EEVEE: POST-DEPLOYMENT PROMPT LEARNING ACROSS HETEROGENEOUS TASKS

Anonymous authors
Paper under double-blind review

ABSTRACT

Prompt learning has become an important approach to adapting large language models to downstream tasks. However, some existing methods require access to model internals, while others are largely designed for single-dataset settings, limiting their practical applicability. In real-world deployments, models are often accessible only as black boxes, while incoming queries form heterogeneous streams drawn from multiple datasets, domains, and task distributions. We therefore propose EEVEE, a *multi-dataset post-deployment prompt learning* framework for LLM agents, enabling prompt learning under heterogeneous task streams. To mitigate cross-dataset interference, EEVEE introduces a *router* that partitions incoming inputs into task clusters and assigns them to suitable prompt configurations. This design is optimized via a *router-prompt co-evolution strategy*, which employs interleaved router and prompt learning phases to address their mutual dependency. Experiments across multiple datasets demonstrate that the framework improves robustness under heterogeneous data streams while maintaining single-benchmark learning capability and efficiency. Specifically, EEVEE improves average multi-benchmark scores by 10.38 and 24.32 points over Qwen3-4B-Instruct and DeepSeek-V3.2, surpassing SOTA methods GEPA and ACE by up to 37.2% and 48.2%.

1 INTRODUCTION

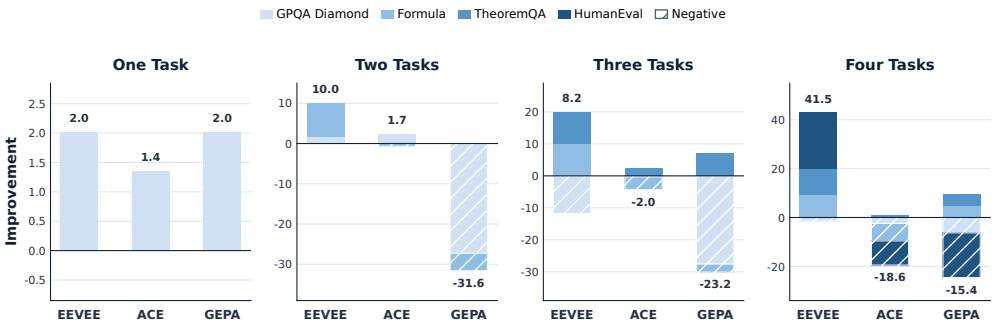


Figure 1: Incremental multi-benchmark retention as tasks are added in order: GPQA Diamond, Formula, TheoremQA, and HumanEval. Each bar stacks per-benchmark improvements for tasks seen so far: solid upward blocks are gains, and hatched downward blocks are retention losses. The number above or below each bar is the summed improvement.

Improving the capabilities of large language models has been a central goal of recent research. Model training, including supervised fine-tuning and reinforcement learning, has enabled substantial improvements in instruction following (Ouyang et al., 2022). For specific downstream tasks, prompt optimization effectively improves model behavior by refining its instructions. This approach is particularly useful when access to model internals is unavailable, making further model training impractical (Shen et al., 2023). Prior work demonstrates that automatically optimized instructions can outperform manually designed prompts on reasoning tasks (Yang et al., 2024). More recently, GEPA shows that reflective prompt optimization can outperform reinforcement learning on evaluated

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.

tasks while requiring substantially fewer rollouts (Agrawal et al., 2026). These results highlight the potential of prompt learning for adapting models to downstream requirements without updating their weights.

Existing work has explored a variety of approaches to prompt learning, but limitations in model access and adaptation scope constrain their practical applicability. First, some methods still require access to model internals. Prompt tuning learns continuous prompts through backpropagation (Lester et al., 2021), while mixture-of-prompt methods such as MEPT and PT-MoE optimize continuous prompt representations and routing components within the model's computation (Zeng et al., 2025; Li et al., 2025). However, privacy and commercial restrictions often make internal representations and gradients inaccessible—a constraint that itself motivates black-box prompt learning (Shen et al., 2023). Such requirements therefore limit the applicability of these methods to deployed black-box models. Second, black-box approaches such as GEPA and ACE optimize prompts or contexts through reflection and feedback, but primarily study adaptation within individual datasets or benchmarks (Agrawal et al., 2026; Zhang et al., 2026). In real-world deployment, incoming queries often come from heterogeneous domains, task formats, and capability mixtures. This setting exposes cross-dataset interference: even when datasets differ clearly, updates for one domain can still harm another when different tasks share an adaptation context. As Figure 1 shows, adding benchmarks causes GEPA and ACE to accumulate negative retention on previous tasks, illustrating interference within a shared adaptation context. This motivates a framework that can preserve specialization while still learning from a mixed stream of tasks.

To address these limitations, we propose EEVEE, a *multi-dataset post-deployment prompt learning* framework for LLM agents, enabling black-box models to learn prompts across heterogeneous task streams. Following prior feedback-driven prompt learning methods (Pryzant et al., 2023; Agrawal et al., 2026; Zhang et al., 2026), we focus on verifiable tasks, as LLMs remain limited in reliably identifying and correcting their own errors without external feedback (Huang et al., 2024; Kamoi et al., 2024). In this setting, a model receives a stream of examples drawn from multiple datasets and domains rather than one stationary source. Instead of forcing all inputs through one adaptation path, EEVEE introduces a *router* that partitions the stream into task clusters and assigns each cluster to a suitable prompt configuration. This preserves prompt-based adaptation while reducing destructive interference across domains. The prompt learner and router are coupled: routing determines which examples each prompt learns from, while prompt behavior determines which routing policy is useful. We therefore introduce a *router-prompt co-evolution strategy* that interleaves router and prompt learning phases, allowing routing decisions and prompt updates to improve together rather than being fixed or trained in isolation. To make this co-evolution practical, we further design a three-stage training process that initializes useful prompt slots, explores coupled updates efficiently, and then converges under a stable router.

We evaluate EEVEE on multiple datasets and show that it consistently outperforms competitive baselines for multi-dataset post-deployment prompt learning. Across the four-benchmark suite, EEVEE improves the average score by 10.38 and 24.32 points over Qwen3-4B-Instruct (Yang et al., 2025) and DeepSeek-V3.2 (DeepSeek-AI et al., 2025), and by up to 37.2% and 48.2% over GEPA and ACE. In the incremental multi-benchmark setting, EEVEE ends with a +41.53 cumulative retention gain after all tasks are introduced, while GEPA and ACE end at -15.36 and -18.58. Single-benchmark and token-cost analyses further show that the routing design remains competitive in conventional single-task settings while avoiding ACE's large prompt expansion.

2 METHODS

2.1 FRAMEWORK OVERVIEW

EEVEE targets multi-dataset post-deployment prompt learning, where a mixed stream contains different domains, formats, and evaluation rules. A single evolving prompt can let updates for one task family interfere with another, so EEVEE maintains a set of specialized prompts and a router that chooses which prompt should handle each input. We denote the prompt set by $\mathcal{P} = \{p_1, \ldots, p_K\}$ and the router by R. The target model M is fixed.

During adaptation, EEVEE uses observed correctness feedback to update its router and prompts; answering a new query does not require its ground-truth answer.

At inference time, the router first selects a slot and the target model then answers with the corresponding prompt:

$$z = R(x; \mathcal{P}) \in \{1, \ldots, K\}, \qquad \hat{y} = M(x; p_z).$$

This preserves prompt-based adaptation while allowing different inputs to invoke different specialized behaviors.

To obtain strong performance, the router itself must be learned: default or manually written routers do not reliably recover behaviorally useful partitions, as shown in Table 2. Router learning iterates the router prompt and has a mutual dependency with prompt learning: routing determines which examples each prompt sees, while prompt quality determines whether a routing decision is good. EEVEE therefore learns both through router-prompt co-evolution.

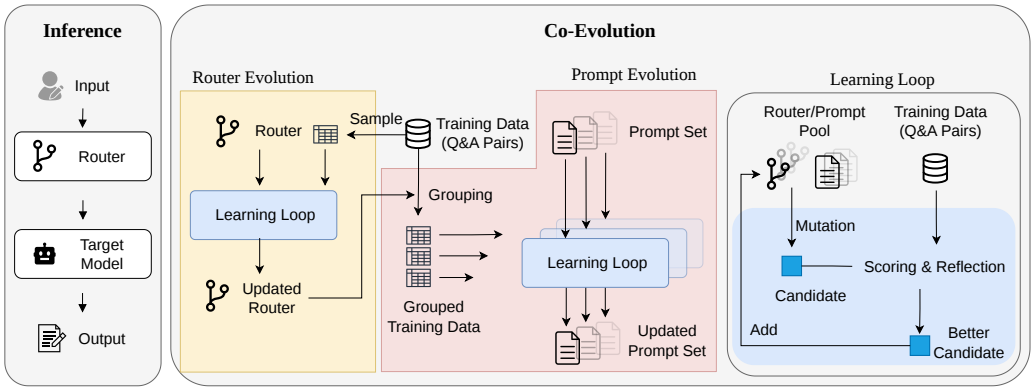


Figure 2: Main framework of EEVEE. Inference routes each input to a specialized prompt; learning co-evolves the router and prompt set through mutation, analysis, reflection, scoring, and regrouping.

2.2 ROUTER-PROMPT CO-EVOLUTION

The mixed adaptation data is split into training and validation sets, $\mathcal{D}_{\text{tr}}$ and $\mathcal{D}_{\text{val}}$. Each co-evolution cycle alternates two operations: router evolution fixes the prompt set and searches for a better router, then prompt evolution fixes this router and updates each slot prompt on its routed data. Write $\mathcal{P}_T = \{p_{k,T}\}_{k=1}^{K}$. In a cycle starting from R_T and $\mathcal{P}_T$,

$$R_{T+1} = \text{RouterEvolve}(R_T; \mathcal{P}_T, \mathcal{D}_{\text{tr}}, \mathcal{D}_{\text{val}}), \qquad \mathcal{P}_{T+1} = \mathcal{P}_T,$$
$$\mathcal{P}_{T+2} = \left\{\text{PromptEvolve}(p_{k,T+1}; R_{T+1}, \mathcal{D}_{\text{tr},k}^{T+1}, \mathcal{D}_{\text{val},k}^{T+1})\right\}_{k=1}^{K}, \quad R_{T+2} = R_{T+1}.$$

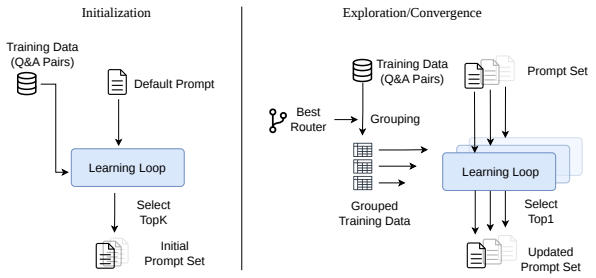
The next router phase uses the updated prompt set, so routing decisions and prompt specialization improve each other rather than being optimized in isolation.

Router evolve. Given R_T and fixed prompts $\mathcal{P}_T$, EEVEE initializes a temporary router pool $\mathcal{B}_R$ and repeatedly samples a router mini-batch $\mathcal{D}_{\text{RM}}$. To make errors attributable to assignment rather than prompt incapability, $\mathcal{D}_{\text{RM}}$ is sampled only from training examples that at least one current slot prompt can solve. Each update step samples reference routers from $\mathcal{B}_R$ and mutates them into R_{mut}, allowing the search to redesign routing rules rather than only refine an existing router.

EEVEE evaluates R_{mut} on $\mathcal{D}_{\text{RM}}$. Since router quality is observed through downstream prompt correctness, EEVEE analyzes cases where the routed slot fails but another slot succeeds, explaining why the better slot matches the task. Reflection uses these analyses and ground-truth correctness to produce R_{ref}. Let $s_{\text{mb}}(\cdot)$ denote mini-batch downstream score. EEVEE keeps

$$R^\star = \arg\max_{R \in \{R_{\text{mut}}, R_{\text{ref}}\}} s_{\text{mb}}(R),$$

evaluates $R^\star$ on $\mathcal{D}_{\text{val}}$, and admits it into $\mathcal{B}_R$ only if it improves over the phase baseline R_T. When the router score plateaus, the phase outputs R_{T+1} while keeping $\mathcal{P}_T$ fixed.



$$p_k = \arg\max_{p \in \mathcal{B} \setminus \mathcal{S}_{k-1}} \Delta(p; \mathcal{S}_{k-1}),$$
$$\Delta(p; \mathcal{S}) = \left| C_F(p) \setminus \bigcup_{q \in \mathcal{S}} C_F(q) \right|,$$
$$\mathcal{S}_k = \mathcal{S}_{k-1} \cup \{p_k\}.$$

Figure 3: Three-stage training design and initialization rule. Left: initialization builds prompts, exploration alternates router and prompt evolution, and convergence fixes the router for larger-budget prompt learning. Right: greedy coverage rule for top-K prompt selection.

Router candidates are scored by downstream accuracy, consistency, and balance. Let $a_R(x, y) = \mathbf{1}[M(x; p_{R(x;\mathcal{P}_T)}) = y]$ denote routed correctness under R and the fixed prompt set:

$$S_R(R) = \lambda_{\text{acc}} A(R) + \lambda_{\text{con}} C(R) + \lambda_{\text{bal}} B(R), \quad A(R) = \frac{1}{|\mathcal{D}_{\text{val}}|} \sum_{(x,y)\in\mathcal{D}_{\text{val}}} a_R(x, y),$$
$$C(R) = \beta_{\text{in}} \,\text{Compact}(R) + \beta_{\text{out}} \,\text{Separate}(R), \quad B(R) = \gamma_{\text{use}} \frac{|\mathcal{K}_R|}{K} + \gamma_{\text{bal}} \,\text{Balance}(\pi_R).$$

Here $\mathcal{K}_R$ is the set of labels used by R and π_R is its empirical label distribution. Compact rewards same-label examples with similar cached correctness vectors, while Separate rewards behaviorally distinguishable labels. The weights are annealed from consistency and balance toward downstream accuracy. Appendix B.2 shows that performance is stable across alternative annealing, weighting, and prompt-search settings.

Prompt evolve. After router evolution, R_{T+1} routes the full data into slot-specific groups:

$$\mathcal{D}^{T+1}_{\text{tr},k} = \{(x, y) \in \mathcal{D}_{\text{tr}} : R_{T+1}(x; \mathcal{P}_T) = k\}, \quad \mathcal{D}^{T+1}_{\text{val},k} = \{(x, y) \in \mathcal{D}_{\text{val}} : R_{T+1}(x; \mathcal{P}_T) = k\}.$$

Each non-empty slot is evolved independently and in parallel. For slot k, EEVEE initializes a temporary prompt pool $\mathcal{B}_{P,k}$ and samples $\mathcal{D}_{\text{PM}} \subset \mathcal{D}^{T+1}_{\text{tr},k}$. Prompt evolution also uses mutation and reflection, but without the router-specific analysis step: it proposes p_{mut} from reference prompts and mini-batch examples, then directly reflects from question, target answer, model answer, and correctness to obtain p_{ref}. The better mini-batch prompt

$$p^\star = \arg\max_{p \in \{p_{\text{mut}}, p_{\text{ref}}\}} s_{\text{mb}}(p)$$

is evaluated on the routed validation set with score $s^k_{\text{val}}(\cdot)$.

EEVEE stores prompts in a Pareto-front pool. Each prompt is represented by its correctness vector over $\mathcal{D}^{T+1}_{\text{val},k}$; a prompt is dominated if another is at least as correct on every example and strictly better on one. Thus the frontier preserves complementary prompts. A candidate enters $\mathcal{B}_{P,k}$ only when

$$s^k_{\text{val}}(p^\star) > s^k_{\text{val}}(p_\emptyset), \qquad p^\star \in \text{ParetoFront}(\mathcal{B}_{P,k} \cup \{p^\star\}),$$

where $p_\emptyset$ is the empty prompt. The empty-prompt floor removes ineffective edits, and the Pareto-front rule preserves diverse useful prompts. When all non-empty slots plateau, prompt evolution returns the updated prompt set.

2.3 TRAINING STAGES

EEVEE uses three stages with distinct roles: initialization creates usable prompt slots, exploration searches over coupled router-prompt designs, and convergence spends a larger budget after routing stabilizes.

Initialization. Router evolution needs a prompt set that can reveal whether a routing decision is useful; otherwise accuracy reflects prompt weakness rather than router quality. EEVEE therefore first initializes a diverse prompt set before router learning. Initialization runs prompt learning on the

mixed training set and keeps a Pareto-front pool: because frontier prompts cover complementary examples, selecting from this pool yields specialized prompts with distinguishable behavior. Let $\mathcal{B}$ be the initialization pool after dominated candidates are removed, and let F_i be the reduced Pareto frontier for validation example i. With coverage $C_F(p) = \{i : p \in F_i\}$ and $\mathcal{S}_0 = \emptyset$, EEVEE greedily retains prompts by the rule in Figure 3, where Δ denotes additional coverage. The retained prompts form $\mathcal{P}^0$ and provide specialized behaviors from which the initial router can be written.

Exploration. Exploration starts from $(R_0, \mathcal{P}^0)$ and alternates router and prompt evolution under lightweight budgets. Frequent switching is necessary for efficiency: fully optimizing prompts under an unstable router wastes budget, while optimizing a router against stale prompts can overfit to obsolete prompt behavior. As described in Section 2.2, EEVEE enables an annealing mechanism for router-candidate scoring, shifting weights from consistency and balance toward downstream accuracy so early steps preserve diverse routing behaviors and later steps favor accurate, stable routing.

Convergence. Because exploration must switch phases frequently, it cannot fully optimize every slot prompt. Once annealing and alternating updates identify a stable router, convergence fixes $R^\star$, reroutes $\mathcal{D}_{\text{tr}}$ and $\mathcal{D}_{\text{val}}$, and spends a larger prompt-learning budget within each slot. This enables EEVEE to find strong prompts under a fixed router rather than continuing to move the partition.

3 EXPERIMENTS

3.1 SETTINGS

We evaluate EEVEE in a *multi-dataset post-deployment prompt learning* setting over four benchmarks: GPQA Diamond Rein et al. (2023), Formula Wang et al. (2025), TheoremQA Chen et al. (2023), and HumanEval Chen et al. (2021). GPQA Diamond tests closed-book knowledge QA; Formula and TheoremQA emphasize mathematical and symbolic reasoning; HumanEval evaluates code generation. We choose these benchmarks to cover commonly encountered task domains, including mathematical reasoning, code generation, and knowledge-based question answering. Here, post-deployment refers to prompt learning with fixed, black-box models. Our main experiments optimize prompts on a prepared adaptation split and evaluate them on held-out examples; Appendix B.4 additionally evaluates strict online test-then-adapt learning. All prompt templates used by EEVEE, including the benchmark input templates, are listed in Appendix E. To reduce randomness, we run stochastic settings multiple times and report averaged scores.

3.2 MAIN RESULTS

Table 1 compares average performance over three runs against two strong reflection-based prompt-learning baselines, GEPA Agrawal et al. (2026) and ACE Zhang et al. (2026), plus the unadapted target model. On Qwen3-4B-Instruct Yang et al. (2025), EEVEE reaches 51.75 average score, improving over the target-model baseline by 10.38 points and outperforming GEPA and ACE by 14.02 and 16.83 points. Its gains over the baseline are +9.33 on Formula, +10.48 on TheoremQA, and +23.17 on HumanEval, where the final score reaches 72.63. On DeepSeek-V3.2 DeepSeek-AI et al. (2025), EEVEE reaches 64.07 average score, improving over the target-model baseline by 24.32 points and over GEPA by 8.24 points; the per-benchmark gains are +30.55 on Formula, +18.63 on TheoremQA, and +50.00 on HumanEval, with HumanEval reaching 92.82.

We also examine how performance changes as benchmarks are progressively added to the adaptation stream. As shown in Figure 1, EEVEE maintains positive cumulative retention, reaching +41.53 after all four benchmarks, whereas GEPA and ACE end at -15.36 and -18.58, respectively. Appendix B.2 further examines robustness to changes in prompt-slot count, optimization hyperparameters, task mixture, and task order.

3.3 ABLATIONS

To assess whether our key design choices are effective, we ablate the main components of EEVEE on Qwen3-4B-Instruct. We replace the evolved router prompt with default text (Default Router) or a GPT-5.4-written prompt (Manual Router) to examine the necessity of router evolution. Both router

Table 1: Main results on the four-benchmark suite. Scores are percentages averaged over three runs; colored subscripts denote differences from the target-model baseline.

Target model	Method	GPQA Diamond	Formula	TheoremQA	HumanEval	Avg.
Qwen3-4B Instruct	Baseline	56.00	45.22	14.79	49.46	41.37
	ACE	48.93$_{-7.07}$	39.67$_{-5.55}$	15.84$_{+1.05}$	35.23$_{-14.23}$	34.92$_{-6.45}$
	GEPA	50.84$_{-5.16}$	49.83$_{+4.61}$	19.62$_{+4.83}$	30.62$_{-18.84}$	37.73$_{-3.64}$
	EEVEE	54.55$_{-1.45}$	54.55$_{+9.33}$	25.27$_{+10.48}$	72.63$_{+23.17}$	**51.75**$_{+10.38}$
DeepSeek-V3.2	Baseline	64.98	30.00	21.21	42.82	39.75
	ACE	55.89$_{-9.09}$	37.78$_{+7.78}$	27.05$_{+5.84}$	78.59$_{+35.77}$	49.83$_{+10.08}$
	GEPA	41.75$_{-23.23}$	60.56$_{+30.56}$	31.72$_{+10.51}$	89.29$_{+46.47}$	55.83$_{+16.08}$
	EEVEE	63.08$_{-1.90}$	60.55$_{+30.55}$	39.84$_{+18.63}$	92.82$_{+50.00}$	**64.07**$_{+24.32}$

prompts are given in Appendix E.4. The No Co-evolution variant first evolves the router and then evolves the prompts in a separate stage, testing whether interleaved router–prompt optimization is necessary. Table 2 compares these variants with the full method and the unadapted baseline.

The full method reaches 51.75 average score, 8.17 points above the default router (43.58), 14.57 above the manual router (37.18), and 8.87 above no co-evolution (42.88). Default routing and no co-evolution improve the baseline by only 2.21 and 1.51 points, respectively, while the manual router is 4.19 points below the baseline. These gaps show that EEVEE needs both learned routing and interleaved router–prompt optimization; static partitions or two-stage learning do not capture the mutual dependence between routing decisions and prompt behavior. The four benchmarks are clearly distinct from one another, so routing by dataset identity is a natural alternative. However, even with 100% classification accuracy, dataset-ID routing reaches only 47.98, below EEVEE's 51.75 (Appendix B.1). Appendix B.1 further reports its combination with independently optimized GEPA and ACE prompts and router ablations on DeepSeek-V3.2.

Table 2: Ablation results for the main components of EEVEE on Qwen3-4B-Instruct.

Variant	GPQA Diamond	Formula	TheoremQA	HumanEval	Avg.
Baseline	56.00	45.22	14.79	49.46	41.37
Default Router	47.03	47.11	26.10	54.06	43.58
Manual Router	35.24	42.11	23.96	47.42	37.18
No Co-evolution	39.84	47.39	24.81	59.49	42.88
EEVEE	54.55	54.55	25.27	72.63	51.75

3.4 GENERALIZATION

To assess the generalization of EEVEE, we examine its applicability across individual datasets, the transferability of learned prompts across models, and performance on held-out benchmarks.

Single-dataset performance. To examine whether EEVEE works across a range of datasets, we compare it with existing methods under single-dataset adaptation. Since these methods primarily study individual benchmarks, we evolve prompts independently on each dataset for a matched comparison. Figure 4 includes FiNER (Loukas et al., 2022) and IFBench (Pyatkin et al., 2025) in addition to the main benchmark suite, extending the evaluation to datasets studied by prior work. EEVEE is broadly competitive with GEPA and ACE across these settings. It reaches 55.25 on Formula and 73.17 on HumanEval, outperforming both baselines on these two benchmarks, and achieves 25.27 on TheoremQA. The no-router variant also performs strongly on Formula and HumanEval, showing that our prompt learning design remains effective without router specialization.

Cross-model transfer. The main results show that EEVEE can learn prompts with different target models. Here, we further investigate whether prompts learned with one model can be transferred directly to another, avoiding the need to repeat prompt evolution for each target model. We directly

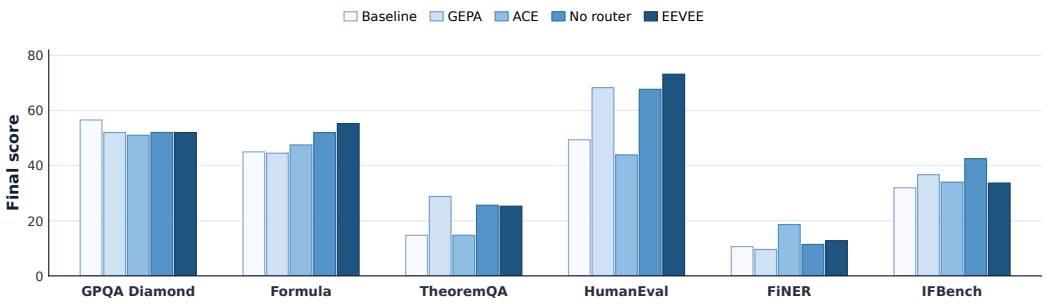


Figure 4: Single-benchmark results Loukas et al. (2022); Pyatkin et al. (2025). Bars show final scores after learning independently on each benchmark. The no-router variant keeps one retained prompt, isolating prompt learning without router specialization.

transfer the router prompt and specialized prompt set learned with Qwen3-4B-Instruct (Yang et al., 2025) to DeepSeek-V3.2 (DeepSeek-AI et al., 2025) in non-thinking mode, which replaces Qwen for both routing and answer generation. As shown in Table 3, the transferred prompts raise the DeepSeek-V3.2 average from 39.75 to 54.10, with gains of +12.28 on Formula, +11.68 on TheoremQA, and +34.22 on HumanEval. These results demonstrate that the learned prompts can be reused across the two models without additional prompt evolution on the target model.

Held-out benchmark generalization. Prompt learning can overfit the adaptation distribution and generalize poorly to unseen tasks (Fu et al., 2026). To examine this issue, we compare EEVEE, GEPA, and ACE on held-out MBPP (Austin et al., 2021) and MMLU-Pro (Wang et al., 2024b) after adaptation on the four primary benchmarks. Here, although we observe some degradation on these held-out benchmarks, EEVEE remains more stable than the other methods, with a smaller decline in average performance (Table 3). These results suggest that EEVEE's gains on the multi-benchmark suite do not come at the expense of greater performance degradation on the evaluated held-out benchmarks. Additional results on inference-time gating are provided in Appendix B.4.

Table 3: Cross-model and held-out benchmark generalization. Left: prompts learned with Qwen3-4B-Instruct are evaluated on DeepSeek-V3.2 alongside the source-model result. Right: prompts learned on the main four-benchmark suite are evaluated on MBPP and MMLU-Pro with a shared unadapted baseline. Scores are averaged over three runs.

Cross-Model Generalization

Setting	GPQA Diamond	Formula	TheoremQA	HumanEval	Avg.
DeepSeek base	64.98	30.00	21.21	42.82	39.75
Qwen3 EEVEE	54.55	54.55	25.27	72.63	51.75
Qwen3 transfer	60.27	42.28	32.89	77.04	**54.10**

Held-Out Generalization

Method	MBPP	MMLU Pro	Avg.
Baseline	69.29	70.74	70.01
EEVEE	**70.42**	68.92	69.67
GEPA	68.20	68.85	68.53
ACE	67.47	69.32	68.39

3.5 TOKEN COST

Because EEVEE adds a router before answer generation, a natural question is whether this additional step introduces substantial token overhead during evaluation and inference. To assess this overhead, we measure the average input and output token usage per test example across the four benchmarks. Averaged over the four benchmarks in Figure 5, EEVEE uses 4.32k total tokens per example, close to GEPA's 3.47k and far below ACE's 21.30k. The input-token gap is similar: EEVEE uses 3.00k input tokens on average, compared with 2.44k for GEPA and 20.35k for ACE. ACE incrementally edits playbook bullets, which can accumulate redundant entries and lengthen prompts as tasks and data grow. EEVEE's router therefore adds only modest overhead while using about 4.9× fewer total tokens than ACE. We focus on per-query inference cost because learned prompts are reused across subsequent queries, allowing the one-off adaptation cost to be amortized over continued use. Appendix B.5 also reports adaptation cost to show EEVEE is competitive in adaptation time.

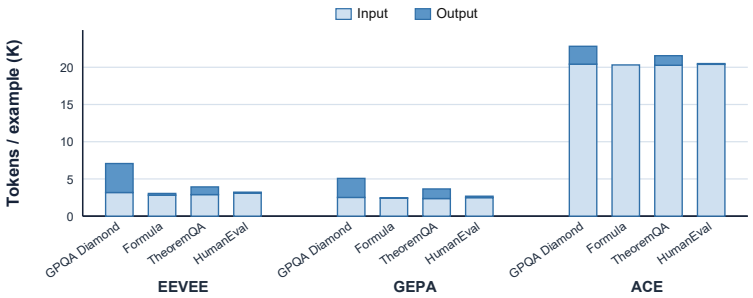


Figure 5: Average token usage per test example with Qwen3-4B-Instruct. Each method group contains four benchmark bars, each stacked into input and output tokens.

3.6 Case Study: What Does Prompt Learning Capture?

We compare the empty prompt with the final learned router and prompt set across six completed runs, three per target model. Table 4 suggests a task-property pattern: code and formula tasks benefit when feedback can be translated into reusable procedures for solving problems and presenting answers. HumanEval, Formula, and TheoremQA improve in all six runs, although TheoremQA shows more mixed flips. GPQA Diamond is the only benchmark with more regressions than recoveries, suggesting a limitation on knowledge-intensive tasks.

Table 4: Six-run empty-vs-final retest over three Qwen3-4B-Instruct and three DeepSeek-V3.2 runs. Δ is average final-minus-empty change; W$\rightarrow$R and R$\rightarrow$W count per-example flips in that order; Runs+ counts positive-Δ runs.

Type	Benchmark	Qwen Δ	DeepSeek Δ	W$\rightarrow$R; R$\rightarrow$W	Runs+
Code	HumanEval	+23.2	+48.8	193; 16	6 of 6
Formula	Formula	+9.5	+31.7	268; 21	6 of 6
Math QA	TheoremQA	+10.0	+18.2	632; 166	6 of 6
Domain QA	GPQA Diamond	-3.7	-7.7	55; 89	1 of 6
Overall	Mixed suite	+9.6	+21.1	1148; 292	6 of 6

Figure 6 illustrates what these learned procedures capture and where they fall short. For executable programs and formula-grounded computation, feedback teaches reusable rules for preserving interfaces, handling edge cases and units, and producing parseable answers. For knowledge-intensive QA, however, stronger generic reasoning does not necessarily supply missing domain knowledge. In the GPQA Diamond case, the learned response explicitly applies density formulas but assumes constant density for planets of the same composition, underweighting gravitational compression. These cases suggest that prompt learning is particularly effective when feedback yields reusable procedures, while more elaborate reasoning alone may not resolve knowledge-dependent errors. Appendix D provides representative learned prompts and output excerpts.

4 Related Work

Prompt learning. Prompt learning adapts language models while keeping their weights fixed. Gradient-based approaches optimize discrete triggers, soft prompts, or prefixes (Shin et al., 2020; Li & Liang, 2021; Lester et al., 2021), with MEPT and PT-MoE extending this direction through multiple prompt experts and routing (Zeng et al., 2025; Li et al., 2025). However, their reliance on gradients or internal representations limits their applicability to deployed black-box models. Black-box approaches instead optimize prompts or contexts through scores and feedback (Zhou et al., 2022; Yang et al., 2024; Pryzant et al., 2023; Guo et al., 2024; Fernando et al., 2023; Khattab et al., 2023; Opsahl-Ong et al., 2024; Yuksekgonul et al., 2024), including reflective prompt evolution in GEPA (Agrawal et al., 2026), playbook adaptation in ACE (Zhang et al., 2026), and parallel

Formula: unit scale

Task: Compute free cash flow from operating cash flow and capital expenditure.

Baseline: Reverses subtraction and keeps the million-scale decimal, producing a negative value.

Learned: Applies the dollar-scale formula and emits 400000.00.

HumanEval: executable body

Task: Sum even values at odd indices.

Baseline: Writes a bare expression without the required indented return.

Learned: Produces an executable body with an accumulator and return.

GPQA Diamond: knowledge underweighted

Task: Select the densest Earth-like exoplanet from mass and composition cues.

Baseline: Uses the rocky-planet prior that higher mass increases self-compression and density.

Learned: Uses formulaic density reasoning, treats same composition as constant density, and selects Earth.

Figure 6: Case-study flips. Columns summarize behavior; excerpts are in Appendix D.

trace aggregation in Combee (Li et al., 2026). Although these methods enable adaptation without model-internal access, they primarily study individual datasets or benchmarks. MoP (Wang et al., 2024a) also routes queries among expert prompts, but it targets a single task with a partition fixed by input similarity, which in our mixed-stream setting falls short of learning the router jointly with the prompts (Table 2). Cross-dataset memory-extraction work (Yang et al., 2026) extends adaptation across datasets, but remains focused on memory extraction rather than heterogeneous tasks with different capability requirements. EEVEE addresses this broader setting by jointly learning routing and specialized prompts across heterogeneous task streams.

Agent adaptation. Agent adaptation uses feedback and accumulated experience to improve behavior. Self-Refine and Reflexion establish feedback loops through iterative revision and verbal reflection (Madaan et al., 2023; Shinn et al., 2023). To make experience useful beyond an immediate revision, generative agents organize observations into memories and reflections (Park et al., 2023), while ExpeL extracts reusable insights from successes and failures for later task solving (Zhao et al., 2024a). To maintain this knowledge as experience accumulates, A-MEM links and updates memories (Xu et al., 2025). Experience can also be retained as executable skills rather than natural-language memory. Voyager builds a reusable skill library through environmental feedback (Wang et al., 2023), and LearnAct learns and refines actions from task failures (Zhao et al., 2024b). At the program level, evolutionary discovery agents similarly use evaluation feedback to guide successive code revisions for scientific and algorithmic search (Novikov et al., 2025; Sharma, 2025; Xu et al., 2026). These approaches retain feedback in memories, skills, or programs. EEVEE focuses on organizing adaptation across heterogeneous task streams, jointly learning routing and specialized prompts to decouple multi-task commonality from task-specific behavior.

5 CONCLUSION

We introduced EEVEE, a multi-dataset post-deployment prompt learning framework for LLM agents facing heterogeneous task streams. To reduce cross-dataset interference, EEVEE maintains router-conditioned prompts, so inputs can be assigned to prompts specialized for compatible task behavior. To address the mutual dependence between routing and prompts, EEVEE uses three-stage co-evolution to initialize prompts, explore coupled updates, and refine prompts under a stable router. Experiments show strong gains over prompt-learning baselines in mixed-dataset settings, with benefits increasing across tasks and reasonable held-out and cross-model transfer. Case studies suggest prompt learning works best when feedback becomes reusable procedures, output contracts, or task-solving strategies. Given the limitations of LLMs' self-evaluation and reflection, we currently focus on verifiable tasks, while also examining performance under sparse and noisy feedback in Appendix B.3. As models become more capable, we hope to extend this framework toward broader online adaptation beyond prepared adaptation splits, building on our initial test-then-adapt experiments in Appendix B.4, and ultimately toward prompt learning without external feedback.

Reproducibility statement

The paper is empirical and does not present formal theoretical results or proofs. Reproducing the main claims requires reproducing the mixed-benchmark adaptation protocol, model endpoints or comparable served checkpoints, benchmark splits, router–prompt evolution settings, and final-test evaluation described in Appendix A; the complete prompt templates are listed in Appendix E. Because router and prompt evolution are stochastic, exact routers and prompt texts may vary across runs even under the same settings; Table 12 reports this run-to-run variation for the main average score.

AI use statement

LLMs are core components of the method: they act as the target model being adapted, the router, and the prompt mutation, prompt reflection, router mutation, router analysis, and router reflection operators. In the ablation study, GPT-5.4 was used once to write a fixed manual router, which was then held constant for evaluation. For research and writing assistance, we used LLMs for literature searches, drafting and polishing portions of the manuscript, document formatting, and figure creation. All resulting outputs were checked.

Ethics statement

The experiments use public benchmarks and model calls. We do not collect new human-subject data, run crowdsourcing studies, or introduce a dataset containing personal information.

EEVEE can make heterogeneous-task prompt adaptation more efficient by reducing the need to maintain one prompt-learning run per task family. The main practical risk is that noisy, incomplete, or distribution-shifted feedback can reinforce incorrect heuristics. Adapted prompts should be validated on held-out data before use, and benchmark gains should not be interpreted as deployment reliability guarantees.

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. In *International Conference on Learning Representations*, 2026.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Wenhu Chen, Ming Yin, Max Ku, Pan Lu, Yixin Wan, Xueguang Ma, Jianyu Xu, Xinyi Wang, and Tony Xia. TheoremQA: A theorem-driven question answering dataset. *arXiv preprint arXiv:2305.12524*, 2023.

DeepSeek-AI, Aixin Liu, Aoxue Mei, Bangcai Lin, Bing Xue, Bingxuan Wang, Bingzheng Xu, Bochao Wu, Bowei Zhang, Chaofan Lin, et al. DeepSeek-V3.2: Pushing the frontier of open large language models. *arXiv preprint arXiv:2512.02556*, 2025.

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

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

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

Jie Huang, Xinyun Chen, Swaroop Mishra, Huaixiu Steven Zheng, Adams Wei Yu, Xinying Song, and Denny Zhou. Large language models cannot self-correct reasoning yet. In *International Conference on Learning Representations*, 2024. URL https://arxiv.org/abs/2310.01798.

Ryo Kamoi, Yusen Zhang, Nan Zhang, Jiawei Han, and Rui Zhang. When can LLMs actually correct their own mistakes? a critical survey of self-correction of LLMs. *Transactions of the Association for Computational Linguistics*, 12:1417–1440, 2024. URL https://arxiv.org/abs/2406.01297.

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. *arXiv preprint arXiv:2310.03714*, 2023.

Brian Lester, Rami Al-Rfou, and Noah Constant. The power of scale for parameter-efficient prompt tuning. In *Proceedings of EMNLP*, 2021.

Hanchen Li, Runyuan He, Qizheng Zhang, Changxiu Ji, Qiuyang Mang, Xiaokun Chen, Lakshya A. Agrawal, Wei-Liang Liao, Eric Yang, Alvin Cheung, James Zou, Kunle Olukotun, Ion Stoica, and Joseph E. Gonzalez. Combee: Scaling prompt learning for self-improving language model agents. *arXiv preprint arXiv:2604.04247*, 2026.

Xiang Lisa Li and Percy Liang. Prefix-tuning: Optimizing continuous prompts for generation. In *Proceedings of ACL-IJCNLP*, 2021.

Zongqian Li, Yixuan Su, and Nigel Collier. PT-MoE: An efficient finetuning framework for integrating mixture-of-experts into prompt tuning. In *Advances in Neural Information Processing Systems*, 2025.

Lefteris Loukas, Manos Fergadiotis, Ilias Chalkidis, Eirini Spyropoulou, Prodromos Malakasiotis, Ion Androutsopoulos, and Georgios Paliouras. FiNER: Financial numeric entity recognition for XBRL tagging. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics*, 2022.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. Self-refine: Iterative refinement with self-feedback. In *Advances in Neural Information Processing Systems*, 2023.

Alexander Novikov, Ngan Vu, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco J. R. Ruiz, Abbas Mehrabian, M. Pawan Kumar, Abigail See, Swarat Chaudhuri, George Holland, Alex Davies, Sebastian Nowozin, Pushmeet Kohli, and Matej Balog. AlphaEvolve: A coding agent for scientific and algorithmic discovery. *arXiv preprint arXiv:2506.13131*, 2025.

Krista Opsahl-Ong, Michael J. Ryan, Josh Purtell, David Broman, Christopher Potts, Matei Zaharia, and Omar Khattab. Optimizing instructions and demonstrations for multi-stage language model programs. In *Proceedings of EMNLP*, 2024.

Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul Christiano, Jan Leike, and Ryan Lowe. Training language models to follow instructions with human feedback. *arXiv preprint arXiv:2203.02155*, 2022.

Joon Sung Park, Joseph C. O'Brien, Carrie J. Cai, Meredith Ringel Morris, Percy Liang, and Michael S. Bernstein. Generative agents: Interactive simulacra of human behavior. In *Proceedings of UIST*, 2023.

Reid Pryzant, Dan Iter, Jerry Li, Yin Tat Lee, Chenguang Zhu, and Michael Zeng. Automatic prompt optimization with "gradient descent" and beam search. In *Proceedings of EMNLP*, 2023.

Valentina Pyatkin, Saumya Malik, Victoria Graf, Hamish Ivison, Shengyi Huang, Pradeep Dasigi, Nathan Lambert, and Hannaneh Hajishirzi. Generalizing verifiable instruction following. In *Advances in Neural Information Processing Systems*, 2025.

David Rein, Betty Li Hou, Asa Cooper Stickland, Jackson Petty, Richard Yuanzhe Pang, Julien Dirani, Julian Michael, and Samuel R. Bowman. GPQA: A graduate-level google-proof q&a benchmark. *arXiv preprint arXiv:2311.12022*, 2023.

Asankhaya Sharma. OpenEvolve: An open-source evolutionary coding agent, 2025. URL https://github.com/algorithmicsuperintelligence/openevolve.

Maohao Shen, Soumya Ghosh, Prasanna Sattigeri, Subhro Das, Yuheng Bu, and Gregory Wornell. Reliable gradient-free and likelihood-free prompt tuning. In *Findings of the Association for Computational Linguistics: EACL 2023*, pp. 2416–2429, 2023. doi: 10.18653/v1/2023.findings-eacl.183.

Taylor Shin, Yasaman Razeghi, Robert L. Logan IV, Eric Wallace, and Sameer Singh. AutoPrompt: Eliciting knowledge from language models with automatically generated prompts. In *Proceedings of EMNLP*, 2020.

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

Dannong Wang, Jaisal Patel, Daochen Zha, Steve Y. Yang, and Xiao-Yang Liu. FinLoRA: Benchmarking LoRA methods for fine-tuning LLMs on financial datasets. *arXiv preprint arXiv:2505.19819*, 2025.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models. *arXiv preprint arXiv:2305.16291*, 2023.

Ruochen Wang, Sohyun An, Minhao Cheng, Tianyi Zhou, Sung Ju Hwang, and Cho-Jui Hsieh. One prompt is not enough: Automated construction of a mixture-of-expert prompts. In *International Conference on Machine Learning*, 2024a.

Yubo Wang, Xueguang Ma, Ge Zhang, Yuansheng Ni, Abhranil Chandra, Shiguang Guo, Weiming Ren, Aaran Arulraj, Xuan He, Ziyan Jiang, et al. MMLU-Pro: A more robust and challenging multi-task language understanding benchmark. *arXiv preprint arXiv:2406.01574*, 2024b.

Weixian Xu, Tiantian Mi, Yixiu Liu, Yang Nan, Zhimeng Zhou, Lyumanshan Ye, Lin Zhang, Yu Qiao, and Pengfei Liu. ASI-Evolve: Ai accelerates ai. *arXiv preprint arXiv:2603.29640*, 2026.

Wujiang Xu, Zujie Liang, Kai Mei, Hang Gao, Juntao Tan, and Yongfeng Zhang. A-MEM: Agentic memory for llm agents. In *Advances in Neural Information Processing Systems*, 2025. URL https://arxiv.org/abs/2502.12110.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

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

Yuqing Yang, Tengxiao Liu, Wang Bill Zhu, Taiwei Shi, Linxin Song, and Robin Jia. Self-evolving llm memory extraction across heterogeneous tasks. *arXiv preprint arXiv:2604.11610*, 2026.

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

Runjia Zeng, Guangyan Sun, Qifan Wang, Tong Geng, Sohail Dianat, Xiaotian Han, Raghuveer Rao, Xueling Zhang, Cheng Han, Lifu Huang, and Dongfang Liu. MEPT: Mixture of expert prompt tuning as a manifold mapper. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, 2025.

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. In *International Conference on Learning Representations*, 2026.

Andrew Zhao, Daniel Huang, Quentin Xu, Matthieu Lin, Yong-Jin Liu, and Gao Huang. ExpeL: Llm agents are experiential learners. In *Proceedings of the AAAI Conference on Artificial Intelligence*, 2024a. URL https://arxiv.org/abs/2308.10144.

Haiteng Zhao, Chang Ma, Guoyin Wang, Jing Su, Lingpeng Kong, Jingjing Xu, Zhi-Hong Deng, and Hongxia Yang. Empowering large language model agents through action learning. In *Conference on Language Modeling*, 2024b. URL https://arxiv.org/abs/2402.15809.

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

LIMITATIONS

Although EEVEE improves multi-dataset prompt learning, limitations remain. Like other LLM-based evolutionary procedures, it cannot guarantee exact performance reproduction across runs, since stochastic search can produce different routers and prompt sets. Its feedback loop still relies on ground-truth or rule-based labels to accumulate task knowledge; Appendix B.3 shows it degrades gracefully when that signal is sparse or noisy, but a deployment with no evaluative signal at all cannot drive updates. It does not require a prepared adaptation set: Appendix B.4 evaluates a strict test-then-adapt stream, under a different evaluation protocol from the prepared-split experiments. Our mixed-stream experiments combine four benchmarks chosen to cover the main capability types a deployed model encounters, and because each benchmark already contains problems of many types, these four provide sufficient diversity without adding further benchmarks. Moreover, each added family brings its own feedback to learn from, so the adaptation budget naturally grows with the number of task families. Figure 1 shows that EEVEE maintains a positive cumulative gain as tasks are added from one to four, and we leave substantially larger mixtures to future work. A practical risk is distribution mismatch: if adaptation data is noisy or misaligned with the real application, learned prompts may generalize weakly or even degrade performance. Deployment should check data quality and distribution similarity before updates.

A REPRODUCIBILITY AND EXPERIMENTAL DETAILS

This section records the core settings used for the main four-benchmark experiments in Table 1. The benchmark suite consists of GPQA Diamond Rein et al. (2023), Formula Wang et al. (2025), TheoremQA Chen et al. (2023), and HumanEval Chen et al. (2021). The held-out generalization study uses MBPP Austin et al. (2021) and MMLU-Pro Wang et al. (2024b) after prompt learning on the four primary benchmarks. The single-benchmark diagnostic also includes FiNER Loukas et al. (2022) and IFBench Pyatkin et al. (2025). For each benchmark, we cap the benchmark size at 500 examples, split examples into train and test partitions with a $0.5/0.5$ split, reserve half of the training partition as validation data, exclude train and validation examples from the final test set, and use data seed 42.

The main target models are Qwen3-4B-Instruct Yang et al. (2025) and DeepSeek-V3.2 DeepSeek-AI et al. (2025) in non-thinking mode. For the Qwen3-4B-Instruct runs, all model roles use the same Qwen endpoint with temperature 0.7, top-p 0.8, and a maximum generation length of 16,384 tokens. For the DeepSeek-V3.2 non-thinking runs, the endpoint uses temperature 1.0, top-p 0.95, maximum generation length 8,192 tokens, and disabled thinking. The main benchmark configurations use rule-based benchmark judges when available.

Evolution settings. Each main run retains four bootstrap prompts. Bootstrap prompt evolution uses a budget of 10 candidate steps. The router–prompt co-evolution phase uses a total mini-step budget of 150, router and prompt windows of 3, and phase-switch threshold 0.005. The router score during evolution combines downstream score, routing consistency, and balance with weights $0.6/0.2/0.2$; the final router selection uses downstream score only, with weights $1.0/0.0/0.0$. Prompt evolution uses a final per-label budget of 60, minibatch size 5, and 4 parallel prompt slots.

Execution settings and resources. The Qwen3-4B-Instruct main runs use 50 concurrent request workers for evaluation, router calls, and final testing. The DeepSeek-V3.2 non-thinking main runs use 10 concurrent request workers for the same roles. All reported final tests use three repeats, a maximum prompt length of 2,000 tokens, at most 3 retries, and at most 5 logged examples for diagnostic traces. The experiments call hosted or locally served models and do not train or fine-tune local model weights; accordingly, no local GPU training resource is required. Wall-clock time is backend- and rate-limit-dependent, so the paper reports token usage instead. In Figure 5, EEVEE uses 4.32k total tokens per test example on average, compared with 3.47k for GEPA and 21.30k for ACE.

B ADDITIONAL ANALYSES

This appendix reports studies that extend the main experiments. Unless noted otherwise, the target model is Qwen3-4B-Instruct and the protocol matches Appendix A.

B.1 ADDITIONAL ABLATION STUDIES

Equal-Budget Routing Controls. Because the four benchmarks have clear domain boundaries, a simple dataset-identity router might already recover most of the benefit that Table 2 attributes to EEVEE. We therefore add four external controls under a matched budget, so that every routed variant evolves the same total amount of prompt text and the rows differ only in how inputs are assigned to slots: a mixed single prompt, random routing, a supervised dataset-ID classifier trained on adaptation-train labels, and an oracle router handed the true source dataset.

Table 5: Equal-budget routing controls on Qwen3-4B-Instruct, sorted by average score.

Routing design	GPQA Diamond	Formula	TheoremQA	HumanEval	Avg.
No adaptation	56.00	45.22	14.79	49.46	41.37
Random router	28.62	44.17	25.39	43.90	35.52
Mixed single prompt	35.69	55.33	28.67	48.78	42.12
Oracle dataset ID	53.20	42.33	23.58	69.51	47.15
Learned classifier	53.87	40.67	**29.88**	67.48	47.98
EEVEE	**54.55**	**54.55**	25.27	**72.63**	**51.75**

Dataset identity is easy to recover: the classifier reaches **100% dataset-ID accuracy**. Yet it averages 47.98 and the oracle 47.15, both below EEVEE's 51.75, and EEVEE exceeds the oracle on all four benchmarks. Random routing collapses to 35.52, below no adaptation at all, so added prompt capacity explains nothing on its own. Neither straightforward dataset separation nor extra prompt text accounts for the gains; the dataset-ID oracle is a source-identification control, not an upper bound on prompt optimization.

Combining dataset classification with single-dataset prompt learning. A natural alternative is to combine the dataset-ID classifier with prompts optimized independently by GEPA or ACE on each dataset, under a comparable overall evaluation budget. Combining the single-dataset results in Figure 4 with the classifier's 100% dataset-ID accuracy, we derive the performance of this composition on the same test examples: each example is assigned to its corresponding dataset-specific prompt. The resulting average scores are 48.41 for GEPA and 39.29 for ACE, compared with 51.75 for EEVEE (Table 6). EEVEE outperforms both combinations on three of the four benchmarks, with single-dataset GEPA performing better only on TheoremQA. Moreover, this alternative requires dataset-identity

labels to train the classifier and organize separate prompt-learning runs—additional supervision that is not assumed in our heterogeneous-stream setting.

Table 6: Derived performance of dataset-ID routing combined with independently optimized prompts on Qwen3-4B-Instruct.

Method	GPQA Diamond	Formula	TheoremQA	HumanEval	Avg.
No adaptation	56.00	45.22	14.79	49.46	41.37
Classifier + single-dataset ACE	51.02	47.50	14.73	43.90	39.29
Classifier + single-dataset GEPA	52.03	44.50	**28.82**	68.30	48.41
EEVEE (mixed stream)	**54.55**	**54.55**	25.27	**72.63**	**51.75**

Composed scores reuse the existing single-dataset evaluations rather than additional joint runs; these use two optimization runs with one test evaluation each, whereas EEVEE follows the main-results protocol.

Learned routing partition. To examine how the learned router assigns inputs across prompt slots, Table 7 reports the percentage of test examples from each benchmark routed to each slot in one representative run, whose partition is consistent with the other two runs. In those runs, a small fraction of items from other benchmarks is also redistributed, but we present this run because only TheoremQA is split, which makes the partition easier to analyze. Table 7 shows why beating the oracle is possible: the router groups inputs by the capabilities and output behaviors they share rather than by dataset, so each slot sees a more coherent set of examples and its prompt captures their common structure better than a dataset-aligned assignment would. TheoremQA, which mixes formula computation, scientific multiple choice, and mathematical derivation, is therefore spread across slots, and its items improve in every slot. In the representative run, compared with the empty prompt on the same items, accuracy rises from 7.6% to 25.1% in the formula slot, 22.3% to 35.4% in the math slot, 39.6% to 44.3% in the multiple-choice slot, and 2.9% to 26.5% in the code slot. Sharing slots with TheoremQA does not hurt the other benchmarks either: each stays within 0.7 points of its single-benchmark score, and GPQA Diamond is 2.5 points higher (Figure 4).

Table 7: The partition the router actually learns: percentage of each benchmark's test items sent to each slot in one representative run. Slot labels summarize the behavior its learned prompt encodes; dashes are zero.

Learned slot behavior	GPQA Diamond	Formula	TheoremQA	HumanEval
Formula use, numeric output format	–	**100.0**	**62.4**	–
Scientific multiple choice	**100.0**	–	11.6	–
Math/combinatorics, "Solution:" contract	–	–	23.9	–
Function completion and execution	–	–	2.1	**100.0**

Cross-Model Router Ablations. Table 2 covers Qwen3-4B-Instruct only. We repeat the router ablations on DeepSeek-V3.2 under matched budgets; the DeepSeek router variants use three evaluations per test example, while the baseline and full EEVEE rows reproduce the averages from Table 1.

The stronger model substantially improves fixed routing: the manual router rises from 37.18 on Qwen3-4B-Instruct to 60.93 on DeepSeek-V3.2. Full EEVEE nevertheless remains best at 64.07, so better intrinsic routing ability narrows but does not remove the benefit of router–prompt co-evolution.

B.2 Robustness Analyses

Slot-Count Sensitivity. We vary only the number of prompt slots K, holding the model, data, budget, and evaluation fixed. Routing is vacuous at $K = 1$, so that setting is the mixed single-prompt control from Table 5, which spends the entire budget on prompt evolution.

These results highlight both the importance of evolving multiple specialized prompts and the robustness of the approach to moderate changes in slot count. With only one or two prompt slots,

Table 8: Router ablations on both target models.

Model	Variant	GPQA Diamond	Formula	TheoremQA	HumanEval	Avg.
Qwen3-4B-Instruct	Baseline	56.00	45.22	14.79	49.46	41.37
	Default Router	47.03	47.11	26.10	54.06	43.58
	Manual Router	35.24	42.11	23.96	47.42	37.18
	EEVEE	54.55	54.55	25.27	72.63	**51.75**
DeepSeek-V3.2	Baseline	64.98	30.00	21.21	42.82	39.75
	Default Router	44.78	57.83	32.85	90.24	56.43
	Manual Router	58.59	56.17	39.94	89.02	60.93
	EEVEE	63.08	60.55	39.84	92.82	**64.07**

Table 9: Four-benchmark average as a function of the number of prompt slots.

Slots	$K = 1$	$K = 2$	$K = 4$	$K = 6$
Avg.	42.12	42.04	**51.75**	48.74

performance drops substantially, limiting the gains from prompt evolution. In contrast, $K = 4$ and $K = 6$ achieve comparable performance, both substantially outperforming the one- and two-slot settings.

Mixture Composition and Task Order. To test sensitivity to how the adaptation stream is composed, we hold the test set fixed and vary only the task composition of a 200-example adaptation set, split 100/100 into internal train and validation. This adaptation set is smaller than the one used for the main runs, so absolute scores sit below the 51.75 of Table 1; only the comparison across mixtures is meaningful.

Table 10: Four-benchmark average under different adaptation-set mixtures, given as GPQA Diamond/TheoremQA/Formula/HumanEval example counts.

Mixture	50/50/50/50	80/40/40/40	40/80/40/40	40/40/80/40	40/40/40/80
Avg.	48.59	40.79	48.18	**50.71**	50.53

Performance is stable across most mixtures. The GPQA Diamond-heavy mixture is the clear outlier, consistent with the knowledge-heavy limitation discussed in Section 3.6: over-weighting the one task where learned prompts help least also degrades the rest. For task order, shuffled and blocked streams over identical examples obtain 49.05 and 50.19. Because EEVEE canonicalizes eligible example IDs before seeded minibatch sampling, stored order does not systematically change which data drive optimization, and the similar scores confirm this.

Hyperparameter Robustness. We test whether EEVEE is sensitive to the specific router-score and prompt-search hyperparameters used in the main experiments. For each hyperparameter configuration, we run three independent trials, first average the three trials within that configuration, and then compare the resulting eight configuration-level means. The configurations vary the annealing target for router scores, the consistency and balance weights, and the final prompt-search budget, minibatch size, and temporary prompt-pool size.

The eight configuration-level averages range from 45.05 to 50.97, a span of 5.92 points, with a sample standard deviation of 1.73 points across configurations. Thus the aggregate result is stable under the tested hyperparameter perturbations. Individual benchmarks can move more than the average, but no configuration collapse is observed, and every configuration improves over its corresponding initial-empty baseline in macro average.

Main-Result Variation Across Runs. Table 12 reports the mean and sample standard deviation of the main average score over the three independent runs used in Table 1. The average score is

Table 11: Hyperparameter robustness on Qwen3-4B-Instruct. Scores are percentages; each row is the mean over three independent runs for one configuration. Avg. is the macro average over the four benchmarks.

Configuration	GPQA Diamond	Formula	TheoremQA	HumanEval	Avg.
No-decay annealing	49.94	52.17	25.66	58.67	46.61
Soft-landing annealing	45.01	57.44	21.33	69.24	48.26
Strong-regularizer annealing	45.34	46.61	24.57	63.69	45.05
Classic reference	44.78	60.06	26.20	57.99	47.26
Prompt search tighter	47.92	53.33	27.41	63.01	47.92
Prompt search wider	48.15	62.39	23.94	61.38	48.96
Consistency-heavy weights	51.74	50.33	26.79	63.96	48.20
Balance-heavy weights	49.16	54.89	26.93	72.90	50.97

stable for EEVEE: the standard deviation is 1.62 points on Qwen3-4B-Instruct and 1.08 points on DeepSeek-V3.2.

Table 12: Mean and sample standard deviation of the main average score over three runs.

Target model	Method	Average score
Qwen3-4B-Instruct	Baseline	41.37 ± 0.51
	ACE	34.92 ± 4.14
	GEPA	37.73 ± 1.73
	EEVEE	$\mathbf{51.75 \pm 1.62}$
DeepSeek-V3.2	Baseline	39.75 ± 1.52
	ACE	49.83 ± 2.07
	GEPA	55.83 ± 4.48
	EEVEE	$\mathbf{64.07 \pm 1.08}$

Adaptation Cost Scaling. We measure how the adaptation budget in Figure 5 grows along three axes. Expanding the adaptation pool from 200 to 681 examples ($3.41\times$) raises model calls from 34,072 to 107,233 ($3.15\times$) and tokens from 48.33M to 155.31M ($3.21\times$), so cost is approximately linear in example count. Across 200-example mixtures, totals range from 37.90M to 59.46M tokens, so sequence length and task composition move the constant factor but not the trend. Increasing slots from one to six raises tokens by only 10.7% (140.78M to 155.84M), because global budgets are shared across slots and validation examples are partitioned among them rather than duplicated. With examples and slots fixed, task diversity introduces no separate per-task loop, so if each additional task contributes a fixed number of examples the dominant cost again grows approximately linearly with task count.

Compute. All experiments call hosted or locally served models and train no local weights. The additional studies reported in this appendix consumed approximately 1.1K H200 GPU-hours in total.

B.3 Adaptation with Imperfect Feedback

Given the limitations of current language models in reliable self-evaluation and reflection (Huang et al., 2024; Kamoi et al., 2024), we focus on prompt learning with verifiable feedback. In practice, however, feedback may be incomplete or inaccurate. As a step toward prompt learning without external feedback, we investigate how EEVEE performs when correctness feedback is sparse or noisy.

Sparse-p provides correctness feedback for only $p\%$ of adaptation examples; the remainder yield no evolution signal. *Noisy-p* flips the binary correctness signal for $p\%$ of adaptation examples, leaving everything else unchanged. Each resulting system is evaluated on the same held-out questions from the four benchmarks, averaged over three repeats, and reported as the fraction of clean-feedback performance retained.

Table 13: Clean-feedback performance retained under degraded feedback.

Feedback condition	Sparse-50	Sparse-25	Noisy-10	Noisy-20
Retained performance	88.4%	89.7%	89.9%	85.7%

EEVEE continues to improve over the unadapted baseline under both sparse and noisy feedback. Reducing feedback availability from 50% to 25% does not lead to a further performance drop in these experiments. In contrast, increasing feedback corruption from 10% to 20% leads to a larger decline. Although noisy feedback still permits positive adaptation gains, these gains are substantially smaller than those obtained with clean feedback. These results suggest that, within the evaluated settings, feedback accuracy matters more than feedback availability, motivating further work toward learning with less external supervision. Delayed feedback can be folded in when it arrives, and in the absence of any reliable signal EEVEE keeps its current router and prompts rather than updating without evidence.

B.4 Online Adaptation and Inference-Time Gating

Online Test-then-Adapt Evaluation. This experiment removes the pre-built adaptation split entirely. All examples form one fixed stream; each is scored the first time it is encountered and can influence only subsequent updates, so no external adaptation/test split exists and no future example is ever visible. *Cumulative* updates from every example observed so far, whereas *new-window* updates from only the most recent window; both retain the previous router and prompts rather than restarting from scratch.

Table 14: Strict prequential test-then-adapt evaluation on Qwen3-4B-Instruct. Scores are averaged over first encounters across the full stream, whereas Table 1 evaluates a held-out test partition after adaptation. The two tables therefore use different evaluation sets and adaptation histories, so their absolute scores are not directly comparable.

Method	GPQA Diamond	Formula	TheoremQA	HumanEval	Avg.
ACE online	29.12	42.23	16.92	28.86	29.28
EEVEE cumulative	**51.01**	50.67	**24.71**	47.97	43.59
EEVEE new-window	50.00	**52.95**	23.44	**49.49**	**43.97**

Both variants beat ACE on all four benchmarks, improving the average by 14.31 and 14.69 points.

Inference-Time Gating for Knowledge-Heavy Tasks. Learned prompts can reduce performance on some held-out tasks. To mitigate this issue, we explore two simple inference-time gates that determine whether to apply the learned prompt or fall back to the empty prompt. *Selection 2-of-3* applies an evolved prompt only when it receives at least two of three router votes, and otherwise falls back to the empty prompt. The *harm-aware* gate is a TF-IDF logistic regression that predicts whether the selected prompt is likely to hurt the input, using only the question and prompt text and no dataset identity. We evaluate on MMLU-Pro to measure knowledge protection and on MBPP to check that beneficial prompts are not rejected wholesale. Both gates use the completed routers and prompts from three independent EEVEE runs.

Table 15: Inference-time gating on held-out tasks. MMLU-Pro rows are a matched first pass, so they differ slightly from the three-repeat averages in Table 3.

Held-out task	Empty	EEVEE	Harm-aware	Selection 2-of-3
MMLU-Pro, matched first pass	70.65	68.81	69.48	**70.24**
MBPP, three-repeat average	69.29	**70.42**	70.33	70.09

On MMLU-Pro the harm-aware gate and Selection recover 0.67 and 1.43 points, leaving Selection only 0.41 below the empty prompt. On MBPP, where evolved prompts help, the gates retain 70.33 and

70.09 against the ungated 70.42. Although gating partially mitigates the degradation on MMLU-Pro, it does not fully recover the unadapted baseline and slightly reduces performance on MBPP, leaving the underlying generalization issue unresolved.

B.5 Adaptation Cost

Figure 5 reports the per-example inference cost. Table 16 reports the one-off adaptation cost that precedes it, covering router evolution, prompt evolution, reflection, candidate evaluation, and validation.

Table 16: Adaptation-phase cost on Qwen3-4B-Instruct, with per-example test cost repeated for reference.

Method	Adaptation calls	Adaptation tokens	Wall-clock	Test tokens/example
ACE	3,529	58.36M	2.90 h	21.30k
GEPA	77,715	226.81M	8.59 h	**3.47k**
EEVEE	104,460	154.71M	7.86 h	4.32k

Compared with GEPA, EEVEE uses 31.8% fewer adaptation tokens at similar wall-clock time, though it makes more and shorter calls. ACE adapts far more cheaply but deploys a $4.9\times$ larger inference-time context, so its cost is paid again on every query rather than once.

Because ACE's default single-epoch schedule uses far fewer adaptation tokens than the other two methods, we also run ACE with more offline epochs while keeping all other settings fixed. With two epochs, ACE uses 106.1M adaptation tokens and reaches an average score of 38.45 ± 1.24; with three epochs, it uses 162.0M tokens, slightly more than EEVEE's 154.7M, and reaches 40.45 ± 0.93. Even with a matched budget, ACE remains 11.3 points below EEVEE (51.75) and does not exceed the unadapted baseline (41.37). This indicates that increasing the number of epochs does not change the comparison, so we adopt ACE's default single-epoch schedule for the main comparison.

C Assets

The experiments use public benchmarks, public or provider-served model checkpoints, and published prompt-learning baselines such as GEPA and ACE. This work introduces a method rather than a new dataset or model checkpoint.

D Prompt and Case Study Details

This appendix provides excerpts from the diagnostic retest discussed in Section 3.6. The retest compares the empty prompt against the final router and prompt set from six completed EEVEE runs: three Qwen3-4B-Instruct runs and three DeepSeek-V3.2 runs. The raw run logs store the full router decisions, answer calls, and per-example flip bundles. We do not reproduce all six prompt sets; instead, we include representative learned prompt excerpts from one Qwen3-4B-Instruct run and one DeepSeek-V3.2 run, followed by output excerpts for the three main-paper cases.

D.1 Learned Prompt Excerpts

Qwen3-4B code slot. In a representative Qwen3-4B-Instruct run, the code-oriented slot learned a task-execution policy for incomplete Python functions: preserve the interface, infer the exact continuation, cover edge cases, and avoid extra text that would break execution.

```
You are given a task description that defines an incomplete Python
function or problem to solve. Carefully read the input, extract
the exact task requirements, and generate only the precise output
expected by the task without extra text, explanations, markdown,
comments, or formatting.
```

```
Identify the task type:
- Writing a Python function body with specific inputs and outputs
- Solving a mathematical or logical problem
- Validating conditions such as bracket matching or palindrome
 ↪  checks
- Processing structured data such as strings, arrays, and
 ↪  dictionaries

Parse the input format and constraints:
- Note data types, edge cases, boundary values, and examples
- Pay close attention to ordering and special constraints

Understand the output contract:
- If the task says output ONLY the continuation,
  write only the function body.
```

Qwen3-4B science slot. The same representative Qwen3-4B-Instruct run routes many GPQA Diamond examples to a scientific-reasoning slot. This prompt encourages systematic physical and mathematical modeling, but it does not provide task-specific missing knowledge for every possible GPQA Diamond domain.

```
You are an expert problem solver specializing in analytical
 ↪  reasoning,
physical modeling, combinatorial mathematics, and scientific
 ↪  computation.
Your task is to answer precise, domain-specific questions that
 ↪  require
careful application of physical laws, mathematical principles, or
combinatorial logic, based on exact input constraints and known
 ↪  scientific
or mathematical facts.

For multiple-choice questions with options (A-D), select the
 ↪  single correct
choice and present it in the format: "Answer: (X)".

For physical or astronomical problems involving equilibrium,
 ↪  energy,
temperature, or orbital dynamics, apply known physical laws such
 ↪  as the
Stefan-Boltzmann law, Kepler's laws, Doppler shift formula,
 ↪  blackbody
radiation, and Newtonian gravity.
```

DeepSeek-V3.2 formula slot. In a representative DeepSeek-V3.2 run, the formula slot learned a stricter numerical-output policy. This slot is especially aligned with the Formula benchmark, where the input gives an explicit formula and the evaluator expects a compact numeric answer.

```
You are a finance calculation assistant. Your task is to compute
 ↪  the
answer using the formula and data provided in the user's question,
then output only the numeric result with exactly two decimal
 ↪  places
and no other text.

Procedure:
1. Identify the formula given in the user's message.
2. Identify the explanation of each variable in the formula.
3. Extract the numeric values for each variable from the question
 ↪  text.
```

```
   Convert all percentage inputs into their decimal equivalent.
4. Substitute the values into the formula and perform the
↪  calculation.
5. If the formula calculates a financial rate, yield, return, or
↪  cost,
   do not multiply the decimal result by 100.
6. Round the final result to two decimal places.
7. Output only the resulting number, with no words, units, labels,
   currency symbols, or percentage signs.
```

D.2 Representative Raw Outputs

Formula: unit scale and sign. For a free-cash-flow computation, the required operation is operating cash flow minus capital expenditure. The target is a dollar-scale scalar value. The empty response flips the sign and keeps the million-scale decimal:

```
-0.40
```

The learned response applies the provided formula at the correct scale and emits the strict numeric answer:

```
Answer: 400000.00
```

This is representative of the Formula gains: the learned prompt pairs formula application with unit-scale discipline and a parseable final representation.

HumanEval: executable continuation. For a HumanEval function-completion task, the required behavior is to sum even elements at odd indices and return the sum. The empty response captures the expression but does not produce an executable function body:

```
sum(lst[i] for i in range(1, len(lst), 2) if lst[i] % 2 == 0)
```

The learned response preserves the logic while satisfying the continuation contract:

```
    total = 0
    for i in range(1, len(lst), 2):
        if lst[i] % 2 == 0:
            total += lst[i]
    return total
```

This is representative of the HumanEval gains: the learned behavior combines implementation logic with the executable contract expected by the evaluator.

GPQA Diamond: domain knowledge underweighted after stronger reasoning. For an exoplanet-density question, the target is the higher-mass Earth-composition planet. The empty response uses a domain-specific mass-radius prior:

```
For rocky planets of similar composition, radius grows sublinearly
↪  with
mass because stronger gravity compresses the material. A
↪  five-Earth-mass
rocky planet is therefore denser than Earth, while a
↪  half-Earth-mass
rocky planet is less dense.

Answer: the higher-mass Earth-composition option.
```

The learned response performs a more explicit density-formula comparison, but it treats equal composition as if it implied equal density:

```
Density is mass divided by volume. For a spherical planet,
rho = M / (4/3 pi R^3). If composition is the same, density is
constant; the radius scales with mass so that all
↪  Earth-composition
options remain approximately Earth density. Since the Earth-mass,
Earth-radius option is exactly Earth-like, choose that option.
```

The error is not a lack of reasoning steps. The learned prompt induces a stronger generic reasoning pattern, but the reasoning is grounded in the wrong physical prior: same composition does not imply constant density for rocky exoplanets, because self-compression changes the mass-radius relation.

E System Prompts

This appendix lists every fixed prompt used by EEVEE, reproduced verbatim from the code used for all reported runs. Fields in double braces are filled at run time, and `{% ... %}` blocks are Jinja loops and conditionals. Each LLM call sends the rendered template as a single user message, without a separate system message. GEPA and ACE use the prompts of their official implementations.

E.1 Template Overview

Table 17: Fixed prompts used by EEVEE and where each is used.

Prompt	Output	Used in
Target model	$\hat{y}$	Answer generation for every prompt evaluation (Section 2.1)
Routing	z	Slot selection for every routed input (Section 2.1); Selection 2-of-3 gate (Appendix B.4)
Prompt mutation	p_{mut}	Prompt evolve (Section 2.2); initialization and convergence (Section 2.3)
Prompt reflection	p_{ref}	Prompt evolve (Section 2.2); initialization and convergence (Section 2.3)
Router mutation	R_{mut}	Router evolve (Section 2.2)
Router analysis	failure analysis	Router evolve (Section 2.2)
Router reflection	R_{ref}	Router evolve (Section 2.2)
Default router	R_0	Start of exploration (Section 2.3); Default Router (Tables 2 and 8); online runs (Appendix B.4)
Manual router	fixed R	Manual Router (Tables 2 and 8)

E.2 Inference Prompts

Target model. The target model answers input x with slot prompt p_z, giving $\hat{y} = M(x; p_z)$ (Section 2.1). The same template is used whenever a prompt is evaluated, including the empty prompt.

```
{{ prompt_text }}
{% if context %}

## Context
{{ context }}
{% endif %}

## Question
{{ question }}
```

Routing. The router receives its router prompt, the slot labels A–D with their slot prompts, and the input, and returns one label, giving $z = R(x; \mathcal{P})$ (Section 2.1). The Selection 2-of-3 gate in Appendix B.4 queries this template three times per input.

```
{{ router_prompt }}

Available labels:
{% for item in slot_entries %}
- {{ item.slot_label }}
  Auxiliary prompt for {{ item.slot_label }}:
  ```
  {{ item.prompt_text }}
  ```
{% endfor %}

Task:
{{ visible_text }}

Return JSON only:
{"label": "A", "reason": "short reason"}
```

E.3 Evolution Prompts

Prompt mutation. Writes p_{mut} from mini-batch questions and reference prompts sampled from the slot's prompt pool (Section 2.2). The same template is used in initialization and convergence (Section 2.3).

```
You are a prompt engineer writing one new system prompt.
The prompt you design should help the model answer questions. You
↪  may refer to sampled question examples and historical prompt
↪  designs.

## Example Questions
These are raw questions from the current minibatch. Do not answer
↪  them.
Use them to infer the shared task patterns, output contracts, and
↪  failure modes.
{% for sample in examples %}
### Example {{ loop.index }}
Question:
{{ sample.get("question", "") }}
{% if sample.get("context") %}

Additional Context:
{{ sample.get("context") }}
{% endif %}
{% endfor %}
{% if context_prompts %}

## Reference Prompts
{% for item in context_prompts %}
- Prompt {{ loop.index }}
```
{{ item.prompt_text }}
```
{% endfor %}
{% endif %}

## Task
Write one new system prompt as plain text.
Use the examples and reference prompts to propose a prompt that
↪  can improve performance on tasks like these.
```

```
Read the inputs carefully, identify the input format, and infer a
 ↪  detailed task description for the task the assistant is
 ↪  expected to solve.
Infer the real task and output contract from the questions and
 ↪  preserve it exactly.
Identify niche, domain-specific, and task-specific factual
 ↪  information that may not be reliably available to the
 ↪  assistant in the future, and include the useful parts
 ↪  explicitly in the prompt.
If the examples suggest a generalizable strategy for solving the
 ↪  task, include that strategy in the prompt as well.
Keep the answer in the same format and artifact the task expects.
If the task expects only a continuation, function body, short
 ↪  answer, JSON object, XML snippet, or another partial output,
 ↪  do not rewrite it into a full function, wrapper, markdown
 ↪  block, explanation, or any other expanded format.
Provide enough concrete detail to make the instruction useful for
 ↪  future unseen examples, even if that makes it longer.
Provide the new prompt within ``` blocks.
```

Prompt reflection. Rewrites p_{mut} into p_{ref} from its mini-batch questions, target answers, model answers, and correctness (Section 2.2).

```
You are an expert prompt engineer. A candidate system prompt
 ↪  scored {{ eval_score }} ({{ correct }}/{{ total }} correct) on
 ↪  the current minibatch.
The model answers questions according to the designed prompt. We
 ↪  provide the current prompt and the model's outputs on sample
 ↪  tasks; improve the prompt based on this.

## Current Prompt
```
{{ prompt_text }}
```

## Evaluation Results
{% for case in all_cases %}
### #{{ loop.index }} {% if case.is_correct %}[CORRECT]{% else
 ↪  %}[INCORRECT]{% endif %}
Question: {{ case.question }}
Expected: {{ case.target }}
Model Output: {{ case.answer }}
{% endfor %}

## Task
Improve the prompt based on the results above.
Read the evaluation results carefully, identify the input format,
 ↪  and infer a detailed task description for the task the
 ↪  assistant is expected to solve.
Read the model outputs and the corresponding results carefully.
 ↪  Identify niche, domain-specific, and task-specific factual
 ↪  information that appears to be missing from the current
 ↪  prompt, especially information that may not be reliably
 ↪  available to the assistant in the future.
If the results suggest a generalizable strategy for solving the
 ↪  task, include that strategy in the prompt as well.
If the failures suggest missing decision rules, normalization
 ↪  rules, edge-case handling, or domain facts, add those details
 ↪  explicitly to the prompt.
Keep the answer in the same format and artifact the task expects.
```

```
Preserve output contracts that are already helping, and only
↪  change instructions that appear to hurt performance on
↪  examples like these.
Provide enough concrete detail to make the instruction useful for
↪  future unseen examples, even if that makes it longer.
Provide the new prompt within ``` blocks.
```

Router mutation. Writes R_{mut} from reference routers sampled from $\mathcal{B}_R$, the current slot prompts, and mini-batch inputs labeled with a slot that solves them (Section 2.2).

```
You are an expert prompt engineer writing one new prompt for a
↪  router.
The router selects a label based on the task content; each label
↪  maps to a different prompt that steers how the model answers.
↪  To give each task the right kind of guidance, the router
↪  prompt must explain clearly when to choose which label. Use
↪  the task examples and prior router prompt designs to shape the
↪  new router prompt.
Below you will find: example tasks like those the router may see,
↪  the current label set, the auxiliary prompt applied after each
↪  choice to guide the model's answer, illustrative routing
↪  examples, and prior router prompts for reference.

## Example Tasks
These are raw routed examples. Do not solve them.
Use them to infer the shared routing patterns, label distinctions,
↪  output-contract differences, and failure modes.

## Available Labels
{% for item in slot_entries %}
- {{ item.slot_label }}
  Auxiliary prompt for {{ item.slot_label }}:
  ```
  {{ item.prompt_text }}
  ```
{% endfor %}

{% for sample in support_examples %}
### Example {{ loop.index }}
Task Input:
{{ sample.get("visible_text", "") }}
Best Label:
{{ sample.get("best_label", "") }}
{% endfor %}

{% if router_a or router_b %}

## Reference Router Prompts
### Prompt A
```
{{ router_a }}
```

### Prompt B
```
{{ router_b }}
```
{% endif %}

## Task
```

```
Write one new router system prompt as plain text.
Use the routed examples, reference router prompts, and available
 ↪  labels to propose a router prompt that can improve routing
 ↪  performance on examples like these.
Read the inputs carefully, identify the input format, and infer a
 ↪  detailed routing description for how tasks should be assigned
 ↪  to the available labels.
Infer the superior routing choice pattern from the examples and
 ↪  preserve it exactly.
Identify niche, domain-specific, task-specific, and
 ↪  format-specific routing information that may not be reliably
 ↪  available to the assistant in the future, and include the
 ↪  useful parts explicitly in the router prompt.
If the examples suggest a generalizable routing strategy, include
 ↪  that strategy in the router prompt as well.
Preserve the label set and the routing contract exactly.
If routing depends on subtle output-contract differences, artifact
 ↪  requirements, reasoning style, domain cues, or normalization
 ↪  rules, state those routing rules explicitly instead of relying
 ↪  on keyword overlap alone.
Provide enough concrete detail to make the router instruction
 ↪  useful for future unseen examples, even if that makes it
 ↪  longer.
Write the routing policy directly in terms of when to choose A, B,
 ↪  C, D, etc.
You may use the auxiliary prompts above to understand what each
 ↪  label is trying to help with. Do not explicitly reference or
 ↪  quote them in the router prompt.
Provide the new router prompt within ``` blocks.
```

Router analysis. For an input whose routed slot fails while another slot succeeds, explains why the better slot matches the task, using both slot prompts, both model answers, and the ground truth (Section 2.2).

```
Our router is responsible for choosing the right routing label for
 ↪  the task. However, it sometimes picks the wrong label; analyze
 ↪  and explain why these tasks should use the better label.
Read the task carefully, identify the input format, and infer what
 ↪  signal the router should notice here.
Compare the wrong label and the better label using the downstream
 ↪  prompt texts, the model outputs, and the ground truth.
First identify what capability, output constraint, reasoning
 ↪  style, or domain handling this task actually needs.
Then explain why the better label is a better match because its
 ↪  downstream prompt is more likely to provide that needed
 ↪  behavior.
Be concrete about why the wrong label is misleading here and what
 ↪  routing rule would generalize to future examples.
Keep the answer short but concrete. Your answer should follow this
 ↪  pattern: the router should choose label X' rather than label
 ↪  X, because choosing X' helps the model ... whereas choosing X
 ↪  leads to ...

Task:
{{ visible_text }}

Ground truth:
{{ ground_truth }}

Wrong label: {{ wrong_slot_label }}
```

```
Wrong prompt:
```
{{ wrong_prompt_text }}
```
Wrong answer:
{{ wrong_answer }}

Better label: {{ better_slot_label }}
Better prompt:
```
{{ better_prompt_text }}
```
Better answer:
{{ better_answer }}
```

Router reflection. Rewrites R_{mut} into R_{ref} from the router analyses of up to five such failures (Section 2.2).

```
You are an expert prompt engineer improving a prompt for a router.
The router selects a label based on the task content; each label
 ↪  maps to a different prompt that steers how the model answers.
 ↪  To give each task the right kind of guidance, the router
 ↪  prompt must spell out clearly when to pick which label. The
 ↪  current router prompt leads to poor choices on some tasks;
 ↪  analyze these failures and improve the router prompt so the
 ↪  router makes better selections.
Below you will find: the current router prompt, every label the
 ↪  router may choose, the auxiliary prompt applied after each
 ↪  choice to guide the model's answer, and failure cases to learn
 ↪  from.

## Current Router Prompt
```
{{ router_prompt }}
```

## Available Labels
{% for item in slot_entries %}
- {{ item.slot_label }}
  Auxiliary prompt for {{ item.slot_label }}:
  ```
  {{ item.prompt_text }}
  ```
{% endfor %}

## Routed Failure Analyses
{% for item in bad_cases %}
### #{{ loop.index }}
Task:
{{ item.visible_text }}
Wrong Label: {{ item.wrong_slot_label }}
Better Label: {{ item.better_slot_label }}
Routing Analysis: {{ item.suggestion }}
{% endfor %}

## Task
Improve the router prompt based on the routed failure analyses
 ↪  above.
```

```
Read the failure analyses carefully, identify the input format,
 ↪  and infer a detailed routing description for how tasks should
 ↪  be assigned to the available labels.
Read carefully. Identify niche, domain-specific, task-specific,
 ↪  and format-specific routing information that appears to be
 ↪  missing from the current router prompt, especially information
 ↪  that may not be reliably available to the assistant in the
 ↪  future.
If the failures suggest a generalizable routing strategy, include
 ↪  that strategy in the router prompt as well.
If the failures suggest missing decision rules, normalization
 ↪  rules, edge-case handling, or domain facts that affect
 ↪  routing, add those details explicitly to the router prompt.
Only change routing instructions that appear to hurt performance
 ↪  on examples like these, and preserve routing rules that are
 ↪  already helping.
Provide enough concrete detail to make the router instruction
 ↪  useful for future unseen examples, even if that makes it
 ↪  longer.
Write the routing policy directly in terms of when to choose A, B,
 ↪  C, D, etc.
You may use the auxiliary prompts above to understand what each
 ↪  label is trying to help with. Do not explicitly reference or
 ↪  quote them in the router prompt.
Provide the new router prompt within ``` blocks.
```

E.4 Fixed Router Prompts

Default router. The initial router R_0 from which exploration starts (Section 2.3), and the fixed router of the Default Router variant in Tables 2 and 8. It also initializes the router in the online experiment (Appendix B.4).

```
Choose the slot whose prompt is most likely to help solve the
 ↪  task. Use prompt intent and output constraints, not keyword
 ↪  overlap.
```

Manual router. The fixed router of the Manual Router variant in Tables 2 and 8, written once by GPT-5.4.

```
Route the task to the auxiliary prompt that is most likely to
 ↪  produce a correct answer.

Base the decision on the task's actual requirements and each
 ↪  auxiliary prompt's intent. Prioritize compatibility in this
 ↪  order:
1. Required output contract: exact format, answer length, allowed
 ↪  text, precision, structure, and any prohibitions.
2. Core capability needed: reasoning style, calculation, coding,
 ↪  extraction, classification, verification, constraint
 ↪  following, or domain-specific knowledge.
3. Problem structure: open-ended versus constrained answer,
 ↪  standalone versus context-dependent, single-step versus
 ↪  multi-step, deterministic versus judgment-heavy.
4. Failure risk: prefer prompts that explicitly guard against the
 ↪  most likely mistake for this task, such as format drift,
 ↪  missed edge cases, arithmetic slips, unsupported assumptions,
 ↪  or over-explaining.
```

```
Do not choose by label name, label order, or superficial keyword
 ↪  overlap. If several labels fit, choose the most specific
 ↪  prompt that satisfies all task constraints; if none is clearly
 ↪  specialized, choose the safest general prompt that does not
 ↪  conflict with the required output.
```

E.5 Benchmark Input Templates

Each benchmark example is formatted with the template below and inserted into the question field of the target-model template.

GPQA Diamond. Used in the four-benchmark suite.

```
Here are some example questions from experts.
Answer the final question yourself, following the format of the
 ↪  previous questions exactly.

Question: {{ question }}
Choices:
(A) {{ choices.A }}
(B) {{ choices.B }}
(C) {{ choices.C }}
(D) {{ choices.D }}
Answer:
```

TheoremQA. Used in the four-benchmark suite.

```
You are supposed to provide a solution to a given problem.


Problem:
{{ question }}
Solution:
```

HumanEval. Used in the four-benchmark suite; `{{ prompt }}` is the original HumanEval function signature and docstring.

```
{{ prompt }}

IMPORTANT OUTPUT REQUIREMENT FOR THIS TASK:
- The above is an incomplete Python function. In your final
 ↪  answer, output ONLY the continuation (the function body),
 ↪  starting from the indented implementation.
- Do NOT repeat the `def` line or function signature -- only write
 ↪  the body that completes the function.
- Do not include markdown fences, explanations, or any extra text.
```

Formula and FiNER. Formula, in the four-benchmark suite, and FiNER, in the single-benchmark study (Figure 4), use the instruction text supplied with each FinLoRA example unchanged.

IFBench. Used in the single-benchmark study (Figure 4); the original IFBench prompt is used unchanged.

MBPP. Used as a held-out benchmark (Table 3). The three few-shot examples are MBPP tasks 2, 3, and 4.

```
{% for ex in fewshot_examples %}Example task: {{ ex.text }}
Your code should pass these tests:
{{ ex.test_list[0] }}
```

```
{{ ex.test_list[1] }}
{{ ex.test_list[2] }}
Solution:
```python
{{ ex.code }}
```

{% endfor %}Now solve this task: {{ text }}
Your code should pass these tests:
{{ test_list[0] }}
{{ test_list[1] }}
{{ test_list[2] }}

IMPORTANT OUTPUT REQUIREMENT FOR THIS TASK:
- In your final answer, output ONLY the complete Python function
 ↪ definition (including the `def` line).
- Do not include markdown fences, explanations, or any extra text.
```

MMLU-Pro. Used as a held-out benchmark (Tables 3 and 15). The subject header is followed by the question and its options, one per line with letters up to J.

```
The following are multiple choice questions (with answers) about
 ↪  {{ subject }}. Think step by step and then output the answer
 ↪  in the format of "The answer is (X)" at the end.


Question: {{ question }}
Options: A. {{ option_A }}
B. {{ option_B }}
...
Answer: Let's think step by step.
```