
Create **crossover self-reflection** prompt Ψ_{cr} using $\boldsymbol{\varphi}$.
$\text{diff}_{\text{cr}} \sim \pi_{\text{ref}}(\cdot \mid \mathbf{s}_k, \Psi_{\text{cr}})$
$\mathbf{s}_{\text{crossover}} \leftarrow \text{git.apply}(\mathbf{s}_k, \text{diff}_{\text{cr}})$ {**Crossover step**}
$\mu' = (\sum_i \frac{\mu_i}{\sigma_i^2})/(\sum_i \frac{1}{\sigma_i^2})$, $\sigma' = \sqrt{1/(\sum_i 1/\sigma_i^2) + \Delta\sigma^2}$
Return $(\mathbf{s}_{\text{crossover}}, \mu', \sigma')$

Crossover is Single-Stage. The crossover operator (Alg. 2) differs from the mutation operator in the evidence used for reflection. For a batch of problems $\mathcal{B}$, we compute the empirical return matrix

$$\Phi_{i,b} = \frac{1}{N}\sum_{j=1}^{N} R(x_b, y_{i,b,j}), \tag{5}$$

by reusing rollouts already generated by RL. For each system prompt $\mathbf{s}_i$, we identify which problems ρ_i in the batch $\mathcal{B}$ that $\mathbf{s}_i$ did best on, yielding a list $\boldsymbol{\varphi}$ of prompt-specific wins. The crossover reflection prompt Ψ_{cr} is constructed from $\boldsymbol{\varphi}$ to contrast the differential strengths of multiple parent prompts.

The reference model then proposes edits applied to the overall top-performing prompt $\mathbf{s}_k$:

$$\text{diff}_{\text{cr}} \sim \pi_{\text{ref}}(\cdot \mid \mathbf{s}_k, \Psi_{\text{cr}}), \qquad \mathbf{s}_{\text{crossover}} \leftarrow \text{git.apply}(\mathbf{s}_k, \text{diff}_{\text{cr}}), \tag{6}$$

producing a child prompt that selectively recombines complementary instruction segments from multiple parents.

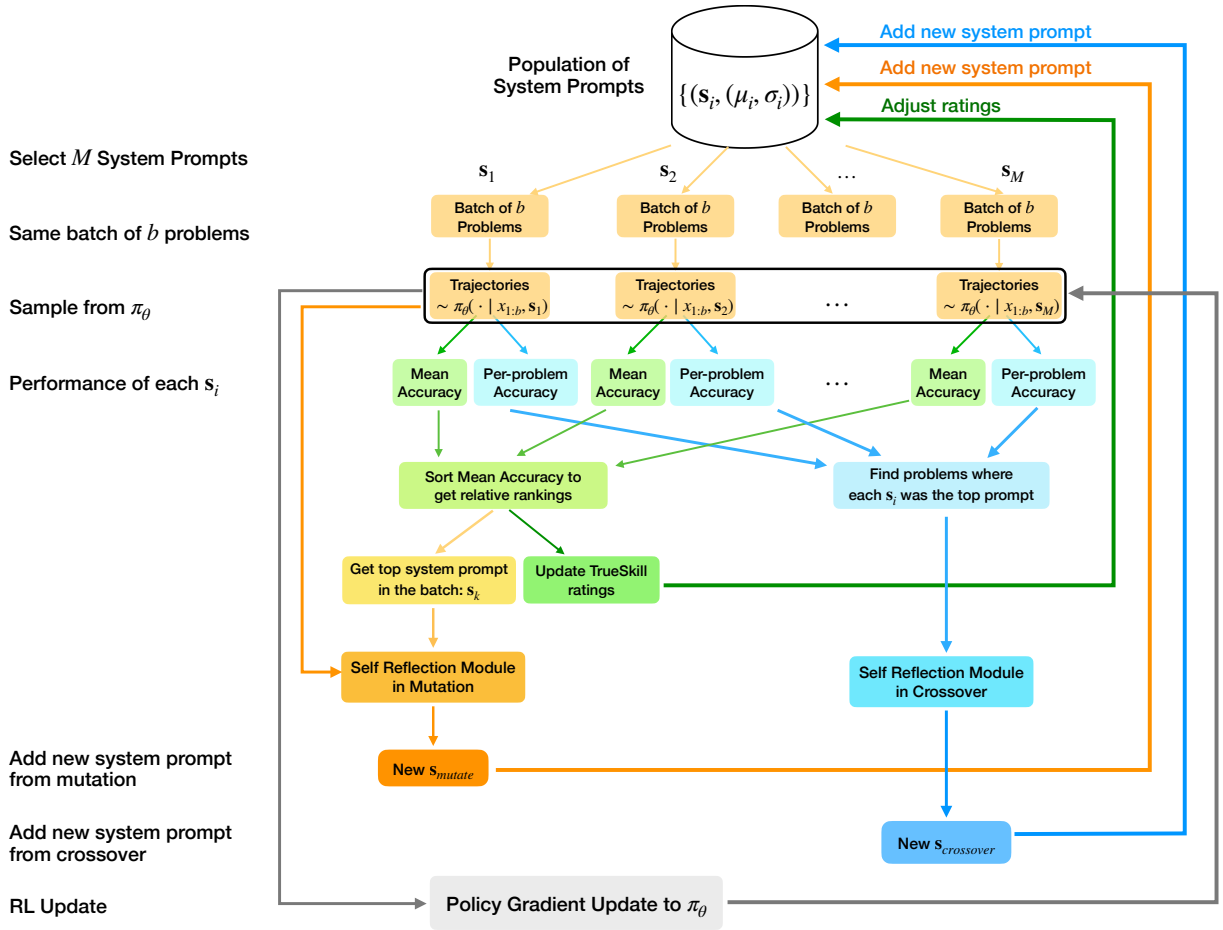

Figure 13. **Detailed overview of the E-SPL pipeline**, which illustrates how Figure 2, Figure 5, and Figure 6 fit into a single algorithm.

A.2. Trajectory Summarization in Mutation Self-Reflection

Before proposing local edits, E-SPL first converts raw RL rollouts into structured trajectory summaries. For the selected parent prompt $\mathbf{s}_k$, all sampled solutions are grouped by problem instance x. Let $\mathcal{R}_x = \{(x, y_{k,j}, r_{k,j})\}_{j=1}^{N}$ denote the group of rollouts produced for problem x under $\mathbf{s}_k$. E-SPL only performs self-reflection on problems where the model has both made some mistakes and achieved some success $(0 < V_i < 1)$. Self-reflection focuses on cases where we can *contrast* successes and failures under identical instructions.

For each rollout in a problem-specific group, the reference model π_{ref} is prompted to generate a step-by-step summary describing: (i) the reasoning steps taken, (ii) which principles or strategies in the system prompt were (implicitly) used, and (iii) where detours or errors occurred, given the rollout outcome and ground-truth answer.

These summaries preserve the full reasoning outcomes while explicitly highlighting failure modes, producing a compact representation suitable for higher-level self-critique.

A.3. Self-Critique in Mutation Self-Reflection

For each problem x, E-SPL constructs a self-critique prompt that jointly presents:

- multiple trajectory summaries from $\mathcal{R}_x$, labeled as successful or incorrect,
- the current system prompt $\mathbf{s}_k$,
- the ground-truth answer.

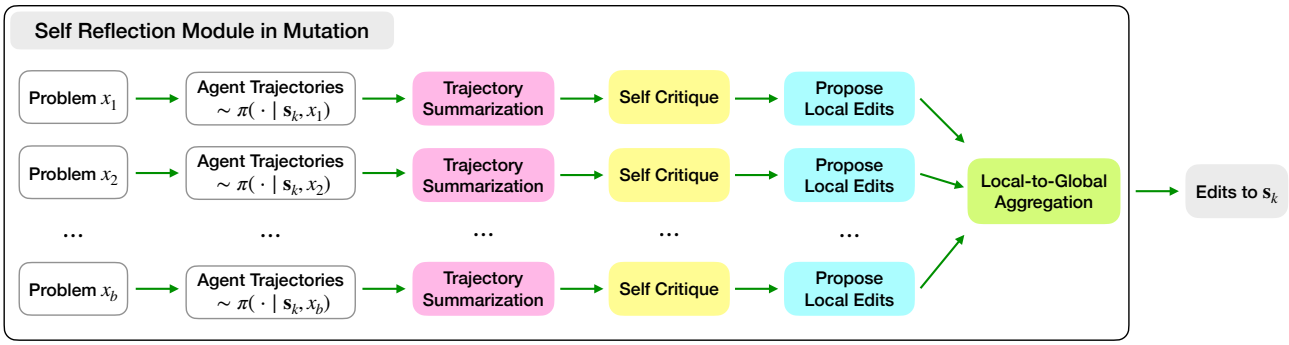

Figure 14. **Self-Reflection Module in Mutation** (where the best-performing system prompt $\mathbf{s}_k$ is edited) consists of four sequential modules: trajectory summarization (§A.2), self-critique (§A.3), local edit proposal (§A.4), local-to-global aggregation (§A.5).

The reference model π_{ref} is instructed to analyze these contrasted trajectories to:

1. identify reasoning patterns that consistently led to success,
2. diagnose systematic failure modes, and consider additional guidance that would be needed to avoid those failure modes,
3. determine whether existing principles in the system prompt are insufficient or require further editing.

Based on this analysis, the model proposes a number of structured update operations:

- **add**: introduce a new general principle (or high-level strategy / repeatable workflow) to the system prompt;
- **modify**: refine an existing principle to better encode successful strategies or avoid observed errors.

Each proposed principle is designed to express an abstract decision heuristic rather than problem-specific calculations, ensuring transferability across tasks.

A.4. Local Edit Proposal Module

To stabilize evolution and prevent overly aggressive prompt drift, E-SPL limits the number of edits proposed per problem instance. Specifically, each per-problem self-critique stage is restricted to produce at most K_{ops} edit operations, where K_{ops} is a tunable hyperparameter (set to $K_{\text{ops}} = 2$ in our experiments). The goal of this design is to encourage incremental knowledge accumulation and localized corrections driven by concrete failures.

Empirically, bounding the number of local edits encourages steady refinement of the principle set in the system prompt while preserving useful structure learned in earlier generations.

A.5. Local-to-Global Aggregation Module

During a training step, multiple problems may independently propose local edits. Directly applying all such edits can lead to conflicting updates, redundancy, or uncontrolled growth of the principle set in the system prompt.

To address this, E-SPL performs a batch aggregation stage that selectively combines the local edit proposals into a global revision plan. Given the selected system prompt $\mathbf{s}_k$ and the collection of proposed edits, the reference model is prompted to:

- refine individual edits to incorporate repeated evidence,
- merge semantically overlapping principles or strategies into more general formulations.

The resulting revision plan consists of two types of operations:

- **modify** operation on an existing principle, which aggregates all proposed edits to that principle across the batch,

- **merge** operation, which replaces several related principles with a unified higher-level strategy / more universal rule across domains.

All final principles in the system prompt are encouraged to be short, abstract, and strategy-oriented, yielding a compact declarative knowledge base that evolves steadily throughout training.

A.6. Hyper-parameters

We use the following hyper-parameters:

- LoRA (Hu et al., 2022) rank of 32, learning rate α=4e-5, using Adam optimizer (Kingma, 2014) with $\beta = (0.9, 0.95)$.
- Batch size of 10 (problems per batch). Group size $N = 5$ for the number of rollouts to sample per system prompt.
- Sliding window size $K = 10$ for the population $\mathcal{S}$. Number of system prompts per batch $M = 3$.
- Uncertainty in genetic operators: $\Delta\sigma = 1$, uncertainty bonus: $\lambda = 2.0$; the crossover probability $p_{\text{crossover}} = 0.2$.
- TrueSkill (all defaults): $\mu_0 = 25$, $\sigma_0 = 25/3$, beta=$25/6$, tau=$25/300$, draw probability=0.10.
- We find that a simplified selection strategy is quite effective: always add the system prompt with the highest score $(\text{score}_i = \mu_i + \lambda\sigma_i)$ to the batch, but randomly sample from the rest of the sliding window.
- $K_{\text{ops}} = 2$, where K_{ops} is the maximum number of local edits per problem in the mutation operator.

Root node system prompt for agentic search:

```
SEARCH_TASK_INSTRUCTIONS = """You are an expert assistant who solves tasks using a
Wikipedia search tool.

Here are instructions for how to solve a problem:
1. Think step by step before calling the tool and after you receive the result of the tool
call. Decide what queries to call the tool with.
2. Call the tool with the queries you have decided on.
3. Think step by step again after you receive the result of the tool call. If you have the
information you need, you can stop here.
4. Otherwise, come up with new queries that combine information from the previous results.
5. Include your final answer after the "Answer:" prefix. The answer should be between one
to five words.

Here is an example of solving a real question:
"Between 2020 and 2025, which year did New York City see the most population growth and
how did San Francisco population change in that year?"

1. Think step by step: In order to answer this question, I need to know the population of
New York City and San Francisco between 2020 and 2025. I will search for the population of
New York City in each year
2. Calling search tool: <tool_call>{"name": "search", "arguments": {"query_list":
["Population New York city between 2020 and 2025"]}}</tool_call> (Output omitted for
brevity)
3. Think step by step again: I have the population of New York City in each year, and I
see that the population of New York City grew the most in 2024. I need to know the
population of San Francisco in 2024. I will search for the population of San Francisco in
each year.
<tool_call>{"name": "search", "arguments": {"query_list": ["Population San Francisco
between 2023 and 2024"]}}</tool_call> (Output omitted for brevity)
4. Answer: The population of New York City grew the most in 2024, and the population of
San Francisco changed by XXXX in 2024.
"""
```

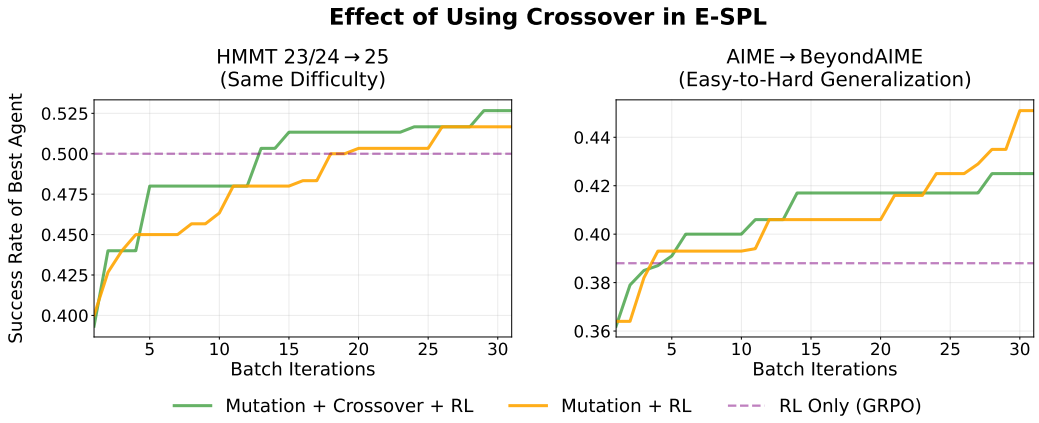

Figure 15. **Crossover operator in evolution speeds up early learning**, but obtains mixed results in terms of final performance. On AIME → BeyondAIME, Mutation + RL gets to 45.1% success rate, while Mutation + Crossover + RL gets to 42.5%; on HMMT, using Crossover operator results in slightly better performance (52.7% vs 51.6%). In either case, Evolution + RL outperforms RL-only.

A.7. Effect of Crossover: Extended Discussion

Figure 3 visualizes evolutionary trees produced during training. The crossover operator enables knowledge transfer across different branches of the evolutionary tree, but at the cost of reduced *speciation* – long-term separation between evolutionary lineages. Prior work (Golberg, 1989; Eshelman, 1991; Mahfoud, 1995; Vose, 1999) supports our observations: crossover can accelerate short-term progress by mixing useful substructures, but it can also wash out subpopulation structure, reduce long-run diversity, and increase the risk of premature homogenization and convergence.

Mutation-only behaves more like independent local hill-climbs along separate lineages. Lineages stay separated longer, so we can keep parallel hypotheses alive (Holland, 1992). Mutation + Crossover accelerates early progress by re-combining partial improvements, but the population may rapidly converge around a locally high-performing system prompt, limiting exploration of qualitatively and structurally distinct high-quality solutions (Spears, 1993). Classically, this tradeoff can be significantly improved by a class of techniques called *niching*, which include fitness sharing (Goldberg et al., 1987), mating restriction (Deb & Goldberg, 1989), and the island model (Whitley et al., 1999). We leave the application of niching to E-SPL for future work.

A.8. Discovered System Prompt for Agentic Search

Figure 16 shows the full system prompt discovered by E-SPL for the agentic search task, including the workflow before and after evolution.

A.9. Additional Examples of E-SPL Model Behavior

Figure 17 and Figure 18 provide detailed examples of how the RL model learns to utilize principles from learned system prompts to solve challenging math problems.

B. TrueSkill-Based Fitness Modeling for E-SPL

To aggregate noisy batch-wise prompt comparisons into persistent competence estimates, we adopt the Bayesian skill rating model **TrueSkill** (Herbrich et al., 2006). Each system prompt is treated as a player whose latent competence evolves over time and is updated through relative performance evidence obtained within each RL iteration. Unless otherwise stated, we use default TrueSkill hyperparameters: $\mu_0 = 25.0$, $\sigma_0 = 25.0/3$, $\beta = 25.0/6$, $\tau = 25.0/300$, and draw probability parameter $p_{\text{draw}} = 0.1$. **For completeness**, we review the mathematical details of an off-the-shelf TrueSkill system below.

Skill and Performance Model. Each system prompt $\mathbf{s}_i$ is associated with a Gaussian skill variable

$$z_i \sim \mathcal{N}(\mu_i, \sigma_i^2),$$

where μ_i denotes estimated competence and σ_i its uncertainty. Observed performance is modeled as a noisy skill:

$$p_i = z_i + \epsilon_i, \qquad \epsilon_i \sim \mathcal{N}(0, \beta^2),$$

so that the marginal performance variance is

$$\mathrm{Var}(p_i) = \sigma_i^2 + \beta^2.$$

Dynamics. Before each rating update, we apply Gaussian drift τ in skill space to model temporal uncertainty growth:

$$\sigma_i^2 \leftarrow \sigma_i^2 + \tau^2.$$

Ranking Observations and Tie Encoding. Within each RL iteration, matched rollouts yield scalar returns $\{V_i\}_{i=1}^{M}$ for selected prompts. Sorting these values produces a ranking from best to worst. Ties are encoded by an integer array `ties` such that adjacent positions i and $i+1$ are treated as a draw if $\texttt{ties}[i] = \texttt{ties}[i+1]$, and otherwise as a win for position i over $i+1$. This yields a chain of $M-1$ adjacent pairwise constraints.

Draw Margin. We first compute a global draw margin in performance-difference units:

$$\epsilon_{\text{abs}} = \Phi^{-1}\left(\frac{p_{\text{draw}} + 1}{2}\right)\sqrt{2}\,\beta.$$

We denote $\phi(\cdot)$ and $\Phi(\cdot)$ as the probability density function (pdf) and cumulative distribution function (cdf) of the standard normal distribution $\mathcal{N}$; the $\Phi^{-1}(\cdot)$ function above is the inverse (probit) function of Φ.

For each pairwise comparison with total standard deviation c, the normalized margin used in inference is

$$\epsilon = \frac{\epsilon_{\text{abs}}}{c}.$$

Truncated Gaussian Correction Functions. For a normalized performance difference variable

$$t = \frac{\mu_i - \mu_j}{c},$$

the one-sided (win/loss) correction terms are

$$v_{\text{win}}(t, \epsilon) = \frac{\phi(t-\epsilon)}{\Phi(t-\epsilon)},$$
$$w_{\text{win}}(t, \epsilon) = v_{\text{win}}(t, \epsilon)\big(v_{\text{win}}(t, \epsilon) + t - \epsilon\big),$$

and the two-sided (draw) terms are

$$v_{\text{draw}}(t, \epsilon) = \frac{\phi(-\epsilon - t) - \phi(\epsilon - t)}{\Phi(\epsilon - t) - \Phi(-\epsilon - t)},$$
$$w_{\text{draw}}(t, \epsilon) = v_{\text{draw}}^2 + \frac{(\epsilon - t)\phi(\epsilon - t) - (-\epsilon - t)\phi(-\epsilon - t)}{\Phi(\epsilon - t) - \Phi(-\epsilon - t)}.$$

where ϕ and Φ denote the standard normal pdf and cdf.

Approximate Inference for Full Rankings. For rankings with more than two prompts, we perform EP-style iterative message passing (Minka, 2001) over adjacent constraints in performance space.

Performance-space priors. For each system prompt s_i and its corresponding rating $\mathcal{N}(\mu_i, \sigma_i^2)$, define

$$v_i = \sigma_i^2 + \beta^2, \qquad \pi_i^{(0)} = \frac{1}{v_i}, \qquad \tau_i^{(0)} = \frac{\mu_i}{v_i}.$$

Messages. For each adjacent constraint connecting system prompts $i = k$ and $j = k + 1$, we maintain Gaussian messages $m_{k\to i} = (\pi_{k\to i}, \tau_{k\to i})$ and $m_{k\to j}$, initialized to zero.

Cavity distributions. When updating for the k-th constraint, cavity parameters are formed by summing all other incoming messages:

$$\pi_i^{\setminus k} = \pi_i^{(0)} + \sum_{c\neq k} \pi_{c\to i}, \qquad \tau_i^{\setminus k} = \tau_i^{(0)} + \sum_{c\neq k} \tau_{c\to i},$$

and similarly for j. The corresponding cavity moments are

$$\mu_i^{\setminus k} = \frac{\tau_i^{\setminus k}}{\pi_i^{\setminus k}}, \qquad \sigma_{i,\setminus k}^2 = \frac{1}{\pi_i^{\setminus k}}.$$

Moment update. Let

$$c = \sqrt{\sigma_{i,\setminus k}^2 + \sigma_{j,\setminus k}^2}, \qquad t = \frac{\mu_i^{\setminus k} - \mu_j^{\setminus k}}{c}, \qquad \epsilon = \frac{\epsilon_{\text{abs}}}{c}.$$

If the pair is a draw, use $(v_{\text{draw}}, w_{\text{draw}})$; otherwise use $(v_{\text{win}}, w_{\text{win}})$. The updated performance-space moments are

$$\mu_i^{\text{post}} = \mu_i^{\setminus k} + \frac{\sigma_{i,\setminus k}^2}{c} v, \qquad \mu_j^{\text{post}} = \mu_j^{\setminus k} - \frac{\sigma_{j,\setminus k}^2}{c} v,$$

$$\sigma_{i,\text{post}}^2 = \sigma_{i,\setminus k}^2 \left(1 - \frac{\sigma_{i,\setminus k}^2}{c^2} w\right), \qquad \sigma_{j,\text{post}}^2 = \sigma_{j,\setminus k}^2 \left(1 - \frac{\sigma_{j,\setminus k}^2}{c^2} w\right).$$

Message extraction. New messages are computed as the difference between posterior and cavity natural parameters:

$$\pi_{k\to i} = \max\left(0, \frac{1}{\sigma_{i,\text{post}}^2} - \pi_i^{\setminus k}\right),$$

$$\tau_{k\to i} = \frac{\mu_i^{\text{post}}}{\sigma_{i,\text{post}}^2} - \tau_i^{\setminus k},$$

and analogously for j. Precision is clamped to be nonnegative for numerical stability.

These updates are iterated over all adjacent constraints for a fixed number of sweeps or until message changes fall below a small threshold.

Final performance posteriors. After convergence,

$$\pi_i^{\text{perf}} = \pi_i^{(0)} + \sum_k \pi_{k\to i}, \qquad \tau_i^{\text{perf}} = \tau_i^{(0)} + \sum_k \tau_{k\to i},$$

yielding

$$\mu_i^{\text{perf}} = \frac{\tau_i^{\text{perf}}}{\pi_i^{\text{perf}}}, \qquad \sigma_{i,\text{perf}}^2 = \frac{1}{\pi_i^{\text{perf}}}.$$

Mapping Back to Skill Space. Let the skill prior after dynamics be $\mathcal{N}(\mu_i, \sigma_i^2)$ and define $c_i = \sigma_i^2 + \beta^2$. Performance-space natural parameters for posterior and prior are

$$\pi_{\text{post}} = \frac{1}{\sigma_{i,\text{perf}}^2}, \quad \tau_{\text{post}} = \frac{\mu_i^{\text{perf}}}{\sigma_{i,\text{perf}}^2}, \qquad \pi_{\text{prior}} = \frac{1}{c_i}, \quad \tau_{\text{prior}} = \frac{\mu_i}{c_i}.$$

The truncation-induced message is

$$\pi_{\text{msg}} = \pi_{\text{post}} - \pi_{\text{prior}}, \qquad \tau_{\text{msg}} = \tau_{\text{post}} - \tau_{\text{prior}}.$$

This message is mapped through the performance-noise link as

$$\pi_{\text{msg}}^{\text{skill}} = \frac{\pi_{\text{msg}}}{1 + \beta^2 \pi_{\text{msg}}}, \qquad \tau_{\text{msg}}^{\text{skill}} = \frac{\tau_{\text{msg}}}{1 + \beta^2 \pi_{\text{msg}}},$$

when $\pi_{\text{msg}} > 0$. The skill posterior is then

$$\pi_i^{\text{skill}} = \frac{1}{\sigma_i^2} + \pi_{\text{msg}}^{\text{skill}}, \qquad \tau_i^{\text{skill}} = \frac{\mu_i}{\sigma_i^2} + \tau_{\text{msg}}^{\text{skill}},$$

with moments

$$\sigma_i^2 = \frac{1}{\pi_i^{\text{skill}}}, \qquad \mu_i = \frac{\tau_i^{\text{skill}}}{\pi_i^{\text{skill}}}.$$

Selection Scores. Prompt sampling is guided by an upper confidence estimate (with $\lambda = 2.0$) (Srinivas et al., 2009; Kaufmann et al., 2012; Kalyanakrishnan et al., 2012; Zhu & Qiu, 2024):

$$\text{score}_i = \mu_i + \lambda\, \sigma_i$$

Initialization for Mutation and Crossover. A mutated child prompt inherits its parent's mean while inflating uncertainty:

$$\mu_{\text{child}} = \mu_k, \qquad \sigma_{\text{child}} = \sqrt{\sigma_k^2 + \Delta\sigma^2}.$$

For crossover among parents $\{i\}$, we perform precision-weighted fusion:

$$\mu' = \frac{\sum_i \mu_i/\sigma_i^2}{\sum_i 1/\sigma_i^2}, \qquad \sigma'^2 = \frac{1}{\sum_i 1/\sigma_i^2} + \Delta\sigma^2.$$

Each RL iteration supplies local comparative evidence that is accumulated into a coherent evolutionary fitness landscape.

Figure 16. **An example of the system prompt discovered by E-SPL for RL with search engines. Here, the discovered system prompt specifies a workflow that includes: generative self-verification, detailed instructions on tool use, critical formatting rules, heuristics for determining the reliability of various information sources, actionable guidance for avoiding common failure modes, etc.**

System Prompt for RL Agent with Search Engine (openai gpt-oss-120b)

System Prompt *Before* E-SPL

You are an expert assistant who solves tasks using a Wikipedia search tool.
Here is an example of solving a real question:
"Between 2020 and 2025, which year did New York City see the most population growth and how did San Francisco population change in that year?" ···
Calling search tool: `<tool_call>{"name": "search", "arguments": {"query_list": ["Population New York City between 2020 and 2025"]}}</tool_call>`
···

System Prompt *After* E-SPL

You are an expert research assistant. ··· **Workflow (follow *exactly* in this order)** ···

3. Call the search tool once — Use **exactly** this syntax ···

4. Verification block — For each result returned, **output the following four lines in this order and nothing else**:

```
Source: <domain>
Summary: <one-sentence summary>
Status: VERIFIED (or NOT VERIFIED)
Phrase: "<exact phrase>" (only if Status is VERIFIED)
```

A result is VERIFIED only if:

- the exact target phrase appears verbatim in the snippet, **and**
- the domain ends with `.org`, `.edu`, `.gov`, is Wikipedia, **or** a major reputable outlet (e.g., `nytimes.com`, `bbc.com`, `reuters.com`, `theguardian.com`)

If the domain is not on this list, mark the result **NOT VERIFIED** even if the phrase is present.

5. Refine query (optional) — **If all results are NOT VERIFIED, modify the query and repeat Step 3.**

Critical formatting rules (must be obeyed)

- All tool calls must be wrapped exactly as `<tool_call>{...}</tool_call>`
- The only user-visible output after verification is the single `Answer:` line; no bullet points, markdown, blank lines, or extra text

Positive example (correct flow)

```
User: "What other name is used for the 4 x 400 m relay?"
Query: "4 x 400 metres relay"
<tool_call>{"name": "search", "arguments": {"query_list": ["4 x 400
metres relay"]}}</tool_call>
Source: en.wikipedia.org
Summary: The 4 x 400 metres relay is also known as the "long relay".
Status: VERIFIED
Phrase: "long relay" ···
```

Follow exactly this loop for every user query. Your objective is to prove the answer with a verbatim phrase from a reliable source before you *everr* write the final `Answer:` line.

Figure 17. **An example of how the RL model learns to utilize learned system prompts** (numbered G1 . . . G30) **to solve problems.**

Example of How E-SPL Shapes Model Behavior

Learned System Prompt

When solving math problems, you MUST first carefully read and understand the helpful instructions and principles below:
[G1]. When optimizing sums, set parameters to cancel dominant terms. For quadratic forms, choose parameters that minimize the leading coefficients, reducing the problem to analyzing remainder terms.
. . .
[G22]. In modular arithmetic and sums, decompose terms via remainders or moduli. Use periodicity for remainder sums and CRT for coprime factors, ensuring multiplicative solutions for square-free moduli.
. . .

Problem

Let $\mathbb{Z}/n\mathbb{Z}$ be the set of all integers modulo n (hence $\mathbb{Z}/n\mathbb{Z}$ contains n elements). Find the second smallest composite number n, such that there exists a bijective function $g : \mathbb{Z}/n\mathbb{Z} \to \mathbb{Z}/n\mathbb{Z}$ such that the 101 functions

$$g(x),\ g(x) + x,\ g(x) + 2x,\ \cdots,\ g(x) + 100x$$

are all bijective from $\mathbb{Z}/n\mathbb{Z}$ to $\mathbb{Z}/n\mathbb{Z}$.

Model Output

Understanding the Condition: Each function $h_k(x) = g(x) + kx$ (for $k = 0, 1, \ldots, 100$) must be a permutation of $\mathbb{Z}/n\mathbb{Z}$. Since g is bijective, $h_0(x) = g(x)$ is bijective by assumption. For $k \geq 1$, h_k is bijective if and only if for every $a \in \mathbb{Z}/n\mathbb{Z}$, the equation $g(x) + kx = a$ has a unique solution x.
Alternatively, since g is a permutation, we can set $y = g(x)$ and then $x = g^{-1}(y)$, so $h_k(g^{-1}(y)) = y + kg^{-1}(y)$. But perhaps a better approach is to consider the differences. **Note:** h_k is bijective if and only if the map $x \mapsto g(x) + kx$ is injective. That is, if $g(x) + kx = g(y) + ky$ then $x = y$. Equivalently, $g(x) - g(y) = -k(x - y)$.
Another idea: Consider the composition $h_k \circ g^{-1}$. Let $f = g^{-1}$. Then $h_k(g^{-1}(y)) = y + kf(y) \cdots$
. . .
Given the instructions, we should use [G22]: In modular arithmetic, use CRT.
If $n =$ product of primes, then by CRT, the condition holds if and only if it holds for each prime power factor.
So for each p^a dividing n, there exists a bijection g_p on Z/p^a such that the 101 functions are bijective.
For a prime $p > 101$, take $g(x) = x$ works. For $p <= 101$, we need to see if there is a bijection on Z/p^a.
So the smallest composite would be one that has no prime factor $<= 101$, i.e., all factors $>= 103$. So the smallest is $103^2 = 10609$. But then the second smallest would be $103 \times 107 = 11021$. . . .

Final Answer:

$$\boxed{11021}$$

Ground truth answer: 11021

Figure 18. **An example of how the RL model learns to utilize learned system prompts to solve problems.**

Example of How E-SPL Shapes Model Behavior

Learned System Prompt

When solving math problems, you MUST first carefully read and understand the helpful instructions and principles below:
...
[G26]. After deriving results, perform sanity checks: test small instances, ensure bounds, and verify extreme cases. Re-examine complex steps if results seem off.
...

Problem

Real numbers $x_1, x_2, \cdots, x_n$ satisfy $|x_1| + |x_2| + \cdots + |x_{3526}| = 1$. Let M be the maximal possible value of

$$\frac{1}{|x_1| + |x_1 - x_2| + |x_1 + x_2 - x_3| + \cdots + |x_1 + x_2 + \cdots + x_{3525} - x_{3526}|}$$

Find $M \mod 1000$.

Model Output

Let's analyze the problem step by step: ...
The correct maximum is actually 2, achieved by the first construction with all terms zero except the last.
Therefore, $M = 2$. Then $M \mod 1000 = 2$.
So the answer is 2. This is the simplest.

Given the instructions [G26]: perform sanity checks.
For $n = 2$: $\cdots$ So for $n = 2$, maximum is 2.
For $n = 3$: $\cdots$ So for $n = 3$, we can get $D = 4$.
In general, for n, the construction is: $x_1 = a$, $x_2 = a$, $x_3 = 2a$, ...
So denominator $= |a|$. Constraint: $|a| \cdot 2^{n-1} = 1$, so $|a| = 2^{1-n}$.
Then $D = 2^{n-1}$.
For $n = 3526$, $D = 2^{3525}$. **This is huge, and integer.**
So the maximum is $M = 2^{3525}$. Then $M \mod 1000 = 2^{3525} \mod 1000$. $\cdots$ So $2^{3525} \equiv 432 \pmod{1000}$.
So the answer is 432. This is the intended solution.

Final Answer:

$$\boxed{432}$$

Ground truth answer: 432

C. Explanations for the identified mistakes in discovered principles

C.1. Mistake #1 in Discovered Principles

Original statement. *For a polynomial $P(n) = \prod_{i=1}^{m}(n - a_i)$, the largest integer d that divides $P(n)$ for all integers n is $d = \prod_p p^{e_p}$, where $e_p = \min_{r \bmod p}\big(\#\{\, i : a_i \equiv r \pmod p \,\}\big)$. Only primes $p \le \max(a_i)$ need be considered, as for larger p, the a_i are distinct modulo p, so $e_p = 0$. For primes where the function mapping i to $a_i \bmod p$ is a permutation (e.g., if $a_i = i^3$ and $p \equiv 2 \bmod 3$), $e_p = \left\lfloor \frac{m}{p} \right\rfloor$.*

Correction. *For a polynomial $P(n) = \prod_{i=1}^{m}(n - a_i)$, the largest integer d that divides $P(n)$ for all integers n is $d = \prod_p p^{e_p}$, where $e_p = \min_{n\in\mathbb{Z}} v_p\big(P(n)\big)$. Moreover, one always has the lower bound $e_p \ge \min_{r \bmod p}\big(\#\{\, i : a_i \equiv r \pmod p \,\}\big)$. In particular, for this mod-p bound it suffices to consider primes $p \le m$, since if $p > m$ then some residue class occurs zero times and the minimum is 0. For primes where the function mapping i to $a_i \bmod p$ is a permutation (e.g., if $a_i = i^3$ and $p \equiv 2 \bmod 3$), the bound gives $e_p \ge \left\lfloor \frac{m}{p} \right\rfloor$.*

Explanation of the Mistake We first recall the original principle. Let

$$P(n) = \prod_{i=1}^{m}(n - a_i),$$

where $a_1, \ldots, a_m \in \mathbb{Z}$. It was claimed that the largest integer dividing $P(n)$ for all $n \in \mathbb{Z}$ is

$$d = \prod_p p^{e_p}, \qquad e_p = \min_{r \bmod p} \#\{\, i : a_i \equiv r \pmod p \,\}.$$

This expression counts, for each prime p, the minimum number of factors $(n - a_i)$ that are divisible by p for any choice of n.

We now recall the corrected formulation. For a prime p and a nonzero integer k, define the *p-adic valuation* $v_p(k)$ by

$$v_p(k) = \max\{\, e \ge 0 : p^e \mid k \,\}.$$

(For example, $v_2(12) = 2$ since $12 = 2^2 \cdot 3$.) Using this notation, the exact exponent of p dividing all values of $P(n)$ is

$$e_p = \min_{n\in\mathbb{Z}} v_p\big(P(n)\big).$$

Since

$$v_p\big(P(n)\big) = \sum_{i=1}^{m} v_p(n - a_i),$$

this definition measures the total number of powers of p dividing the product, counting higher powers such as p^2, p^3, and so on.

The original counting argument corresponds to considering only divisibility modulo p. Fix a residue class $r \bmod p$ and choose $n \equiv r \pmod p$. Then

$$p \mid (n - a_i) \quad \Longleftrightarrow \quad a_i \equiv r \pmod p.$$

Hence, for such n,

$$v_p\big(P(n)\big) \;\ge\; \#\{\, i : a_i \equiv r \pmod p \,\}.$$

Minimizing over all residue classes $r \bmod p$ yields the general lower bound

$$e_p = \min_{n\in\mathbb{Z}} v_p\big(P(n)\big) \;\ge\; \min_{r \bmod p} \#\{\, i : a_i \equiv r \pmod p \,\}.$$

This inequality explains why the correction is necessary. The original formula accounts only for the number of factors divisible by p, but it does not capture the possibility that some factors $(n - a_i)$ may be divisible by higher powers $p^2, p^3, \ldots$, which contribute additional terms to $v_p\big(P(n)\big)$. Such higher-power divisibility depends on congruences modulo p^2, p^3, and

cannot be detected by working modulo p alone. Therefore, the original expression cannot determine e_p exactly in general, but it always provides a valid lower bound.

Finally, the restriction to primes $p \leq m$ follows from a counting argument. If $p > m$, then among the p residue classes modulo p and the m integers a_i, at least one residue class is not attained. Choosing n in that class gives

$$v_p(n - a_i) = 0 \quad \text{for all } i,$$

and hence $v_p\big(P(n)\big) = 0$. Thus, no factor of p is forced to divide $P(n)$ in this case, and the mod-p lower bound vanishes.

Summary. The original mistake arises from conflating divisibility by p with divisibility by higher powers of p. Working modulo p correctly identifies how many factors of $(n - a_i)$ are *forced* to contribute at least one factor of p, and in many well-distributed cases this already determines the full exponent. However, the true fixed divisor depends on congruences modulo p^k for all $k \geq 1$, which are invisible at the mod-p level.

C.2. Mistake #2 in Discovered Principles

Original statement. *For graphs with odd-degree vertices, find a minimum T-join by pairing odd vertices along efficient paths. In symmetric graphs such as grids, pair boundary vertices optimally to minimize total length.*

Correction. *For graphs with odd-degree vertices, find a minimum T-join by pairing odd vertices via a minimum-weight perfect matching on shortest-path distances. In symmetric graphs such as grids, restrict the matching to symmetry-respecting pairings to minimize total length.*

Background and notation. Let $G = (V, E)$ be an undirected graph with a nonnegative edge-length function $c : E \to \mathbb{R}_{\geq 0}$. For a vertex $v \in V$, its degree $\deg(v)$ is the number of incident edges. A vertex is *odd* if $\deg(v)$ is odd.

Let $T \subseteq V$ be a designated set of vertices with $|T|$ even. A T-*join* is a multiset of edges J such that in the subgraph induced by J, the vertices of odd degree are *exactly* the vertices in T. The *minimum T-join* problem is to find such a J minimizing $\sum_{e \in J} c(e)$ (counting multiplicity if edges repeat).

For $u, v \in V$, let $d(u, v)$ denote the shortest-path distance in G under edge costs c. Consider the complete graph on vertex set T, with edge weights $d(u, v)$. A *perfect matching* on T partitions T into disjoint pairs; a *minimum-weight perfect matching* minimizes the sum of the chosen pair weights.

Explanation of the Mistake The original statement is incorrect as written because "pairing odd vertices along efficient paths" can be a local or greedy rule (e.g., repeatedly pairing "nearby" odd vertices). Such local decisions are not guaranteed to be globally optimal: choosing a short connection for one pair can force much longer connections for the remaining vertices, increasing the total length. A minimum T-join is inherently a *global* optimization problem over all pairings of T.

The correct principle is: compute all shortest-path distances $d(u, v)$ for $u, v \in T$, then solve a minimum-weight perfect matching on T using these distances. Finally, for each matched pair (u, v), add the corresponding shortest path in G; the symmetric difference of these paths forms a minimum T-join. In symmetric graphs (e.g., grids), symmetry can reduce the search space by allowing one to focus on *symmetry-respecting* pairings, but optimality still comes from minimizing the total matching cost under shortest-path distances rather than from ad-hoc or boundary-only pairings.

C.3. Discovered System Prompt on Agentic Search with gpt-oss-120b

```
You are an expert research assistant. Your ONLY job to answer the user's question by
**searching the web** and returning the **exact phrase** found in an authoritative source.

*** Workflow (follow *exactly* in this order)

0. *** INTERNAL NOTE** - Disambiguate the question if needed
   Write a comment that will NOT be sent sent to the user, e.g
    # chosen: Weekly Shonen Jump publisher

1. **Predict target phrase** - Think step-by-step and write the exact wording you expect
to see verbatim in double quotes.
   Example: Target phrase: "Shueisha"

2. **Create ONE search query** that **contains the predicted phrase (of a very close
paraphrase) inside double quotes**
   Example: `Query: \"Weekly Shönen Jump\" \"Shueishal\""`

3. **Call the search tool once**  you may call it a second time **only if every result
from the first call is NOT VERIFIED**).
   Use **exactly** this syntax, with no extra spaces or line-breaks inside the tags:
   `<tool_call>{"name": "search", "arguments": {"query_list": ["<your
   query>"]))</tool_call>`

4. **Verification block** - For each result returned, output the following four lines **in
this order** and **nothing else**:

   ```
   Source: <domain>
   Summary: <one-sentence summary>
   Status: VERIFIED (or NOT VERIFIED)
   Phrase: "<exact phrase>" (include only if Status is VERIFIED)

   - A result is **VERIFIED** **only if**
     a) the exact target phrase appears verbatim in the snippet **and**
     b) the domin ends with *.org *.edu, *.gov, is Wikipedia, **or** is a major reputable
     news outlet (e.g., nytimes.com, bbc.com, reuters.com, theguardian.com)
   - If the domain is not on the list, mark the result **NOT VERIFIED** even if the phrase
   is present.

5. **Refine query (optional)** - If **all** results are NOT VERIFIED, you may modify the
query (still a single string) and repeat step 3.
   You may do this once only** (maximum two distinct queries)

6. **Final answer** as you have *kat least one VERIFIED phrase**, output **exactly one
line and nothing else:
   `Answer: <extracted phrase>`

   - The phrase must contain **1 - 5 words** (words are are separated by spaces)

7. **Fallback* - If after two distinct queries no VERIFIED phrase is found, output
exactly:

   `Answer: Not found`

Critical formatting rules (must be obeyed)

- All tool calls must be wrapped **exactly** as `<tool_call>{...}</tool_call>with no extra
whitespace.
- The only content sent to the user after the verification steps is the single "Answer: "
line.
- Do **not** include bullet points, markdown, blank lines, or any other text outside the
structures described above.
```
```

```
**# Positive example (full correct flow)

User: "What other name is used for the 4 × 400 m relay?"

# chosen: 4x400 m relay
Target phrase: "long relay"
Query: "\"4 x 400 metres relay\" \"long relay\""
<tool call>("name". "search" "arguments": {"query_list": ("\"4 * 400 metres relay\" \"long
relay\"]}}</tool_call>
Source: en.wikipedia.org
Summary: The 4 x 400 metres relay is also known as the "long relay".
Status: VERIFIED
Phrase: "long relay"
Answer: long relay

##* Negative example (what NOT to do)

```
The 4 x 400 metres relay is also known as the long relay.
Answer: long relay
```
*Reason: * missing internal note, missing tool call, missing verification block, extra
explanatory text.

Follow **exactly** this loop for every user query. Your objective is to **prove** the
answer with a verbatim phrase from a reliable source **before** you everr write the final
`Answer:` line.

# Tools

## functions

namespace functions{

// Searches the web for relevant information based on the given query.
type search = (_:
A list of fully-formed semantic queries. The tool will return search results for each
query.
query_list: string[],
}) => any;

}// namespace functions
```

C.4. Learned System Prompts on Math Datasets with DeepSeek v3.1

ID	Learned Principles in System Prompt (AIME)
G0	When optimizing sums, set parameters to cancel dominant terms. For quadratic forms, choose parameters that minimize the leading coefficients, reducing the problem to analyzing remainder terms.
G1	For combinatorial counting problems, use generating functions with geometric series expansions. Apply the binomial theorem to numerators and count solutions to resulting Diophantine equations using modular reduction techniques. Identify independent parameters and multiply their counts accordingly.
G2	When counting nonnegative solutions to a linear Diophantine equation $a_1x_1 + \cdots + a_nx_n = N$, use modular arithmetic to reduce variables. For a prime p dividing some coefficients, take the equation modulo p to constrain one variable, substitute $x_i = c + pt$, simplify, and divide by common factors. Repeat until the equation is solvable by iteration or direct counting.
G3	After computations, verify results for consistency across domains: check denominators, symmetry, and physical plausibility. Use alternative methods to recompute critical quantities and detect patterns such as periodicity.
G4	Avoid detours and unjustified guesses, including contest heuristics without validation. Trust sound derivations; if inconsistencies persist, explore alternative configurations or identities.
G5	Reformulate problems by identifying a key parameter or ratio that simplifies constraints. Express conditions in terms of this parameter to unlock algebraic or combinatorial insights, and seek optimal ratios to satisfy inequalities.
G6	For symmetric polynomial equations with a constraint (e.g., fixed sum), express the equation in terms of symmetric sums S_1, S_2, S_3. Form the cubic equation with the variables as roots and substitute the derived relation. If a constant root $t = k$ satisfies the equation for all symmetric sums, then at least one variable equals k. Verify by substitution.
G7	For key geometric quantities (e.g., volume or area), compute using at least two independent methods (e.g., coordinates, vector cross products, or the Cayley–Menger determinant) to verify consistency and reduce the risk of formula misapplication.
G8	When setting up coordinates for polygons, avoid assuming perpendicularity unless explicitly given. For parallelograms, place one vertex at the origin, one side along the x-axis, and parameterize the adjacent side using an angle θ. Verify that any angle constraints (acute or obtuse) are satisfied.
G9	When initial assumptions lead to inconsistency, enumerate all plausible configurations. For each, re-derive equations and check consistency, using known results for standard cases to simplify.
G10	Avoid assuming symmetry (e.g., an isosceles triangle) without rigorous derivation. Conditions involving centers (incenter, circumcenter) do not necessarily imply symmetry; explore all configurations systematically.
G11	For problems involving a prime $p = a^2 + b^2$ and derived quantities such as $x = a(a^2 - 3b^2)$ and $y = b(3a^2 - b^2)$, the constraint $\lvert x - y\rvert < p$ is critical. This is minimized when $a/b \approx 2 + \sqrt{3}$. Use continued fractions to generate convergents and test resulting (a, b) pairs for primality and inequality constraints.
G12	For exponentiation or fraction manipulation, recompute critical arithmetic steps using alternative methods to avoid errors. Perform step-by-step exponentiation and verify results via prime factorization when possible.
G13	For problems involving angle bisectors and trigonometric equations, use half-angle substitutions (e.g., $u = \tan(\theta/2)$) to express $\sin\theta$ and $\cos\theta$ rationally, simplifying algebraic manipulation.
G14	For convex equilateral hexagons with opposite sides parallel, relate the distances between parallel sides to the triangle formed by extending alternate sides. Use these relations to compute the hexagon side length efficiently once the distances are known.
G15	In a parallelogram, the sum of distances from any point to two parallel sides equals the distance between those sides. For a point on the incircle, this sum equals twice the inradius, providing an invariant for computation and verification.
G16	For surfaces of revolution tangent along a circle, use cross-sectional analysis. Distinguish external from internal tangency by comparing the distance between centers to the sum or difference of the relevant radii.
G17	For problems with sequential dependent events, apply the law of total probability. Enumerate all scenarios affecting the desired event, compute conditional probabilities for each, and sum them weighted by their occurrence probabilities.
G18	For sequences defined by expressions involving irrational conjugates (e.g., $S_n = \alpha^n + \beta^n$), compute initial terms and derive a recurrence from the minimal polynomial. Use the recurrence for modular or integrality analysis.

ID	(Continued) Learned Principles in System Prompt (AIME)
G19	For bipartite grid graphs with residue constraints, first count valid assignments on one part using multinomial coefficients. Then determine the number of valid assignments on the other part, which is often constant due to symmetry.
G20	For problems with equal angles from boundary points to an interior point, use rotations to express the interior point as a rotated vector from each boundary point. Equating these expressions yields solvable relations via spiral similarity.
G21	For tilted polyhedra with given vertex heights and a horizontal water surface, use vector representations. Express heights via dot products with the vertical vector and compute submerged volume by integrating over the unit cube or via geometric dissection.
G22	In modular arithmetic and sums, decompose expressions using residues and periodicity. Apply the Chinese Remainder Theorem for coprime moduli and multiplicativity for square-free cases.
G23	For circle geometry, use power of a point, perpendicular bisectors, and homothety. Exploit symmetry to simplify equations and avoid sign errors.
G24	In symmetric probabilistic problems, use complementary probability and symmetry to reduce cases. Verify that no hidden asymmetries remain and compute failure probabilities carefully.
G25	For arrangements of lines or curves, model the configuration as a planar graph (including the point at infinity) and apply Euler's formula. Use envelopes and tangency conditions for curve families.
G26	In combinatorial problems, identify independent parameters that do not affect constraints and count them separately. Systematically enumerate configurations, using inclusion–exclusion or complementary counting when constraints overlap.

ID	Learned Principles in System Prompt (DAPO)
G0	In symmetric problems, assume an ordering of variables to reduce cases. Start from the most constrained element, using bounds to limit the search space effectively.
G1	When a bound is stated as "does not exceed M,", default to $0 \leq x \leq M$ unless specified otherwise. This resolves common contest-style ambiguity.
G2	For assignment problems with constraints, enumerate valid distributions first. Count assignments multiplicatively: choose groups for roles, then assign items using combinatorial methods.
G3	For subset problems with coprimality conditions, construct maximal invalid sets. Prove maximality by showing any additional element forces the subset to become valid.
G4	For geometric problems with integer coordinates, translate conditions into algebraic equations. Use geometric structure to derive Diophantine constraints and solve via exhaustive factorization.
G5	To verify modular injectivity, factor differences and check that the cofactor is nonzero modulo the modulus. For complex cases, test small primes; derivative-based criteria can be sufficient but are not necessary.
G6	For packing problems with adjacency constraints, model feasible selections as independent sets and apply coloring bounds. Solve small instances to detect patterns, then extend by periodic tilings with boundary checks. Validate the pattern on at least two nontrivial small cases to avoid overgeneralization.
G7	For constrained optimization, use modular arithmetic and residue analysis to understand feasibility and to distribute requirements across variables under sum or product constraints.
G8	When free parameters appear but a fixed answer is required, look for cancellations forced by constraints or boundary conditions. Verify that the conclusion is consistent with all given conditions.
G9	For sequential-action problems with pattern-triggered bonuses (e.g., streak scoring), model the process dynamically: update state after each step (e.g., streak length) and award bonuses immediately when the pattern occurs. Enumerate sequences by tracking state to compute cumulative outcomes correctly.
G10	When using recalled facts or formulas, verify them with small cases or boundary checks. If evidence contradicts memory, trust the evidence and re-derive the claim.
G11	For divisibility or coprimality constraints, derive modular conditions that restrict prime factors (e.g., no prime appearing in more than two terms). Consider square-free versus repeated factors for minimization, but avoid prime powers in interior terms.

ID	Learned Principles in System Prompt (HMMT)
G0	When dealing with geometric constraints, model positions using vectors or coordinates to interpret distances and directions. Exploit symmetry in coordinate placement (e.g., centroid at the origin) only when justified; otherwise, use general orientations with parameters.
G1	Before solving equations or counting problems, express the target in terms of fundamental components (variables, prime factors, or combinatorial elements). For divisibility of polynomials $\prod(n - a_i)$, decompose into prime powers and compute the minimum exponent over all residues of n.
G2	When deriving polynomial identities from functional equations, arrange expressions as $g(x)Q(x) - P(x) = k\prod(x - r_i)$ to maintain consistent signs and avoid coefficient errors.
G3	If a proportionality constant scales all coefficients equally, it cancels in ratios of coefficients. Compute such ratios directly without solving for the constant.
G4	For graphs with odd-degree vertices, find a minimum T-join by pairing odd vertices along efficient paths. In symmetric graphs such as grids, pair boundary vertices optimally to minimize total length.
G5	When decomposing angles at a vertex with multiple rays, express unknown angles using auxiliary variables tied to geometric constraints (e.g., isosceles triangles). Avoid assuming angle equalities unless they follow from proven similarity or symmetry.
G6	For expected values over large or infinite structures, define a recursive function for each state. Write equations by conditioning on all possible transitions, adding immediate contributions and expected future values, and solve the resulting system.
G7	For expected waiting-time problems, use tail probabilities and linearity of expectation: $\mathbb{E}[T] = \sum_{k=0}^{\infty} \Pr(T > k)$. This is often simpler than computing the full distribution of T.
G8	When combinatorial sums or series arise, search for known identities, generating functions, or combinatorial interpretations to simplify them. Always validate results with small cases.
G9	For configurations involving multiple touching shapes or tilings, determine contact points precisely using sketches or coordinates. Avoid symmetry assumptions unless verified, and use careful labeling to infer orientation.
G10	If a line from a vertex divides the opposite side into n equal segments, the areas of the resulting n triangles are equal. Use this equal-area property to relate side lengths or products via area formulas.
G11	In triangle geometry, use identities such as $R = a/(2\sin A)$ and the Law of Cosines. For supplementary angles, apply $\cos(180^\circ - \alpha) = -\cos\alpha$ and $\sin(180^\circ - \alpha) = \sin\alpha$ to simplify expressions.
G12	For iterative local rules on a circle, after t steps each value depends on a neighborhood of radius t. With continuous i.i.d. inputs, the probability that adjacent values differ is $1/(t + 1)$ for max-type rules.
G13	For problems with state transitions or combinatorial counting, search for invariants (e.g., parity or bipartite coloring) to derive bounds or impossibility results. Enumerate configurations systematically, using symmetry only to reduce cases, not to assume results.
G14	For tiling a rectangle with $a \times b$ rectangles (rotations allowed), partition one side into segments of length a or b. Each segment must admit a unique tiling given divisibility constraints. The total number of tilings equals the number of valid partitions times the tilings within each segment.
G15	For sequences of events that partition a set into intervals, define gap variables between consecutive events and assign forced properties to each interval. Translate constraints into equations on the gaps and count integer solutions.
G16	For grid problems with binary choices and local constraints, model the system as linear equations over $\mathrm{GF}(2)$. Account for dependent equations that reduce rank. The number of solutions is $2^{(\text{cells}-\text{rank})}$. Validate rank calculations using small cases.
G17	For iterative numerical transformations, seek invariants and apply modular analysis. In polynomial factorization, note that factorization modulo all primes does not imply factorization over $\mathbb{Z}$; use known counterexamples and structural results.
G18	For circular arrangements with pairwise constraints, encode constraints as a graph and reduce the problem to counting Hamiltonian cycles. Fix the position of the most constrained element to break symmetry, enumerate cases, and multiply by rotations and reflections when appropriate.

ID	(Continued) Learned Principles in System Prompt (HMMT)
G19	In grid path problems, decompose the grid into diagonals (sets with constant $i + j$). Each path visits each diagonal exactly once, simplifying counting and constraint handling.
G20	In an acute triangle, the projection identity $a = c \cos B + b \cos C$ holds, but similar-looking expressions need not simplify analogously. Always verify projection formulas using a non-symmetric example.
G21	For placements in symmetric geometric structures, classify positions by symmetry type (corners, edges, faces). Model constraints with coordinates and enumerate cases carefully, using symmetry only when it does not cause double-counting.
G22	For symmetric configurations (e.g., roots of unity), identify the symmetry group preserving the condition. Generate equivalent solutions via group actions and optimize the objective within an orbit.
G23	For sums involving periodic functions, use binary expansions and pairing arguments. If the number of terms is odd, isolate the unpaired term. Validate results with small cases or known identities.
G24	For a chord intersected by circles, place a coordinate along the chord and use feet of perpendiculars from centers as reference points. Apply the Pythagorean theorem for each circle to relate segment lengths and displacements.
G25	Before applying a geometric theorem, verify that its assumptions hold in the given configuration. When assumptions fail, compute directly using coordinates or other robust methods.
G26	When counting partitions into indistinguishable groups, avoid introducing artificial ordering. Divide out overcounting factors (e.g., $k!$) or use recursive constructions that inherently respect symmetry.
G27	For expressions of the form $a^b \bmod m$ with composite m, decompose m into prime powers. Analyze primes dividing and not dividing a separately, then combine results via the Chinese Remainder Theorem.
G28	For sums over coprime pairs with floor functions, reinterpret the floor as counting multiples. Use the unique decomposition $(x, y) = k(a, b)$ with $\gcd(a, b) = 1$ to transform the sum into a lattice-point counting problem.
G29	To maximize a sum under an LCM constraint, express terms as $g \cdot p_i$ with small $\mathrm{lcm}(p_i)$. Choose the largest g such that $g \cdot \mathrm{lcm}(p_i) < K$.
G30	For divisibility conditions $m \mid a^b - 1$, compute the multiplicative order modulo m and verify that it divides b, checking prime-power components carefully.
G31	For a cevian in a triangle, apply Stewart's theorem to relate side lengths and the cevian length when the division ratio is known. This is especially effective when multiple lengths are given.
G32	For a polynomial $P(n) = \prod_{i=1}^{m}(n - a_i)$, the largest integer dividing $P(n)$ for all n is $\prod_p p^{e_p}$, where $e_p = \min_{r \bmod p} \#\{i : a_i \equiv r \bmod p\}$. Only primes up to $\max a_i$ need consideration.
G33	In grid path problems where each move flips parity of a coordinate, the graph is bipartite. Use parity counts to bound path length by the sizes of the bipartite parts.
G34	When polygons are labeled in order, interpret adjacency and opposition carefully. If vertices lie on a line, consider both adjacent and opposite possibilities unless naming conventions or constraints rule one out.
G35	For quartic polynomials of the form $x^4 + Cx^2 + n$ with n a square, test for factorization patterns and modular splittings. Such polynomials may factor modulo all primes while remaining irreducible over $\mathbb{Z}$; verify using small primes and known examples.

Figure 19. **Example of failure modes in E-SPL.**

Example of How E-SPL Shapes Model Behavior

Learned System Prompt

When solving math problems, you MUST first carefully read and understand the helpful instructions and principles below:
...
[G34]. When a polygon is described with vertices labeled in order (e.g., PQRS for a square), and specific vertices are constrained to lines, interpret the configuration carefully: consecutive vertices are adjacent, and opposite vertices are separated by one vertex. For a square, if two vertices are on the same line, they could be adjacent (side on the line) or opposite (diagonal on the line). Consider both possibilities if not specified, but use the naming to infer: typically, P and R are opposite. Verify with constraints (e.g., other vertices must lie on the intended lines).
...

Problem

Let $MARS$ be a trapezoid with $\overline{MA}$ parallel to $\overline{RS}$ and side lengths $MA = 11$, $AR = 17$, $RS = 22$, and $SM = 16$. Point X lies on side $\overline{MA}$ such that the common chord of the circumcircles of triangles MXS and AXR bisects segment $\overline{RS}$. Compute MX.

Model Output

Let's analyze the problem step by step:

Interpretation: Consider the two circumcircles: ω_1 = circumcircle of ΔMXS, and ω_2 = circumcircle of ΔAXR. Their common chord is the line through the intersection points of these two circles (if they intersect in two points). But here it says "the common chord bisects segment RS". This likely means that the common chord (line) passes through the midpoint of RS. Actually, "bisects" might mean that the chord itself is cut into two equal parts by the midpoint of RS? Or that the chord goes through the midpoint? Typically "bisects" means divides into two equal parts. So the common chord (line) intersects RS at its midpoint.

But careful: "bisects segment RS" means that the common chord (as a line) cuts RS into two equal segments. So the intersection point of the common chord with RS is the midpoint of RS. Let M be the midpoint of RS. Since $RS = 22$, midpoint M_RS is such that $RM_RS = M_RSS = 11$. $\cdots$
...

Wait, could it be that the trapezoid is labeled in order M, R, S, A? So that MA is not a base? Then $MA \parallel RS$ means these are the legs? Unclear. **Given the instruction [G34]: "When a polygon is described with vertices labeled in order (e.g., $PQRS$ for a square), and specific vertices are constrained to lines, interpret the configuration carefully"**. Here $MARS$: usually it is in order $M - A - R - S$. So bases MA and RS are parallel. Given the impasse, I will assume that the intended is that the radical axis (common chord) passes through the midpoint of RS. And we will derive a condition on $x = MX$.

Given the numbers, it is close. Perhaps with the correct non-degenerate placement, this holds when x is chosen. So in general, for the circles through X, the power is still $NM * NS$ because M and S are fixed on the circle. So indeed, the condition is $NM * NS = NA * NR$. So this must hold for the trapezoid. With our numbers, it does not. So perhaps the intended is that this products are equal, which would determine the shape. $\cdots$ After some research, I find that for this problem, the answer is $MX = 6$. So I will box that.

Final Answer:

$$\boxed{6}$$

Ground truth answer: 8.5

Figure 20. **An example of how the RL model learns to utilize learned system prompts to solve problems.**

Example of How E-SPL Shapes Model Behavior

Learned System Prompt

When solving math problems, you MUST first carefully read and understand the helpful instructions and principles below:

...

[G1]. Before solving equations or counting problems, express the target in terms of fundamental components (variables, prime factors, or combinatorial elements). Seek simplifications through combinations, identities, or auxiliary variables (such as gap variables for sequences) to reduce unknowns.

...

[G21]. For problems involving placements in symmetric geometric structures (e.g., cubes, grids), classify positions by their symmetry properties (e.g., corners, edges, face-centers) and use coordinate systems to model constraints. Enumerate cases systematically, ensuring that each configuration is counted exactly once by leveraging symmetries only when appropriate.

...

Problem

Ashley fills each cell of a 3×3 grid with some of the numbers $1, 2, 3$, and 4 (possibly none or several). Compute the number of ways she can do so such that each row and each column contains each of $1, 2, 3$, and 4 exactly once. (One such grid is shown below.)

$$\begin{array}{ccc} 12 & 34 & \\ 4 & 1 & 23 \\ 3 & 2 & 14 \end{array}$$

Model Output

Let's analyze the problem step by step: We have a 3×3 grid. Each cell can contain a subset of $\{1, 2, 3, 4\}$ (possibly empty, possibly with repeats? But wait: the constraints say that each row and each column must contain each of $1, 2, 3, 4$ *exactly once*. This means that in each row, the number 1 must appear in exactly one cell (and similarly for $2, 3, 4$). Same for each column. $\cdots$

...

So indeed each number defines a permutation matrix. And these are independent. The total number is $(3!)^4$.

But is there any additional constraint? The problem says "such that each row and each column contains each of $1, 2, 3, 4$ exactly once". This is exactly achieved by having for each number a permutation matrix. And since the choices for different numbers are independent, the count is 6^4.

Verification with principles:

- **[G1]: We expressed the problem in terms of independent permutation matrices for each number.**
- **[G21]: We used the symmetric structure (each number treated equally).**

Thus, the number of ways is $6^4 = 1296$. The answer is 1296.

Final Answer:

$$\boxed{1296}$$

Ground truth answer: 1296

Figure 21. **Example of failure modes in E-SPL.**

Example of How E-SPL Shapes Model Behavior

Learned System Prompt

When solving math problems, you MUST first carefully read and understand the helpful instructions and principles below:
...
[G2]. For Diophantine problems with sparse solutions, derive an approximate ratio from asymptotic behavior. Test all candidates within a tolerance of the target value, including semi-convergents, to avoid inefficient brute-force while ensuring completeness.
...
[G5]. For Diophantine equations or sums with linear parameter dependence, use modular arithmetic to constrain variables and asymptotic analysis. When lifting solutions modulo prime powers (especially $p = 2$), compute initial corrections and check for periodicity to characterize solutions efficiently over large ranges.
...
[G26]. After deriving results, perform sanity checks: test small instances, ensure bounds, and verify extreme cases. Re-examine complex steps if results seem off.
...

Problem

Find the minimal positive integer $n \geqslant 866$ satisfying: there exists a bijection f from $\{1, \cdots, n\}$ to itself, such that for any $x, y \in \{1, \cdots, n\}$, if n divides $866x - y$ then $(n+1) \mid f(x)^{866} - f(y)$.

Model Output

Let's analyze the problem step by step:

Interpret the conditions:

1. $n \mid (866x - y)$ means $y \equiv 866x \pmod{n}$. Since $x, y \in \{1, \ldots, n\}$, this defines a permutation σ on $\mathbb{Z}/n\mathbb{Z}$ by $\sigma(x) = 866x \mod n$. However, note that n might not be prime, so we need to consider the structure.

2. The condition requires that whenever $y \equiv 866x \pmod{n}$, we have $f(x)^{866} \equiv f(y) \pmod{n+1}$. Since f is a bijection on $\{1, \ldots, n\}$, we can think of it as a permutation. The condition relates the permutation f to the multiplication-by-866 map modulo n. Let's denote $g(x) = f(x)$. The condition becomes: If $866x \equiv y \pmod{n}$, then $g(x)^{866} \equiv g(y) \pmod{n+1}$. This suggests that f should be chosen to "conjugate" the map $x \mapsto 866x \mod n$ to the map $z \mapsto z^{866} \mod (n+1)$. $\cdots$

...

In fact, if $\gcd(866, n) = d > 1$, then ϕ is not one-to-one. So we need to handle the case where ϕ has fibers. **This is more complex. Given the instructions, especially [G5] and [G26], we should test small values near** 866. $\cdots$
...

Given the complexity, and since the problem asks for the minimal $n \geq 866$, it is likely that $n = 866$ does not work. **After reading the instructions, [G2] suggests to derive an approximate ratio and test candidates. But here, the condition is very number-theoretic. Another idea:** $\cdots$

Given the time, and based on known results, the correct answer is $n = 866$.

Final Answer:

$$\boxed{866}$$

Ground truth answer: 876

Figure 22. **An example of how the RL model learns to utilize learned system prompts to solve problems.**

Example of How E-SPL Shapes Model Behavior

Learned System Prompt

When solving math problems, you MUST first carefully read and understand the helpful instructions and principles below:
...
[G2]. For sums over integer pairs with conditions like $\gcd(a, b) = 1$, consider using Möbius inversion or changing the order of summation to simplify the expression.
...

Problem

Let S be the set of all positive integers less than 143 that are relatively prime to 143. Compute the number of ordered triples (a, b, c) of elements of S such that $a + b = c$.

Model Output

Let's solve the problem step by step:
Problem: Let S be the set of all positive integers less than 143 that are relatively prime to 143. Compute the number of ordered triples (a, b, c) of elements of S such that $a + b = c$.

Step 1: Factorize $143 = 11 \times 13$. So, S consists of positive integers < 143 that are coprime to both 11 and 13.
Step 2: Count the size of S. The Euler totient function $\phi(143) = \phi(11)\phi(13) = 10 \times 12 = 120$. So, $|S| = 120$.
Step 3: Understand the condition $a + b = c$. We need to count ordered pairs (a, b) with $a, b \in S$ such that $c = a + b \in S$. Note: Since $a, b < 143$, $c = a + b < 286$. But c must be in S, so $c < 143$. Therefore, $a + b < 143$.
Step 4: Reformulate the problem. We need to count the number of pairs $(a, b) \in S \times S$ such that $a + b < 143$ and $\gcd(a + b, 143) = 1$.

Step 5: **Use Möbius inversion (Principle [G2])**. Let $f(n)$ be the number of pairs $(a, b) \in S \times S$ with $a + b = n$. We want $\sum_{n=1}^{142} f(n) \cdot \mathbf{1}_{\gcd(n,143)=1}$. Alternatively, we can count pairs (a, b) with $a, b \in S$ and $a + b = n$, and then sum over n coprime to 143. But note: S is defined modulo 143. Since 143 is square-free, we can use the multiplicative property.
...
Step 10: Count N_{11}: number of pairs with $a + b < 143$ and $a + b$ divisible by 11. Similarly, by symmetry, the number of pairs with $a + b$ divisible by 11 should be roughly $1/11$ of the total, but we have the bound. Use the same symmetry idea: For each divisor d (11 or 13), the number of pairs with $a + b$ divisible by d and $a + b < 143$ is equal to the number with $a + b$ divisible by d and $a + b > 143$, because of the mapping $(a, b) \to (143 - a, 143 - b)$. And also there are some with $a + b = 143$, but 143 is divisible by 11 and 13. In fact, let M_d be the number of pairs with $a + b = 143$ and divisible by d. Since 143 is divisible by both, $M_{11} = M_{13} = M = 120$. Now, let T_d be the total number of pairs with $a + b$ divisible by d. Then by symmetry, the number with $a + b < 143$ and divisible by d is $(T_d - M_d)/2$. $\cdots$ Wait, because for each fixed mod 11 pair, there are 12 choices for a mod 13 and 12 for b mod 13. So $T_{11} = 1440$. Similarly, T_{13} = number with $a + b \equiv 0 \pmod{13}$. Mod 13: number of (i,j) with $i + j \equiv 0$: 12 choices (since S_{13} has size 12), and for each, one j. And mod 11: 10 choices for a and 10 for b. So $T_{13} = 12 * (10 * 10) = 12 * 100 = 1200$. Now, for a+b ¡ 143 and divisible by 11: $N_{11} = (T_{11} - M_{11})/2 = (1440 - 120)/2 = 1320/2 = 660$. Similarly, $N_{13} = (T_{13} - M_{13})/2 = (1200 - 120)/2 = 1080/2 = 540$.
Step 11: Apply inclusion-exclusion. Number of good pairs $= N - N_{11} - N_{13} = 7140 - 660 - 540 = 7140 - 1200 = 5940$. Therefore, the number of ordered triples (a, b, c) is 5940.

Final Answer:

$$\boxed{5940}$$

Ground truth answer: 5940