Reflective Prompt Tuning through Language Model Function-Calling

Anonymous ACL submission

Abstract

Large language models (LLMs) have become increasingly capable of following instructions and complex reasoning, 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 manual effort while preserving inference-time flexibility. However, existing methods often search over prompt candidates or use fixed critique-refine pipelines driven by individual examples or small batches, limiting their ability to capture systematic error patterns and make targeted edits grounded in failure history. We propose Reflective Prompt Tuning (RPT), a framework that uses LLM function calling to simulate the iterative workflow of human prompt engineers. An LLM optimizer calls a diagnostic function that evaluates the target model over an entire optimization set, summarizes recurring failure modes, and returns a structured diagnostic report. The optimizer uses this report, together with an accumulated memory of prior reports, to revise the prompt for the next iteration. RPT further supports confidence-aware optimization by using calibration signals in diagnostic feedback and final prompt selection. Across three reasoning tasks, RPT improves over initial prompts by up to 12.9 points, remains competitive with state of the art, and improves confidence calibration. Our analyses show that RPT is especially effective on multi-hop and mathematical reasoning, producing targeted prompt revisions that align with diagnosed failure patterns and lead to gains in task performance and calibration.[1]

1 Introduction

Large language models (LLMs) have become increasingly adept at following instructions and performing complex reasoning, making contextual prompting the dominant mechanism for adapting model behavior to downstream tasks (Lou et al., 2024; Wei et al., 2022; Kojima et al., 2022). Prompts let users specify objectives, constraints, and output formats 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. Crafting effective prompts is often a manual and iterative process that relies on trial and error and, in some cases, requires substantial expertise (Zamfirescu-Pereira et al., 2023; Knoth et al., 2024). Moreover, LLMs exhibit unpredictable sensitivity to seemingly minor choices such as formatting, phrasing, and instruction ordering, so prompt effectiveness may not generalize reliably across settings (Zhuo et al., 2024; Sclar et al., 2024). These challenges have motivated automated prompt optimization methods that aim to reduce manual prompt-engineering effort by automatically searching for, selecting, or revising prompts based on task objectives (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, reasoning steps, or evaluator feedback, and proposes prompt revisions. However, existing methods have several limitations. First, many follow fixed context-updating pipelines. For example, ACE (Zhang et al., 2026) updates an auxiliary playbook of reusable strategies inserted into a fixed prompt template. While this can improve stability, it limits the optimizer's ability to make arbitrary prompt-level revisions. Second, updates in each iteration are often driven by individual examples (Zhang et al., 2026) or minibatch subsets (Opsahl-Ong et al., 2024a; Agrawal et al., 2026; Yuksekgonul et al., 2025), making optimization sensitive to local rather than recurring failures. Third, most methods lack

[1] We will release our code upon publication.

explicit memory over prior diagnostic reports and prompt revisions, limiting credit assignment across iterations. Finally, prompt selection is typically driven by task performance alone, leaving broader reliability properties outside the optimization criterion. Although GEPA (Agrawal et al., 2026) incorporates auxiliary evaluation signals, its prompt selection remains primarily task-performance driven.

To address these limitations, we propose Reflective Prompt Tuning (RPT), a framework that leverages LLMs' function-calling capabilities to mimic the iterative workflow of human prompt engineers. Modern LLMs can call external functions, inspect structured outputs, and reason over feedback from those calls to guide subsequent decisions. RPT builds on these capabilities by using an LLM as an active prompt optimizer that inspects model behavior and revises the prompt through an explicit diagnostic function. Starting from a seed prompt, the optimizer iteratively calls the diagnostic function to evaluate the target model and return a structured diagnostic report. This function collects behavioral traces, critiques incorrect responses by diagnosing their failure modes, clusters these diagnoses to identify recurring failure patterns, and summarizes where the current prompt breaks down. The optimizer conditions on this report together with an accumulated memory of prior reports and prompt revisions, enabling it to reason about persistent failures and previous refinement attempts rather than treating each update in isolation. RPT further supports confidence-aware optimization by incorporating calibration diagnostics into both the feedback shown to the optimizer and the development-set criterion used to select the final prompt.

We evaluate RPT on three reasoning tasks spanning 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, we compare RPT against state-of-the-art automated prompt-optimization baselines, including ACE (Zhang et al., 2026), GEPA (Agrawal et al., 2026), and MIPRO (Opsahl-Ong et al., 2024a). Across tasks, RPT consistently improves over initial prompts, achieving gains of up to +12.9 points on HotPotQA, +12.4 points on LiveBench-Math, and +11.7 points on Formula, while remaining competitive with state-of-the-art baselines. Our confidence-aware experiments further show that incorporating calibration signals into both diagnostic feedback and final prompt selection improves calibration alongside task performance. Finally, analysis of optimization traces shows that RPT produces targeted prompt revisions aligned with diagnosed failure modes, offering insight into why and how prompts are revised across iterations. Together, these results suggest that tool-calling LLMs can enable scalable and interpretable prompt optimization.

2 Reflective Prompt Tuning (RPT)

We present **Reflective Prompt Tuning** (RPT), a diagnosis-driven prompt optimization framework. RPT automates the iterative workflow of prompt engineers: run a prompt, inspect outputs, identify recurring failures, revise the prompt, and repeat. Recent advances in LLM function calling and reasoning over tool outputs enable LLMs to serve as prompt optimizers (Schick et al., 2023; Gou et al., 2024; Yuksekgonul et al., 2025). We first formulate prompt optimization as selecting a prompt that improves task performance and confidence calibration (Section 2.1). We then describe how RPT constructs diagnostic feedback and reflectively revises prompts based on diagnosed failures (Section 2.2). All prompts used in RPT are in Appendix A.

2.1 Problem Statement

Let f_θ be the target model and p_t the prompt at optimization iteration t. Given input x, the model produces

$$f_\theta(x; p_t) = (r, \hat{y}, c), \tag{1}$$

where r is the reasoning trace, $\hat{y}$ is the final answer, and c is the reported confidence. We assume an optimization set D_{train}, a development set D_{dev}, and a held-out test set D_{test}. The goal is to generate candidate prompts $\{p_0, \ldots, p_T\}$ and select a final prompt p^* using development-set performance. Let

$$\mathcal{O}(p; D) = \{\mu_1(p; D), \ldots, \mu_n(p; D)\} \tag{2}$$

denote the set of evaluation metrics for prompt p on dataset D, including task performance metrics and confidence calibration error. We use a scalar selection function Φ to combine these metrics and select the final prompt:

$$p^* = \arg\max_{p_t \in \{p_0, \ldots, p_T\}} \Phi\left(\mathcal{O}(p_t; D_{\text{dev}})\right) \tag{3}$$

Appendix F further shows that prompts often grow during optimization, but longer prompts do

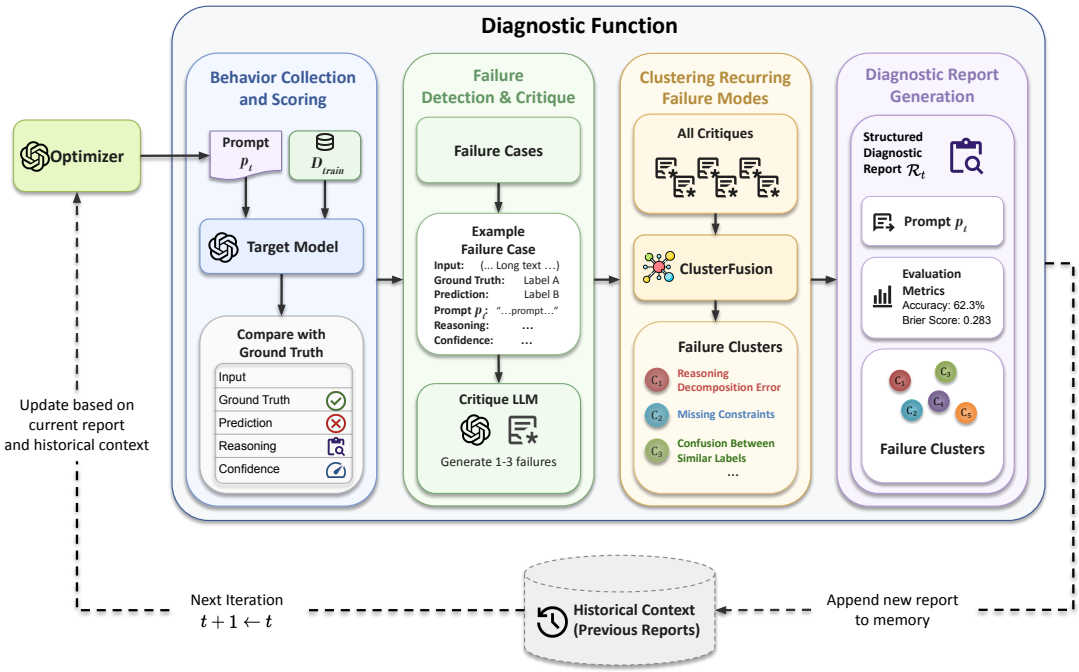

Figure 1: Overview of Reflective Prompt Tuning (RPT). At each iteration, the optimizer calls a diagnostic function that evaluates the current prompt on D_{train}, critiques failures, clusters recurring failure modes, and returns a structured report. The optimizer uses this report and prior reports to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

not necessarily yield better development performance, motivating development-set selection. In the confidence-aware setting, Φ jointly accounts for task performance and calibration by rewarding higher task scores while penalizing miscalibration, for example through a negative Brier-score term. The selected prompt p^* is then evaluated on the held-out test set D_{test}.

2.2 Methodology Overview

We formulate RPT as a two-stage textual update process, illustrated in Figure 1. First, RPT constructs response-level feedback (Section 2.2.1): given the current prompt p_t, the diagnostic function evaluates target-model outputs on D_{train}, critiques incorrect responses, identifies recurring failure modes, and summarizes them with aggregate metrics into a diagnostic report $\mathcal{R}_t$. Second, RPT translates this report into a prompt-level revision (Section 2.2.2): conditioned on p_t, $\mathcal{R}_t$, and a memory of prior reports, optimizer infers likely prompt shortcomings and produces the next prompt p_{t+1}.

2.2.1 Constructing Diagnostic Feedback

The diagnostic function connects target-model behavior to the optimizer LLM. Given the current prompt p_t, the optimizer invokes this function to evaluate the target model on the full optimization set D_{train} and return a structured diagnostic report $\mathcal{R}_t$. The report captures not only *how well* the prompt performs, but also how target-model outputs fail and which failures recur across the dataset.

Behavior collection and scoring. The diagnostic function first runs the target model f_θ with prompt p_t on each example $(x_i, y_i) \in D_{\text{train}}$. For each example, it records the reasoning trace r_i, final answer $\hat{y}_i$, and reported confidence c_i. It then computes task-specific performance metrics, along with average confidence and Brier score for calibration. These metrics capture overall prompt quality, but do not explain the causes of failures.

Failure detection and critique. Next, the function identifies failed examples using the task-specific evaluator:

$$\mathcal{I}_t = \{(x_i, y_i, \hat{y}_i, r_i, c_i) \mid i \text{ is incorrect}\} \quad (4)$$

Next, for each failed example $i \in \mathcal{I}_t$, a critique LLM generates concise response-level diagnoses of how the target-model output fails with respect to the expected answer y_i and the evaluation criteria. Since RPT elicits confidence as part of the target-model output, the critique also assesses whether the reported confidence c_i is appropriate based on the response's correctness and quality. These diagnoses capture local issues such as incorrect reason-

ing, unsupported evidence use, formatting errors, or overconfident incorrect answers.

Each failed instance may yield up to three diagnoses to improve coverage and reduce sensitivity to individual critiques. Let the resulting pool of sample-level failure diagnoses be:

$$\mathcal{Z}_t = \{z_{i,j} \mid i \in \mathcal{I}_t,\ j \leq 3\} \tag{5}$$

Identifying recurring failure modes. The diagnoses in $\mathcal{Z}_t$ provide local feedback about individual failures, but prompt revision benefits from identifying patterns that recur across the optimization set. To convert response-level critiques into dataset-level diagnostic feedback, RPT applies ClusterFusion (Xu et al., 2025) to $\mathcal{Z}_t$, grouping semantically similar diagnoses into recurring failure topics:

$$\mathcal{C}_t = \{(a_k, d_k, S_k)\}_{k=1}^{K}, \tag{6}$$

where a_k is a short topic label, d_k describes the failure mode, and S_k contains representative examples. This aggregation compresses local critiques into a compact summary of systematic target-model failures, helping the optimizer infer prompt-level shortcomings and propose targeted revisions. The number of topics K controls the summary granularity (details on K selection in Appendix C).

Diagnostic report generation. The diagnostic function returns a structured report

$$\mathcal{R}_t = \left(p_t, \mathcal{O}(p_t; D_{\text{train}}), \mathcal{C}'_t\right), \tag{7}$$

where p_t is the current prompt, $\mathcal{O}(p_t; D_{\text{train}})$ contains aggregate metrics, and $\mathcal{C}'_t \subseteq \mathcal{C}_t$ denotes the retained subset of clustered failure topics, with representative examples and summaries. We retain a subset $\mathcal{C}'_t$ to keep the report focused on prominent recurring patterns; details are provided in Appendix C. Together, these components turn feedback from scalar scoring into structured diagnosis.

History is maintained in an external memory outside the diagnostic function. At iteration t, the optimizer receives the current report $\mathcal{R}_t$ together with prior reports $\mathcal{M}_{<t}$. After the iteration, the current report is appended for future use:

$$\mathcal{M}_{<t+1} = \text{Append}(\mathcal{M}_{<t}, \mathcal{R}_t). \tag{8}$$

This lets the optimizer reason over the optimization trajectory rather than only the current report. In practice, memory grows linearly with the iteration budget T, but remains manageable because each report stores only aggregate metrics and a filtered set of recurring failure clusters.

2.2.2 Reflective Prompt Revision with Memory

Given the current diagnostic report $\mathcal{R}_t$ and the external memory of prior reports $\mathcal{M}_{<t}$, the optimizer identifies which recurring response-level failures indicate shortcomings of the current prompt and generates a revision. Formally,

$$p_{t+1} = \text{LLM}_{\text{opt}}(p_t, \mathcal{R}_t, \mathcal{M}_{<t}). \tag{9}$$

The optimizer treats diagnostic reports as evidence for revision: it inspects aggregate metrics, recurring failure topics, representative examples, and previous prompt changes.

The external memory $\mathcal{M}$ helps address the credit-assignment challenge in prompt optimization (Opsahl-Ong et al., 2024b; Yuksekgonul et al., 2025). A prompt edit may improve some metrics while worsening others, a failure may require several revisions to resolve, and repeated failures may indicate ineffective prior edits. By conditioning on prior reports, the optimizer can track persistent failures, previous revision attempts, and performance changes over time. Thus, RPT treats history as memory over the optimization trajectory rather than treating each update as an independent proposal.

3 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)). Additional dataset statistics and details can be found in Appendix B

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

Baselines. We compare RPT against three state-of-the-art automated prompt-optimization baselines: Agentic Context Engineering (ACE; Zhang et al. (2026)), GEPA (Agrawal et al., 2026), and MIPRO (Opsahl-Ong et al., 2024b). Additional baseline and implementation details are provided in Appendix C.

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	40.7	47.6	73.5	76.5	63.6
	RPT (ours)	55.5	**68.4**	58.1	**70.5**	72.3	84.0	**74.3**
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**	40.7	43.8	73.5	77.5	62.3
	RPT (ours)	55.5	65.2	58.1	**66.2**	72.3	74.0	68.5
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	40.7	43.2	73.5	75.5	61.3
	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	40.7	42.5	73.5	74.0	60.0
	RPT (ours)	55.5	64.6	58.1	**64.9**	72.3	73.5	67.7

Table 1: Benchmark results for different prompt optimizers evaluated on GPT-4.1 as the target model. Columns report task-specific metrics: accuracy for HotPotQA and Formula, and task score for LiveBench-Math. Final denotes performance of the optimized prompt. Best final scores within each optimizer and dataset are shown in **bold**. RPT consistently improves over initial prompts and remains competitive with state-of-the-art baselines.

Evaluation. We report task-specific performance metrics: accuracy for HotPotQA and Formula, and task score for LiveBench-Math[2]. For calibration, we report Brier score using the model's verbalized confidence (Xiong et al., 2024).

4 Results and Analyses

We evaluate RPT from three perspectives. First, we compare RPT-optimized prompts against seed prompts and state-of-the-art baselines, while studying the effect of optimizer LLM size (Section 4.1). Second, we examine whether confidence-aware optimization improves calibration without sacrificing task performance (Section 4.2). Finally, we analyze optimization traces to study persistent failures, diagnosis–patch alignment, and associations with subsequent performance gains (Section 4.3).

4.1 RPT Is Competitive with SOTA Baselines

Table 1 reports the task performance of prompts optimized by RPT and the baseline prompt optimizers described in Section 3. For each task and method, we report the performance of the initial prompt and the performance of the optimized prompt selected via development-set performance[3].

[2]Following LiveBench, task score is averaged across four math tasks: https://github.com/LiveBench/LiveBench/tree/main/livebench/process_results/math.

[3]For Formula, we use the initial prompt from ACE; for HotPotQA, we adapt this template to the QA setting; and for LiveBench-Math, we adapt the initial prompt from GEPA.

Observation 1: RPT is strongest on tasks with recurring reasoning failures. Across optimizer LMs, RPT achieves the best final performance on LiveBench-Math for every optimizer setting, improving over the initial prompt by up to +12.4 points. On HotPotQA, RPT is also competitive: it achieves the best final performance with GPT-5 and remains close to the strongest baseline under other instantiations. GEPA and MIPRO perform competitively on HotPotQA, but provide smaller gains on LiveBench-Math; their lower initial scores also suggest that implementation-specific choices affect their absolute performance. Formula shows a different pattern: ACE consistently achieves the best final performance, while RPT is competitive mainly when paired with GPT-5. More broadly, RPT appears well-suited to tasks where recurring failures can be diagnosed and translated into targeted prompt revisions. However, it may be less advantageous for domain-specific computation, where localized instance-level updates or predefined prompt structures may be more effective.

Observation 2: RPT benefits from stronger optimizer LLMs. Optimizer choice has a clear impact on RPT's performance. Compared to GPT-5-mini, using GPT-5 increases RPT's Aggregate score from 68.5 to 74.3, with gains across all three tasks. Within the Gemini family, Gemini-3.1-Pro similarly improves over Gemini-3.1-Flash-Lite, increasing Aggregate from 67.7 to 70.1. This pattern

Optimizer LLM	Method	HotPotQA		LiveBench-Math		Formula	
		Task Score ↑	Brier ↓	Task Score ↑	Brier ↓	Task Score ↑	Brier ↓
GPT-5	GEPA-C	57.6→66.8	.410→.296	40.1→42.2	.543→.519	73.5→75.5	.262→.246
	RPT	55.5→**68.4**	.438→**.241**	58.1→**70.5**	.347→**.174**	72.3→**84.0**	.272→**.129**
GPT-5-mini	GEPA-C	57.6→**65.6**	.410→.309	40.1→40.1	.543→.543	73.5→72.5	.262→.260
	RPT	55.5→65.2	.438→.309	58.1→**65.6**	.347→**.287**	72.3→**74.0**	.272→**.256**
Gemini-3.1-Pro	GEPA-C	57.6→**68.0**	.410→**.306**	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	57.6→**66.4**	.410→**.325**	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

Table 2: Confidence-aware optimization results. GEPA-C denotes GEPA with confidence feedback. Each cell reports initial→final performance and Brier; higher performance and lower Brier are better.

is expected because RPT places a demanding burden on the optimizer: it must perform credit assignment over diagnostic feedback and prior prompt revisions, identify unresolved failures, and translate recurring failure modes into targeted prompt edits. Compared with the baselines, RPT achieves the best aggregate performance with GPT-5 and is nearly tied with ACE under Gemini-3.1-Pro, while ACE remains stronger with smaller optimizer LLMs. GEPA and MIPRO generally trail in aggregate performance, partly due to lower initial prompt performance on LiveBench-Math.

4.2 Confidence Signals Improve Calibration

We next ask whether confidence-aware prompt optimization can improve both task performance and calibration. This matters because verbalized confidence is often used as a proxy for answer reliability in abstention, routing, human review, and risk-sensitive deployment (Wen et al., 2025; Chuang et al., 2025; dela Cruz et al., 2025; Wang et al., 2026). ACE and MIPRO do not directly expose calibration diagnostics to the optimizer without substantial modification, while GEPA can use them as auxiliary feedback. In contrast, RPT incorporates calibration into both diagnostic feedback and final prompt selection

Table 2 compares RPT with confidence-aware GEPA. GEPA shows that calibration feedback can help: on HotPotQA, it improves both task performance and Brier score across optimizer LLMs. However, gains are more limited on LiveBench-Math and Formula. With GPT-5-mini as optimizer, confidence feedback yields no gain on LiveBench-Math and slightly hurts Formula performance, suggesting that it may distract a less capable optimizer. RPT more consistently improves both task performance and calibration. Although prompt optimization cannot access internal uncertainty estimates or logits, our results show that calibration can improve when treated as a first-class optimization signal. By incorporating calibration into both the diagnostic loop and prompt-selection objective, RPT better aligns self-reported confidence with empirical correctness while also improving task performance.

4.3 What Does RPT Learn from Diagnostics?

Beyond final task performance, RPT produces structured optimization traces at each iteration. We analyze these traces to understand how RPT improves prompts over time, focusing on the GPT-5 optimizer since it performs best in our experiments.

Across tasks, we collect failure diagnoses from each iteration and derive prompt-update instances by using GPT-4.1 to extract atomic differences between consecutive prompts, p_t and p_{t+1}. We then apply ClusterFusion, as described in Section 2, to group diagnoses and prompt updates into 10 *failure* topics and 10 *patch* topics, respectively. To relate topics to performance, we compute next-iteration metric changes by comparing metrics under p_t with those after evaluating p_{t+1}. Thus, positive Δ task score and negative Δ Brier indicate improvement. Because this analysis relies on optimization traces, we interpret results as associations rather than causal effects.

4.3.1 Does RPT Produce Targeted Revisions?

We next examine whether RPT performs targeted credit assignment from diagnosed failures to prompt revisions. For each failure topic F_i and patch topic P_j, we compute $P(P_j \mid F_i)$ as the fraction of transitions containing P_j among those containing F_i. Failure topics diagnosed at iteration t are assigned to transition $t \rightarrow t+1$, and patch topics are extracted from the corresponding prompt

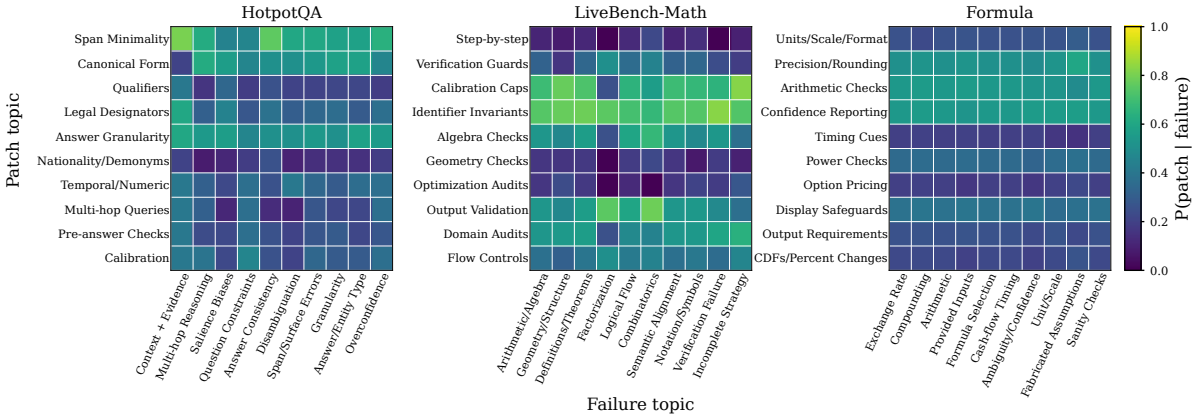

Figure 2: Failure-to-patch alignment across datasets. Each heatmap reports $P(\text{patch topic} \mid \text{failure topic})$, showing which prompt revisions tend to follow each diagnosed failure type.

update (topic presence is binary within each transition). This measures whether specific failures systematically lead to specific prompt edits, which would indicate targeted credit assignment rather than generic prompt rewriting.

Figure 2 shows that the specificity of this failure-to-patch mapping varies across tasks. On HotPotQA, several answer-control patches, such as span minimality, canonical-form preference, and answer granularity matching, appear across many failure types, reflecting the benchmark's sensitivity to exact answer form. However, multi-hop reasoning failures more often trigger relation- and query-handling patches, suggesting meaningful failure-specific credit assignment beyond generic answer-format control (optimized HotPotQA prompt in Appendix G). On LiveBench-Math, the alignment concentrates around verification-oriented patches, including stepwise protocols, arithmetic checks, output validation, and notation or invariant handling. This indicates that the optimizer maps diverse mathematical failures to structured reasoning and checking mechanisms.

Formula exhibits a broader pattern: many distinct failure topics lead to similar domain-level safeguards rather than sharply different patches. This suggests that RPT identifies relevant domain controls, but performs less fine-grained credit assignment on this task than on HotPotQA or LiveBench-Math. This weaker failure-specificity may partly explain why RPT yields smaller gains on Formula and falls behind ACE (Table 1).

4.3.2 Do Prompt Patches Predict Gains?

We next examine whether prompt updates are followed by improvements in task performance or calibration. For each patch topic present at iteration t, we compute the average change in task score and Brier score after evaluating the revised prompt p_{t+1} on D_{train}. This analysis identifies which edits tend to precede better task performance or calibration.

Figure 3 shows that 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 precision/rounding, power-consistency checks, and calibration-related patches yield smaller gains. Overall, RPT's prompt revisions are often useful and failure-specific: patches introducing verification steps, answer-form constraints, arithmetic checks, or unit-handling rules tend to improve task score while reducing Brier. Formula is more mixed, with some specialized safeguards showing weak or negative short-term associations, likely due to targeting harder or more persistent domain-specific failures. Appendix E shows that the most actionable diagnoses translate into explicit behavioral constraints, while Appendix D shows that the most persistent failures are task-specific reasoning errors.

5 Related Work

Prompting offers a flexible way to adapt LLMs to downstream tasks without parameter updates.

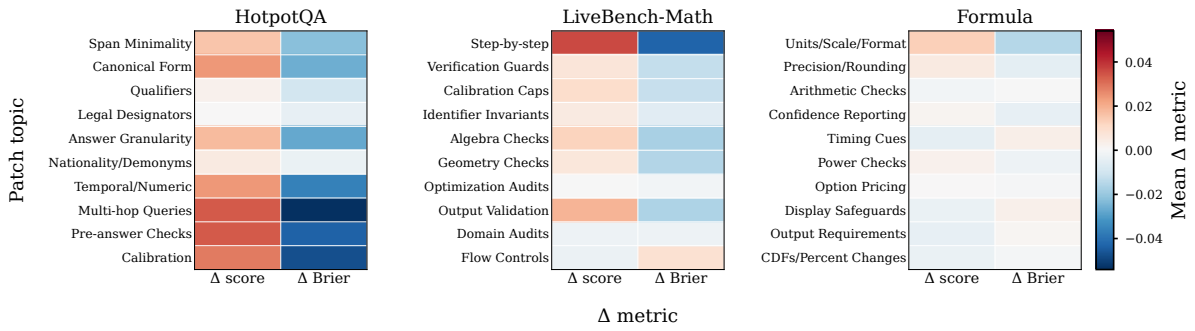

Figure 3: Patch topics and next-iteration metric changes. Each cell reports the average Δ task score or Δ Brier on D_{train} after a prompt update containing that patch; higher Δ task score and lower Δ Brier indicate improvement.

However, prompt design remains labor-intensive and sensitive to formatting, phrasing, demonstrations, and instruction order (Lu et al., 2022; Sclar et al., 2023), motivating automated prompt optimization methods that reduce manual effort (Shin et al., 2020; Yuksekgonul et al., 2025; Opsahl-Ong et al., 2024a; Agrawal et al., 2026).

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., 2024b). RPT also uses LLMs for prompt optimization, but leverages function calling to simulate the iterative workflow of human prompt engineers: evaluating the current prompt, diagnosing systematic failures, and using structured diagnostic feedback for explicitly diagnosis-driven revisions.

Reflective optimization methods. Recent methods use rich textual feedback to guide prompt or program optimization. TextGrad backpropagates natural-language feedback through computation graphs (Yuksekgonul et al., 2025), while GEPA uses execution and evaluation traces as reflective feedback for prompt proposals (Agrawal et al., 2026). MIPRO addresses credit assignment in multi-stage LLM programs with program- and data-aware proposal strategies and Bayesian search (Opsahl-Ong et al., 2024b). RPT builds on reflective feedback, but centers each iteration on a diagnostic function that evaluates the current prompt over the full optimization split and returns aggregate metrics and recurring failures. The optimizer uses this report together with prior reports to guide the next update. While GEPA can optionally use confidence and calibration as auxiliary signals, RPT incorporates them directly into both the diagnostic report and final prompt-selection criterion.

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 to improve future inference or context construction (Krause et al., 2019; Zhang et al., 2026). RPT uses memory at prompt-optimization level: conditioning on prior reports and prompt revisions helps the optimizer reason over past refinements, avoid repetitive edits, and improve credit assignment.

6 Conclusion

We introduced Reflective Prompt Tuning (RPT), a diagnosis-driven framework that uses LLM function calling to optimize prompts through structured feedback and memory over prior revisions. Across three reasoning tasks, RPT improves over seed prompts and remains competitive with state of the art, especially on multi-hop and mathematical reasoning. We also show that confidence-aware optimization improves calibration alongside task performance, and that RPT produces prompt revisions aligned with diagnosed failures. These results highlight function-calling LLMs as a promising approach to scalable and interpretable prompt tuning.

Limitations

Our study has several limitations. First, we evaluate RPT on three reasoning tasks: multi-hop question answering, mathematical reasoning, and domain-specific numerical reasoning. While these tasks cover different forms of reasoning, they do not capture the full range of prompt-optimization settings, such as open-ended generation, coding, dialogue, tool-using agents, or long-horizon interactive tasks. In addition, our experiments use GPT-4.1 as the target model and frontier proprietary LLMs as optimizers. The effectiveness of RPT may differ for smaller open-source models, weaker optimizer LMs, or settings where function calling is unavailable.

Second, RPT is more computationally expensive than prompt optimizers that use individual examples or small minibatches. Each iteration evaluates the target model on the full optimization set, critiques failed examples, clusters diagnoses, and conditions the optimizer on prior reports. Although the diagnostic reports are compressed and the memory remains manageable under our iteration budgets, scaling RPT to much larger datasets or longer optimization trajectories may require more aggressive sampling, report compression, or retrieval over memory.

Finally, RPT improves prompts but cannot guarantee that prompting alone can resolve all failures. Some persistent errors, especially deeper mathematical reasoning failures or domain-specific convention errors, may require complementary interventions such as better tools, external validators, retrieval, fine-tuning, or changes to the target model itself. Similarly, our confidence-aware setting relies on verbalized confidence, which is only a black-box proxy for uncertainty. Although RPT improves calibration in our experiments, self-reported confidence may remain sensitive to prompting and should be validated carefully before being used in high-stakes downstream decisions.

Ethics Statement

RPT is a general prompt-optimization framework, and its impact depends on the task objective, data, and target model to which it is applied. A potential risk is that the same diagnosis-driven optimization loop could be used to improve prompts for harmful applications, such as generating misleading content, evading safety filters, or improving model behavior on undesirable objectives. More broadly, optimizing prompts for a narrow development objective can amplify dataset biases or lead to overfitting to benchmark-specific artifacts.

There are also privacy and data-governance concerns when applying RPT to proprietary or user-generated data. Diagnostic reports may contain representative examples or summaries of model failures; if the optimization data includes sensitive information, these reports could inadvertently expose private content. In our experiments, we use benchmark datasets and retain filtered failure clusters rather than all instance-level critiques, but deployments should apply appropriate data minimization, access control, and redaction procedures.

Finally, RPT relies on strong proprietary LLMs for optimization in our main experiments, which may introduce reproducibility, cost, and access concerns. We release code and experimental details to support reproducibility, but future work should study open-model instantiations and lower-cost variants.

ChatGPT was used for writing support and minor editing. All scientific contributions, experimental design, analysis, and conclusions are the responsibility of the authors.

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. 2026. Gepa: Reflective prompt evolution can outperform reinforcement learning.

Yu-Neng Chuang, Prathusha Kameswara Sarma, Parikshit Gopalan, John Boccio, Sara Bolouki, Xia Hu, and Helen Zhou. 2025. Learning to route llms with confidence tokens. *Preprint*, arXiv:2410.13284.

Jane Arleth dela Cruz, Iris Hendrickx, and Martha Larson. 2025. Evaluating large language models for confidence-based check set selection. In *Findings of the Association for Computational Linguistics: ACL 2025*, pages 16249–16265, Vienna, Austria. Association for Computational Linguistics.

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

GoogleAI. 2026b. 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/.

Zhibin Gou, Zhihong Shao, Yeyun Gong, yelong shen, Yujiu Yang, Minlie Huang, Nan Duan, and Weizhu

Chen. 2024. ToRA: A tool-integrated reasoning agent for mathematical problem solving. In *The Twelfth International Conference on Learning Representations*.

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

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. 2024. Dspy: Compiling declarative language model calls into self-improving pipelines.

Nils Knoth, Antonia Tolzin, Andreas Janson, and J. Leimeister. 2024. Ai literacy and its implications for prompt engineering strategies. *Comput. Educ. Artif. Intell.*, 6:100225.

Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. 2022. 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. Curran Associates Inc.

Ben Krause, Emmanuel Kahembwe, Iain Murray, and Steve Renals. 2019. Dynamic evaluation of transformer language models. *Preprint*, arXiv:1904.08378.

Renze Lou, Kai Zhang, and Wenpeng Yin. 2024. Large language model instruction following: A survey of progresses and challenges. *Computational Linguistics*, 50(3):1053–1095.

Yao Lu, Max Bartolo, Alastair Moore, Sebastian Riedel, and Pontus Stenetorp. 2022. Fantastically ordered prompts and where to find them: Overcoming few-shot prompt order sensitivity. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics*, pages 8086–8098. Association for Computational Linguistics.

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

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

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

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

Reid Pryzant, Dan Iter, Jerry Li, Yin Lee, Chenguang Zhu, and Michael Zeng. 2023. Automatic prompt optimization with "gradient descent" and beam search. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 7957–7968, Singapore. Association for Computational Linguistics.

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, and 2 others. 2025. A systematic survey of automatic prompt optimization techniques. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 33078–33110, Suzhou, China. Association for Computational Linguistics.

Pranab Sahoo, Ayush Kumar Singh, Sriparna Saha, Vinija Jain, Samrat Mondal, and Aman Chadha. 2025. A systematic survey of prompt engineering in large language models: Techniques and applications. *Preprint*, arXiv:2402.07927.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessí, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: language models can teach themselves to use tools. In *Proceedings of the 37th International Conference on Neural Information Processing Systems*, NIPS '23, Red Hook, NY, USA. Curran Associates Inc.

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, and 12 others. 2025. The prompt report: A systematic survey of prompt engineering techniques. *Preprint*, arXiv:2406.06608.

Melanie Sclar, Yejin Choi, Yulia Tsvetkov, and Alane Suhr. 2023. Quantifying language models' sensitivity to spurious features in prompt design or: How i learned to start worrying about prompt formatting. *arXiv preprint arXiv:2310.11324*.

Melanie Sclar, Yejin Choi, Yulia Tsvetkov, and Alane Suhr. 2024. 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.

Taylor Shin, Yasaman Razeghi, Robert L. Logan IV, Eric Wallace, and Sameer Singh. 2020. AutoPrompt: Eliciting Knowledge from Language Models with Automatically Generated Prompts. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 4222–4235, Online. Association for Computational Linguistics.

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

Dannong Wang, Jaisal Patel, Daochen Zha, Steve Y. Yang, and Xiao-Yang Liu. 2025. Finlora: Benchmarking lora methods for fine-tuning llms on financial datasets. *Preprint*, arXiv:2505.19819.

Jiawei Wang, Yanfei Zhou, Siddartha Devic, and Deqing Fu. 2026. Are llm decisions faithful to verbal confidence? *Preprint*, arXiv:2601.07767.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. 2022. 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. Curran Associates Inc.

Bingbing Wen, Jihan Yao, Shangbin Feng, Chenjun Xu, Yulia Tsvetkov, Bill Howe, and Lucy Lu Wang. 2025. Know your limits: A survey of abstention in large language models. *Preprint*, arXiv: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. 2025. Livebench: A challenging, contamination-free LLM benchmark. In *The Thirteenth International Conference on Learning Representations*.

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

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

Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V Le, Denny Zhou, and Xinyun Chen. 2024. Large language models as optimizers. *The Twelfth International Conference on Learning Representations*.

Zhilin Yang, Peng Qi, Saizheng Zhang, Yoshua Bengio, William Cohen, Ruslan Salakhutdinov, and Christopher D. Manning. 2018. HotpotQA: A dataset for diverse, explainable multi-hop question answering. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 2369–2380, Brussels, Belgium. Association for Computational Linguistics.

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

J.D. Zamfirescu-Pereira, Richmond Y. Wong, Bjoern Hartmann, and Qian Yang. 2023. 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*.

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. 2026. Agentic context engineering: Evolving contexts for self-improving language models.

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

Yongchao Zhou, Andrei Ioan Muresanu, Ziwen Han, Keiran Paster, Silviu Pitis, Harris Chan, and Jimmy Ba. 2023. Large language models are human-level prompt engineers. *Preprint*, arXiv:2211.01910.

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

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 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.

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

Table 3: Datasets used for prompt optimization, prompt selection, and final evaluation.

B 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 retrieve math questions from the `2024-08-31` LiveBench release (most recent to date). 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 3.

C Additional Experimental Details

RPT configuration. Each RPT optimizer is instructed to call a single diagnostic function at the beginning of each iteration. The diagnostic function evaluates the current prompt on the optimization split, collects the target model's structured outputs, critiques incorrect responses, clusters failure diagnoses with ClusterFusion (Xu et al., 2025), and summarizes recurring failure modes together with aggregate metrics. The optimizer then uses this diagnostic report, along with the memory of prior reports, to revise the prompt or stop if performance has plateaued or the iteration budget has been reached.

Clustering model. For failure-mode clustering, we use ClusterFusion with GPT-4.1 (OpenAI, 2025b) as the clustering model across all RPT instantiations. Based on empirical tuning, we set the number of clusters to $K = 10$ for HotPotQA and LiveBench-Math, and $K = 20$ for Formula, which has a larger optimization set. We keep K fixed across optimizer LLMs for each dataset. To keep the diagnostic report focused on prominent recurring patterns, we include only clusters whose size exceeds 10% of the total diagnosis pool.

Baseline configuration. ACE, GEPA, and MIPRO are run with the same target model and task splits as RPT. For each baseline, we follow the settings in the original paper or released implementation when available.

For MIPRO, we use the official DSPy[4] implementation of MIPROv2, which jointly optimizes instructions and few-shot demonstrations, with `auto="heavy"` to maximize optimization performance. For GEPA, we use the released implementation and adapt the AIME configuration, scaling the optimization budget according to the relative train/development set sizes in our tasks. For ACE, we use the released repository for Formula and adapt its instructions to the other datasets. We keep the default setting of one epoch for all tasks[5].

For GEPA, we additionally evaluate a confidence-aware variant in which confidence and calibration diagnostics are provided as auxiliary side information to the reflection prompt, while prompt selection remains driven primarily by task performance.

For baseline implementations and external artifacts, we use publicly available resources and comply with their respective licenses and terms of use.

D 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

[4] https://dspy.ai/api/optimizers/MIPROv2/
[5] https://github.com/ace-agent/ace

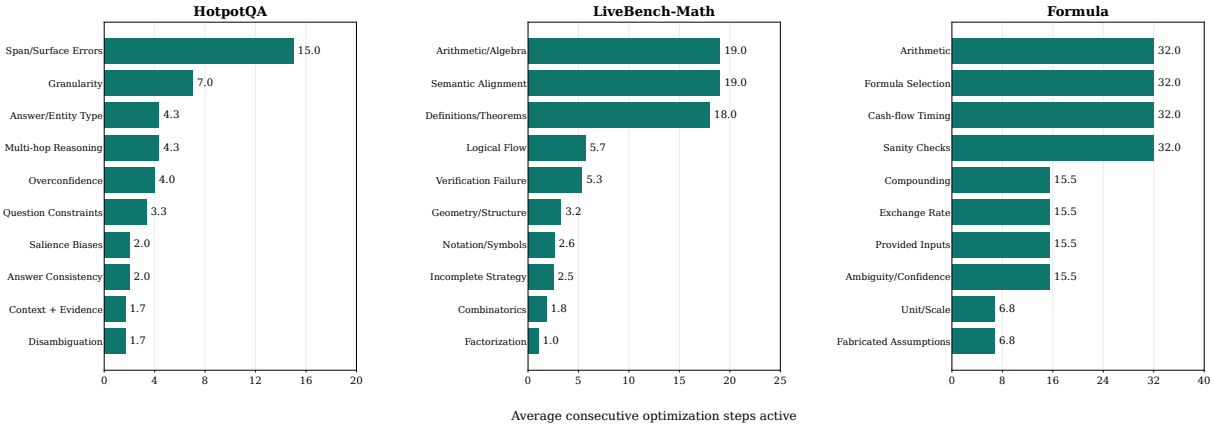

Figure 4: Average persistence of failure topics across optimization iterations for HotPotQA, LiveBench-Math, and Formula. Persistence is measured as average run length: the number of consecutive iterations for which a failure topic remains active.

active. Longer runs indicate failures that persist despite prompt revisions, while shorter runs suggest failures that are more transient or easier to address.

Figure 4 shows that persistent failures are task-specific. On HotPotQA, the longest-lived issues involve span extraction, surface-form errors, granularity or answer-type mismatches, and multi-hop reasoning. On LiveBench-Math, persistent failures center on arithmetic and algebraic computation, semantic misalignment, and misuse of mathematical definitions or conventions. On Formula, they involve arithmetic calculation, metric definition and formula selection, timing conventions, and domain constraints. Overall, the most persistent failures are not generic formatting errors, but deeper task-specific reasoning failures. This suggests that some failures require repeated refinement, and others may reflect limitations that are difficult to overcome through prompting alone.

E Actionability of Diagnosed Failure Modes

We analyze which diagnosed failure modes are most actionable for the optimizer. For each failure topic active at iteration t, we compute the average change in task score and Brier score after evaluating the revised prompt at iteration $t + 1$. This analysis is associative rather than causal, but indicates which diagnosed failures tend to precede useful prompt updates.

Figure 5 shows that actionable diagnoses vary by task. On HotPotQA, failures involving multi-hop reasoning, question-cue interpretation, span extraction, and granularity mismatches are followed by some of the largest task-score gains and Brier reductions, suggesting that concrete answer-selection errors can often be translated into effective prompt edits. On LiveBench-Math, diagnoses such as combinatorial errors, notation confusion, logical-flow failures, and arithmetic or theorem-application errors are generally followed by task-score gains, though persistent mathematical failures often require multiple revisions.

On Formula, failure topics are less discriminative: most are followed by small task-score gains and Brier reductions, likely because domain-specific failures often co-occur. Overall, RPT is most effective when diagnoses can be converted into explicit behavioral constraints, such as answer-span control, verification steps, or unit and format handling. More complex mathematical or domain-convention failures remain useful to diagnose, but may require complementary interventions beyond prompt revision.

F Prompt Length and Development Performance

We analyze how prompt length changes during RPT optimization and how these changes relate to development-set performance. Figure 6 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 con-

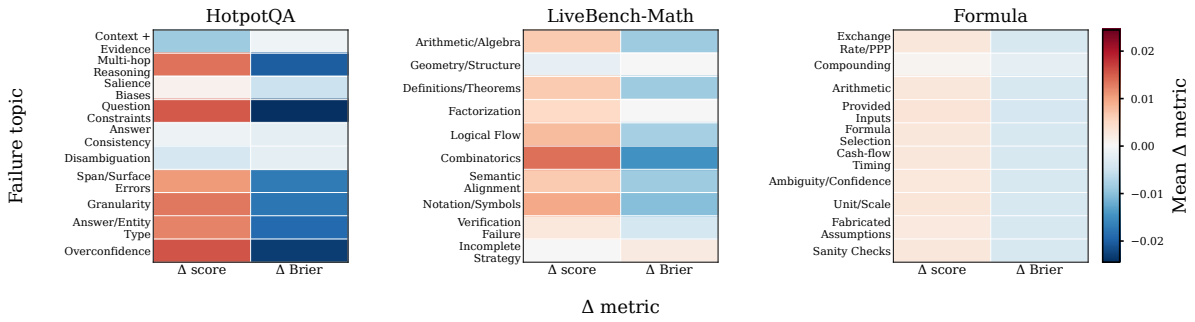

Figure 5: Average next-iteration metric changes associated with each diagnosed failure topic for HotPotQA, LiveBench-Math, and Formula. For each failure topic present at iteration t, we report the mean change in task score and Brier score from t to $t+1$. Higher Δ task score and lower Δ Brier indicate improvement.

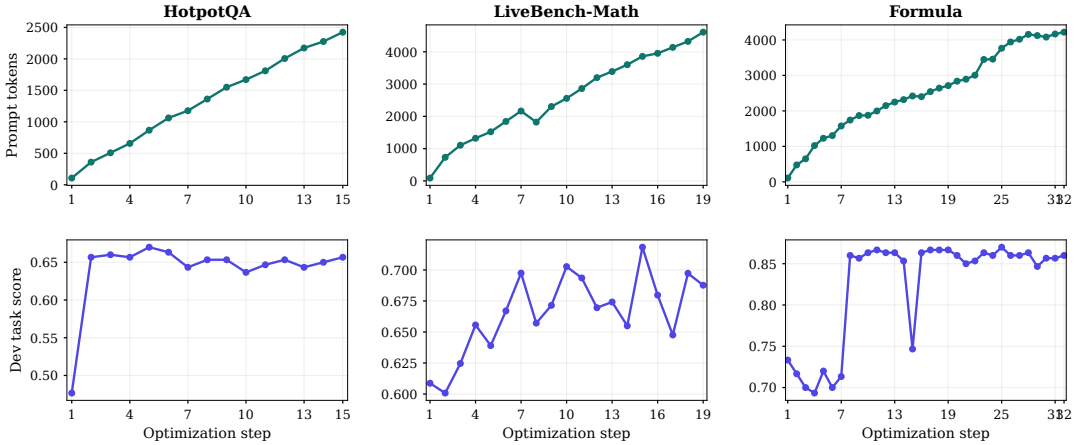

Figure 6: Prompt length and development-set task score across RPT optimization iterations for HotPotQA, LiveBench-Math, and Formula. Prompt length generally increases over time, while development performance improves early and then plateaus or fluctuates, motivating development-set prompt selection rather than selecting the final iteration by default.

tinues to grow. On LiveBench-Math, dev performance improves overall but fluctuates substantially, with later, longer prompts not always 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.

G 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.