RB-PROMPT: DETECTING AND REPAIRING PROMPT SATURATION IN REFLECTIVE OPTIMIZATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Prompt optimization adapts large language models to new tasks without changing their weights. Reflective optimizers rewrite prompts using feedback on model errors, but can keep adding instructions after performance stops improving. We call this *prompt saturation*: search spends evaluation budget on longer prompts with little gain. We address this problem with RB-PROMPT, which detects saturation from recent scores and instruction growth. It returns to a branch's best saved prompt and asks reflection to remove or combine unnecessary rules. Search continues with a penalty for added instruction length and Bayesian selection guided by past rewrite gains. We evaluate on HotpotQA, GSM8K, and an existing document extraction task at a major global automotive manufacturer, using three model families. Under matched task-evaluation budgets, RB-PROMPT has the highest or joint-highest mean in the main comparison across all nine settings, with the largest gains on HotpotQA and extraction. It achieves higher scores while using 40–45% fewer optimization tokens than TextBO-GEPA. Matched-state comparisons show larger repair gains on HotpotQA and document extraction. Adding repair to GEPA and TextBO-GEPA raises mean scores without changing their parent-selection rules. These findings support adaptive branch repair as a reusable component of reflective prompt optimization.

1 INTRODUCTION

Many of today's most capable large language models (LLMs) are offered as services. Users often access them through an API rather than by hosting their weights (Brown et al., 2020; Bommasani et al., 2021). This leaves users with only a few ways to adapt the model's behavior, and the most direct is to change the prompt. Prompt optimization requires no gradient access or model-training infrastructure. One effective approach is reflection. The optimizer runs the model on the task, collects plain-language descriptions of its errors, and asks an LLM to rewrite the prompt to avoid them. GEPA (Agrawal et al., 2026) follows this approach and reports better performance than reinforcement learning on several tasks with far fewer task evaluations. Early rewrites often improve performance by adding guidance that the original prompt lacked.

However, continued reflection can keep adding text after the main errors have been addressed. Later rewrites may add rules that are redundant, too specific, or in conflict with earlier rules. Performance can then stall or fall while the instructions grow. We call this pattern *prompt saturation*. ESPO (Liu et al., 2026a) reports a similar pattern in GEPA, with prompts growing up to three times longer without improving accuracy. GEPA retains earlier prompts and returns the one with the highest evaluation score (Agrawal et al., 2026). However, the optimizer still spends budget testing increasingly long prompts that provide no further gain.

Related prompt optimizers address only parts of this problem. VISTA (Liu et al., 2026b) can restart from an empty prompt, but uses a fixed probability to decide when. TextReg (Fu et al., 2026) checks growth after each edit and guides the next rewrite toward a shorter, more general prompt. ESPO diagnoses errors together and selects among complete candidate prompts to limit repeated rule accumulation. The challenge is therefore to decide when further rewriting is worth its evaluation cost. Intervening too early can interrupt useful progress, while intervening too late spends budget on rewrites that provide little or no benefit.

To address this gap, we introduce Reflective–Bayesian prompt optimization (RB-PROMPT), which compares recent score gains with instruction growth on each branch. When accepted rewrites add substantial text without a meaningful net gain, it returns to the best saved prompt on that branch.

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.

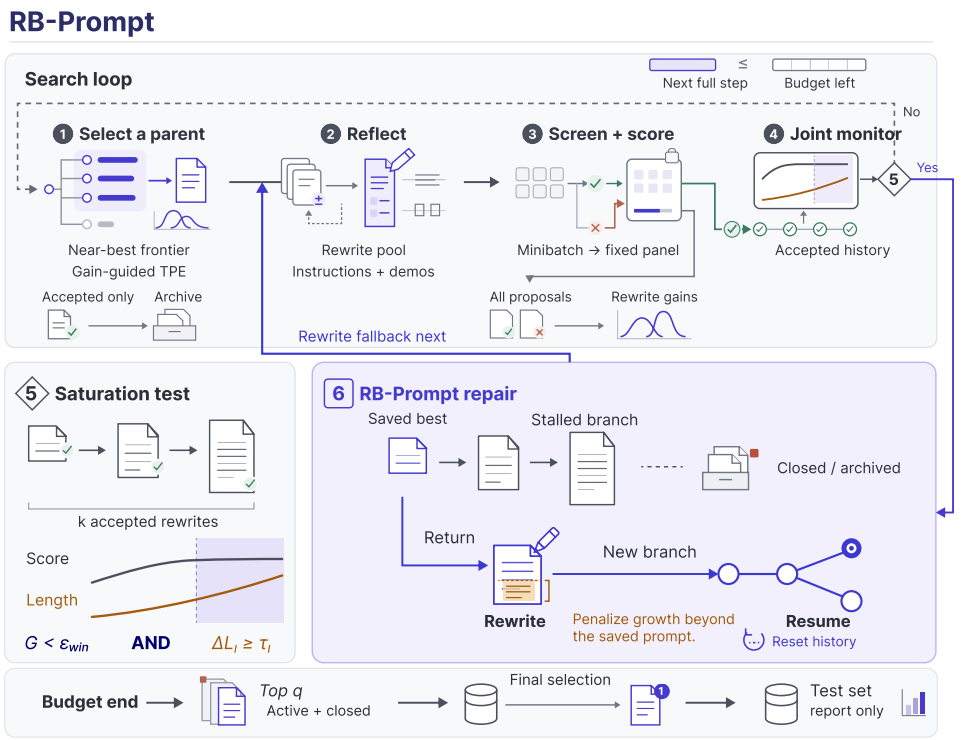

Figure 1: **Detecting and repairing prompt saturation with RB-PROMPT.** *1–4:* Search selects and rewrites prompts, scoring every proposal. *5:* The detector flags instruction growth with little score gain over k accepted rewrites. *6:* Repair restores the branch's best saved prompt, asking reflection to remove or combine unnecessary rules. Search resumes on a new branch, resetting detector history and penalizing growth. Final selection considers prompts from active and closed branches.

Reflection is then asked to remove or combine unnecessary rules, and added instruction length is penalized when new candidates are screened. The optimizer continues searching with the remaining budget. Throughout optimization, a Tree-structured Parzen Estimator (TPE) uses gains from earlier rewrites to choose which saved prompt to rewrite next. Figure 1 summarizes the method.

We test RB-PROMPT on HotpotQA (Yang et al., 2018), GSM8K (Cobbe et al., 2021), and an existing industrial document extraction task using Claude Sonnet 4.6 (Anthropic, 2026), GPT-4o (OpenAI, 2024), and Qwen2.5-VL (Bai et al., 2025). Against the main baselines at matched budgets, RB-PROMPT leads eight settings and ties TextReg on GPT-4o GSM8K. Its largest gains are on HotpotQA and document extraction. These tasks require combining facts across documents or filling large, nested schemas. They also show more instruction growth without improvement than GSM8K, giving repair more opportunity to help. RB-PROMPT achieves these gains while using 3–11% fewer optimization tokens than GEPA and 40–45% fewer than TextBO-GEPA (Kang & Yoganarasimhan, 2026). To test repair directly, we compare searches from the same saved state. Matched forks show larger gains on HotpotQA and extraction. We also show that adding repair to GEPA and TextBO-GEPA improves their mean scores on all three tasks without changing how they select prompts. The benefit of repair therefore extends to other selectors. This paper makes the following contributions:

- We measure how often prompt saturation occurs across tasks, models, and optimizers, and when it appears during search.
- We introduce RB-PROMPT, which uses recent scores and instruction growth to detect saturation and repair affected branches. Gains from earlier rewrites guide the choice of which saved prompt to rewrite next.
- We use matched controls to assess the roles of instruction growth, repair, and parent selection. These comparisons test repair against rollback alone and assess its benefit with other selectors.

2 Related Work

Reflective prompt optimizers use feedback on model errors to guide changes to the prompt. GEPA (Agrawal et al., 2026) rewrites prompts from execution feedback and favors parents that perform best on more search examples. TextGrad (Yuksekgonul et al., 2025), ProTeGi (Pryzant et al., 2023), and PromptAgent (Wang et al., 2024) use feedback to guide edits. VISTA (Liu et al., 2026b) tests likely causes of failure before rewriting. TextBO (Kang & Yoganarasimhan, 2026) compares proposed edits before task evaluation. RB-Prompt uses past rewrite gains to select saved prompts for rewriting. Instruction growth with little recent score gain triggers branch repair.

Table 1: Comparison with the closest work.

Method	What it checks or uses	What it does
VISTA	Failure hypotheses; fixed restart probability	Tests hypotheses; occasionally restarts from an empty prompt
CAPO (Zehle et al., 2025)	Task score and total prompt length	Races candidates and penalizes length
TextReg (Fu et al., 2026)	Instruction growth and rule specificity after an edit	Filters feedback and guides subsequent rewriting
ESPO[a]	Errors considered together and candidate-selection stability	Generates, consolidates, and selects complete prompts
RB-Prompt	Low score gain and instruction growth across accepted rewrites	Returns to a saved prompt and continues with repair

[a]We exclude ESPO from experiments because we could not locate its implementation.

RB-Prompt uses the history of a branch to decide when repair is needed. It requires both substantial instruction growth and little net gain on the same search examples across several accepted rewrites. This differs from TextReg's checks on individual edits and CAPO's length penalty throughout search (Table 1). In RB-Prompt, the length penalty begins only after repair is triggered. ESPO addresses repeated rule accumulation by changing how complete prompts are generated and selected; RB-Prompt intervenes when an existing branch shows growth without gain.

Other work addresses different aspects of search. TARE (Wan et al., 2026) studies whether small paraphrases change a prompt's performance, whereas our detector follows changes across successive rewrites. MIPROv2 (Opsahl-Ong et al., 2024) uses TPE to choose among fixed pools of instructions and examples. RB-Prompt uses TPE to select saved prompts for further rewriting, but its detector and repair can also be used with other parent-selection rules.

3 Method

3.1 Problem Setting and Reflective Search

Let Φ be a frozen system with one or more LLM calls. For input x and prompt p, it returns $\hat{y} = \Phi(x; p)$. A metric $\mu(\hat{y}, y) \in [0, 1]$ scores the output against its reference y, and $s(p; \mathcal{D})$ denotes the mean score on $\mathcal{D}$. We seek a prompt that maximizes expected score on unseen examples within a budget B of task evaluations. One task evaluation runs and scores a prompt on one labeled example.

We use four non-overlapping data groups. The rewrite pool $\mathcal{D}_{\text{rw}}$ supplies minibatches for feedback and demonstrations. The fixed score panel $\mathcal{D}_{\text{sc}}$ supplies scores for the Pareto frontier, parent selection, and saturation detection. These search scores are not independent validation evidence. After search, we use $\mathcal{D}_{\text{sel}}$ to choose from at most q saved prompts and $\mathcal{D}_{\text{test}}$ to evaluate the returned prompt. The budget $B = B_{\text{search}} + B_{\text{select}}$ covers search and final selection (Appendix A.3).

A prompt $p = (I, E)$ contains a mutable instruction I and up to m demonstrations E. Instruction length $L_I(p) = T(I)$ counts tokens using the same tokenizer for every model. Fixed task text and inputs are excluded because reflection cannot rewrite them. We record demonstration length L_E and total rendered mutable-prompt length L_P separately. Appendix A specifies the rendering rules.

The archive $\mathcal{P}$ stores accepted prompts, each linked to its parent. A prompt's branch is the path from the initial prompt or most recent repair point to that prompt. On the same rewrite minibatch, a child

must have a higher active screening score, or the same score with a shorter instruction. We form the Pareto frontier $\mathcal{F}$ from active prompts. A prompt is excluded from the frontier if another active prompt performs at least as well on every score-panel example and better on at least one. Prompts on closed branches remain in the archive for final selection.

To create a child, the optimizer runs a parent $p = (I, E)$ on a fresh minibatch $M \subset \mathcal{D}_{\mathrm{rw}}$. A reflection LLM uses the outputs and written feedback t_j to rewrite the instruction:

$$I' = \textsc{Reflect}(I, \{(x_j, \hat{y}_j, t_j)\}_{j \in M}) \,. \tag{1}$$

Successful rewrite-pool outputs can also become demonstrations, showing useful behavior instead of adding another instruction. Demonstrations cannot use score-panel, final-selection, or test examples. Appendix A gives the full demonstration rule.

3.2 Detecting Prompt Saturation

The detector tracks each branch separately. Let $\mathcal{D}_{\mathrm{sc}} = \{(x_i, y_i)\}_{i=1}^{n_{\mathrm{sc}}}$, and let $p_{b,r}$ be the r-th accepted prompt on branch b. After at least k accepted rewrites, we measure the net gain over the last k rewrites on the fixed score panel:

$$G_{b,r} = \frac{1}{n_{\mathrm{sc}}} \sum_{i=1}^{n_{\mathrm{sc}}} \left[\mu\big(\Phi(x_i; p_{b,r}), y_i\big) - \mu\big(\Phi(x_i; p_{b,r-k}), y_i\big) \right] . \tag{2}$$

We flag the branch as saturated when

$$G_{b,r} < \epsilon_{\mathrm{win}} \quad \text{and} \quad L_I(p_{b,r}) - L_I(p_{b,r-k}) \geq \tau_I . \tag{3}$$

Repair requires little net score gain and substantial instruction growth over the same window. A score plateau alone does not trigger repair. Appendix A gives the threshold values and their rationale.

3.3 Repairing a Saturated Branch

When Eq. 3 holds, the optimizer returns to the branch's prompt with the highest average score on the score panel, breaking ties by shorter instructions. This fallback is $p_b^\dagger$. The old branch closes for parent selection, but its prompts remain available for final selection. The fallback starts a branch with an empty detector history. The next rewrite starts from this fallback and asks reflection to remove or combine unnecessary rules (Appendix A). Candidates on a repaired branch are screened with

$$s_\beta(p; M, p_b^\dagger) = s(p; M) - \beta \, \frac{\max\{0, L_I(p) - L_I(p_b^\dagger)\}}{L_I(p_b^\dagger)} . \tag{4}$$

$\beta \geq 0$ affects screening only; $\beta = 0$ leaves all other repair steps unchanged. Let $a_b(p; M)$ denote the active screening score: $s(p; M)$ on a normal branch and $s_\beta(p; M, p_b^\dagger)$ on a repaired branch. A child p' is accepted if $a_b(p'; M) > a_b(p; M)$, or if the scores tie and $L_I(p') < L_I(p)$. This rule lets an equally scoring shorter repair enter the archive, while score gains can justify added length. The same tie rule applies on normal branches. The detector becomes active after k accepted rewrites.

3.4 Gain-Guided Parent Selection

The selector uses gains from earlier rewrites, rather than current scores alone, to choose which saved prompt to rewrite next. Let $s_{\mathrm{sc}}(p) = s(p; \mathcal{D}_{\mathrm{sc}})$. If parent p_t produces child p_t^+ at step t, the observed rewrite gain is

$$\Delta_t^{\mathrm{rw}} = s_{\mathrm{sc}}(p_t^+) - s_{\mathrm{sc}}(p_t). \tag{5}$$

We record this gain for every proposed child, including those rejected by the minibatch screen.

Ranking by gain alone can favor weak parents with more room to improve. We therefore restrict selection to Pareto-frontier prompts whose mean score is within δ of the frontier's best:

$$\mathcal{F}_t^{\mathrm{eligible}} = \left\{ p \in \mathcal{F}_t : s_{\mathrm{sc}}(p) \geq \max_{\tilde{p} \in \mathcal{F}_t} s_{\mathrm{sc}}(\tilde{p}) - \delta \right\}, \tag{6}$$

where δ is fixed before the main runs.

We rank eligible parents with a Tree-structured Parzen Estimator (TPE; Bergstra et al., 2011). Each history record pairs a parent's features $z_i(p_i)$, saved before rewriting, with its observed gain Δ_i^{rw}. TPE splits these records at a fixed gain quantile and fits feature densities $\ell_t(z)$ and $h_t(z)$ to the high-gain and remaining groups, respectively. It then selects

$$p_t = \arg \max_{p \in \mathcal{F}_t^{\mathrm{eligible}}} \frac{\ell_t\big(z_t(p)\big)}{h_t\big(z_t(p)\big)}. \tag{7}$$

This ratio ranks parents rather than predicting an exact gain. During warm-up, we use GEPA-style coverage-weighted selection. Appendix A.2 gives the features, warm-up and tie-breaking rules, and selector-only study.

Algorithm 1 summarizes the procedure. Here, $\mathcal{H}$ stores rewrite-gain records, and p_{force} denotes the required parent after repair, or $\perp$ when none is pending.

Algorithm 1 RB-PROMPT procedure.

Score p_0 on $\mathcal{D}_{\mathrm{sc}}$; $\mathcal{P} \leftarrow \{p_0\}$, $\mathcal{F} \leftarrow \{p_0\}$, $\mathcal{H} \leftarrow \varnothing$, $p_{\mathrm{force}} \leftarrow \perp$
while the next complete step fits within B_{search} **do**
  $p \leftarrow p_{\mathrm{force}}$ if $p_{\mathrm{force}} \neq \perp$; otherwise $p \leftarrow \textsc{Select}(\mathcal{F}_t^{\mathrm{eligible}}, \mathcal{H})$ ▷ Eqs. 6, 7
  $p_{\mathrm{force}} \leftarrow \perp$; draw $M \subset \mathcal{D}_{\mathrm{rw}}$; form p' by reflection and demonstration selection ▷ Eq. 1
  $\Delta_t^{\mathrm{rw}} \leftarrow s_{\mathrm{sc}}(p') - s_{\mathrm{sc}}(p)$; $\mathcal{H} \leftarrow \mathrm{append}(\mathcal{H}, (z_t(p), \Delta_t^{\mathrm{rw}}))$ ▷ Eq. 5
  if $(a_b(p'; M), -L_I(p')) > (a_b(p; M), -L_I(p))$ **then** $\mathcal{P} \leftarrow \mathcal{P} \cup \{p'\}$; update $\mathcal{F}$ over active prompts
    if branch b is saturated **then** ▷ Eq. 3
      $p_b^\dagger \leftarrow \arg\max_{\tilde{p} \in b} s_{\mathrm{sc}}(\tilde{p})$ ▷ tie: smaller L_I
      close b; start a repaired branch at $p_b^\dagger$ with empty detector history; update $\mathcal{F}$ over active prompts; $p_{\mathrm{force}} \leftarrow p_b^\dagger$ ▷ Eq. 4
$\mathcal{Q} \leftarrow \textsc{Top}_q(\mathcal{P}; s_{\mathrm{sc}}, -L_I)$
$p_{\mathrm{out}} \leftarrow \arg\max_{p \in \mathcal{Q}} s(p; \mathcal{D}_{\mathrm{sel}})$ ▷ tie: smaller L_I
return p_{out}

4 TASKS AND DATASETS

We evaluate RB-PROMPT on HotpotQA, GSM8K, and a proprietary extraction task to examine whether saturation and the benefits of repair vary with instruction growth. All tasks use the non-overlapping data groups defined in Section 3.1; Appendix Table 4 lists their sizes.

HotpotQA (multi-step question answering). HotpotQA (Yang et al., 2018) requires combining facts from several Wikipedia documents. We use the distractor setting, where relevant documents are mixed with unrelated ones. Because official test labels are unavailable, we reserve 1,000 examples from the public validation split exclusively for final testing. We report answer exact match as the primary metric and answer F1 as a secondary metric.

GSM8K (mathematical reasoning). GSM8K (Cobbe et al., 2021) contains grade-school mathematics problems with numeric answers. We evaluate on the full 1,319-example official test split, scoring only the final numeric answer by exact match, regardless of any intermediate reasoning.

Heterogeneous document extraction (proprietary data). Our third task evaluates RB-PROMPT on an existing document extraction task at a major global automotive manufacturer. Using scanned documents provided by the company, we test whether prompt optimization can improve extraction for a real industrial workflow. Each multi-page PDF is rendered as page images and mapped to a large, deeply nested JSON output under a fixed schema. Documents vary in layout and contain visual noise, skew, and sections with separate headers, line items, and totals. We use PDFs with matching reference JSON, keeping all records from each PDF in the same data group. Appendix Table 4 shows the full partition. The PDFs and reference JSON cannot be released.

Every optimizer modifies only the instruction: demonstrations are disabled ($m = 0$, $L_E = 0$), and the schema and extraction pipeline remain fixed. The template, schema, glossary, chunk description, and document images are excluded from L_I, L_E, and L_P, but included in full-request cost.

Extraction metric. We score JSON outputs using fixed weights of 0.45 for structure, 0.40 for numbers, and 0.15 for text. Appendix E gives scoring details and compares alternative weights (Table 14). We choose these weights to prioritize structure and numeric accuracy because the extracted records must follow the schema and preserve quantities and totals. Structure checks fields and nesting; numbers earn partial credit based on relative error; text is normalized for spacing, case, and dates. We report component and combined scores.

5 Experimental Setup

Questions and models. We ask whether growth helps identify useful intervention points, whether repair improves on rollback or continued search, and whether its benefit carries across selectors. We run Claude Sonnet 4.6, GPT-4o, and Qwen2.5-VL-72B separately. Each model performs all required roles within its runs: task execution, feedback generation, rewriting, and proposal judging. Task execution uses temperature 0; reflection uses 0.7. Appendix A.3 specifies versions and decoding.

Methods and controls. The main comparison includes the starting prompt, MIPROv2, CAPO, TextReg, GEPA, TextBO-GEPA, and RB-PROMPT. Secondary baselines and adaptations are in Appendix A.4. All methods keep the same frozen task system. Additionally, we add the same detector and repair to GEPA and TextBO-GEPA to test whether repair depends on our selector.

To test whether instruction growth helps decide when to repair, we compare our detector with two score-only controls. The first removes the growth condition but keeps the same window, gain threshold, fallback, and repair. The second also delays triggers and randomly skips some, so repair happens at similar times and with similar frequency to our detector. We set these delays and skipping probabilities using separate search-only pilot runs. This tests whether tracking growth helps beyond simply repairing later or less often (Appendix B).

Selector control. To test how choosing the next prompt affects performance, we disable repair and compare four selection rules: TPE, coverage-weighted selection, uniform random selection, and choosing the highest-scoring prompt. Coverage-weighted selection favors prompts that score best on more score-panel examples within the eligible set. All four use the same eligibility rule (Eq. 6), rewriting method, demonstration rule, and evaluation schedule. Different choices can lead to different saved prompts, so we keep the eligibility rule fixed rather than requiring identical prompt sets. We also test TPE on the full frontier, without the score-margin filter, to measure the filter's effect (Appendix A.2).

Evaluation budgets. For each task and model, methods use the same data groups, starting prompt, model settings, and matched random seeds. They receive the same sequence of rewrite minibatches unless their example-selection rule requires otherwise. Search allows up to $2{,}000$ task evaluations on HotpotQA and GSM8K and 300 on extraction, including evaluations of rejected prompts. After search, we compare up to three saved prompts on the selection set and evaluate the chosen prompt on the separate test set. Appendix A.3 gives the data splits and budget details.

Prompt length and cost. We measure instruction, demonstration, and total prompt length using the same tokenizer across models. Only instruction length affects saturation detection and repair. Comparisons with shared demonstrations use the same number, token limit, and format. Optimization cost includes all input and output tokens used during search and final selection, including fixed context, demonstrations, reflection, proposal judging, and retries. Extra evaluations used to plot progress are recorded separately and never guide search (Appendix A.3).

Testing repair from the same starting point. To test whether repair helps after saturation, we run RB-PROMPT with repair disabled and save the search state when the detector first triggers. We use 16 eligible runs per task and model. From each state, we compare four options: continue rewriting; return to the fallback and rewrite normally; return to the fallback and apply repair; or stop at the fallback. The three continuing options use the same next eight rewrite minibatches and evaluation allowance, with no further repairs. This tests what helps at a detected saturation point, not how often saturation occurs (Appendix B).

Repeats and uncertainty. Main comparisons and controls use 16 matched runs per task and model; secondary comparisons use eight. We report differences between methods and 95% confidence intervals that account for variation across both runs and test examples. For extraction, we resample

whole PDFs rather than individual fields. We apply Holm's correction (Holm, 1979) to the three primary contrasts across all task–model settings, accounting for multiple testing across these 27 tests. Other comparisons are exploratory. Appendix C gives the statistical procedures and detailed results.

6 RESULTS

Performance across tasks. Table 2 compares the main methods under the same search budget and final-selection rule. RB-PROMPT leads eight of nine settings and ties TextReg on GPT-4o GSM8K. Relative to GEPA, it gains 7.4–10.0 points on HotpotQA, 6.0–10.0 on extraction, and 0.6–1.2 on GSM8K. It also scores higher than TextBO-GEPA in all nine settings. The gains are largest on HotpotQA and extraction; GSM8K remains close to TextReg.

Table 2: Held-out scores ($\times 100$), averaged over 16 runs. Bold marks the highest mean among these methods, including ties, not statistical significance. HQA: HotpotQA; Doc.: extraction.

	Claude Sonnet 4.6			GPT-4o			Qwen2.5-VL-72B		
Method	HQA	GSM8K	Doc.	HQA	GSM8K	Doc.	HQA	GSM8K	Doc.
Start	59.2	94.8	69.5	55.7	91.6	66.9	53.1	90.4	68.0
MIPROv2	64.1	96.6	76.4	60.9	92.9	74.5	58.2	91.8	77.1
CAPO	64.9	96.6	77.9	61.7	93.1	75.7	59.4	92.0	78.2
TextReg	66.0	97.4	78.9	62.8	**93.8**	76.2	60.7	93.4	79.0
GEPA	66.1	96.9	78.0	62.2	93.2	75.8	60.1	92.5	78.3
TextBO-GEPA	68.1	97.2	79.9	64.2	93.7	77.6	62.5	93.0	79.8
RB-PROMPT	**73.5**	**97.5**	**87.2**	**72.2**	**93.8**	**85.8**	**67.7**	**93.7**	**84.3**

Does repair help after prompt saturation? Matched forks compare repair with continued rewriting and rollback (Figure 2). Full repair improves on continued rewriting by 5.7–7.6 points on HotpotQA and 5.7–8.4 on extraction. Repair has higher test scores in 48/48 HotpotQA, 40/48 GSM8K, and 48/48 extraction forks (ties are not wins). Beyond rollback, repair adds 4.5–6.4 points on HotpotQA and 4.3–7.2 on extraction. GSM8K gains are smaller: 0.2–0.7 points over continuing and 0.1–0.6 over rollback; only Qwen's intervals exclude zero. Appendix Tables 9 and 6 give paired intervals, corrected tests, and all four paths, including stopping at the fallback.

Does instruction growth help decide when to repair? Score-only detection can trigger repair even when further rewrites would improve performance without it. Delaying repair and triggering it less often reduces these cases. Requiring instruction growth reduces them further. On HotpotQA and extraction, search improves within eight further rewrites after about 22% of triggers from the timing-matched score-only control, compared with 12–13% from our detector (Appendix Table 5). The difference is smaller on GSM8K: 33% versus 30%. These rates show how often search improves without repair; they do not show whether repair would have helped or harmed performance. In complete runs, our detector achieves higher mean scores than the timing-matched control on HotpotQA and extraction (see Appendix Table 9). Confidence intervals that also account for variation across test examples include zero in all six settings (Figure 2). Instruction growth therefore helps identify searches less likely to improve without repair.

Does repair transfer? Adding the same detector and repair to GEPA and TextBO-GEPA raises mean scores in all nine settings; five of six GSM8K intervals include zero (Figure 2). TextBO-GEPA with repair exceeds RB-PROMPT on two HotpotQA settings and GPT-4o GSM8K (Table 7).

Performance and optimization cost. Figure 3 compares test scores at four search budgets. At each budget, the reported cost includes the evaluations used to choose the final prompt. RB-PROMPT uses 3–11% fewer total optimization tokens than GEPA and 40–45% fewer than TextBO-GEPA at the full allowance. These savings do not imply it is the cheapest method: CAPO and TextReg cost less. On GSM8K, RB-PROMPT scores slightly higher than GEPA and TextBO-GEPA, but the score gaps are much smaller. Appendix Table 15 separates search from final selection.

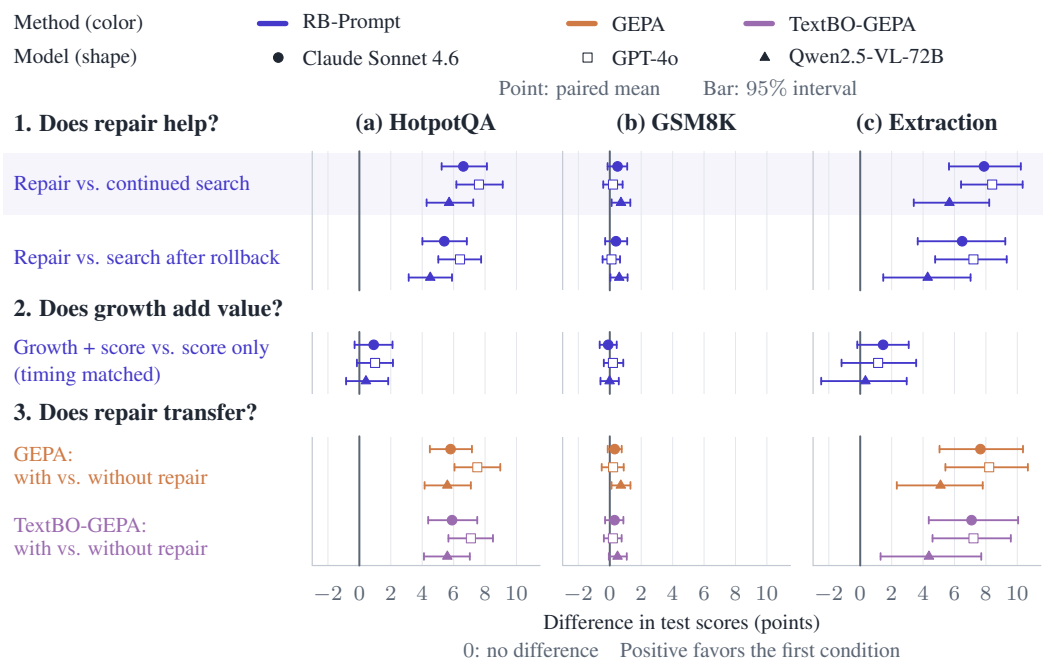

Figure 2: Repair's benefit varies by task. Bars show pointwise 95% crossed-bootstrap intervals. The first group forks the same saved state; the other groups compare complete matched runs.

7 Analysis and Discussion

A trigger is a warning, not proof of failure. Unintervened branch traces show instruction growth continuing after recent score gains become small (Appendix Figure 4). Table 11 reports how often the joint condition appears and how much budget remains. Because these traces are aligned on the detector's own condition, their plateau is partly selected by construction. They establish that the pattern occurs, not that every remaining evaluation is wasted. The matched forks in Figure 2 provide the stronger test: they compare what happens after different actions at the same state.

Repair across prompt-selection methods. On HotpotQA and extraction, repair improves on rollback followed by ordinary rewriting. Returning to the saved prompt alone therefore does not account for its gains. Adding repair to GEPA and TextBO-GEPA also raises mean scores while keeping their selectors fixed (Table 7). With repair disabled, selector differences are smaller, with no clear TPE advantage (Table 8). Together, these comparisons support using repair with different selectors, rather than attributing its benefit to TPE. They assess the repair package; they do not isolate the growth penalty's contribution.

Why gains differ across tasks. GSM8K starts near its final accuracy and has fewer growth-qualified triggers (Table 5). Its repair gains are smaller, and most paired intervals include zero. HotpotQA and extraction leave more room for instruction changes, consistent with their larger gains. The available room for improvement and trigger frequency help interpret this pattern, although these experiments do not separate their effects.

Choosing between score and cost. Figure 3 shows a tradeoff beyond the final score ranking. On GSM8K, TextReg reaches nearly the same mean score as RB-PROMPT at lower cost, making the available budget relevant to the choice of optimizer. RB-PROMPT's larger gains on HotpotQA and extraction give a stronger reason to use repair there. Shorter instructions accompany lower token use, but proposal counts, response lengths, demonstrations, and fixed inputs also affect cost. The savings reflect the full search and final-selection procedure; they cannot be attributed to instruction length alone.

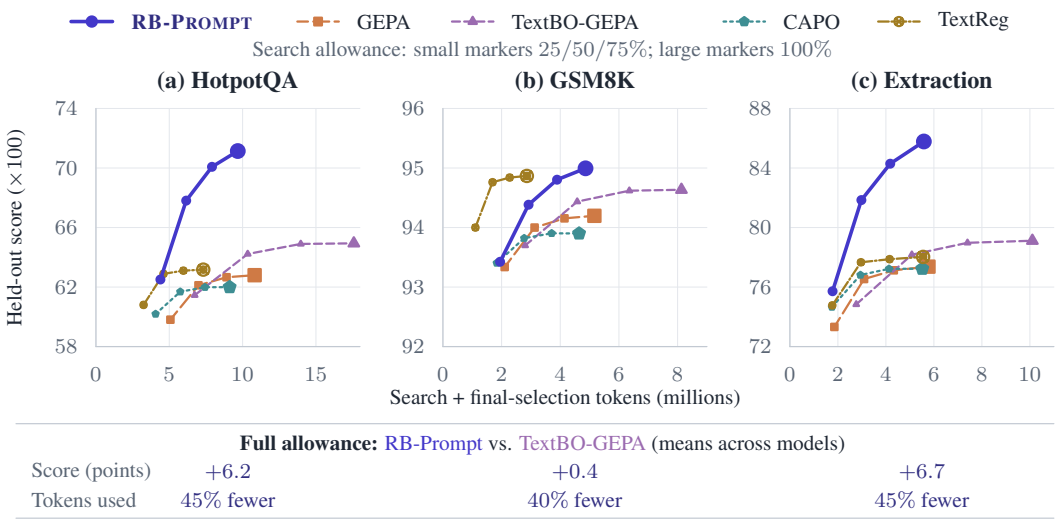

Figure 3: RB-Prompt scores higher with fewer tokens than GEPA and TextBO-GEPA on all three tasks. Points average three models and 16 runs/model; large markers match Table 2. Score and token ranges differ by task. The strip shows full-allowance differences, not significance. Only complete steps are charged; offline checkpoint tests do not guide search.

8 LIMITATIONS

The extraction task uses proprietary company documents from a real industrial workflow. We cannot release these documents or their reference JSON, which limits independent reproduction of the extraction results. Our evaluation covers two public English benchmarks and this extraction task, not open-ended generation, coding, or multilingual use.

The detector uses scores already collected during search to flag branches whose instructions keep growing without much improvement. This provides a simple rule for trying repair, but some flagged branches may recover on their own, while branches that stall without growing fall outside its scope. We therefore evaluate whether acting on the signal makes better use of the remaining budget. Because search repeatedly consults the same score panel, we use separate data for final selection and testing rather than treating search scores as evidence of performance on new examples. These comparisons separate the benefit of repair from the extra value of tracking growth, which remains uncertain against the timing-matched control in individual settings.

Each model performs all required roles, keeping model choice fixed when comparing optimizers. Differences across models, however, may reflect both task performance and the quality of reflection, which our experiments do not separate. Our efficiency results similarly describe the tested setup: we count tokens used throughout search and final selection, but the effect of these savings on cost and latency depends on pricing, caching, and the serving system.

9 CONCLUSION

Prompt saturation gives reflective search a practical intervention point: recent rewrites add instruction text without meaningful score gains. RB-PROMPT returns to a saved prompt and repairs the branch while search continues. Gains are largest on HotpotQA and extraction; GSM8K differences remain less certain. Matched forks and transfer comparisons support reconsidering growing, stalled branches, without isolating the growth penalty's contribution. The detector reuses search scores, so the decision to intervene requires no separate evaluation stage. Repair also preserves earlier prompts for final selection, allowing search to change direction without discarding prior progress. Future work should test broader tasks and budgets, and identify which repair steps account for the observed gains.

REFERENCES

Lakshya A. Agrawal, Shangyin Tan, Dilara Soylu, Noah Ziems, Rishi Khare, Krista Opsahl-Ong, Arnav Singhvi, Herumb Shandilya, Michael J. Ryan, Meng Jiang, Christopher Potts, Koushik Sen, Alex Dimakis, Ion Stoica, Dan Klein, Matei Zaharia, and Omar Khattab. GEPA: Reflective prompt evolution can outperform reinforcement learning. In *International Conference on Learning Representations (ICLR)*, 2026.

Anthropic. Claude Sonnet 4.6 system card. https://www.anthropic.com/claude-sonnet-4-6-system-card, 2026.

Shuai Bai, Keqin Chen, Xuejing Liu, Jialin Wang, Wenbin Ge, et al. Qwen2.5-VL technical report. *arXiv preprint arXiv:2502.13923*, 2025.

James Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for hyper-parameter optimization. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2011.

Rishi Bommasani, Drew A. Hudson, Ehsan Adeli, Russ Altman, Simran Arora, Sydney von Arx, Michael S. Bernstein, Jeannette Bohg, Antoine Bosselut, Emma Brunskill, et al. On the opportunities and risks of foundation models. *arXiv preprint arXiv:2108.07258*, 2021.

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, et al. Language models are few-shot learners. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2020.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.

Lucheng Fu, Ye Yu, Yiyang Wang, Yiqiao Jin, Haibo Jin, B. Aditya Prakash, and Haohan Wang. TextReg: Mitigating prompt distributional overfitting via regularized text-space optimization. *arXiv preprint arXiv:2605.21318*, 2026. URL https://arxiv.org/abs/2605.21318.

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 (ICLR)*, 2024.

Sture Holm. A simple sequentially rejective multiple test procedure. *Scandinavian Journal of Statistics*, 6(2):65–70, 1979. URL https://www.jstor.org/stable/4615733.

Enoch Hyunwook Kang and Hema Yoganarasimhan. TextBO: Bayesian optimization in language space for eval-efficient self-improving AI. In *ICLR Workshop on Recursive Self-Improvement*, 2026. URL https://openreview.net/forum?id=h0XYERyeL6.

Lihao Liu, Peng Tang, Kunwar Yashraj Singh, and Shabnam Ghadar. ESPO: Error-structured prompt optimization via diagnose, diversify, and stabilize. *arXiv preprint arXiv:2609.04197*, 2026a. URL https://arxiv.org/abs/2609.04197.

Shiyan Liu, Qifeng Xia, Qiyun Xia, Yisheng Liu, Xinyu Yu, and Rui Qu. Reflection in the dark: Exposing and escaping the black box in reflective prompt optimization. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 4: Student Research Workshop)*, pp. 76–109. Association for Computational Linguistics, 2026b. URL https://aclanthology.org/2026.acl-srw.8/.

OpenAI. GPT-4o system card. *arXiv preprint arXiv:2410.21276*, 2024.

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 (EMNLP)*, 2024.

Reid Pryzant, Dan Iter, Jerry Li, Yin Tat 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 (EMNLP)*, 2023.

Frank Wan, Lucheng Fu, Haoxin Liu, Yiqiao Jin, Hui Leong, Eric Jiang, Hejia Geng, Jinhe Bi, Yunpu Ma, Xiangru Tang, B. Aditya Prakash, Yizhou Sun, and Wei Wang. Beyond magic words: Sharpness-aware prompt evolving for robust large language models with TARE. In *International Conference on Learning Representations*, 2026. URL https://proceedings.iclr.cc/paper_files/paper/2026/hash/bbd35a35873e1e447b14b0932af87af0-Abstract-Conference.html.

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

Zhilin Yang, Peng Qi, Saizheng Zhang, Yoshua Bengio, William W. 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 (EMNLP)*, 2018.

Mert Yuksekgonul, Federico Bianchi, Joseph Boen, Sheng Liu, Pan Lu, Zhi Huang, Carlos Guestrin, and James Zou. Optimizing generative AI by backpropagating language model feedback. *Nature*, 639:609–616, 2025.

Tom Zehle, Moritz Schlager, Timo Heiß, and Matthias Feurer. CAPO: Cost-aware prompt optimization. In *Proceedings of the Fourth International Conference on Automated Machine Learning*, volume 293 of *Proceedings of Machine Learning Research*, pp. 18/1–45. PMLR, 2025. URL https://proceedings.mlr.press/v293/zehle25a.html.

Yongchao Zhou, Andrei Ioan Muresanu, Ziwen Han, Keiran Paster, Silviu Pitis, Harris Chan, and Jimmy Ba. Large language models are human-level prompt engineers. In *International Conference on Learning Representations (ICLR)*, 2023.

DISCLOSURE OF AI USE

In this work, we used generative AI tools to edit and polish the writing of this paper for clarity and readability. We have not used generative AI tools for any of the other tasks requiring disclosure. We have reviewed all AI-assisted edits to ensure they accurately reflect our intended meaning and did not introduce errors or unsupported claims. We take responsibility for the final content of this work, including text, claims, or artifacts produced with the aid of generative AI.

APPENDIX

A IMPLEMENTATION AND EVALUATION PROTOCOL

A.1 SAVED STATE, SCREENING, AND REPAIR

For each saved prompt, the archive stores its instruction, demonstrations, parent, branch, and scores on each score-panel example. These records let us trace the accepted rewrites from the branch's starting prompt or most recent repair point. When repair closes a branch, its prompts leave parent selection unless they also belong to an open branch. The fallback starts a new branch with its saved scores and demonstrations, but an empty detector history. It receives the next rewrite even if it is outside the Pareto frontier. All saved prompts remain in the archive for final selection.

For each rewrite, we compare the parent and child on the same minibatch, using the penalized score in Eq. 4 only on repaired branches. This allows a shorter repair to enter the archive even if it only matches the fallback's score. The penalty is scaled by the fallback's instruction length, which is nonzero. After each accepted rewrite, we check the detector once a full window is available. A window of k rewrites contains $k + 1$ prompts, including the root when the window starts there.

On public-task minibatches, exact match changes in steps of $1/8$. At $\beta = .02$, erasing a one-example advantage requires a difference in penalized excess length of at least $6.25L_I(p_b^\dagger)$. Below this, the penalty preserves non-tied preferences; tied raw scores already favor shorter instructions.

Repair request. When the score and growth conditions both hold, the next rewrite uses the request below. Later rewrites return to the ordinary request, while the growth penalty remains active on the repaired branch.

> Rewrite the saved instruction using the supplied execution feedback. Remove redundant, conflicting, or overly specific rules; combine rules that express the same requirement. Preserve useful task guidance and all fixed schema and output constraints. Do not add examples, labels, or facts from the score panel. Prefer a shorter instruction when it covers the same cases, but retain details supported by the feedback. Return only the revised instruction.

The feedback supplied with this request comes only from the current rewrite minibatch. The rewriting model receives no score-panel labels, outputs, or error explanations.

Prompt format and demonstrations. The part of the prompt that the optimizer can change contains the instruction, followed by complete demonstrations with labeled inputs and outputs. We count instruction tokens as L_I and demonstration tokens as L_E. We also tokenize the combined text, including separators, to obtain L_P. Because joining text can change how it is tokenized, L_P can differ from $L_I + L_E$. All three counts use `cl100k_base`. The fixed schema, task template, glossary, and document inputs are outside this block, but their tokens are included when we count the full cost of each request.

In the shared reflective setup, demonstrations come from successful outputs on earlier rewrite-pool examples. We rank them by score and break ties using the run's random seed. Each must have an exact-match score of 1, and its source example must be absent from the current minibatch. We keep at most four complete demonstrations within 1,024 tokens on each public task; extraction uses none. We do not shorten demonstrations to fit or draw them from the score panel, selection set, or test set. Methods that search for their own demonstrations keep that procedure under the same cap. TextReg changes only the instruction.

A.2 FIXED SETTINGS AND GAIN-GUIDED SELECTION

Table 3: Settings used in the experiments. Values separated by slashes refer to HotpotQA, GSM8K, and extraction, in that order. Detector and selector settings are shared across all tasks and models.

Parameter	Value
Window k; gain threshold ϵ_{win}	3 accepted rewrites; 0.05
Instruction-growth threshold τ_I; penalty β	40 tokens; 0.02
Eligibility margin δ; TPE warm-up	0.03; 8 transitions
TPE good fraction; prior mixture; bandwidth	0.25; 0.20; 0.15
Rewrite minibatch size	8 / 8 / 1
Demonstration count; token cap	4/4/0; 1,024/1,024/0
Final-selection candidates q	At most 3
Main / secondary repeats; fork horizon	16 / 8; 8 complete steps

The gain threshold $\epsilon_{\text{win}} = 0.05$ sets how much improvement counts as meaningful. On the 40-example public score panels, this means two more correct answers overall; on extraction, it means five points in the combined score. A gain equal to the threshold counts as meaningful. This rule guides repair decisions; it does not test statistical significance.

Before each rewrite, we record four features of the parent prompt for TPE: its mean score-panel score, the gain from the rewrite that produced it, its depth from the current branch root, and its instruction length. We scale these features as

$$z(p) = \left(s_{\text{sc}}(p),\ \frac{\Delta_{\text{in}} + 1}{2},\ \min\{d/32, 1\},\ \min\{\log(1 + L_I)/\log(8193), 1\} \right).$$

At a branch root, incoming gain and depth are both zero. The history includes every proposed rewrite, including rejected children. After eight recorded rewrites, we sort their gains and put the highest $\max(1, \lfloor 0.25|\mathcal{H}| \rfloor)$ in the high-gain group, breaking ties by chronological order. For each group, we fit a Gaussian kernel density to each feature with bandwidth 0.15, mixed with a 0.20 uniform prior on $[0, 1]$. Multiplying the four feature densities gives ℓ and h in Eq. 7; we use log densities for computation. TPE thus treats the four measured features independently, with the prior providing regularization, and does not fit prompt embeddings. If parents tie in the resulting ranking, we prefer higher score, then shorter instruction, then seeded order.

During warm-up, we select parents by coverage: a prompt's weight counts how many score-panel examples it scores best on within the eligible set, including ties. To compare selection rules, we disable repair and use the same eligibility rule across policies, keeping this definition of coverage where it is used. Once policies choose different parents, their saved prompts can also differ. The full-frontier TPE control removes the eligibility filter during both warm-up and learned selection. Table 8 reports this control separately.

A.3 DATA, MODEL VERSIONS, AND BUDGETS

Table 4: Data-group sizes and task-evaluation budgets. The groups do not overlap. On public tasks, rewriting, scoring, and final selection use training data. Testing uses public validation data for HotpotQA and the full official test split for GSM8K. All pages and fields from an extraction PDF stay in the same group.

Task	Rewrite	Score	Selection	Test	Search cap	Select cap
HotpotQA	160	40	300	1,000	2,000	900
GSM8K	160	40	200	1,319	2,000	600
Extraction	55	10	15	15	300	45

The extraction documents come from the industrial task in Section 4. We use the same data groups for every method and run. Run seeds change the rewrite order and sampled proposals, while the

test examples remain fixed. In GEPA and TextBO-GEPA, `trainset` refers to our rewrite pool and `valset` to our score panel. Neither includes the separate selection or test sets (Section 3.1; Table 4).

The model versions used in our runs are `claude-sonnet-4-6`, `gpt-4o-2024-11-20`, and `Qwen/Qwen2.5-VL-72B-Instruct`. Each model fills all required roles. Task execution and proposal judging use temperature 0; feedback and rewriting use 0.7. Output limits are 512 tokens for HotpotQA, 1,024 for GSM8K, and 4,096 for extraction; feedback and rewriting allow 2,048 tokens. Where a provider does not support seed control, identical responses across runs are not guaranteed. Every method receives the same rendered document pages and schema. We keep truncated or invalid outputs in the evaluation and score them as failures.

For RB-PROMPT, scoring the initial prompt costs n_{sc} task evaluations. Each rewrite then costs $2|M| + n_{\text{sc}}$: we evaluate the parent and child on the minibatch, then the child on the full score panel, even if screening rejects it. We reuse the parent's saved score-panel results. The budgets therefore allow 35 rewrites on each public task and 24 on extraction, using 2,000 and 298 evaluations, respectively. Search stops when the next complete step would exceed the budget. Baselines follow the same accounting rule, including evaluations used to compare candidates internally. Proposal and judging calls also count toward token use, even when they use no new labels.

After search, we rank saved prompts by mean score-panel score, breaking ties by shorter instruction. Up to three distinct rendered prompts then pass to final selection, where each is evaluated once on the full selection set. We return the highest-scoring prompt, breaking ties by shorter instruction and then seeded order, and evaluate only that prompt on the test set. The unchanged starting prompt requires no search and only one selection pass.

To plot progress, we repeat this selection and testing procedure at four fixed search checkpoints. These extra evaluations never guide optimization. Each point counts the search up to that checkpoint and one final-selection stage, as if search had ended there. The costs of the other three selection and test passes are recorded separately as work needed for the analysis.

A.4 BASELINES AND TASK ADAPTATIONS

We compare all methods using the same frozen task system and scoring code. Each keeps its own way of proposing prompts, but stops before a step would exceed the shared budget. The tables therefore compare methods adapted to our tasks and budgets; they do not reproduce the original papers' experiments or reuse their published benchmark scores.

MIPROv2 searches combinations of 16 instruction candidates and at most 16 demonstration sets. CAPO keeps its evolutionary proposals and candidate racing, which allocates evaluations among competing prompts. Its objective balances score and length without a hard instruction-length cap. TextReg regularizes each edit; it allows the screening score to fall by 0.01 on GSM8K and by zero on the other tasks.

GEPA selects parents from the frontier using coverage weights. TextBO-GEPA keeps that selector and uses four proposals with up to three local refinement steps. We count every proposal, judging, and task call. Variants marked `+ repair` add our detector and repair to these methods.

We also compare five secondary baselines over eight runs each. These are few-shot prompting; APE (Zhou et al., 2023), with 16 initial candidates; OPRO (Yang et al., 2024), using the best eight previous prompt–score pairs; EvoPrompt (Guo et al., 2024), with a population of eight and mutation and crossover; and VISTA, adapted to use four hypotheses per round. VISTA restarts with probability 0.1 and uses the same schema and evaluation accounting. These comparisons provide broader context and are outside the primary statistical tests.

B DETECTOR AND INTERVENTION CONTROLS

Matching when and how often repair occurs. The timing-matched control delays score-only triggers and randomly skips some of them so that repair occurs at similar times and with similar frequency to our detector. We choose these settings using 32 separate pilot runs per task and model, with access only to rewrite and score-panel information. We try delays of 0–3 accepted rewrites and

probabilities of keeping a trigger in $\{0.25, 0.5, 0.75, 1\}$. For each setting, we measure the absolute differences from our detector in trigger frequency and median trigger depth. We divide the depth difference by the maximum number of rewrite steps, add the two differences, and choose the setting with the smallest sum. Ties favor a shorter delay, then a higher probability of keeping a trigger.

When a delayed trigger comes due, we check that the score-only condition still holds, without using instruction length. All settings are fixed before the main runs, and pilot costs are recorded separately. This procedure matches timing and frequency approximately; the two detectors can still trigger at different search states.

Checking triggers without repair. To see whether flagged branches recover on their own, we run RB-PROMPT with repair disabled for 64 independent runs per task and model. A run is eligible only when eight further steps fit within the budget. We apply all detector rules to these same search histories and follow each triggered state for the next eight steps. A recovery means exceeding the best score-panel value at the trigger by at least ϵ_{win}.

Table 5 separates how often a first trigger occurs among eligible runs from how often search later recovers among triggered runs. For the timing-matched and joint detectors, the median first trigger occurs at depth seven, and trigger frequencies differ by about two percentage points. Recovery is measured on the score panel, so these rates describe continued search; they do not establish whether repair would help on held-out examples.

Table 5: Triggers and later recovery with repair disabled. Trigger is the percentage of 192 eligible runs per task that trigger the detector. Later gain is the percentage of triggered runs that recover within eight further steps. Recovery alone does not show whether repair would have helped or harmed.

	HotpotQA		**GSM8K**		**Extraction**	
Detector	Trigger	Later gain	Trigger	Later gain	Trigger	Later gain
Score only	85.9	34.5	62.5	42.5	90.6	34.5
Timing matched	68.8	22.0	31.2	33.3	76.6	22.4
Joint	67.2	12.4	29.7	29.8	75.0	13.2

Testing repair from the same starting point. For each task and model, we take the first 16 eligible runs in seed order where the joint detector triggers. Selection therefore does not depend on how much the score appears to have fallen. At each trigger, we copy the archive, saved scores, branch history, position in the minibatch sequence, and random state. This gives all four choices the same starting point. Continue keeps the active branch. Rollback starts from the same fallback as repair, but uses ordinary reflection and no growth penalty. Repair adds the repair request and the penalty in Eq. 4. Return fallback stops search and returns that prompt.

The three continuing choices retain RB-PROMPT's parent selector and receive eight complete steps: 448 additional task evaluations on each public task and 96 on extraction. We disable later triggers to isolate the effect of the first intervention. All four choices use the common final-selection rule, with only one candidate for Return fallback. Search before the trigger is a shared cost, so it does not count toward any savings from repair.

Controls over complete searches. Table 7 compares changes to detection and repair over full runs. No repair disables both. Score-only and timing-matched repair change only the trigger rule. Repair with $\beta = 0$ retains every full-repair step at each trigger, removing only the growth penalty (Table 12). Random restart replaces fallback repair with a fresh proposal from the rewrite pool. Always penalize applies the length penalty from the start, using the initial instruction as its fixed reference, with no trigger or rollback. Finally, `+ repair` adds our detector, fallback, request, and penalty to the named optimizer while keeping its original parent selector.

Table 6: Held-out scores ($\times 100$) after choosing different actions at the same saved states. Each column averages 16 matched states. Continue, Rollback, and Repair receive eight further steps; Return fallback stops immediately. These scores describe the saved-state comparison, not full runs.

	Claude Sonnet 4.6			GPT-4o			Qwen2.5-VL-72B		
Condition	HQA	GSM8K	Doc.	HQA	GSM8K	Doc.	HQA	GSM8K	Doc.
Continue	65.5	96.3	77.4	61.6	92.6	75.2	59.5	91.9	77.5
Rollback	66.7	96.4	78.8	62.8	92.7	76.4	60.7	92.0	78.9
Repair	72.1	96.8	85.3	69.2	92.8	83.6	65.2	92.6	83.2
Return fallback	65.9	96.3	77.7	62.0	92.6	75.8	59.9	91.9	78.5

Table 7: Held-out scores ($\times 100$) for controls over complete searches, averaged over 16 runs in each task–model setting.

	Claude Sonnet 4.6			GPT-4o			Qwen2.5-VL-72B		
Condition	HQA	GSM8K	Doc.	HQA	GSM8K	Doc.	HQA	GSM8K	Doc.
No repair	68.0	97.1	80.2	65.0	93.5	78.5	62.9	93.2	80.1
Score-only repair	72.0	97.2	85.2	70.6	93.3	84.2	66.7	93.5	83.1
Timing-matched repair	72.6	97.6	85.8	71.2	93.6	84.7	67.3	93.7	84.0
Random restart	68.0	97.0	80.3	65.4	93.3	78.4	62.1	92.9	78.7
Always penalize	68.6	97.3	81.2	66.0	93.6	79.3	62.7	93.2	79.7
GEPA + repair	71.9	97.2	85.6	69.7	93.4	84.1	65.7	93.2	83.4
TextBO-GEPA + repair	74.0	97.5	87.0	71.3	93.9	84.9	68.1	93.5	84.2

Table 8: Held-out scores ($\times 100$) for different parent-selection rules, averaged over 16 runs with repair disabled. The first four use the same eligibility rule. TPE eligible uses the No repair runs from Table 7.

	Claude Sonnet 4.6			GPT-4o			Qwen2.5-VL-72B		
Condition	HQA	GSM8K	Doc.	HQA	GSM8K	Doc.	HQA	GSM8K	Doc.
TPE eligible	68.0	97.1	80.2	65.0	93.5	78.5	62.9	93.2	80.1
Coverage eligible	67.6	97.0	80.0	64.6	93.4	77.9	62.5	93.1	79.6
Uniform eligible	67.3	96.9	79.6	64.3	93.3	77.8	62.2	93.0	79.4
Best eligible	67.7	97.0	79.9	64.7	93.4	78.1	62.6	93.1	79.6
TPE full	67.6	97.1	80.2	64.6	93.5	78.4	62.5	93.2	79.7

C Statistical Tests and Detailed Results

Our main statistical tests address three comparisons: repair with the joint detector versus timing-matched repair, repair versus continued search, and repair versus ordinary rollback. We test each comparison in all nine task–model settings, giving 27 tests. These comparisons are fixed by the study's questions before examining the results.

Let $d_{r,i} = s^A_{r,i} - s^B_{r,i}$ be the test-score difference between methods A and B on run r and example i. We average over runs and examples. A fork win has a run-mean difference above 10^{-12}; differences within that tolerance are ties. Denominators include all forks, including ties. Counts describe direction on the fixed test set, not statistical significance. Our crossed bootstrap independently resamples run and example IDs with replacement, sharing sampled indices between methods. The 2.5th and 97.5th percentiles of 1,500 draws give 95% intervals. Extraction resamples whole PDFs, keeping their fields together.

We also use an exact two-sided sign-flip test on the paired run means. This compares the absolute observed mean difference with those from all 2^{16} ways of assigning signs to the 16 run differences. The test holds the test examples fixed and assumes that signs are exchangeable under the null hypothesis of no difference. The bootstrap also accounts for variation across test examples, so its intervals and the sign-flip test can lead to different conclusions. We apply Holm's correction to all 27 primary tests together. Other comparisons and sensitivity analyses are exploratory.

Table 9 reports models separately. Result files include run scores, paired medians, intervals, and unadjusted and corrected p-values. Bold values in Table 2 mark highest means, including ties, rather than statistical significance.

GSM8K illustrates why both sources of uncertainty matter. Repair improves on continued search by 0.2–0.7 points and on rollback by 0.1–0.6 points. Against continuation, wins/ties/losses are 14/1/1 for Claude, 10/2/4 for GPT-4o, and 16/0/0 for Qwen. Both Qwen intervals exclude zero, while the Claude and GPT-4o intervals include zero. The sign-flip tests remain significant after Holm correction for Claude and Qwen, but not GPT-4o. For Claude, accounting for variation across test examples in the bootstrap makes the gains less certain.

Table 9: Paired score differences in points, with a 95% confidence interval for each comparison. † marks an exact paired sign-flip $p < 0.05$ after Holm correction over all 27 tests. These tests hold the test examples fixed. The result files give all p-values and individual run scores.

Task	Claude	GPT-4o	Qwen
Joint vs timing			
HotpotQA	+0.91 [-0.30,+2.11]	+1.00 [-0.16,+2.14]†	+0.42 [-0.84,+1.83]
GSM8K	-0.10 [-0.64,+0.44]	+0.20 [-0.38,+0.86]	-0.01 [-0.59,+0.57]
Extraction	+1.46 [-0.19,+3.09]	+1.14 [-1.19,+3.56]	+0.33 [-2.48,+2.96]
Repair vs continue			
HotpotQA	+6.62 [+5.24,+8.12]†	+7.61 [+6.17,+9.13]†	+5.71 [+4.28,+7.25]†
GSM8K	+0.49 [-0.14,+1.11]†	+0.20 [-0.42,+0.82]	+0.71 [+0.12,+1.31]†
Extraction	+7.88 [+5.65,+10.22]†	+8.40 [+6.42,+10.34]†	+5.68 [+3.41,+8.21]†
Repair vs rollback			
HotpotQA	+5.41 [+4.01,+6.84]†	+6.41 [+5.03,+7.75]†	+4.51 [+3.14,+5.90]†
GSM8K	+0.39 [-0.29,+1.11]†	+0.11 [-0.46,+0.65]	+0.60 [+0.04,+1.13]†
Extraction	+6.49 [+3.66,+9.23]†	+7.21 [+4.77,+9.32]†	+4.29 [+1.47,+7.02]†

D Saturation Traces and Sensitivity

To measure how often saturation appears, we apply the joint detector after search to branch histories collected without repair, using 16 runs per task, model, and optimizer. Table 11 reports trigger frequency and the fraction of the rewrite budget remaining when a trigger occurs. This fraction

Table 10: Held-out scores (×100) for secondary baselines, averaged over eight runs in each task–model setting. Few-shot extraction is omitted because the extraction task uses no demonstrations.

	Claude Sonnet 4.6			GPT-4o			Qwen2.5-VL-72B		
Condition	HQA	GSM8K	Doc.	HQA	GSM8K	Doc.	HQA	GSM8K	Doc.
Few-shot	63.2	95.7	–	60.0	92.0	–	57.3	90.9	–
APE	62.3	94.8	74.7	59.1	91.1	72.5	56.4	90.0	75.2
OPRO	62.9	95.4	75.2	59.7	91.7	73.5	57.0	90.6	75.8
EvoPrompt	63.5	96.0	75.9	60.3	92.3	73.8	57.6	91.2	76.5
VISTA adapted	64.4	96.9	76.9	61.2	93.2	74.9	58.5	92.1	77.4

counts proposed rewrites, including rejected ones, and is computed only for runs with a trigger. It measures the opportunity to change course; the remaining work may still be useful without repair.

Figure 4 shows how scores and instruction lengths change around the first trigger. We include only traces with three accepted rewrites on each side of that point. Because the trigger itself requires low gain and growth, the aligned curves partly reflect how these traces were selected. The matched comparisons from the same saved states test whether acting on the signal helps.

Table 11: Saturation in searches without repair. Trigger gives the percentage of 48 runs per task and optimizer with a trigger. Remaining gives the percentage of the proposed-rewrite budget left in triggered runs; this work could still be useful without repair.

	HotpotQA		GSM8K		Extraction	
Method	Trigger	Remaining	Trigger	Remaining	Trigger	Remaining
GEPA	81	37	38	33	85	42
TextBO-GEPA	62	40	35	38	88	43

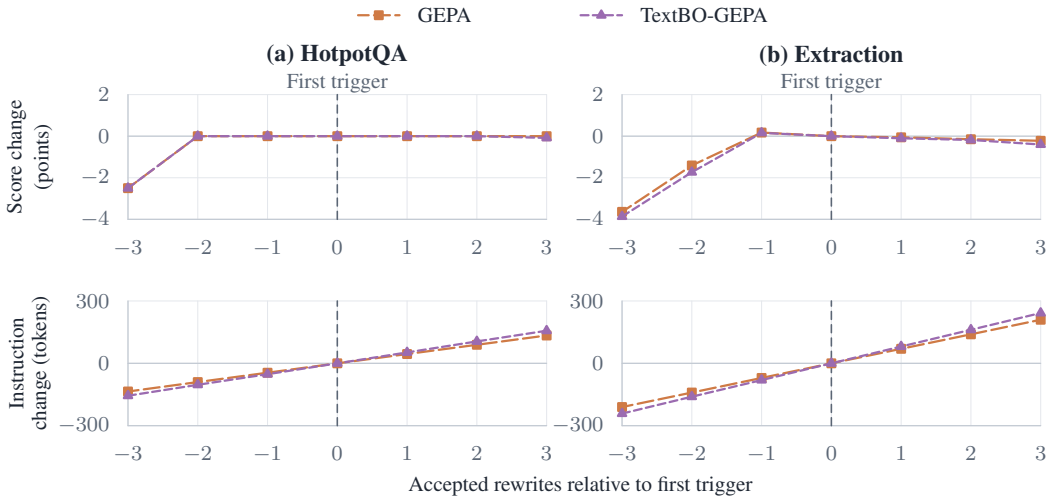

Figure 4: Branches without repair, aligned at their first joint trigger. Curves show mean changes in score-panel score and instruction length from that point. Every offset uses the same traces, each with the full displayed window and equal weight. Since alignment selects a period of low gain and growth, the plateau alone does not show that longer instructions cause lower scores. Columns show tasks and rows show score and length. Colors and markers match Figure 3.

To see how the settings affect performance, Table 12 varies one parameter at a time around the shared default, using eight runs per model. These changes make repair occur earlier or later, alter

the growth penalty, or change which parents are eligible. The results describe sensitivity within these ranges. We do not use them to choose separate settings for each task after seeing test scores, and they do not establish behavior outside the tested ranges.

Table 12: Change in held-out score points from the first eight matched default runs, averaged over three models. Each setting uses eight runs per model; these comparisons do not select the main settings.

Changed setting	HotpotQA	GSM8K	Extraction
$k = 2$	-0.57	+0.07	-1.21
$k = 4$	+0.13	-0.03	-0.52
$\epsilon_{\text{win}} = .025$	-0.48	-0.04	-0.82
$\epsilon_{\text{win}} = .075$	-0.28	-0.23	-0.60
$\tau_I = 20$	-0.38	-0.14	-0.93
$\tau_I = 80$	-0.27	-0.04	+0.03
$\beta = 0$	-0.68	-0.03	-1.13
$\beta = .04$	-0.37	-0.14	-0.11
$\delta = 0$	-0.38	-0.14	-0.41
$\delta = .06$	+0.22	-0.04	-0.38

E Secondary Metrics and Cost Accounting

Extraction scoring. We parse each JSON output and compare it with the reference under the fixed schema. Invalid JSON receives zero. Valid outputs receive separate scores for structure, numbers, and text. The structure score S is F1 over JSON paths and their types, including containers and indexed array positions. This penalizes both missing and extra structure. Array order follows the schema; we do not reorder entries to improve matches after seeing predictions.

For each reference numeric value v, the prediction $\hat{v}$ receives $\max\{0, 1 - |\hat{v} - v| / \max(|v|, 1)\}$. Missing or non-numeric predictions receive zero. We average these scores over reference numeric fields to obtain N. The text score T averages exact match after normalizing Unicode, case, and whitespace. We convert declared date fields to ISO dates when their meaning is unambiguous; missing text receives zero. Reference nulls count only toward structure.

We combine these scores for each PDF as $0.45S + 0.40N + 0.15T$, then average equally over PDFs. If a component has no eligible reference fields, we omit its weight and rescale the remaining weights to sum to one. Table 13 reports the separate components alongside the combined score. It also reports HotpotQA answer F1, computed over normalized answer tokens, as a secondary measure alongside exact match.

Table 13: Secondary metrics ($\times 100$). The combined extraction score uses $0.45S + 0.40N + 0.15T$ before rounding and matches Table 2.

Method	HQA F1	Structure	Numeric	Text	Combined
Claude					
GEPA	78.4	81.0	74.0	79.3	78.0
RB-Prompt	83.2	89.0	84.9	88.0	87.2
GPT-4o					
GEPA	76.9	79.2	71.5	77.3	75.8
RB-Prompt	83.0	87.8	83.3	86.7	85.8
Qwen					
GEPA	74.6	81.4	74.4	79.6	78.3
RB-Prompt	79.2	86.5	81.5	85.2	84.3

Changing metric weights. Table 14 reweights the same per-PDF component scores, averaging equally over PDFs and runs. It compares the default with equal weights and three nearby choices. Prompts remain those selected under the default metric; this tests evaluation sensitivity, not how different weights would change search or final selection.

Table 14: Extraction scores ($\times 100$) with different metric weights. Parentheses rank the six optimizers per model before rounding; ties share a rank. Weights are (structure, numeric, text); Equal uses $1/3$ each.

Method	(.45, .40, .15)	Equal	(.50, .35, .15)	(.40, .45, .15)	(.40, .35, .25)
Claude					
MIPROv2	76.4 (6)	76.6 (6)	76.8 (6)	76.0 (6)	76.6 (6)
CAPO	77.9 (5)	78.0 (5)	78.2 (5)	77.5 (5)	78.1 (5)
TextReg	78.9 (3)	79.0 (3)	79.2 (3)	78.5 (3)	79.0 (3)
GEPA	78.0 (4)	78.1 (4)	78.3 (4)	77.6 (4)	78.1 (4)
TextBO-GEPA	79.9 (2)	80.0 (2)	80.2 (2)	79.6 (2)	80.1 (2)
RB-PROMPT	87.2 (1)	87.3 (1)	87.4 (1)	87.0 (1)	87.3 (1)
GPT-4o					
MIPROv2	74.5 (6)	74.7 (6)	74.9 (6)	74.1 (6)	74.7 (6)
CAPO	75.7 (5)	75.8 (5)	76.1 (5)	75.3 (5)	75.9 (5)
TextReg	76.2 (3)	76.4 (3)	76.6 (3)	75.8 (3)	76.4 (3)
GEPA	75.8 (4)	76.0 (4)	76.2 (4)	75.5 (4)	76.0 (4)
TextBO-GEPA	77.6 (2)	77.8 (2)	78.0 (2)	77.3 (2)	77.8 (2)
RB-PROMPT	85.8 (1)	85.9 (1)	86.0 (1)	85.6 (1)	85.9 (1)
Qwen					
MIPROv2	77.1 (6)	77.3 (6)	77.5 (6)	76.8 (6)	77.3 (6)
CAPO	78.2 (5)	78.3 (5)	78.5 (5)	77.8 (5)	78.3 (5)
TextReg	79.0 (3)	79.1 (3)	79.3 (3)	78.7 (3)	79.2 (3)
GEPA	78.3 (4)	78.5 (4)	78.7 (4)	78.0 (4)	78.5 (4)
TextBO-GEPA	79.8 (2)	79.9 (2)	80.1 (2)	79.5 (2)	80.0 (2)
RB-PROMPT	84.3 (1)	84.4 (1)	84.5 (1)	84.0 (1)	84.4 (1)

Optimization tokens and costs. Table 15 separates search and selection tokens. Cost records further separate task execution, feedback and reflection, proposals and judging, and retries, including fixed context and images. Costs use each model's tokenizer or provider usage counts; prompt lengths use the shared tokenizer. Hosted cost estimates use uncached prices per million input/output tokens of \$3/\$15 for Sonnet 4.6 and \$2.50/\$10 for GPT-4o, as of September 9, 2026.[1]

Table 15: Search and final-selection tokens (millions), averaged equally across the three models.

	HotpotQA		GSM8K		Extraction	
Method	Search	Select	Search	Select	Search	Select
CAPO	6.66	2.46	3.70	0.94	4.90	0.62
TextReg	5.36	1.96	2.31	0.55	4.92	0.63
GEPA	7.57	3.25	4.03	1.13	5.10	0.67
TextBO-GEPA	14.20	3.38	6.98	1.15	9.42	0.69
RB-PROMPT	6.95	2.72	3.84	1.02	4.95	0.64

[1] Anthropic pricing; OpenAI pricing.