# GEPA: Reflective Prompt Evolution Can Outperform Reinforcement Learning

**Lakshya A Agrawal**[1], **Shangyin Tan**[1], **Dilara Soylu**[2], **Noah Ziems**[4],
**Rishi Khare**[1], **Krista Opsahl-Ong**[5], **Arnav Singhvi**[2,5], **Herumb Shandilya**[2],
**Michael J Ryan**[2], **Meng Jiang**[4], **Christopher Potts**[2], **Koushik Sen**[1],
**Alexandros G. Dimakis**[1,3], **Ion Stoica**[1], **Dan Klein**[1], **Matei Zaharia**[1,5], **Omar Khattab**[6]

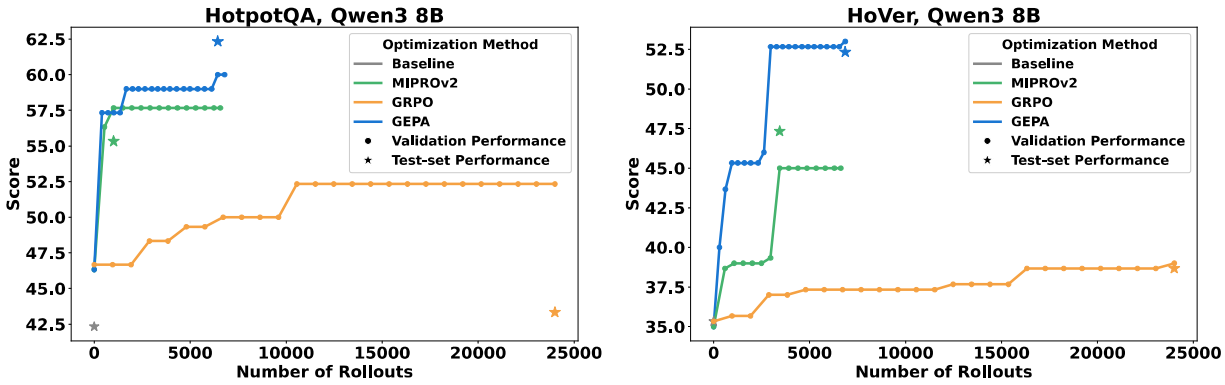
[1]UC Berkeley [2]Stanford University [3]BespokeLabs.ai [4]Notre Dame [5]Databricks [6]MIT



## Abstract

Large language models (LLMs) are increasingly adapted to downstream tasks via reinforcement learning (RL) methods like Group Relative Policy Optimization (GRPO), which often require thousands of rollouts to learn new tasks. We argue that the interpretable nature of *language* can often provide a much richer learning medium for LLMs, compared with policy gradients derived from sparse, scalar rewards. To test this, we introduce GEPA (**Ge**netic-**Pa**reto), a prompt optimizer that thoroughly incorporates *natural language reflection* to learn high-level rules from trial and error. Given any AI system containing one or more LLM prompts, GEPA samples system-level trajectories (e.g., reasoning, tool calls, and tool outputs) and reflects on them in natural language to diagnose problems, propose and test prompt updates, and combine complementary lessons from the Pareto frontier of its own attempts. As a result of GEPA's design, it can often turn even just a few rollouts into a large quality gain. Across four tasks, GEPA outperforms GRPO by 10% on average and by up to 20%, while using up to 35x fewer rollouts. GEPA also outperforms the leading prompt optimizer, MIPROv2, by over 10% across two LLMs, and demonstrates promising results as an inference-time search strategy for code optimization.

## 1 Introduction




(a) HotpotQA, Qwen3 8B

(b) HoVer, Qwen3 8B

Figure 1: A comparison of the learning behavior of our proposed GEPA prompt optimizer against a state-of-the-art prompt optimizer (MIPROv2) and the GRPO (24,000 rollouts) algorithm. As more rollouts are sampled, the prompt optimizers can learn much more quickly than GRPO. GEPA substantially outperforms both GRPO and MIPROv2 in final score. The Test-set star markers demonstrate the performance gap in a held-out set of questions.

Large language models (LLMs) have enabled the development of agents and systems that combine fuzzy natural-language behavior specification with tools like retrieval and code execution. These types of systems raise the question of how LLMs should be "optimized" for the best downstream performance within their harness. One popular approach for adapting LLMs to downstream tasks is reinforcement learning with verifiable rewards (RLVR), including

algorithms such as Group Relative Policy Optimization (GRPO) (Shao et al., 2024). Such RL methods cast success metrics as a scalar reward observed at the end of each rollout (Lambert, 2025) and use these rewards to estimate gradients for policy improvement.

While these RL approaches are effective, they typically require tens of thousands of rollouts in practice to fit new tasks. For example, recent works leveraging GRPO across a range of tasks typically use up to hundreds of thousands of rollouts for training (Chen et al., 2025b; Wu et al., 2025c; Zhang et al., 2025; Jin et al., 2025; Si et al., 2025; Wang et al., 2025a; Java et al., 2025; Chen et al., 2025a; Wu et al., 2025a; Sha et al., 2025; Lin et al., 2025; Peng et al., 2025; Song et al., 2025). This sample inefficiency can quickly become a serious bottleneck: many downstream LLM applications invoke expensive tool calls, have limited inference budget for sampling from the LLM itself, or simply cannot finetune the weights of the largest or best-performing LLMs.

We observe that the rollouts sampled from even highly sophisticated LLM systems can be serialized into traces of natural (and formal) language, as they contain nothing but the instructions of each LLM module, the resulting LLM reasoning chains, tool calls, and potentially the internal workings of the reward function (for example, compiler error messages, before they are collapsed into scalar rewards). Because such serialized trajectories can be readily understood by modern LLMs, we argue that *algorithms that learn deliberately in natural language by reflecting on these trajectories* can potentially make much more effective use of the strong language priors that LLMs have, compared with standard RL approaches.

To operationalize this, we introduce GEPA (Genetic-Pareto), a *reflective* prompt optimizer for compound AI systems that merges textual reflection with multi-objective evolutionary search. GEPA iteratively mutates every prompt within the AI system in light of natural language feedback drawn from new rollouts. In each mutation, the candidate prompt is derived from an ancestor, accumulating high-level lessons derived from observations and LLM feedback. To avoid the local optima that afflict greedy prompt updates, GEPA maintains a Pareto front: instead of evolving only the global best prompt, it stochastically explores the top-performing prompts for each problem instance, thereby diversifying strategies and encouraging robust generalization.

We evaluate GEPA on four diverse tasks—multi-hop reasoning (HotpotQA; Yang et al. 2018), instruction following (IFBench; Pyatkin et al. 2025b), privacy-aware delegation (PUPA; Li et al. 2025a), and retrieval-augmented verification (HoVer; Jiang et al. 2020)—using both open (Qwen3 8B; Yang et al. 2025; Team 2025) and proprietary (GPT-4.1 mini; OpenAI 2025) models. Our results show that GEPA demonstrates robust generalization and is highly sample-efficient: on Qwen3 8B, GEPA outperforms GRPO (24,000 rollouts with LoRA) by up to 19% while requiring up to 35$\times$ fewer rollouts. Overall, GEPA achieves an average improvement of +10% over GRPO across all tasks. Furthermore, GEPA surpasses the previous state-of-the-art prompt optimizer, MIPROv2 (Opsahl-Ong et al., 2024), on every benchmark and model, obtaining aggregate optimization gains of +14%, more than doubling the gains achieved by MIPROv2 (+7%).

Even qualitatively, GEPA generated prompts can be highly effective. Figure 2 highlights excerpts from a prompt crafted by GEPA for the query creation module of a multi-hop question answering system (Used in HotpotQA). We also find that in most cases, even a single *reflective* prompt update can give large improvements (as highlighted in the optimization trajectory in Figure 5). These results demonstrate that reflective prompt evolution using language feedback enables substantial sample efficiency and robust generalization, providing a practical path to optimizing complex, real-world AI workflows in data- or budget-constrained environments. Finally, we also show promising preliminary results demonstrating GEPA's use as an inference-time search strategy for code optimization over NPUEval (Kalade & Schelle, 2025) and KernelBench (Ouyang et al., 2025).

## 2 Problem Statement

**Compound AI Systems.** We follow related work in defining a **compound AI system** as any modular system composed of one or more language model (LLM) invocations, potentially interleaved with external tool calls, orchestrated through arbitrary control flow. This definition subsumes a broad class of real-world LLM-based AI systems, including *agents*, *multi-agent systems*, and general-purpose scaffolding techniques like ReAct (Yao et al., 2023), Archon (Saad-Falcon et al., 2025), etc. Following Soylu et al. (2024); Khattab et al. (2024); Opsahl-Ong et al. (2024); Tan et al. (2025), we formalize such a system as $\Phi = (M, C, \mathcal{X}, \mathcal{Y})$, where $M = \langle M_1, \ldots, M_{|M|} \rangle$ denotes language modules, $C$ specifies control flow logic, and $\mathcal{X}$, $\mathcal{Y}$ are global input/output schemas. Each module $M_i = (\pi_i, \theta_i, \mathcal{X}_i, \mathcal{Y}_i)$ is an LLM subcomponent: $\pi_i$ is its (system) prompt including instructions and few-shot demonstrations; $\theta_i$ the underlying model weights; $\mathcal{X}_i, \mathcal{Y}_i$ are input/output schemas. At runtime, $C$ orchestrates the sequencing and invocation of modules—e.g., passing outputs from one module to another, invoking modules conditionally, or leveraging tool APIs. This way, $C$ can invoke different modules in any order multiples of times.

**Seed Prompt for Second-Hop of Multi-Hop QA System**

Given the fields `question`, `summary_1`, produce the fields `query`.

**GEPA's Optimized Prompt for Second-Hop of Multi-Hop QA System, GPT-4.1 Mini**

**You will be given two input fields: `question` and `summary_1`. Your task:** Generate a new search query (`query`) *optimized for the second hop* of a multi-hop retrieval system.

- The original user question is typically complex and requires information from multiple documents to answer.
- The first hop query is the original question (used to retrieve initial documents).
- Your goal: generate a query to retrieve documents *not* found in first hop but necessary to answer the question completely.

**Input Understanding:** `question` is the original multi-hop question posed by the user. `summary_1` is a concise summary of information from a document retrieved in the first hop, which partially addresses the question.

**Purpose and Context:**

- Your generated `query` aims to find the *missing pieces* of information needed to fully answer the question. ...
- The query must retrieve relevant documents *NOT* found in first hop ... for final answer extraction.

**Key Observations and Lessons:**

- First-hop documents often cover one entity or aspect.
- Remaining relevant documents often involve connected or higher-level concepts mentioned in `summary_1` but not explicitly asked in the original question. The `query` should target these *missing*, but logically linked, documents.
- Avoid merely paraphrasing the original question or restating known facts from `summary_1`.
- Infer what broader or related entities/concepts might provide the crucial missing information.
- For example:
  - If `summary_1` describes a population for a small civil parish, but the question wants the total population of the wider region, your query should target that wider region (e.g., "Madeira archipelago population in 2011").
  - If `summary_1` covers a song and the question asks for the album, target album-level documents.

**How to Build the Query:**

- Identify entities or topics mentioned in `summary_1` that are related but different from first-hop documents.
- Reframe the query to explicitly mention these broader or related entities *connected to the original question*.
- Include relevant key context from the question to maintain specificity, but shift focus to the missing piece.
- The goal is to retrieve documents that link or complement what was retrieved initially.

**Practical Strategy:**

- Read the `summary_1` carefully to spot references to bigger contexts or other entities not covered in the first hop.
- Ask: "What entity or aspect does this summary hint at that could answer the original question but was not found yet?"
- Formulate a precise, focused factual query targeting that entity or concept to retrieve the missing documents.

**Output:**

- Produce `query` as a clear, concise question or keyword phrase designed for efficient retrieval of second-hop documents.
- Ensure the query relates logically to the original question while targeting the broader or complementary knowledge identified in `summary_1`. ... Do not include the original question or simply rephrase it. Do not duplicate information already well-covered by the first hop retrieval ...

Figure 2: This figure shows an example prompt generated by GEPA for the second-hop document retrieval to be performed in a multi-hop question-answer system, along with the seed prompt it started with. Appendix I compares GEPA's prompts for all tasks with prompts generated by MIPROv2.

**Compound AI System Optimization.** Given $\Phi$, let $\Pi_\Phi = \langle \pi_1, \ldots, \pi_{|M|} \rangle$ denote the collection of all module prompts and $\Theta_\Phi = \langle \theta_1, \ldots, \theta_{|M|} \rangle$ the set of module weights. The learnable parameters are thus $\langle \Pi, \Theta \rangle_\Phi$. For a task instance $(x, m)$—where $x$ maps to the input schema $\mathcal{X}$ and $m$ contains evaluator metadata (e.g., gold answers, evaluation rubrics, code unit tests)—the system induces an output $y = \Phi(x; \langle \Pi, \Theta \rangle_\Phi)$. A metric $\mu : \mathcal{Y} \times \mathcal{M} \to [0, 1]$ then measures the output quality of $y$ with respect to metadata $m$ (for example by calculating, exact match, F1, pass rate, etc.). The optimization problem is thus defined by:

$$\langle \Pi^*, \Theta^* \rangle_\Phi = \arg \max_{\langle \Pi, \Theta \rangle_\Phi} \mathbb{E}_{(x,m)\sim\mathcal{T}} \left[ \mu\big(\Phi(x; \langle \Pi, \Theta \rangle_\Phi),\, m\big) \right], \tag{1}$$

where $\mathcal{T}$ is a task distribution.

**Sample-Efficient Optimization.** In many real-world scenarios, rollouts—concretely, invocations of $\Phi$ plus evaluation by $\mu$—are often computationally, monetarily, or timewise expensive. The optimizer is thus limited to at most $B$ rollouts on a dataset $\mathcal{D}_{\text{train}} = \{(x, m)_i\}_{i=1}^{N}$ with full access to $\mu$. The goal is to identify parameters $\langle \Pi^*, \Theta^* \rangle_\Phi$ that maximize held-out performance, subject to not exceeding the rollout budget $B$:

$$\langle \Pi^*, \Theta^* \rangle_\Phi = \arg \max_{\langle \Pi, \Theta \rangle_\Phi} \mathbb{E}_{(x,m)\sim\mathcal{T}} \left[ \mu\big(\Phi(x; \langle \Pi, \Theta \rangle_\Phi), m\big) \right] \quad \text{s.t.} \quad \#\text{rollouts} \leq B. \tag{2}$$

This formulation captures the core challenge motivating our work: *How can we extract maximal learning signal from every expensive rollout to enable effective adaptation of complex, modular AI systems in low-data or budget-constrained settings?*

## 3 GEPA: Reflective Prompt Evolution

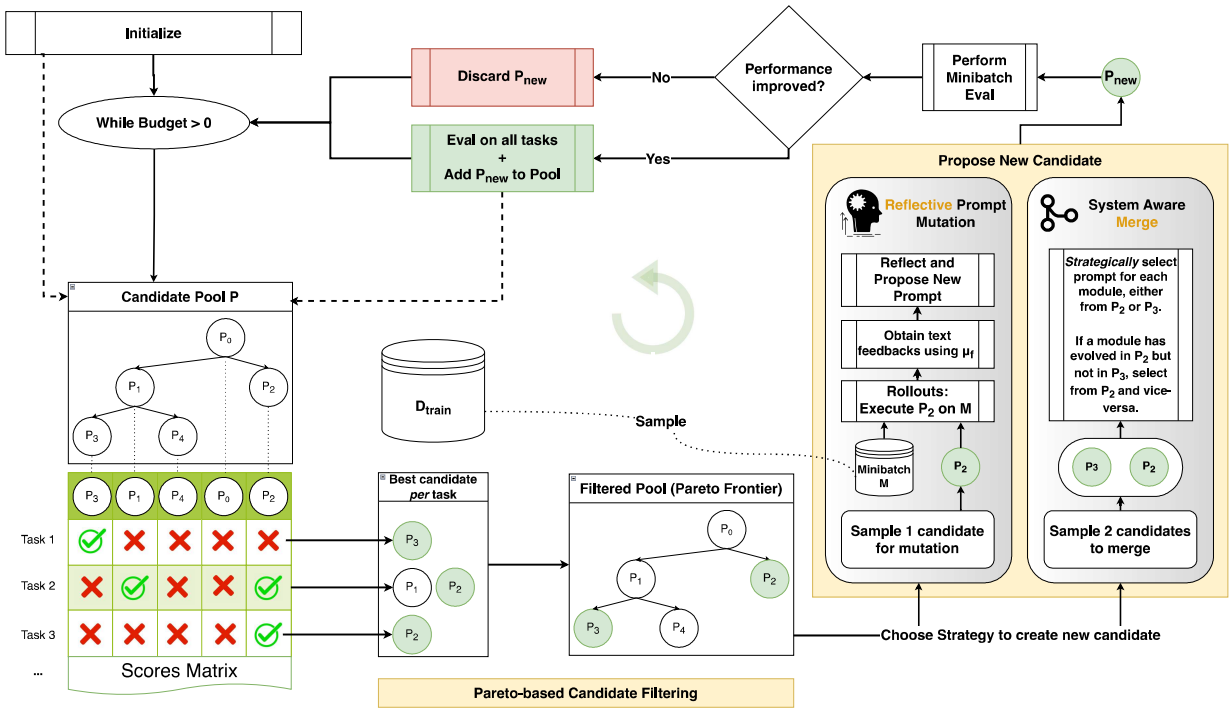



Figure 3: GEPA works iteratively—proposing a new candidate in every iteration by improving some existing candidates using one of the two strategies (Reflective Prompt Mutation (Section 3.2) or System Aware Merge (Appendix F)), first evaluating them on a minibatch, and if improved, evaluating on a larger dataset. Instead of selecting the best performing candidate to mutate always, which can lead to a local-optimum, GEPA introduces Pareto-based candidate sampling (Section 3.3), which filters and samples from the list of best candidates *per* task, ensuring sufficient diversity. Overall, these design decisions allow GEPA to be highly sample-efficient while demonstrating strong generalization.

We introduce GEPA (Genetic-Pareto), a sample-efficient optimizer for compound AI systems motivated by three core principles: genetic prompt evolution (Section 3.1), reflection using natural language feedback (Section 3.2), and Pareto-based candidate selection (Section 3.3). Figure 3 gives an overview of GEPA and the full GEPA algorithm is formalized in Figure 4.

GEPA receives the following inputs: A compound AI system $\Phi$ instantiated with simple prompts to be optimized, training dataset $D_{train}$ (consisting of task instances $(x, m)$ as described in Section 2), the standard evaluation metric $\mu$ for the task, a feedback function $\mu_f$ (introduced in Section 3.2) and the total rollout budget $B$.

### 3.1 Genetic Optimization Loop

Given a compound AI system $\Phi$, the goal of the optimization process is to identify a set of parameters $\langle \Pi, \Theta \rangle_\Phi$ that maximize the score over a task distribution. GEPA starts by initializing a *candidate pool* $\mathcal{P}$, where a *candidate* is

**Algorithm 1** GEPA: Reflective Evolutionary Prompt Optimizer

**Require:** Inputs: System $\Phi$, dataset $\mathcal{D}_{\text{train}}$, eval metric $\mu$, feedback function $\mu_f$, budget $B$
**Require:** Hyperparams: minibatch size $b$, Pareto set size $n_{pareto}$
1: Split $\mathcal{D}_{\text{train}}$ into $\mathcal{D}_{\text{feedback}}$, $\mathcal{D}_{\text{pareto}}$, s.t. $|D_{pareto}| = n_{pareto}$
2: Initialize candidates $\mathcal{P} \leftarrow [\Phi]$, parents $\mathcal{A} \leftarrow$ [None]
3: **for** each $(x_i, m_i)$ in $\mathcal{D}_{\text{pareto}}$ **do**
4: $S_\Phi[i] \leftarrow \mu(\Phi(x_i), m_i)$
5: **end for**
6: **while** budget $B$ not exhausted **do**
7: $k \leftarrow$ SELECTCANDIDATE($\mathcal{P}, S$)
8: $j \leftarrow$ SELECTMODULE($\Phi_k$)
9: $\mathcal{M} \leftarrow$ minibatch of size $b$ from $\mathcal{D}_{\text{feedback}}$
10: Gather feedback, scores, traces for $\Phi_k[j]$ on $\mathcal{M}$ using $\mu_f$
11: $\pi'_j \leftarrow$ UPDATEPROMPT($\pi_j$, feedbacks, traces[j])
12: $\Phi' \leftarrow$ Copy of $\Phi_k$ w/ module $j$ updated by $\pi'_j$
13: $\sigma, \sigma' \leftarrow$ avg score on $\mathcal{M}$ (before, after)
14: **if** $\sigma'$ improved **then**
15: Add $\Phi'$ to $\mathcal{P}$; Add $k$ to $\mathcal{A}$
16: **for** each $(x_i, m_i)$ in $\mathcal{D}_{\text{pareto}}$ **do**
17: $S_{\Phi'}[i] \leftarrow \mu(\Phi'(x_i), m_i)$
18: **end for**
19: **end if**
20: **end while**
21: **return** $\Phi^*$ maximizing average score on $\mathcal{D}_{\text{pareto}}$

**Algorithm 2** Pareto-based candidate selection

**function** SELECTCANDIDATE($\mathcal{P}, S$)
// Build instance-wise Pareto sets
**for** each $i$ **do**
$s^*[i] \leftarrow \max_k S_{\mathcal{P}[k]}[i]$
$\mathcal{P}^*[i] \leftarrow \{\mathcal{P}[k] : S_{\mathcal{P}[k]}[i] = s^*[i]\}$
**end for**
$\mathcal{C} \leftarrow$ unique candidates in $\bigcup_i \mathcal{P}^*[i]$
$D \leftarrow \emptyset$
**while** there exists $\Phi \in \mathcal{C} \setminus D$ dominated by another in $\mathcal{C} \setminus D$ **do**
$D \leftarrow D \cup \{\Phi\}$
**end while**
Remove $D$ from each $\mathcal{P}^*[i]$ to get $\hat{\mathcal{P}}^*[i]$
Let $f[\Phi] =$ number of $i$ for which $\Phi \in \hat{\mathcal{P}}^*[i]$
Sample $\Phi_k$ from $\hat{\mathcal{C}}$ with probability $\propto f[\Phi_k]$
**return** index $k$ of $\Phi_k$ in $\mathcal{P}$
**end function**

Figure 4: (**Left**) GEPA's core algorithm for reflective prompt evolution. GEPA works iteratively, in each iteration, selecting some of the current candidates to evolve (line 7), executing the identified candidate on a minibatch of rollouts, while utilizing a special *feedback function* $\mu_f$ to gather module specific feedback when available (lines 9-10, described in detail in Section 3.2), using an LLM to reflectively update the prompt (line 11), and evaluating whether the system instantiated with the new prompt improved the performance on the minibatch (line 14). If improved, GEPA then proceeds to evaluate the new system candidate on the full $D_{pareto}$ set, adding it to the list of candidates tracked and marking the new system's parent. (**Right**) The SelectCandidate subprocedure used by GEPA's core algorithm is tasked with identifying the best candidate to evolve in the next optimization iteration. GEPA's chief candidate selection strategy is to find non-dominated candidates in the Pareto frontier (of all task instances), and stochastically select one of them based on their appearance frequency in the Pareto front.

a concrete instantiation of the learnable parameters of the compound system, $\langle \Pi, \Theta \rangle_\Phi$. Initially, the candidate pool consists only of the base system's parameters as the sole candidate. GEPA then proceeds in an optimization loop, iteratively proposing new candidates and adding them to the pool, continuing this process until the evaluation budget is exhausted.

Iteratively, GEPA proposes increasingly effective candidates by modifying existing ones through *mutation* or *crossover*, informed by learning signals from newly gathered rollouts and while tracking each new candidates' ancestry. This enables GEPA to accumulate lessons along the genetic tree as optimization progresses. Each new candidate inherits learning signals from its parents, as well as signals from the current rollout.

During each iteration, GEPA identifies promising candidates from the candidate pool (candidate selection), proposes a new candidate—possibly by mutating prompts in a module based on reflective feedback or by performing crossover between two candidates—and evaluates this new variant on a minibatch of tasks. If the newly proposed candidate demonstrates improved performance relative to its parent(s) on the local minibatch, then GEPA adds the new candidate to the *candidate pool* $\mathcal{P}$. This involves tracking internal data structures including tracking the ancestry of the new candidate, along with the full evaluation of the new candidate on a $D_{pareto}$, a validation set used for candidate selection.

After the budget is depleted, GEPA returns the candidate with the best aggregate performance on $D_{pareto}$.

### 3.2 REFLECTIVE PROMPT MUTATION

Natural language traces generated during the execution of a compound AI system offer rich *visibility* into the behavior and responsibilities of each module, as they capture the intermediate inferences and underlying reasoning steps. When these traces are paired with the final outcome of the system (e.g., success or failure), they provide substantial *diagnostic* value, allowing practitioners to trace errors or successes back to specific decisions made at the module level. LLMs can then leverage these traces via *reflection* to perform implicit credit assignment, attributing responsibility for the final outcome to the relevant modules. This process of *reflection* can then be used to make targeted updates to individual modules, making large and effective updates to the whole system's behavior.

GEPA operationalizes this as follows: Given a selected *candidate* to mutate in the current iteration of the optimization loop, GEPA updates the system with the *candidate* parameters, selects a target module within the system to improve (via round robin to ensure all modules receive updates), and generates a few rollouts over a minibatch sampled from the training dataset, recording their outcomes (success/failure). By examining the execution traces of the system, GEPA identifies the target module's inputs, outputs, and reasoning. With this, GEPA uses an LLM to reflectively examine this information, attributing successes or failures to elements of the module's prompt (or omission thereof), and propose new instructions for the target module. A new candidate is then proposed as a copy of the current candidate, with the target module's prompt updated to the new proposed prompt. The meta-prompt used by GEPA to perform reflective prompt update is presented in Appendix B.

**Evaluation trace as diagnostic signal:** While the system's own execution traces already provide useful information to enable successful *reflection* and prompt updates, we identify another source of highly diagnostic information: The evaluation metric $\mu$. Often, the evaluation metric $\mu$ applies rich strategies to perform evaluations to arrive at a final score. For example, code evaluation environments run a series of steps (compilation, execution, profiling, etc.) each of which produce natural language traces, before providing a scalar reward.

We propose the use of these *evaluation traces* in addition to the system's own *execution traces* to perform reflective credit assignment, and targeted prompt updates. GEPA operationalizes this as a simple update to the evaluation metric $\mu$, to create a *feedback function* $\mu_f$, which identifies relevant textual traces produced during the evaluation metric's execution, and returns the final score along with `feedback_text`. Whenever available, such a feedback function can also provide module-level feedback (for example, in multi-hop systems, the evaluator can provide feedback after each hop of the system).

### 3.3 PARETO-BASED CANDIDATE SELECTION

GEPA is a highly modular algorithm capable of supporting various strategies for selecting candidates in each iteration of optimization. Crucially, the choice of candidate selection strategy determines the exploration-exploitation tradeoff adopted by the optimizer. A naive strategy is to always select the best-performing candidate in the pool. However, this can cause the optimizer to get stuck in a local optimum within the prompt space: once a dominant strategy is found, it becomes difficult to surpass, and the optimizer exhausts its budget without learning new, potentially better strategies. An example search tree generated with this strategy is demonstrated in Figure 6a. Specifically, note how the search process found one new strategy (the first child node), and then kept trying to improve it, failing to do so across many iterations, finally exhausting all the rollouts budget.

To address this, GEPA employs a Pareto-based "illumination" strategy (Mouret & Clune, 2015), as shown in Algorithm 2. Specifically, GEPA identifies the highest score achieved for each individual training instance across all candidates in the pool, creating a "Pareto frontier" of scores achieved by the optimization process so far. GEPA then compiles a list of candidates that achieve the best score on at least one training task. This filters the pool down to candidates that incorporate "winning" strategies, preserving every valuable insight discovered in any reflective mutation. Next, GEPA prunes candidates that are strictly dominated: for instance, if Candidate 2 has the best score on Task 1 only, but Candidate 3 achieves that same best score on Task 1 and the best on Task 2, Candidate 2 is removed. Finally, GEPA stochastically samples a candidate from this pruned list, assigning higher selection probability to candidates that achieved the best score across more training instances.

In practice, this strategy helps GEPA escape local optima without expanding the search excessively. By focusing resources on promising candidates that have already demonstrated impactful, "winning" strategies, GEPA efficiently balances exploration and exploitation, allowing for continual improvement within the optimization budget.

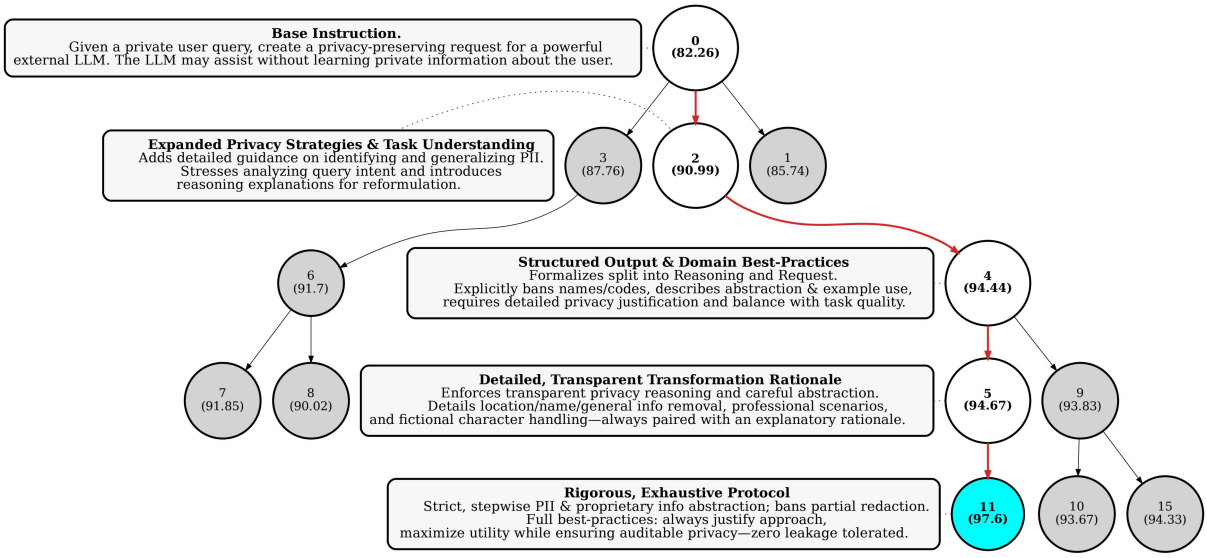



Figure 5: GEPA's reflective prompt mutation systematically incorporates task-specific nuances, leading to substantial improvements in performance. This figure visualizes the optimization trajectory taken by GEPA, presenting an annotated subtree from Figure 23d (for the privacy-preserving delegation task PUPA) to demonstrate the iterative enhancements made to the prompts. The progression from the base prompt (candidate 0) to the best performing prompt (candidate 11) is highlighted with red arrows, and key prompt changes at each step are annotated beside the corresponding nodes. Full-length instructions for these iterations are provided in Appendix G.1. Each prompt refinement in this trajectory adds targeted nuances informed by ongoing optimization, illustrating how GEPA's process accumulates lessons to continually boost task performance.

## 4 Evaluation Setup

In this section, we detail our experimental setup. For each benchmark, we adopt a standard three-way data split: train, validation, and test. The train split is fully accessible to the optimizers, allowing them to read and utilize the text and labels of the training instances for program tuning. Although optimizers may monitor the performance of candidate parameters (like model checkpoints) by tracking scores on the validation set (to implement early stopping, for example), direct access to the content of validation instances is restricted. The test set remains entirely held out and is inaccessible throughout the optimization process; performance on this split is only measured post-optimization to assess the performance of the optimized program.

### 4.1 Benchmarks, Reference compound AI systems, and Feedback Functions

To rigorously evaluate the performance of GEPA and and compare it against current state-of-the-art compound AI system optimizers, we assemble a diverse suite of benchmarks mostly obtained from Tan et al. (2025), each paired with available Compound AI Systems.

**HotpotQA** (Yang et al., 2018) is a large-scale question-answering dataset consisting of 113K Wikipedia-based question-answer pairs. It features questions that require reasoning over multiple supporting documents. We modify the last hop of the HoVerMultiHop program (described below) to answer the question instead of generating another query, and the rest of the system remains unmodified. The textual feedback module identifies the set of relevant documents remaining to be retrieved at each stage of the program, and provides that as feedback to the modules at that stage. We use 150 examples for training, 300 for validation, and 300 for testing.

**IFBench** (Pyatkin et al., 2025b) introduced a benchmark specifically designed to assess language models' ability to follow precise human instructions, especially output constraints (e.g., "answer only with yes or no", or "mention a word at least three times"). The IFBench test set consists of 58 new and out-of-distribution output constraints and instructions to test system's ability to generalize to new task constraints. Pyatkin et al. (2025b) also release IFTrain and IF-RLVR Train data (Pyatkin et al., 2025a) which are used for training. We split the IF-RLVR Train into our train/val

sets, and IFBench as our test set in order to ensure that the optimizers do not access the new, unseen constraints being tested in IFBench. We design a 2-stage system, that first attempts to answer the user query, and then in the second stage, rewrites the answer following the constraints. The textual feedback module provides the descriptions of constraints satsified and failed-to-be-satisifed by the system's response. Our splits contain 150 training examples, 300 for validation, and 294 for testing.

**HoVer** (Jiang et al., 2020) is an open-domain multihop fact extraction and claim verification benchmark built on a Wikipedia-based corpus requiring complex reasoning across multiple sentences and documents, typically involving multiple wikipedia articles. Following Tan et al. (2025), the systems are evaluated for their ability to write queries in multiple hops to retrieve all relevant wikipedia documents (gold documents) required to make the claim. We obtain the HoverMultiHop program from Tan et al. (2025), which performs up to 3-hop retrievals using 2 query writer modules, and 2 document summary modules. The textual feedback module simply identifies the set of correct documents retrieved, and the set of documents remaining to be retrieved, and returns them as feedback text. For HoVer, we use 150 examples for training, 300 for validation, and 300 for testing.

**PUPA** (Li et al., 2025a) propose the task of Privacy-Conscious Delegation: addressing real-world user queries using an ensemble of trusted and untrusted models. The core challenges are maintaining high response quality while minimizing leakage of personally identifiable information (PII) to untrusted models. Li et al. (2025a) also present PAPILLON, a compound AI system consisting of 2 modules, a user query rewriter and a response rewriter, run over the trusted model, along with an intermediate call to the untrusted model with the rewritten query. The feedback text simply provides the breakdown of the aggregate score, consisting of a response quality score and a PII leakage score. The dataset is split into 111 training examples, 111 for validation, and 221 for testing.

### 4.2 Models and Inference Parameters

We evaluate GEPA and baseline optimizers using two contemporary LLMs, chosen to represent both open-source and commercial model families. Each compound AI system is instantiated once per model, with all modules (e.g., retrievers, rewriters, answer generators) relying on the same model. All models are allowed a context window of upto 16384 tokens for inference.

**Qwen3 8B (Yang et al., 2025):** For our open-source experiments (including GRPO), we use `Qwen3-8B`. Following the recommended settings as per Team (2025), we use a decoding temperature of 0.6, top-p of 0.95, and top-k of 20 for training as well as inference.

**GPT-4.1 Mini (OpenAI, 2025):** For comparison with large commercial models, we use `GPT-4.1 mini` (`openai/gpt-4.1-mini-2025-04-14`) accessed via the OpenAI API with a model temperature of 1.0.

### 4.3 Optimizers

**Baseline:** The base program is directly evaluated without any further optimization applied.

**MIPROv2 (Opsahl-Ong et al., 2024):** MIPROv2 is a widely used compound AI system prompt optimizer and has been integrated into the DSPy (Khattab et al., 2024) and llama-prompt-ops (AI, 2025) frameworks. It works by jointly optimizing both instructions and demonstrations using Bayesian optimization. For each program module, it first bootstraps candidate sets of instructions and demonstrations, assigning uniform priors over their utilities. Candidate assignments are proposed with the Tree-Structured Parzen Estimator (TPE), and the Bayesian model is updated based on evaluation scores to favor high-performing candidates. The most probable sets of instructions and demonstrations are then selected and validated to obtain the final optimized program configuration. In order to

All MIPROv2 optimization runs are performed with the $auto = heavy$ setting, which corresponds to proposing 18 instruction candidates and 18 bootstrapped few-shot sets. Hence, across benchmarks, the exact number of rollouts varies depending on the number of trials it takes to bootstrap examples (finding 18 successful solution instances), the required number of Bayesian search steps (determined by the number of modules in the system), and size of the valset. Overall, MIPROv2's rollouts ranged from a minimum of 2270 (for PUPA) to maximum of 6926 (for HoVer).

**GRPO (Shao et al., 2024):** Group Relative Policy Optimization (GRPO) is a reinforcement learning algorithm that estimates advantages in a group-relative manner. We use the GRPO implementation for compound AI systems provided and open-sourced by Ziems, Soylu, and Agrawal et al. (2025) to perform our experiments. Across all training runs, each training step uses a group size of 12, with 4 training instances per step (total batch size 48, with per device train batch size 1). Training employs LoRA (Hu et al., 2022) with rank dimension 16, $\alpha = 64$, and dropout 0.05, using bf16 precision targeting the projection modules $[\mathrm{q}, \mathrm{k}, \mathrm{v}, \mathrm{o}, \mathrm{up}, \mathrm{down}, \mathrm{gate}]$. We use a learning rate of $1 \times 10^{-5}$, $\beta = 0.01$, reward scale normalization, and gradient norm clipping of 0.1. Gradients are accumulated for 20 steps before each

update, with a "constant with warmup learning" rate scheduler. Non-reentrant gradient checkpointing is enabled to further reduce memory usage. We manually explore several values for [LR, beta, norm clipping] hyperparameters. All GRPO optimizations run for 500 training steps, amounting to fixed 24,000 rollouts, with validation performed every 20 training steps, which is used to implement early stopping. All training experiments are performed on 1xH100/A100 (80 GB memory) with separate GPUs for inference rollouts.

**GEPA:** GEPA is our optimizer, based on the algorithm described in Section 3. We evaluate 2 variants of our main optimizer GEPA: **GEPA** and **GEPA+Merge**, along with 2 ablations created by replacing the Pareto-based sampling strategy with a naive, SelectBestCandidate strategy (SelectBestCandidate and SelectBestCandidate+Merge). All GEPA optimization runs use a minibatch size of 3, and merge is invoked a maximum of 5 times during the optimization run, when enabled. To ensure a fair comparison with MIPROv2, we align the computational budget between GEPA and MIPROv2 on a per-benchmark basis. The training set from each benchmark is used as $D_{feedback}$ (which is used to derive the training signals, as discussed in Section 3) and the validation set is used as $D_{pareto}$. Specifically, since MIPROv2's total rollout budget depends on factors such as validation set size and the number of modules, we first record the number of rollouts expended by MIPROv2 for each benchmark, and then cap GEPA's optimization to match this rollout budget. While differences in proposal and validation procedures cause the exact budget usage by the systems to be slightly different, the discrepancy is always within 10.15%. This protocol ensures that any performance differences arise from the optimization algorithms themselves, rather than from differences in search budget. The exact rollout countts for each optimizer is visualized in Appendix C.

## 5 Results and Analysis

Table 1: Benchmark results for different optimizers over Qwen3 8B and GPT-4.1 Mini models across multiple tasks.

| Model | HotpotQA | IFBench | Hover | PUPA | Aggregate | Improvement |
|---|---|---|---|---|---|---|
| **Qwen3-8B** | | | | | | |
| Baseline | 42.33 | 36.90 | 35.33 | 80.82 | 48.85 | — |
| MIPROv2 | 55.33 | 36.22 | 47.33 | 81.55 | 55.11 | +6.26 |
| GRPO | 43.33 | 35.88 | 38.67 | 86.66 | 51.14 | +2.29 |
| GEPA | 62.33 | **38.61** | **52.33** | **91.85** | **61.28** | **+12.44** |
| GEPA+Merge | **64.33** | 28.23 | 51.67 | 86.26 | 57.62 | +8.78 |
| **GPT-4.1 mini** | | | | | | |
| Baseline | 38.00 | 47.79 | 46.33 | 78.57 | 52.67 | — |
| MIPROv2 | 58.00 | 49.15 | 48.33 | 83.37 | 59.71 | +7.04 |
| GEPA | **69.00** | 52.72 | 51.67 | 94.47 | 66.97 | +14.29 |
| GEPA+Merge | 65.67 | **55.95** | **56.67** | **96.46** | **68.69** | **+16.02** |

Table 1 and Figure 9 summarize our main results, from which we derive the following observations:

**Observation 1: Reflective Prompt Evolution is highly sample-efficient and can outperform weight-space reinforcement learning:** Across all four benchmarks, GEPA demonstrates rapid adaptation and robust generalization in compound AI systems—outperforming GRPO (24,000 rollouts with LoRA) by up to 19% while using up to 35× fewer rollouts.

GEPA attains optimal test set performance on HotpotQA, IFBench, HoVer, and PUPA with only 6,438, 678 (35× fewer), 6,858, and 2,157 (11× fewer) rollouts, respectively—surpassing GRPO by 19%, 2.73%, 13.66%, and 5.19% on these tasks. Notably, GEPA matches GRPO's best validation scores after only 402, 330, 1179, and 306 rollouts respectively, achieving up to 78× greater sample efficiency. Furthermore, the combined GEPA+Merge approach outperforms GRPO by an even wider margin of 21% at a comparable rollout budget as GEPA. We especially highlight the +8.16% achieved by GEPA+Merge on IFBench with GPT-4.1 mini, even though it contains new, completely out-of-domain constraints in the test set.

It is also important to note that the majority of GEPA's counted rollouts are allocated to the validation set, where scores are utilized solely for candidate selection and not for producing learning signals. If we restrict the analysis to train set rollouts—the rollouts actually used for learning—GEPA requires just 737, 79, 558, and 269 training rollouts to reach optimal performance on HotpotQA, IFBench, HoVer, and PUPA, respectively. To match GRPO's best validation scores, GEPA achieves this with only 102, 32, 6, and 179 train rollouts, underscoring the high sample efficiency of learning based on reflective prompt evolution.

Since tracking candidates' validation performance accounts for the majority of GEPA's rollout budget, sample efficiency could be further improved by evaluating on a smaller validation set or by tracking scores on dynamically selected validation subsets instead of the full set—both of which we propose as directions for future work.

Figures 1a, 11c, 1b and 13c show the full performance-vs-rollouts curve for all optimizers over benchmarks HotpotQA, IFBench, HoVer and PUPA, respectively.

**Observation 2: Reflective prompt evolution enables Instruction-Optimization alone to outperform joint Instruction and Few-Shot Optimization:** We compare GEPA with MIPROv2—a state-of-the-art joint instruction and few-shot optimizer—using two leading models (GPT-4.1 mini and Qwen3 8B) across four diverse tasks. Our experiments show that GEPA consistently outperforms MIPROv2 in all settings, achieving margins as high as 11.1% for GPT-4.1 mini and 10.3% for Qwen3 8B. Furthermore, GEPA and GEPA+Merge more than double the aggregate gains over baseline seen with MIPROv2 across all benchmarks and both models (+16.02% and +14.29% vs +7.04% for MIPROv2).

While prior works such as Opsahl-Ong et al. (2024) and Wan et al. (2024) have provided compelling evidence for the effectiveness of few-shot example optimization—often outperforming instruction-based approaches—our findings suggest an exciting shift in this trend. We attribute this primarily to recent advances in the instruction-following and self-reflective abilities of LLMs, as well as the design choices in GEPA that capitalize on these improved capabilities. To further contextualize our findings, we redo the study on *generalization gap* (the difference between validation and test set performance for optimized prompts) as proposed by Wan et al. (2024). The results presented in figure 14 reinforce these observations: reflectively evolved instructions now demonstrate a lower generalization gap, underscoring both advancements in model capabilities and the benefits of GEPA's design. We see this as a reflection of the continuous evolution of LLMs and GEPA's ability to effectively leverage these improvements.

Finally, we provide the full-length optimized prompts produced by GEPA for all systems, benchmarks, and models in Appendix I, alongside the corresponding MIPROv2 prompts. Notably, in contrast to prior findings where instruction optimization yielded improvements primarily through quasi-exemplars (Wan et al., 2024), GEPA's prompts frequently contain detailed *declarative* instructions for completing the task, as illustrated in Figure 2.

**Observation 3: The next-candidate selection strategy strongly influences the optimization trajectory and final performance, with Pareto-based sampling providing a distinct advantage.** GEPA seeks to iteratively refine prompts by leveraging feedback from new rollouts. In order to test the impact of our Pareto-based candidate selection strategy, we consider a straightforward baseline for instantiating `SelectCandidate` strategy: always selecting the currently best-performing candidate. As shown by the ablation results in Table 2, this approach often leads to suboptimal exploration of the prompt search space ultimately leading to poor performance—GEPA with Pareto-based sampling strategy outperforms the SelectBestCandidate strategy by as much as 8.17%, maintaining an aggregate margin of +6.4% across all benchmarks. Figure 6 illustrates the stark difference in optimization trajectories between this naïve strategy and our proposed Pareto-based sampling-strategy. Always choosing the current best candidate tends to yield immediate improvement in the next iteration, but then causes the optimizer to stall, expending its entire rollout budget attempting to further improve this specific candidate. In contrast, our Pareto-based sampling method expands the search by considering all Pareto-optimal candidates (representing all the "winning" strategies discovered so far), ensuring a tight balance between exploration and exploitation tradeoffs—ultimately converging to a higher-performing solution within the same rollout budget.

Table 2: Benchmark results for different optimizers over Qwen3 8B and GPT-4.1 Mini models across multiple tasks.

| Model | HotpotQA | IFBench | Hover | PUPA | Aggregate | Improvement |
|---|---|---|---|---|---|---|
| **Qwen3-8B** | | | | | | |
| SelectBestCandidate | 58.33 | 30.44 | 45.33 | 85.45 | 54.89 | – |
| GEPA | **62.33** | **38.61** | **52.33** | **91.85** | **61.28** | **+6.4** |

**Observation 4: Instruction-optimized prompts are computationally cheaper and generalize better than few-shot demonstration prompts:** In addition to their strong generalization capabilities, reflectively evolved instructions offer a significant practical advantage: they are often much shorter and thus computationally more efficient than few-shot demonstration prompts. This advantage becomes especially clear for complex tasks, where even a single few-shot demonstration can be prohibitively long. The problem is further exacerbated when few-shot examples are optimized using state-of-the-art methods such as MIPROv2, which jointly optimizes *multiple* demonstrations to be used simultaneously, further increasing prompt length.