REFLECTIVE PROMPT TUNING: FROM DIAGNOSING RECURRING FAILURES TO HISTORY-AWARE REVISION

Anonymous authors
Paper under double-blind review

ABSTRACT

Large language models (LLMs) have become increasingly capable of following instructions and reasoning through complex problems, making prompting a flexible interface for adapting models without parameter updates. Yet prompt design remains labor-intensive and highly sensitive to formatting, phrasing, and instruction order, motivating automated prompt optimization methods that reduce this manual effort. Existing methods search over prompt candidates or revise prompts from feedback on individual examples or small batches, without maintaining a history of diagnosed failures across prompt versions. As a result, revisions are guided by local feedback rather than by which failures recur across examples and which persist or resolve across iterations. We propose Reflective Prompt Tuning (RPT), a diagnosis-driven prompt optimization framework that grounds each revision in a dataset-level view of the current prompt's recurring failures and in the optimization history. At each iteration, RPT evaluates the target model on the entire optimization set, critiques its incorrect responses, clusters the resulting critiques into recurring failure modes, and returns them with aggregate metrics as a structured report. An LLM optimizer then either revises the prompt, conditioned on this report and a memory of prior reports. RPT also optimizes for confidence calibration, using it in both its diagnostic reports and prompt selection. Across three reasoning tasks and four optimizer LLMs, we find that **(i)** RPT improves over initial prompts by up to 14.1 points and is the only method competitive on all three tasks, while also improving calibration and keeping prompts compact, **(ii)** full-set diagnosis, failure clustering, and memory each contribute, and memory lets the optimizer stop once performance plateaus, and **(iii)** revisions are aligned with the diagnosed failures, though weaker optimizers map distinct failures onto a few generic edits.

1 INTRODUCTION

Large language models (LLMs) increasingly rely on contextual prompting to adapt model behavior to downstream tasks (Lou et al., 2024; Wei et al., 2022; Kojima et al., 2022). Prompts specify task conditions, objectives, and constraints without modifying model parameters, enabling rapid adaptation across applications (Sahoo et al., 2025; Schulhoff et al., 2025). Despite this flexibility, prompt design remains a major bottleneck. Drafting an effective prompt is often a manual, trial-and-error process in which single observations are easily over-generalized, and fixing some failures reintroduces others (Zamfirescu-Pereira et al., 2023; Knoth et al., 2024). Prompt quality is also unstable, since LLMs are sensitive to minor choices of formatting, phrasing, and ordering (Zhuo et al., 2024; Sclar et al., 2024). These challenges have motivated automated prompt optimization methods that iteratively propose or revise prompts against measured task metrics (Ramnath et al., 2025).

The current state of the art increasingly uses textual feedback to guide prompt optimization (Shinn et al., 2023; Yuksekgonul et al., 2025; Agrawal et al., 2026). In this paradigm, an optimizer inspects signals such as execution traces or evaluator feedback, and proposes prompt revisions. However, prompt optimization faces four challenges, and existing methods address them only partially. First, individual failures can be noisy or idiosyncratic, and the same underlying deficiency surfaces in many forms, so the patterns that should drive revision emerge only across many examples. Updates are nonetheless driven by individual examples (Zhang et al., 2026) or minibatch subsets (Opsahl-Ong et al., 2024; Agrawal et al., 2026; Yuksekgonul et al., 2025), making optimization sensitive to local rather than recurring failures. Second, optimization is inherently sequential: an edit may resolve one

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.

issue while introducing another, some failures persist across several revisions, and the value of an intervention becomes clear only at later evaluations. Existing methods do not carry this information forward: the optimizer never sees which failures earlier prompts produced, so failures that persist or resolve with a delay are not recognized as such. Third, an effective revision is not always an addition: the same diagnosis may call for rewriting, consolidating, or removing an instruction, whereas ACE (Zhang et al., 2026) appends entries to an itemized playbook and deliberately avoids rewriting it. Finally, prompt selection is typically driven by task performance alone, leaving reliability and deployment cost outside the criterion: GEPA (Agrawal et al., 2026) uses auxiliary signals but still selects prompts primarily on task performance, and ACE's playbook grows with the optimization set and is prepended to every test query.

To address these limitations, we propose Reflective Prompt Tuning (RPT; Figure 1), a diagnosis-driven prompt optimization framework in which an LLM optimizer is equipped with an external diagnostic function and a memory. Starting from a seed prompt, the optimizer calls the diagnostic function, which evaluates the target model on the *full optimization set*, critiques incorrect responses, clusters the critiques into *recurring failure modes*, and returns them with aggregate metrics as a structured report. Each report thereby pairs the evaluated prompt with the failures it produced and is appended to the memory. Conditioned on the current report and the memory, the optimizer then either stops or revises the prompt by adding, rewriting, consolidating, or removing instructions, and calls the diagnostic function again. RPT supports confidence-aware optimization while accounting for inference efficiency: calibration enters both diagnosis and prompt selection, which also penalizes prompt length. Modeled on a human prompt engineer's workflow, the optimizer reviews recurring failures and prior edits, then uses this accumulated evidence to decide whether and how to revise.

We evaluate RPT on three reasoning tasks: multi-hop reasoning over textual evidence with HotPotQA (Yang et al., 2018), mathematical reasoning with LiveBench-Math (White et al., 2025), and domain-specific numerical reasoning with Formula (Wang et al., 2025). Using GPT-4.1 as the target model and four proprietary optimizer LLMs, we compare RPT against state-of-the-art prompt optimization baselines: ACE (Zhang et al., 2026), GEPA (Agrawal et al., 2026), and MIPRO (Opsahl-Ong et al., 2024). RPT improves over initial prompts by up to 12.9 points on HotPotQA, 14.1 on LiveBench-Math, and 11.7 on Formula, and is the only method competitive on all three, using prompts at least 23 times shorter than ACE's in every setting; RPT gains grow with optimizer capability. It also improves calibration alongside task performance under every optimizer. Ablations show that full-set diagnosis, failure clustering, and cross-iteration memory each contribute, and optimization traces show that revisions are aligned with the diagnosed failures, though weaker optimizers collapse distinct failures into a few generic edits. Because diagnosis is exposed to the optimizer as a callable function rather than fixed in the optimization loop, the design extends to optimizers that choose not only how to revise but what evidence to gather, a direction we discuss in Appendix R.

2 PRELIMINARIES

LLM Prompting. For a fixed target model f, each instance input x is paired with a shared task-level prompt π specifying the task objectives and constraints, such as the output format. The model is commonly prompted to provide reasoning with its answer (Wei et al., 2022; Kojima et al., 2022); we additionally require a confidence estimate (Xiong et al., 2024). Given input x and prompt π, the model produces $f(\pi, x) = (r, \hat{y}, c)$, where r is the reasoning trace, $\hat{y}$ the final answer, and $c \in [0, 1]$ the verbalized confidence. Each input is paired with evaluator metadata m, such as a reference answer or scoring program, and a task metric $\mu(\hat{y}, m) \in [0, 1]$ scores the answer against this metadata.

Prompt Optimization. Prompt optimization starts from a seed prompt π_0 and optimizes it against one or more objectives defined over the task's distribution of instances (Ramnath et al., 2025). Most methods optimize a single objective, the expected task performance (Agrawal et al., 2026),

$$\pi^\star \;=\; \arg\max_{\pi} \; \mathbb{E}_{(x,m)\sim p_{\text{data}}}\big[\mu(\hat{y}, m)\big], \tag{1}$$

where $(r, \hat{y}, c) = f(\pi, x)$. We instead optimize prompts with respect to three objectives, which together promote *accuracy*, *reliability*, and *inference efficiency*:

1. **Task performance**, for accuracy: the prompted model should produce high-quality outputs on task instances, as in Eq. 1.

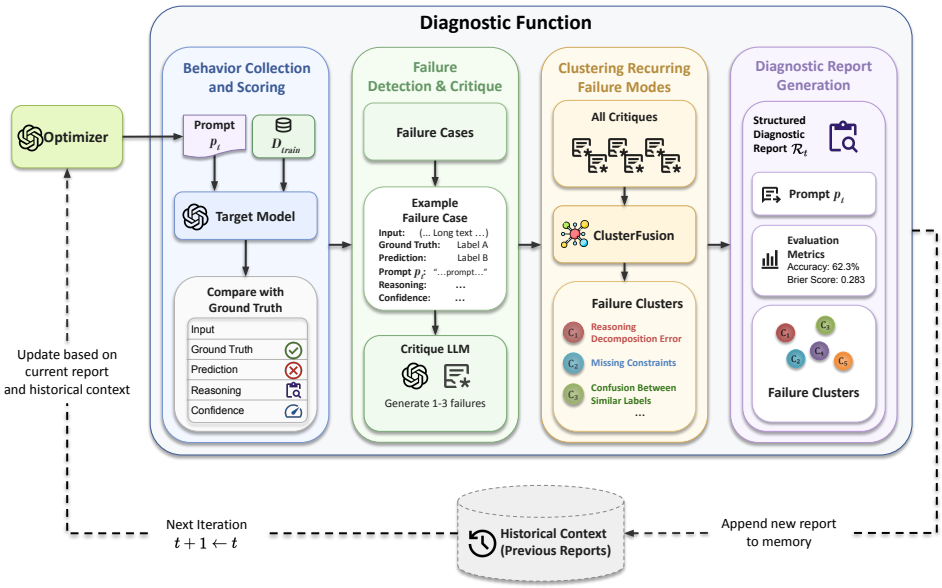


Figure 1: Overview of Reflective Prompt Tuning (RPT). At each iteration, the optimizer calls a diagnostic function that evaluates the current prompt on $\mathbb{D}_{\text{train}}$, critiques failures, clusters recurring failure modes, and returns a structured report. The optimizer uses this report and prior reports to generate the next prompt.

2. **Calibration**, for reliability: the model's stated confidence should reflect the probability that its answer is correct. Verbalized confidence is widely used as a proxy for answer reliability in abstention, routing, and human review (Wen et al., 2025; Chuang et al., 2025; dela Cruz et al., 2025; Wang et al., 2026); improving accuracy without improving calibration can therefore limit its usefulness in these settings.
3. **Prompt length**, for inference efficiency: the prompt should remain concise, since it is prepended to every query and adds to the cost and latency of each inference call.

Because these objectives are estimated from data, prompt optimization uses an optimization set $\mathbb{D}_{\text{train}}$ to generate candidate prompts $\pi_1, \ldots, \pi_T$ over T iterations. A final prompt is then selected from $\{\pi_0, \ldots, \pi_T\}$ using a development set $\mathbb{D}_{\text{dev}}$ and evaluated on a held-out test set $\mathbb{D}_{\text{test}}$.

Evaluating a Prompt. We estimate each objective on a dataset $\mathbb{D} = \{(x_i, m_i)\}_{i=1}^{|\mathbb{D}|}$, writing $o_i = (r_i, \hat{y}_i, c_i) = f(\pi, x_i)$ for the response to example i under prompt π. Accuracy is measured by task performance, $\mu_{\text{task}}(\pi, \mathbb{D}) = \frac{1}{|\mathbb{D}|}\sum_{i=1}^{|\mathbb{D}|}\mu(o_i, m_i)$. Reliability is measured by the **Brier score**, $\text{BS}(\pi, \mathbb{D}) = \frac{1}{|\mathbb{D}|}\sum_{i=1}^{|\mathbb{D}|}(c_i - z_i)^2$, where $z_i = 1$ if the answer receives full credit and 0 otherwise. Because Brier score reflects both accuracy and calibration, we also report expected calibration error (ECE), which more directly measures their agreement (Appendix J). Finally, efficiency is measured by prompt length, $\ell(\pi) = |\pi|_{\text{chars}}/1000$, in thousands of characters. Because inference cost is token-based, we also report the selected prompt length in tokens (Appendix H). Together, these form the metric vector:

$$\boldsymbol{\mu}(\pi, \mathbb{D}) = \big[\, \mu_{\text{task}}(\pi, \mathbb{D}),\ \ \text{BS}(\pi, \mathbb{D}),\ \ \ell(\pi) \,\big]^\top. \tag{2}$$

Selecting among candidates requires combining these metrics into a single criterion Ψ, and the returned prompt is

$$\hat{\pi} \ = \ \underset{\pi \in \{\pi_0, \ldots, \pi_T\}}{\arg\max} \ \ \Psi\big(\boldsymbol{\mu}(\pi, \mathbb{D}_{\text{dev}})\big). \tag{3}$$

Reflective Prompt Optimization. Reflective prompt optimizers use natural-language feedback to revise a prompt based on its observed behavior (Pryzant et al., 2023; Yuksekgonul et al., 2025). We view these methods as comprising two stages: a *feedback* stage that diagnoses why model responses

fail, and an *update* stage that uses this diagnosis to revise the prompt. Methods differ in how they construct feedback and turn it into revisions, and in what they carry across iterations, such as earlier prompt versions (Yuksekgonul et al., 2025) or entries accumulated in the optimized artifact (Zhang et al., 2026). Appendix Q compares these choices across existing reflective methods and RPT.

3 REFLECTIVE PROMPT TUNING

RPT instantiates the two stages of reflective optimization (Section 2) with a *diagnostic function* that critiques individual failures and aggregates them into recurring failure modes, and an LLM *optimizer* that revises the prompt using the current report and a memory of earlier prompts, diagnoses, and metrics. Modeled on the iterative workflow of a human prompt engineer, RPT proceeds in four steps, illustrated in Figure 1 and Algorithm 1. First, the optimizer calls the diagnostic function, which runs the target model on the optimization set, evaluates its responses, and critiques failed examples (Section 3.1). Second, the function groups these critiques into recurring failure modes and returns them with the current prompt and aggregate metrics in a structured report, which is appended to memory (Section 3.2). Third, conditioned on the current report and prior reports, the optimizer either revises the prompt and repeats the process or stops (Section 3.3). Finally, RPT selects the final prompt on the development set using task performance, calibration, and prompt length (Section 3.4). All prompts are given in Appendix A.

3.1 DIAGNOSING FAILURES

When the optimizer calls the diagnostic function $\mathbb{F}$ on the current prompt π_t, $\mathbb{F}$ runs the target model on every $(x_i, m_i) \in \mathbb{D}_{\text{train}}$, recording the responses $o_i = (r_i, \hat{y}_i, c_i) = f(\pi_t, x_i)$, and computes the metrics $\boldsymbol{\mu}(\pi_t, \mathbb{D}_{\text{train}})$ of Eq. 2. These metrics summarize how well π_t performs, but not why it fails. $\mathbb{F}$ therefore treats every example that the task metric does not fully credit, $\mathbb{I}_t = \{\, i : \mu(\hat{y}_i, m_i) < 1 \,\}$, as a failure, and passes each to a critique model LLM_{crit} together with its input, evaluator metadata, and response. The critique returns up to three concise diagnoses $\mathbb{Z}_i$, each a short label with a definition and the evidence in the response that supports it; allowing several diagnoses improves coverage and reduces sensitivity to any single critique. Since the response includes stated confidence, the critique also evaluates it, so overconfidence can be diagnosed directly rather than only reflected in the Brier score. The diagnoses of all failures form the pool $\mathbb{Z}_t = \bigcup_{i \in \mathbb{I}_t} \mathbb{Z}_i$.

3.2 IDENTIFYING RECURRING FAILURE MODES

A single failure can be noisy or idiosyncratic, and the pool is too verbose and redundant to pass to the optimizer directly, particularly once earlier reports are included. $\mathbb{F}$ therefore groups semantically similar diagnoses into K failure topics using CLUSTERFUSION (Xu et al., 2025), a hybrid of embedding clustering and LLM topic extraction (Section 5.3 supports this choice: replacing CLUSTERFUSION with a single LLM categorization call degrades performance),

$$\mathbb{C}_t = \big\{(a_k, d_k, \mathbb{S}_k, n_k)\big\}_{k=1}^{K}, \tag{4}$$

where the label a_k and definition d_k describe the failure topic, $\mathbb{S}_k$ holds the evidence cited by representative diagnoses assigned to it, and n_k counts those diagnoses. Only topics holding more than a fraction $\tau = 0.1$ of the pool are retained, $\mathbb{C}'_t = \{\, k \in \mathbb{C}_t : n_k > \tau\,|\mathbb{Z}_t| \,\}$, which keeps the report focused on prominent patterns and bounds it at $1/\tau$ topics (we study the values of K in Appendix E.1). $\mathbb{F}$ then returns the report

$$\mathbb{R}_t = \big(\, \pi_t,\ \boldsymbol{\mu}(\pi_t, \mathbb{D}_{\text{train}}),\ \boldsymbol{\delta}_t,\ \mathbb{C}'_t \,\big), \tag{5}$$

where the trajectory context $\boldsymbol{\delta}_t$ gives the change in each metric relative to the previous iteration and to the best iteration so far, the index of the best iteration, and the number of consecutive iterations without improvement (schema, including how the best iteration is identified, in Appendix C). Each report thus pairs a prompt with the recurring failures it produced.

3.3 REVISING THE PROMPT

Each report is appended to a memory of earlier reports, which is provided to the optimizer LLM_{opt} together with the current report. The optimizer then decides whether to revise the prompt or stop:

$$a_t = \text{LLM}_{\text{opt}}\big(\pi_t, \mathbb{R}_t, \mathbb{M}_{<t}\big), \qquad a_t \in \big\{\textsc{Revise}(\pi'),\ \textsc{Stop}\big\}, \qquad \mathbb{M}_{<t+1} = \mathbb{M}_{<t} \,\|\, (\mathbb{R}_t), \tag{6}$$

Table 1: Benchmark results for different prompt optimizers with GPT-4.1 as the target model. Columns report exact match for HotPotQA, task score for LiveBench-Math, and accuracy for Formula. Final denotes the prompt selected on the development set, and Aggregate is the mean final score across the three tasks. The best final score for each optimizer and task is shown in **bold**. With GPT-5, RPT is the only method competitive on all three tasks.

Optimizer LLM	Method	HotPotQA		LiveBench-Math		Formula		Aggregate
		Initial	Final	Initial	Final	Initial	Final	
GPT-5	ACE	55.4	66.6	54.1	61.6	71.1	**85.5**	71.2
	GEPA	46.6	64.4	40.6	46.2	72.5	74.0	61.5
	MIPRO	48.8	66.8	42.7	44.0	73.5	78.5	63.1
	RPT (ours)	55.5	**68.4**	58.1	**72.2**	72.3	84.0	**74.9**
GPT-5-mini	ACE	55.4	65.0	54.1	64.9	71.1	**85.0**	**71.6**
	GEPA	46.6	62.4	40.6	46.9	72.5	74.5	61.3
	MIPRO	48.8	65.6	42.7	44.4	73.5	77.0	62.3
	RPT (ours)	55.5	**66.2**	58.1	**65.6**	72.3	74.0	68.8
Gemini-3.1-Pro	ACE	55.4	65.8	54.1	60.8	71.1	**84.5**	**70.4**
	GEPA	46.6	**66.4**	40.6	46.9	72.5	75.5	62.9
	MIPRO	48.8	65.2	42.7	44.2	73.5	75.0	61.5
	RPT (ours)	55.5	65.8	58.1	**70.4**	72.3	74.0	70.1
Gemini-3.1-Flash-Lite	ACE	55.4	**66.6**	54.1	56.6	71.1	**83.0**	**68.7**
	GEPA	46.6	64.6	40.6	42.7	72.5	78.5	61.9
	MIPRO	48.8	63.6	42.7	43.6	73.5	71.5	59.6
	RPT (ours)	55.5	64.6	58.1	**64.9**	72.3	73.5	67.7

where $\|$ denotes concatenation and $\mathbb{M}_{<0} = (\,)$. A revision sets $\pi_{t+1} = \pi'$, and the optimizer calls $\mathbb{F}$ again to diagnose it; the loop ends when the optimizer stops or an iteration budget $T_{\max}$ is reached. The optimizer treats the report as evidence rather than instruction: it inspects the metrics, the retained topics and their representative evidence, and the earlier prompts and failures recorded in memory, and decides which instructions to add, rewrite, consolidate, or remove (Appendix P shows an optimized prompt). The next iteration's report then shows whether the targeted failure topics persist. The optimizer decides when to stop from the trajectory context and memory; without memory, it exhausts its iteration budget on every task (Section 5.3).

3.4 Selecting the Final Prompt

Each candidate is also evaluated on $\mathbb{D}_{\text{dev}}$ with the metrics of Eq. 2 as it is produced. Once the loop ends, RPT returns the candidate among $\pi_0, \ldots, \pi_T$ that maximizes Eq. 3, with

$$\Psi\big(\boldsymbol{\mu}(\pi, \mathbb{D}_{\text{dev}})\big) = \mu_{\text{task}}(\pi, \mathbb{D}_{\text{dev}}) - w_{\text{cal}}\,\text{BS}(\pi, \mathbb{D}_{\text{dev}}) - w_{\text{len}}\,\ell(\pi), \tag{7}$$

where $w_{\text{cal}} = 0.1$ and $w_{\text{len}} = 0.005$ throughout the experiments. Appendix E.5 describes how the weights were set, and Appendix F shows that selection is stable on Formula and HotPotQA and moderately sensitive to the length weight on LiveBench-Math. Development-set scores are used only for selection and are never shown to the optimizer.

4 Experimental Setup

Tasks and datasets. We optimize and evaluate prompts on three reasoning tasks: multi-hop reasoning over textual evidence (HotPotQA; Yang et al. (2018)), mathematical reasoning (LiveBench-Math; White et al. (2025)), and domain-specific numerical reasoning (Formula; Wang et al. (2025)). Each task is split into an optimization set $\mathbb{D}_{\text{train}}$, a development set $\mathbb{D}_{\text{dev}}$ for prompt selection, and a held-out test set $\mathbb{D}_{\text{test}}$, following the split constructions used in prior work (statistics in Appendix D).

Models. We use GPT-4.1 (OpenAI, 2025a) as the target model for RPT and all baselines. As optimizer LLMs, we use frontier models from two families and at different scales, GPT-5 and GPT-5-mini (OpenAI, 2025b) and Gemini-3.1-Pro (GoogleAI, 2026a) and Gemini-3.1-Flash-Lite (GoogleAI,

2026b), and instantiate every method with each in turn. All four support function calling, which RPT's optimizer uses to call the diagnostic function. Within that function, the critique LLM is the same model as the optimizer, while ClusterFusion runs on GPT-4.1 (details in Appendix E).

Baselines. We compare RPT with three state-of-the-art prompt optimizers: the reflective methods ACE (Zhang et al., 2026) and GEPA (Agrawal et al., 2026), and the search-based MIPRO (Opsahl-Ong et al., 2024). Following Zhang et al. (2026), we use each method's released configuration: MIPRO and GEPA with `auto="heavy"`, and ACE in offline mode with its defaults. We do not budget-match them because they distribute computation differently across model roles and between optimization and deployment; Appendix I reports both costs. For calibration, we also evaluate GEPA-C, which receives calibration diagnostics as auxiliary feedback but selects prompts by task performance. All methods use the same target model and data splits (Appendix E.3).

Evaluation. We report task-specific performance metrics: exact match for HotPotQA, accuracy for Formula, and task score for LiveBench-Math (Appendix D). For calibration, we report Brier score using the model's verbalized confidence (Xiong et al., 2024). We also report *optimization cost*, measured in model calls, and deployed *prompt size* in tokens (Appendix I).

5 Results and Analyses

We evaluate RPT on the three objectives of Section 2: task performance and prompt length against state-of-the-art baselines (Section 5.1), and calibration (Section 5.2). We then ablate RPT's components (Section 5.3) and analyze its optimization trajectories (Section 5.4).

5.1 RPT Is Competitive on All Tasks with Compact Prompts

Table 1 reports the target model's performance under the initial prompt and under the prompts optimized by RPT and the baselines described in Section 4, each selected on the development set. RPT, GEPA, and MIPRO are seeded with the same task instruction, and all initial scores are measured on the test split under greedy decoding; differences in initial scores reflect each method's own scaffold, such as the DSPy prompt adapter (Khattab et al., 2024) used by GEPA and MIPRO or ACE's generator template (Appendix E.2). We therefore interpret initial-to-final changes only within a method and compare final performance across methods.

Observation 1: Method strengths are task-specific, and with GPT-5, only RPT is competitive on all three tasks. RPT improves over its initial prompt on every task and optimizer, by up to 12.9 points on HotPotQA, 14.1 on LiveBench-Math, and 11.7 on Formula, and never falls below it, whereas MIPRO regresses on Formula under Gemini-3.1-Flash-Lite. On LiveBench-Math, RPT is best under every optimizer and leads ACE by 8.3–10.6 points under three of the four (roughly twice the standard error); GEPA and MIPRO remain far behind, though part of that gap reflects their scaffold, which starts 15–17 points lower. On Formula, ACE leads under every optimizer, by 9.5–11.0 points except with GPT-5, where RPT trails by only 1.5. HotPotQA does not separate the methods: under each optimizer, all four fall within 1.2–4.0 points of one another (a spread comparable to the standard error). No baseline is competitive on all three tasks, whereas RPT is with GPT-5.

Observation 2: RPT benefits most from optimizer strength. Across optimizers, RPT's aggregate ranges from 67.7 to 74.9, while ACE's stays between 68.7 and 71.6; moving from GPT-5-mini to GPT-5 raises RPT's aggregate by 6.3 points but changes each baseline's by at most 0.8. RPT achieves the best aggregate with GPT-5 and nearly ties ACE with Gemini-3.1-Pro, while ACE remains stronger with the smaller optimizers. The difference is concentrated on Formula, where RPT gains 11.7 points with GPT-5 but only 1.2–1.7 with each of the other optimizers. With GPT-5, RPT is within 1.5 points of ACE on Formula, so clustering is not what limits it there; what differs across optimizers is how well each turns the diagnosed failures into domain-specific instructions. GPT-5-mini and Gemini-3.1-Flash-Lite, for instance, answer Formula's distinct failures with the same generic format revisions (Section 5.4, Appendix K). Because the critique model is the optimizer model, optimizer strength affects diagnosis and revision together, consistent with prior findings that optimizer capability shapes LLM-as-optimizer methods (Yang et al., 2024).

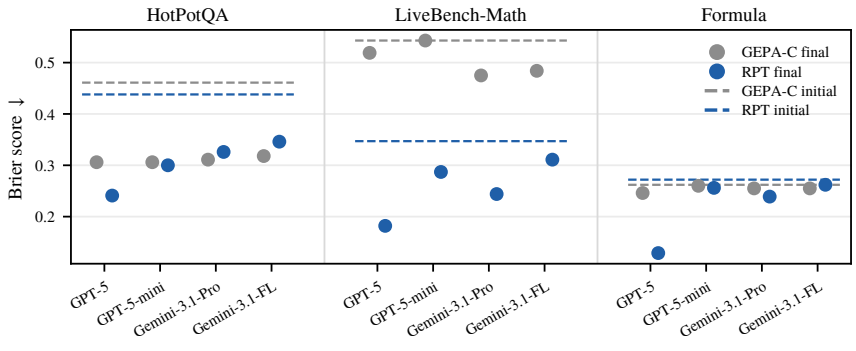


Figure 2: Brier score (lower is better) of RPT and GEPA-C under each optimizer. Markers show selected prompts; dashed lines show each method's initial prompt, shared across optimizers.

Table 2: Component ablations under the GPT-5 optimizer. *w/o memory* conditions the optimizer on the current report only; *w/o clustering* replaces ClusterFusion with a single LLM categorization call; *w/o full-set* replaces clustered full-set diagnosis with an equal volume of raw, unclustered critiques from randomly sampled failures. Task is the task score of Table 1.

	HotPotQA		LiveBench-Math		Formula	
	Task ↑	Brier ↓	Task ↑	Brier ↓	Task ↑	Brier ↓
RPT	**68.4**	**.241**	**72.2**	**.182**	**84.0**	**.129**
w/o memory	67.2	.320	65.2	.219	73.0	.266
w/o clustering	68.2	.260	63.8	.216	83.0	.142
w/o full-set	67.2	.279	63.3	.241	76.5	.178

Observation 3: RPT's selected prompts are compact. Appendix H reports the length of every selected prompt, both in thousands of characters, the unit of the selection objective (Table 7), and in tokens (Table 6). In tokens, RPT's selected prompts never exceed 2.2K, whereas ACE's prompts, dominated by their playbooks, range from 9K to 207K and are at least 23× longer than RPT's in every setting; MIPRO's prompts, which embed few-shot demonstrations, reach 5K–18K on HotPotQA and LiveBench-Math, and GEPA's are of similar size to RPT's. RPT's longest prompts are those selected with GPT-5, most markedly on Formula, where GPT-5's gains most exceed the other optimizers' (Observation 2). Matching ACE's accuracy does not require its length: under GPT-5, the two methods are within the standard error of each other on HotPotQA and Formula, yet ACE's prompts are about 120× and 35× longer, a cost incurred on every query.

5.2 RPT Improves Calibration Alongside Task Performance

We compare RPT with GEPA-C, the only baseline that can use calibration signals without substantial modification. GEPA-C receives them as auxiliary feedback but selects prompts on task performance alone, whereas RPT uses them both in its diagnostic reports and in its selection criterion (Section 3.4). Figure 2 shows the Brier score of both methods from initial to final prompt, and Table 5 (Appendix G) reports the full results, including task scores.

RPT lowers the Brier score from its initial prompt in every setting except Formula under GPT-5-mini, where selection returns the seed prompt, and outperforms GEPA-C with both GPT optimizers on every task and on LiveBench-Math under every optimizer. GEPA-C is better calibrated only with the Gemini optimizers, mainly on HotPotQA. As with task performance, RPT's calibration gains grow with optimizer strength: on HotPotQA its Brier score reaches 0.241 with GPT-5 but stays above 0.32 with the Gemini optimizers, while GEPA-C's remains near 0.31. Since the Brier score also rewards accuracy, part of this advantage reflects RPT's higher task scores; we therefore also measure expected calibration error, which isolates calibration and likewise improves substantially (Appendix J).

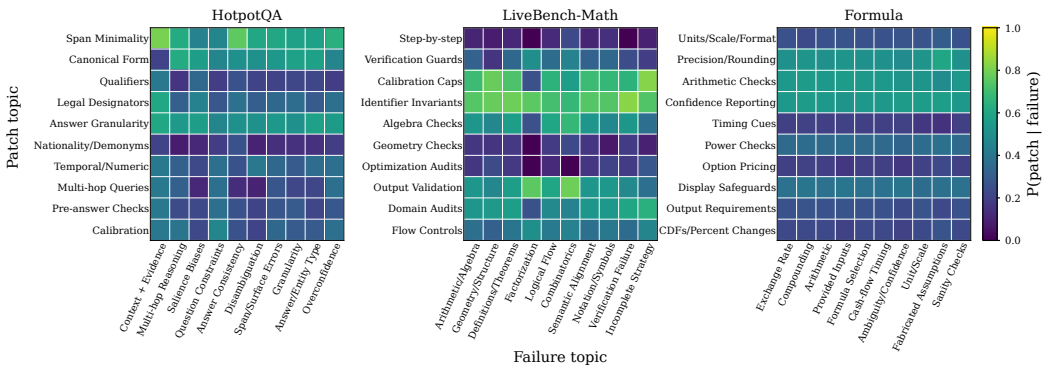


Figure 3: Failure-to-patch alignment across datasets. Each heatmap reports $P(\text{patch topic} \mid \text{failure topic})$, showing which prompt revisions tend to follow each diagnosed failure type.

5.3 Which Components Drive RPT's Gains?

We ablate each of RPT's three design choices, diagnosis over the full optimization set, clustering, and memory, by replacing one at a time, with GPT-5 as the optimizer (Table 2). No variant outperforms RPT on any metric, but each component's importance depends on the task.

Memory. Removing memory hurts most on Formula, where task score falls from 84.0 to 73.0, the Brier score more than doubles, and the model becomes severely overconfident (0.995 average confidence at 73% accuracy). Without memory, the optimizer also exhausts the 40-iteration budget on every task, whereas RPT stops after 15, 19, and 32, so memory helps recognize a plateau.

Clustering. Replacing ClusterFusion with a single categorization call lowers LiveBench-Math task score from 72.2 to 63.8, with smaller effects elsewhere. Passing raw critiques without aggregation was not viable, as they overflowed the optimizer's context window within a few iterations.

Full-set diagnosis. Replacing clustered diagnosis over the full optimization set with an equal volume of raw critiques from sampled failures lowers task score on Formula (84.0 to 76.5) and LiveBench-Math (72.2 to 63.3) and worsens the Brier score on every task, suggesting that aggregating failures across the whole set matters beyond the amount of feedback. Since this variant also removes clustering, its much larger loss than w/o clustering on Formula (76.5 vs. 83.0) suggests that diagnosing all failures matters the most there.

5.4 What Does RPT Learn from Diagnostics?

RPT's traces record each iteration's diagnosed failures and the revision that follows. We use them to study how RPT revises prompts under GPT-5, its best-performing optimizer (Appendix K covers the others). For each *transition*, a pair of consecutive prompts (π_t, π_{t+1}), we extract atomic edits (*patches*) with GPT-4.1, and group diagnoses and patches with ClusterFusion into 10 failure and 10 patch topics per task, chosen by manual inspection. As the traces are observational, we read the results as associations, not causal effects.

Revisions are failure-specific to varying degrees across tasks. For each failure topic F_i and patch topic Q_j, we compute $P(Q_j \mid F_i)$ as the fraction of transitions containing Q_j among those containing F_i, treating topic presence within a transition as binary (Figure 3). On HotPotQA, answer-control patches such as span minimality and answer granularity follow many failure types, reflecting the benchmark's sensitivity to exact answer form, while multi-hop failures more often trigger relation- and query-handling patches (optimized prompt in Appendix P). On LiveBench-Math, alignment concentrates on a few patch families (calibration caps, identifier invariants, and output validation) applied across most failure topics. Formula is broader still: many distinct failures lead to the same domain-level safeguards, such as option-pricing, timing, and percent-change checks, which are domain-specific even when not failure-specific. With GPT-5-mini and Gemini-3.1-Flash-Lite, by contrast, many distinct failures map onto one or two recurring patch topics, on Formula mostly

generic format patches (Appendix K), consistent with their small Formula gains (Section 5.1). These optimizers thus still use the report, but differ in how they turn diagnoses into edits.

Revisions resolve more failures than they introduce. Alignment shows which edits follow which diagnoses, but not whether those failures subsequently decrease. Pooling each run's diagnoses into a fixed topic set, we compare examples exhibiting a failure before a revision with their outcomes afterward. Net resolution, the fraction of failing examples that become correct minus the fraction of previously correct examples that newly exhibit the failure, is small but positive on all three tasks (Appendix L), so revisions resolve more instances of a diagnosed failure than they introduce. The most persistent failure topics remain active throughout a run and are task-specific: surface-form errors on HotPotQA, arithmetic and semantic-alignment errors on LiveBench-Math, and arithmetic, formula-selection, and cash-flow-timing errors on Formula (Appendix N). Appendix M relates failure and patch topics to next-iteration gains.

6 Related Work

Automated prompt optimization. A growing body of work uses optimization procedures, and increasingly LLMs themselves, to propose, revise, or select prompts. AutoPrompt searches for discrete trigger tokens (Shin et al., 2020), while APE and OPRO use LLMs to generate natural-language prompt candidates from task examples or prior candidate-score pairs (Zhou et al., 2023; Yang et al., 2024). Other methods edit prompts using textual gradients or evolutionary search (Pryzant et al., 2023; Guo et al., 2024), and recent systems optimize modular LLM programs by searching over instructions and demonstrations (Khattab et al., 2024; Opsahl-Ong et al., 2024). RPT instead organizes prompt optimization around diagnosis: it aggregates failures across the optimization set into recurring failure modes and uses this diagnostic state to guide direct prompt revision.

Reflective optimization methods. Recent methods use textual feedback to guide prompt optimization. TextGrad backpropagates natural-language feedback through computation graphs (Yuksekgonul et al., 2025), while GEPA reflects on execution and evaluation traces to propose prompt mutations (Agrawal et al., 2026); MIPRO instead uses Bayesian search over program configurations (Opsahl-Ong et al., 2024). RPT builds on reflective feedback but gives it a different role: each iteration aggregates critiques across the optimization set into recurring failure modes, forming a dataset-level diagnostic state that the optimizer uses together with prior prompt—diagnosis—metric reports to revise the prompt. Appendix Q compares how these methods construct feedback, perform updates, and carry information across iterations. RPT additionally incorporates calibration into both diagnosis and prompt selection, whereas GEPA can use it as auxiliary reflective feedback.

Memory and adaptive context. Recent methods improve LLM behavior by accumulating feedback or reusable strategies over time. Reflexion stores verbal reflections from past trials (Shinn et al., 2023), while Agent-Pro evolves agent policies through reflection on interactive experience (Zhang et al., 2024). Dynamic Cheatsheet and ACE build external playbooks of lessons and strategies that are reused at inference time (Suzgun et al., 2026; Zhang et al., 2026). RPT instead uses memory as optimizer-side state: it retains prior prompt–failure reports so the optimizer can track persistent or resolved failures, relate them to earlier revisions, and avoid treating each update independently.

7 Conclusion

We introduced Reflective Prompt Tuning (RPT), a diagnosis-driven prompt optimization framework in which an LLM optimizer revises prompts from failure modes that recur across the full optimization set and from a memory of earlier prompts and their failures. Across three reasoning tasks, RPT with a strong optimizer is the only method competitive on all of them, while improving calibration and keeping prompts compact. Ablations show that full-set diagnosis, clustering, and memory each contribute, and traces show that revisions target the diagnosed failures. RPT's advantage depends on optimizer strength, most on domain-specific computation; its optimization cost exceeds that of per-sample methods; and it exercises its diagnostic function only minimally, leaving optimizers that choose what evidence to gather to future work (Appendix R).

AI USE STATEMENT

In this work, we used generative AI tools (Claude, Anthropic; ChatGPT, OpenAI) for brainstorming, feedback on methodology and analysis, drafting and editing, and assistance with code implementation for experiments, analysis, plotting, and figures. We did not use generative AI tools to formulate mathematical claims or translate text. LLMs are also components of the method and analyses studied in this work, as described throughout the paper.

We reviewed all AI-assisted work: numerical claims were checked against experiment logs, code and outputs were verified by the authors, and AI-assisted text was reviewed and edited for accuracy and clarity. We take responsibility for the final content of this work.

ETHICS STATEMENT

Data and human subjects. Our experiments use three publicly available benchmarks, HotPotQA, LiveBench-Math, and Formula, in accordance with their licenses, and involve no human subjects, crowdsourced annotation, or personal data.

Potential misuse and bias. RPT is a general prompt optimization framework whose impact depends on the objective, data, and target model to which it is applied. The same diagnosis-driven loop could be used to optimize prompts for harmful objectives, such as generating misleading content or evading safety filters, and optimizing against a narrow development set can amplify dataset biases or overfit to benchmark artifacts. A risk specific to RPT is that its reports retain only failure modes covering more than a tenth of the diagnoses, so rare failure types, which may concentrate on underrepresented inputs, receive less attention during optimization; applications where such inputs matter should also audit failures at the instance level.

Reliance on calibrated confidence. RPT improves the calibration of verbalized confidence, which downstream systems may use for abstention, routing, or human review. Calibration measured on a benchmark's test split need not hold under distribution shift, so it should not be read as a guarantee that confidence can be trusted on new inputs.

Privacy. Diagnosis sends every failing response, with its input and evaluator metadata, to the critique model, and the resulting reports, which cite evidence from failing responses, accumulate in the optimizer's memory. When these models are third-party APIs and the optimization data contains sensitive information, both steps could expose it. Our experiments use only public benchmarks; deployments on proprietary or user data should apply data minimization, access control, and redaction, or use locally hosted models.

Reproducibility, access, and cost. All target and optimizer models in our experiments are proprietary, some in preview versions that may change or be withdrawn, and RPT's gains depend on optimizer strength, which may limit access and exact reproducibility. RPT also issues more optimization calls than per-sample methods, although its compact prompts reduce the recurring cost of inference (Appendix I). We will release our code, prompts, and experimental details upon publication; open-model instantiations and lower-cost variants remain future work. Our use of AI assistance in preparing this paper is described in the AI Use Statement.

REPRODUCIBILITY STATEMENT

We describe RPT in Section 3, with pseudocode in Appendix B, the structure of the diagnostic report in Appendix C, and every prompt it uses (seed, critique, and optimizer prompts) in Appendix A. Its hyperparameters, including the number of topics, the retention threshold, the iteration budget, and the stopping procedure, are given in Appendix E.1, and the selection weights, how they were set, and the sensitivity of selection to them in Appendices E.5 and F. Datasets, splits, and scoring are described in Appendix D; baseline configurations, method scaffolds, and the measurement protocol in Appendices E.2 and E.3; metrics in Appendix E.4; and prompt lengths and optimization cost in Appendices H and I. All models are accessed through public APIs. Because these APIs are not fully

deterministic and may change over time, exact scores may not be reproducible; we therefore report initial scores as means over repeated evaluations and give test-set standard errors (Section 5.1). We will release our code and all selected prompts upon publication.

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, Alexandros G. Dimakis, Ion Stoica, Dan Klein, Matei Zaharia, and Omar Khattab. Gepa: Reflective prompt evolution can outperform reinforcement learning. 2026.

Yu-Neng Chuang, Prathusha Kameswara Sarma, Parikshit Gopalan, John Boccio, Sara Bolouki, Xia Hu, and Helen Zhou. Learning to route llms with confidence tokens, 2025. URL https://arxiv.org/abs/2410.13284.

Jane Arleth dela Cruz, Iris Hendrickx, and Martha Larson. Evaluating large language models for confidence-based check set selection. In Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar (eds.), *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 16249–16265, Vienna, Austria, July 2025. Association for Computational Linguistics. ISBN 979-8-89176-256-5. doi: 10.18653/v1/2025.findings-acl.836. URL https://aclanthology.org/2025.findings-acl.836/.

GoogleAI. Best for complex tasks and bringing creative concepts to life. https://deepmind.google/models/gemini/pro/, 2026a.

GoogleAI. Gemini 3.1 flash-lite: Built for intelligence at scale. https://blog.google/innovation-and-ai/models-and-research/gemini-models/gemini-3-1-flash-lite/, 2026b.

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 *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=ZG3RaNIsO8.

Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan, Saiful Haq, Ashutosh Sharma, Thomas T. Joshi, Hanna Moazam, Heather Miller, Matei Zaharia, and Christopher Potts. Dspy: Compiling declarative language model calls into self-improving pipelines. 2024.

Nils Knoth, Antonia Tolzin, Andreas Janson, and J. Leimeister. Ai literacy and its implications for prompt engineering strategies. *Comput. Educ. Artif. Intell.*, 6:100225, 2024. URL https://api.semanticscholar.org/CorpusId:269273689.

Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. Large language models are zero-shot reasoners. In *Proceedings of the 36th International Conference on Neural Information Processing Systems*, NIPS '22, Red Hook, NY, USA, 2022. Curran Associates Inc. ISBN 9781713871088.

Renze Lou, Kai Zhang, and Wenpeng Yin. Large language model instruction following: A survey of progresses and challenges. *Computational Linguistics*, 50(3):1053–1095, 09 2024. ISSN 0891-2017. doi: 10.1162/coli_a_00523. URL https://doi.org/10.1162/coli_a_00523.

OpenAI. Introducing gpt-4.1 in the api. https://openai.com/index/gpt-4-1/, 2025a. Version: 2025-04-14.

OpenAI. Gpt-5 system card. https://cdn.openai.com/gpt-5-system-card.pdf, 2025b. Version: 2025-08-13.

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 Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen (eds.), *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pp. 9340–9366, Miami, Florida, USA, November 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.emnlp-main.525. URL https://aclanthology.org/2024.emnlp-main.525/.

Reid Pryzant, Dan Iter, Jerry Li, Yin Lee, Chenguang Zhu, and Michael Zeng. Automatic prompt optimization with "gradient descent" and beam search. In Houda Bouamor, Juan Pino, and Kalika Bali (eds.), *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 7957–7968, Singapore, December 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.emnlp-main.494. URL https://aclanthology.org/2023.emnlp-main.494/.

Kiran Ramnath, Kang Zhou, Sheng Guan, Soumya Smruti Mishra, Xuan Qi, Zhengyuan Shen, Shuai Wang, Sangmin Woo, Sullam Jeoung, Yawei Wang, Haozhu Wang, Han Ding, Yuzhe Lu, Zhichao Xu, Yun Zhou, Balasubramaniam Srinivasan, Qiaojing Yan, Yueyan Chen, Haibo Ding, Panpan Xu, and Lin Lee Cheong. A systematic survey of automatic prompt optimization techniques. In Christos Christodoulopoulos, Tanmoy Chakraborty, Carolyn Rose, and Violet Peng (eds.), *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 33078–33110, Suzhou, China, November 2025. Association for Computational Linguistics. ISBN 979-8-89176-332-6. doi: 10.18653/v1/2025.emnlp-main.1681. URL https://aclanthology.org/2025.emnlp-main.1681/.

Pranab Sahoo, Ayush Kumar Singh, Sriparna Saha, Vinija Jain, Samrat Mondal, and Aman Chadha. A systematic survey of prompt engineering in large language models: Techniques and applications, 2025. URL https://arxiv.org/abs/2402.07927.

Sander Schulhoff, Michael Ilie, Nishant Balepur, Konstantine Kahadze, Amanda Liu, Chenglei Si, Yinheng Li, Aayush Gupta, HyoJung Han, Sevien Schulhoff, Pranav Sandeep Dulepet, Saurav Vidyadhara, Dayeon Ki, Sweta Agrawal, Chau Pham, Gerson Kroiz, Feileen Li, Hudson Tao, Ashay Srivastava, Hevander Da Costa, Saloni Gupta, Megan L. Rogers, Inna Goncearenco, Giuseppe Sarli, Igor Galynker, Denis Peskoff, Marine Carpuat, Jules White, Shyamal Anadkat, Alexander Hoyle, and Philip Resnik. The prompt report: A systematic survey of prompt engineering techniques, 2025. URL https://arxiv.org/abs/2406.06608.

Melanie Sclar, Yejin Choi, Yulia Tsvetkov, and Alane Suhr. Quantifying language models' sensitivity to spurious features in prompt design or: How i learned to start worrying about prompt formatting. *The Twelfth International Conference on Learning Representations*, abs/2310.11324, 2024. URL https://arxiv.org/pdf/2310.11324.pdf.

Taylor Shin, Yasaman Razeghi, Robert L. Logan IV, Eric Wallace, and Sameer Singh. AutoPrompt: Eliciting Knowledge from Language Models with Automatically Generated Prompts. In Bonnie Webber, Trevor Cohn, Yulan He, and Yang Liu (eds.), *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 4222–4235, Online, November 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.emnlp-main.346. URL https://aclanthology.org/2020.emnlp-main.346/.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik R. Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems*, 2023.

Mirac Suzgun, Mert Yuksekgonul, Federico Bianchi, Dan Jurafsky, and James Zou. Dynamic cheatsheet: Test-time learning with adaptive memory. In Vera Demberg, Kentaro Inui, and Lluís Marquez (eds.), *Proceedings of the 19th Conference of the European Chapter of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 7080–7106, Rabat, Morocco, March 2026. Association for Computational Linguistics. ISBN 979-8-89176-380-7. doi: 10.18653/v1/2026.eacl-long.333. URL https://aclanthology.org/2026.eacl-long.333/.

Dannong Wang, Jaisal Patel, Daochen Zha, Steve Y. Yang, and Xiao-Yang Liu. Finlora: Benchmarking lora methods for fine-tuning llms on financial datasets, 2025. URL https://arxiv.org/abs/2505.19819.

Jiawei Wang, Yanfei Zhou, Siddartha Devic, and Deqing Fu. Are llm decisions faithful to verbal confidence?, 2026. URL https://arxiv.org/abs/2601.07767.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. In *Proceedings of the 36th International Conference on Neural Information Processing Systems*, NIPS '22, Red Hook, NY, USA, 2022. Curran Associates Inc. ISBN 9781713871088.

Bingbing Wen, Jihan Yao, Shangbin Feng, Chenjun Xu, Yulia Tsvetkov, Bill Howe, and Lucy Lu Wang. Know your limits: A survey of abstention in large language models, 2025. URL https://arxiv.org/abs/2407.18418.

Colin White, Samuel Dooley, Manley Roberts, Arka Pal, Benjamin Feuer, Siddhartha Jain, Ravid Shwartz-Ziv, Neel Jain, Khalid Saifullah, Sreemanti Dey, Shubh-Agrawal, Sandeep Singh Sandha, Siddartha Venkat Naidu, Chinmay Hegde, Yann LeCun, Tom Goldstein, Willie Neiswanger, and Micah Goldblum. Livebench: A challenging, contamination-free LLM benchmark. In *The Thirteenth International Conference on Learning Representations*, 2025.

Miao Xiong, Zhiyuan Hu, Xinyang Lu, YIFEI LI, Jie Fu, Junxian He, and Bryan Hooi. Can LLMs express their uncertainty? an empirical evaluation of confidence elicitation in LLMs. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=gjeQKFxFpZ.

Yiming Xu, Yuan Yuan, Vijay Viswanathan, and Graham Neubig. Clusterfusion: Hybrid clustering with embedding guidance and llm adaptation. *arXiv preprint arXiv:2512.04350*, 2025.

Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V Le, Denny Zhou, and Xinyun Chen. Large language models as optimizers. *The Twelfth International Conference on Learning Representations*, 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 Ellen Riloff, David Chiang, Julia Hockenmaier, and Jun'ichi Tsujii (eds.), *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pp. 2369–2380, Brussels, Belgium, October-November 2018. Association for Computational Linguistics. doi: 10.18653/v1/D18-1259. URL https://aclanthology.org/D18-1259/.

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.

J.D. Zamfirescu-Pereira, Richmond Y. Wong, Bjoern Hartmann, and Qian Yang. Why johnny can't prompt: How non-ai experts try (and fail) to design llm prompts. *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*, 2023. URL http://dl.acm.org/citation.cfm?id=3581388.

Qizheng Zhang, Changran Hu, Shubhangi Upasani, Boyuan Ma, Fenglu Hong, Vamsidhar Kamanuru, Jay Rainton, Chen Wu, Mengmeng Ji, Hanchen Li, Urmish Thakker, James Zou, and Kunle Olukotun. Agentic context engineering: Evolving contexts for self-improving language models, 2026. URL https://arxiv.org/abs/2510.04618.

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 Lun-Wei Ku, Andre Martins, and Vivek Srikumar (eds.), *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 5348–5375, Bangkok, Thailand, August 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.acl-long.292. URL https://aclanthology.org/2024.acl-long.292/.

Yongchao Zhou, Andrei Ioan Muresanu, Ziwen Han, Keiran Paster, Silviu Pitis, Harris Chan, and Jimmy Ba. Large language models are human-level prompt engineers, 2023. URL https://arxiv.org/abs/2211.01910.

Jingming Zhuo, Songyang Zhang, Xinyu Fang, Haodong Duan, Dahua Lin, and Kai Chen. Prosa: Assessing and understanding the prompt sensitivity of llms. In *Conference on Empirical Methods in Natural Language Processing*, 2024. URL https://api.semanticscholar.org/CorpusId:273375563.

A RPT PROMPTS

This section lists the seed prompts, the optimizer prompt (shared across all datasets), and the dataset-specific critic prompts used in RPT.

A.1 SEED PROMPTS

Formula Seed Prompt

System message
You are an analysis expert tasked with answering questions using your knowledge.
User instruction

- Show your reasoning step-by-step
- Be concise but thorough in your analysis
- Double-check your calculations and logic before providing the final answer

Your output should be a JSON with fields:

- reasoning: your chain of thought / reasoning / thinking process, detailed analysis and calculations
- answer: your concise final answer.
- confidence: a number in [0,1] representing your confidence in the final answer.

HotpotQA Seed Prompt

System message
You are tasked with answering questions using only the provided context.
User instruction

- Reason step by step using only the provided context.
- Be concise but thorough in your justification.
- Before answering, verify that your answer is supported by the context.

Your output should be a JSON with fields:

- justification: a context-grounded explanation of how you reached the answer.
- answer: your concise final answer.
- confidence: a number in [0,1] representing your confidence in the final answer.

LiveBench-Math Seed Prompt

System message
Solve the math problem step by step and give the final answer in exactly the format requested by the question.
User instruction
Your output should be a JSON with fields:

- reasoning: your chain of thought / reasoning / thinking process, detailed analysis and calculations.
- answer: final answer in exactly the format requested by the question.
- confidence: a number in [0,1] representing your confidence in the final answer.

Output only valid JSON that matches the required schema.

A.2 CRITIQUE PROMPTS

Formula Critic Prompt

You are a strict evaluation critic for formula-construction failures. You are given ONE QA trace with:

- question
- gold answer
- predicted answer
- model confidence
- model reasoning

Your job is to diagnose why the model produced the wrong answer.
Instructions:

1. Produce 1-3 failure_modes with:
   - label: 2-6 words, consistent across similar errors
   - definition: Comprehensive explanation of the failure mode
   - why: brief, self-contained explanation for THIS example, e.g. "The question asked for the city's location relative to Rome, but the model returned the city name instead."
   - basis: cite what in trace/reasoning shows this
2. Focus on actionable failure modes.
3. If you cannot identify a clear failure mode, return an empty list.
4. Output ONLY valid JSON matching the schema.

HotpotQA Critic Prompt

You are a strict evaluation critic for QA failures. You are given ONE QA trace:

- question
- context (titles + snippets)
- gold answer
- predicted answer
- model confidence
- model reasoning

Your goal is to diagnose WHY the target model produced the wrong answer.
Instructions:

1. Produce 1-3 failure_modes with:
   - label: 2-6 words, consistent across similar errors
   - definition: Comprehensive explanation of the failure mode
   - why: brief explanation for THIS example
   - basis: cite what in the trace/reasoning shows this
2. Make labels concrete and clusterable:
   - Prefer labels like "wrong bridge entity" over long sentences.
   - Do not include entity names, dates, or example-specific details in labels.
3. If you cannot identify a clear failure mode, return an empty list.
4. Output ONLY valid JSON matching the schema (no extra text).

LiveBench-Math Critic Prompt

You are a strict evaluation critic for math failures. You are given one failed model attempt with:

- task metadata
- question
- gold answer
- predicted answer
- model confidence
- model reasoning

Your goal is to diagnose why the model failed.
Instructions:

1. Produce 1-3 failure_modes with:
   - label: 2-6 words, consistent across similar errors
   - definition: Comprehensive explanation of the failure mode
   - why: brief, self-contained explanation for THIS example

- basis: cite what in the trace/reasoning shows this

2. Make labels concrete and clusterable.
3. If you cannot identify a clear failure mode, return an empty list.
4. Output only valid JSON matching the schema.

A.3 Shared Optimizer Prompt

Optimizer Prompt

You are the Reflective Prompt Tuning (RPT) controller.
Your goal is to iteratively improve a PromptProgram for the target task.

At each iteration you must:

1. Call evaluate_prompt exactly once on the CURRENT PromptProgram.
2. Read the returned evaluation report with insights.
3. Output either a PATCH or STOP.

Optimization target:

- Primary: improve task performance on the training split.
- Secondary: improve calibration (lower Brier / reduce overconfidence) without hurting task performance.

Decision guidance:

- When current_summary is provided, use it as the primary decision signal, especially current_summary.metrics and any deltas vs previous/best.
- Use history only to detect trajectory, regressions, and previously ineffective edits.

Patch constraints:

- Edits should be minimal, targeted to the failure modes, and designed to address their underlying issues with concrete guidance.
- Avoid vague or generic guidance that only restates the failure; specify concrete checks, comparisons, extractions, or verifications.
- Prefer revising, merging, deleting, or reorganizing existing instructions over adding new broad rules.
- Keep the output contract stable (JSON schema and required fields).
- Avoid redundant or conflicting rules; consolidate instructions when possible.

Stop condition:

- Output STOP if training-set performance has plateaued or further edits are unlikely to help.

Hard rule:

- Do NOT propose a PATCH or STOP decision before calling evaluate_prompt and receiving its result.

B The RPT Algorithm

Algorithm 1 gives the full procedure of Section 3. Each iteration diagnoses the current prompt (lines 3–4; Section 3.1), clusters the diagnoses and assembles the report (lines 5–6; Section 3.2), and lets the optimizer revise the prompt or stop (lines 8–11; Section 3.3); the final prompt is selected once the loop ends (line 14; Section 3.4). Every candidate is also evaluated on $\mathbb{D}_{\text{dev}}$ as it is produced (line 7), but these scores are used only for selection and never enter the report.

Algorithm 1 Reflective Prompt Tuning (RPT)

Require: Target model f; seed prompt π_0; splits $\mathbb{D}_{\text{train}}, \mathbb{D}_{\text{dev}}$; task metric μ; critique model LLM_{crit}; optimizer LLM_{opt}; iteration budget $T_{\max}$; topic count K; retention threshold τ; selection weights $w_{\text{cal}}, w_{\text{len}}$
Ensure: Selected prompt $\hat{\pi}$

$\mathbb{M} \leftarrow (\,)$; $t \leftarrow 0$
loop
  $o_i \leftarrow f(\pi_t, x_i)$ **for all** $(x_i, m_i) \in \mathbb{D}_{\text{train}}$ ▷ optimizer calls $\mathbb{F}$
  $\mathbb{I}_t \leftarrow \{\, i : \mu(o_i, m_i) < 1 \,\}$; $\mathbb{Z}_t \leftarrow \bigcup_{i \in \mathbb{I}_t} \text{LLM}_{\text{crit}}(x_i, m_i, o_i)$ ▷ critique
  $\mathbb{C}_t \leftarrow \textsc{ClusterFusion}(\mathbb{Z}_t, K)$; $\mathbb{C}'_t \leftarrow \{\, k \in \mathbb{C}_t : n_k > \tau |\mathbb{Z}_t| \,\}$ ▷ cluster
  $\mathbb{R}_t \leftarrow (\pi_t, \boldsymbol{\mu}(\pi_t, \mathbb{D}_{\text{train}}), \delta_t, \mathbb{C}'_t)$ ▷ report, Eq. 5
  compute $\boldsymbol{\mu}(\pi_t, \mathbb{D}_{\text{dev}})$ ▷ for selection only
  $a_t \leftarrow \text{LLM}_{\text{opt}}(\pi_t, \mathbb{R}_t, \mathbb{M})$; $\mathbb{M} \leftarrow \mathbb{M} \,\|\, (\mathbb{R}_t)$ ▷ revise or stop
  if $a_t = \textsc{Stop}$ **or** $t + 1 = T_{\max}$ **then break**
  end if
  $\pi_{t+1} \leftarrow \pi'$ **where** $a_t = \textsc{Revise}(\pi')$; $t \leftarrow t + 1$
end loop
$T \leftarrow t$
return $\hat{\pi} = \arg\max_{\pi \in \{\pi_0, \ldots, \pi_T\}} \Psi\big(\boldsymbol{\mu}(\pi, \mathbb{D}_{\text{dev}})\big)$ ▷ Eqs. 3, 7

C Diagnostic Report Schema

The diagnostic function is exposed to the optimizer as a single callable tool whose return value is the structured report $\mathbb{R}_t$ of Eq. 5; each report is also appended to the memory, so earlier reports reach the optimizer in the same format (Section 3.3). The report has three components.

Current prompt. The prompt π_t evaluated in this iteration, returned verbatim so that the optimizer edits the same object it was scored on.

Aggregate metrics and trajectory context. Summary statistics over $\mathbb{D}_{\text{train}}$ under π_t: the task score, the Brier score, and the average stated confidence. These are accompanied by the trajectory context δ_t, computed against earlier reports: the current iteration number, the change in each metric relative to the previous iteration and to the best iteration so far, the index of the best iteration, whether the most recent revision improved task performance, and the number of consecutive iterations without improvement. The best iteration is identified by a weighted score on $\mathbb{D}_{\text{train}}$ (Appendix E.5). The optimizer prompt (Appendix A.3) instructs the optimizer to treat the current metrics as the primary decision signal and the trajectory context as evidence of regression, stagnation, or previously ineffective edits; the optimizer uses both to decide whether to revise the prompt or stop (Section 3.3).

Retained failure topics. Each topic in $\mathbb{C}'_t$ is reported with its label a_k, its definition d_k, the number n_k of diagnoses assigned to it, and the evidence $\mathbb{S}_k$ cited by representative diagnoses, so that the optimizer can inspect concrete instances rather than topic labels alone. Topics are ordered by size, and only those holding more than a fraction $\tau = 0.1$ of the diagnosis pool are included (Section 3.2).

D Optimization Datasets

HotPotQA. HotPotQA (Yang et al., 2018) is a multi-hop question answering benchmark in which the model is given a question and supporting passages, and must combine evidence across passages to produce the answer. We use it as a task for multi-hop reasoning over textual evidence, requiring models to identify relevant information, connect evidence across hops, and extract a concise answer.

LiveBench-Math. LiveBench-Math (White et al., 2025) is the math category of LiveBench, a benchmark designed to reduce contamination through regularly updated questions and automatic scoring against objective ground-truth answers. We use LiveBench-Math to evaluate mathematical reasoning, including problem decomposition, intermediate computation, and final-answer generation. Specifically, we use the retrieved math questions from the `2024-08-31` LiveBench release. Task

score is the mean per-question score under LiveBench's official scorers. Following GEPA (Agrawal et al., 2026), we evenly split the resulting set of 368 questions (shuffled with Python random seed 0) into train, development, and test sets.

Formula. Formula (Wang et al., 2025) is a financial reasoning benchmark built around the eXtensible Business Reporting Language (XBRL). Following ACE (Zhang et al., 2026), we use Formula as a domain-specific numerical reasoning task. It requires models to apply financial concepts and perform computations over structured financial data. Dataset statistics are reported in Table 11.

E Implementation Details

E.1 RPT Configuration

Each optimizer is instructed to call the diagnostic function once at the start of every iteration, then return either a revised prompt or a stop decision. The diagnostic function evaluates the current prompt on $\mathbb{D}_{\text{train}}$, critiques incorrect responses, clusters the diagnoses with ClusterFusion (Xu et al., 2025), and returns the report of Eq. 5 (schema in Appendix C).

Clustering. ClusterFusion combines embedding-guided partitioning with LLM topic extraction and record assignment. We use GPT-4.1 as its LLM across all RPT instantiations. We set the number of topics K by manual inspection of the topics produced from optimization-set diagnosis pools in preliminary runs, choosing values at which topics were internally coherent and distinct from one another: $K = 10$ for HotPotQA and LiveBench-Math, and $K = 20$ for Formula, where with fewer topics, failures involving different financial concepts were merged into shared topics. K is held fixed across optimizers within a benchmark. Only topics holding more than a fraction $\tau = 0.1$ of the diagnosis pool are retained (Section 3.2), which bounds each report at ten topics regardless of K. In practice far fewer are retained: under the GPT-5 optimizer, reports retained on average 4.7, 4.2, and 2.4 topics on HotPotQA, LiveBench-Math, and Formula, since the finer partition on Formula yields smaller topics, fewer of which exceed the threshold. The trace analysis of Section 5.4 uses a separate clustering, with 10 topics per task.

Stopping. The iteration budget is $T_{\max} = 40$, and the loop ends when the optimizer issues a stop decision or the budget is reached. We number iterations from 1, so iteration 1 evaluates the seed prompt. From the trajectory context of each report (Appendix C), the optimizer sees the current iteration number, the change in each metric relative to the previous and best iterations, whether the last revision improved task performance on $\mathbb{D}_{\text{train}}$, and the number of consecutive iterations without improvement. Under the GPT-5 optimizer, runs stopped at iteration 15 (HotPotQA), 19 (LiveBench-Math), and 32 (Formula), with stop reasons citing plateaued performance and already-targeted failure modes. We impose no fixed patience threshold.

E.2 Initial Prompts and Method Scaffolds

RPT, GEPA, and MIPRO start from the same task instruction, but each method wraps it differently, and these differences persist into the initial scores of Table 1.

Seed prompt. The shared instruction specifies the task and the reasoning requirement (Appendix A.1). The methods differ in how the output format is specified: RPT's seed states it as a JSON contract with fields for reasoning, answer, and confidence, whereas GEPA and MIPRO define the same outputs as DSPy fields. MIPRO starts with zero demonstrations. ACE has no seed prompt in this sense: its initial artifact is a generator template with an empty playbook, which additionally mandates bullet identifiers and a task-specific output schema.

Scaffold. The methods also differ in how the seed reaches the target model. RPT sends its prompt unmodified. GEPA and MIPRO run inside DSPy, whose adapter re-renders the instruction and output fields into DSPy's format, inserts an explicit reasoning field, and parses the response through DSPy's output handling. ACE's template is inseparable from the method, since it must reference the playbook that ACE constructs. These scaffolds are integral to the methods and cannot be removed without changing the method being evaluated, so we do not equalize them.

Table 3: Initial artifacts, output formats, and scaffolds. All methods use GPT-4.1 as the target model, identical data splits, and greedy decoding. Optimizer models and budgets are given in Appendix E.

Method	Initial prompt	Output format	Scaffold
RPT	Shared task instruction	JSON contract in the prompt	Sent unmodified
GEPA	Shared task instruction	DSPy output fields	DSPy adapter; adds reasoning field
MIPRO	Shared task instruction, zero demonstrations	DSPy output fields	DSPy adapter; adds reasoning field
ACE	Generator template, empty playbook	Template-mandated schema with bullet identifiers	Template references playbook

Measurement. All scores are computed on $\mathbb{D}_{\text{test}}$ under greedy decoding. Because API decoding is not fully deterministic even at temperature zero, target-model outputs vary across runs; each initial score is therefore the mean over 4 evaluations of the same prompt.

Resulting differences. Initial scores differ across methods: RPT's scaffold starts highest on HotPotQA and LiveBench-Math and DSPy's lowest, while on Formula RPT starts above ACE but below GEPA and MIPRO. We therefore compare methods by final performance and use initial-to-final changes only within a method; ACE, whose initial scores are closest to RPT's, gives the comparison least affected by scaffolds. Table 3 summarizes each method's initial prompt, output format, and scaffold.

E.3 Baseline Configuration

For MIPRO, we use the official DSPy implementation of MIPROv2, jointly optimizing instructions and demonstrations with `auto="heavy"`. For GEPA, we use the released implementation with `auto="heavy"`. For ACE, we use the released repository in offline mode with its default configuration of one epoch and batch size 1. For GEPA, we additionally evaluate a confidence-aware variant (GEPA-C) in which calibration diagnostics are supplied as auxiliary reflective feedback, while prompt selection remains task-performance driven. For baseline implementations and external artifacts, we use publicly available resources and comply with their respective licenses and terms of use.

E.4 Metric Definitions

Let c_i be the model's reported confidence on example i and $y_i \in \{0, 1\}$ its correctness. The Brier score is the mean squared error of the confidence, $\frac{1}{n}\sum_i (c_i - y_i)^2$, and so reflects both calibration and accuracy. We also measure Expected Calibration Error, which partitions examples into B (set to 10) equal-frequency confidence bins and averages the absolute gap between mean confidence and empirical accuracy within each, $\sum_b \frac{|B_b|}{n} \left|\bar{c}_{B_b} - \bar{y}_{B_b}\right|$, isolating calibration from the accuracy level.

E.5 Prompt Selection Objective

The selection function Ψ of Eq. 3 scores each candidate prompt $\pi_t \in \{\pi_0, \ldots, \pi_T\}$ on $\mathbb{D}_{\text{dev}}$ as a weighted combination of task performance, calibration, and prompt length, to account for performance, reliability, and cost (Eq. 7),

$$\Psi(\pi) \;=\; \mu_{\text{task}}(\pi, \mathbb{D}_{\text{dev}}) \;-\; w_{\text{cal}}\, \text{BS}(\pi, \mathbb{D}_{\text{dev}}) \;-\; w_{\text{len}}\, \ell(\pi), \qquad \ell(\pi) = |\pi|_{\text{chars}}/1000, \tag{8}$$

with $w_{\text{cal}} = 0.1$ and $w_{\text{len}} = 0.005$ for every benchmark, optimizer, and ablation. Only the weights' ratios affect the $\arg\max$, so the task weight is fixed to 1. The prompt with the highest Ψ on $\mathbb{D}_{\text{dev}}$ is evaluated on $\mathbb{D}_{\text{test}}$.

We set the weights manually in preliminary runs, using only optimization- and development-set results. The weights specify the trade-off among the three objectives rather than being fit to data, and were chosen so that task performance dominates. Because only their ratios affect the $\arg\max$,

Table 4: Distinct prompts selected across the 20 weight settings per benchmark (GPT-5 optimizer), with the number of settings selecting each, its length, and its test performance. *Rep.*: selected under the reported weights; *new*: evaluated on $\mathbb{D}_{\text{test}}$ after the reported weights were fixed.

Benchmark	Iteration	Settings	Chars	Test task	Test Brier	
Formula	11	10	8,312	84.0	.129	Rep.
	8	5	7,241	83.5	.133	new
	25	5	15,462	83.0	.132	
HotPotQA	5	17	4,035	68.4	.241	Rep.
	2	3	1,752	68.2	.287	new
LiveBench-Math	7	8	9,844	72.2	.182	Rep., new
	10	2	11,748	71.0	.182	
	15	9	17,715	70.5	.174	
	1	1	408	58.9	.342	new

they can be read as exchange rates: lowering the Brier score by 0.1 is worth one point of task score, and each $2{,}000$ characters of prompt length costs one point. The secondary terms therefore matter only among candidates close in task score. We include calibration because verbalized confidence is used downstream for abstention and routing (Section 5.2), and prompt length because the prompt is prepended to every test query. Appendix F reports how the selected prompts and their test performance change under alternative weights.

Effect of the length term. The length term reflects the observation in Appendix O that prompts grow steadily during optimization while development performance plateaus, so later, longer prompts are frequently no better than earlier, shorter ones. Under the GPT-5 optimizer, selection returns iterations 5, 7, and 11 on HotPotQA, LiveBench-Math, and Formula, prompts 2.0–2.8$\times$ shorter in tokens than the terminal prompt of the same run (Figure 18). Two cases show the trade-off directly. On Formula, the iteration-25 prompt achieved slightly higher development accuracy than the iteration-11 prompt (0.870 versus 0.867) but was not selected because it was substantially longer. On LiveBench-Math, iteration 15 leads iteration 7 on the task and calibration terms by the equivalent of about two task points, but it is 7,871 characters longer (17,715 versus 9,844), which the length term prices at about four task points; iteration 7 is therefore selected, scoring 72.2 on $\mathbb{D}_{\text{test}}$ against 70.5 for the longer iteration-15 prompt.

F Sensitivity to Prompt Selection Weights

The selection weights of Appendix E.5 were manually set, so in this section we examine how much the reported results depend on them. All analyses in this section reuse the GPT-5 optimizer trajectories.

Setting. Using the GPT-5 optimizer, we re-select a prompt from each existing trajectory under alternative weights and evaluate it on $\mathbb{D}_{\text{test}}$; no optimization is re-run, so the analysis measures the sensitivity of the selection step with the trajectories held fixed. With the task weight fixed to 1, as in Eq. 7, we vary the calibration weight over $\{0, 0.05, 0.1, 0.2, 0.5\}$ and the length weight (per 1,000 characters) over $\{0, 0.0025, 0.005, 0.01\}$, giving 20 settings per benchmark (60 total).

Results. As many settings select the same prompt, the 60 settings yield only nine distinct prompts (Table 4). Test task score varies by 1.0 point across the grid on Formula (83.0–84.0) and by 0.2 points on HotPotQA (68.2–68.4) (Figure 4). LiveBench-Math is more sensitive to the length weight: 19 of its 20 settings stay within 70.5–72.2, while the strongest length penalty without a calibration term selects the seed prompt (iteration 1; 58.9). **These results indicate that test performance is largely insensitive to the selection weights: in 59 of the 60 settings, task score differs from that obtained under the reported weights by at most 1.7 points.**

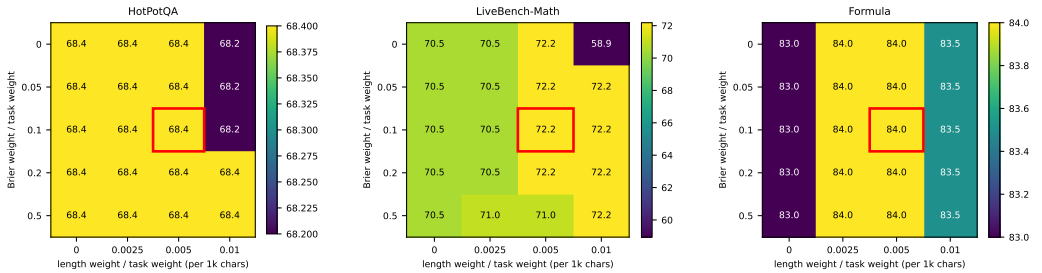


Figure 4: Test task score of the prompt selected on $\mathbb{D}_{\text{dev}}$ under each weight setting (GPT-5 optimizer). Rows vary the calibration weight and columns the length weight per 1,000 characters, both relative to the task weight; the reported weights are outlined in red.

G Full Calibration Results

Table 5 reports the task score and Brier score of RPT and GEPA-C for every optimizer and task, from the initial to the final prompt, complementing Figure 2, which shows the Brier scores. Expected calibration error for RPT's initial and selected prompts is reported in Appendix J.

Table 5: Confidence-aware optimization results. GEPA-C is GEPA with calibration diagnostics supplied as auxiliary feedback (Section 4). Each cell reports the value for the initial→final prompt. Task denotes exact match for HotPotQA, accuracy for Formula, and the official task score for LiveBench-Math; higher task score and lower Brier are better. For each optimizer and task, the better final value of the two methods is shown in **bold**.

Optimizer	Method	HotPotQA		LiveBench-Math		Formula	
		Task ↑	Brier ↓	Task ↑	Brier ↓	Task ↑	Brier ↓
GPT-5	GEPA-C	51.8→65.6	.461→.306	40.1→42.2	.543→.519	73.5→75.5	.262→.246
	RPT	55.5→**68.4**	.438→**.241**	58.1→**72.2**	.347→**.182**	72.3→**84.0**	.272→**.129**
GPT-5-mini	GEPA-C	51.8→65.0	.461→.306	40.1→40.1	.543→.543	73.5→72.5	.262→.260
	RPT	55.5→**66.2**	.438→**.300**	58.1→**65.6**	.347→**.287**	72.3→**74.0**	.272→**.256**
Gemini-3.1-Pro	GEPA-C	51.8→65.0	.461→**.311**	40.1→46.9	.543→.475	73.5→**74.5**	.262→.255
	RPT	55.5→**65.8**	.438→.326	58.1→**70.4**	.347→**.244**	72.3→74.0	.272→**.239**
Gemini-3.1-FL	GEPA-C	51.8→64.2	.461→**.318**	40.1→45.6	.543→.484	73.5→**74.5**	.262→**.255**
	RPT	55.5→**64.6**	.438→.346	58.1→**64.9**	.347→**.311**	72.3→73.5	.272→.262

H Selected Prompt Length

Tables 7 and 6 report the length of the selected prompts behind Table 1, in thousands of characters, the unit of the selection objective (Section 3.4), and in tokens (`cl100k_base`); Figure 5 plots their mean over the three tasks against aggregate score. Lengths measure the fixed text prepended to every query, excluding per-example inputs: for RPT, the full prompt, including its JSON output contract; for GEPA and MIPRO, the optimized instruction and, for MIPRO, its few-shot demonstrations, excluding the output-field formatting that DSPy's adapter adds; and for ACE, its generator template and playbook. Since RPT's counts include its output format while GEPA's and MIPRO's exclude theirs, the comparison with those two methods is conservative for RPT. The Initial rows give each method's seed prompt, which varies by at most 15 tokens across optimizers and is averaged over them.

ACE's prompts, dominated by their playbooks, are the longest in every setting, ranging from 9,060 to 206,808 tokens, at least 23 and up to about 2,000 times the length of RPT's selected prompt, the latter against RPT's unoptimized seed under GPT-5-mini on Formula. Their size also depends strongly on the optimizer: averaged over tasks, they are largest with GPT-5-mini and smallest with

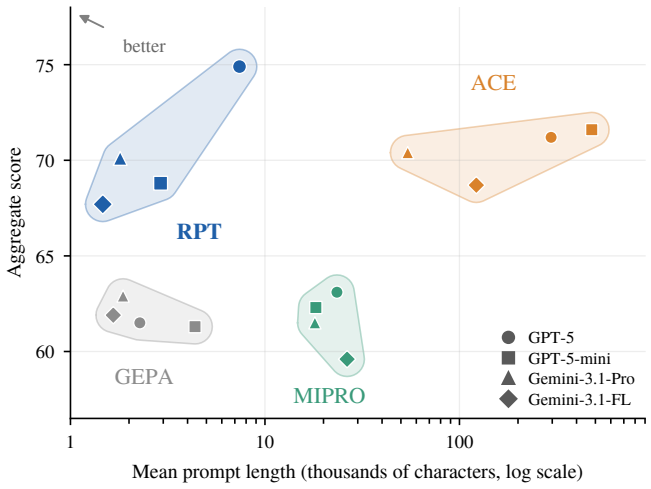


Figure 5: Aggregate score against mean selected-prompt length (log scale) for each method and optimizer LLM; upper left is better. Lengths average the selected prompts of the three tasks (Table 7), in thousands of characters.

Gemini-3.1-Pro, nearly an order of magnitude apart, growing from a template of about 1.6 thousand characters. MIPRO's prompts run to 5,046–17,806 tokens on HotPotQA and LiveBench-Math, driven by their few-shot demonstrations, and are shorter on Formula. GEPA's and RPT's selected prompts stay in the hundreds to low thousands of tokens; RPT's are shorter than GEPA's in most settings outside GPT-5 and longer in all three with GPT-5, whose selected prompts are RPT's longest (7.4 thousand characters on average, against 1.5–2.9 with the other optimizers). Under GPT-5-mini on Formula, RPT's selected prompt is the seed itself, so the difference from its initial score in Table 1 reflects variation between evaluations of the same prompt rather than optimization. Under GPT-5, selected prompts are also shorter than the terminal prompts of their runs, which continue to grow while development performance plateaus (Appendix O).

I Cost Analysis

Table 8 reports per-role call counts under the GPT-5 optimizer for RPT and ACE, the strongest baseline in our comparison; GEPA and MIPRO run inside DSPy's optimization loop, which does not expose per-role call counts without modification. Prompt sizes for all methods and optimizers are reported in Appendix H.

Optimization cost. RPT issues 5.6–8.6× ACE's total call volume. The overhead is concentrated in target-model evaluation and clustering, both of which run on GPT-4.1; over optimizer-model calls alone, the gap narrows to 2.5–2.7×. RPT also issues an order of magnitude fewer update calls than ACE (30–64 versus 654–1,190), since diagnosis is aggregated over the optimization set rather than performed per sample, and only failing responses are critiqued (on average 106, 38, and 97 per iteration, declining as the prompt improves), whereas ACE reflects on every sample. Diagnosis, meaning the evaluation of the optimization set together with its critiques and clustering, accounts for about two-thirds of RPT's calls, and development-set evaluation for nearly all of the rest, since every candidate prompt is scored on the full $\mathbb{D}_{\text{dev}}$. Reducing either, for instance by evaluating on subsets during exploration, remains future work (Appendix R).

Early stopping. Optimization terminates on an optimizer-issued stop rather than at the iteration budget: the three runs halted at 15, 19, and 32 of 40 iterations, avoiding 25, 21, and 8 further full-set evaluations. Development-set selection additionally returns a prompt from earlier in each run (iterations 5, 7, and 11), so later iterations often confirm a plateau rather than improve on it. A smaller budget would therefore suffice on some tasks, though the best iteration is identifiable only after the fact.

Table 6: Length of the selected prompts in tokens (`cl100k_base`). Initial rows give each method's seed prompt, averaged over optimizers; the counting boundary for each method is described in the text.

Optimizer LLM	Method	HotPotQA	LiveBench-Math	Formula
Initial	ACE	349	364	346
	GEPA	81	22	82
	MIPRO	81	22	82
	RPT	101	82	101
GPT-5	ACE	106,320	53,819	70,233
	GEPA	646	390	557
	MIPRO	7,605	9,855	1,830
	RPT	861	2,159	1,992
GPT-5-mini	ACE	68,816	58,503	206,808
	GEPA	542	1,719	686
	MIPRO	7,921	5,046	1,530
	RPT	330	1,373	101
Gemini-3.1-Pro	ACE	16,629	10,550	9,060
	GEPA	509	255	417
	MIPRO	7,162	6,408	1,188
	RPT	371	449	283
Gemini-3.1-Flash-Lite	ACE	19,184	10,759	49,195
	GEPA	350	229	473
	MIPRO	7,408	17,806	905
	RPT	190	309	428

Table 7: Length of the selected prompts in thousands of characters, the unit of the selection objective. Initial rows give each method's seed prompt, averaged over optimizers; Mean is the average over the three tasks, as plotted in Figure 5.

Optimizer LLM	Method	HotPotQA	LiveBench-Math	Formula	Mean
Initial	ACE	1.6	1.7	1.6	1.6
	GEPA	0.4	0.1	0.4	0.3
	MIPRO	0.4	0.1	0.4	0.3
	RPT	0.5	0.4	0.5	0.5
GPT-5	ACE	440.7	182.6	264.6	296.0
	GEPA	2.9	1.6	2.2	2.3
	MIPRO	32.7	30.6	7.0	23.4
	RPT	4.0	9.8	8.3	7.4
GPT-5-mini	ACE	310.0	232.4	894.3	478.9
	GEPA	2.5	7.5	3.1	4.4
	MIPRO	33.5	15.4	5.8	18.3
	RPT	1.5	6.7	0.5	2.9
Gemini-3.1-Pro	ACE	78.8	44.9	39.0	54.2
	GEPA	2.4	1.2	2.0	1.9
	MIPRO	31.7	18.0	4.5	18.1
	RPT	1.7	2.4	1.3	1.8
Gemini-3.1-Flash-Lite	ACE	94.1	48.0	224.2	122.1
	GEPA	1.7	1.1	2.1	1.7
	MIPRO	31.3	44.3	3.7	26.4
	RPT	1.0	1.6	1.9	1.5

Deployment cost. Because the prompt is prepended to every test query, its size is the only cost term that recurs indefinitely. Under GPT-5, RPT's selected prompts contain 861–2,159 tokens, against 53,819–106,320 for ACE, whose playbook accumulates entries from every sample and so grows with $|\mathbb{D}_{\text{train}}|$ (Appendix H). On HotPotQA, ACE's prompt over a single pass of the test set consumes more

Table 8: Per-role call counts under the GPT-5 optimizer. ACE's optimizer row covers its Reflector and Curator calls. RPT completed 15, 19, and 32 iterations; ACE completed one epoch of 300, 123, and 500 steps.

Dataset	Role (model)	ACE	RPT
HotPotQA	Target, train (GPT-4.1)	770	4,500
	Critique (GPT-5)	—	1,596
	Clustering (GPT-4.1)	—	2,413
	Optimizer (GPT-5)	654	30
	Target, dev (GPT-4.1)	900	4,500
LiveBench-Math	Target, train (GPT-4.1)	320	2,337
	Critique (GPT-5)	—	718
	Clustering (GPT-4.1)	—	1,823
	Optimizer (GPT-5)	276	38
	Target, dev (GPT-4.1)	246	2,337
Formula	Target, train (GPT-4.1)	1,286	16,000
	Critique (GPT-5)	—	3,098
	Clustering (GPT-4.1)	—	5,130
	Optimizer (GPT-5)	1,190	64
	Target, dev (GPT-4.1)	1,500	9,600

input tokens than the entire optimization-phase difference between the two methods. Prefix caching substantially reduces the marginal cost of a static playbook, so these profiles are best read as different placements of cost: **RPT concentrates cost once, in optimization, and returns a compact prompt that remains inexpensive to deploy.**

J Expected Calibration Error

Brier score is the mean squared error of the reported confidence and therefore reflects both calibration and accuracy. To measure calibration independently of the accuracy level, we additionally report Expected Calibration Error (Appendix E.4) for RPT's initial and selected prompts under the GPT-5 optimizer.

Table 9: Task score, Brier score, and ECE for RPT's initial and selected prompts. ECE is computed from the same per-example outputs as Table 5.

Dataset	Prompt	Task Score	Brier	ECE
HotPotQA	Initial	55.5	0.438	0.445
	Final	68.4	0.241	0.163
LiveBench-Math	Initial	58.1	0.347	0.440
	Final	72.2	0.182	0.207
Formula	Initial	72.3	0.272	0.281
	Final	84.0	0.129	0.021

ECE falls from 0.445 to 0.163 on HotPotQA, 0.440 to 0.207 on LiveBench-Math, and 0.281 to 0.021 on Formula. Since ECE compares confidence to accuracy within bins rather than rewarding accuracy directly, these reductions indicate that RPT's confidence values become better calibrated, not only more often correct.

K Failure-to-Patch Alignment Across Optimizers

Section 5.4 analyzes failure-to-patch alignment under the GPT-5 optimizer. Here we apply the same procedure to GPT-5-mini (Figure 6), Gemini-3.1-Pro (Figure 7), and Gemini-3.1-Flash-Lite

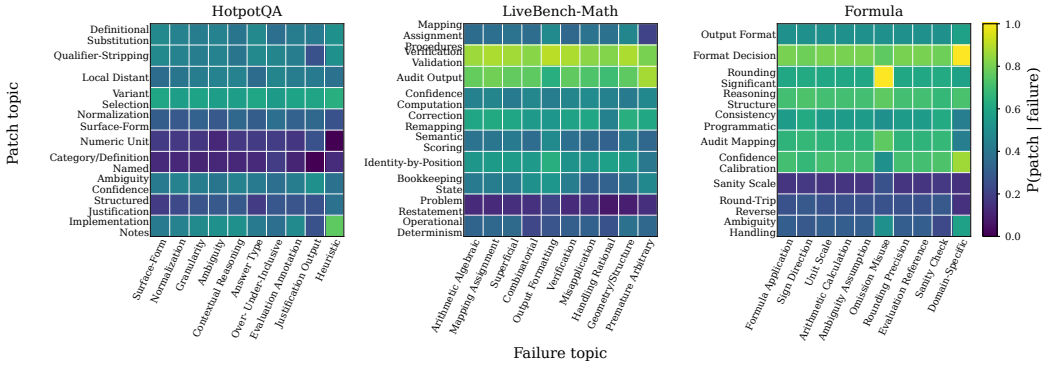


Figure 6: Failure-to-patch alignment under the GPT-5-mini optimizer. Each heatmap reports $P(\text{patch topic} \mid \text{failure topic})$.

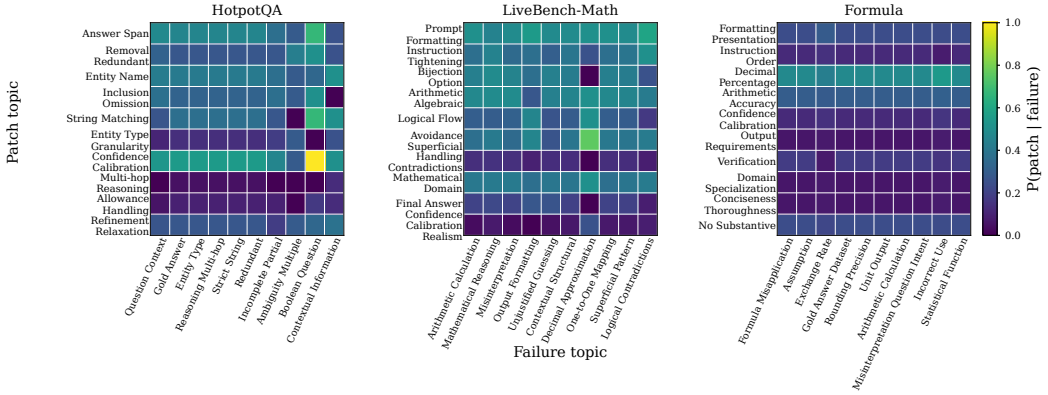


Figure 7: Failure-to-patch alignment under the Gemini-3.1-Pro optimizer.

(Figure 8), using identical topic counts and transition definitions. Topics are derived independently per run, so topic identities are not comparable across optimizers; only the shape of each alignment map is.

Weaker optimizers produce visibly flatter maps, in which a single patch topic is applied at high probability regardless of the diagnosed failure. Under Gemini-3.1-Flash-Lite, *Format Identification* dominates Formula and *Mapping Placeholders* dominates LiveBench-Math, each recurring across nearly all failure topics. GPT-5-mini shows the same pattern on LiveBench-Math, where *Verification Validation* and *Audit Output* are applied almost uniformly, and on Formula, where *Format Decision* spans every failure type. Gemini-3.1-Pro, the stronger of the two Gemini models, is more differentiated: on HotPotQA its confidence-calibration patches concentrate on boolean-question failures rather than spreading evenly, and several patch topics remain near zero for most failures.

These maps are not flat everywhere, so weaker optimizers are not ignoring the diagnostic report; rather, they compress many distinct diagnoses into a small number of recurring edit types. Optimizer strength therefore affects how diagnoses are differentiated into edits rather than whether the report is used at all.

L Failure Resolution Across the Optimization Trajectory

Alignment indicates which patches follow which diagnoses, but not whether the diagnosed failures subsequently decrease. Measuring this directly is complicated by the fact that failure clusters are

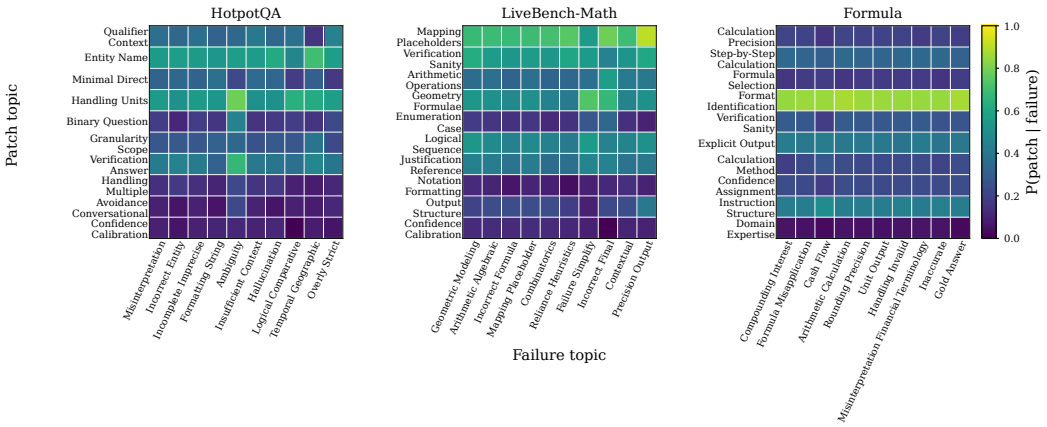


Figure 8: Failure-to-patch alignment under the Gemini-3.1-Flash-Lite optimizer.

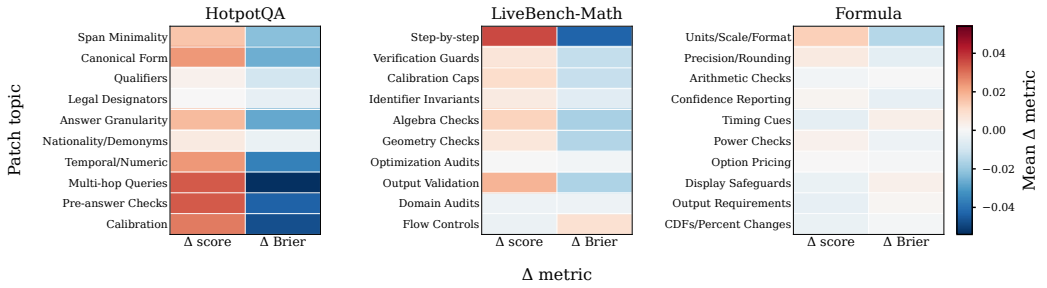


Figure 9: Patch topics and next-iteration metric changes. Each cell reports the average Δ task score or Δ Brier on $\mathbb{D}_{\text{train}}$ after a prompt update containing that patch; higher Δ task score and lower Δ Brier indicate improvement.

recomputed independently at each iteration, so cluster identities are not stable over time. We therefore pool all diagnoses from a run into a fixed run-level topic set and evaluate each transition against that fixed set.

For each topic, we identify the examples exhibiting that failure before a revision and compare their outcomes afterward. *Resolved rate* is the fraction of those examples that become correct; *emerged rate* is the fraction of previously correct examples that newly exhibit the failure, normalized by the same failing-population size. We micro-average counts across topics and transitions and define net resolution as the difference.

Table 10: Per-transition failure resolution under the GPT-5 optimizer.

Dataset	Resolved	Emerged	Net
HotPotQA	0.0672	0.0536	+0.0137
LiveBench-Math	0.1378	0.1224	+0.0153
Formula	0.1496	0.1396	+0.0100

Net resolution is positive on all three datasets. Per-transition gains are modest, but they accumulate over the 15–32 iterations of each run and are consistent with the task-score improvements in Table 1. An undirected revision process would be expected to resolve and introduce failures at roughly symmetric rates, so this asymmetry indicates that revisions make cumulative progress against the

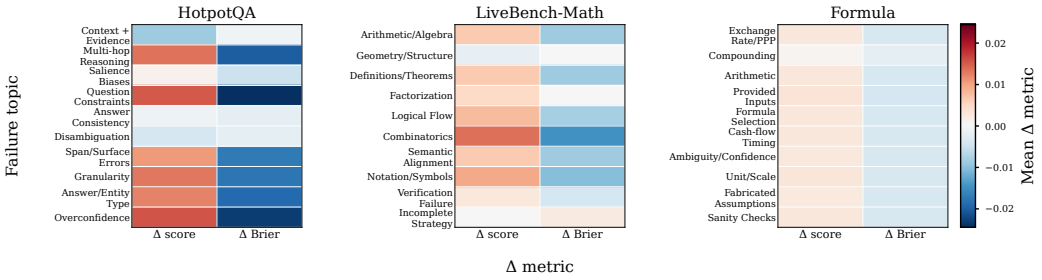


Figure 10: Average next-iteration metric changes associated with each diagnosed failure topic for HotPotQA, LiveBench-Math, and Formula. For each failure topic present at iteration t, we report the mean change in task score and Brier score from t to $t+1$. Higher Δ task score and lower Δ Brier indicate improvement.

Table 11: Datasets used for prompt optimization, prompt selection, and final evaluation.

Dataset	Train	Dev	Test
HotPotQA (Yang et al., 2018)	300	300	500
LiveBench-Math (White et al., 2025)	123	123	122
Formula (Wang et al., 2025)	500	300	200

failures they diagnose. It does not establish causal attribution between any specific diagnosis and any specific edit.

M Next-Iteration Associations

We report which patch topics and which diagnosed failure topics precede improvements in the next iteration. For each topic present at iteration t, we compute the average change in task score and Brier score on $\mathbb{D}_{\text{train}}$ after evaluating p_{t+1}; positive Δ task score and negative Δ Brier indicate improvement.

Both analyses share a limitation: a transition typically contains several patches, and the metric change is attributed to every topic present, so effects are not separable across co-occurring topics. They are additionally confounded with iteration order, since early revisions produce larger metric changes than later ones. We therefore report these results as descriptive associations and base the claims in Section 5.4 on the failure-resolution analysis instead.

Under GPT-5 (Figures 9 and 10), useful patches differ by task but share a common structure: they impose concrete controls on the model's reasoning or output. On HotPotQA, the strongest gains are associated with relation and multi-hop handling, pre-answer verification, confidence calibration, and answer granularity matching. On LiveBench-Math, gains are associated with step-by-step solution protocols, output validation, arithmetic checks, and confidence calibration. On Formula, the clearest improvements come from unit, scale, and format handling, while some specialized domain safeguards show weak or negative short-term associations, likely because they are introduced for harder or more persistent failures. The corresponding failure-topic analysis shows that the most actionable diagnoses are concrete failures that can be translated into explicit behavioral constraints.

Across the remaining optimizers (Figures 11–16), confidence calibration is the most consistently beneficial patch topic: it produces the largest Δ task score and largest Brier reduction on HotPotQA under GPT-5-mini and on LiveBench-Math under Gemini-3.1-Pro. Effect magnitudes are also larger and less uniform than under GPT-5, reaching ± 0.08 for individual topics under Gemini-3.1-Flash-Lite compared with ± 0.04 under GPT-5. Since each topic average is computed over the transitions in which that topic appears, these larger magnitudes partly reflect fewer contributing transitions rather than stronger effects.

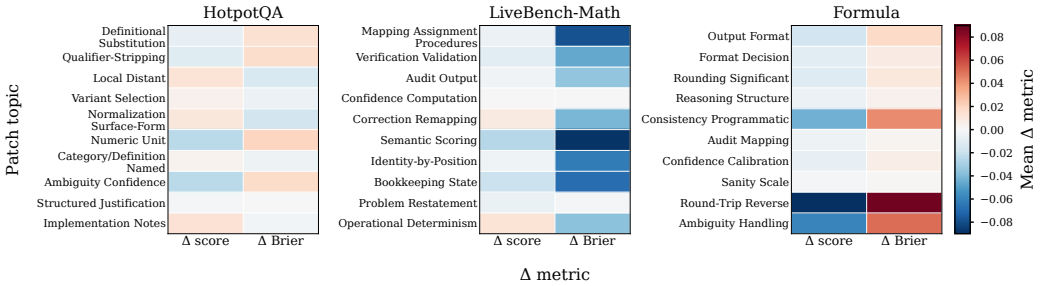


Figure 11: Patch topics and next-iteration metric changes under GPT-5-mini.

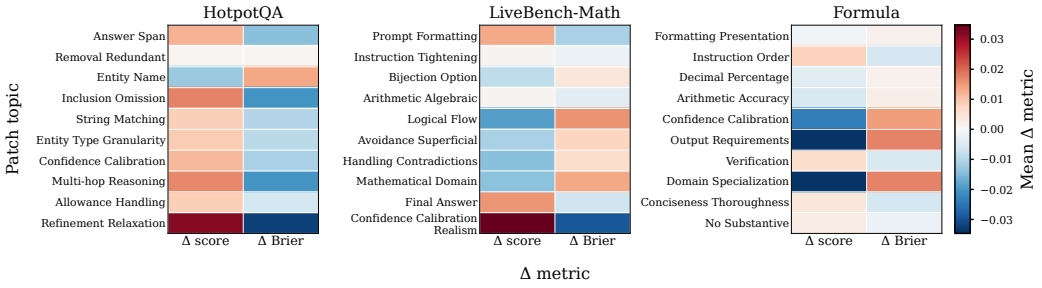


Figure 12: Patch topics and next-iteration metric changes under Gemini-3.1-Pro.

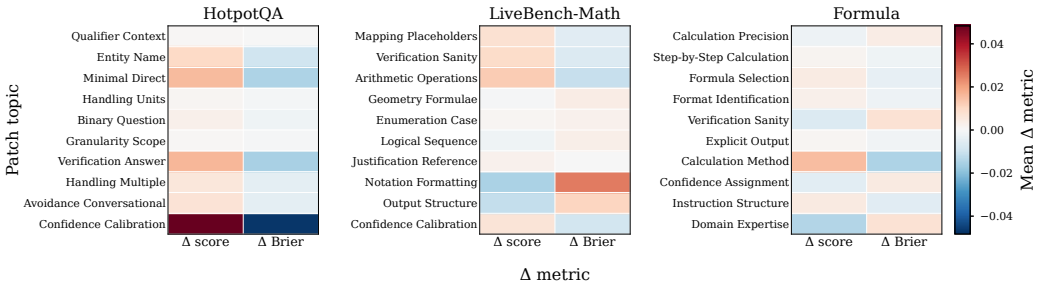


Figure 13: Patch topics and next-iteration metric changes under Gemini-3.1-Flash-Lite.

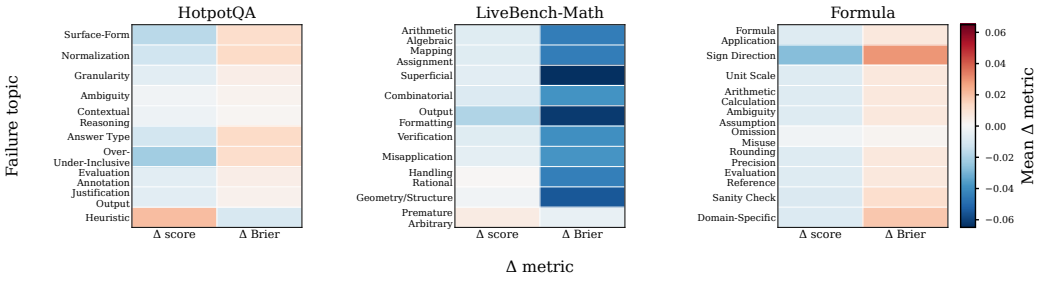


Figure 14: Failure topics and next-iteration metric changes under GPT-5-mini.

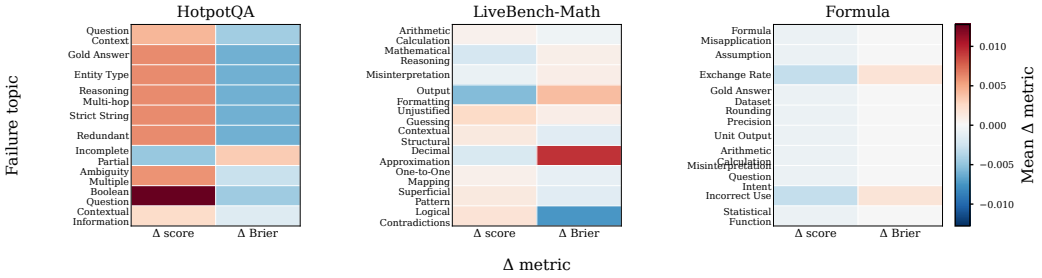


Figure 15: Failure topics and next-iteration metric changes under Gemini-3.1-Pro.

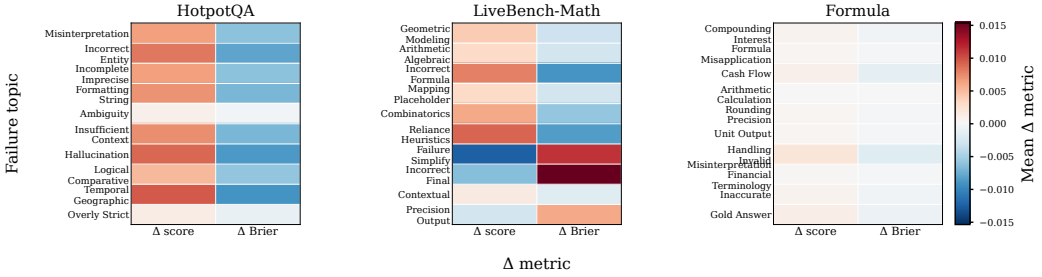


Figure 16: Failure topics and next-iteration metric changes under Gemini-3.1-Flash-Lite.

N Failure-Mode Persistence

We measure the persistence of each failure topic by its average run length, defined as the number of consecutive iterations in which the topic remains active. Longer runs indicate failures that persist despite prompt revisions, while shorter runs suggest failures that are more transient or easier to address.

Figure 17 shows that persistent failures are task-specific, and that the most persistent topics remain active for an entire run. On HotPotQA, the longest-lived issues concern answer form, namely span-extraction and surface-form errors, followed by granularity or answer-type mismatches and multi-hop reasoning; under exact-match scoring, these failures persist even as the prompt accumulates instructions that target them (Appendix P). On LiveBench-Math, persistent failures center on arithmetic and algebraic computation, semantic misalignment, and misuse of mathematical definitions or conventions, and on Formula, on arithmetic calculation, metric definition and formula selection, timing conventions, and domain constraints; on these two tasks, the most persistent failures are reasoning and computation errors rather than formatting issues. Failures that remain active throughout a run may require repeated refinement, or may reflect limitations of the target model that prompting alone cannot overcome.

O Prompt Length and Development Performance

We analyze how prompt length changes during RPT optimization and how these changes relate to development-set performance. Figure 18 plots the number of prompt tokens and the corresponding development task score across optimization iterations for HotPotQA, LiveBench-Math, and Formula.

Across all tasks, prompt length tends to increase as the optimizer incorporates additional constraints, checks, and task-specific instructions. However, development performance does not increase monotonically with prompt length. On HotPotQA, the largest gain occurs early, after which performance remains relatively stable while prompt length continues to grow. On LiveBench-Math, dev performance improves overall but fluctuates substantially, with later, longer prompts not always

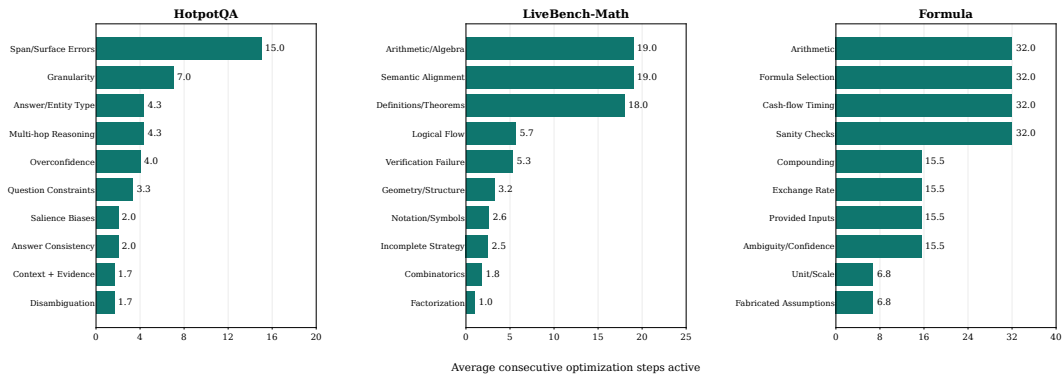


Figure 17: Average persistence of failure topics across optimization iterations for HotPotQA, LiveBench-Math, and Formula. Persistence is measured as average run length: the number of consecutive iterations for which a failure topic remains active.

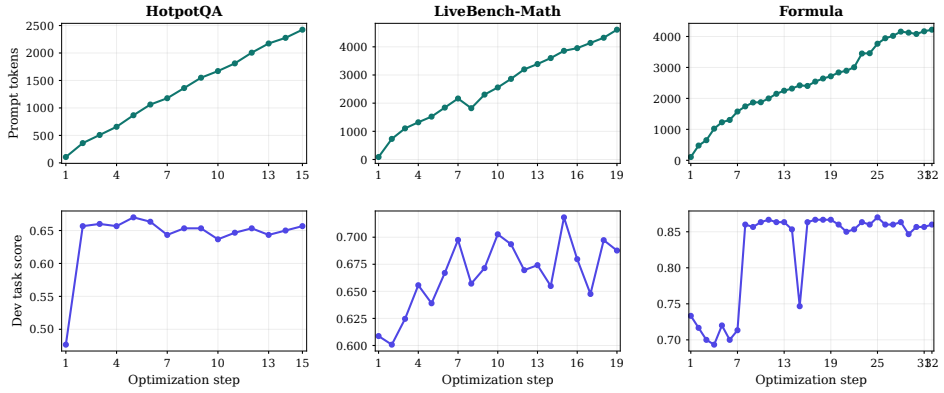


Figure 18: Prompt length and development-set task score across RPT optimization iterations for HotPotQA, LiveBench-Math, and Formula. Prompt length generally increases over time, while development performance improves early and then plateaus or fluctuates, motivating development-set prompt selection rather than selecting the final iteration by default.

outperforming earlier shorter ones. On Formula, performance jumps after early revisions and then largely plateaus, despite continued prompt growth.

These trends suggest that longer prompts are not inherently better. Instead, useful prompt growth comes from adding targeted constraints, while later edits may introduce redundancy or task-specific overfitting. This further motivates selecting the final prompt by development-set performance, rather than simply using the last prompt produced by the optimization loop.

P Example Prompt Revision

Below, we show an example prompt revision produced by RPT for HotPotQA with GPT-5 as the optimizer. The initial prompt contains only high-level instructions to reason from context, verify support, and produce a JSON output. The selected prompt is more targeted: it adds explicit guidance for question parsing, supporting-span extraction, answer-type and granularity matching, multi-hop reasoning, temporal or comparative constraints, and confidence calibration.

This example illustrates how RPT converts diagnostic feedback into concrete prompt edits. The optimized prompt targets recurring HotPotQA failures by adding explicit controls for minimal span

extraction, surface-form matching, multi-hop relation tracing, and confidence calibration under ambiguity.

Original Prompt - HotpotQA

System Message. You are tasked with answering questions using only the provided context.
User Instruction.

- Reason step by step using only the provided context.
- Be concise but thorough in your justification.
- Before answering, verify that your answer is supported by the context.

Your output should be a JSON with fields:

- justification: a context-grounded explanation of how you reached the answer.
- answer: your concise final answer.
- confidence: a number in [0,1] representing your confidence in the final answer.

Optimized Prompt - HotpotQA

System Message. You are tasked with answering questions using only the provided context.
User Instruction. Instructions:

- Use only the provided context. Parse the question to identify: (a) the subject, (b) the relation/attribute requested, (c) any constraints (time/comparative qualifiers such as based on, before/after, first/last/only), and (d) the required answer type and granularity/format, including whether a unit or administrative/institutional qualifier is explicitly requested.
- Find the specific supporting span(s) in the context. When multiple surface forms/aliases exist, select the exact contiguous span in the context that best matches the question wording; if none match, use the canonical full form as written in context, preserving capitalization, diacritics, and suffixes/post-nominals (e.g., Jr., Sr., III). If the question asks for a unit or an administrative/institutional qualifier, select the span that includes it; if not, and both prefixed and bare forms appear, prefer the bare form.
- Answer extraction: copy the minimal contiguous span verbatim from the context (casing and numerals). Do not add extra words, qualifiers, punctuation, or units unless explicitly requested. Match the question's granularity. For numbers, output digits only; if the context has modifiers like "about/circa/approximately 100", return just the bare value "100". If a full date is given but the question asks for a year, return only the 4-digit year. If the question requests a value-with-unit, copy the exact number+unit as one contiguous span (e.g., "5 kilometers"). For place/organization names, include prefixes like "City of", "County of", "Province of", "University of", or "Department of" only if (a) the question targets that unit, or (b) the context consistently uses that prefixed form without an alternative bare form. Trim leading/trailing articles or punctuation unless they are part of the official name.
- Yes/No questions: answer exactly yes or no in lowercase with no punctuation.
- Relation/compositional queries: follow all required hops and return the final requested attribute/value, not an intermediate entity or related item. Verify subject–object direction (e.g., "X acquired Y" vs "X was acquired by Y"). Multi-hop checklist: (1) find the bridge entity tied to the subject, (2) locate the next hop that yields the requested attribute, (3) ensure the final answer span appears verbatim in the context.
- Comparative/temporal qualifiers: when prompts include first/last/only/earliest/latest/highest/lowest or a time restriction, (1) collect all candidate spans linked to the subject, (2) apply the specified ordering/constraint to choose the single correct candidate, (3) return only that final span.

- Disambiguation: if multiple candidate entities/mentions exist (including same-named entities), choose the one whose qualifiers (role, location, dates, affiliation) satisfy the question's constraints; ignore distractors.
- Before answering, run a final span checklist: (a) type and administrative level match the request, (b) temporal/comparative constraint satisfied, (c) surface form matches the question when available, else canonical full form in context, (d) span is minimal, contiguous, and free of extraneous punctuation.
- Justification: briefly cite the key supporting phrase(s) and reasoning in ≤ 3 sentences.
- Confidence calibration: set confidence based on evidence and ambiguity.
  - 0.85 if exactly one unambiguous supporting span directly answers the question with no alias/format ambiguity.
  - $\sim$0.55 if supported but there are multiple aliases/forms, qualifier/unit risks, or minor ambiguity (cap confidence here in such cases).
  - $\sim$0.35 if partially supported or requires weaker inferences across snippets.
  - $\leq$0.25 if the support is tenuous or potentially contradicted.

Output JSON fields:

- justification: a context-grounded explanation of how you reached the answer.
- answer: the bare minimal span only (no quotes or extra words).
- confidence: a number in [0,1] representing your confidence in the final answer.

Q Feedback, Update, and Memory in Reflective Optimizers

Table 12 places RPT and the reflective optimizers among our baselines in the two-stage view of Section 2, recording how each instantiates the feedback and update stages and what it carries across iterations. Not every prompt optimizer follows this reflective paradigm. MIPRO, which we also compare against, searches instead: it proposes candidate instructions and demonstrations and uses Bayesian optimization over task scores to choose among their combinations, without reflecting on failures (Opsahl-Ong et al., 2024).

Table 12: How reflective prompt optimizers instantiate the feedback and update stages, and what each carries across iterations.

Method	Feedback stage	Update stage	Carried across iterations
TextGrad	Textual gradient for each example in a minibatch, concatenated	Optimizer step rewriting the prompt, reverted if validation accuracy drops	Best prompt so far; optionally, earlier values of the prompt (momentum)
GEPA	Per-instance feedback on a minibatch	Reflective mutation, kept if the minibatch score improves	Candidate pool with per-instance scores, used to choose which candidate to mutate
ACE	Reflector critique of a single trajectory	Curator merging new entries into a playbook	Playbook entries with helpful and harmful counters, deployed with the prompt
RPT	Critiques over the full optimization set, clustered into recurring failure modes	Optimizer revision of the prompt, or a decision to stop	Reports pairing each prompt with its failures and metrics, used only during optimization

The last column relates to momentum in gradient-based optimization, where each update depends on past gradients rather than only the current one. TextGrad's momentum shows the optimizer earlier values of the optimized prompt alongside the current gradient, but not the gradients of earlier steps, whereas RPT's memory shows earlier feedback, the textual counterpart of past gradients, together with the prompts that produced it.

R Discussion and Future Work

RPT exposes diagnosis to the optimizer as a callable function rather than fixing it in the optimization loop. In this work, the optimizer's only choice about diagnosis is whether to call it again after revising the prompt or to stop, and each call evaluates the full optimization set, so the interface is exercised in a minimal form. We therefore do not attribute the reported gains to function calling itself, but to what the diagnostic function returns and what the memory retains (Section 5.3). The design does, however, allow new diagnostic capabilities to be added as further functions without changing the optimization loop: evaluation on a subset of chosen size, inspection of a single failure topic, or validation of a candidate edit before it is adopted. An optimizer given such functions would choose not only how to revise but what evidence to gather. Such choices bear directly on optimization cost: full-set diagnosis accounts for the largest share of RPT's calls (Appendix I), and an optimizer that diagnoses on a subset while exploring, and on the full set only when it expects improvement, could reduce it substantially. Whether an optimizer given these choices allocates its budget better than a fixed schedule is an open question we leave to future work.