RBEPO: FROM ROOT-CAUSE DIAGNOSIS TO BRANCHING EVOLUTIONARY PROMPT OPTIMIZATION

Anonymous authors
Paper under double-blind review

ABSTRACT

With the widespread deployment of large language models in agentic systems and domain-specific tasks, using execution feedback to enable system self-evolution has attracted increasing attention. Automatic prompt optimization provides a lightweight path for systems to continually improve task performance without updating model parameters. However, identifying problems from execution feedback does not necessarily reveal effective repair strategies; joint rewrites across multiple directions obscure the actual effects and reuse potential of individual repairs. We propose RBEPO (Root-cause-guided Branching Evolutionary Prompt Optimization), a prompt optimization framework centered on cross-instance failure modes that jointly organizes targeted branching search and mechanism-level experience accumulation. RBEPO infers repair directions through per-instance error analysis and cross-instance root-cause clustering, generates and evaluates targeted rewrite branches for individual failure modes, and integrates effective, complementary repairs. Building on these evaluated rewrites, the method distills rewrite attempts and their evaluation results into cross-iteration experience to guide subsequent rewrites. To control the resulting candidate-evaluation cost, prompt-edit fragility-guided racing is introduced. Across multiple tasks and models, RBEPO consistently achieves higher test scores than the compared baselines: its average score is approximately 3.35 points higher than that of the strongest baseline in aggregate, while its final prompts are on average only about 1/6 as long. Comparisons of evaluation strategies further show that this strategy reduces the total sample-evaluation volume to about 38% of full evaluation while maintaining performance close to that of full evaluation.

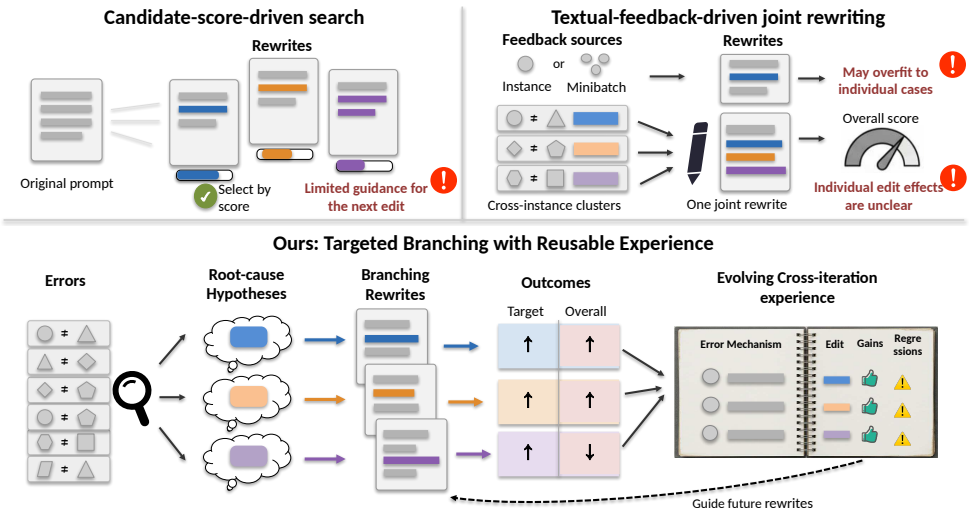


Figure 1: Comparison of prompt optimization paradigms. RBEPO organizes feedback with cross-instance failure modes, validates targeted rewrite branches, and reuses cross-iteration experience.

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

1 INTRODUCTION

With the widespread deployment of large language models in agentic systems and domain-specific tasks, using execution feedback to enable system self-evolution has attracted increasing attention (Fang et al., 2025; Ouyang et al., 2026; Zhang et al., 2026d;a). This line of research investigates how systems can identify problems and accumulate experience from task execution to improve subsequent performance, reducing reliance on repeated manual adjustments. Automatic Prompt Optimization (APO) uses task data and evaluation feedback to search for more effective task instructions, enabling systems to adjust their task behavior without updating model parameters and offering a lightweight, iterative path to feedback-driven improvement (Agrawal et al., 2026; Zhang et al., 2026c; Suzgun et al., 2026).

Existing research has explored different approaches to automatic prompt optimization. One line focuses on search guided by candidate evaluation, exploring the prompt space through language-model-based generation, Bayesian optimization, or evolutionary search (Zhou et al., 2022; Guo et al., 2024; Opsahl-Ong et al., 2024; Zehle et al., 2025). Another uses error analysis and textual feedback from task execution to guide rewriting, providing semantic guidance beyond candidate scores. Textual feedback is organized at different granularities: reflection and repair suggestions for individual examples or execution traces (Shinn et al., 2023); prompt rewriting guided by execution traces and evaluation feedback from mini-batches (Pryzant et al., 2023; Yuksekgonul et al., 2024; Agrawal et al., 2026); and cross-instance analysis to identify shared failure modes (Singh et al., 2026; Bayat et al., 2026).

However, identifying failure modes shared across examples does not mean that effective repair strategies have been found. Root-cause explanations and their corresponding rewrites must still be tested through evaluation; when a rewrite mixes multiple repair directions, gains in some directions may mask ineffectiveness or regressions in others. Aggregate scores make it difficult to determine whether each repair resolves its target problem and are insufficient to reveal which rewrite strategies merit reuse in subsequent search. Repair targets, actual changes, and evaluation results therefore need to be linked so that repair effects can be tested and mechanism-level experience accumulated for subsequent search.

To this end, we propose RBEPO (Root-cause-guided Branching Evolutionary Prompt Optimization), a prompt optimization framework centered on cross-instance failure modes that jointly organizes targeted branching search and mechanism-level experience accumulation. RBEPO uses per-instance error analysis and cross-instance root-cause clustering to infer repair directions, generates complete prompt branches for different failure modes, and separately evaluates and integrates effective, complementary repairs. Building on this process, RBEPO summarizes rewrite targets, actual changes, and per-instance evaluation results into error-mechanism–rewrite-mechanism relations, forming mechanism-level experience for subsequent candidate generation within the same task. Figure 1 compares RBEPO with representative optimization approaches in terms of feedback organization, candidate generation, and experience use. To control the evaluation cost introduced by branching search, RBEPO builds on CAPO's racing approach (Zehle et al., 2025) and further uses historical per-instance score degradation to adjust the evaluation order, reducing evaluation effort on unfavorable candidates.

Our main contributions are summarized as follows:

1. We propose RBEPO, a framework that requires no model parameter updates, uses cross-instance failure modes to define repair targets for targeted branches, summarizes rewrites and evaluation results into mechanism-level experience, and applies fragility-guided racing to control evaluation cost.
2. We compare RBEPO with MIPRO v2 (Opsahl-Ong et al., 2024), GEPA (Agrawal et al., 2026), and CAPO (Zehle et al., 2025) on four public datasets and two target models. Under the configurations used here, RBEPO improves the average test score by approximately 3.35 points over CAPO, the strongest single baseline in aggregate, while its final prompt is on average about 1/6 as long.
3. We use cumulative ablations to validate experience reuse, branching and merging, and structured diagnosis, and compare sequential-evaluation strategies; fragility-guided racing reduces evaluations to about 38% of full evaluation with comparable effectiveness.

2 Related Work

Search-Based Prompt Optimization. Search-based prompt optimization explores the prompt space through discrete search (Shin et al., 2020), language-model generation (Zhou et al., 2022; Yang et al., 2024), evolutionary search (Guo et al., 2024; Fernando et al., 2023), or planning-based search (Wang et al., 2024). EvoPrompt (Guo et al., 2024) evolves prompt populations using LLM-based mutation and crossover, while Promptbreeder (Fernando et al., 2023) co-evolves task prompts and the mutation prompts guiding their revision, thereby optimizing candidate generation itself. MIPRO v2 (Opsahl-Ong et al., 2024) jointly searches instructions and few-shot demonstrations, whereas SPRIG (Zhang et al., 2026b) uses system-prompt components as search units. In these searches, candidate prompts are compared and selected using evaluation scores, but scores alone offer limited insight into error causes or specific repair directions.

Feedback-Driven Diagnosis and Prompt Rewriting. Textual feedback provides evidence beyond scalar scores for candidate rewriting. ProTeGi (Pryzant et al., 2023) and TextGrad (Yuksekgonul et al., 2024) use textual gradients; GEPA (Agrawal et al., 2026) performs reflective rewriting from execution traces and evaluation feedback, combined with evolutionary search and candidate merging; PromptWizard (Agarwal et al., 2024) combines prompt mutation, critique feedback, and instruction reconstruction. Cross-instance diagnosis organizes scattered feedback into recurring repair requirements. RPT (Bayat et al., 2026) aggregates diagnoses of failed responses, and ETGPO (Singh et al., 2026) builds error categories from failure trajectories; ERGO (Cui et al., 2026) derives decision rules from label confusions; CARO (Chu et al., 2026) diagnoses confusion modes separately, generates and consolidates mode-specific repairs, and evaluates and selects candidates. StraGo (Wu et al., 2024) derives rewrite strategies from successful and failed examples, whereas Contrastive Reflection (Koh et al., 2026) uses contrastive evidence from error-heavy regions for targeted edits, gated by validation gains and optional regression checks. RBEPO generates branches from shared root-cause hypotheses, jointly checks target-cluster and overall improvements for local candidates, merges validated complementary repairs, and distills rewrite targets, actual changes, and observed outcomes into mechanism-level experience.

Historical Feedback and Cross-Iteration Experience Reuse. Historical feedback can support subsequent rewriting through diagnostic reports, rejected edits, or structured experience: RPT retains reports across iterations (Bayat et al., 2026), Contrastive Reflection records rejected edits, and ACE (Zhang et al., 2026c) updates feedback into playbooks with effect counts. More broadly, self-evolving-agent research studies how experience is accumulated and reused during task execution: Reflexion (Shinn et al., 2023) and Agent-Pro (Zhang et al., 2024) perform verbal and policy-level reflection, Dynamic Cheatsheet (Suzgun et al., 2026) maintains an adaptive memory of reusable solution snippets at test time, ReasoningBank (Ouyang et al., 2026) distills reasoning memory from successful and failed trajectories, MemRL (Zhang et al., 2026d) uses runtime reinforcement learning to update episodic-memory utility values and optimize retrieval policies, a recent survey (Fang et al., 2025) reviews this lifelong-agent paradigm, and Hyperagents (Zhang et al., 2026a) extends self-improvement to the meta-level modification procedure itself. These works mostly operate on cross-task execution memory or agent policies; ACE's playbook directly participates in task execution, whereas RBEPO's task-internal experience is summarized from rewrite targets, actual changes, and validation results to guide subsequent candidate generation.

Evaluation Efficiency and Budget Allocation. Candidate evaluation can control cost through minibatch screening, surrogate evaluation, and sequential elimination: MIPRO v2 uses minibatch scores to update a surrogate, GEPA expands evaluation for candidates that pass screening, and racing eliminates unfavorable candidates early through sequential evaluation (Maron & Moore, 1993; Birattari et al., 2002; López-Ibáñez et al., 2016); CAPO (Zehle et al., 2025) applies racing in evolutionary search. RBEPO retains blockwise comparison and early elimination, while ordering examples by historical per-instance score degradation.

3 Problem Definition

Given a fixed target model, an optimization set $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^{n}$, and an initial prompt p_0, prompt optimization seeks, without changing model parameters, a prompt with better task performance under a given search budget. Here, x_i and y_i denote the input and reference output, respectively.

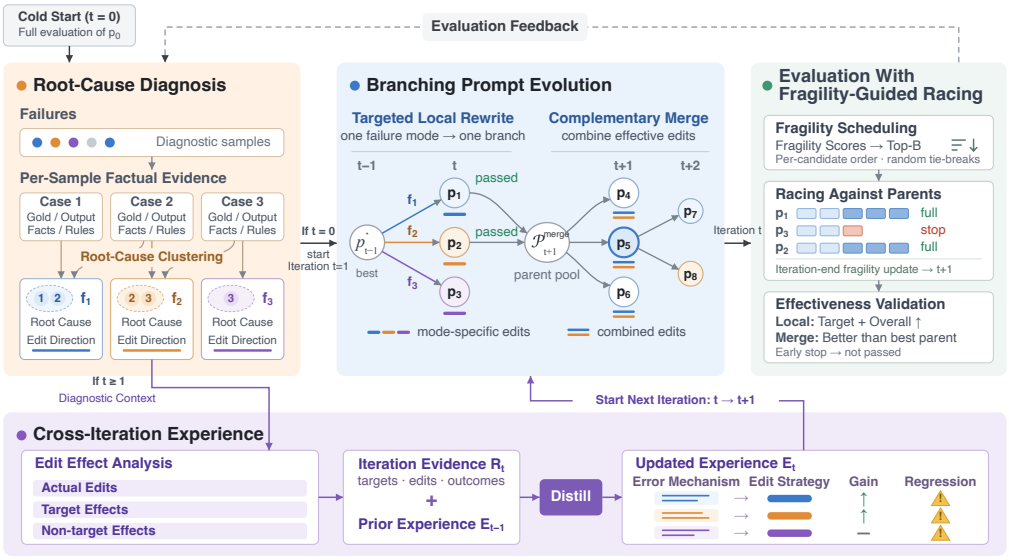


Figure 2: Overview of RBEPO. At initialization, the initial prompt is evaluated and diagnosed, and the method proceeds directly to the first rewrite. Each iteration begins with candidate generation; after evaluation, effectiveness is determined; combined with feedback diagnosis, the rewrite effects are summarized and the experience is updated for use in the next rewrite.

Let p denote a prompt version, $s_i(p)$ its task score on example i, and $S(p)$ its task metric on the full optimization set; larger values indicate better task performance for both quantities, and per-example scores are lower-bounded by 0.

Version selection is based on $S(p)$. Let $\mathcal{P}^{\text{full}}_{\leq t}$ denote the set of prompts that have completed full evaluation by iteration t, and the current best prompt is defined as

$$p_t^\star \in \arg\max_{p \in \mathcal{P}^{\text{full}}_{\leq t}} S(p). \tag{1}$$

When search terminates, the current best prompt is returned. The optimization set is used for search and version selection; the test set is not involved in the optimization process.

4 Method

We propose RBEPO (Root-cause-guided Branching Evolutionary Prompt Optimization), which uses failure modes to link diagnosis, targeted rewriting, complementary-branch merging, and cross-iteration experience accumulation, and reduces candidate-evaluation cost through prompt-edit fragility-guided sequential evaluation. This section follows the notation in Section 3.

4.1 Method Overview

Initialization ($t = 0$). We fully evaluate the initial prompt p_0 and diagnose its failure modes from the evaluation feedback to prepare a basis for the first rewrite. We set $p_0^\star = p_0$.

Iterative optimization ($t \geq 1$). Each iteration begins with candidate generation. The first iteration performs targeted rewriting from the initial diagnosis; later iterations combine existing diagnoses with historical experience and choose between targeted local rewriting and complementary-branch merging. After candidates are evaluated, we determine whether each rewrite is effective; once failure diagnosis is complete, we summarize the rewrite effects and distill experience to guide the next rewrite. At most T iterations are performed; if no candidate can be generated, the search stops early and returns the current best prompt. Figure 2 shows the overall workflow.

4.2 FAILURE-MODE DIAGNOSIS

Through per-instance factual analysis and cross-instance root-cause clustering, RBEPO summarizes errors that can be explained by the same root-cause hypothesis and share a consistent repair direction into failure modes.

Diagnostic example selection. For a prompt p, we select the evaluated examples scoring below the mean for diagnosis. If all evaluated examples have the same score and none reaches the task maximum, all of them are included. Let $\mathcal{I}^{\mathrm{eval}}(p) \subseteq \{1, \ldots, n\}$ denote the set of evaluated example indices and $\mathcal{I}^{\mathrm{diag}}(p)$ the set of selected diagnostic indices. If $\mathcal{I}^{\mathrm{diag}}(p) = \varnothing$, no failure mode is constructed.

Per-instance factual analysis. We independently analyze each example in $\mathcal{I}^{\mathrm{diag}}(p)$, recording the discrepancy between the gold answer and the model output, the outcome-determining facts and their sources, the relevant rules, and the direct error.

Cross-instance root-cause clustering. Based on the per-instance factual analysis, we summarize root-cause hypotheses and rewrite directions, and cluster examples that share a consistent root-cause hypothesis and rewrite direction, yielding the failure-mode set $\mathcal{F}(p)$ of prompt p. Each mode $f_k \in \mathcal{F}(p)$ contains a failure cluster, a shared root-cause hypothesis, and a rewrite direction, and the example indices of its failure cluster are denoted $\mathcal{I}^{\mathrm{cluster}}(f_k) \subseteq \mathcal{I}^{\mathrm{diag}}(p)$. One example may belong to multiple independent modes, so the failure clusters need not be disjoint.

4.3 FAILURE-MODE-DRIVEN BRANCH EVOLUTION

In iteration t, based on $p^{\star}_{t-1}$, the diagnosis, prompt history, and historical experience (whose distillation and update are described in Section 4.5), we choose between targeted local rewriting and complementary-branch merging.

Targeted local rewriting. From the failure-mode set $\mathcal{F}_t = \mathcal{F}(p^{\star}_{t-1})$ of the current best prompt, we select the top K modes by the number of examples in their failure clusters, denoted $\mathcal{F}^{\mathrm{sel}}_t$. For each selected mode, combined with historical experience, we generate one complete prompt candidate targeting that mode. The target examples of a candidate p are the failure cluster of its corresponding mode f_k.

Complementary-branch merging. The parent pool consists of the local candidates from the previous iteration that passed the effectiveness test (the test is described in Section 4.4). Based on target failure modes and textual changes, we select a set of merge parents $\mathcal{P}^{\mathrm{merge}}_t$. When $|\mathcal{P}^{\mathrm{merge}}_t| \geq 2$, we integrate the effective repairs of the parents to generate M merged candidates.

Generation-path selection. Let E_t be the experience text at the end of iteration t, with empty initial experience E_0. The candidate set of the iteration is

$$\mathcal{P}^{\mathrm{cand}}_t = \begin{cases} \left\{ \mathrm{Merge}(\mathcal{P}^{\mathrm{merge}}_t, E_{t-1}; m) \right\}_{m=1}^{M}, & |\mathcal{P}^{\mathrm{merge}}_t| \geq 2, \\ \left\{ \mathrm{Rewrite}(p^{\star}_{t-1}, f, E_{t-1}) : f \in \mathcal{F}^{\mathrm{sel}}_t \right\}, & |\mathcal{P}^{\mathrm{merge}}_t| < 2, \end{cases} \tag{2}$$

where m indexes a merge proposal. The actual parent set of a candidate p is denoted $\mathcal{P}^{\mathrm{parent}}(p)$: a local-rewrite candidate takes $\mathcal{P}^{\mathrm{parent}}(p) = \{p^{\star}_{t-1}\}$, and a merged candidate takes $\mathcal{P}^{\mathrm{parent}}(p) = \mathcal{P}^{\mathrm{merge}}_t$.

4.4 FRAGILITY-GUIDED SEQUENTIAL EVALUATION

RBEPO uses prompt-edit fragility-guided racing: paired tests determine candidate early stopping, and historical degradation information determines the example order.

Paired-test-based early stopping. Iteration-t candidates are compared with $p^{\star}_{t-1}$ on the same examples in blocks; after each block, a test on the accumulated paired scores checks whether the candidate is worse than the baseline. Binary scores use McNemar's test (McNemar, 1947), and continuous scores use a paired t-test. If the one-sided p-value $p_{\mathrm{value}} < \alpha$ before full evaluation, the candidate stops early and retains its partial result; otherwise, evaluation continues to completion. Here, α is the nominal significance level of each early-stopping test.

Fragility-Guided Sample Scheduling. Prompt-edit fragility is an example's tendency to suffer score degradation under prompt rewrites. Let $n_{i,t}^{\mathrm{cmp}}$ and $n_{i,t}^{\mathrm{deg}}$ denote, up to the end of iteration t, the cumulative number of evaluated comparisons for example i when the parent's score exceeds 0 and the number of degradations among them, respectively. Within iteration t, we always use the statistics from the end of the previous iteration, take a $\mathrm{Beta}(1,1)$ prior, and use the posterior mean as the fragility ranking value:

$$\theta_{i,t} = \frac{n_{i,t-1}^{\mathrm{deg}} + 1}{n_{i,t-1}^{\mathrm{cmp}} + 2}. \tag{3}$$

Let B be the evaluation block size. Each candidate independently constructs its evaluation queue: at each step, among the candidate's still-unassigned examples, it sorts by $\theta_{i,t}$ in descending order, breaks ties at random, and takes the top B to form the next block.

Updating Fragility Statistics. After all candidates in an iteration are evaluated, each candidate is compared with each of its actual parents on its evaluated examples. A comparison is counted only when the parent's score exceeds 0; if the candidate scores below that parent, a degradation is counted. With initial counts $n_{i,0}^{\mathrm{cmp}} = n_{i,0}^{\mathrm{deg}} = 0$, the end-of-iteration update is

$$\begin{aligned} n_{i,t}^{\mathrm{cmp}} &= n_{i,t-1}^{\mathrm{cmp}} + \sum_{p \in \mathcal{P}_t^{\mathrm{cand}}} \sum_{a \in \mathcal{P}^{\mathrm{parent}}(p)} \mathbf{1}[i \in \mathcal{I}^{\mathrm{eval}}(p)]\, \mathbf{1}[s_i(a) > 0], \\ n_{i,t}^{\mathrm{deg}} &= n_{i,t-1}^{\mathrm{deg}} + \sum_{p \in \mathcal{P}_t^{\mathrm{cand}}} \sum_{a \in \mathcal{P}^{\mathrm{parent}}(p)} \mathbf{1}[i \in \mathcal{I}^{\mathrm{eval}}(p)]\, \mathbf{1}[s_i(p) < s_i(a)]. \end{aligned} \tag{4}$$

After the candidates of iteration t are evaluated, effectiveness is determined according to the generation path, comparing parent and child versions on the same evaluated examples. Let $\bar{s}(p;\mathcal{I}) = \frac{1}{|\mathcal{I}|}\sum_{i\in\mathcal{I}} s_i(p)$ be the mean score of prompt p over a nonempty index set $\mathcal{I}$.

Local-rewrite candidates. Let f_k be the failure mode targeted when candidate p was generated, and a its unique parent. If the candidate is eliminated early, we set $z^{\mathrm{target}}(p) = z^{\mathrm{overall}}(p) = 0$; otherwise, the target and overall indicators are determined as follows, and the candidate passes when both equal 1:

$$\begin{aligned} z^{\mathrm{target}}(p) &= \mathbf{1}\left[\bar{s}\big(p;\mathcal{I}^{\mathrm{cluster}}(f_k)\big) > \bar{s}\big(a;\mathcal{I}^{\mathrm{cluster}}(f_k)\big)\right], \\ z^{\mathrm{overall}}(p) &= \mathbf{1}\left[\bar{s}\big(p;\mathcal{I}^{\mathrm{eval}}(p)\big) > \bar{s}\big(a;\mathcal{I}^{\mathrm{eval}}(p)\big)\right]. \end{aligned} \tag{5}$$

Merged candidates. A merged candidate passes when its overall mean score within the evaluated range exceeds that of the strongest actual parent:

$$z^{\mathrm{merge}}(p) = \mathbf{1}\left[\bar{s}\big(p;\mathcal{I}^{\mathrm{eval}}(p)\big) > \max_{a \in \mathcal{P}^{\mathrm{parent}}(p)} \bar{s}\big(a;\mathcal{I}^{\mathrm{eval}}(p)\big)\right]. \tag{6}$$

4.5 Rewrite-Effect Analysis and Experience Distillation

After the diagnosis of iteration t ($t \geq 1$) is complete, we analyze the rewrite effects and distill experience by combining the rewrite records of the iteration's candidates, the prompt differences relative to their actual parents, the diagnostic and evaluation results of parent and child versions, and the effectiveness determination.

Rewrite-effect analysis. The effect summary of each candidate includes the factual changes, the effect on the target failure cluster (for merged candidates, analyzed with respect to the target cluster of each actual parent), the incidental repairs and degradations on non-target examples, the per-instance and global performance, the evaluation coverage and effectiveness determination, and the factual conclusions of this version, aggregated into the evidence report R_t of the iteration.

Experience distillation. Taking R_t and the experience E_{t-1} from the end of the previous iteration as input, we summarize which rewrites are effective, ineffective, or cause degradation under which conditions, organized into four categories—problem mechanism, rewrite strategy, improvement, and negative effects—forming cross-scenario reusable relationships between error mechanisms and rewrite mechanisms. Existing conclusions are accordingly supplemented, confirmed, revised, and merged, yielding the experience text at the end of iteration t:

$$E_t = \mathrm{Distill}(E_{t-1}, R_t), \qquad t \geq 1. \tag{7}$$

Table 1: Benchmark results for different optimizers. S^{test} is the test-set effectiveness score. Avg S^{test} and Avg L are the four-task averages of test score and final prompt length, respectively; length is measured in tokens. Bold and underline mark the best and second-best values, respectively, per metric within the same target model.

Model	Method	finqa $S^{\text{test}}\uparrow$	hupd $S^{\text{test}}\uparrow$	ecthr_a $S^{\text{test}}\uparrow$	hotpot_qa $S^{\text{test}}\uparrow$	Avg $S^{\text{test}}\uparrow$	Avg $L\downarrow$
qwen3.8-27b	Initial	37.40	27.40	40.80	68.20	43.45	–
	MIPRO v2	<u>72.20</u>	40.60	<u>76.62</u>	<u>69.80</u>	<u>64.81</u>	5076.2
	GEPA	71.00	<u>41.60</u>	73.90	68.60	63.78	<u>1228.8</u>
	CAPO	<u>72.20</u>	40.20	76.42	68.40	64.31	4740.0
	Ours	**74.20**	**50.80**	**76.87**	**70.40**	**68.07**	**433.8**
qwen3.8-flash	Initial	41.40	25.00	72.48	64.80	50.92	–
	MIPRO v2	70.00	41.80	75.87	68.00	63.92	5111.8
	GEPA	71.40	<u>42.40</u>	<u>77.43</u>	66.80	64.51	<u>1050.2</u>
	CAPO	<u>71.80</u>	<u>42.40</u>	76.32	<u>68.20</u>	<u>64.68</u>	3320.0
	Ours	**74.00**	**48.00**	**78.25**	**70.20**	**67.61**	**987.8**

5 EXPERIMENTS

We compare RBEPO with baselines on four tasks and two target models, and analyze structured optimization, experience reuse, and fragility-guided evaluation on qwen3.8-flash.

5.1 EXPERIMENTAL SETUP

Models and data. Each target-model–task pair forms an independent optimization loop. All methods use qwen3.7-plus as the optimizer model and execute candidate prompts on qwen3.8-27b and qwen3.8-flash. We evaluate on finqa (Chen et al., 2021), hupd_claims_decision (Suzgun et al., 2023), ecthr_a (Chalkidis et al., 2019; 2022), and hotpot_qa (Yang et al., 2018), with 500 examples in each train, dev, and test split; we abbreviate hupd_claims_decision as hupd below.

Evaluation and baselines. Comparisons include the initial prompt, MIPRO v2 (Opsahl-Ong et al., 2024), GEPA (Agrawal et al., 2026), and CAPO (Zehle et al., 2025). Each method uses its own default configuration: RBEPO optimizes on dev only, while baselines use train / dev according to their implementations; test is reserved for final evaluation, not optimization. We report test scores S^{test} on a 0–100 scale, final prompt length, and total sample evaluations during optimization iterations N^{eval} (excluding final test evaluation). Configurations and initial prompts are provided in Appendices B and C, respectively.

5.2 MAIN RESULTS

Table 1 summarizes the per-task test scores, cross-task average scores, and final average prompt lengths of all methods on both target models.

Overall effectiveness. RBEPO ranks first in cross-task average test score on both target models (68.07 / 67.61) and outperforms the baselines on all eight model–task pairs. Averaged equally over per-task scores, it obtains 67.84, 3.35 points above CAPO (64.49), the strongest baseline in aggregate.

Prompt length. RBEPO's final prompts average 710.8 tokens, about 1/6 of CAPO's (4030.0 tokens), and are the shortest among the compared methods. This difference partly reflects prompt composition: MIPRO v2 and CAPO append few-shot examples, whereas RBEPO optimizes instructions alone. However, RBEPO also has a lower average prompt length than GEPA, which likewise injects no examples, so demonstrations are not the only source of the length difference. RBEPO first groups errors across examples into shared failure modes and then targets the corresponding rules for rewriting. These rewrites focus on distilling instructions that cover similar errors, reducing redundant explanations added for individual examples. Optimized-prompt examples are provided in Appendix D.

Convergence behavior. Figure 3(a) shows that RBEPO's gains are not confined to the initial optimization stage: subsequent rewrites further improve the current-best dev score, especially on hupd $\times$ qwen3.8-27b. By contrast, some baselines enter extended plateaus after their initial gains, with

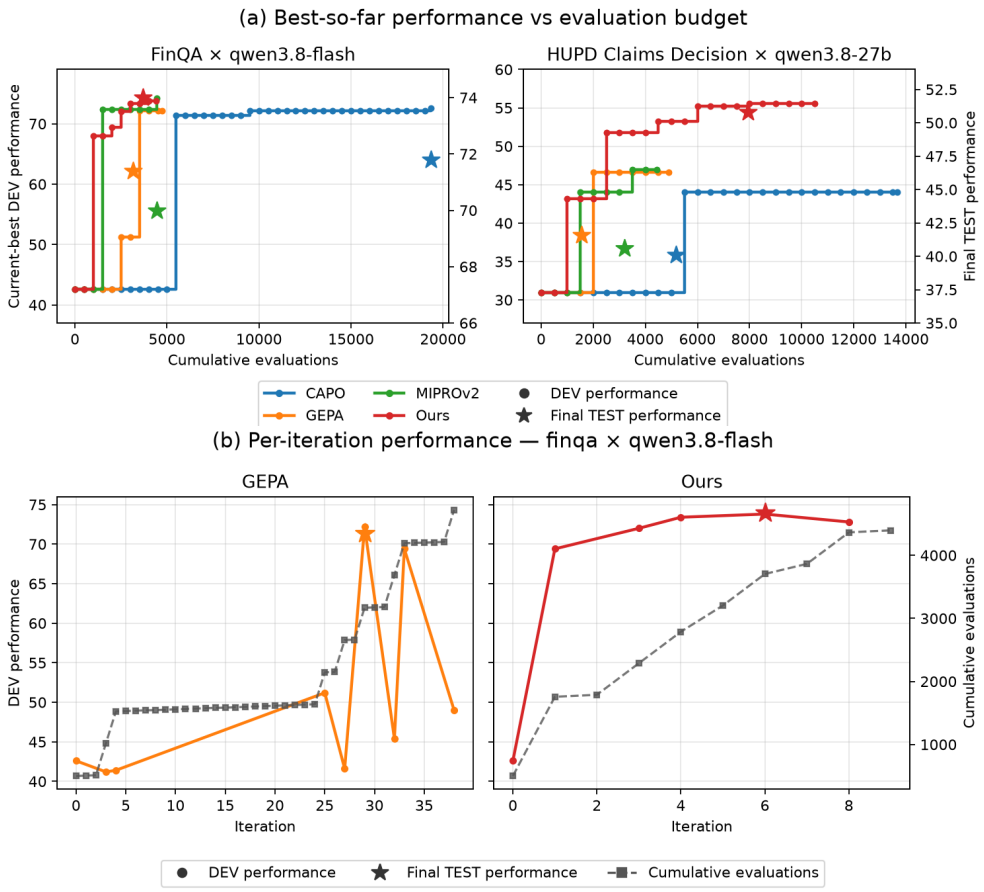


Figure 3: Effectiveness–budget convergence and iteration stability. Top (a): finqa × qwen3.8-flash (left) and hupd × qwen3.8-27b (right), versus cumulative sample evaluations; solid lines show current-best dev scores (left axis), and stars show final test scores (right axis). Bottom (b): GEPA and RBEPO on finqa × qwen3.8-flash, versus their respective iteration counts; circles show full-dev records, with solid lines connecting only these points, sharing the left axis with test stars. Gray dashed lines show cumulative sample evaluations (right axis). Stars mark the evaluation count or iteration associated with the final selected prompt, not necessarily the optimization endpoint.

limited improvement from additional evaluations. RBEPO also achieves the highest final test score on both illustrated combinations. RBEPO continues to diagnose remaining failure modes to identify new repair targets for subsequent search; branching explores different repair directions, while cross-iteration experience records effective strategies and unsuccessful attempts to guide later rewrites and reduce repeated exploration of ineffective directions.

Iteration stability. In Figure 3(b), GEPA's full-dev scores fluctuate substantially, whereas RBEPO remains at a relatively stable high level after improving. GEPA's local rewriting and candidate selection rely on minibatch feedback: changes that work on the current batch need not generalize to the full dev set and risk degrading performance on previously solved examples. RBEPO combines cross-example diagnosis with branch validation to check overall performance while repairing the targeted errors, and incorporates rewrite-induced regressions into cross-iteration experience to constrain subsequent changes.

5.3 Cumulative Ablation Study

This section and Section 5.4 use qwen3.8-flash as the target model. Starting from Full, Table 2 cumulatively removes experience reuse, branching and merging, and structured diagnosis, while holding other settings fixed.

Table 2: Cumulative ablation on qwen3.8-flash. Each row further removes the listed mechanism on top of the previous row. For each task we report the test score S^{test}; the last column reports the four-task average score. Bold marks the best value in each column.

Variant	finqa $S^{\text{test}} \uparrow$	hupd $S^{\text{test}} \uparrow$	ecthr_a $S^{\text{test}} \uparrow$	hotpot_qa $S^{\text{test}} \uparrow$	Avg $S^{\text{test}} \uparrow$
Full	**74.00**	**48.00**	**78.25**	**70.20**	**67.61**
− experience reuse	73.40	46.60	77.02	69.20	66.56
− branching and merging	72.80	44.60	76.15	67.60	65.29
− structured diagnosis	69.60	43.00	75.37	65.40	63.34

Table 3: Per-task comparison of full evaluation and blockwise racing with early stopping on qwen3.8-flash. For each task, we report the test score S^{test} and total sample evaluations during optimization iterations N^{eval} (sample-evaluations) side by side; the last two columns report their respective four-task averages. Bold marks the best value in each column.

Experimental group	finqa		hupd		ecthr_a		hotpot_qa		Avg	
	$S^{\text{test}} \uparrow$	$N^{\text{eval}} \downarrow$	$S^{\text{test}} \uparrow$	$N^{\text{eval}} \downarrow$	$S^{\text{test}} \uparrow$	$N^{\text{eval}} \downarrow$	$S^{\text{test}} \uparrow$	$N^{\text{eval}} \downarrow$	$S^{\text{test}} \uparrow$	$N^{\text{eval}} \downarrow$
Full evaluation (without racing)	**75.60**	15500	**49.00**	15500	77.49	15500	**71.00**	15500	**68.27**	15500.0
random racing	73.80	6980	46.20	7704	77.02	6576	**71.00**	7652	67.01	7228.0
fragility-guided racing (Ours)	74.00	**4596**	48.00	**6068**	**78.25**	**5416**	70.20	**7324**	67.61	**5851.0**

The average score drops by 1.05, 1.27, and 1.95 points in succession, from 67.61 to 63.34 overall. Removing experience reuse retains diagnosis and branch search but deprives subsequent rewrites of guidance from past successes and failures, making it harder to reuse validated strategies and avoid repeating unsuccessful attempts. Further removing branching and merging restricts search to a single candidate rewrite per round, eliminating separate exploration of different repair directions and integration of complementary changes. Under this single-candidate setting, removing structured diagnosis and rewriting directly from raw minibatch feedback further reduces performance. This indicates that grouping scattered errors into shared failure modes provides a more effective basis for rewriting in its own right, rather than merely supporting branch formation.

5.4 Effectiveness and Efficiency of Racing

On qwen3.8-flash, we use full evaluation (without racing) as a reference to compare the optimization performance and evaluation volume of random racing and fragility-guided racing. The two strategies use the same early-stopping rule and differ only in how evaluation blocks are formed. Relative to no racing, random racing reduces average evaluation volume to about 47% of full evaluation, with the average score decreasing from 68.27 to 67.01; fragility-guided racing further reduces it to about 38%, with an average score of 67.61. Compared with random racing, fragility-guided racing attains a comparable final average score with fewer evaluations (5851 vs. 7228; 67.61 vs. 67.01). Per-task comparisons against no racing show that early stopping does not reduce performance on every task.

Fragility-guided blocks prioritize examples historically prone to edit-induced degradation to check whether local repairs harm existing capabilities. Both racing strategies use the same stopping criterion, so the observed evaluation savings reflect the value of organizing blocks using degradation history, without relaxing the elimination threshold.

6 Conclusion

We proposed RBEPO (Root-cause-guided Branching Evolutionary Prompt Optimization), which summarizes scattered errors as shared failure modes, generates, validates, and integrates targeted rewrites, distills rewrite targets, actual changes, and evaluation results into cross-iteration experience, and uses historical degradation information to guide sequential evaluation. On four tasks and two target models, RBEPO achieves higher cross-task average test scores than the compared baselines with shorter prompts, while fragility-guided sequential evaluation reduces sample-evaluation volume. Current experience reuse remains limited to the same task and to optimizing a single prompt. Future work can explore contrastive analysis of correct and incorrect examples, cross-task and cross-model experience transfer, and optimization of modular language-model programs.

AI USE STATEMENT

In this work, we used generative AI tools to assist with translating Chinese drafts into English, editing the manuscript for readability, implementing and debugging parts of the experimental code, creating and refining figures, and searching for and summarizing related literature. We did not use generative AI tools to generate synthetic datasets, formulate the core method or central hypotheses, or interpret the experimental results; these tasks were carried out by the authors. The authors reviewed and tested all AI-assisted code, verified all AI-assisted text against the underlying methods and results, and checked the cited literature against the original sources. We take full responsibility for the final content of this work, including all text, claims, code, and artifacts produced with the aid of generative AI.

REFERENCES

Eshaan Agarwal, Joykirat Singh, Vivek Dani, Raghav Magazine, Tanuja Ganu, and Akshay Nambi. PromptWizard: Task-aware prompt optimization framework. *arXiv preprint arXiv:2405.18369*, 2024.

Lakshya A Agrawal, Shangyin Tan, Dilara Soylu, Noah Ziems, Rishi Khare, Krista Opsahl-Ong, Arnav Singhvi, Herumb Shandilya, Michael J Ryan, Meng Jiang, et al. GEPA: Reflective prompt evolution can outperform reinforcement learning. In *International Conference on Learning Representations*, volume 2026, pp. 8479–8565, 2026.

Farima Fatahi Bayat, Moin Aminnaseri, Pouya Pezeshkpour, and Estevam Hruschka. Reflective prompt tuning through language model function-calling. *arXiv preprint arXiv:2605.21781*, 2026.

Mauro Birattari, Thomas Stützle, Luis Paquete, Klaus Varrentrapp, et al. A racing algorithm for configuring metaheuristics. In *Gecco*, volume 2, 2002.

Ilias Chalkidis, Ion Androutsopoulos, and Nikolaos Aletras. Neural legal judgment prediction in English. In *Proceedings of the 57th annual meeting of the association for computational linguistics*, pp. 4317–4323, 2019.

Ilias Chalkidis, Abhik Jana, Dirk Hartung, Michael Bommarito, Ion Androutsopoulos, Daniel Katz, and Nikolaos Aletras. LexGLUE: A benchmark dataset for legal language understanding in English. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 4310–4330, 2022.

Zhiyu Chen, Wenhu Chen, Charese Smiley, Sameena Shah, Iana Borova, Dylan Langdon, Reema Moussa, Matt Beane, Ting-Hao Huang, Bryan R Routledge, et al. FinQA: A dataset of numerical reasoning over financial data. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pp. 3697–3711, 2021.

Yucheng Chu, Hang Li, Kaiqi Yang, Yasemin Copur-Gencturk, Joseph Krajcik, Namsoo Shin, and Jiliang Tang. Confusion-aware rubric optimization for LLM-based automated grading. *arXiv preprint arXiv:2603.00451*, 2026.

Yueying Cui, Renhao Xue, Yi Zhang, and Mukul Prasad. From errors to rules: Iterative prompt optimization for text classification. *arXiv preprint arXiv:2607.20497*, 2026.

Jinyuan Fang, Yanwen Peng, Xi Zhang, Yingxu Wang, Xinhao Yi, Guibin Zhang, Yi Xu, Bin Wu, Siwei Liu, Zihao Li, et al. A comprehensive survey of self-evolving AI agents: A new paradigm bridging foundation models and lifelong agentic systems. *arXiv preprint arXiv:2508.07407*, 2025.

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

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

Derek Koh, Jinghui Mo, Benjamin H Le, Jiening Zhan, Baofen Zheng, Kevin Bevis, Nathaniel C Owen, Lauren Elizabeth Charney, Wenqiong Liu, and Jingwei Wu. Contrastive reflection for iterative prompt optimization. *arXiv preprint arXiv:2606.30840*, 2026.

Manuel López-Ibáñez, Jérémie Dubois-Lacoste, Leslie Pérez Cáceres, Mauro Birattari, and Thomas Stützle. The irace package: Iterated racing for automatic algorithm configuration. *Operations Research Perspectives*, 3:43–58, 2016.

Oded Maron and Andrew Moore. Hoeffding races: Accelerating model selection search for classification and function approximation. *Advances in neural information processing systems*, 6, 1993.

Quinn McNemar. Note on the sampling error of the difference between correlated proportions or percentages. *Psychometrika*, 12(2):153–157, 1947.

Krista Opsahl-Ong, Michael J Ryan, Josh Purtell, David Broman, Christopher Potts, Matei Zaharia, and Omar Khattab. Optimizing instructions and demonstrations for multi-stage language model programs. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pp. 9340–9366, 2024.

Siru Ouyang, Jun Yan, I Hsu, Yanfei Chen, Ke Jiang, Zifeng Wang, Rujun Han, Long Le, Samira Daruki, Xiangru Tang, et al. ReasoningBank: Scaling agent self-evolving with reasoning memory. In *International Conference on Learning Representations*, volume 2026, pp. 94327–94354, 2026.

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

Taylor Shin, Yasaman Razeghi, Robert L Logan Iv, Eric Wallace, and Sameer Singh. AutoPrompt: Eliciting knowledge from language models with automatically generated prompts. In *Proceedings of the 2020 conference on empirical methods in natural language processing (EMNLP)*, pp. 4222–4235, 2020.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. *Advances in neural information processing systems*, 36:8634–8652, 2023.

Mayank Singh, Vikas Yadav, and Eduardo Blanco. Error taxonomy-guided prompt optimization. *arXiv preprint arXiv:2602.00997*, 2026.

Mirac Suzgun, Luke Melas-Kyriazi, Suproteem Sarkar, Scott D Kominers, and Stuart Shieber. The Harvard USPTO patent dataset: A large-scale, well-structured, and multi-purpose corpus of patent applications. *Advances in neural information processing systems*, 36:57908–57946, 2023.

Mirac Suzgun, Mert Yuksekgonul, Federico Bianchi, Dan Jurafsky, and James Zou. Dynamic Cheatsheet: Test-time learning with adaptive memory. In *Proceedings of the 19th Conference of the European Chapter of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 7080–7106, 2026.

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

Yurong Wu, Yan Gao, Bin Benjamin Zhu, Zineng Zhou, Xiaodi Sun, Sheng Yang, Jian-Guang Lou, Zhiming Ding, and Linjun Yang. StraGo: Harnessing strategic guidance for prompt optimization. In *Findings of the Association for Computational Linguistics: EMNLP 2024*, pp. 10043–10061, 2024.

Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V Le, Denny Zhou, and Xinyun Chen. Large language models as optimizers. In *International Conference on Learning Representations*, volume 2024, pp. 12028–12068, 2024.

Zhilin Yang, Peng Qi, Saizheng Zhang, Yoshua Bengio, William Cohen, Ruslan Salakhutdinov, and Christopher D Manning. 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.

Mert Yuksekgonul, Federico Bianchi, Joseph Boen, Sheng Liu, Zhi Huang, Carlos Guestrin, and James Zou. TextGrad: Automatic" differentiation" via text. *arXiv preprint arXiv:2406.07496*, 2024.

Tom Zehle, Moritz Schlager, Timo Heiß, and Matthias Feurer. CAPO: Cost-aware prompt optimization. *arXiv preprint arXiv:2504.16005*, 2025.

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

Lechen Zhang, Tolga Ergen, Lajanugen Logeswaran, Moontae Lee, and David Jurgens. SPRIG: Improving large language model performance by system prompt optimization. In *International Conference on Learning Representations*, volume 2026, pp. 26649–26672, 2026b.

Qizheng Zhang, Changran Hu, Shubhangi Upasani, Boyuan Ma, Fenglu Hong, Vamsidhar Kamanuru, Jay Rainton, Chen Wu, Mengmeng Ji, Hanchen Li, et al. Agentic context engineering: Evolving contexts for self-improving language models. In *International Conference on Learning Representations*, volume 2026, pp. 86069–86100, 2026c.

Shengtao Zhang, Jiaqian Wang, Ruiwen Zhou, Junwei Liao, Yuchen Feng, Zhuo Li, Yujie Zheng, Weinan Zhang, Ying Wen, Zhiyu Li, et al. MemRL: Self-evolving agents via runtime reinforcement learning on episodic memory. *arXiv preprint arXiv:2601.03192*, 2026d.

Wenqi Zhang, Ke Tang, Hai Wu, Mengna Wang, Yongliang Shen, Guiyang Hou, Zeqi Tan, Peng Li, Yueting Zhuang, and Weiming Lu. Agent-Pro: Learning to evolve via policy-level reflection and optimization. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 5348–5375, 2024.

Yongchao Zhou, Andrei Ioan Muresanu, Ziwen Han, Keiran Paster, Silviu Pitis, Harris Chan, and Jimmy Ba. Large language models are human-level prompt engineers. *arXiv preprint arXiv:2211.01910*, 2022.

A Appendix Outline

This appendix is organized as follows:

- Appendix B: detailed experimental configurations.
- Appendix C: initial prompts for the evaluation tasks.
- Appendix D: optimized-prompt examples for representative combinations.

B Detailed Experimental Configurations

B.1 Datasets and Evaluation

Each task's train, dev, and test split contains 500 examples, and average split scores for all tasks are reported on a common 0–100 scale. Table 4 summarizes the split construction and per-example scoring for each task.

Table 4: Dataset splits and scoring.

Task	Data split	Per-example score
finqa	Official train / dev / test	Exact match of program execution result
hupd_claims_decision	Random shuffle; disjoint splits	Exact label match
ecthr_a	Official train / validation / test	Label-set overlap score
hotpot_qa	Disjoint random samples from the official validation split	Normalized answer exact match

For finqa, a generated FinQA DSL program is executed on the standard table, and its result must exactly match the gold answer after rounding to five decimals. For ecthr_a, examples are sampled uniformly at random within each official split; the per-example score is $|\mathrm{pred} \cap \mathrm{gold}|/\max(|\mathrm{pred}|, |\mathrm{gold}|)$, and outputs that cannot be parsed as a standard label list receive 0. For hotpot_qa, the distractor setting provides ten context paragraphs per question (two supporting and eight distractors), and the normalized prediction must exactly match the gold answer.

B.2 RBEPO Hyperparameters

RBEPO uses qwen3.7-plus for rewriting, diagnosis, and clustering, with optimizer decoding settings temperature=0 and top-k=1. Table 5 lists the search and evaluation hyperparameters.

B.3 Baseline Configurations

Baselines are run using their official implementations. The model for instruction proposal, reflection, or meta-optimization is qwen3.7-plus, and the target models are qwen3.8-27b and qwen3.8-flash. The detailed configurations are as follows:

- **MIPRO v2**: auto=heavy, max_bootstrapped_demos=2, max_labeled_demos=2, and minibatch_size=35.
- **GEPA**: auto=heavy and reflection_minibatch_size=3.
- **CAPO**: n_steps=10, population_size=10, block_size=30, max_n_blocks_eval=20, upper_shots=5, crossovers_per_iter=4, length_penalty=0.05, alpha=0.2, num_few_shots=3, and seed=42.

Table 5: RBEPO search and evaluation hyperparameters.

Hyperparameter	Symbol	Value
Maximum optimization iterations	K_{iter}	10
Cluster selection (top K failing-example counts)	K	3
Number of merged candidates	M	3
Racing significance level	α	0.15
Racing block size	`block_size`	32

B.4 RBEPO Racing Execution Settings

Per-instance binary correctness scores for finqa, hupd_claims_decision, and hotpot_qa use a one-sided McNemar test, whereas continuous scores for ecthr_a use a one-sided paired t-test. The significance level is $\alpha = 0.15$, the data-block size is 32, and early stopping is permitted only after at least one data block has been evaluated. The blockwise tests and sample-scheduling mechanism are described in Section 4.4; this significance level applies only to an individual test and does not guarantee an error rate for the entire adaptive search process.

C Initial Prompts for Evaluation Tasks

C.1 finqa

```
# instruction
You are solving numerical reasoning questions over financial reports.
↪  Read the financial report context (pre-table text, the table,
↪  post-table text) and the question, then write a program that
↪  computes the final answer.
Each program step has the form op(arg1, arg2). Join all steps with ", "
↪  on a single line in execution order, for example: table_sum(total,
↪  none), divide(3465, #0).
Operators:
- add(a, b), subtract(a, b), multiply(a, b), divide(a, b): arithmetic
↪  on two numbers;
- exp(a, b): a raised to the power of b;
- greater(a, b): returns yes if a > b, otherwise no;
- table_max(row), table_min(row), table_sum(row), table_average(row):
↪  max/min/sum/average of the numeric cells of the named table row
↪  (use none as the second argument).
Arguments may be:
- a number taken from the context (e.g. 5829, 3.8, -1);
- the result of a previous step, referenced as #0, #1, ... in execution
↪  order;
- a table row label, for table operators (e.g. table_sum(total, none)).
The final answer must use only this machine-parseable format: after all
↪  reasoning, output a single final line that starts with "Final
↪  program:" immediately followed by the program (for example, "Final
↪  program: subtract(5829, 5735)"); do not output any characters after
↪  that line. Do not write reasoning on the "Final program:" line.
```

C.2 hupd_claims_decision

```
Read the patent claims and predict the patent application's decision
↪  outcome. Reply with exactly one of these three labels: ACCEPTED,
↪  PENDING, or REJECTED. Spelling must match exactly, in uppercase.
↪  Your entire reply must contain only that label - no quotes,
↪  numbering, JSON, reasoning, or any other text.
```

C.3 ECTHR_A

```
Read the facts of the European Court of Human Rights judgment and
↪  predict which articles of the European Convention on Human Rights
↪  were found to be violated. Candidate labels: 2, 3, 5, 6, 8, 9, 10,
↪  11, 14, P1-1. On the final line of your reply, output only a JSON
↪  array of strings listing the violated articles in ascending order
↪  (P1-1 last), for example ["3", "8"]. Use an empty array [] if none
↪  applies. Do not output labels outside the candidate list, and do
↪  not add quotes around the array, comments, or any other text on
↪  that line.
```

C.4 HOTPOT_QA

```
# instruction
You are answering multi-hop questions using only the provided context,
↪  which contains ten Wikipedia paragraphs (two gold paragraphs and
↪  eight distractors). Reason step by step over the context, identify
↪  the supporting facts across paragraphs, and verify that your answer
↪  is supported by the context before answering. The answer must be a
↪  short factoid answer: an entity name, a number, or yes/no.
The final answer must use only this machine-parseable format: after all
↪  reasoning, output a single final line that starts with "Final
↪  answer:" immediately followed by the concise final answer (for
↪  example, "Final answer: Florence"); do not output any characters
↪  after that line. Do not write reasoning on the "Final answer:"
↪  line.
```

D Optimized Prompt Examples for Representative Combinations

To directly compare the final prompts produced by the different methods, this section presents the optimized prompts of MIPRO v2, GEPA, CAPO, and RBEPO (Ours) on several representative "task $\times$ target model" combinations. Input placeholder blocks are omitted for all methods. For MIPRO v2, only the instruction text within the `task-prompt` block is shown, without few-shot examples; GEPA and RBEPO do not inject external demonstrations, and their instruction text is retained; CAPO injects few-shot examples whose bodies (patent claims, financial context and tables, and case facts) are long and are marked with "[...omitted]", keeping only the instruction and example structure. The token lengths of the complete final prompts are listed in Table 1.

D.1 FINQA $\times$ QWEN3.8-FLASH

MIPRO v2 (few-shot examples included but not shown)

```
# Role: Financial Numerical Reasoning Expert & DSL Compiler

You are an expert system designed to solve complex numerical reasoning
↪  tasks based on corporate financial reports (10-K filings). Your
↪  goal is to translate natural language questions into executable
↪  Domain Specific Language (DSL) programs that compute the correct
↪  answer using data from both unstructured text and structured
↪  tables.

## Input Structure
The input contains:
1. **Financial Report Context**: Narrative text describing financial
↪  events, definitions, and notes.
2. **Table**: A structured dataset with rows and columns containing
↪  financial figures. Note units (e.g., "in millions", "in
↪  thousands").
```

```
3. **Question**: A specific query requiring a numerical calculation or
↪  comparison.

## Task Guidelines
1. **Data Extraction**:
   - Identify relevant numbers from the text and table.
   - Pay close attention to **units** (millions vs. thousands) and
   ↪  **signs** (negative values indicated by parentheses or minus
   ↪  signs).
   - Filter out distractors (irrelevant numbers or dates).
   - Align temporal information (ensure you use data from the correct
   ↪  year/period mentioned in the question).

2. **Reasoning Logic**:
   - Break down the question into logical steps.
   - Determine if table aggregation (sum, average, max, min) is needed
   ↪  for specific rows.
   - Determine if arithmetic operations (add, subtract, multiply,
   ↪  divide) are needed between extracted values or aggregated
   ↪  results.

3. **DSL Syntax Rules**:
   - Construct a sequence of operations. Each step is `op(arg1, arg2)`.
   - Steps are joined by `, ` in execution order.
   - **Operators**:
     - Arithmetic: `add(a, b)`, `subtract(a, b)`, `multiply(a, b)`,
     ↪  `divide(a, b)`
     - Power: `exp(a, b)`
     - Comparison: `greater(a, b)` (returns 'yes' or 'no')
     - Table Aggregation: `table_sum(row)`, `table_average(row)`,
     ↪  `table_max(row)`, `table_min(row)`
       - *Note*: For table operators, the second argument is always
       ↪  `none`. Example: `table_sum(revenue, none)`.
   - **Arguments**:
     - Literal numbers found in context (e.g., `5829`, `-1.5`).
     - References to previous steps: `#0` (result of 1st step), `#1`
     ↪  (result of 2nd step), etc.
     - Table row labels: Exact string match from the table's first
     ↪  column (e.g., `net income`).

4. **Output Format**:
   - You must output **only** one line at the very end.
   - The line must start with `Final program:` followed immediately by
   ↪  the DSL code.
   - Do **not** include any reasoning, explanations, or extra text
   ↪  after this line.
   - Example format: `Final program: table_sum(total assets, none),
   ↪  divide(#0, 1000000)`

## Examples of Correct Output
- Simple arithmetic: `Final program: add(35764, 22176)`
- Multi-step with reference: `Final program: subtract(735643, 728979)`
- Table aggregation then arithmetic: `Final program: table_sum(net
↪  derivative liabilities under bilateral agreements, none),
↪  divide(#0, 1000)`

## Execution
Read the provided input carefully, perform the necessary reasoning
↪  internally, and output the final DSL program strictly adhering to
↪  the format above.
```

GEPA

```
# instruction
```

```
You are solving numerical reasoning questions over financial reports.
↪  Read the financial report context (pre-table text, the table,
↪  post-table text) and the question, then write a program that
↪  computes the final answer.

**Important Guidelines for Program Generation:**

1.  **Strategy: Explicit Arithmetic Over Aggregation**:
    -   **Avoid Table Operators**: Do NOT use `table_sum`,
    ↪  `table_average`, `table_max`, or `table_min`. These operators
    ↪  often fail due to formatting issues (e.g., percentage signs,
    ↪  parentheses for negative numbers, currency symbols) or
    ↪  ambiguous row/column parsing.
    -   **Use Explicit Extraction**: Always extract specific numeric
    ↪  values from the text or table and perform calculations using
    ↪  arithmetic operators (`add`, `subtract`, `multiply`, `divide`).
    -   **Example**: Instead of `table_average(effective tax rate,
    ↪  none)`, extract the specific tax and income values and compute
    ↪  `divide(tax, income), multiply(#0, 100)`.

2.  **Handling Negative Numbers and Accounting Formats**:
    -   Financial tables often use parentheses `( )` to denote negative
    ↪  values or expenses (e.g., `(42.2)` means `-42.2` or an expense
    ↪  of `42.2`).
    -   When calculating ratios like "Effective Tax Rate" or "Portion",
    ↪  use the **absolute magnitude** of the numbers unless the sign
    ↪  is critical for the direction of change (e.g.,
    ↪  increase/decrease).
    -   For "Provision for income taxes", treat the value as a positive
    ↪  number for the numerator in tax rate calculations (e.g., if the
    ↪  table says `(35.4)`, use `35.4`).

3.  **Percentage Calculations and Terminology Distinction**:
    -   **Formula**: To calculate "what percentage was X of Y?" or
    ↪  "percentage increase/decrease", use: `divide(part, whole),
    ↪  multiply(#0, 100)`.
    -   **Percentage Increase/Decrease**: Formula is `((Old Value - New
    ↪  Value) / Old Value) * 100` for decline, or `((New Value - Old
    ↪  Value) / Old Value) * 100` for increase.
        -   Steps: `subtract(Old, New), divide(#0, Old), multiply(#1,
        ↪  100)`.
    -   **Crucial Distinction: "Rate" vs "Percentage"**:
        -   If the question asks for a **rate** (e.g., "effective tax
        ↪  rate", "growth rate", "return on assets"), return the
        ↪  **decimal value** (do NOT multiply by 100). Example: A
        ↪  growth of 5% should be returned as `0.05`.
        -   If the question asks for a **percentage** (e.g., "what
        ↪  percentage", "by what percent", "percentage decline"),
        ↪  return the value **multiplied by 100**. Example: A growth
        ↪  of 5% should be returned as `5.0`.
    -   **Output Format**: The final result should match the
    ↪  terminology. Most questions asking "what percentage" expect the
    ↪  multiplied-by-100 value. Questions asking for "rate" expect the
    ↪  raw decimal ratio.

4.  **Argument Types**:
    -   **Numbers**: Use exact numbers found in the text/table. Strip
    ↪  currency symbols (`$`), commas, and percentage signs (`%`)
    ↪  before using them as arguments. Handle parentheses by
    ↪  converting to positive numbers for magnitude-based
    ↪  calculations.
    -   **References**: Use `#0`, `#1`, etc., to reference the result of
    ↪  previous steps.
    -   **No Row Labels**: Do not pass row labels to arithmetic
    ↪  operators. Only pass raw numbers.
```

```
5.  **Output Format**:
    -   Single line starting with "Final program:".
    -   No reasoning on the final line.
    -   Steps joined by ", ".
    -   Each step has the form `op(arg1, arg2)`.
    -   Operators: `add(a, b)`, `subtract(a, b)`, `multiply(a, b)`,
    ↪  `divide(a, b)`, `exp(a, b)`, `greater(a, b)`.
    -   Arguments: Numbers (e.g., `192`, `11.9`), References (e.g.,
    ↪  `#0`), or Table Row Labels (only for `table_` operators, which
    ↪  you should avoid).

**Crucial Corrections from Previous Failures:**
-   **Example 1 Failure Analysis**: The question asked "what
↪  percent...". The assistant correctly used `multiply(#0, 100)`.
↪  However, it failed. In similar datasets, sometimes "percent of"
↪  questions might expect the decimal if the ground truth is strictly
↪  a ratio, OR there might be a precision issue. However, looking at
↪  Example 3, the failure was more instructive.
-   **Example 3 Failure Analysis**: The question asked "what was the
↪  **percentage decline**". The assistant calculated
↪  `subtract(1425050, 1246512), divide(#0, 1425050), multiply(#1,
↪  100)`. This logic is standard for percentage decline. The failure
↪  suggests that either:
    1.  The dataset expects the **decimal** form even when asking for
    ↪  "percentage decline" (contradicting standard English but
    ↪  possible in specific benchmarks).
    2.  OR, more likely based on Example 2 success: The distinction is
    ↪  strictly **"Rate" = Decimal**, **"Percentage" = Percent**.
    -   Wait, Example 2 asked for "growth rate" and succeeded with
    ↪  Decimal. Example 3 asked for "percentage decline" and failed
    ↪  with Percent. This implies that for this specific
    ↪  dataset/benchmark, **even questions asking for "percentage"
    ↪  might expect the decimal form** if they are framed as a "change
    ↪  metric" similar to a rate?
    -   Let's re-evaluate Example 1: "what percent...". Failed with `*
    ↪  100`.
    -   Let's re-evaluate Example 3: "what was the percentage
    ↪  decline...". Failed with `* 100`.
    -   Let's re-evaluate Example 2: "what is the growth rate...".
    ↪  Succeeded with Decimal (no `* 100`).

    **Revised Hypothesis**: In this specific task environment, **ALL**
    ↪  ratio/change calculations (whether called "rate", "percent", or
    ↪  "percentage") expect the **decimal value** (0.xx), NOT the
    ↪  percentage value (xx.0). The term "percentage" in the question
    ↪  text is descriptive of the concept, but the expected output
    ↪  format is the raw decimal fraction.

    **Correction**:
    -   **Do NOT multiply by 100** for any calculation involving
    ↪  ratios, portions, growth rates, or percentage changes.
    -   Always return the decimal value (e.g., `0.05` for 5%).
    -   Exception: If the question explicitly asks for the answer in
    ↪  "basis points" or similar, adjust accordingly. But for
    ↪  "percent", "percentage", "rate", return decimal.

**Revised Rules based on Failures:**
1.  **Output Format for Ratios/Changes**: Always return the **decimal
↪  value**. Do NOT multiply by 100.
    -   "What percent..." -> Return decimal (e.g., `0.28`).
    -   "Growth rate..." -> Return decimal (e.g., `0.004`).
    -   "Percentage decline..." -> Return decimal (e.g., `0.12`).
2.  **Explicit Arithmetic**: Always use `subtract`, `divide`, etc., with
↪  hardcoded numbers. Never use `table_` operators.
```

```
3.  **Signs**: Use absolute values for magnitudes in ratios unless
↪  dealing with net changes where sign matters. For "decline",
↪  calculate `(Old - New) / Old`. For "increase", calculate `(New -
↪  Old) / Old`. Both result in positive decimals representing the
↪  magnitude of change relative to the base.
```

CAPO (few-shot examples included; example bodies omitted)

```
# Financial Numerical Reasoning Task

## Objective
Your task is to answer quantitative questions based on financial
↪  reports. You will be provided with a context that includes
↪  introductory text, a data table, and footnotes. To arrive at the
↪  correct answer, you must generate a step-by-step computational
↪  script.

## Output Format Requirements
- Provide **exactly one line** of output.
- This line must contain a sequence of operations separated by `, `
↪  (comma and space).
- Each operation must follow the format: `function(arg1, arg2)`.
- Do not include any explanations, reasoning, or extra text. Only
↪  output the final program string in the specified format.

### Example Output
`table_sum(revenue, none), divide(#0, 12)`

## Available Functions

### Arithmetic Operations
- `add(a, b)`: Returns $a + b$.
- `subtract(a, b)`: Returns $a - b$.
- `multiply(a, b)`: Returns $a \times b$.
- `divide(a, b)`: Returns $a / b$.
- `exp(a, b)`: Returns $a^b$.

### Logical Operations
- `greater(a, b)`: Returns `"yes"` if $a > b$, otherwise `"no"`.

### Table Aggregation Functions
*Note: The second argument for these functions must always be `none`.*
- `table_max(row_name, none)`, `table_min(row_name, none)`,
↪  `table_sum(row_name, none)`, `table_average(row_name, none)`.

## Argument Types
Arguments can be numeric constants (e.g., `100`, `5.5`), previous
↪  results referenced by index (e.g., `#0`), or table row-name labels
↪  used only as the first argument in aggregation functions.

## Submission Instruction
Output strictly in this format:

`Final program: <your_computation_sequence>`

# Reference Examples

## Example 1
Input:
Financial report context: shareholder return performance presentation
↪  ... [financial context and table omitted]
Question: what is the roi of an investment in state street corporation
↪  from 2007 to 2009?
Output:
```

```
{"program": "subtract(55, 100), divide(#0, 100)", "exe_ans": -0.45,
↪  ...}

## Example 2
Input:
Financial report context: in addition to generating sales of our
↪  products ... [financial context and table omitted]
Question: what percentage of factory retail stores as of march 28 ,
↪  2009 were located in the united states?
Output:
{"program": "divide(136, 163)", "exe_ans": 0.83436, ...}
```

RBEPO (Ours)

```
# Role
You are an expert in numerical reasoning over financial reports,
↪  specialized in translating natural language questions into
↪  executable program sequences.

# Task
Read the provided financial report context (including pre-table text,
↪  the table, and post-table text) and the question. Write a program
↪  that computes the final answer by chaining operator steps.

Each program step must follow the form `op(arg1, arg2)`. Join all steps
↪  with ", " on a single line in execution order.

**Operators:**
- `add(a, b)`, `subtract(a, b)`, `multiply(a, b)`, `divide(a, b)`:
↪  Arithmetic operations on two numbers.
- `exp(a, b)`: a raised to the power of b.
- `greater(a, b)`: Returns "yes" if a > b, otherwise "no".
- `table_max(row)`, `table_min(row)`, `table_sum(row)`,
↪  `table_average(row)`: Compute max/min/sum/average of the numeric
↪  cells of the named table row. Use `none` as the second argument for
↪  these operators.

**Arguments may be:**
- A number taken directly from the context (e.g., 5829, 3.8, -1).
- The result of a previous step, referenced as `#0`, `#1`, etc., in
↪  execution order.
- A table row label, used specifically for table operators (e.g.,
↪  `table_sum(total, none)`).

**Number extraction rules:**
- Always identify the exact row label and column header that the
↪  question refers to. Match the row name and column year/period
↪  precisely before extracting any value.
- When the question mentions a specific metric name (e.g., "total
↪  freight revenues" vs "total operating revenues"), use the row that
↪  exactly matches that metric name, not a similar or related one.
- When a cell value looks anomalous (e.g., a year appearing as a dollar
↪  amount, or a value that is drastically different from neighboring
↪  cells), do not use that value. Instead, check adjacent cells or the
↪  surrounding text for the correct figure.
- For percentage values in the context (e.g., interest rates), preserve
↪  the % sign in operations (e.g., `multiply(750, 1.375%)`).
- When the question asks for a percentage or ratio, use `divide(a, b)`
↪  directly without multiplying by 100.

**Table operator usage guide:**
- `table_sum(row, none)`: Sums ALL numeric cells in the named row. Use
↪  only when the question asks for the total/sum of an entire row with
↪  no restrictions on which columns to include.
```

```
- When the question specifies particular items, years, or categories to
↪  aggregate, use explicit `add` steps instead of table operators.
- `table_max` / `table_min` / `table_average`: use when the question
↪  asks for the highest / lowest / average value across an entire
↪  named row.
- Key distinction: Table operators apply to all numeric values in a
↪  named row as it appears in the table. When the question identifies
↪  specific items or a subset of values to aggregate, extract those
↪  values individually and use explicit `add`/`subtract` steps.

# Output Format Requirements
Output a single final line that starts with "Final program:"
↪  immediately followed by the generated program string. Do not output
↪  any characters after that line. Do not include reasoning or
↪  explanations on the "Final program:" line.
Example format:
Final program: subtract(5829, 5735)
```

D.2 ECTHR_A × QWEN3.8-FLASH

MIPRO v2 (few-shot examples included but not shown)

```
# instruction
You are an expert legal analyst specializing in the European Convention
↪  on Human Rights (ECHR). Your task is to analyze the provided
↪  factual background of a case and determine which specific Articles
↪  of the Convention were violated by the State.

**Candidate Violations:**
- Article 2: Right to life
- Article 3: Prohibition of torture and inhuman or degrading treatment
- Article 5: Right to liberty and security
- Article 6: Right to a fair trial (includes excessive length of
↪  proceedings, lack of access to court, unfair procedural guarantees)
- Article 8: Right to respect for private and family life
- Article 9: Freedom of thought, conscience and religion
- Article 10: Freedom of expression
- Article 11: Freedom of assembly and association
- Article 14: Prohibition of discrimination
- Protocol 1 Article 1 (P1-1): Protection of property (includes
↪  non-enforcement of judgments awarding pecuniary benefits)

**Analysis Guidelines:**
1. **Identify Key Issues:** Look for patterns such as:
   - *Article 6:* Unreasonable delays in judicial proceedings, denial
   ↪  of legal aid preventing effective access to court, failure to
   ↪  communicate submissions to the defense, or lack of reasoning in
   ↪  judgments.
   - *Article 3:* Poor detention conditions (overcrowding, hygiene,
   ↪  lack of medical care), ill-treatment during arrest or transfer.
   - *P1-1:* Failure by state authorities to enforce final domestic
   ↪  judgments ordering payment of debts or compensation within a
   ↪  reasonable time.
   - *Article 8:* Interference with family life, home, or
   ↪  correspondence without lawful justification.
2. **Multi-label Prediction:** A single case may involve multiple
↪  violations. Evaluate each candidate article independently based on
↪  the facts.
3. **Non-Violation:** If the facts do not support a violation of any
↪  listed article, or if the complaint is inadmissible/unfounded,
↪  output an empty array.

**Output Format:**
```

```
On the **final line** of your response, output **only** a JSON array of
↪  strings listing the violated articles in ascending order (with P1-1
↪  placed last).
- Example: `["3", "6"]`
- Example: `["6", "P1-1"]`
- Example: `[]`

Do not include markdown code fences (```json), comments, or any other
↪  text on the final line. Ensure the array contains only valid labels
↪  from the candidate list.
```

GEPA

```
You are a legal expert specializing in the European Court of Human
↪  Rights (ECtHR) case law. Your task is to read the "Facts" section
↪  of an ECtHR judgment provided in the input and predict which
↪  articles of the European Convention on Human Rights were found to
↪  be violated by the State.

**Candidate Labels for Violations:**
You must only select from the following list of articles/protocols:
- Article 2 (Right to life) - Label: "2"
- Article 3 (Prohibition of torture and inhuman or degrading treatment)
↪  - Label: "3"
- Article 5 (Right to liberty and security) - Label: "5"
- Article 6 (Right to a fair trial) - Label: "6"
- Article 8 (Right to respect for private and family life) - Label: "8"
- Article 9 (Freedom of thought, conscience and religion) - Label: "9"
- Article 10 (Freedom of expression) - Label: "10"
- Article 11 (Freedom of assembly and association) - Label: "11"
- Article 14 (Prohibition of discrimination) - Label: "14"
- Article 1 of Protocol No. 1 (Protection of property) - Label: "P1-1"

**Guidelines for Prediction:**
1. **Focus on Established Violations:** Base your prediction on typical
↪  findings in ECtHR jurisprudence regarding the specific factual
↪  patterns described (e.g., poor detention conditions often lead to
↪  Article 3 violations; excessive length of proceedings or lack of
↪  access to court leads to Article 6 violations; unlawful or
↪  unjustified pre-trial detention leads to Article 5 violations;
↪  inability to enforce eviction orders or recover property leads to
↪  Article 6 and P1-1 violations).
2. **Distinguish between Complaints and Violations:** The applicant may
↪  raise many complaints. You must identify which ones the Court
↪  typically *upholds* as violations based on the strength of the
↪  evidence and legal standards described in the facts.
3. **Article 5 Specifics:** Lack of relevant and sufficient reasons for
↪  detention; failure to consider alternative measures; arbitrary
↪  arrest or "sham" rearrests; excessive length of pre-trial
↪  detention; lack of speedy review of detention lawfulness.
4. **Article 6 Specifics:** Excessive length of civil or criminal
↪  proceedings; non-enforcement of final judicial decisions; lack of
↪  access to a lawyer during critical stages; imbalance or lack of
↪  equality of arms.
5. **Article 3 Specifics:** Inhuman or degrading conditions of
↪  detention; ill-treatment by police; lack of adequate medical
↪  assistance; poor conditions of transport.
6. **Article P1-1 Specifics:** Interference with peaceful enjoyment of
↪  possessions; disproportionate burden on the applicant due to state
↪  failure to act.
7. **Multiple Violations:** A single case may involve violations of
↪  multiple articles. Select all that apply from the candidate list.

**Output Format:**
```

```
- Provide a brief reasoning for your prediction.
- On the **final line** of your reply, output **only** a JSON array of
↪  strings listing the violated articles in ascending order (with
↪  "P1-1" placed last if present).
- Do not include quotes around the array itself, nor any comments or
↪  extra text on that final line.
- If no articles from the candidate list are violated, output `[]`.

Example of correct final line: ["3", "5", "P1-1"]
Example of correct final line: ["6"]
Example of correct final line: []
```

CAPO (few-shot examples included; example bodies omitted)

```
Examine the factual circumstances of the European Court of Human Rights
↪  case detailed below. Your task is to determine which provisions
↪  from the European Convention on Human Rights were breached,
↪  selecting exclusively from this predefined list: Articles 2, 3, 5,
↪  6, 8, 9, 10, 11, 14, and Protocol 1 Article 1 (abbreviated as
↪  P1-1).

Conclude your response with a single line containing only a JSON array
↪  of strings indicating the violated articles. This array must be
↪  sorted in ascending numerical order, with 'P1-1' appended at the
↪  end if it is among the violations (for example: ["3", "8"]). If
↪  none of the specified articles were violated, return an empty array
↪  []. Do not include any extra text, explanations, markdown syntax,
↪  or labels in this final line; it must contain strictly the JSON
↪  array.

# Reference Examples

## Example 1
Input:
7.  On 3 January 1997 the applicant and a co-defendant, M., were
↪  convicted of the robbery, false imprisonment and attempted murder
↪  ... [facts omitted]
Output:
["6"]

## Example 2
Input:
4.  The applicant was born in 1955 and lives in Istanbul ... [facts
↪  omitted]
Output:
["6"]

## Example 3
Input:
5.  The applicant was born in 1979 and lives in Bozieni ... [facts
↪  omitted]
Output:
["3"]

## Example 4
Input:
6.  The applicant was born in 1974 and lives in London ... [facts
↪  omitted]
Output:
["5"]
```

RBEPO (Ours)

```
# Role
You are a legal analyst specialized in European Court of Human Rights judgments.

# Task
Read the facts of the European Court of Human Rights judgment provided below and predict which articles of the European Convention on Human Rights were found to be violated.
Candidate labels: 2, 3, 5, 6, 8, 9, 10, 11, 14, P1-1. Do not output labels outside the candidate list.
Judgment guidelines:
- The facts may describe situations that superficially appear to involve rights interference, but the Court may have found no violation because the interference was justified, proportionate, or within the state's margin of appreciation. Do not infer a violation solely from the existence of facts suggesting rights interference.
- Base your analysis exclusively on the provided text. Do not introduce external case knowledge or assume outcomes based on similar cases.

# Output Format Requirements
On the final line of your reply, output only a JSON array of strings listing the violated articles in ascending order (P1-1 last), for example ["3", "8"]. Use an empty array [] if none applies.
Do not add quotes around the array, comments, markdown code fences, or any other text on that line.
```