# Code export from: ${FOLDER}

## src/gepa/__init__.py

```
# Copyright (c) 2025 Lakshya A Agrawal and the GEPA contributors
# https://github.com/gepa-ai/gepa

from .adapters import default_adapter
from .api import optimize
from .core.adapter import EvaluationBatch, GEPAAdapter
from .core.result import GEPAResult
from .examples import aime
from .utils.stop_condition import (
    CompositeStopper,
    FileStopper,
    MaxMetricCallsStopper,
    NoImprovementStopper,
    ScoreThresholdStopper,
    SignalStopper,
    StopperProtocol,
    TimeoutStopCondition,
)
```

## src/gepa/adapters/README.md

```
# GEPA Adapters

> GEPA 🤝 Any Framework

This directory provides the interface to allow GEPA to plug into systems and frameworks of your choice! GEPA can interface with any system consisting of text components, by implementing `GEPAAdapter` in [../core/adapter.py](../core/adapter.py).

Currently, GEPA has the following adapters:
- [DSPy Adapter](./dspy_adapter/): This adapter integrates GEPA into [DSPy](https://dspy.ai/), to allow it to optimize any DSPy module's signature instructions.
- [Default Adapter](./default_adapter/): This adapter integrates GEPA into a single-turn LLM environment, where the task is specified as a user message, and an answer string must be present in the assistant response. GEPA optimizes the system prompt.
- [AnyMaths Adapter](./anymaths_adapter/): This adapter integrates GEPA with litellm and ollama to solve single-turn mathematical problems.

If there are any frameworks you would like GEPA integrated into, please create an issue or PR!
```

## src/gepa/adapters/__init__.py

## src/gepa/adapters/anymaths_adapter/README.md

````
# ⭐ AnyMaths: GEPA Adapter for Solving Math Word Problems ⭐
**AnyMaths Adapter** is a GEPA Adapter for any dataset that contains math word problems of varying complexity and structure. It is designed to handle a wide range of mathematical tasks, including arithmetic, algebra, reasoning, and more.

---

### 🔭 Auxiliary requirements
Some auxiliary requirements for using the AnyMaths Adapter are found in `src/gepa/adapters/anymaths_adapter/requirements.txt`. Simply install the required packages listed in that file via:
* If using `uv`:
    ```bash
    uv pip install -r src/gepa/adapters/anymaths_adapter/requirements.txt
    ```
* If using `pip`:
    ```bash
    pip install -r src/gepa/adapters/anymaths_adapter/requirements.txt
    ```

### ✍️ Preparing the dataset
In `src/gepa/examples/anymaths-bench/train_anymaths.py`, a sample function to prepare a dataset is provided via `init_dataset`. This function demonstrates how to load and preprocess the dataset for training and evaluation. Notably, it includes steps for data augmentation and splitting the dataset into training, validation, and test sets. We recommend to find and download datasets from [Hugging Face dataset hub](https://huggingface.co/datasets).

#### Best format for a custom dataset
If you have a custom dataset, it is best to follow the following schema:
```
{
    "question": ...,
    "solution": ...,
    "answer": ...
}
```
Remarks:
- `question` must be a string/text.
- `solution` must be a string/text.
- `answer` must be a purely numerical, no other text or units associated.

We recommend you to upload your custom dataset to the Hugging Face dataset hub to fully utilize `datasets.load_dataset`.

---

### 🧰 Adapter Design
The AnyMaths Adapter can work for any LiteLLM supported providers (e.g., OpenAI, Google Vertex, HuggingFace, Groq, vLLM, Ollama, etc.). For this instance, we opt to choose Ollama to show that **this adapter can work for local use if one has no access to expensive GPUs or paid APIs.** But, you may freely choose this adapter with any other LiteLLM-supported provider.

---

### ✍️ Preparing the seed prompt
The seed prompt is the initial instruction you provide to the base (target) model. It sets the context for the task at hand and this prompt evolves or changes over time toward maximizing the model's performance. The default failure score (i.e., score if the model outputs are incorrect or does not satisfy a set metric) is zero.

Set the seed prompt in a separate directory under `src/gepa/examples/anymaths-bench/prompt-templates`. Inside this directory is a file `instruction_prompt.txt` which contains the seed prompt.
````

````
---

### 🪞 Specifying the reflection LM
The reflection LM is a language model used to generate feedback based on the output by the base model. You can specify the reflection LM by setting the `reflection_lm` argument when calling the `gepa.optimize` function.

A sample `reflection_lm` function call can be found in `src/gepa/examples/anymaths-bench/train_anymaths.py`.

---

### 🤸 Running a full AnyMaths Adapter training
To run a full training session using the AnyMaths Adapter, you can use the following command:
```bash
python src/gepa/examples/anymaths-bench/train_anymaths.py --anymaths_dset_name ... --train_size ... --val_size ... --test_size ... --base_lm ... --use_api_base --api_base_url ... --reflection_lm ... --use_api_reflection --api_reflection_url ... --reflection_minibatch_size ... --budget ... --max_litellm_workers ... --seed ...
```
- `--anymaths_dset_name`: The Hugging Face `Dataset` name to use for training (e.g., `"openai/gsm8k"`, `"MathArena/aime_2025"`).
- `--train_size`: The size of the training set to use.
- `--val_size`: The size of the validation set to use.
- `--test_size`: The size of the test set to use.
- `--base_lm`: The base language model to use for GEPA training (e.g. `"ollama/qwen3:4b"`).
- `--use_api_base`: Enable this flag if you want to use the Ollama API for the base model. Otherwise, do not include this in your arguments if you are using provider APIs (e.g., OpenAI, Google Vertex, etc.).
- `--api_base_url`: (Base model) The URL to get completions for the base model. Example: Ollama uses the default `http://localhost:11434`. There is no need to set this up if you are using provider APIs. **Note: API keys and provider credentials must be set beforehand.**
- `--reflection_lm`: The reflection language model to generate feedback from base model outputs (e.g., `"ollama/qwen3:8b"`).
- `--use_api_reflection`: Similar with `--use_api_base`. Enable this flag if you want to use a specific endpoint to get completions from the reflection model.
- `--reflection_minibatch_size`: The minibatch size for the reflection LM to reflect against (default is 8).
- `--max_litellm_workers`: The maximum number of LiteLLM workers to use.
- `--budget`: The budget for the GEPA training (default is 500).
- `--seed`: The seed for the random number generator for reproducibility (default is 0).

#### 🔳 Training command examples

1. (Purely) **Using Ollama**:
    ```bash
    python src/gepa/examples/anymaths-bench/train_anymaths.py --anymaths_dset_name "openai/gsm8k" --train_size 50 --val_size 50 --test_size 50 --base_lm "ollama/qwen3:4b" --use_api_base --api_base_url "http://localhost:11434" --reflection_lm "ollama/qwen3:8b" --use_api_reflection --api_reflection_url "http://localhost:11434" --reflection_minibatch_size 8 --budget 500 --max_litellm_workers 4 --seed 0
    ```
2. (Purely) **Using Google Vertex** for Gemini users:
    ```bash
    python src/gepa/examples/anymaths-bench/train_anymaths.py --anymaths_dset_name "openai/gsm8k" --train_size 50 --val_size 50 --test_size 50 --base_lm "vertex_ai/gemini-2.5-flash-lite" --reflection_lm "vertex_ai/gemini-2.5-flash" --reflection_minibatch_size 8 --budget 500 --max_litellm_workers 4 --seed 0
    ```
3. **Using Google Vertex as base (target) LM, Ollama as reflection LM**:
    ```bash
    python src/gepa/examples/anymaths-bench/train_anymaths.py --anymaths_dset_name "openai/gsm8k" --train_size 50 --val_size 50 --test_size 50 --base_lm "vertex_ai/gemini-
````
````

````
2.5-flash-lite" --reflection_lm "ollama/qwen3:8b" --use_api_reflection --
api_reflection_url "http://localhost:11434" --reflection_minibatch_size 8 --budget 500 --
max_litellm_workers 4 --seed 0
    ```
4. **Using Ollama as base (target) LM, Google Vertex as reflection LM**:
    ```bash
    python src/gepa/examples/anymaths-bench/train_anymaths.py --anymaths_dset_name
"openai/gsm8k" --train_size 50 --val_size 50 --test_size 50 --base_lm "ollama/qwen3:4b" --
use_api_base --api_base_url "http://localhost:11434" --reflection_lm "vertex_ai/gemini-
2.5-flash" --reflection_minibatch_size 8 --budget 500 --max_litellm_workers 4 --seed 0
    ```

Once the training has completed, you may replace the optimal prompt found in
`src/gepa/examples/anymaths-bench/prompt-templates/optimal_prompt.txt`.

---

### 🔬 Model evaluation after GEPA training
`src/gepa/examples/anymaths-bench/eval_default.py` is used to perform model evaluation on
the test split. Feel free to modify this script to fit your custom evaluation scheme.
Example: `"openai/gsm8k"` - `test` is the dataset split used for benchmarking. The
evaluation scores will be displayed in the terminal once the evaluation has been
completed.

How to run the evaluation script:
1. **Using Ollama**:
    ```bash
    python src/gepa/examples/anymaths-bench/eval_default.py --anymaths_dset_name
"openai/gsm8k" --model "ollama/qwen3:4b" --use_api_url --api_url "http://localhost:11434"
--batch_size 8 --max_litellm_workers 4 --which_prompt "seed"
    ```
2. **Use Google Vertex** for Gemini users:
    ```bash
    python src/gepa/examples/anymaths-bench/eval_default.py --anymaths_dset_name
"openai/gsm8k" --model "vertex_ai/gemini-2.5-flash-lite" --batch_size 8 --
max_litellm_workers 4 --which_prompt "seed"
    ```
- `--anymaths_dset_name`: The name of the AnyMaths dataset to use for evaluation (default
is `"openai/gsm8k"`).
- `--model`: The model to evaluate (default is `"ollama/qwen3:4b"`).
- `--use_api_url`: Whether to use the API URL (default is `False`).
- `--api_url`: The API URL to use (default is `"http://localhost:11434"`).
- `--batch_size`: The batch size for evaluation (default is `8`).
- `--max_litellm_workers`: The maximum number of LiteLLM workers to use (default is `4`).
- `--which_prompt`: The prompt to use for evaluation (default is `"seed"`, choices are
`"seed"` and `"optimized"`).

**Note: The model that was used in GEPA training must also be the same model in performing
model evaluation.**

---

### 🧪 Experiments
| Dataset | Base LM | Reflection LM | Accuracy, % (Before GEPA) $\uparrow$ | Accuracy, %
(After GEPA) $\uparrow$ | GEPA Budget | Train-Val-Test Split Samples Used in GEPA
Optimization |
| ------- | ------- | ------------- | ------------------------ | ---------------------- | ---
--------- | ------ |
| `"openai/gsm8k"` | `"ollama/qwen3:4b"` | `"ollama/qwen3:8b"` | 18 | 23 (**+5**) | 500 |
50-50-50 |
| `"openai/gsm8k"` | `"vertex_ai/gemini-2.5-flash-lite"` | `"vertex_ai/gemini-2.5-flash"`
| 31 | 33 (**+2**) | 500 | 50-50-50 |
| `"openai/gsm8k"` | `"ollama/qwen3:0.6b"` | `"ollama/qwen3:8b"` | 7 | 5 (**-2**) | 500 |
50-50-50 |
| `"openai/gsm8k"` | `"ollama/gemma3:1b"` | `"ollama/gemma3:4b"` | 9 | 38 (**+29**) | 500
| 50-50-50 |
````
````

**Notice of WIP**: More tests will be done soon on other models (preferrably, small language models first).

---

### 🏛️ Prompt bank of optimal prompts

* Model: `"ollama/qwen3:4b"`, Dataset: `"openai/gsm8k"`, Budget: `500`:
    ```
    ### Task Instruction: Solve Multi-Step Mathematical Problems with Precision and Contextual Understanding

    You are tasked with solving problems that require careful parsing of contextual information, breaking down multi-step calculations, and ensuring accuracy in arithmetic and logical reasoning. Follow these steps to address diverse problem types (e.g., percentages, cost calculations, score determination, and distance computations):

    ---

    #### **1. Parse the Problem**
    - **Identify Key Values**: Extract numbers, percentages, fractions, and relationships (e.g., "40% of 60 students," "6 more than half of Ella\'s score").
    - **Understand Relationships**: Determine if values are additive, multiplicative, or comparative (e.g., "round trips" imply doubling one-way distances, "cost per item" requires multiplication).
    - **Clarify Ambiguities**: Resolve unclear phrasing (e.g., "half the score" refers to half the total items, not half the incorrect answers).

    ---

    #### **2. Break Down the Problem**
    - **Segment into Steps**: Divide the problem into smaller, manageable parts (e.g., calculate individual components before summing).
    - **Apply Formulas**: Use appropriate mathematical operations (e.g., percentage = part/whole × 100, total cost = (item count × price)).
    - **Account for Context**: Adjust calculations based on problem specifics (e.g., "round trip" requires doubling one-way distance, "score" may involve subtracting incorrect answers from total items).

    ---

    #### **3. Perform Calculations**
    - **Use Precise Arithmetic**:
        - For percentages: $ \\text{Percentage} \\times \\text{Total} $.
        - For fractions: $ \\frac{\\text{Numerator}}{\\text{Denominator}} \\times \\text{Value} $.
        - For multi-step operations: Follow order of operations (PEMDAS) and verify intermediate results.
    - **Avoid Common Errors**:
        - Misinterpreting phrases like "half the score" (e.g., half of total items, not half of incorrect answers).
        - Confusing "round trips" (up + down) with single trips.
        - Incorrectly applying percentages to the wrong base (e.g., 40% of students vs. 40% of total score).

    ---

    #### **4. Validate the Answer**
    - **Check Logical Consistency**: Ensure results align with problem constraints (e.g., total students = sum of groups, total cost = sum of individual costs).
    - **Verify Units and Formatting**: Confirm answers match required formats (e.g., boxed numbers, currency symbols, or percentage notation).
    - **Cross-Validate with Examples**: Compare calculations against similar problems (e.g., "If 40% of 60 students = 24, then 60 - 24 = 36").

    ---

    #### **5. Finalize the Response**
    ```

```
    - **Present the Answer Clearly**: Use the exact format requested (e.g., `\boxed{36}` for numerical answers, `$77.00` for currency).
    - **Include Step-by-Step Reasoning**: Explicitly show calculations (e.g., `18 * $2.50 = $45.00`, `8 * $4.00 = $32.00`).
    - **Highlight Key Decisions**: Note critical choices (e.g., "Half of Ella's score = 36 items / 2 = 18 items").

    ---

    ### **Examples of Problem Types**
    1. **Percentage Problems**:
        - *Input*: "40% of 60 students got below B."
        - *Solution*: $ 0.40 \\times 60 = 24 $, $ 60 - 24 = 36 $.
    2. **Cost Calculations**:
        - *Input*: "18 knobs at $2.50 each and 8 pulls at $4.00 each."
        - *Solution*: $ 18 \\times 2.50 + 8 \\times 4.00 = 45 + 32 = 77 $.
    3. **Score Determination**:
        - *Input*: "Ella got 4 incorrect answers; Marion got 6 more than half of Ella's score."
        - *Solution*: Total items = 40, Ella's correct = 36, half = 18, Marion = 18 + 6 = 24.

    ---

    ### **Key Niche Information**
    - **Percentages**: Always apply to the total (e.g., 40% of 60 students = 24 students, not 40% of 40 items).
    - **Round Trips**: Double one-way distances (e.g., 30,000 feet up + 30,000 feet down = 60,000 per trip).
    - **Score Calculations**: Subtract incorrect answers from total items (e.g., 40 items - 4 incorrect = 36 correct).
    - **Currency Formatting**: Use decimal points and symbols (e.g., `$77.00`, not `77`).

    ---

    ### **Final Output Format**
    Always conclude with:
        `Final Answer: \boxed{<result>}`
    For non-numeric answers, use:
        `Final Answer: <result>`

    Ensure calculations are explicitly shown and errors are corrected based on problem context.
    ```
* Model: `"vertex_ai/gemini-2.5-flash-lite"`, Dataset: `"openai/gsm8k"`, Budget: `500`:
    ```
    You are an AI assistant that solves mathematical word problems. You will be given a question and you need to provide a step-by-step solution to the problem. Finally, you will provide the answer to the question.

    When outputting the final answer, make sure there are no other text or explanations included, just the answer itself.

    The following fields are what you need to include in your response:
    - final_answer: The final answer to the question.
    - solution_pad: The step-by-step solution to the problem.

    Here are specific guidelines for generating your response:

    1.  **Understand the Problem Thoroughly:** Carefully read and analyze the word problem to ensure a complete understanding of all given information, constraints, and the specific question being asked. Pay close attention to units and how different quantities relate to each other.

    2.  **Formulate the Step-by-Step Solution (solution_pad):**
        *   Develop a clear, logical, and sequential step-by-step solution. Each step should be a distinct operation or deduction required to move closer to the final answer.
        *   Clearly state what is being calculated or determined in each step.
```

```
        *   Perform all necessary calculations with high precision and accuracy. Double-check all numerical operations (addition, subtraction, multiplication, division, etc.) to prevent errors.
        *   If the problem involves converting between different forms of a quantity (e.g., converting a monetary value into a count of items, or time units), explicitly show this conversion as a step.
            *   **Domain-Specific Interpretation Example:** If Barry has "$10.00 worth of dimes", first convert this value to the number of dimes (since a dime is $0.10, Barry has $10.00 / $0.10 = 100 dimes). If the problem then states Dan has "half that amount" and asks for the number of dimes Dan has, interpret "half that amount" as half the *number* of dimes Barry has (100 dimes / 2 = 50 dimes), rather than half the monetary value. Always aim for the most logical interpretation that leads to the requested unit in the final answer.
        *   The `solution_pad` field must *only* contain the clean, direct step-by-step solution. Do not include any internal monologues, self-corrections, re-evaluations, alternative thought processes, or debugging notes within this field.

    3.  **Calculate and Output the Final Answer:**
        *   Based on your thoroughly computed step-by-step solution, determine the exact numerical answer to the question.
        *   The `final_answer` field must contain *only* the numerical value. Do not include any currency symbols (e.g., "$"), units (e.g., "dimes", "hours"), or any other descriptive text or explanation in this field. For example, if the answer is 4625 dollars, output `4625`. If the answer is 52 dimes, output `52`.
        *   Ensure the final answer numerically matches the result of your `solution_pad` calculations.'
    ```
* Model: `"ollama/qwen3:0.6b"`, Dataset: `"openai/gsm8k"`, Budget: `500`:
    ```
    ### Instruction for Solving Math Word Problems

    **Task Description:**
    Solve multi-step math word problems by carefully analyzing the relationships between quantities, translating them into mathematical expressions, and performing accurate calculations. Ensure all components of the problem are addressed, and verify that percentages, fractions, and arithmetic operations are applied correctly.

    **Key Requirements:**
    1. **Parse Relationships:**
    - Identify explicit and implicit relationships (e.g., "20 fewer than," "1/10 less," "10 more than").
    - Define variables clearly (e.g., let Arts = x, then Maths = x - 20).

    2. **Translate to Equations:**
    - Convert word-based relationships into algebraic expressions or equations.
    - For percentage changes, apply the correct formula (e.g., "1/10 less" means 90% of the original value).

    3. **Account for All Components:**
    - Ensure all subjects, quantities, or data points mentioned in the problem are included in the final calculation.
    - Aggregate totals by summing individual values (e.g., total marks = sum of all subject scores).

    4. **Verify Arithmetic and Logic:**
    - Check for arithmetic errors (e.g., 1/10 of 70 = 7, not 70 - 7 = 63).
    - Validate that the solution aligns with the problem's constraints (e.g., no negative scores unless explicitly allowed).

    5. **Document Step-by-Step Reasoning:**
    - Break down the problem into logical steps, explicitly showing calculations (e.g., total birds = sum of daily totals).
    - Use parentheses and order of operations to avoid errors (e.g., 5 sites × 7 birds/site = 35 birds).

    6. **Final Answer Validation:**
    - Ensure the final answer matches the problem's question (e.g., total marks, average per site, or time difference).
```

```
    - Recheck all steps to confirm consistency with the problem’s context.

    **Example Application:**
    For a problem like:
    *"Amaya scored 20 fewer in Maths than Arts. She scored 10 more in Social Studies than Music. If she scored 70 in Music and 1/10 less in Maths, what is the total marks?"*
    - Define variables: Arts = x, Maths = x - 20, Social Studies = 70 + 10 = 80.
    - Calculate Maths: 1/10 less than Arts → Maths = x - 20 = 0.9x.
    - Solve for x: x - 20 = 0.9x → x = 200 (Arts), Maths = 180.
    - Total marks = 70 (Music) + 180 (Maths) + 200 (Arts) + 80 (Social Studies) = **296**.

    **Error Prevention:**
    - Avoid misinterpreting "1/10 less" as 1/10 of the value (instead, it means 90% of the original).
    - Ensure all subjects are included (e.g., Music, Maths, Arts, Social Studies in the example).
    - Double-check totals by summing individual components.
    ```
* Mode: `"ollama/gemma3:1b"`, Dataset: `"openai/gsm8k"`, Budget: `500`:
    ```
    You are a specialized assistant designed to solve complex, realistic word problems, prioritizing accuracy and detailed reasoning. Your primary goal is to meticulously determine the numerical answer while demonstrating a thorough understanding of the problem's context, constraints, and relevant domain knowledge. You will be provided with problem descriptions that often include specific units, quantities, and contextual details, aiming for scenarios mirroring real-world applications like logistics, resource management, and engineering calculations.

    **Here's your process:**

    1.  **Deconstruction & Contextual Understanding:** Carefully read the entire problem description. Identify *all* numerical values, units (e.g., kg, mph, minutes, dollars, meters, seconds), and relevant contextual details. Pay close attention to *all* constraints or limitations mentioned (e.g., “cannot exceed,” “up to,” “at most,” “within a tolerance of”).  Crucially, identify the *assumptions* embedded within the problem — what is being taken for granted?  For instance, is the problem implicitly stating that surfaces are flat, or that materials behave ideally?

    2.  **Unit Analysis and Conversion:**  Recognize that the problem fundamentally involves numerical calculations. However, rigorous unit conversion is paramount. Ensure
    all calculations are performed with appropriate units. If units are mixed, perform conversions *accurately*. Specifically, you must be able to handle conversions between common units: kilograms (kg), miles per hour (mph), minutes, dollars, meters, seconds, Newtons (N), cubic meters (m³), and other quantities.  **Important:** Weight and mass are frequently confused.  Remember that weight is a force (typically measured in Newtons — N), while mass is a measure of matter (typically measured in kilograms — kg). Always consider the context when given a weight value — it *usually* implies the force due to gravity.

    3.  **Strategic Approach Selection:** For most problems, a straightforward algebraic approach will be effective.  Setting up equations based on the given information is usually the most efficient method.  However, consider scenarios where a simpler calculation (e.g., direct multiplication or division) is sufficient.  Furthermore, recognize that some problems may benefit from applying principles of physics (e.g., Newton’s Laws of Motion, conservation of energy) to derive equations.

    4.  **Step-by-Step Calculation with Intermediate Results:** Clearly articulate *each* step of your calculation, showing all intermediate results.  This is absolutely crucial for verification, debugging, and demonstrating your thought process. Include units with every calculation.

    5.  **Final Answer with Units and Precision:** Provide the final numerical answer, *always* including the correct units.  Pay attention to the level of precision required by the problem — often, rounding will be necessary.

    6.  **Critical Considerations — Domain-Specific Knowledge is Key:**

        *   **Weight & Mass:** *Master* the distinction between weight and mass.  Weight is force (N), mass is matter (kg).
```

```
        *   **Velocity & Distance:** Remember the relationship: distance = velocity * time. Be mindful of velocity as a vector (magnitude and direction).
        *   **Area & Volume:**  Be proficient with basic geometric formulas (rectangles, cubes, cylinders, spheres - relevant formulas will be provided in the problem).
        *   **Cargo Loading:**  Problems frequently involve loading crates, where the maximum weight capacity of a crate (typically 20 kg) is a critical constraint. Consider the impact of sub-optimal loading arrangements.
        *   **Fluid Mechanics (where applicable):**  Some problems may involve fluid flow, requiring knowledge of concepts like pressure, viscosity, and flow rate.
        *   **Energy & Work:** Be familiar with the concepts of work, potential energy, kinetic energy, and their relationships.
        *   **Linear Motion:** Understand concepts such as acceleration, displacement, and average speed.
        *   **Rotational Motion:** When problems involve rotating objects, understanding angular velocity, angular acceleration, torque, and moment of inertia are essential.

    7.  **Verification and Validation:**  After arriving at a solution, briefly outline *how* you verified your answer. Did you check your units?  Did you perform a sanity check (e.g., is the answer physically plausible)?

    8. **Error Handling:** If a problem is ambiguous or contains conflicting information, state your assumptions and explain how they affect your solution.
    ```

---

### 🔍 Observations on the structure of the derived optimal prompt
- For small language models:
    * Goal-oriented: The prompt starts by clearly stating the overall task, which is to solve multi-step math problems with precision.
    * Chain-of-Thought: It breaks down the problem-solving process into a detailed, numbered sequence of five steps: **Parse**, **Break Down**, **Calculate**, **Validate**, and **Finalize**.
    * Instruction Detail: Each step includes specific instructions on how to perform the task, such as identifying key values, applying formulas, and avoiding common errors.
    * Few-shot Learning: The prompt provides concrete **examples** of different problem types (percentage, costs, scores) to show the model how to apply the instructions.
    * Knowledge Base: It includes *key niche information* section that acts as a mini-rulebook, highlighting specific details and common pitfalls like "round trips" and currency formatting.
    * Structured Output: The prompt ends by defining a strict **final output format** to ensure the model's answer is consistent and easy to read.
    * Contextual Awareness: The prompt encourages the model to consider the broader context of the problem, including any implicit assumptions or constraints that may not be explicitly stated. Filed under *emergent* behavior.

- For provider models:
    * Fewer tokens: The prompt is more concise, using fewer tokens to convey the same information, which can lead to faster processing and lower costs.
    * Straightforward: Main instruction and output format are placed at the first parts of the prompt. Detailed guidelines are provided in a structured manner to facilitate understanding after the main instruction and output format.

### ❓ Weird results and observations
- In the case of `"ollama/qwen3:0.6b"`, the score did not improve as expected with additional context. Moreover, in the optimal prompt, the specific instruction to generate the expected JSON object was removed. In a similar fashion, we can also observe this omission in the optimal prompt for `"ollama/qwen3:4b"`.

---

### 👨‍🔬 Contributor
This adapter was contributed by **Emmanuel G. Maminta**. [[LinkedIn]](https://linkedin.com/in/egmaminta) [[GitHub]](https://github.com/egmaminta)

## src/gepa/adapters/anymaths_adapter/__init__.py

```
from .anymaths_adapter import AnyMathsAdapter
```

## src/gepa/adapters/anymaths_adapter/anymaths_adapter.py

```
from typing import Any, TypedDict

import litellm
from pydantic import BaseModel, Field

from gepa.core.adapter import EvaluationBatch, GEPAAdapter


class AnyMathsDataInst(TypedDict):
    input: str
    additional_context: dict[str, str]
    answer: str


class AnyMathsTrajectory(TypedDict):
    data: AnyMathsDataInst
    full_assistant_response: str


class AnyMathsRolloutOutput(TypedDict):
    full_assistant_response: str


class AnyMathsStructuredOutput(BaseModel):
    final_answer: str = Field(
        ..., description="The final answer to the mathematical problem (i.e., no units, no
other text)"
    )
    solution_pad: str = Field(..., description="The solution pad containing the step-by-
step solution to the problem.")


class AnyMathsAdapter(GEPAAdapter[AnyMathsDataInst, AnyMathsTrajectory,
AnyMathsRolloutOutput]):
    """AnyMaths Adapter is a GEPAAdapter for any dataset that contains mathematical word
problems
    of varying complexity and structure. It is designed to handle a wide range of
mathematical
    tasks, including arithmetic, algebra, and more.

    Note: Ollama must be installed and configured to use this adapter.
    """

    def __init__(
        self,
        model: str,
        failure_score: float = 0.0,
        api_base: str | None = "http://localhost:11434",
        max_litellm_workers: int = 10,
    ) -> None:
        import litellm

        self.model = model
        self.failure_score = failure_score
        self.litellm = litellm
```

```
        self.max_litellm_workers = max_litellm_workers
        self.api_base = api_base

        if self.model.startswith("ollama"):
            assert self.api_base is not None, "API base URL must be provided when using
Ollama."

        if self.api_base is None or self.api_base == "":
            self.api_base = None

    def evaluate(
        self,
        batch: list[AnyMathsDataInst],
        candidate: dict[str, str],
        capture_traces: bool = False,
    ) -> EvaluationBatch[AnyMathsTrajectory, AnyMathsRolloutOutput]:
        import ast

        outputs: list[AnyMathsRolloutOutput] = []
        scores: list[float] = []
        trajectories: list[AnyMathsTrajectory] | None = [] if capture_traces else None

        if not candidate:
            raise ValueError("Candidate must contain at least one component text.")

        system_content = next(iter(candidate.values()))

        litellm_requests = []

        for data in batch:
            user_content = f"{data['input']}"

            messages = [
                {"role": "system", "content": system_content},
                {"role": "user", "content": user_content},
            ]

            litellm_requests.append(messages)

        try:
            responses = self.litellm.batch_completion(
                model=self.model,
                messages=litellm_requests,
                api_base=self.api_base,
                max_workers=self.max_litellm_workers,
                format=AnyMathsStructuredOutput.model_json_schema(),
                response_format={
                    "type": "json_object",
                    "response_schema": AnyMathsStructuredOutput.model_json_schema(),
                    "enforce_validation": True,
                },
            )
        except litellm.exceptions.JSONSchemaValidationError as e:
            raise e

        for data, response in zip(batch, responses, strict=False):
            correct_output_format = True
            try:
                assistant_response =
ast.literal_eval(response.choices[0].message.content.strip())
            except Exception:
                assistant_response = "Assistant failed to respond with the correct answer
or format."
                correct_output_format = False

            if correct_output_format:
                structured_assistant_response = f"Assistant's Solution:
{assistant_response['solution_pad']}\n"
```

```
                structured_assistant_response += f"Final Answer:
{assistant_response['final_answer']}"
                output = {"full_assistant_response": structured_assistant_response}
                score = 1.0 if data["answer"] in assistant_response["final_answer"] else
self.failure_score
            else:
                output = {"full_assistant_response": assistant_response}
                score = self.failure_score

            outputs.append(output)
            scores.append(score)

            if capture_traces:
                trajectories.append({"data": data, "full_assistant_response":
output["full_assistant_response"]})
        # Return results for the entire batch (not just the first item)
        return EvaluationBatch(outputs=outputs, scores=scores, trajectories=trajectories)

    def make_reflective_dataset(
        self,
        candidate: dict[str, str],
        eval_batch: EvaluationBatch[AnyMathsTrajectory, AnyMathsRolloutOutput],
        components_to_update: list[str],
    ) -> dict[str, list[dict[str, Any]]]:
        ret_d: dict[str, list[dict[str, Any]]] = {}

        assert len(components_to_update) == 1
        comp = components_to_update[0]

        items: list[dict[str, Any]] = []
        trace_instances = list(zip(eval_batch.trajectories, eval_batch.scores,
eval_batch.outputs, strict=False))

        for trace_instance in trace_instances:
            traj, score, _ = trace_instance
            data = traj["data"]
            generated_outputs = traj["full_assistant_response"]

            if score > 0.0:
                feedback = f"The generated response is correct. The final answer is:
{data['answer']}."
            else:
                additional_context_str = "\n".join(f"{k}: {v}" for k, v in
data["additional_context"].items())
                if additional_context_str:
                    feedback = (
                        f"The generated response is incorrect. The correct answer is:
{data['answer']}. "
                        "Ensure that the correct answer is included in the response
exactly as it is. "
                        f"Here is some additional context that might be
helpful:\n{additional_context_str}"
                    )
                else:
                    feedback = (
                        f"The generated response is incorrect. The correct answer is:
{data['answer']}. "
                        "Ensure that the correct answer is included in the response
exactly as it is."
                    )

            d = {"Inputs": data["input"], "Generated Outputs": generated_outputs,
"Feedback": feedback}

            items.append(d)

        ret_d[comp] = items
```

```
        if len(items) == 0:
            raise Exception("No valid predictions found for any module.")

        return ret_d
```

### src/gepa/adapters/default_adapter/README.md

### src/gepa/adapters/default_adapter/__init__.py

### src/gepa/adapters/default_adapter/default_adapter.py

```
# Copyright (c) 2025 Lakshya A Agrawal and the GEPA contributors
# https://github.com/gepa-ai/gepa

from collections.abc import Mapping, Sequence
from typing import Any, Protocol, TypedDict

from gepa.core.adapter import EvaluationBatch, GEPAAdapter


# DataInst, Trajectory, RolloutOutput
class DefaultDataInst(TypedDict):
    input: str
    additional_context: dict[str, str]
    answer: str


class DefaultTrajectory(TypedDict):
    data: DefaultDataInst
    full_assistant_response: str


class DefaultRolloutOutput(TypedDict):
    full_assistant_response: str


DefaultReflectiveRecord = TypedDict(
    "DefaultReflectiveRecord",
    {
        "Inputs": str,
        "Generated Outputs": str,
        "Feedback": str,
    },
)


class ChatMessage(TypedDict):
    role: str
    content: str


class ChatCompletionCallable(Protocol):
```

```
    def __call__(self, messages: Sequence[ChatMessage]) -> str: ...


class DefaultAdapter(GEPAAdapter[DefaultDataInst, DefaultTrajectory,
DefaultRolloutOutput]):
    def __init__(
        self,
        model: str | ChatCompletionCallable,
        failure_score: float = 0.0,
        max_litellm_workers: int = 10,
        litellm_batch_completion_kwargs: dict[str, Any] = {},
    ):
        if isinstance(model, str):
            import litellm

            self.litellm = litellm
        self.model = model

        self.failure_score = failure_score
        self.max_litellm_workers = max_litellm_workers
        self.litellm_batch_completion_kwargs = litellm_batch_completion_kwargs

    def evaluate(
        self,
        batch: list[DefaultDataInst],
        candidate: dict[str, str],
        capture_traces: bool = False,
    ) -> EvaluationBatch[DefaultTrajectory, DefaultRolloutOutput]:
        outputs: list[DefaultRolloutOutput] = []
        scores: list[float] = []
        trajectories: list[DefaultTrajectory] | None = [] if capture_traces else None

        system_content = next(iter(candidate.values()))

        litellm_requests = []

        for data in batch:
            user_content = f"{data['input']}"

            messages: list[ChatMessage] = [
                {"role": "system", "content": system_content},
                {"role": "user", "content": user_content},
            ]

            litellm_requests.append(messages)

        try:
            if isinstance(self.model, str):
                responses = [
                    resp.choices[0].message.content.strip()
                    for resp in self.litellm.batch_completion(
                        model=self.model, messages=litellm_requests,
max_workers=self.max_litellm_workers, **self.litellm_batch_completion_kwargs
                    )
                ]
            else:
                responses = [self.model(messages) for messages in litellm_requests]
        except Exception as e:
            raise e

        for data, assistant_response in zip(batch, responses, strict=False):
            output: DefaultRolloutOutput = {"full_assistant_response": assistant_response}
            score = 1.0 if data["answer"] in assistant_response else 0.0

            outputs.append(output)
            scores.append(score)

            if trajectories is not None:
```

```
                trajectories.append({"data": data, "full_assistant_response":
assistant_response})

        return EvaluationBatch(outputs=outputs, scores=scores, trajectories=trajectories)

    def make_reflective_dataset(
        self,
        candidate: dict[str, str],
        eval_batch: EvaluationBatch[DefaultTrajectory, DefaultRolloutOutput],
        components_to_update: list[str],
    ) -> Mapping[str, Sequence[Mapping[str, Any]]]:
        ret_d: dict[str, list[DefaultReflectiveRecord]] = {}

        assert len(components_to_update) == 1
        comp = components_to_update[0]

        trajectories = eval_batch.trajectories
        assert trajectories is not None, "Trajectories are required to build a reflective
dataset."

        items: list[DefaultReflectiveRecord] = []
        trace_instances = list(zip(trajectories, eval_batch.scores, eval_batch.outputs,
strict=False))

        for trace_instance in trace_instances:
            traj, score, _ = trace_instance
            data = traj["data"]
            generated_outputs = traj["full_assistant_response"]

            if score > 0.0:
                feedback = (
                    f"The generated response is correct. The response include the correct
answer '{data['answer']}'"
                )
            else:
                additional_context_str = "\n".join(f"{k}: {v}" for k, v in
data["additional_context"].items())
                feedback = f"The generated response is incorrect. The correct answer is
'{data['answer']}'. Ensure that the correct answer is included in the response exactly as
it is. Here is some additional context that might be helpful:\n{additional_context_str}"

            d: DefaultReflectiveRecord = {
                "Inputs": data["input"],
                "Generated Outputs": generated_outputs,
                "Feedback": feedback,
            }

            items.append(d)

        ret_d[comp] = items

        if len(items) == 0:
            raise Exception("No valid predictions found for any module.")

        return ret_d
```

## src/gepa/adapters/dspy_adapter/README.md

```
# DSPy <> GEPA

[dspy_adapter.py](dspy_adapter.py) provides an example adapter to allow GEPA to optimize
the signature instructions of any DSPy program. The most up-to-date version of this
adapter is live in the DSPy repository at [gepa_utils.py]
(https://github.com/stanfordnlp/dspy/blob/main/dspy/teleprompt/gepa/gepa_utils.py).
```

```
> The best way to use this adapter is from within DSPy itself.

Extensive tutorials on using GEPA with DSPy are available at [dspy.GEPA tutorials]
(https://dspy.ai/tutorials/gepa_ai_program/).
```

### src/gepa/adapters/dspy_adapter/__init__.py

### src/gepa/adapters/dspy_adapter/dspy_adapter.py

```
"""
This file provides an example adapter allowing GEPA to optimize text components of DSPy
programs (instructions and prompts).
The most up-to-date version of this file is in the DSPy repository:
https://github.com/stanfordnlp/dspy/blob/main/dspy/teleprompt/gepa/gepa_utils.py
"""

import logging
import random
from typing import Any, Callable, Protocol

from dspy.adapters.chat_adapter import ChatAdapter
from dspy.adapters.types import History
from dspy.evaluate import Evaluate
from dspy.primitives import Example, Prediction
from dspy.teleprompt.bootstrap_trace import TraceData

from gepa.core.adapter import EvaluationBatch, GEPAAdapter


class LoggerAdapter:
    def __init__(self, logger: logging.Logger):
        self.logger = logger

    def log(self, x: str):
        self.logger.info(x)


DSPyTrace = list[tuple[Any, dict[str, Any], Prediction]]


class ScoreWithFeedback(Prediction):
    score: float
    feedback: str


class PredictorFeedbackFn(Protocol):
    def __call__(
        self,
        predictor_output: dict[str, Any],
        predictor_inputs: dict[str, Any],
        module_inputs: Example,
        module_outputs: Prediction,
        captured_trace: DSPyTrace,
    ) -> ScoreWithFeedback:
        """
        This function is used to provide feedback to a specific predictor.
        The function is called with the following arguments:
```

```
        - predictor_output: The output of the predictor.
        - predictor_inputs: The inputs to the predictor.
        - module_inputs: The inputs to the whole program --- `Example`.
        - module_outputs: The outputs of the whole program --- `Prediction`.
        - captured_trace: The trace of the module's execution.
        # Shape of trace is: [predictor_invocation_idx -> Tuple[Predictor,
PredictorInputs, Prediction]]
        # Each trace is a tuple of (Predictor, PredictorInputs, Prediction)

        The function should return a `ScoreWithFeedback` object.
        The feedback is a string that is used to guide the evolution of the predictor.
        """
        ...


class DspyAdapter(GEPAAdapter[Example, TraceData, Prediction]):
    def __init__(
        self,
        student_module,
        metric_fn: Callable,
        feedback_map: dict[str, Callable],
        failure_score=0.0,
        num_threads: int | None = None,
        add_format_failure_as_feedback: bool = False,
        rng: random.Random | None = None,
    ):
        self.student = student_module
        self.metric_fn = metric_fn
        self.feedback_map = feedback_map
        self.failure_score = failure_score
        self.num_threads = num_threads
        self.add_format_failure_as_feedback = add_format_failure_as_feedback
        self.rng = rng or random.Random(0)

        # Cache predictor names/signatures
        self.named_predictors = list(self.student.named_predictors())

    def build_program(self, candidate: dict[str, str]):
        new_prog = self.student.deepcopy()
        for name, pred in new_prog.named_predictors():
            if name in candidate:
                pred.signature = pred.signature.with_instructions(candidate[name])
        return new_prog

    def evaluate(self, batch, candidate, capture_traces=False):
        program = self.build_program(candidate)

        if capture_traces:
            # bootstrap_trace_data-like flow with trace capture
            from dspy.teleprompt.bootstrap_trace import bootstrap_trace_data

            trajs = bootstrap_trace_data(
                program=program,
                dataset=batch,
                metric=self.metric_fn,
                num_threads=self.num_threads,
                raise_on_error=False,
                capture_failed_parses=True,
                failure_score=self.failure_score,
                format_failure_score=self.failure_score,
            )
            scores = []
            outputs = []
            for t in trajs:
                outputs.append(t["prediction"])
                if hasattr(t["prediction"], "__class__") and t.get("score") is None:
                    scores.append(self.failure_score)
                else:
```

```
                    score = t["score"]
                    if hasattr(score, "score"):
                        score = score["score"]
                    scores.append(score)
            return EvaluationBatch(outputs=outputs, scores=scores, trajectories=trajs)
        else:
            evaluator = Evaluate(
                devset=batch,
                metric=self.metric_fn,
                num_threads=self.num_threads,
                return_all_scores=True,
                return_outputs=True,
                failure_score=self.failure_score,
                provide_traceback=True,
                max_errors=len(batch) * 100,
            )
            res = evaluator(program)
            outputs = [r[1] for r in res.results]
            scores = [r[2] for r in res.results]
            scores = [s["score"] if hasattr(s, "score") else s for s in scores]
            return EvaluationBatch(outputs=outputs, scores=scores, trajectories=None)

    def make_reflective_dataset(self, candidate, eval_batch, components_to_update):
        from dspy.teleprompt.bootstrap_trace import FailedPrediction

        program = self.build_program(candidate)

        ret_d: dict[str, list[dict[str, Any]]] = {}
        for pred_name in components_to_update:
            module = None
            for name, m in program.named_predictors():
                if name == pred_name:
                    module = m
                    break
            assert module is not None

            items: list[dict[str, Any]] = []
            for data in eval_batch.trajectories or []:
                trace = data["trace"]
                example = data["example"]
                prediction = data["prediction"]
                module_score = data["score"]
                if hasattr(module_score, "score"):
                    module_score = module_score["score"]

                trace_instances = [t for t in trace if
t[0].signature.equals(module.signature)]
                if not self.add_format_failure_as_feedback:
                    trace_instances = [t for t in trace_instances if not isinstance(t[2],
FailedPrediction)]
                if len(trace_instances) == 0:
                    continue

                selected = None
                for t in trace_instances:
                    if isinstance(t[2], FailedPrediction):
                        selected = t
                        break

                if selected is None:
                    if isinstance(prediction, FailedPrediction):
                        continue
                    selected = self.rng.choice(trace_instances)

                inputs = selected[1]
                outputs = selected[2]

                new_inputs = {}
```

```
                new_outputs = {}

                contains_history = False
                history_key_name = None
                for input_key, input_val in inputs.items():
                    if isinstance(input_val, History):
                        contains_history = True
                        assert history_key_name is None
                        history_key_name = input_key

                if contains_history:
                    s = "```json\n"
                    for i, message in enumerate(inputs[history_key_name].messages):
                        s += f"  {i}: {message}\n"
                    s += "```"
                    new_inputs["Context"] = s

                for input_key, input_val in inputs.items():
                    if contains_history and input_key == history_key_name:
                        continue
                    new_inputs[input_key] = str(input_val)

                if isinstance(outputs, FailedPrediction):
                    s = "Couldn't parse the output as per the expected output format. The
model's raw response was:\n"
                    s += "```\n"
                    s += outputs.completion_text + "\n"
                    s += "```\n\n"
                    new_outputs = s
                else:
                    for output_key, output_val in outputs.items():
                        new_outputs[output_key] = str(output_val)

                d = {"Inputs": new_inputs, "Generated Outputs": new_outputs}
                if isinstance(outputs, FailedPrediction):
                    adapter = ChatAdapter()
                    structure_instruction = ""
                    for dd in adapter.format(module.signature, [], {}):
                        structure_instruction += dd["role"] + ": " + dd["content"] + "\n"
                    d["Feedback"] = "Your output failed to parse. Follow this
structure:\n" + structure_instruction
                    # d['score'] = self.failure_score
                else:
                    feedback_fn = self.feedback_map[pred_name]
                    fb = feedback_fn(
                        predictor_output=outputs,
                        predictor_inputs=inputs,
                        module_inputs=example,
                        module_outputs=prediction,
                        captured_trace=trace,
                    )
                    d["Feedback"] = fb["feedback"]
                    assert fb["score"] == module_score, (
                        f"Currently, GEPA only supports feedback functions that return the
same score as the module's score. However, the module-level score is {module_score} and
the feedback score is {fb.score}."
                    )
                    # d['score'] = fb.score
                items.append(d)

            if len(items) == 0:
                # raise Exception(f"No valid predictions found for module
{module.signature}.")
                continue
            ret_d[pred_name] = items

        if len(ret_d) == 0:
            raise Exception("No valid predictions found for any module.")
```

```
        return ret_d

    # TODO: The current DSPyAdapter implementation uses the GEPA default
propose_new_texts.
    # We can potentially override this, to use the instruction proposal similar to
MIPROv2.

    # def propose_new_texts(
    #     self,
    #     candidate: Dict[str, str],
    #     reflective_dataset: Dict[str, List[Dict[str, Any]]],
    #     components_to_update: List[str]
    # ) -> Dict[str, str]:
    #     if self.adapter.propose_new_texts is not None:
    #         return self.adapter.propose_new_texts(candidate, reflective_dataset,
components_to_update)

    #     from .instruction_proposal import InstructionProposalSignature
    #     new_texts: Dict[str, str] = {}
    #     for name in components_to_update:
    #         base_instruction = candidate[name]
    #         dataset_with_feedback = reflective_dataset[name]
    #         new_texts[name] = InstructionProposalSignature.run(
    #             lm=self.reflection_lm,
    #             input_dict={
    #                 "current_instruction_doc": base_instruction,
    #                 "dataset_with_feedback": dataset_with_feedback
    #             }
    #         )['new_instruction']
    #     return new_texts
```

## src/gepa/adapters/dspy_full_program_adapter/README.md

````
# DSPy Full Program Adapter

This adapter lets GEPA evolve entire DSPy programs—including signatures, modules, and
control flow—not just prompts or instructions.

## Usage

First, install DSPy (version 3.0.2 or higher) with: `pip install 'dspy>=3.0.2'`.

Now, let's use GEPA to generate a DSPy program to solve MATH (benchmark). We start with a
very very simple DSPy program `dspy.ChainOfThought("question -> answer")`:
```python
from gepa import optimize
from gepa.adapters.dspy_full_program_adapter.full_program_adapter import DspyAdapter
import dspy

# Standard DSPy metric function
def metric_fn(example, pred, trace=None):
    ...

# Start with a basic program. This code block must export a `program` that shows how the
task should be performed
seed_program = """import dspy
program = dspy.ChainOfThought("question -> answer")"""

# Run optimization
reflection_lm = dspy.LM(model="openai/gpt-4.1", max_tokens=32000) # <-- This LM will only
be used to propose new DSPy programs
result = optimize(
    seed_candidate={"program": seed_program},
````
````

```
    trainset=train_data,
    valset=val_data,
    adapter=DspyAdapter(
        task_lm=dspy.LM(model="openai/gpt-4.1-nano", max_tokens=32000), # <-- This LM will
be used for the downstream task
        metric_fn=metric_fn,
        reflection_lm=lambda x: reflection_lm(x)[0],
    ),
    max_metric_calls=2000,
)

# Get the evolved program
optimized_program_code = result.best_candidate["program"]
print(optimized_program_code)
```

Using dspy.ChainOfThought with GPT-4.1 Nano achieves a score of **67%**, while the following GEPA-optimized program boosts performance to **93%**!

```
import dspy
from typing import Optional

class MathQAReasoningSignature(dspy.Signature):
    """
    Solve the given math word problem step by step, showing all necessary reasoning and
calculations.
    - First, provide a clear, detailed, and logically ordered reasoning chain, using
equations and algebraic steps as needed.
    - Then, extract the final answer in the required format, strictly following these
rules:
        * If the answer should be a number, output only the number (no units, unless
explicitly requested).
        * If the answer should be an algebraic expression, output it in LaTeX math mode
(e.g., \frac{h^2}{m}).
        * Do not include explanatory text, units, or extra formatting in the answer field
unless the question explicitly requests it.
    Common pitfalls:
        - Including units when not required.
        - Restating the answer with extra words or formatting.
        - Failing to simplify expressions or extract the final answer.
    Edge cases:
        - If the answer is a sum or list, output only the final value(s) as required.
        - If the answer is an expression, ensure it is fully simplified.
    Successful strategies:
        - Use step-by-step algebraic manipulation.
        - Double-check the final answer for correct format and content.
    """
    question: str = dspy.InputField(desc="A math word problem to solve.")
    reasoning: str = dspy.OutputField(desc="Step-by-step solution, with equations and
logic.")
    answer: str = dspy.OutputField(desc="Final answer, strictly in the required format
(see instructions).")

class MathQAExtractSignature(dspy.Signature):
    """
    Given a math word problem and a detailed step-by-step solution, extract ONLY the final
answer in the required format.
    - If the answer should be a number, output only the number (no units, unless
explicitly requested).
    - If the answer should be an algebraic expression, output it in LaTeX math mode (e.g.,
\frac{h^2}{m}).
    - Do not include explanatory text, units, or extra formatting in the answer field
unless the question explicitly requests it.
    - If the answer is a sum or list, output only the final value(s) as required.
    """
    question: str = dspy.InputField(desc="The original math word problem.")
    reasoning: str = dspy.InputField(desc="A detailed, step-by-step solution to the
problem.")
```

```
    answer: str = dspy.OutputField(desc="Final answer, strictly in the required format.")

class MathQAModule(dspy.Module):
    def __init__(self):
        super().__init__()
        self.reasoner = dspy.ChainOfThought(MathQAReasoningSignature)
        self.extractor = dspy.Predict(MathQAExtractSignature)

    def forward(self, question: str):
        reasoning_pred = self.reasoner(question=question)
        extract_pred = self.extractor(question=question,
reasoning=reasoning_pred.reasoning)
        return dspy.Prediction(
            reasoning=reasoning_pred.reasoning,
            answer=extract_pred.answer
        )

program = MathQAModule()
```

A fully executable notebook to run this example is in
[src/gepa/examples/dspy_full_program_evolution/example.ipynb]
(../../examples/dspy_full_program_evolution/example.ipynb)

**src/gepa/adapters/dspy_full_program_adapter/dspy_program_proposal_signature.py**

```
# Copyright (c) 2025 Lakshya A Agrawal and the GEPA contributors
# https://github.com/gepa-ai/gepa

from typing import Any, ClassVar

import yaml

from gepa.proposer.reflective_mutation.base import Signature


class DSPyProgramProposalSignature(Signature):
    prompt_template = """I am trying to solve a task using the DSPy framework. Here's a
comprehensive overview of DSPy concepts to guide your improvements:

Signatures:
- Signatures define tasks declaratively through input/output fields and explicit
instructions.
- They serve as blueprints for what the LM needs to accomplish.

Signature Types:
- Simple signatures: Specified as strings like "input1, ..., inputN -> output1, ...,
outputM" (e.g., "topic -> tweet").
- Typed signatures: Create a subclass of dspy.Signature with a detailed docstring that
includes task instructions, common pitfalls, edge cases, and successful strategies. Define
fields using dspy.InputField(desc="...", type=...) and dspy.OutputField(desc="...",
type=...) with pydantic types such as str, List[str], Literal["option1", "option2"], or
custom classes.

Modules:
- Modules specify __how__ to solve the task defined by a signature.
- They are composable units inspired by PyTorch layers, using language models to process
inputs and produce outputs.
- Inputs are provided as keyword arguments matching the signature's input fields.
- Outputs are returned as dspy.Prediction objects containing the signature's output
fields.
- Key built-in modules:
```

```
  - dspy.Predict(signature): Performs a single LM call to directly generate the outputs
from the inputs.
  - dspy.ChainOfThought(signature): Performs a single LM call that first generates a
reasoning chain, then the outputs (adds a 'reasoning' field to the prediction).
  - Other options: dspy.ReAct(signature) for reasoning and acting, or custom chains.
- Custom modules: Subclass dspy.Module. In __init__, compose sub-modules (e.g., other
Predict or ChainOfThought instances). In forward(self, **kwargs), define the data flow:
call sub-modules, execute Python logic if needed, and return dspy.Prediction with the
output fields.

Example Usage:
```
# Simple signature
simple_signature = "question -> answer"

# Typed signature
class ComplexSignature(dspy.Signature):
    \"\"\"
    <Detailed instructions for completing the task: Include steps, common pitfalls, edge
cases, successful strategies. Include domain knowledge...>
    \"\"\"
    question: str = dspy.InputField(desc="The question to answer")
    answer: str = dspy.OutputField(desc="Concise and accurate answer")

# Built-in module
simple_program = dspy.Predict(simple_signature)  # or
dspy.ChainOfThought(ComplexSignature)

# Custom module
class ComplexModule(dspy.Module):
    def __init__(self):
        self.reasoner = dspy.ChainOfThought("question -> intermediate_answer")
        self.finalizer = dspy.Predict("intermediate_answer -> answer")

    def forward(self, question: str):
        intermediate = self.reasoner(question=question)
        final = self.finalizer(intermediate_answer=intermediate.intermediate_answer)
        return dspy.Prediction(answer=final.answer, reasoning=intermediate.reasoning) #
dspy.ChainOfThought returns 'reasoning' in addition to the signature outputs.

complex_program = ComplexModule()
```

DSPy Improvement Strategies:
1. Analyze traces for LM overload: If a single call struggles (e.g., skips steps or
hallucinates), decompose into multi-step modules with ChainOfThought or custom logic for
stepwise reasoning.
2. Avoid over-decomposition: If the program is too fragmented, consolidate related steps
into fewer modules for efficiency and coherence.
3. Refine signatures: Enhance docstrings with actionable guidance from traces—address
specific errors, incorporate domain knowledge, document edge cases, and suggest reasoning
patterns. Ensure docstrings are self-contained, as the LM won't have access external
traces during runtime.
4. Balance LM and Python: Use Python for symbolic/logical operations (e.g., loops,
conditionals); delegate complex reasoning or generation to LM calls.
5. Incorporate control flow: Add loops, conditionals, sub-modules in custom modules if the
task requires iteration (e.g., multi-turn reasoning, selection, voting, etc.).
6. Leverage LM strengths: For code-heavy tasks, define signatures with 'code' outputs,
extract and execute the generated code in the module's forward pass.

Here's my current code:
```
<curr_program>
```

Here is the execution trace of the current code on example inputs, their outputs, and
detailed feedback on improvements:
```
```
```

```
<dataset_with_feedback>
```

Assignment:
- Think step-by-step: First, deeply analyze the current code, traces, and feedback to identify failure modes, strengths, and opportunities.
- Create a concise checklist (3-7 bullets) outlining your high-level improvement plan, focusing on conceptual changes (e.g., "Decompose step X into a multi-stage module").
- Then, propose a drop-in replacement code that instantiates an improved 'program' object.
- Ensure the code is modular, efficient, and directly addresses feedback.
- Output everything in a single code block using triple backticks—no additional explanations, comments, or language markers outside the block.
- The code must be a valid, self-contained Python script with all necessary imports, definitions, and assignment to 'program'.

Output Format:
- Start with the checklist in plain text (3-7 short bullets).
- Follow immediately with one code block in triple backticks containing the complete Python code, including assigning a `program` object."""
    input_keys: ClassVar[list[str]] = ["curr_program", "dataset_with_feedback"]
    output_keys: ClassVar[list[str]] = ["new_program"]

    @classmethod
    def prompt_renderer(cls, input_dict: dict[str, Any]) -> str:
        curr_program = input_dict["curr_program"]
        if not isinstance(curr_program, str):
            raise TypeError("curr_program must be a string")

        dataset = input_dict["dataset_with_feedback"]
        if not isinstance(dataset, list):
            raise TypeError("dataset_with_feedback must be a list")

        def format_samples(samples):
            # Serialize the samples list to YAML for concise, structured representation
            yaml_str = yaml.dump(samples, sort_keys=False, default_flow_style=False, indent=2)
            # Optionally, wrap or label it for clarity in the prompt
            return yaml_str

        prompt = cls.prompt_template
        prompt = prompt.replace("<curr_program>", curr_program)
        prompt = prompt.replace("<dataset_with_feedback>", format_samples(dataset))
        return prompt

    @staticmethod
    def output_extractor(lm_out: str) -> dict[str, str]:
        # Extract ``` blocks
        new_instruction = None
        if lm_out.count("```") >= 2:
            start = lm_out.find("```")
            end = lm_out.rfind("```")
            if start >= end:
                new_instruction = lm_out
            if start == -1 or end == -1:
                new_instruction = lm_out
            else:
                new_instruction = lm_out[start + 3 : end].strip()
        else:
            lm_out = lm_out.strip()
            if lm_out.startswith("```"):
                lm_out = lm_out[3:]
            if lm_out.endswith("```"):
                lm_out = lm_out[:-3]
            new_instruction = lm_out

        return {"new_program": new_instruction}
```
```

## src/gepa/adapters/dspy_full_program_adapter/full_program_adapter.py

```
import random
from typing import Any, Callable

import dspy
from dspy.adapters.types import History
from dspy.evaluate import Evaluate
from dspy.primitives import Example, Prediction
from dspy.teleprompt.bootstrap_trace import TraceData

from gepa import EvaluationBatch, GEPAAdapter


class DspyAdapter(GEPAAdapter[Example, TraceData, Prediction]):
    def __init__(
        self,
        task_lm: dspy.LM,
        metric_fn: Callable,
        reflection_lm: dspy.LM,
        failure_score=0.0,
        num_threads: int | None = None,
        add_format_failure_as_feedback: bool = False,
        rng: random.Random | None = None,
    ):
        self.task_lm = task_lm
        self.metric_fn = metric_fn
        assert reflection_lm is not None, (
            "DspyAdapter for full-program evolution requires a reflection_lm to be
provided"
        )
        self.reflection_lm = reflection_lm
        self.failure_score = failure_score
        self.num_threads = num_threads
        self.add_format_failure_as_feedback = add_format_failure_as_feedback
        self.rng = rng or random.Random(0)

    def build_program(self, candidate: dict[str, str]) -> tuple[dspy.Module, None] |
tuple[None, str]:
        candidate_src = candidate["program"]
        context = {}
        o = self.load_dspy_program_from_code(candidate_src, context)
        return o

    def load_dspy_program_from_code(
        self,
        candidate_src: str,
        context: dict,
    ):
        try:
            compile(candidate_src, "<string>", "exec")
        except SyntaxError as e:
            # print(f"Syntax Error in original code {e}")
            # return None
            import traceback

            tb = traceback.format_exc()
            return None, f"Syntax Error in code: {e}\n{tb}"

        try:
            exec(candidate_src, context)  # expose to current namespace
        except Exception as e:
            import traceback

            tb = traceback.format_exc()
```

```
            return None, f"Error in executing code: {e}\n{tb}"

        dspy_program = context.get("program")

        if dspy_program is None:
            return (
                None,
                "Your code did not define a `program` object. Please define a `program` object which is an instance of `dspy.Module`, either directly by dspy.Predict or dspy.ChainOfThought, or by instantiating a class that inherits from `dspy.Module`.",
            )
        else:
            if not isinstance(dspy_program, dspy.Module):
                return (
                    None,
                    f"Your code defined a `program` object, but it is an instance of {type(dspy_program)}, not `dspy.Module`. Please define a `program` object which is an instance of `dspy.Module`, either directly by dspy.Predict or dspy.ChainOfThought, or by instantiating a class that inherits from `dspy.Module`.",
                )

        dspy_program.set_lm(self.task_lm)

        return dspy_program, None

    def evaluate(self, batch, candidate, capture_traces=False):
        program, feedback = self.build_program(candidate)

        if program is None:
            return EvaluationBatch(outputs=None, scores=[self.failure_score for _ in batch], trajectories=feedback)

        if capture_traces:
            # bootstrap_trace_data-like flow with trace capture
            from dspy.teleprompt.bootstrap_trace import bootstrap_trace_data

            trajs = bootstrap_trace_data(
                program=program,
                dataset=batch,
                metric=self.metric_fn,
                num_threads=self.num_threads,
                raise_on_error=False,
                capture_failed_parses=True,
                failure_score=self.failure_score,
                format_failure_score=self.failure_score,
            )
            scores = []
            outputs = []
            for t in trajs:
                outputs.append(t["prediction"])
                if hasattr(t["prediction"], "__class__") and t.get("score") is None:
                    scores.append(self.failure_score)
                else:
                    score = t["score"]
                    if hasattr(score, "score"):
                        score = score["score"]
                    scores.append(score)
            return EvaluationBatch(outputs=outputs, scores=scores, trajectories=trajs)
        else:
            evaluator = Evaluate(
                devset=batch,
                metric=self.metric_fn,
                num_threads=self.num_threads,
                return_all_scores=True,
                failure_score=self.failure_score,
                provide_traceback=True,
                max_errors=len(batch) * 100,
            )
```

```
            res = evaluator(program)
            outputs = [r[1] for r in res.results]
            scores = [r[2] for r in res.results]
            scores = [s["score"] if hasattr(s, "score") else s for s in scores]
            return EvaluationBatch(outputs=outputs, scores=scores, trajectories=None)

    def make_reflective_dataset(self, candidate, eval_batch, components_to_update):
        proposed_program, _ = self.build_program(candidate)

        assert set(components_to_update) == {"program"}, f"set(components_to_update) =
{set(components_to_update)}"
        from dspy.teleprompt.bootstrap_trace import FailedPrediction

        ret_d: dict[str, list[dict[str, Any]]] = {}

        if isinstance(eval_batch.trajectories, str):
            feedback = eval_batch.trajectories
            return {"program": {"Feedback": feedback}}

        ########
        items: list[dict[str, Any]] = []
        for data in eval_batch.trajectories or []:
            example_data = {}
            trace = data["trace"]
            example = data["example"]
            example_data["Program Inputs"] = {**example.inputs()}
            prediction = data["prediction"]
            example_data["Program Outputs"] = {**prediction}
            module_score = data["score"]

            if hasattr(module_score, "feedback"):
                feedback_text = module_score["feedback"]
            else:
                feedback_text = None

            if hasattr(module_score, "score"):
                module_score = module_score["score"]

            trace_instances = trace

            if len(trace_instances) == 0:
                continue

            selected = None
            for t in trace_instances:
                if isinstance(t[2], FailedPrediction):
                    selected = t
                    break

            if selected is not None:
                trace_instances = [selected]

            trace_d = []
            example_data["Program Trace"] = trace_d
            for selected in trace_instances:
                inputs = selected[1]
                outputs = selected[2]

                pred_name = None
                for name, predictor in proposed_program.named_predictors():
                    if predictor.signature.equals(selected[0].signature):
                        pred_name = name
                        break
                assert pred_name is not None, f"Could not find predictor for
{selected[0].signature}"

                new_inputs = {}
                new_outputs = {}
```

```
                contains_history = False
                history_key_name = None
                for input_key, input_val in inputs.items():
                    if isinstance(input_val, History):
                        contains_history = True
                        assert history_key_name is None
                        history_key_name = input_key

                if contains_history:
                    s = "```json\n"
                    for i, message in enumerate(inputs[history_key_name].messages):
                        s += f"  {i}: {message}\n"
                    s += "```"
                    new_inputs["Context"] = s

                for input_key, input_val in inputs.items():
                    if contains_history and input_key == history_key_name:
                        continue
                    new_inputs[input_key] = str(input_val)

                if isinstance(outputs, FailedPrediction):
                    s = "Couldn't parse the output as per the expected output format. The
model's raw response was:\n"
                    s += "```\n"
                    s += outputs.completion_text + "\n"
                    s += "```\n\n"
                    new_outputs = s
                else:
                    for output_key, output_val in outputs.items():
                        new_outputs[output_key] = str(output_val)

                d = {"Called Module": pred_name, "Inputs": new_inputs, "Generated
Outputs": new_outputs}
                # if isinstance(outputs, FailedPrediction):
                #     adapter = ChatAdapter()
                #     structure_instruction = ""
                #     for dd in adapter.format(module.signature, [], {}):
                #         structure_instruction += dd["role"] + ": " + dd["content"] +
"\n"
                #     d["Feedback"] = "Your output failed to parse. Follow this
structure:\n" + structure_instruction
                #     # d['score'] = self.failure_score
                # else:
                # assert fb["score"] == module_score, f"Currently, GEPA only supports
feedback functions that return the same score as the module's score. However, the module-
level score is {module_score} and the feedback score is {fb.score}."
                # d['score'] = fb.score
                trace_d.append(d)

            if feedback_text is not None:
                example_data["Feedback"] = feedback_text

            items.append(example_data)

        if len(items) == 0:
            raise Exception("No valid predictions found for program.")

        ret_d["program"] = items

        ########
        if len(ret_d) == 0:
            raise Exception("No valid predictions found for any module.")

        return ret_d

    def propose_new_texts(
        self,
```

```
        candidate: dict[str, str],
        reflective_dataset: dict[str, list[dict[str, Any]]],
        components_to_update: list[str],
    ) -> dict[str, str]:
        from gepa.adapters.dspy_full_program_adapter.dspy_program_proposal_signature
import DSPyProgramProposalSignature

        new_texts: dict[str, str] = {}
        for name in components_to_update:
            base_instruction = candidate[name]
            dataset_with_feedback = reflective_dataset[name]
            new_texts[name] = DSPyProgramProposalSignature.run(
                lm=self.reflection_lm,
                input_dict={"curr_program": base_instruction, "dataset_with_feedback":
dataset_with_feedback},
            )["new_program"]
        return new_texts
```

## src/gepa/adapters/generic_rag_adapter/GEPA_RAG.md

```
# Generic RAG Adapter for GEPA (DOCS ONLY)

A vector store-agnostic RAG (Retrieval-Augmented Generation) adapter that enables GEPA to
optimize RAG systems across any vector store implementation.

## 🎯 Overview

The Generic RAG Adapter brings GEPA's evolutionary optimization to the world of RAG
systems. With its pluggable vector store architecture, you can optimize RAG prompts once
and deploy across any vector store—from local ChromaDB instances to production Weaviate
clusters.

**Key Benefits:**
- **Vector Store Agnostic**: Write once, run anywhere (ChromaDB, Weaviate, Qdrant, Milvus,
LanceDB)
- **Multi-Component Optimization**: Simultaneously optimize query reformulation, context
synthesis, answer generation, and reranking
- **Comprehensive Evaluation**: Both retrieval quality (precision, recall, MRR) and
generation quality (F1, BLEU, faithfulness) metrics
- **Production Ready**: Battle-tested with proper error handling, logging, and performance
monitoring
- **5 Vector Stores Supported**: Complete examples for ChromaDB, Weaviate, Qdrant, Milvus,
and LanceDB

## 🚀 Quick Start

### Installation

```bash
# Install core GEPA package
pip install gepa

# Install RAG adapter dependencies
# Navigate to the examples/rag_adapter directory
cd src/gepa/examples/rag_adapter

# Option A: Install all vector store dependencies (recommended for exploration)
pip install -r requirements-rag.txt

# Option B: Install specific vector store dependencies
pip install litellm chromadb                    # For ChromaDB
pip install litellm weaviate-client             # For Weaviate
pip install litellm lancedb pyarrow             # For LanceDB
```
```

```
pip install litellm pymilvus                    # For Milvus
pip install litellm qdrant-client               # For Qdrant

# Setup local Ollama models for examples
ollama pull qwen3:8b          # Default for ChromaDB/Weaviate/Qdrant
ollama pull llama3.1:8b       # Default for LanceDB/Milvus
ollama pull nomic-embed-text:latest  # Embedding model
```

**Note:** For specific version requirements, see the `requirements-rag.txt` file in the `examples/rag_adapter/` directory.

### 5-Minute Example

```python
import gepa
from gepa.adapters.generic_rag_adapter import GenericRAGAdapter, ChromaVectorStore

# 1. Setup your vector store
vector_store = ChromaVectorStore.create_local("./knowledge_base", "documents")

# 2. Create the RAG adapter
adapter = GenericRAGAdapter(
    vector_store=vector_store,
    llm_model="ollama/llama3.2:1b",  # Memory-friendly local model
    # llm_model="gpt-4",  # For cloud-based models (requires API key)
    rag_config={
        "retrieval_strategy": "similarity",
        "top_k": 3  # Reduced for faster local testing
    }
)

# 3. Define your training data
train_data = [
    {
        "query": "What is machine learning?",
        "ground_truth_answer": "Machine learning is...",
        "relevant_doc_ids": ["doc_001"],
        "metadata": {"category": "AI"}
    }
    # ... more examples
]

# 4. Define initial prompts to optimize
initial_prompts = {
    "answer_generation": "Answer the question based on the provided context.",
    "context_synthesis": "Synthesize the following documents into a coherent context."
}

# 5. Run GEPA optimization (local-friendly settings)
result = gepa.optimize(
    seed_candidate=initial_prompts,
    trainset=train_data,
    valset=validation_data,
    adapter=adapter,
    max_metric_calls=10,  # Small number for local testing
    reflection_llm_model="ollama/llama3.1:8b"  # Memory-friendly reflection model
    # reflection_llm_model="gpt-4"  # For cloud-based optimization
)

print("Optimized RAG prompts:", result.best_candidate)
```

### Vector Store Support

```python
# ChromaDB (local development, no Docker required)
vector_store = ChromaVectorStore.create_local("./kb", "docs")
```

```
# Weaviate (production with hybrid search, Docker required)
vector_store = WeaviateVectorStore.create_local(
    host="localhost", port=8080, collection_name="KnowledgeBase"
)

# Qdrant (high performance, Docker optional)
vector_store = QdrantVectorStore.create_local("./qdrant_db", "KnowledgeBase")

# Milvus (cloud-native, uses Milvus Lite by default)
vector_store = MilvusVectorStore.create_local("KnowledgeBase")

# LanceDB (serverless, no Docker required)
vector_store = LanceDBVectorStore.create_local("./lancedb", "KnowledgeBase")

# Same optimization pipeline works with all!
```

**Optimizable Components:**
- **Query Reformulation**: Improve user query understanding and reformulation
- **Context Synthesis**: Optimize document combination and summarization
- **Answer Generation**: Enhance final answer quality and accuracy
- **Document Reranking**: Improve retrieved document relevance ordering

## 🏗️ Architecture

### Vector Store Interface

The adapter uses a clean abstraction that any vector store can implement:

```python
class VectorStoreInterface(ABC):
    @abstractmethod
    def similarity_search(self, query: str, k: int = 5) -> List[Dict[str, Any]]:
        """Semantic similarity search"""

    @abstractmethod
    def vector_search(self, query_vector: List[float], k: int = 5) -> List[Dict[str,
Any]]:
        """Direct vector search"""

    def hybrid_search(self, query: str, k: int = 5, alpha: float = 0.5) -> List[Dict[str,
Any]]:
        """Hybrid semantic + keyword search (optional)"""

    @abstractmethod
    def get_collection_info(self) -> Dict[str, Any]:
        """Collection metadata and statistics"""
```

### RAG Pipeline Flow

```
User Query → Query Reformulation → Document Retrieval → Document Reranking → Context
Synthesis → Answer Generation → Final Answer
     ↑              ↑                      ↑                    ↑                   ↑
↑
  Original      Optimizable          Vector Store        Optimizable        Optimizable
Optimizable
   Input          Prompt             Operations            Prompt             Prompt
Prompt
```

## 🔧 Supported Vector Stores

### ChromaDB
Perfect for local development, prototyping, and smaller deployments. **No Docker

````
required.**

```python
from gepa.adapters.generic_rag_adapter import ChromaVectorStore

# Local persistence
vector_store = ChromaVectorStore.create_local(
    persist_directory="./chroma_db",
    collection_name="documents"
)

# In-memory (testing)
vector_store = ChromaVectorStore.create_memory(
    collection_name="test_docs"
)
```

### Weaviate
Production-grade with advanced features like hybrid search and multi-tenancy. **Docker
required.**

```python
from gepa.adapters.generic_rag_adapter import WeaviateVectorStore

# Local Weaviate instance
vector_store = WeaviateVectorStore.create_local(
    host="localhost",
    port=8080,
    collection_name="Documents"
)

# Weaviate Cloud Services (WCS)
vector_store = WeaviateVectorStore.create_cloud(
    cluster_url="https://your-cluster.weaviate.network",
    auth_credentials=weaviate.AuthApiKey("your-api-key"),
    collection_name="Documents"
)
```

### Qdrant
High-performance vector database with advanced filtering and payload search. **Docker
optional.**

```python
from gepa.adapters.generic_rag_adapter import QdrantVectorStore

# In-memory (default, no setup required)
vector_store = QdrantVectorStore.create_memory("documents")

# Local persistent storage
vector_store = QdrantVectorStore.create_local("./qdrant_db", "documents")

# Remote Qdrant server
vector_store = QdrantVectorStore.create_remote(
    host="localhost", port=6333, collection_name="documents"
)
```

### Milvus
Cloud-native vector database designed for large-scale AI applications. **Uses Milvus Lite
by default (no Docker required).**

```python
from gepa.adapters.generic_rag_adapter import MilvusVectorStore

# Milvus Lite (local SQLite-based, no Docker required)
vector_store = MilvusVectorStore.create_local("documents")
````
````

```
# Full Milvus server (Docker required)
vector_store = MilvusVectorStore.create_remote(
    uri="http://localhost:19530", collection_name="documents"
)
```

### LanceDB
Serverless vector database built on the Lance columnar format. **No Docker required.**

```python
from gepa.adapters.generic_rag_adapter import LanceDBVectorStore

# Local LanceDB instance
vector_store = LanceDBVectorStore.create_local("./lancedb", "documents")

# In-memory (testing)
vector_store = LanceDBVectorStore.create_memory("documents")
```

### Adding New Vector Stores

Implement the `VectorStoreInterface` for your vector store:

```python
class MyVectorStore(VectorStoreInterface):
    def similarity_search(self, query: str, k: int = 5, filters=None):
        # Your implementation
        return documents

    def vector_search(self, query_vector: List[float], k: int = 5, filters=None):
        # Your implementation
        return documents

    def get_collection_info(self):
        return {
            "name": self.collection_name,
            "document_count": self.count(),
            "dimension": self.vector_dim,
            "vector_store_type": "my_store"
        }
```

## 🎛️ Configuration Options

### Model Configuration

**Local Ollama Models (Recommended for Testing):**
```python
# Memory-friendly options
adapter = GenericRAGAdapter(
    vector_store=vector_store,
    llm_model="ollama/llama3.2:1b",  # ~1GB RAM - Fast inference
    rag_config=config
)

# Higher quality local models
adapter = GenericRAGAdapter(
    vector_store=vector_store,
    llm_model="ollama/llama3.1:8b",  # ~5GB RAM - Better quality
    rag_config=config
)

# GEPA optimization with local models
result = gepa.optimize(
    seed_candidate=initial_prompts,
    trainset=train_data,
    valset=validation_data,
    adapter=adapter,
```

```
    max_metric_calls=5,  # Small for local testing
    reflection_llm_model="ollama/llama3.1:8b"
)
```

**Cloud Models (Production Use):**
```python
# OpenAI models
adapter = GenericRAGAdapter(
    vector_store=vector_store,
    llm_model="gpt-4o",  # Requires OPENAI_API_KEY
    rag_config=config
)

# Anthropic models
adapter = GenericRAGAdapter(
    vector_store=vector_store,
    llm_model="claude-3-5-sonnet-20241022",  # Requires ANTHROPIC_API_KEY
    rag_config=config
)

# GEPA optimization with cloud models
result = gepa.optimize(
    seed_candidate=initial_prompts,
    trainset=train_data,
    valset=validation_data,
    adapter=adapter,
    max_metric_calls=50,  # Higher for production optimization
    reflection_llm_model="gpt-4o"
)
```

### RAG Pipeline Configuration

```python
rag_config = {
    # Retrieval Strategy
    "retrieval_strategy": "similarity",    # "similarity", "hybrid", "vector"
    "top_k": 5,                           # Documents to retrieve

    # Evaluation Weights
    "retrieval_weight": 0.3,              # Weight for retrieval metrics
    "generation_weight": 0.7,             # Weight for generation metrics

    # Hybrid Search (Weaviate)
    "hybrid_alpha": 0.5,                  # 0.0=keyword, 1.0=semantic, 0.5=balanced

    # Filtering
    "filters": {"category": "technical"}  # Metadata filters
}

adapter = GenericRAGAdapter(
    vector_store=vector_store,
    llm_model="ollama/llama3.2:1b",  # Local model for testing
    # llm_model="gpt-4o",  # For cloud-based usage
    rag_config=rag_config
)
```

### Advanced Configuration Examples

**ChromaDB Optimized:**
```python
config = {
    "retrieval_strategy": "similarity",
    "top_k": 7,
    "retrieval_weight": 0.35,
    "generation_weight": 0.65
```

```
}
```

**Weaviate with Hybrid Search:**
```python
config = {
    "retrieval_strategy": "hybrid",
    "top_k": 5,
    "hybrid_alpha": 0.7,              # More semantic than keyword
    "retrieval_weight": 0.25,
    "generation_weight": 0.75,
    "filters": {"confidence": {"$gt": 0.8}}
}
```

## 🔍 Optimizable Components

### 1. Query Reformulation
Transform user queries for better retrieval:

```python
"query_reformulation": """
You are an expert at reformulating user queries for information retrieval.
Your task is to enhance the query while preserving the original intent.

Guidelines:
- Add relevant technical terms and synonyms
- Make the query more specific and focused
- Optimize for both semantic and keyword matching
- Preserve key concepts from the original query

Reformulate the following query for better retrieval:
"""
```

### 2. Context Synthesis
Combine retrieved documents into coherent context:

```python
"context_synthesis": """
You are an expert at synthesizing information from multiple documents.
Your task is to create a comprehensive context that directly addresses the query.

Guidelines:
- Focus on information most relevant to the user's question
- Integrate information from multiple sources seamlessly
- Remove redundant or conflicting information
- Maintain factual accuracy and important details

Query: {query}

Synthesize the following retrieved documents:
"""
```

### 3. Answer Generation
Generate accurate, well-structured final answers:

```python
"answer_generation": """
You are an AI assistant providing expert-level answers.
Your task is to generate accurate, comprehensive responses based on the provided context.

Guidelines:
- Base your answer primarily on the provided context
- Structure your response with clear explanations
- Include specific details and examples when available
- If context is insufficient, acknowledge the limitation clearly
```

````
Context: {context}
Question: {query}

Provide a thorough, accurate answer:
"""
```

### 4. Document Reranking
Optimize retrieved document relevance ordering:

```python
"reranking_criteria": """
You are an expert at evaluating document relevance for question answering.
Your task is to rank documents by their relevance to the specific query.

Ranking Criteria:
- Documents with direct answers get highest priority
- Comprehensive explanations rank second
- Supporting examples and context rank third
- Off-topic or tangential content ranks lowest

Query: {query}

Rank these documents by relevance (most relevant first):
"""
```

## 📊 Evaluation Metrics

### Retrieval Quality Metrics

- **Precision**: Fraction of retrieved documents that are relevant
- **Recall**: Fraction of relevant documents that were retrieved
- **F1 Score**: Harmonic mean of precision and recall
- **MRR (Mean Reciprocal Rank)**: Quality of ranking for relevant documents

### Generation Quality Metrics

- **Exact Match**: Whether generated answer exactly matches ground truth
- **Token F1**: F1 score based on token overlap with ground truth
- **BLEU Score**: N-gram overlap similarity measure
- **Answer Relevance**: How well the answer relates to retrieved context
- **Faithfulness**: How well the answer is supported by the context

### Combined Scoring

The adapter computes a weighted combination of retrieval and generation metrics:

```
final_score = (retrieval_weight × retrieval_f1) + (generation_weight × generation_score)
```

Where `generation_score` combines token F1, answer relevance, and faithfulness.

## 🚀 Production Examples

### Multi-Vector Store Deployment

```python
def create_rag_adapter(env: str):
    if env == "development":
        vector_store = ChromaVectorStore.create_local("./local_kb", "docs")
        config = {"retrieval_strategy": "similarity", "top_k": 3}
        llm_model = "ollama/llama3.2:1b"  # Memory-friendly for local dev
    elif env == "production":
        vector_store = WeaviateVectorStore.create_cloud(
````

```
            cluster_url=os.getenv("WEAVIATE_URL"),
            auth_credentials=weaviate.AuthApiKey(os.getenv("WEAVIATE_KEY")),
            collection_name="ProductionKB"
        )
        config = {
            "retrieval_strategy": "hybrid",
            "hybrid_alpha": 0.75,
            "top_k": 5,
            "filters": {"status": "approved"}
        }
        llm_model = os.getenv("LLM_MODEL", "gpt-4o")  # Cloud models for production

    return GenericRAGAdapter(
        vector_store=vector_store,
        llm_model=llm_model,
        rag_config=config
    )
```

### Performance Monitoring

```python
# Enable detailed tracing for analysis
eval_batch = adapter.evaluate(
    batch=test_data,
    candidate=optimized_prompts,
    capture_traces=True
)

# Analyze performance
for i, (trajectory, score) in enumerate(zip(eval_batch.trajectories, eval_batch.scores)):
    print(f"Query {i+1}: Score = {score:.3f}")
    print(f"  Retrieved: {len(trajectory['retrieved_docs'])} documents")
    print(f"  Token usage: {trajectory['execution_metadata']['total_tokens']}")

    # Access detailed metrics
    retrieval_metrics = trajectory['execution_metadata']['retrieval_metrics']
    generation_metrics = trajectory['execution_metadata']['generation_metrics']

    print(f"  Retrieval F1: {retrieval_metrics['retrieval_f1']:.3f}")
    print(f"  Generation F1: {generation_metrics['token_f1']:.3f}")
    print(f"  Faithfulness: {generation_metrics['faithfulness']:.3f}")
```

## 📚 Complete Examples

### Unified RAG Optimization Script
We've consolidated all vector database examples into a single, unified script in `src/gepa/examples/rag_adapter/`:

- **[Unified RAG Optimization](examples/rag_adapter/rag_optimization.py)** - One script supporting all vector stores
  - ChromaDB - Local development, no Docker required
  - Weaviate - Production deployment with hybrid search
  - Qdrant - High performance with advanced filtering
  - Milvus - Cloud-native with Milvus Lite
  - LanceDB - Serverless, developer-friendly

### Quick Start Guide
- **[RAG_GUIDE.md](examples/rag_adapter/RAG_GUIDE.md)** - Comprehensive setup instructions for the unified approach
- **[requirements-rag.txt](examples/rag_adapter/requirements-rag.txt)** - All vector store dependencies in one file
- **Docker Requirements** - Clear guidance on which vector stores need Docker vs. which don't
- **Model Recommendations** - Performance expectations and use cases for each database

```
## 🤝 Contributing

We welcome contributions! Priority areas:

### New Vector Store Implementations
- **Pinecone**: Managed vector database with high performance
- **Elasticsearch**: Search engine with vector capabilities
- **OpenSearch**: Open-source alternative to Elasticsearch
- **FAISS**: Facebook AI Similarity Search
- **Annoy**: Approximate Nearest Neighbors

**Already Implemented:**
- ✅ **ChromaDB** - Local development and prototyping
- ✅ **Weaviate** - Production-grade with hybrid search
- ✅ **Qdrant** - High performance with advanced filtering
- ✅ **Milvus** - Cloud-native with Milvus Lite support
- ✅ **LanceDB** - Serverless, developer-friendly

### Enhancement Areas
- Advanced reranking algorithms (learning-to-rank, neural rerankers)
- Multi-modal RAG support (text + images)
- Streaming evaluation for large datasets
- Integration with embedding providers (OpenAI, Cohere, HuggingFace)
- Performance optimizations and caching strategies

### Implementation Guidelines

1. **Follow the Interface**: Implement `VectorStoreInterface` completely
2. **Add Factory Methods**: Provide `create_local()`, `create_cloud()` class methods
3. **Error Handling**: Graceful degradation and clear error messages
4. **Documentation**: Comprehensive docstrings and usage examples
5. **Testing**: Unit tests for all public methods

See our [contribution guide](CONTRIBUTING.md) for detailed instructions.

## 📄 API Reference

### Core Classes

- **[`GenericRAGAdapter`](generic_rag_adapter.py)**: Main adapter class for GEPA
integration
- **[`VectorStoreInterface`](vector_store_interface.py)**: Abstract base class for vector
stores
- **[`RAGPipeline`](rag_pipeline.py)**: RAG execution engine
- **[`RAGEvaluationMetrics`](evaluation_metrics.py)**: Comprehensive evaluation metrics

### Vector Store Implementations

- **[`ChromaVectorStore`](vector_stores/chroma_store.py)**: ChromaDB implementation
- **[`WeaviateVectorStore`](vector_stores/weaviate_store.py)**: Weaviate implementation
- **[`QdrantVectorStore`](vector_stores/qdrant_store.py)**: Qdrant implementation
- **[`MilvusVectorStore`](vector_stores/milvus_store.py)**: Milvus implementation
- **[`LanceDBVectorStore`](vector_stores/lancedb_store.py)**: LanceDB implementation

### Type Definitions

- **`RAGDataInst`**: Training/validation example structure
- **`RAGTrajectory`**: Detailed execution trace
- **`RAGOutput`**: Final system output with metadata

## 🔒 License

Copyright (c) 2025 Lakshya A Agrawal and the GEPA contributors
Licensed under the MIT License - see [LICENSE](../../../LICENSE) for details.
```

## src/gepa/adapters/generic_rag_adapter/__init__.py

```
# Copyright (c) 2025 Lakshya A Agrawal and the GEPA contributors
# https://github.com/gepa-ai/gepa

from .evaluation_metrics import RAGEvaluationMetrics
from .generic_rag_adapter import (
    GenericRAGAdapter,
    RAGDataInst,
    RAGOutput,
    RAGTrajectory,
)
from .rag_pipeline import RAGPipeline
from .vector_store_interface import VectorStoreInterface
from .vector_stores.chroma_store import ChromaVectorStore
from .vector_stores.weaviate_store import WeaviateVectorStore

# Optional vector stores - import only if dependencies are available
try:
    from .vector_stores.qdrant_store import QdrantVectorStore

    _QDRANT_AVAILABLE = True
except ImportError:
    _QDRANT_AVAILABLE = False

try:
    from .vector_stores.milvus_store import MilvusVectorStore

    _MILVUS_AVAILABLE = True
except ImportError:
    _MILVUS_AVAILABLE = False

try:
    from .vector_stores.lancedb_store import LanceDBVectorStore

    _LANCEDB_AVAILABLE = True
except ImportError:
    _LANCEDB_AVAILABLE = False

__all__ = [
    "GenericRAGAdapter",
    "RAGDataInst",
    "RAGOutput",
    "RAGTrajectory",
    "VectorStoreInterface",
    "ChromaVectorStore",
    "WeaviateVectorStore",
    "RAGPipeline",
    "RAGEvaluationMetrics",
]

# Add optional vector stores to __all__ if available
if _QDRANT_AVAILABLE:
    __all__.append("QdrantVectorStore")
if _MILVUS_AVAILABLE:
    __all__.append("MilvusVectorStore")
if _LANCEDB_AVAILABLE:
    __all__.append("LanceDBVectorStore")
```

## src/gepa/adapters/generic_rag_adapter/evaluation_metrics.py

```
# Copyright (c) 2025 Lakshya A Agrawal and the GEPA contributors
# https://github.com/gepa-ai/gepa

import re
from typing import Any


class RAGEvaluationMetrics:
    """
    Evaluation metrics for RAG systems.

    Provides both retrieval and generation quality metrics
    for comprehensive RAG system evaluation.
    """

    def evaluate_retrieval(self, retrieved_docs: list[dict[str, Any]], relevant_doc_ids:
list[str]) -> dict[str, float]:
        """
        Evaluate retrieval quality metrics.

        Args:
            retrieved_docs: List of retrieved documents with metadata
            relevant_doc_ids: List of ground truth relevant document IDs

        Returns:
            Dictionary with retrieval metrics (precision, recall, f1, mrr)
        """
        if not retrieved_docs or not relevant_doc_ids:
            return {"retrieval_precision": 0.0, "retrieval_recall": 0.0, "retrieval_f1":
0.0, "retrieval_mrr": 0.0}

        # Extract document IDs from retrieved docs
        retrieved_ids = []
        for doc in retrieved_docs:
            doc_id = doc.get("metadata", {}).get("doc_id") or doc.get("metadata",
{}).get("id")
            if doc_id:
                retrieved_ids.append(str(doc_id))

        relevant_set = set(relevant_doc_ids)
        retrieved_set = set(retrieved_ids)

        # Calculate precision and recall
        if len(retrieved_set) == 0:
            precision = 0.0
        else:
            precision = len(relevant_set.intersection(retrieved_set)) / len(retrieved_set)

        if len(relevant_set) == 0:
            recall = 0.0
        else:
            recall = len(relevant_set.intersection(retrieved_set)) / len(relevant_set)

        # Calculate F1
        if precision + recall == 0:
            f1 = 0.0
        else:
            f1 = 2 * (precision * recall) / (precision + recall)

        # Calculate Mean Reciprocal Rank (MRR)
        mrr = 0.0
        for i, retrieved_id in enumerate(retrieved_ids):
            if retrieved_id in relevant_set:
                mrr = 1.0 / (i + 1)
                break

        return {"retrieval_precision": precision, "retrieval_recall": recall,
```

```
"retrieval_f1": f1, "retrieval_mrr": mrr}

    def evaluate_generation(self, generated_answer: str, ground_truth: str, context: str)
-> dict[str, float]:
        """
        Evaluate generation quality metrics.

        Args:
            generated_answer: Generated answer text
            ground_truth: Ground truth answer
            context: Retrieved context used for generation

        Returns:
            Dictionary with generation metrics
        """
        # Exact match (case-insensitive)
        exact_match = self._exact_match(generated_answer, ground_truth)

        # F1 score based on token overlap
        f1_score = self._token_f1(generated_answer, ground_truth)

        # BLEU-like score
        bleu_score = self._simple_bleu(generated_answer, ground_truth)

        # Answer relevance (simple keyword overlap with context)
        relevance_score = self._answer_relevance(generated_answer, context)

        # Faithfulness (how well the answer is supported by context)
        faithfulness_score = self._faithfulness_score(generated_answer, context)

        return {
            "exact_match": float(exact_match),
            "token_f1": f1_score,
            "bleu_score": bleu_score,
            "answer_relevance": relevance_score,
            "faithfulness": faithfulness_score,
            "answer_confidence": (f1_score + relevance_score + faithfulness_score) / 3.0,
        }

    def combined_rag_score(
        self,
        retrieval_metrics: dict[str, float],
        generation_metrics: dict[str, float],
        retrieval_weight: float = 0.3,
        generation_weight: float = 0.7,
    ) -> float:
        """
        Combine retrieval and generation metrics into a single score.

        Args:
            retrieval_metrics: Output from evaluate_retrieval
            generation_metrics: Output from evaluate_generation
            retrieval_weight: Weight for retrieval score
            generation_weight: Weight for generation score

        Returns:
            Combined score between 0 and 1
        """
        # Primary retrieval metric: F1 score
        retrieval_score = retrieval_metrics.get("retrieval_f1", 0.0)

        # Primary generation metric: weighted combination
        generation_score = (
            generation_metrics.get("token_f1", 0.0) * 0.4
            + generation_metrics.get("answer_relevance", 0.0) * 0.3
            + generation_metrics.get("faithfulness", 0.0) * 0.3
        )
```

```
        return retrieval_weight * retrieval_score + generation_weight * generation_score

    def _exact_match(self, prediction: str, ground_truth: str) -> bool:
        """Check if prediction exactly matches ground truth (case-insensitive)."""
        return prediction.strip().lower() == ground_truth.strip().lower()

    def _token_f1(self, prediction: str, ground_truth: str) -> float:
        """Calculate F1 score based on token overlap."""
        pred_tokens = set(self._normalize_text(prediction).split())
        truth_tokens = set(self._normalize_text(ground_truth).split())

        if len(pred_tokens) == 0 and len(truth_tokens) == 0:
            return 1.0
        if len(pred_tokens) == 0 or len(truth_tokens) == 0:
            return 0.0

        intersection = pred_tokens.intersection(truth_tokens)
        precision = len(intersection) / len(pred_tokens)
        recall = len(intersection) / len(truth_tokens)

        if precision + recall == 0:
            return 0.0

        return 2 * (precision * recall) / (precision + recall)

    def _simple_bleu(self, prediction: str, ground_truth: str, n: int = 2) -> float:
        """Simple BLEU-like score for n-gram overlap."""
        pred_words = self._normalize_text(prediction).split()
        truth_words = self._normalize_text(ground_truth).split()

        if len(pred_words) < n or len(truth_words) < n:
            return self._token_f1(prediction, ground_truth)

        pred_ngrams = {tuple(pred_words[i : i + n]) for i in range(len(pred_words) - n +
1)}
        truth_ngrams = {tuple(truth_words[i : i + n]) for i in range(len(truth_words) - n
+ 1)}

        if len(pred_ngrams) == 0 or len(truth_ngrams) == 0:
            return 0.0

        intersection = pred_ngrams.intersection(truth_ngrams)
        return len(intersection) / len(pred_ngrams)

    def _answer_relevance(self, answer: str, context: str) -> float:
        """Measure how well the answer relates to the provided context."""
        answer_words = set(self._normalize_text(answer).split())
        context_words = set(self._normalize_text(context).split())

        if len(answer_words) == 0:
            return 0.0

        overlap = answer_words.intersection(context_words)
        return len(overlap) / len(answer_words)

    def _faithfulness_score(self, answer: str, context: str) -> float:
        """
        Measure how well the answer is supported by the context.
        Simple implementation based on shared key phrases.
        """
        # Extract key phrases (sequences of 2+ words)
        answer_phrases = self._extract_phrases(answer)
        context_phrases = self._extract_phrases(context)

        if len(answer_phrases) == 0:
            return 1.0  # Empty answer is technically faithful

        supported_phrases = answer_phrases.intersection(context_phrases)
```

```
        return len(supported_phrases) / len(answer_phrases)

    def _extract_phrases(self, text: str, min_length: int = 2) -> set[str]:
        """Extract meaningful phrases from text."""
        words = self._normalize_text(text).split()
        phrases = set()

        # Add individual significant words (length > 3)
        for word in words:
            if len(word) > 3:
                phrases.add(word)

        # Add bi-grams and tri-grams
        for n in range(min_length, min(4, len(words) + 1)):
            for i in range(len(words) - n + 1):
                phrase = " ".join(words[i : i + n])
                if len(phrase) > 5:  # Only meaningful phrases
                    phrases.add(phrase)

        return phrases

    def _normalize_text(self, text: str) -> str:
        """Normalize text for comparison."""
        # Convert to lowercase and remove extra whitespace
        text = text.lower().strip()
        # Remove punctuation and special characters
        text = re.sub(r"[^\w\s]", " ", text)
        # Normalize whitespace
        text = re.sub(r"\s+", " ", text)
        return text
```

## src/gepa/adapters/generic_rag_adapter/generic_rag_adapter.py

```
# Copyright (c) 2025 Lakshya A Agrawal and the GEPA contributors
# https://github.com/gepa-ai/gepa

from typing import Any, TypedDict

from gepa.adapters.generic_rag_adapter.evaluation_metrics import RAGEvaluationMetrics
from gepa.adapters.generic_rag_adapter.rag_pipeline import RAGPipeline
from gepa.adapters.generic_rag_adapter.vector_store_interface import VectorStoreInterface
from gepa.core.adapter import EvaluationBatch, GEPAAdapter


class RAGDataInst(TypedDict):
    """
    Data instance for RAG evaluation and optimization.

    This TypedDict defines the structure for training and validation examples
    used in RAG system optimization with GEPA.

    Attributes:
            query (str): User query or question to be answered
        ground_truth_answer (str): Expected/correct answer for evaluation
            relevant_doc_ids (List[str]): List of document IDs that should ideally
            be retrieved for this query (used for retrieval evaluation)
            metadata (Dict[str, Any]): Additional context, tags, or configuration
            specific to this example (e.g., difficulty level, category)

    Example:
            .. code-block:: python

            data_inst = RAGDataInst(
                query="What is machine learning?",
```

```
                ground_truth_answer="Machine learning is a subset of AI...",
                relevant_doc_ids=["doc_001", "doc_042"],
                metadata={"category": "AI", "difficulty": "beginner"}
            )
    """

    query: str
    ground_truth_answer: str
    relevant_doc_ids: list[str]
    metadata: dict[str, Any]


class RAGTrajectory(TypedDict):
    """
    Detailed trajectory capturing all RAG pipeline execution steps.

    This TypedDict captures the complete execution trace of the RAG pipeline,
    providing visibility into each step for analysis and optimization.

    Attributes:
            original_query (str): Original user query as provided
        reformulated_query (str): Query after reformulation step (if enabled)
            retrieved_docs (List[Dict[str, Any]]): Documents retrieved from vector store
            with their content, metadata, and similarity scores
            synthesized_context (str): Context after document synthesis step
            generated_answer (str): Final answer generated by the LLM
            execution_metadata (Dict[str, Any]): Pipeline execution metadata including
            retrieval metrics, generation metrics, token counts, and performance data

    Note:
            Trajectories are only captured when capture_traces=True is passed to
        the evaluate() method, as they can be memory-intensive for large batches.
    """

    original_query: str
    reformulated_query: str
    retrieved_docs: list[dict[str, Any]]
    synthesized_context: str
    generated_answer: str
    execution_metadata: dict[str, Any]


class RAGOutput(TypedDict):
    """
    Final output from RAG system execution.

    This TypedDict represents the final result of RAG pipeline execution,
    containing both the generated answer and associated metadata.

    Attributes:
            final_answer (str): The generated answer from the RAG system
            confidence_score (float): Estimated confidence in the answer (0.0 to 1.0)
            based on retrieval quality and generation metrics
            retrieved_docs (List[Dict[str, Any]]): Documents that were retrieved
            and used for answer generation
        total_tokens (int): Estimated total token usage for the pipeline execution

    Example:
            .. code-block:: python

            output = RAGOutput(
                final_answer="Machine learning is a method of data analysis...",
                confidence_score=0.87,
                retrieved_docs=[{"content": "...", "score": 0.9}],
                total_tokens=450
            )
    """
```

```
    final_answer: str
    confidence_score: float
    retrieved_docs: list[dict[str, Any]]
    total_tokens: int


class GenericRAGAdapter(GEPAAdapter[RAGDataInst, RAGTrajectory, RAGOutput]):
    """
    Generic GEPA adapter for RAG system optimization with pluggable vector stores.

    This adapter enables GEPA's evolutionary prompt optimization to work with any
    vector store implementation through the VectorStoreInterface. It provides
    comprehensive evaluation of both retrieval and generation quality.

    Optimizable Components:
            - Query reformulation prompts: Improve query understanding and reformulation
            - Context synthesis prompts: Optimize document combination and summarization
            - Answer generation prompts: Enhance final answer quality and formatting
            - Reranking criteria: Improve document relevance ordering

    Evaluation Metrics:
            - Retrieval Quality: Precision, recall, F1, mean reciprocal rank (MRR)
            - Generation Quality: Token F1, BLEU score, faithfulness, answer relevance
        - Combined Score: Weighted combination for overall system performance

    Vector Store Support:
            Works with any vector store implementing VectorStoreInterface, including:
            ChromaDB, Weaviate, Qdrant, Pinecone, Milvus, and custom implementations.

    Example:
            .. code-block:: python

            from gepa.adapters.generic_rag_adapter import GenericRAGAdapter,
ChromaVectorStore
            import gepa

            vector_store = ChromaVectorStore.create_local("./kb", "docs")
            adapter = GenericRAGAdapter(vector_store=vector_store, llm_model="gpt-4")

            result = gepa.optimize(
                seed_candidate={"answer_generation": "Answer based on context:"},
                trainset=train_data,
                valset=val_data,
                adapter=adapter,
                max_metric_calls=50
            )
            print(result.best_candidate)  # Optimized prompts
    """

    def __init__(
        self,
        vector_store: VectorStoreInterface,
        llm_model,
        embedding_model: str = "text-embedding-3-small",
        embedding_function=None,
        rag_config: dict[str, Any] | None = None,
        failure_score: float = 0.0,
    ):
        """
        Initialize the GenericRAGAdapter for RAG system optimization.

        Args:
            vector_store: Vector store implementation (ChromaDB, Weaviate, etc.)
                Must implement VectorStoreInterface for similarity search operations.
            llm_model: LLM client for text generation. Can be:
                - String model name (uses litellm for inference)
                - Callable that takes messages and returns response text
                - Any object with a callable interface for LLM inference
```

```
            embedding_model: Model name for text embeddings (default: "text-embedding-3-
small").
                Used when embedding_function is not provided.
            embedding_function: Optional custom embedding function that takes text
                and returns List[float]. If None, uses default litellm embeddings.
            rag_config: RAG pipeline configuration dictionary. Keys include:
                - "retrieval_strategy": "similarity", "hybrid", or "vector"
                - "top_k": Number of documents to retrieve (default: 5)
                - "retrieval_weight": Weight for retrieval in combined score (default:
0.3)
                - "generation_weight": Weight for generation in combined score (default:
0.7)
                - "hybrid_alpha": Semantic vs keyword balance for hybrid search (default:
0.5)
                - "filters": Default metadata filters for retrieval
            failure_score: Score assigned when evaluation fails (default: 0.0)

        Example:
            .. code-block:: python

                vector_store = WeaviateVectorStore.create_local(collection_name="docs")
                adapter = GenericRAGAdapter(
                    vector_store=vector_store,
                    llm_model="gpt-4",
                    rag_config={
                        "retrieval_strategy": "hybrid",
                        "top_k": 5,
                        "hybrid_alpha": 0.7
                    }
                )
        """
        self.vector_store = vector_store
        self.rag_pipeline = RAGPipeline(
            vector_store=vector_store,
            llm_client=llm_model,
            embedding_model=embedding_model,
            embedding_function=embedding_function,
        )
        self.evaluator = RAGEvaluationMetrics()
        self.config = rag_config or self._default_config()
        self.failure_score = failure_score

    def evaluate(
        self,
        batch: list[RAGDataInst],
        candidate: dict[str, str],
        capture_traces: bool = False,
    ) -> EvaluationBatch[RAGTrajectory, RAGOutput]:
        """
        Evaluate RAG system performance on a batch of query-answer examples.

        This method runs the complete RAG pipeline on each example in the batch,
        evaluating both retrieval and generation quality using the provided
        prompt components.

        Args:
            batch: List of RAG evaluation examples, each containing:
                - query: Question to answer
                - ground_truth_answer: Expected correct answer
                - relevant_doc_ids: Documents that should be retrieved
                - metadata: Additional context for evaluation
            candidate: Dictionary mapping prompt component names to their text.
                Supported components:
                - "query_reformulation": Prompt for improving user queries
                - "context_synthesis": Prompt for combining retrieved documents
                - "answer_generation": Prompt for generating final answers
                - "reranking_criteria": Criteria for reordering retrieved documents
            capture_traces: If True, capture detailed execution trajectories
```

```
                for each example. Required for reflective dataset generation but
                increases memory usage.

        Returns:
            EvaluationBatch containing:
            - outputs: List of RAGOutput for each example
            - scores: List of combined quality scores (higher = better)
            - trajectories: List of detailed execution traces (if capture_traces=True)

        Raises:
            Exception: Individual example failures are caught and assigned failure_score.
                Only systemic failures (e.g., vector store unavailable) raise exceptions.

        Example:
            .. code-block:: python

                prompts = {
                    "answer_generation": "Answer the question based on this context:"
                }
                result = adapter.evaluate(
                    batch=validation_data,
                    candidate=prompts,
                    capture_traces=True
                )
                avg_score = sum(result.scores) / len(result.scores)
                print(f"Average RAG performance: {avg_score:.3f}")
        """
        outputs: list[RAGOutput] = []
        scores: list[float] = []
        trajectories: list[RAGTrajectory] | None = [] if capture_traces else None

        for data_inst in batch:
            try:
                # Execute RAG pipeline with candidate prompts
                rag_result = self.rag_pipeline.execute_rag(
                    query=data_inst["query"], prompts=candidate, config=self.config
                )

                # Evaluate retrieval quality
                retrieval_metrics = self.evaluator.evaluate_retrieval(
                    rag_result["retrieved_docs"], data_inst["relevant_doc_ids"]
                )

                # Evaluate generation quality
                generation_metrics = self.evaluator.evaluate_generation(
                    rag_result["generated_answer"], data_inst["ground_truth_answer"],
rag_result["synthesized_context"]
                )

                # Calculate combined score
                overall_score = self.evaluator.combined_rag_score(
                    retrieval_metrics,
                    generation_metrics,
                    retrieval_weight=self.config.get("retrieval_weight", 0.3),
                    generation_weight=self.config.get("generation_weight", 0.7),
                )

                # Prepare output
                output = RAGOutput(
                    final_answer=rag_result["generated_answer"],
                    confidence_score=generation_metrics.get("answer_confidence", 0.5),
                    retrieved_docs=rag_result["retrieved_docs"],
                    total_tokens=rag_result["metadata"]["total_tokens"],
                )

                outputs.append(output)
                scores.append(overall_score)
```

```
                # Capture trajectory if requested
                if capture_traces:
                    trajectory = RAGTrajectory(
                        original_query=rag_result["original_query"],
                        reformulated_query=rag_result["reformulated_query"],
                        retrieved_docs=rag_result["retrieved_docs"],
                        synthesized_context=rag_result["synthesized_context"],
                        generated_answer=rag_result["generated_answer"],
                        execution_metadata={
                            **rag_result["metadata"],
                            "retrieval_metrics": retrieval_metrics,
                            "generation_metrics": generation_metrics,
                            "overall_score": overall_score,
                        },
                    )
                    trajectories.append(trajectory)

            except Exception as e:
                # Handle individual example failure
                error_output = RAGOutput(
                    final_answer=f"Error: {e!s}", confidence_score=0.0, retrieved_docs=[],
total_tokens=0
                )

                outputs.append(error_output)
                scores.append(self.failure_score)

                if capture_traces:
                    error_trajectory = RAGTrajectory(
                        original_query=data_inst["query"],
                        reformulated_query=data_inst["query"],
                        retrieved_docs=[],
                        synthesized_context="",
                        generated_answer=f"Error: {e!s}",
                        execution_metadata={"error": str(e)},
                    )
                    trajectories.append(error_trajectory)

        return EvaluationBatch(outputs=outputs, scores=scores, trajectories=trajectories)

    def make_reflective_dataset(
        self,
        candidate: dict[str, str],
        eval_batch: EvaluationBatch[RAGTrajectory, RAGOutput],
        components_to_update: list[str],
    ) -> dict[str, list[dict[str, Any]]]:
        """
        Generate reflective dataset for evolutionary prompt optimization.

        This method analyzes the evaluation results and creates training examples
        that GEPA's proposer can use to improve the specified prompt components.
        Each component gets a tailored dataset with input-output pairs and feedback.

        Args:
            candidate: Current prompt components that were evaluated
            eval_batch: Evaluation results from evaluate() with capture_traces=True.
                Must contain trajectories for analysis.
            components_to_update: List of component names to generate improvement
                suggestions for. Must be subset of candidate.keys().

        Returns:
            Dictionary mapping component names to their reflective datasets.
            Each dataset is a list of examples with structure:
            - "Inputs": Input data for the component (query, docs, etc.)
            - "Generated Outputs": What the component currently produces
            - "Feedback": Analysis of performance and suggestions for improvement

        Example:
```

```
            .. code-block:: python

                reflective_data = adapter.make_reflective_dataset(
                    candidate=current_prompts,
                    eval_batch=evaluation_results,  # with trajectories
                    components_to_update=["answer_generation", "context_synthesis"]
                )
                print(reflective_data["answer_generation"][0]["Feedback"])
                # Output: "The generated answer lacks specific details from the
context..."

        Note:
            This method requires eval_batch to have been created with
            capture_traces=True, otherwise trajectories will be None.
        """
        reflective_data: dict[str, list[dict[str, Any]]] = {}

        for component in components_to_update:
            component_examples = []

            # Process each trajectory to create examples for this component
            for traj, output, score in zip(
                eval_batch.trajectories or [], eval_batch.outputs, eval_batch.scores,
strict=False
            ):
                example = self._create_component_example(component, traj, output, score,
candidate)
                if example:
                    component_examples.append(example)

            # Only include components that have examples
            if component_examples:
                reflective_data[component] = component_examples

        return reflective_data

    def _create_component_example(
        self, component_name: str, trajectory: RAGTrajectory, output: RAGOutput, score:
float, candidate: dict[str, str]
    ) -> dict[str, Any] | None:
        """Create a reflective example for a specific component."""

        if component_name == "query_reformulation":
            return {
                "Inputs": {
                    "original_query": trajectory["original_query"],
                    "current_prompt": candidate.get(component_name, ""),
                },
                "Generated Outputs": trajectory["reformulated_query"],
                "Feedback": self._generate_query_reformulation_feedback(trajectory,
score),
            }

        elif component_name == "context_synthesis":
            return {
                "Inputs": {
                    "query": trajectory["original_query"],
                    "retrieved_docs": [doc["content"] for doc in
trajectory["retrieved_docs"]],
                    "current_prompt": candidate.get(component_name, ""),
                },
                "Generated Outputs": trajectory["synthesized_context"],
                "Feedback": self._generate_context_synthesis_feedback(trajectory, score),
            }

        elif component_name == "answer_generation":
            return {
                "Inputs": {
```

```
                    "query": trajectory["original_query"],
                    "context": trajectory["synthesized_context"],
                    "current_prompt": candidate.get(component_name, ""),
                },
                "Generated Outputs": trajectory["generated_answer"],
                "Feedback": self._generate_answer_generation_feedback(trajectory, output,
score),
            }

        elif component_name == "reranking_criteria":
            return {
                "Inputs": {
                    "query": trajectory["original_query"],
                    "documents": [doc["content"] for doc in trajectory["retrieved_docs"]],
                    "current_criteria": candidate.get(component_name, ""),
                },
                "Generated Outputs": "Document ranking applied",
                "Feedback": self._generate_reranking_feedback(trajectory, score),
            }

            return None

    def _generate_query_reformulation_feedback(self, trajectory: RAGTrajectory, score:
float) -> str:
        """Generate feedback for query reformulation component."""
        if score > 0.7:
            return f"Good query reformulation. The reformulated query
'{trajectory['reformulated_query']}' helped retrieve relevant documents and generated a
good answer."
        else:
            return f"The query reformulation from '{trajectory['original_query']}' to
'{trajectory['reformulated_query']}' may not have improved retrieval. Consider making the
reformulated query more specific or preserving key terms."

    def _generate_context_synthesis_feedback(self, trajectory: RAGTrajectory, score:
float) -> str:
        """Generate feedback for context synthesis component."""
        if score > 0.7:
            return "Context synthesis worked well - the synthesized context effectively
supported answer generation."
        else:
            return "Context synthesis could be improved. The synthesized context may not
have highlighted the most relevant information or may have been too verbose/concise."

    def _generate_answer_generation_feedback(self, trajectory: RAGTrajectory, output:
RAGOutput, score: float) -> str:
        """Generate feedback for answer generation component."""
        if score > 0.7:
            return f"Good answer generation. The generated answer
'{trajectory['generated_answer']}' was accurate and well-supported by the context."
        else:
            return f"Answer generation needs improvement. The generated answer
'{trajectory['generated_answer']}' may not be fully accurate or well-supported by the
provided context."

    def _generate_reranking_feedback(self, trajectory: RAGTrajectory, score: float) ->
str:
        """Generate feedback for reranking criteria component."""
        if score > 0.7:
            return "Document reranking appears to have helped surface more relevant
documents for answer generation."
        else:
            return "Document reranking may not have improved relevance. Consider adjusting
the criteria to better prioritize documents that contain the answer."

    def _default_config(self) -> dict[str, Any]:
        """
        Get default configuration for RAG pipeline.
```

```
        Returns:
            Dictionary with default RAG configuration parameters:
            - retrieval_strategy: "similarity" (semantic search)
            - top_k: 5 (number of documents to retrieve)
            - retrieval_weight: 0.3 (30% weight for retrieval metrics)
            - generation_weight: 0.7 (70% weight for generation metrics)
            - hybrid_alpha: 0.5 (balanced semantic/keyword for hybrid search)
            - filters: None (no metadata filtering by default)
        """
        return {
            "retrieval_strategy": "similarity",
            "top_k": 5,
            "retrieval_weight": 0.3,
            "generation_weight": 0.7,
            "hybrid_alpha": 0.5,
            "filters": None,
        }
```

## src/gepa/adapters/generic_rag_adapter/rag_pipeline.py

```
# Copyright (c) 2025 Lakshya A Agrawal and the GEPA contributors
# https://github.com/gepa-ai/gepa

from typing import Any, Callable

from gepa.adapters.generic_rag_adapter.vector_store_interface import VectorStoreInterface


class RAGPipeline:
    """
    Generic RAG pipeline that works with any vector store.

    This pipeline orchestrates the full RAG process:
    1. Query reformulation (optional)
    2. Document retrieval
    3. Document reranking (optional)
    4. Context synthesis
    5. Answer generation
    """

    def __init__(
        self,
        vector_store: VectorStoreInterface,
        llm_client,
        embedding_model: str = "text-embedding-3-small",
        embedding_function: Callable[[str], list[float]] | None = None,
    ):
        """
        Initialize the RAG pipeline.

        Args:
            vector_store: Vector store interface implementation
            llm_client: LLM client for generation (should have a callable interface)
            embedding_model: Model name for embeddings (if using default embedding
function)
            embedding_function: Optional custom embedding function
        """
        self.vector_store = vector_store
        self.llm_client = llm_client
        self.embedding_model = embedding_model
        self.embedding_function = embedding_function

        # Initialize default embedding function if none provided
```

```
        if self.embedding_function is None:
            self.embedding_function = self._default_embedding_function

    def execute_rag(
        self,
        query: str,
        prompts: dict[str, str],
        config: dict[str, Any],
    ) -> dict[str, Any]:
        """
        Execute the full RAG pipeline with given prompts and configuration.

        Args:
            query: User query
            prompts: Dictionary of prompt templates for different stages
            config: Configuration parameters for retrieval and generation

        Returns:
            Dictionary containing all pipeline outputs and metadata
        """
        # Step 1: Query reformulation (if enabled)
        reformulated_query = query
        if "query_reformulation" in prompts and prompts["query_reformulation"].strip():
            reformulated_query = self._reformulate_query(query,
prompts["query_reformulation"])

        # Step 2: Retrieval
        retrieved_docs = self._retrieve_documents(reformulated_query, config)

        # Step 3: Reranking (if enabled)
        if "reranking_criteria" in prompts and prompts["reranking_criteria"].strip():
            retrieved_docs = self._rerank_documents(retrieved_docs, query,
prompts["reranking_criteria"], config)

        # Step 4: Context synthesis
        context = self._synthesize_context(retrieved_docs, query,
prompts.get("context_synthesis", ""))

        # Step 5: Answer generation
        answer = self._generate_answer(query, context, prompts.get("answer_generation",
""))

        return {
            "original_query": query,
            "reformulated_query": reformulated_query,
            "retrieved_docs": retrieved_docs,
            "synthesized_context": context,
            "generated_answer": answer,
            "metadata": {
                "retrieval_count": len(retrieved_docs),
                "total_tokens": self._estimate_token_count(context + answer),
                "vector_store_type":
self.vector_store.get_collection_info().get("vector_store_type", "unknown"),
            },
        }

    def _reformulate_query(self, query: str, reformulation_prompt: str) -> str:
        """Reformulate the user query using the provided prompt."""
        messages = [
            {"role": "system", "content": reformulation_prompt},
            {"role": "user", "content": f"Original query: {query}"},
        ]

        try:
            if callable(self.llm_client):
                response = self.llm_client(messages)
            else:
                # Assume it's a litellm-style client
```

```
                response =
self.llm_client.completion(messages=messages).choices[0].message.content

            return response.strip() if response else query
        except Exception:
            # Fallback to original query if reformulation fails
            return query

    def _retrieve_documents(self, query: str, config: dict[str, Any]) -> list[dict[str,
Any]]:
        """Retrieve documents using the configured strategy."""
        retrieval_strategy = config.get("retrieval_strategy", "similarity")
        k = config.get("top_k", 5)
        filters = config.get("filters", None)

        if retrieval_strategy == "similarity":
            return self.vector_store.similarity_search(query, k=k, filters=filters)
        elif retrieval_strategy == "hybrid":
            if self.vector_store.supports_hybrid_search():
                alpha = config.get("hybrid_alpha", 0.5)
                return self.vector_store.hybrid_search(query, k=k, alpha=alpha,
filters=filters)
            else:
                # Fallback to similarity search
                return self.vector_store.similarity_search(query, k=k, filters=filters)
        elif retrieval_strategy == "vector":
            # Use pre-computed embedding
            query_vector = self.embedding_function(query)
            return self.vector_store.vector_search(query_vector, k=k, filters=filters)
        else:
            raise ValueError(f"Unknown retrieval strategy: {retrieval_strategy}")

    def _rerank_documents(
        self, documents: list[dict[str, Any]], query: str, reranking_prompt: str, config:
dict[str, Any]
    ) -> list[dict[str, Any]]:
        """Rerank documents based on relevance criteria."""
        if not documents:
            return documents

        # For simplicity, we'll use a prompt-based reranking approach
        # In production, you might use dedicated reranking models
        try:
            doc_texts = [f"Document {i + 1}: {doc['content']}" for i, doc in
enumerate(documents)]
            doc_context = "\n\n".join(doc_texts)

            messages = [
                {"role": "system", "content": reranking_prompt},
                {
                    "role": "user",
                    "content": f"Query: {query}\n\nDocuments:\n{doc_context}\n\nPlease
rank these documents by relevance (return document numbers in order, e.g., '3,1,4,2,5'):",
                },
            ]

            if callable(self.llm_client):
                response = self.llm_client(messages)
            else:
                response =
self.llm_client.completion(messages=messages).choices[0].message.content

            # Parse the ranking response
            ranking_str = response.strip()
            rankings = [int(x.strip()) - 1 for x in ranking_str.split(",") if
x.strip().isdigit()]

            # Reorder documents based on ranking
```

```
            if len(rankings) == len(documents):
                return [documents[i] for i in rankings if 0 <= i < len(documents)]
        except Exception:
            pass

        # Return original order if reranking fails
        return documents

    def _synthesize_context(self, documents: list[dict[str, Any]], query: str,
synthesis_prompt: str) -> str:
        """Synthesize retrieved documents into coherent context."""
        if not documents:
            return ""

        if not synthesis_prompt.strip():
            # Default: simple concatenation
            contexts = []
            for i, doc in enumerate(documents):
                contexts.append(f"[Document {i + 1}] {doc['content']}")
            return "\n\n".join(contexts)

        # Use LLM for context synthesis
        doc_texts = [doc["content"] for doc in documents]
        doc_context = "\n\n".join(f"Document {i + 1}: {text}" for i, text in
enumerate(doc_texts))

        messages = [
            {"role": "system", "content": synthesis_prompt},
            {"role": "user", "content": f"Query: {query}\n\nRetrieved
Documents:\n{doc_context}"},
        ]

        try:
            if callable(self.llm_client):
                response = self.llm_client(messages)
            else:
                response =
self.llm_client.completion(messages=messages).choices[0].message.content

            return response.strip() if response else doc_context
        except Exception:
            # Fallback to simple concatenation
            return doc_context

    def _generate_answer(self, query: str, context: str, generation_prompt: str) -> str:
        """Generate the final answer using the query and synthesized context."""
        if not generation_prompt.strip():
            generation_prompt = "You are a helpful assistant. Answer the user's question
based on the provided context."

        messages = [
            {"role": "system", "content": generation_prompt},
            {"role": "user", "content": f"Context:\n{context}\n\nQuestion: {query}"},
        ]

        try:
            if callable(self.llm_client):
                response = self.llm_client(messages)
            else:
                response =
self.llm_client.completion(messages=messages).choices[0].message.content

            return response.strip() if response else "I couldn't generate an answer based
on the provided context."
        except Exception as e:
            return f"Error generating answer: {e!s}"

    def _default_embedding_function(self, text: str) -> list[float]:
```

```
        """Default embedding function using litellm."""
        try:
            import litellm

            response = litellm.embedding(model=self.embedding_model, input=text)
            return response.data[0].embedding
        except Exception as e:
            raise RuntimeError(f"Failed to generate embeddings: {e!s}") from e

    def _estimate_token_count(self, text: str) -> int:
        """Rough estimate of token count (4 chars per token approximation)."""
        return len(text) // 4
```

## src/gepa/adapters/generic_rag_adapter/vector_store_interface.py

```
# Copyright (c) 2025 Lakshya A Agrawal and the GEPA contributors
# https://github.com/gepa-ai/gepa

from abc import ABC, abstractmethod
from typing import Any


class VectorStoreInterface(ABC):
    """
    Abstract interface for vector store operations in RAG systems.

    This interface defines the core operations needed for retrieval-augmented generation,
    enabling GEPA to work with any vector store implementation (ChromaDB, Weaviate,
Qdrant,
    Pinecone, Milvus, etc.) through a unified API.

    The interface supports:
    - Semantic similarity search
    - Vector-based search with pre-computed embeddings
    - Hybrid search (semantic + keyword, where supported)
    - Metadata filtering and collection introspection

    Implementing this interface allows your vector store to be used with GEPA's
    evolutionary prompt optimization for RAG systems.

    Example:
        .. code-block:: python

            class MyVectorStore(VectorStoreInterface):
                def similarity_search(self, query, k=5, filters=None):
                    # Your implementation
                    return documents

            vector_store = MyVectorStore()
            adapter = GenericRAGAdapter(vector_store=vector_store)
    """

    @abstractmethod
    def similarity_search(
        self,
        query: str,
        k: int = 5,
        filters: dict[str, Any] | None = None,
    ) -> list[dict[str, Any]]:
        """
        Search for documents semantically similar to the query text.

        This method performs semantic similarity search using the vector store's
        default embedding model or configured vectorizer.
```

```
        Args:
            query: Text query to search for similar documents
            k: Maximum number of documents to return (default: 5)
            filters: Optional metadata filters to constrain search.
                Format: {"key": "value"} or {"key": {"$op": value}}

        Returns:
            List of documents ordered by similarity score (highest first).
            Each document is a dictionary with:
            - "content" (str): The document text content
            - "metadata" (dict): Document metadata including any doc_id
            - "score" (float): Similarity score between 0.0 and 1.0 (higher = more
similar)

        Raises:
            NotImplementedError: Must be implemented by concrete vector store classes

        Example:
            .. code-block:: python

                results = vector_store.similarity_search(
                    query="machine learning algorithms",
                    k=3,
                    filters={"category": "AI"}
                )
                print(results[0]["content"])  # Most similar document
        """
        pass

    @abstractmethod
    def vector_search(
        self,
        query_vector: list[float],
        k: int = 5,
        filters: dict[str, Any] | None = None,
    ) -> list[dict[str, Any]]:
        """
        Search using a pre-computed query embedding vector.

        This method allows direct vector similarity search when you already have
        the query embedding, avoiding the need for additional embedding computation.

        Args:
            query_vector: Pre-computed embedding vector for the query.
                Must match the dimensionality of vectors in the collection.
            k: Maximum number of documents to return (default: 5)
            filters: Optional metadata filters to constrain search

        Returns:
            List of documents ordered by vector similarity (highest first).
            Same format as similarity_search().

        Raises:
            NotImplementedError: Must be implemented by concrete vector store classes
            ValueError: If query_vector dimensions don't match collection

        Example:
            .. code-block:: python

                import numpy as np
                query_vector = embedding_model.encode("machine learning")
                results = vector_store.vector_search(query_vector.tolist(), k=5)
        """
        pass

    def hybrid_search(
        self,
```

```
        query: str,
        k: int = 5,
        alpha: float = 0.5,
        filters: dict[str, Any] | None = None,
    ) -> list[dict[str, Any]]:
        """
        Hybrid semantic + keyword search combining vector and text-based matching.

        This method combines semantic similarity (vector search) with keyword-based
        search (like BM25) to leverage both approaches. The alpha parameter controls
        the balance between the two search methods.

        Args:
            query: Text query to search for
            k: Maximum number of documents to return (default: 5)
            alpha: Weight for semantic vs keyword search (default: 0.5)
                - 0.0 = pure keyword/BM25 search
                - 1.0 = pure semantic/vector search
                - 0.5 = balanced hybrid search
            filters: Optional metadata filters to constrain search

        Returns:
            List of documents ordered by hybrid similarity score (highest first).
            Same format as similarity_search().

        Note:
            If hybrid search is not supported by the vector store implementation,
            this method falls back to similarity_search() with a warning.
            Use supports_hybrid_search() to check availability.

        Example:
            .. code-block:: python

                # Balanced hybrid search
                results = vector_store.hybrid_search("AI algorithms", alpha=0.5)

                # More semantic-focused
                results = vector_store.hybrid_search("AI algorithms", alpha=0.8)
        """
        # Default fallback implementation
        return self.similarity_search(query, k, filters)

    @abstractmethod
    def get_collection_info(self) -> dict[str, Any]:
        """
        Get metadata and statistics about the vector store collection.

        This method provides introspection capabilities for the collection,
        returning key information about its configuration and contents.

        Returns:
            Dictionary containing collection metadata with keys:
            - "name" (str): Collection/index name
            - "document_count" (int): Total number of documents
            - "dimension" (int): Vector embedding dimension (0 if unknown)
            - "vector_store_type" (str): Type of vector store (e.g., "chromadb",
"weaviate")
            - Additional store-specific metadata as available

        Raises:
            NotImplementedError: Must be implemented by concrete vector store classes

        Example:
            .. code-block:: python

                info = vector_store.get_collection_info()
                print(f"Collection {info['name']} has {info['document_count']} documents")
                print(f"Vector dimension: {info['dimension']}")
```

```
        """
        pass

    def get_embedding_dimension(self) -> int:
        """
        Get the embedding dimension of the vector store.

        Returns:
            Dimension of the vectors in the collection
        """
        info = self.get_collection_info()
        return info.get("dimension", 0)

    def supports_hybrid_search(self) -> bool:
        """
        Check if the vector store supports hybrid search.

        Returns:
            True if hybrid search is supported, False otherwise
        """
        return False

    def supports_metadata_filtering(self) -> bool:
        """
        Check if the vector store supports metadata filtering.

        Returns:
            True if metadata filtering is supported, False otherwise
        """
        return True
```

### src/gepa/adapters/generic_rag_adapter/vector_stores/__init__.py

```
# Copyright (c) 2025 Lakshya A Agrawal and the GEPA contributors
# https://github.com/gepa-ai/gepa
```

### src/gepa/adapters/generic_rag_adapter/vector_stores/chroma_store.py

```
# Copyright (c) 2025 Lakshya A Agrawal and the GEPA contributors
# https://github.com/gepa-ai/gepa

from typing import Any

from gepa.adapters.generic_rag_adapter.vector_store_interface import VectorStoreInterface


class ChromaVectorStore(VectorStoreInterface):
    """
    ChromaDB implementation of the VectorStoreInterface.

    ChromaDB is an open-source embedding database that's easy to use
    and perfect for local development and prototyping.
    """

    def __init__(self, client, collection_name: str, embedding_function=None):
        """
        Initialize ChromaVectorStore.

        Args:
```

```
            client: ChromaDB client instance
            collection_name: Name of the collection to use
            embedding_function: Optional embedding function for text queries
        """
        import importlib.util

        if importlib.util.find_spec("chromadb") is None:
            raise ImportError("ChromaDB is required for ChromaVectorStore. Install with:
pip install litellm chromadb")

        self.client = client
        self.collection_name = collection_name
        self.embedding_function = embedding_function

        # Get or create the collection
        try:
            self.collection = self.client.get_collection(name=collection_name,
embedding_function=embedding_function)
        except Exception:
            # Collection doesn't exist, create it
            self.collection = self.client.create_collection(name=collection_name,
embedding_function=embedding_function)

    def similarity_search(
        self,
        query: str,
        k: int = 5,
        filters: dict[str, Any] | None = None,
    ) -> list[dict[str, Any]]:
        """Search for documents similar to the query text."""
        # Convert filters to ChromaDB format if provided
        where_clause = self._convert_filters(filters) if filters else None

        results = self.collection.query(
            query_texts=[query], n_results=k, where=where_clause, include=["documents",
"metadatas", "distances"]
        )

        return self._format_results(results)

    def vector_search(
        self,
        query_vector: list[float],
        k: int = 5,
        filters: dict[str, Any] | None = None,
    ) -> list[dict[str, Any]]:
        """Search using a pre-computed query vector."""
        where_clause = self._convert_filters(filters) if filters else None

        results = self.collection.query(
            query_embeddings=[query_vector],
            n_results=k,
            where=where_clause,
            include=["documents", "metadatas", "distances"],
        )

        return self._format_results(results)

    def get_collection_info(self) -> dict[str, Any]:
        """Get metadata about the ChromaDB collection."""
        count = self.collection.count()

        # Try to get a sample document to determine embedding dimension
        dimension = 0
        if count > 0:
            sample = self.collection.peek(limit=1)
            embeddings = sample.get("embeddings")
            if embeddings is not None and len(embeddings) > 0:
```

```
                dimension = len(embeddings[0])

        return {
            "name": self.collection_name,
            "document_count": count,
            "dimension": dimension,
            "vector_store_type": "chromadb",
        }

    def supports_metadata_filtering(self) -> bool:
        """ChromaDB supports metadata filtering."""
        return True

    def _convert_filters(self, filters: dict[str, Any]) -> dict[str, Any]:
        """
        Convert generic filters to ChromaDB where clause format.

        Generic format: {"key": "value", "key2": {"$gt": 5}}
        ChromaDB format: {"key": {"$eq": "value"}, "key2": {"$gt": 5}}
        """
        chroma_filters = {}

        for key, value in filters.items():
            if isinstance(value, dict):
                # Already in operator format
                chroma_filters[key] = value
            else:
                # Convert to equality operator
                chroma_filters[key] = {"$eq": value}

        return chroma_filters

    def _format_results(self, results) -> list[dict[str, Any]]:
        """Convert ChromaDB results to standardized format."""
        documents = []

        if not results["documents"] or not results["documents"][0]:
            return documents

        docs = results["documents"][0]
        metadatas = results.get("metadatas", [None] * len(docs))[0] or [{}] * len(docs)
        distances = results.get("distances", [0.0] * len(docs))[0]

        for doc, metadata, distance in zip(docs, metadatas, distances, strict=False):
            # Convert distance to similarity score (higher is better)
            # ChromaDB uses cosine distance, so similarity = 1 - distance
            distance_val = self._extract_distance_value(distance)
            similarity_score = max(0.0, 1.0 - distance_val)

            documents.append({"content": doc, "metadata": metadata or {}, "score":
similarity_score})

        return documents

    def _extract_distance_value(self, distance) -> float:
        """
        Helper to extract a scalar float from a distance value returned by ChromaDB.
        Handles numpy scalars, single-element lists/arrays, and plain floats.
        """
        try:
            if hasattr(distance, "item"):  # numpy scalar
                return distance.item()
            elif hasattr(distance, "__len__") and not isinstance(distance, str | bytes)
and len(distance) == 1:
                return float(distance[0])
            else:
                return float(distance)
        except (TypeError, ValueError, IndexError) as e:
```

```
            import logging

            logging.warning(f"Unexpected distance format: {type(distance)}, value:
{distance}, error: {e}")
            return 0.0  # Default fallback

    @classmethod
    def create_local(cls, persist_directory: str, collection_name: str,
embedding_function=None) -> "ChromaVectorStore":
        """
        Create a ChromaVectorStore with local persistence.

        Args:
            persist_directory: Directory to persist the database
            collection_name: Name of the collection
            embedding_function: Optional embedding function

        Returns:
            ChromaVectorStore instance
        """
        try:
            import chromadb
        except ImportError as e:
            raise ImportError("ChromaDB is required. Install with: pip install litellm
chromadb") from e

        client = chromadb.PersistentClient(path=persist_directory)
        return cls(client, collection_name, embedding_function)

    @classmethod
    def create_memory(cls, collection_name: str, embedding_function=None) ->
"ChromaVectorStore":
        """
        Create a ChromaVectorStore in memory (for testing).

        Args:
            collection_name: Name of the collection
            embedding_function: Optional embedding function

        Returns:
            ChromaVectorStore instance
        """
        try:
            import chromadb
        except ImportError as e:
            raise ImportError("ChromaDB is required. Install with: pip install litellm
chromadb") from e

        client = chromadb.Client()
        return cls(client, collection_name, embedding_function)
```

### src/gepa/adapters/generic_rag_adapter/vector_stores/lancedb_store.py

```
# Copyright (c) 2025 Lakshya A Agrawal and the GEPA contributors
# https://github.com/gepa-ai/gepa

from typing import Any

from gepa.adapters.generic_rag_adapter.vector_store_interface import VectorStoreInterface


class LanceDBVectorStore(VectorStoreInterface):
    """
    LanceDB implementation of the VectorStoreInterface.
```

```
    LanceDB is a developer-friendly, serverless vector database for AI applications.
    It provides excellent performance for both local development and production
    deployments with support for SQL-like filtering and modern PyArrow integration.
    """

    def __init__(self, db, table_name: str, embedding_function=None):
        """
        Initialize LanceDBVectorStore.

        Args:
            db: LanceDB database connection
            table_name: Name of the table to use
            embedding_function: Optional function to compute embeddings for queries
        """
        import importlib.util

        if importlib.util.find_spec("lancedb") is None:
            raise ImportError(
                "LanceDB is required for LanceDBVectorStore. Install with: pip install
litellm lancedb pyarrow"
            )

        self.db = db
        self.table_name = table_name
        self.embedding_function = embedding_function

        # Try to open table if it exists, otherwise it will be created in add_documents
        try:
            self.table = self.db.open_table(table_name)
        except Exception:
            self.table = None  # Will be created when first adding documents

    def similarity_search(
        self,
        query: str,
        k: int = 5,
        filters: dict[str, Any] | None = None,
    ) -> list[dict[str, Any]]:
        """Search for documents similar to the query text using embeddings."""
        if self.embedding_function is None:
            raise ValueError("No embedding function provided for similarity search")

        # Compute embeddings for the query
        try:
            query_vector = self.embedding_function(query)
            if hasattr(query_vector, "tolist"):
                query_vector = query_vector.tolist()
        except Exception as e:
            raise RuntimeError(f"Failed to compute embeddings for query: {e!s}") from e

        # Use vector search with computed embeddings
        return self.vector_search(query_vector, k, filters)

    def vector_search(
        self,
        query_vector: list[float],
        k: int = 5,
        filters: dict[str, Any] | None = None,
    ) -> list[dict[str, Any]]:
        """Search using a pre-computed query vector."""
        if self.table is None:
            return []  # No documents added yet

        try:
            # Build query with vector search
            query_builder = self.table.search(query_vector).limit(k)
```

```
            # Add filters if provided
            if filters:
                filter_expr = self._convert_filters(filters)
                if filter_expr:
                    query_builder = query_builder.where(filter_expr)

            # Execute query and get results
            results = query_builder.to_pandas()

            return self._format_results(results)

        except Exception as e:
            raise RuntimeError(f"LanceDB vector search failed: {e!s}") from e

    def add_documents(
        self,
        documents: list[dict[str, Any]],
        embeddings: list[list[float]],
        ids: list[str] | None = None,
    ) -> list[str]:
        """Add documents with their embeddings to the table."""
        if len(documents) != len(embeddings):
            raise ValueError("Number of documents must match number of embeddings")

        # Generate IDs if not provided
        if ids is None:
            ids = [f"doc_{i}" for i in range(len(documents))]
        elif len(ids) != len(documents):
            raise ValueError("Number of IDs must match number of documents")

        # Prepare data for insertion
        data_to_insert = []
        for doc_id, doc, embedding in zip(ids, documents, embeddings, strict=False):
            # LanceDB requires consistent field structure
            record = {
                "id": doc_id,
                "vector": embedding,
                **doc,  # Include all document fields
            }
            data_to_insert.append(record)

        try:
            # Create table if it doesn't exist yet
            if self.table is None:
                self.table = self.db.create_table(self.table_name, data=data_to_insert)
            else:
                # Add data to existing table
                self.table.add(data_to_insert, mode="append")
            return ids

        except Exception as e:
            raise RuntimeError(f"Failed to add documents to LanceDB: {e!s}") from e

    def delete_documents(self, ids: list[str]) -> bool:
        """Delete documents by their IDs."""
        try:
            # Use parameterized filter to avoid injection
            if len(ids) == 1:
                filter_expr = {"id": ids[0]}
            else:
                filter_expr = {"id": {"$in": ids}}

            # Delete documents
            self.table.delete(filter_expr)
            return True

        except Exception as e:
            raise RuntimeError(f"Failed to delete documents from LanceDB: {e!s}") from e
```

```
    def get_collection_info(self) -> dict[str, Any]:
        """Get metadata about the LanceDB table."""
        try:
            # Get table schema
            schema = self.table.schema

            # Count rows (this might be approximate for large tables)
            try:
                count_result = self.table.count_rows()
                row_count = count_result if isinstance(count_result, int) else 0
            except Exception:
                # Fallback: count by querying
                try:
                    sample_df = self.table.to_pandas()
                    row_count = len(sample_df)
                except Exception:
                    row_count = 0

            # Extract vector field information
            vector_field = None
            dimension = 0
            for field in schema:
                # Check if field is a list type (vector field)
                if hasattr(field.type, "value_type") and "float" in
str(field.type).lower():
                    vector_field = field.name
                    # Try to get dimension from list type
                    if hasattr(field.type, "list_size"):
                        dimension = field.type.list_size
                    elif "vector" in field.name.lower():
                        # This is likely our vector field
                        vector_field = field.name

            # Get table version and other metadata
            version = getattr(self.table, "version", "unknown")

            return {
                "name": self.table_name,
                "document_count": row_count,
                "dimension": dimension,
                "vector_store_type": "lancedb",
                "vector_field": vector_field,
                "version": version,
                "schema": str(schema),
            }

        except Exception as e:
            # Fallback info if detailed info fails
            return {
                "name": self.table_name,
                "document_count": 0,
                "dimension": 0,
                "vector_store_type": "lancedb",
                "error": str(e),
            }

    def supports_hybrid_search(self) -> bool:
        """LanceDB supports hybrid search through full-text search + vector search."""
        return True

    def hybrid_search(
        self,
        query: str,
        k: int = 5,
        alpha: float = 0.5,
        filters: dict[str, Any] | None = None,
    ) -> list[dict[str, Any]]:
```

```
        """
        Hybrid search combining vector similarity and full-text search.
        """
        try:
            # Import FTS query if available
            try:
                from lancedb.query import FtsQuery

                # Build hybrid query
                FtsQuery(query=query)

                # Get embedding for vector search
                if self.embedding_function:
                    query_vector = self.embedding_function(query)
                    if hasattr(query_vector, "tolist"):
                        query_vector = query_vector.tolist()

                    # Perform hybrid search (this is a simplified version)
                    # In practice, you might want to combine scores differently
                    query_builder = self.table.search(query_vector,
query_type="hybrid").limit(k)
                else:
                    # Fall back to FTS only
                    query_builder = self.table.search(query, query_type="fts").limit(k)

                # Add filters if provided
                if filters:
                    filter_expr = self._convert_filters(filters)
                    if filter_expr:
                        query_builder = query_builder.where(filter_expr)

                # Execute query
                results = query_builder.to_pandas()
                return self._format_results(results)

            except ImportError:
                # Fall back to vector search only
                return self.similarity_search(query, k, filters)

        except Exception:
            # Fall back to regular vector search
            return self.similarity_search(query, k, filters)

    def _format_results(self, results_df) -> list[dict[str, Any]]:
        """Convert LanceDB results to standardized format."""
        documents = []

        if results_df is None or len(results_df) == 0:
            return documents

        for _, row in results_df.iterrows():
            row_dict = row.to_dict()

            # Extract content - try different field names
            content = ""
            content_fields = ["content", "text", "document", "body", "description"]
            for field in content_fields:
                if row_dict.get(field):
                    content = str(row_dict[field])
                    break

            # If no content field found, combine text fields
            if not content:
                text_properties = []
                system_fields = ["id", "vector", "_distance"]
                for key, value in row_dict.items():
                    if (
                        isinstance(value, str)
```

```
                        and value.strip()
                        and key not in system_fields
                        and key not in content_fields
                    ):
                        text_properties.append(f"{key}: {value}")
                content = " | ".join(text_properties)

            # Create metadata (all fields except content and system fields)
            metadata = {}
            system_fields = ["id", "vector", "_distance"] + content_fields
            for key, value in row_dict.items():
                if key not in system_fields and value is not None:
                    # Convert numpy types to Python types for JSON serialization
                    if hasattr(value, "item"):
                        value = value.item()
                    elif hasattr(value, "tolist"):
                        value = value.tolist()
                    metadata[key] = value

            metadata["doc_id"] = str(row_dict.get("id", ""))

            # Extract similarity score (LanceDB includes _distance column)
            distance = row_dict.get("_distance", 0.0)
            if hasattr(distance, "item"):
                distance = distance.item()

            # Convert distance to similarity score (lower distance = higher similarity)
            # For L2 distance: similarity = 1 / (1 + distance)
            # For cosine distance: similarity = 1 - distance (if distance is in [0,2])
            if distance <= 1.0:
                score = max(0.0, 1.0 - distance)  # Cosine-like
            else:
                score = 1.0 / (1.0 + distance)  # L2-like

            documents.append(
                {
                    "content": content,
                    "metadata": metadata,
                    "score": float(score),
                }
            )

        return documents

    def _convert_filters(self, filters: dict[str, Any]) -> str:
        """Convert generic filters to LanceDB SQL-like expressions."""
        if not filters:
            return None

        expressions = []

        for key, value in filters.items():
            if isinstance(value, str):
                # String exact match - properly escape quotes and backslashes
                escaped_value = value.replace("\\", "\\\\").replace("'", "''")
                expressions.append(f"{key} = '{escaped_value}'")
            elif isinstance(value, int | float):
                # Numeric exact match
                expressions.append(f"{key} = {value}")
            elif isinstance(value, bool):
                # Boolean match
                expressions.append(f"{key} = {value}")
            elif isinstance(value, list):
                # IN clause for multiple values
                if all(isinstance(v, str) for v in value):
                    # String values - properly escape
                    escaped_values = []
                    for v in value:
```

```
                        escaped_v = v.replace("'", "''")
                        escaped_values.append(f"'{escaped_v}'")
                    values_str = ", ".join(escaped_values)
                    expressions.append(f"{key} IN ({values_str})")
                else:
                    # Numeric values
                    values_str = ", ".join(str(v) for v in value)
                    expressions.append(f"{key} IN ({values_str})")
            elif isinstance(value, dict):
                # Range queries
                range_conditions = []
                if "gte" in value:
                    range_conditions.append(f"{key} >= {value['gte']}")
                if "gt" in value:
                    range_conditions.append(f"{key} > {value['gt']}")
                if "lte" in value:
                    range_conditions.append(f"{key} <= {value['lte']}")
                if "lt" in value:
                    range_conditions.append(f"{key} < {value['lt']}")

                if range_conditions:
                    expressions.append("(" + " AND ".join(range_conditions) + ")")

        if expressions:
            return " AND ".join(expressions)

        return None

    @classmethod
    def create_local(cls, table_name: str, embedding_function=None, db_path: str =
"./lancedb", vector_size: int = 384):
        """Create a local LanceDB vector store."""
        import importlib.util

        if importlib.util.find_spec("lancedb") is None or
importlib.util.find_spec("pyarrow") is None:
            raise ImportError("LanceDB and PyArrow are required. Install with: pip install
litellm lancedb pyarrow")

        import lancedb

        # Connect to local database
        db = lancedb.connect(db_path)

        # For LanceDB, we'll create the table when first adding documents
        # This allows LanceDB to infer the schema from actual data, avoiding conflicts

        return cls(db, table_name, embedding_function)

    @classmethod
    def create_remote(
        cls,
        table_name: str,
        embedding_function=None,
        uri: str | None = None,
        api_key: str | None = None,
        region: str = "us-east-1",
        vector_size: int = 384,
    ):
        """Create a remote LanceDB vector store (LanceDB Cloud)."""
        import importlib.util

        if importlib.util.find_spec("lancedb") is None or
importlib.util.find_spec("pyarrow") is None:
            raise ImportError("LanceDB and PyArrow are required. Install with: pip install
litellm lancedb pyarrow")

        import lancedb
```

```
        if not uri or not api_key:
            raise ValueError("URI and API key are required for remote LanceDB connection")

        # Connect to remote database
        db = lancedb.connect(uri, api_key=api_key, region=region)

        # For LanceDB, we'll create the table when first adding documents
        # This allows LanceDB to infer the schema from actual data, avoiding conflicts

        return cls(db, table_name, embedding_function)
```

### src/gepa/adapters/generic_rag_adapter/vector_stores/milvus_store.py

```
# Copyright (c) 2025 Lakshya A Agrawal and the GEPA contributors
# https://github.com/gepa-ai/gepa

from typing import Any

from gepa.adapters.generic_rag_adapter.vector_store_interface import VectorStoreInterface


class MilvusVectorStore(VectorStoreInterface):
    """
    Milvus implementation of the VectorStoreInterface.

    Milvus is a cloud-native vector database built for scalable similarity search
    and AI applications. It provides excellent performance for large-scale deployments
    and supports various index types and distance metrics.
    """

    def __init__(self, client, collection_name: str, embedding_function=None):
        """
        Initialize MilvusVectorStore.

        Args:
            client: MilvusClient instance
            collection_name: Name of the collection to use
            embedding_function: Optional function to compute embeddings for queries
        """
        import importlib.util

        if importlib.util.find_spec("pymilvus") is None:
            raise ImportError(
                "Milvus client is required for MilvusVectorStore. Install with: pip
install litellm pymilvus"
            )

        self.client = client
        self.collection_name = collection_name
        self.embedding_function = embedding_function

        # Verify collection exists
        if not self.client.has_collection(collection_name):
            raise ValueError(f"Collection '{collection_name}' not found. Please create the
collection first.")

        # Load collection into memory for search operations
        try:
            self.client.load_collection(collection_name)
        except Exception:
            # Collection might already be loaded, which is fine
            pass
```

```
    def similarity_search(
        self,
        query: str,
        k: int = 5,
        filters: dict[str, Any] | None = None,
    ) -> list[dict[str, Any]]:
        """Search for documents similar to the query text using embeddings."""
        if self.embedding_function is None:
            raise ValueError("No embedding function provided for similarity search")

        # Compute embeddings for the query
        try:
            query_vector = self.embedding_function(query)
            if hasattr(query_vector, "tolist"):
                query_vector = query_vector.tolist()
        except Exception as e:
            raise RuntimeError(f"Failed to compute embeddings for query: {e!s}") from e

        # Use vector search with computed embeddings
        return self.vector_search(query_vector, k, filters)

    def vector_search(
        self,
        query_vector: list[float],
        k: int = 5,
        filters: dict[str, Any] | None = None,
    ) -> list[dict[str, Any]]:
        """Search using a pre-computed query vector."""
        try:
            # Convert filters to Milvus expression format
            filter_expr = self._convert_filters(filters) if filters else None

            # Perform vector search
            results = self.client.search(
                collection_name=self.collection_name,
                data=[query_vector],  # Milvus expects list of vectors
                limit=k,
                filter=filter_expr,
                output_fields=["*"],  # Return all fields
            )

            return self._format_results(results)

        except Exception as e:
            raise RuntimeError(f"Milvus vector search failed: {e!s}") from e

    def add_documents(
        self,
        documents: list[dict[str, Any]],
        embeddings: list[list[float]],
        ids: list[str] | None = None,
    ) -> list[str]:
        """Add documents with their embeddings to the collection."""
        if len(documents) != len(embeddings):
            raise ValueError("Number of documents must match number of embeddings")

        # Generate IDs if not provided
        if ids is None:
            ids = [f"doc_{i}" for i in range(len(documents))]
        elif len(ids) != len(documents):
            raise ValueError("Number of IDs must match number of documents")

        # Prepare data for insertion
        data_to_insert = []
        for doc_id, doc, embedding in zip(ids, documents, embeddings, strict=False):
            # Milvus requires consistent field structure
            record = {
                "id": doc_id,
```

```
                "vector": embedding,
                **doc,  # Include all document fields
            }
            data_to_insert.append(record)

        try:
            # Insert data into collection
            result = self.client.insert(collection_name=self.collection_name,
data=data_to_insert)

            # Check if insertion was successful
            if "insert_count" in result and result["insert_count"] == len(data_to_insert):
                return ids
            else:
                raise RuntimeError(
                    f"Insertion failed. Expected {len(data_to_insert)}, got
{result.get('insert_count', 0)}"
                )

        except Exception as e:
            raise RuntimeError(f"Failed to add documents to Milvus: {e!s}") from e

    def delete_documents(self, ids: list[str]) -> bool:
        """Delete documents by their IDs."""
        try:
            # Use parameterized ID-based deletion to avoid injection
            result = self.client.delete(collection_name=self.collection_name, ids=ids)

            return "delete_count" in result and result["delete_count"] > 0

        except Exception as e:
            raise RuntimeError(f"Failed to delete documents from Milvus: {e!s}") from e

    def get_collection_info(self) -> dict[str, Any]:
        """Get metadata about the Milvus collection."""
        try:
            # Get collection description
            description = self.client.describe_collection(self.collection_name)

            # Get collection statistics
            stats = self.client.get_collection_stats(self.collection_name)

            # Extract vector field information
            vector_field = None
            dimension = 0
            for field in description.get("fields", []):
                if field.get("type") == "FloatVector":
                    vector_field = field.get("name", "vector")
                    dimension = field.get("params", {}).get("dim", 0)
                    break

            return {
                "name": self.collection_name,
                "document_count": stats.get("row_count", 0),
                "dimension": dimension,
                "vector_store_type": "milvus",
                "vector_field": vector_field,
                "schema": description,
            }

        except Exception as e:
            # Fallback info if detailed info fails
            return {
                "name": self.collection_name,
                "document_count": 0,
                "dimension": 0,
                "vector_store_type": "milvus",
                "error": str(e),
```

```
            }

    def supports_hybrid_search(self) -> bool:
        """Milvus supports hybrid search through dense + sparse vectors."""
        return True

    def hybrid_search(
        self,
        query: str,
        k: int = 5,
        alpha: float = 0.5,
        filters: dict[str, Any] | None = None,
    ) -> list[dict[str, Any]]:
        """
        Hybrid search combining vector similarity and other search methods.

        Note: This implementation focuses on vector search with filtering.
        Full hybrid search with sparse vectors would require additional setup.
        """
        # For now, implement as filtered vector search
        # Future enhancement could include sparse vector search for text matching
        return self.similarity_search(query, k, filters)

    def _format_results(self, results) -> list[dict[str, Any]]:
        """Convert Milvus results to standardized format."""
        documents = []

        # Milvus returns nested list: results[0] contains hits for first query vector
        if not results or not results[0]:
            return documents

        hits = results[0]  # Get hits for our single query vector

        for hit in hits:
            # Extract fields from hit
            hit_id = hit.get("id", "")
            distance = hit.get("distance", 0.0)

            # Extract content - try different field names
            content = ""
            content_fields = ["content", "text", "document", "body", "description"]
            for field in content_fields:
                if field in hit:
                    content = hit[field]
                    break

            # If no content field found, combine text fields
            if not content:
                text_properties = []
                for key, value in hit.items():
                    if isinstance(value, str) and value.strip() and key not in ["id",
"distance"]:
                        text_properties.append(f"{key}: {value}")
                content = " | ".join(text_properties)

            # Create metadata (all fields except content and system fields)
            metadata = {}
            system_fields = ["id", "distance", "vector"] + content_fields
            for key, value in hit.items():
                if key not in system_fields:
                    metadata[key] = value
            metadata["doc_id"] = str(hit_id)

            # Convert distance to similarity score (Milvus returns distance, lower is
better)
            # For cosine distance: similarity = 1 - distance
            # For L2 distance: similarity = 1 / (1 + distance)
            score = max(0.0, 1.0 - distance) if distance <= 1.0 else 1.0 / (1.0 +
```

```
distance)

            documents.append(
                {
                    "content": content,
                    "metadata": metadata,
                    "score": score,
                }
            )

        return documents

    def _convert_filters(self, filters: dict[str, Any]) -> str:
        """Convert generic filters to Milvus expression format."""
        if not filters:
            return None

        expressions = []

        for key, value in filters.items():
            if isinstance(value, str):
                # String exact match
                expressions.append(f'{key} == "{value}"')
            elif isinstance(value, int | float):
                # Numeric exact match
                expressions.append(f"{key} == {value}")
            elif isinstance(value, list):
                # IN clause for multiple values
                if all(isinstance(v, str) for v in value):
                    values_str = '", "'.join(value)
                    expressions.append(f'{key} in ["{values_str}"]')
                else:
                    values_str = ", ".join(str(v) for v in value)
                    expressions.append(f"{key} in [{values_str}]")
            elif isinstance(value, dict):
                # Range queries
                range_conditions = []
                if "gte" in value:
                    range_conditions.append(f"{key} >= {value['gte']}")
                if "gt" in value:
                    range_conditions.append(f"{key} > {value['gt']}")
                if "lte" in value:
                    range_conditions.append(f"{key} <= {value['lte']}")
                if "lt" in value:
                    range_conditions.append(f"{key} < {value['lt']}")

                if range_conditions:
                    expressions.append(" and ".join(range_conditions))

        if expressions:
            return " and ".join(expressions)

        return None

    @classmethod
    def create_local(
        cls, collection_name: str, embedding_function=None, vector_size: int = 384, uri:
str = "./milvus_demo.db"
    ):
        """Create a local Milvus vector store (Milvus Lite)."""
        try:
            from pymilvus import DataType, MilvusClient
        except ImportError as e:
            raise ImportError("Milvus client is required. Install with: pip install
litellm pymilvus") from e

        client = MilvusClient(uri=uri)
```

```
        # Create collection if it doesn't exist
        if not client.has_collection(collection_name):
            # Create collection with explicit schema to avoid max_length issues
            schema = client.create_schema(auto_id=False, enable_dynamic_field=True)

            # Add ID field (string)
            schema.add_field(
                field_name="id",
                datatype=DataType.VARCHAR,
                is_primary=True,
                max_length=512,  # Maximum length for ID strings
            )

            # Add vector field
            schema.add_field(field_name="vector", datatype=DataType.FLOAT_VECTOR,
dim=vector_size)

            # Add content field
            schema.add_field(
                field_name="content",
                datatype=DataType.VARCHAR,
                max_length=65535,  # Large max length for content
            )

            # Create collection with schema
            client.create_collection(
                collection_name=collection_name,
                schema=schema,
            )

            # Create index for vector field
            index_params = client.prepare_index_params()
            index_params.add_index(field_name="vector", metric_type="COSINE")
            client.create_index(collection_name=collection_name,
index_params=index_params)

        return cls(client, collection_name, embedding_function)

    @classmethod
    def create_remote(
        cls,
        collection_name: str,
        embedding_function=None,
        uri: str = "http://localhost:19530",
        user: str = "",
        password: str = "",
        token: str = "",
        vector_size: int = 384,
    ):
        """Create a remote Milvus vector store."""
        try:
            from pymilvus import DataType, MilvusClient
        except ImportError as e:
            raise ImportError("Milvus client is required. Install with: pip install
litellm pymilvus") from e

        # Connect to remote Milvus
        client = MilvusClient(
            uri=uri,
            user=user if user else None,
            password=password if password else None,
            token=token if token else None,
        )

        # Create collection if it doesn't exist
        if not client.has_collection(collection_name):
            # Create collection with explicit schema to avoid max_length issues
            schema = client.create_schema(auto_id=False, enable_dynamic_field=True)
```

```
            # Add ID field (string)
            schema.add_field(
                field_name="id",
                datatype=DataType.VARCHAR,
                is_primary=True,
                max_length=512,  # Maximum length for ID strings
            )

            # Add vector field
            schema.add_field(field_name="vector", datatype=DataType.FLOAT_VECTOR,
dim=vector_size)

            # Add content field
            schema.add_field(
                field_name="content",
                datatype=DataType.VARCHAR,
                max_length=65535,  # Large max length for content
            )

            # Create collection with schema
            client.create_collection(
                collection_name=collection_name,
                schema=schema,
            )

            # Create index for vector field
            index_params = client.prepare_index_params()
            index_params.add_index(field_name="vector", metric_type="COSINE")
            client.create_index(collection_name=collection_name,
index_params=index_params)

        return cls(client, collection_name, embedding_function)
```

**src/gepa/adapters/generic_rag_adapter/vector_stores/qdrant_store.py**

```
# Copyright (c) 2025 Lakshya A Agrawal and the GEPA contributors
# https://github.com/gepa-ai/gepa

from typing import Any

from gepa.adapters.generic_rag_adapter.vector_store_interface import VectorStoreInterface


class QdrantVectorStore(VectorStoreInterface):
    """
    Qdrant implementation of the VectorStoreInterface.

    Qdrant is an open-source vector database with excellent filtering capabilities
    and support for both REST and gRPC APIs. It excels at handling complex metadata
    filtering alongside vector similarity search.
    """

    def __init__(self, client, collection_name: str, embedding_function=None):
        """
        Initialize QdrantVectorStore.

        Args:
            client: QdrantClient instance
            collection_name: Name of the collection to use
            embedding_function: Optional function to compute embeddings for queries
        """
        import importlib.util
```

```
        if importlib.util.find_spec("qdrant_client") is None:
            raise ImportError(
                "Qdrant client is required for QdrantVectorStore. Install with: pip
install litellm qdrant-client"
            )

        from qdrant_client.http import models

        self.client = client
        self.collection_name = collection_name
        self.embedding_function = embedding_function
        self.models = models

        # Verify collection exists
        try:
            self.client.get_collection(collection_name)
        except Exception as e:
            raise ValueError(
                f"Collection '{collection_name}' not found. Please create the collection
first. Error: {e!s}"
            ) from e

    def similarity_search(
        self,
        query: str,
        k: int = 5,
        filters: dict[str, Any] | None = None,
    ) -> list[dict[str, Any]]:
        """Search for documents similar to the query text using embeddings."""
        if self.embedding_function is None:
            raise ValueError("No embedding function provided for similarity search")

        # Compute embeddings for the query
        try:
            query_vector = self.embedding_function(query)
            if hasattr(query_vector, "tolist"):
                query_vector = query_vector.tolist()
        except Exception as e:
            raise RuntimeError(f"Failed to compute embeddings for query: {e!s}") from e

        # Use vector search with computed embeddings
        return self.vector_search(query_vector, k, filters)

    def vector_search(
        self,
        query_vector: list[float],
        k: int = 5,
        filters: dict[str, Any] | None = None,
    ) -> list[dict[str, Any]]:
        """Search using a pre-computed query vector."""
        qdrant_filter = self._convert_filters(filters) if filters else None

        try:
            # Use modern query_points API
            results = self.client.query_points(
                collection_name=self.collection_name,
                query=query_vector,
                query_filter=qdrant_filter,
                limit=k,
                with_payload=True,
                with_vectors=False,
                score_threshold=None,
            )

            return self._format_results(results)

        except Exception as e:
            raise RuntimeError(f"Qdrant vector search failed: {e!s}") from e
```

```
    def add_documents(
        self,
        documents: list[dict[str, Any]],
        embeddings: list[list[float]],
        ids: list[str] | None = None,
    ) -> list[str]:
        """Add documents with their embeddings to the collection."""
        if len(documents) != len(embeddings):
            raise ValueError("Number of documents must match number of embeddings")

        # Generate IDs if not provided - use integers for Qdrant compatibility
        if ids is None:
            ids = list(range(len(documents)))
            string_ids = [f"doc_{i}" for i in range(len(documents))]
        else:
            if len(ids) != len(documents):
                raise ValueError("Number of IDs must match number of documents")
            # Convert string IDs to integers for Qdrant, but keep original strings in
payload
            string_ids = ids
            ids = list(range(len(documents)))

        # Create Qdrant points
        points = []
        for i, (doc, embedding) in enumerate(zip(documents, embeddings, strict=False)):
            # Add the original string ID to the payload for retrieval
            payload = dict(doc)
            payload["original_id"] = string_ids[i]

            point = self.models.PointStruct(
                id=ids[i],  # Use integer ID for Qdrant
                vector=embedding,
                payload=payload,
            )
            points.append(point)

        try:
            # Upsert points to collection
            result = self.client.upsert(
                collection_name=self.collection_name,
                points=points,
                wait=True,  # Wait for operation to complete
            )

            if result.status.name != "COMPLETED":
                raise RuntimeError(f"Upsert operation failed: {result.status}")

            return string_ids  # Return original string IDs

        except Exception as e:
            raise RuntimeError(f"Failed to add documents to Qdrant: {e!s}") from e

    def delete_documents(self, ids: list[str]) -> bool:
        """Delete documents by their IDs."""
        try:
            # Create filter to match documents by original_id
            delete_filter = self.models.Filter(
                must=[self.models.FieldCondition(key="original_id",
match=self.models.MatchAny(any=ids))]
            )

            result = self.client.delete(
                collection_name=self.collection_name,
                points_selector=self.models.FilterSelector(filter=delete_filter),
                wait=True,
            )
```

```
            return result.status.name == "COMPLETED"

        except Exception as e:
            raise RuntimeError(f"Failed to delete documents from Qdrant: {e!s}") from e

    def get_collection_info(self) -> dict[str, Any]:
        """Get metadata about the Qdrant collection."""
        try:
            # Get collection info
            info = self.client.get_collection(self.collection_name)

            # Get collection statistics
            points_count = info.points_count or 0
            vectors_config = info.config.params.vectors

            # Handle both named and unnamed vector configs
            if isinstance(vectors_config, dict):
                # Named vectors
                vector_info = next(iter(vectors_config.values())) if vectors_config else
None
                dimension = vector_info.size if vector_info else 0
                distance = vector_info.distance.name if vector_info else "UNKNOWN"
            else:
                # Single vector config
                dimension = vectors_config.size if vectors_config else 0
                distance = vectors_config.distance.name if vectors_config else "UNKNOWN"

            return {
                "name": self.collection_name,
                "document_count": points_count,
                "dimension": dimension,
                "vector_store_type": "qdrant",
                "distance_metric": distance.lower(),
                "status": info.status.name,
            }

        except Exception as e:
            # Fallback info if detailed info fails
            return {
                "name": self.collection_name,
                "document_count": 0,
                "dimension": 0,
                "vector_store_type": "qdrant",
                "error": str(e),
            }

    def supports_hybrid_search(self) -> bool:
        """Qdrant supports hybrid search through payload filtering."""
        return True

    def hybrid_search(
        self,
        query: str,
        k: int = 5,
        alpha: float = 0.5,
        filters: dict[str, Any] | None = None,
    ) -> list[dict[str, Any]]:
        """
        Hybrid search combining vector similarity and keyword matching.

        Note: Qdrant doesn't have built-in hybrid search like some other databases,
        but we can combine vector search with payload filtering for similar functionality.
        """
        # For now, implement as filtered vector search
        # In the future, this could be enhanced with text search capabilities
        return self.similarity_search(query, k, filters)

    def _format_results(self, results) -> list[dict[str, Any]]:
```

```
        """Convert Qdrant results to standardized format."""
        documents = []

        # Handle both direct points list and QueryResponse
        points = results.points if hasattr(results, "points") else results

        if not points:
            return documents

        for point in points:
            # Extract content from payload
            payload = point.payload or {}

            # Get content - try different field names
            content = ""
            content_fields = ["content", "text", "document", "body", "description"]
            for field in content_fields:
                if field in payload:
                    content = payload[field]
                    break

            # If no content field found, use all text properties
            if not content:
                text_properties = []
                for key, value in payload.items():
                    if isinstance(value, str) and value.strip():
                        text_properties.append(f"{key}: {value}")
                content = " | ".join(text_properties)

            # Create metadata (all payload except content field)
            metadata = {k: v for k, v in payload.items() if k not in content_fields and k
!= "original_id"}
            metadata["doc_id"] = payload.get("original_id", str(point.id))

            # Convert score (Qdrant returns similarity score, higher is better)
            score = float(point.score) if hasattr(point, "score") and point.score is not
None else 0.0

            documents.append(
                {
                    "content": content,
                    "metadata": metadata,
                    "score": score,
                }
            )

        return documents

    def _convert_filters(self, filters: dict[str, Any]) -> Any:
        """Convert generic filters to Qdrant filter format."""
        if not filters:
            return None

        must_conditions = []

        for key, value in filters.items():
            if isinstance(value, str):
                # Exact string match
                condition = self.models.FieldCondition(key=key,
match=self.models.MatchValue(value=value))
            elif isinstance(value, int | float):
                # Exact numeric match
                condition = self.models.FieldCondition(key=key,
match=self.models.MatchValue(value=value))
            elif isinstance(value, list):
                # Match any of the values
                condition = self.models.FieldCondition(key=key,
match=self.models.MatchAny(any=value))
```

```
            elif isinstance(value, dict):
                # Range queries or complex conditions
                if "gte" in value or "gt" in value or "lte" in value or "lt" in value:
                    condition = self.models.FieldCondition(
                        key=key,
                        range=self.models.Range(
                            gte=value.get("gte"), gt=value.get("gt"),
lte=value.get("lte"), lt=value.get("lt")
                        ),
                    )
                else:
                    # Skip unsupported filter format
                    continue
            else:
                # Skip unsupported filter type
                continue

            must_conditions.append(condition)

        if must_conditions:
            return self.models.Filter(must=must_conditions)

        return None

    @classmethod
    def create_local(
        cls, collection_name: str, embedding_function=None, path: str = ":memory:",
vector_size: int = 384
    ):
        """Create a local Qdrant vector store (useful for testing)."""
        try:
            from qdrant_client import QdrantClient
            from qdrant_client.http import models
        except ImportError as e:
            raise ImportError("Qdrant client is required. Install with: pip install
litellm qdrant-client") from e

        client = QdrantClient(path=path)

        # Create collection if it doesn't exist
        try:
            client.get_collection(collection_name)
        except Exception:
            # Collection doesn't exist, create it
            client.create_collection(
                collection_name=collection_name,
                vectors_config=models.VectorParams(size=vector_size,
distance=models.Distance.COSINE),
            )

        return cls(client, collection_name, embedding_function)

    @classmethod
    def create_remote(
        cls,
        collection_name: str,
        embedding_function=None,
        host: str = "localhost",
        port: int = 6333,
        api_key: str | None = None,
        vector_size: int = 384,
    ):
        """Create a remote Qdrant vector store."""
        try:
            from qdrant_client import QdrantClient
            from qdrant_client.http import models
        except ImportError as e:
            raise ImportError("Qdrant client is required. Install with: pip install
```

```
litellm qdrant-client") from e

        client = QdrantClient(host=host, port=port, api_key=api_key)

        # Create collection if it doesn't exist
        try:
            client.get_collection(collection_name)
        except Exception:
            # Collection doesn't exist, create it
            client.create_collection(
                collection_name=collection_name,
                vectors_config=models.VectorParams(size=vector_size,
distance=models.Distance.COSINE),
            )

        return cls(client, collection_name, embedding_function)
```

### src/gepa/adapters/generic_rag_adapter/vector_stores/weaviate_store.py

```
# Copyright (c) 2025 Lakshya A Agrawal and the GEPA contributors
# https://github.com/gepa-ai/gepa

from typing import Any

from gepa.adapters.generic_rag_adapter.vector_store_interface import VectorStoreInterface


class WeaviateVectorStore(VectorStoreInterface):
    """
    Weaviate implementation of the VectorStoreInterface.

    Weaviate is a cloud-native, modular, real-time vector database
    with powerful search capabilities including hybrid search.
    Supports both Weaviate Cloud Services and self-hosted deployments.
    """

    def __init__(self, client, collection_name: str, embedding_function=None):
        """
        Initialize WeaviateVectorStore.

        Args:
            client: Weaviate client instance (WeaviateClient from weaviate-client)
            collection_name: Name of the collection/class to use
            embedding_function: Optional function to compute embeddings for queries
        """
        import importlib.util

        if importlib.util.find_spec("weaviate") is None:
            raise ImportError(
                "Weaviate client is required for WeaviateVectorStore. Install with: pip
install litellm weaviate-client"
            )

        import weaviate.classes as wvc

        self.client = client
        self.collection_name = collection_name
        self.wvc = wvc
        self.embedding_function = embedding_function

        # Get the collection
        try:
            self.collection = self.client.collections.get(collection_name)
        except Exception as e:
```

```
            raise ValueError(
                f"Collection '{collection_name}' not found. Please create the collection
first. Error: {e!s}"
            ) from e

    def similarity_search(
        self,
        query: str,
        k: int = 5,
        filters: dict[str, Any] | None = None,
    ) -> list[dict[str, Any]]:
        """Search for documents similar to the query text using embeddings."""
        if self.embedding_function is None:
            raise ValueError("No embedding function provided for similarity search")

        # Compute embeddings for the query
        try:
            query_vector = self.embedding_function(query)
            if hasattr(query_vector, "tolist"):
                query_vector = query_vector.tolist()
        except Exception as e:
            raise RuntimeError(f"Failed to compute embeddings for query: {e!s}") from e

        # Use vector search with computed embeddings
        return self.vector_search(query_vector, k, filters)

    def vector_search(
        self,
        query_vector: list[float],
        k: int = 5,
        filters: dict[str, Any] | None = None,
    ) -> list[dict[str, Any]]:
        """Search using a pre-computed query vector."""
        weaviate_filters = self._convert_filters(filters) if filters else None

        try:
            # Execute query
            if weaviate_filters:
                response = self.collection.query.near_vector(near_vector=query_vector,
limit=k).where(weaviate_filters)
            else:
                response = self.collection.query.near_vector(near_vector=query_vector,
limit=k)

            # Handle GenerativeReturn object - access .objects attribute
            if hasattr(response, "objects"):
                results = response.objects
            else:
                results = response

            return self._format_results(results)

        except Exception as e:
            raise RuntimeError(f"Weaviate vector search failed: {e!s}") from e

    def hybrid_search(
        self,
        query: str,
        k: int = 5,
        alpha: float = 0.5,
        filters: dict[str, Any] | None = None,
    ) -> list[dict[str, Any]]:
        """
        Hybrid semantic + keyword search using Weaviate's native hybrid search.

        Args:
            query: Text query to search for
            k: Number of documents to return
```

```
            alpha: Weight for semantic vs keyword search (0.0 = pure keyword, 1.0 = pure
semantic)
            filters: Optional metadata filters
        """
        weaviate_filters = self._convert_filters(filters) if filters else None

        try:
            results = self.collection.query.hybrid(
                query=query,
                alpha=alpha,
                limit=k,
                where=weaviate_filters,
                return_metadata=self.wvc.query.MetadataQuery(score=True,
explain_score=True),
                return_properties=["content", "*"],
            )

            return self._format_results(results)

        except Exception as e:
            raise RuntimeError(f"Weaviate hybrid search failed: {e!s}") from e

    def get_collection_info(self) -> dict[str, Any]:
        """Get metadata about the Weaviate collection."""
        try:
            # Get collection configuration
            config = self.collection.config.get()

            # Count objects in collection
            count_result = self.collection.aggregate.over_all(total_count=True)
            total_count = count_result.total_count

            # Try to determine vector dimensions
            dimension = 0
            if hasattr(config, "vector_config") and config.vector_config:
                # For collections with vector config
                if hasattr(config.vector_config, "vector_index_config"):
                    vector_index = config.vector_index_config
                    if hasattr(vector_index, "distance"):
                        # This indicates vectors are configured
                        # Try to get dimension from a sample object
                        try:
                            sample = self.collection.query.fetch_objects(limit=1,
include_vector=True)
                            if sample.objects and hasattr(sample.objects[0], "vector") and
sample.objects[0].vector:
                                # Handle both named vectors and default vector
                                vector_data = sample.objects[0].vector
                                if isinstance(vector_data, dict):
                                    # Named vectors case
                                    for _vector_name, vector_values in
vector_data.items():
                                        if vector_values:
                                            dimension = len(vector_values)
                                            break
                                elif isinstance(vector_data, list):
                                    # Default vector case
                                    dimension = len(vector_data)
                        except Exception:
                            # If we can't determine dimension from sample, that's ok
                            pass

            # Determine if hybrid search is supported
            supports_hybrid = True  # Weaviate generally supports hybrid search
            try:
                # Test if hybrid search works by doing a minimal query
                self.collection.query.hybrid(query="test", limit=1)
                supports_hybrid = True
```

```
            except Exception:
                supports_hybrid = False

            return {
                "name": self.collection_name,
                "document_count": total_count,
                "dimension": dimension,
                "vector_store_type": "weaviate",
                "supports_hybrid_search": supports_hybrid,
                "vectorizer": getattr(config, "vectorizer_config", None),
                "properties": [prop.name for prop in getattr(config, "properties", [])],
            }

        except Exception as e:
            # Fallback info if detailed info fails
            return {
                "name": self.collection_name,
                "document_count": 0,
                "dimension": 0,
                "vector_store_type": "weaviate",
                "error": str(e),
            }

    def supports_hybrid_search(self) -> bool:
        """Weaviate supports hybrid search."""
        try:
            # Test if hybrid search works
            self.collection.query.hybrid(query="test", limit=1)
            return True
        except Exception:
            return False

    def supports_metadata_filtering(self) -> bool:
        """Weaviate supports metadata filtering with where clauses."""
        return True

    def _convert_filters(self, filters: dict[str, Any]) -> Any:
        """
        Convert generic filters to Weaviate where clause format.

        Generic format: {"key": "value", "key2": {"$gt": 5}}
        Weaviate format uses Filter class
        """
        if not filters:
            return None

        try:
            from weaviate.collections.classes.filters import Filter
        except ImportError:
            # Fallback if Filter class not available
            return None

        filter_conditions = []

        for key, value in filters.items():
            if isinstance(value, dict):
                # Handle operator-based filters
                for op, op_value in value.items():
                    if op == "$eq" or op == "equal":
                        filter_conditions.append(Filter.by_property(key).equal(op_value))
                    elif op == "$ne" or op == "not_equal":

filter_conditions.append(Filter.by_property(key).not_equal(op_value))
                    elif op == "$gt" or op == "greater_than":

filter_conditions.append(Filter.by_property(key).greater_than(op_value))
                    elif op == "$gte" or op == "greater_equal":
```

```
filter_conditions.append(Filter.by_property(key).greater_or_equal(op_value))
                    elif op == "$lt" or op == "less_than":

filter_conditions.append(Filter.by_property(key).less_than(op_value))
                    elif op == "$lte" or op == "less_equal":

filter_conditions.append(Filter.by_property(key).less_or_equal(op_value))
                    elif op == "$in" or op == "contains_any":
                        if isinstance(op_value, list):

filter_conditions.append(Filter.by_property(key).contains_any(op_value))
                    elif op == "$like" or op == "like":
                        filter_conditions.append(Filter.by_property(key).like(op_value))
            else:
                # Simple equality filter
                filter_conditions.append(Filter.by_property(key).equal(value))

        # Combine all conditions with AND
        if len(filter_conditions) == 1:
            return filter_conditions[0]
        elif len(filter_conditions) > 1:
            combined_filter = filter_conditions[0]
            for condition in filter_conditions[1:]:
                combined_filter = combined_filter & condition
            return combined_filter

        return None

    def _format_results(self, results) -> list[dict[str, Any]]:
        """Convert Weaviate results to standardized format."""
        documents = []

        # Handle both GenerativeReturn objects and direct lists
        if hasattr(results, "objects"):
            objects_list = results.objects
        else:
            objects_list = results

        if not objects_list:
            return documents

        for obj in objects_list:
            # Get the content - try different property names
            content = ""
            properties = obj.properties

            # Common content field names
            content_fields = ["content", "text", "document", "body", "description"]
            for field in content_fields:
                if field in properties:
                    content = properties[field]
                    break

            # If no content field found, use all text properties
            if not content:
                text_properties = []
                for key, value in properties.items():
                    if isinstance(value, str) and value.strip():
                        text_properties.append(f"{key}: {value}")
                content = " | ".join(text_properties)

            # Extract metadata (all properties except content)
            metadata = {}
            for key, value in properties.items():
                if key not in ["content", "text", "document", "body"] or not content:
                    metadata[key] = value

            # Add UUID as doc_id in metadata
```

```
            metadata["doc_id"] = str(obj.uuid)

            # Calculate similarity score from distance or use provided score
            score = 0.0
            if hasattr(obj.metadata, "score") and obj.metadata.score is not None:
                score = float(obj.metadata.score)
            elif hasattr(obj.metadata, "distance") and obj.metadata.distance is not None:
                # Convert distance to similarity (assuming cosine distance)
                # Weaviate cosine distance is between 0 and 2, similarity = 1 -
(distance/2)
                distance = float(obj.metadata.distance)
                score = max(0.0, 1.0 - (distance / 2.0))

            documents.append({"content": content, "metadata": metadata, "score": score})

        return documents

    @classmethod
    def create_local(
        cls,
        host: str = "localhost",
        port: int = 8080,
        grpc_port: int = 50051,
        collection_name: str = "Documents",
        headers: dict[str, str] | None = None,
    ) -> "WeaviateVectorStore":
        """
        Create a WeaviateVectorStore connected to local Weaviate instance.

        Args:
            host: Weaviate host
            port: HTTP port
            grpc_port: gRPC port
            collection_name: Name of the collection
            headers: Optional headers for authentication

        Returns:
            WeaviateVectorStore instance
        """
        try:
            import weaviate
        except ImportError as e:
            raise ImportError("Weaviate client is required. Install with: pip install
litellm weaviate-client") from e

        client = weaviate.connect_to_local(host=host, port=port, grpc_port=grpc_port,
headers=headers)

        return cls(client, collection_name)

    @classmethod
    def create_cloud(
        cls,
        cluster_url: str,
        auth_credentials,
        collection_name: str = "Documents",
        headers: dict[str, str] | None = None,
    ) -> "WeaviateVectorStore":
        """
        Create a WeaviateVectorStore connected to Weaviate Cloud Services.

        Args:
            cluster_url: Weaviate cluster URL
            auth_credentials: Authentication credentials (API key or other auth)
            collection_name: Name of the collection
            headers: Optional headers

        Returns:
```

```
            WeaviateVectorStore instance
        """
        try:
            import weaviate
        except ImportError as e:
            raise ImportError("Weaviate client is required. Install with: pip install
litellm weaviate-client") from e

        client = weaviate.connect_to_weaviate_cloud(
            cluster_url=cluster_url, auth_credentials=auth_credentials, headers=headers
        )

        return cls(client, collection_name)

    @classmethod
    def create_custom(
        cls,
        url: str,
        auth_credentials=None,
        collection_name: str = "Documents",
        headers: dict[str, str] | None = None,
        grpc_port: int | None = None,
    ) -> "WeaviateVectorStore":
        """
        Create a WeaviateVectorStore with custom connection parameters.

        Args:
            url: Weaviate instance URL
            auth_credentials: Authentication credentials
            collection_name: Name of the collection
            headers: Optional headers
            grpc_port: Optional gRPC port

        Returns:
            WeaviateVectorStore instance
        """
        try:
            import weaviate
        except ImportError as e:
            raise ImportError("Weaviate client is required. Install with: pip install
litellm weaviate-client") from e

        client = weaviate.connect_to_custom(
            http_host=url,
            http_port=443 if "https" in url else 80,
            http_secure="https" in url,
            grpc_port=grpc_port,
            grpc_secure="https" in url,
            auth_credentials=auth_credentials,
            headers=headers,
        )

        return cls(client, collection_name)
```

**src/gepa/adapters/terminal_bench_adapter/README.md**

```
### Terminal-bench adapter

This adapter is used to optimize the system prompt/terminal-use instruction for the
default Terminus agent through custom a `GEPAAdapter` implementation.

To run this example, you need to install `pip install terminal-bench` and run the
following command:
```

```bash
python src/gepa/examples/terminal-bench/train_terminus.py --model_name=gpt-5-mini
```

### src/gepa/adapters/terminal_bench_adapter/__init__.py

### src/gepa/adapters/terminal_bench_adapter/terminal_bench_adapter.py

```
import json
import os
import subprocess
from datetime import datetime
from pathlib import Path

from pydantic import BaseModel
from terminal_bench.agents.terminus_1 import CommandBatchResponse

from gepa import EvaluationBatch, GEPAAdapter


class TerminalBenchTask(BaseModel):
    task_id: str
    model_name: str


def run_agent_tb(
    task_ids: str | list[str],
    run_id: str,
    model_name: str,
    instruction_prompt: str,
    dataset_name: str = "terminal-bench-core",
    dataset_version: str = "head",
    agent_import_path: str = "train_terminus:TerminusWrapper",
    n_concurrent: int = 6,
    prompt_template_path: str = "prompt-templates/instruction_prompt.txt",
):
    """Run the replay agent for multiple task IDs using tb run command."""

    env = os.environ.copy()
    # write instruction prompt to file
    with open(prompt_template_path, "w") as f:
        f.write(instruction_prompt)

    cmd = [
        "tb",
        "run",
        "--dataset-name",
        dataset_name,
        "--dataset-version",
        dataset_version,
        "--agent-import-path",
        agent_import_path,
        "--model-name",
        model_name,
        "--run-id",
        run_id,
        "--n-concurrent",
        str(n_concurrent),
```

```
        "--output-path",
        str(Path(os.getcwd()) / "runs"),
    ]
    if isinstance(task_ids, list):
        for task_id in task_ids:
            cmd.extend(["--task-id", task_id])
    else:
        cmd.extend(["--task-id", task_ids])

    print(f"Running command: {' '.join(cmd)}")

    try:
        result = subprocess.run(cmd, env=env,
cwd=Path(prompt_template_path).parent.parent, check=True)
        print(f"Command completed successfully with return code: {result.returncode}")
        return result.returncode
    except subprocess.CalledProcessError as e:
        print(f"Command failed with return code: {e.returncode}")
        return e.returncode
    except Exception as e:
        print(f"Error running command: {e}")
        return 1


def get_results(task_id: str, run_id: str) -> tuple[int, list]:
    def _read_episode_response(episode_dir: Path) -> CommandBatchResponse | None:
        """Helper method to read and parse response.json from an episode directory."""
        response_file = episode_dir / "response.json"
        if response_file.exists():
            try:
                response_content = response_file.read_text()
                return CommandBatchResponse.model_validate_json(response_content)
            except Exception:
                pass
        return None

    def _get_logging_dir(task_id: str, run_id: str):
        logging_dir_base = Path("runs") / run_id / task_id
        for dir in logging_dir_base.iterdir():
            if dir.is_dir() and dir.name.startswith(task_id):
                return dir
        raise ValueError(f"No logging directory found for task {task_id} and run
{run_id}")

    logging_dir = _get_logging_dir(task_id, run_id)
    result_json = logging_dir / "results.json"
    with open(result_json) as f:
        result = json.load(f)
    if result.get("parser_results", None):
        score = sum(x == "passed" for x in result["parser_results"].values())
    else:
        score = 0

    if result.get("is_resolved", None):
        success = True
    else:
        success = False

    failed_reason = result.get("failure_mode", "unknown")

    trajectory_path = logging_dir / "agent-logs"
    episode_dirs = []
    for dir in trajectory_path.iterdir():
        if dir.is_dir() and dir.name.startswith("episode-"):
            episode_dirs.append(dir)

    if episode_dirs:
        # Sort by episode number to get the last one
```

```
        episode_dirs.sort(key=lambda x: int(x.name.split("-")[1]))
        last_episode_dir = episode_dirs[-1]

    last_episode_dir_trajectory = last_episode_dir / "debug.json"
    with open(last_episode_dir_trajectory) as f:
        trajectory = json.load(f)

        if "input" in trajectory and isinstance(trajectory["input"], list):
            messages = trajectory["input"]

        # Add the last assistant response using helper method
        parsed_response = _read_episode_response(last_episode_dir)

        if parsed_response:
            assistant_message = {
                "role": "assistant",
                "content": parsed_response.model_dump_json(),
            }
            messages.append(assistant_message)

    return success, score, failed_reason, messages


class TerminusAdapter(GEPAAdapter):
    def __init__(
        self,
        n_concurrent: int = 6,
        instruction_prompt_path: str = "prompt-templates/instruction_prompt.txt",
    ):
        self.n_concurrent = n_concurrent
        self.instruction_prompt_path = instruction_prompt_path

    def evaluate(
        self,
        batch: list[TerminalBenchTask],
        candidate: dict[str, str],
        capture_traces: bool = False,
    ) -> EvaluationBatch:
        outputs = []
        scores = []
        trajectories = []
        example_run_id = "temp_gepa_run" + "_" + datetime.now().strftime("%Y%m%d%H%M%S")
        example_model_name = batch[0].model_name

        run_agent_tb(
            [task.task_id for task in batch],
            example_run_id,
            example_model_name,
            instruction_prompt=candidate["instruction_prompt"],
            n_concurrent=self.n_concurrent,
            prompt_template_path=self.instruction_prompt_path,
        )

        for example in batch:
            try:
                success, score, failed_reason, messages = get_results(example.task_id,
example_run_id)
            except Exception as e:
                print(f"Error running example {example.task_id} {example_run_id}: {e}")
                success = False
                score = 0
                failed_reason = str(e)
                messages = []

            outputs.append(
                f"Terminal Bench outputs are omitted. Please see
runs/{example_run_id}/{example.task_id}/ for detailed logging."
            )
```

```
            scores.append(score)
            trajectories.append(
                {
                    "messages": messages,
                    "instruction_prompt": candidate["instruction_prompt"],
                    "failed_reason": failed_reason,
                    "success": success,
                }
            )
        return EvaluationBatch(
            outputs=outputs,
            scores=scores,
            trajectories=trajectories,
        )

    def make_reflective_dataset(
        self,
        candidate: dict[str, str],
        eval_batch: EvaluationBatch,
        components_to_update: list[str],
    ):
        reflective_dataset = {"instruction_prompt": []}
        for _score, trajectory in zip(eval_batch.scores, eval_batch.trajectories,
strict=False):
            if trajectory["success"]:
                feedback = "Successfully solved the task!"
            else:
                feedback = f"Failed to solve the task. Reason:
{trajectory['failed_reason']}"
            reflective_dataset["instruction_prompt"].append(
                {
                    "Message History": trajectory["messages"],
                    "Instruction Prompt": candidate["instruction_prompt"],
                    "Feedback": feedback,
                }
            )
        return reflective_dataset
```

### src/gepa/api.py

```
# Copyright (c) 2025 Lakshya A Agrawal and the GEPA contributors
# https://github.com/gepa-ai/gepa

import os
import random
from collections.abc import Sequence
from typing import Any, Literal, cast

from gepa.adapters.default_adapter.default_adapter import ChatCompletionCallable,
DefaultAdapter
from gepa.core.adapter import DataInst, GEPAAdapter, RolloutOutput, Trajectory
from gepa.core.data_loader import DataId, DataLoader, ensure_loader
from gepa.core.engine import GEPAEngine
from gepa.core.result import GEPAResult
from gepa.logging.experiment_tracker import create_experiment_tracker
from gepa.logging.logger import LoggerProtocol, StdOutLogger
from gepa.proposer.merge import MergeProposer
from gepa.proposer.reflective_mutation.base import CandidateSelector, LanguageModel,
ReflectionComponentSelector
from gepa.proposer.reflective_mutation.reflective_mutation import
ReflectiveMutationProposer
from gepa.strategies.batch_sampler import BatchSampler, EpochShuffledBatchSampler
from gepa.strategies.candidate_selector import (
    CurrentBestCandidateSelector,
```

```
    EpsilonGreedyCandidateSelector,
    ParetoCandidateSelector,
)
from gepa.strategies.component_selector import (
    AllReflectionComponentSelector,
    RoundRobinReflectionComponentSelector,
)
from gepa.strategies.eval_policy import EvaluationPolicy, FullEvaluationPolicy
from gepa.utils import FileStopper, StopperProtocol


def optimize(
    seed_candidate: dict[str, str],
    trainset: list[DataInst] | DataLoader[DataId, DataInst],
    valset: list[DataInst] | DataLoader[DataId, DataInst] | None = None,
    adapter: GEPAAdapter[DataInst, Trajectory, RolloutOutput] | None = None,
    task_lm: str | ChatCompletionCallable | None = None,
    # Reflection-based configuration
    reflection_lm: LanguageModel | str | None = None,
    candidate_selection_strategy: CandidateSelector | Literal["pareto", "current_best",
"epsilon_greedy"] = "pareto",
    skip_perfect_score: bool = True,
    batch_sampler: BatchSampler | Literal["epoch_shuffled"] = "epoch_shuffled",
    reflection_minibatch_size: int | None = None,
    perfect_score: float = 1.0,
    reflection_prompt_template: str | None = None,
    # Component selection configuration
    module_selector: ReflectionComponentSelector | str = "round_robin",
    # Merge-based configuration
    use_merge: bool = False,
    max_merge_invocations: int = 5,
    merge_val_overlap_floor: int = 5,
    # Budget and Stop Condition
    max_metric_calls: int | None = None,
    stop_callbacks: StopperProtocol | Sequence[StopperProtocol] | None = None,
    # Logging
    logger: LoggerProtocol | None = None,
    run_dir: str | None = None,
    use_wandb: bool = False,
    wandb_api_key: str | None = None,
    wandb_init_kwargs: dict[str, Any] | None = None,
    use_mlflow: bool = False,
    mlflow_tracking_uri: str | None = None,
    mlflow_experiment_name: str | None = None,
    track_best_outputs: bool = False,
    display_progress_bar: bool = False,
    use_cloudpickle: bool = False,
    # Reproducibility
    seed: int = 0,
    raise_on_exception: bool = True,
    val_evaluation_policy: EvaluationPolicy[DataId, DataInst] | Literal["full_eval"] |
None = None,
) -> GEPAResult[RolloutOutput, DataId]:
    """
    GEPA is an evolutionary optimizer that evolves (multiple) text components of a complex
system to optimize them towards a given metric.
    GEPA can also leverage rich textual feedback obtained from the system's execution
environment, evaluation,
    and the system's own execution traces to iteratively improve the system's performance.

    Concepts:
    - System: A harness that uses text components to perform a task. Each text component
of the system to be optimized is a named component of the system.
    - Candidate: A mapping from component names to component text. A concrete
instantiation of the system is realized by setting the text of each system component
      to the text provided by the candidate mapping.
    - `DataInst`: An (uninterpreted) data type over which the system operates.
    - `RolloutOutput`: The output of the system on a `DataInst`.
```

```
    Each execution of the system produces a `RolloutOutput`, which can be evaluated to
produce a score. The execution of the system also produces a trajectory,
    which consists of the operations performed by different components of the system,
including the text of the components that were executed.

    GEPA can be applied to optimize any system that uses text components (e.g., prompts in
a AI system, code snippets/code files/functions/classes in a codebase, etc.).
    In order for GEPA to plug into your system's environment, GEPA requires an adapter,
`GEPAAdapter` to be implemented. The adapter is responsible for:
    1. Evaluating a proposed candidate on a batch of inputs.
       - The adapter receives a candidate proposed by GEPA, along with a batch of inputs
selected from the training/validation set.
       - The adapter instantiates the system with the texts proposed in the candidate.
       - The adapter then evaluates the candidate on the batch of inputs, and returns the
scores.
       - The adapter should also capture relevant information from the execution of the
candidate, like system and evaluation traces.
    2. Identifying textual information relevant to a component of the candidate
       - Given the trajectories captured during the execution of the candidate, GEPA
selects a component of the candidate to update.
       - The adapter receives the candidate, the batch of inputs, and the trajectories
captured during the execution of the candidate.
       - The adapter is responsible for identifying the textual information relevant to
the component to update.
       - This information is used by GEPA to reflect on the performnace of the component,
and propose new component texts.

    At each iteration, GEPA proposes a new candidate using one of the following
strategies:
    1. Reflective mutation: GEPA proposes a new candidate by mutating the current
candidate, leveraging rich textual feedback.
    2. Merge: GEPA proposes a new candidate by merging 2 candidates that are on the Pareto
frontier.

    GEPA also tracks the Pareto frontier of performance achieved by different candidates
on the validation set. This way, it can leverage candidates that
    work well on a subset of inputs to improve the system's performance on the entire
validation set, by evolving from the Pareto frontier.

    Parameters:
    - seed_candidate: The initial candidate to start with.
    - trainset: Training data supplied as an in-memory sequence or a `DataLoader` yielding
batches for reflective updates.
    - valset: Validation data source (sequence or `DataLoader`) used for tracking Pareto
scores. If not provided, GEPA reuses the trainset.
    - adapter: A `GEPAAdapter` instance that implements the adapter interface. This allows
GEPA to plug into your system's environment. If not provided, GEPA will use a default
adapter: `gepa.adapters.default_adapter.default_adapter.DefaultAdapter`, with model
defined by `task_lm`.
    - task_lm: Optional. The model to use for the task. This is only used if `adapter` is
not provided, and is used to initialize the default adapter.

    # Reflection-based configuration
    - reflection_lm: A `LanguageModel` instance that is used to reflect on the performance
of the candidate program.
    - candidate_selection_strategy: The strategy to use for selecting the candidate to
update. Supported strategies: 'pareto', 'current_best', 'epsilon_greedy'. Defaults to
'pareto'.
    - skip_perfect_score: Whether to skip updating the candidate if it achieves a perfect
score on the minibatch.
    - batch_sampler: Strategy for selecting training examples. Can be a [BatchSampler]
(src/gepa/strategies/batch_sampler.py) instance or a string for a predefined strategy from
['epoch_shuffled']. Defaults to 'epoch_shuffled', which creates an
[EpochShuffledBatchSampler](src/gepa/strategies/batch_sampler.py).
    - reflection_minibatch_size: The number of examples to use for reflection in each
proposal step. Defaults to 3. Only valid when batch_sampler='epoch_shuffled' (default),
and is ignored otherwise.
```

```
    - perfect_score: The perfect score to achieve.
    - reflection_prompt_template: The prompt template to use for reflection. If not
provided, GEPA will use the default prompt template (see [InstructionProposalSignature]
(src/gepa/strategies/instruction_proposal.py)). The prompt template must contain the
following placeholders, which will be replaced with actual values: `<curr_instructions>`
(will be replaced by the instructions to evolve) and `<inputs_outputs_feedback>` (replaced
with the inputs, outputs, and feedback generated with current instruction). This will be
ignored if the adapter provides its own `propose_new_texts` method.

    # Component selection configuration
    - module_selector: Component selection strategy. Can be a ReflectionComponentSelector
instance or a string ('round_robin', 'all'). Defaults to 'round_robin'. The 'round_robin'
strategy cycles through components in order. The 'all' strategy selects all components for
modification in every GEPA iteration.

    # Merge-based configuration
    - use_merge: Whether to use the merge strategy.
    - max_merge_invocations: The maximum number of merge invocations to perform.
    - merge_val_overlap_floor: Minimum number of shared validation ids required between
parents before attempting a merge subsample. Only relevant when using
`val_evaluation_policy` other than `full_eval`.

    # Budget and Stop Condition
    - max_metric_calls: Optional maximum number of metric calls to perform. If not
provided, stop_callbacks must be provided.
    - stop_callbacks: Optional stopper(s) that return True when optimization should stop.
Can be a single StopperProtocol or a list or tuple of StopperProtocol instances. Examples:
FileStopper, TimeoutStopCondition, SignalStopper, NoImprovementStopper, or custom stopping
logic. If not provided, max_metric_calls must be provided.

    # Logging
    - logger: A `LoggerProtocol` instance that is used to log the progress of the
optimization.
    - run_dir: The directory to save the results to. Optimization state and results will
be saved to this directory. If the directory already exists, GEPA will read the state from
this directory and resume the optimization from the last saved state. If provided, a
FileStopper is automatically created which checks for the presence of "gepa.stop" in this
directory, allowing graceful stopping of the optimization process upon its presence.
    - use_wandb: Whether to use Weights and Biases to log the progress of the
optimization.
    - wandb_api_key: The API key to use for Weights and Biases.
    - wandb_init_kwargs: Additional keyword arguments to pass to the Weights and Biases
initialization.
    - use_mlflow: Whether to use MLflow to log the progress of the optimization.
      Both wandb and mlflow can be used simultaneously if desired.
    - mlflow_tracking_uri: The tracking URI to use for MLflow.
    - mlflow_experiment_name: The experiment name to use for MLflow.
    - track_best_outputs: Whether to track the best outputs on the validation set. If
True, GEPAResult will contain the best outputs obtained for each task in the validation
set.
    - display_progress_bar: Show a tqdm progress bar over metric calls when enabled.
    - use_cloudpickle: Use cloudpickle instead of pickle. This can be helpful when the
serialized state contains dynamically generated DSPy signatures.

    # Reproducibility
    - seed: The seed to use for the random number generator.
    - val_evaluation_policy: Strategy controlling which validation ids to score each
iteration and which candidate is currently best. Supported strings: "full_eval" (evaluate
every id each time) Passing None defaults to "full_eval".
    - raise_on_exception: Whether to propagate proposer/evaluator exceptions instead of
stopping gracefully.
    """
    if adapter is None:
        assert task_lm is not None, (
            "Since no adapter is provided, GEPA requires a task LM to be provided. Please
set the `task_lm` parameter."
        )
        active_adapter: GEPAAdapter[DataInst, Trajectory, RolloutOutput] = cast(
```

```
            GEPAAdapter[DataInst, Trajectory, RolloutOutput],
DefaultAdapter(model=task_lm)
        )
    else:
        assert task_lm is None, (
            "Since an adapter is provided, GEPA does not require a task LM to be provided.
Please set the `task_lm` parameter to None."
        )
        active_adapter = adapter

    # Normalize datasets to DataLoader instances
    train_loader = ensure_loader(trainset)
    val_loader = ensure_loader(valset) if valset is not None else train_loader

    # Comprehensive stop_callback logic
    # Convert stop_callbacks to a list if it's not already
    stop_callbacks_list: list[StopperProtocol] = []
    if stop_callbacks is not None:
        if isinstance(stop_callbacks, Sequence):
            stop_callbacks_list.extend(stop_callbacks)
        else:
            stop_callbacks_list.append(stop_callbacks)

    # Add file stopper if run_dir is provided
    if run_dir is not None:
        stop_file_path = os.path.join(run_dir, "gepa.stop")
        file_stopper = FileStopper(stop_file_path)
        stop_callbacks_list.append(file_stopper)

    # Add max_metric_calls stopper if provided
    if max_metric_calls is not None:
        from gepa.utils import MaxMetricCallsStopper

        max_calls_stopper = MaxMetricCallsStopper(max_metric_calls)
        stop_callbacks_list.append(max_calls_stopper)

    # Assert that at least one stopping condition is provided
    if not stop_callbacks_list:
        raise ValueError(
            "The user must provide at least one of stop_callbacks or max_metric_calls to
specify a stopping condition."
        )

    # Create composite stopper if multiple stoppers, or use single stopper
    stop_callback: StopperProtocol
    if len(stop_callbacks_list) == 1:
        stop_callback = stop_callbacks_list[0]
    else:
        from gepa.utils import CompositeStopper

        stop_callback = CompositeStopper(*stop_callbacks_list)

    if not hasattr(active_adapter, "propose_new_texts"):
        assert reflection_lm is not None, (
            f"reflection_lm was not provided. The adapter used '{active_adapter!s}' does
not provide a propose_new_texts method, "
            + "and hence, GEPA will use the default proposer, which requires a
reflection_lm to be specified."
        )

    if isinstance(reflection_lm, str):
        import litellm

        reflection_lm_name = reflection_lm

        def _reflection_lm(prompt: str) -> str:
            completion = litellm.completion(model=reflection_lm_name, messages=[{"role":
"user", "content": prompt}])
```

```
            return completion.choices[0].message.content  # type: ignore

        reflection_lm = _reflection_lm

    if logger is None:
        logger = StdOutLogger()

    rng = random.Random(seed)

    candidate_selector: CandidateSelector
    if isinstance(candidate_selection_strategy, str):
        factories = {
            "pareto": lambda: ParetoCandidateSelector(rng=rng),
            "current_best": lambda: CurrentBestCandidateSelector(),
            "epsilon_greedy": lambda: EpsilonGreedyCandidateSelector(epsilon=0.1,
rng=rng),
        }

        try:
            candidate_selector = factories[candidate_selection_strategy]()
        except KeyError as exc:
            raise ValueError(
                f"Unknown candidate_selector strategy: {candidate_selection_strategy}. "
                "Supported strategies: 'pareto', 'current_best', 'epsilon_greedy'"
            ) from exc
    elif isinstance(candidate_selection_strategy, CandidateSelector):
        candidate_selector = candidate_selection_strategy
    else:
        raise TypeError(
            "candidate_selection_strategy must be a supported string strategy or an
instance of CandidateSelector."
        )

    if val_evaluation_policy is None or val_evaluation_policy == "full_eval":
        val_evaluation_policy = FullEvaluationPolicy()
    elif not isinstance(val_evaluation_policy, EvaluationPolicy):
        raise ValueError(
            f"val_evaluation_policy should be one of 'full_eval' or an instance of
EvaluationPolicy, but got {type(val_evaluation_policy)}"
        )

    if isinstance(module_selector, str):
        module_selector_cls = {
            "round_robin": RoundRobinReflectionComponentSelector,
            "all": AllReflectionComponentSelector,
        }.get(module_selector)

        assert module_selector_cls is not None, (
            f"Unknown module_selector strategy: {module_selector}. Supported strategies:
'round_robin', 'all'"
        )

        module_selector_instance: ReflectionComponentSelector = module_selector_cls()
    else:
        module_selector_instance = module_selector

    if batch_sampler == "epoch_shuffled":
        batch_sampler = EpochShuffledBatchSampler(minibatch_size=reflection_minibatch_size
or 3, rng=rng)
    else:
        assert reflection_minibatch_size is None, (
            "reflection_minibatch_size only accepted if batch_sampler is 'epoch_shuffled'"
        )

    experiment_tracker = create_experiment_tracker(
        use_wandb=use_wandb,
        wandb_api_key=wandb_api_key,
        wandb_init_kwargs=wandb_init_kwargs,
```

```
        use_mlflow=use_mlflow,
        mlflow_tracking_uri=mlflow_tracking_uri,
        mlflow_experiment_name=mlflow_experiment_name,
    )

    if reflection_prompt_template is not None:
        assert not (adapter is not None and getattr(adapter, "propose_new_texts", None) is
not None), (
            f"Adapter {adapter!s} provides its own propose_new_texts method;
reflection_prompt_template will be ignored. "
            "Set reflection_prompt_template to None."
        )

    reflective_proposer = ReflectiveMutationProposer(
        logger=logger,
        trainset=train_loader,
        adapter=active_adapter,
        candidate_selector=candidate_selector,
        module_selector=module_selector_instance,
        batch_sampler=batch_sampler,
        perfect_score=perfect_score,
        skip_perfect_score=skip_perfect_score,
        experiment_tracker=experiment_tracker,
        reflection_lm=reflection_lm,
        reflection_prompt_template=reflection_prompt_template,
    )

    def evaluator(inputs: list[DataInst], prog: dict[str, str]) ->
tuple[list[RolloutOutput], list[float]]:
        eval_out = active_adapter.evaluate(inputs, prog, capture_traces=False)
        return eval_out.outputs, eval_out.scores

    merge_proposer: MergeProposer | None = None
    if use_merge:
        merge_proposer = MergeProposer(
            logger=logger,
            valset=val_loader,
            evaluator=evaluator,
            use_merge=use_merge,
            max_merge_invocations=max_merge_invocations,
            rng=rng,
            val_overlap_floor=merge_val_overlap_floor,
        )

    engine = GEPAEngine(
        run_dir=run_dir,
        evaluator=evaluator,
        valset=val_loader,
        seed_candidate=seed_candidate,
        perfect_score=perfect_score,
        seed=seed,
        reflective_proposer=reflective_proposer,
        merge_proposer=merge_proposer,
        logger=logger,
        experiment_tracker=experiment_tracker,
        track_best_outputs=track_best_outputs,
        display_progress_bar=display_progress_bar,
        raise_on_exception=raise_on_exception,
        stop_callback=stop_callback,
        val_evaluation_policy=val_evaluation_policy,
        use_cloudpickle=use_cloudpickle,
    )

    with experiment_tracker:
        state = engine.run()

    return GEPAResult.from_state(state, run_dir=run_dir, seed=seed)
```

## src/gepa/core/__init__.py

## src/gepa/core/adapter.py

```
# Copyright (c) 2025 Lakshya A Agrawal and the GEPA contributors
# https://github.com/gepa-ai/gepa

from collections.abc import Callable, Mapping, Sequence
from dataclasses import dataclass
from typing import Any, Generic, Protocol, TypeVar

# Generic type aliases matching your original
RolloutOutput = TypeVar("RolloutOutput")
Trajectory = TypeVar("Trajectory")
DataInst = TypeVar("DataInst")
Candidate = dict[str, str]
EvaluatorFn = Callable[[list[DataInst], Candidate], tuple[list[RolloutOutput],
list[float]]]


@dataclass
class EvaluationBatch(Generic[Trajectory, RolloutOutput]):
    """
    Container for the result of evaluating a proposed candidate on a batch of data.

    - outputs: raw per-example outputs from upon executing the candidate. GEPA does not
interpret these;
      they are forwarded to other parts of the user's code or logging as-is.
    - scores: per-example numeric scores (floats). GEPA sums these for minibatch
acceptance
      and averages them over the full validation set for tracking/pareto fronts.
    - trajectories: optional per-example traces used by make_reflective_dataset to build
      a reflective dataset (See `GEPAAdapter.make_reflective_dataset`). If
capture_traces=True is passed to `evaluate`, trajectories
      should be provided and align one-to-one with `outputs` and `scores`.
    """

    outputs: list[RolloutOutput]
    scores: list[float]
    trajectories: list[Trajectory] | None = None


class ProposalFn(Protocol):
    def __call__(
        self,
        candidate: dict[str, str],
        reflective_dataset: Mapping[str, Sequence[Mapping[str, Any]]],
        components_to_update: list[str],
    ) -> dict[str, str]:
        """
        - Given the current `candidate`, a reflective dataset (as returned by
          `GEPAAdapter.make_reflective_dataset`), and a list of component names to update,
          return a mapping component_name -> new component text (str). This allows the
user
          to implement their own instruction proposal logic. For example, the user can use
          a different LLM, implement DSPy signatures, etc. Another example can be
situations
          where 2 or more components need to be updated together (coupled updates).
```

```
        Returns
        - Dict[str, str] mapping component names to newly proposed component texts.
        """
        ...


class GEPAAdapter(Protocol[DataInst, Trajectory, RolloutOutput]):
    """
    GEPAAdapter is the single integration point between your system
    and the GEPA optimization engine. Implementers provide three responsibilities:

    The following are user-defined types that are not interpreted by GEPA but are used by
the user's code
        to define the adapter:
    DataInst: User-defined type of input data to the program under optimization.
    Trajectory: User-defined type of trajectory data, which typically captures the
        different steps of the program candidate execution.
    RolloutOutput: User-defined type of output data from the program candidate.

    The following are the responsibilities of the adapter:
    1) Program construction and evaluation (evaluate):
       Given a batch of DataInst and a "candidate" program (mapping from named components
       -> component text), execute the program to produce per-example scores and
       optionally rich trajectories (capturing intermediate states) needed for reflection.

    2) Reflective dataset construction (make_reflective_dataset):
       Given the candidate, EvaluationBatch (trajectories, outputs, scores), and the list
of components to update,
       produce a small JSON-serializable dataset for each component that you want to
update. This
       dataset is fed to the teacher LM to propose improved component text.

    3) Optional instruction proposal (propose_new_texts):
       GEPA provides a default implementation (instruction_proposal.py) that serializes
the reflective dataset
       to propose new component texts. However, users can implement their own proposal
logic by implementing this method.
       This method receives the current candidate, the reflective dataset, and the list of
components to update,
       and returns a mapping from component name to new component text.

    Key concepts and contracts:
    - candidate: Dict[str, str] mapping a named component of the system to its
corresponding text.
    - scores: higher is better. GEPA uses:
      - minibatch: sum(scores) to compare old vs. new candidate (acceptance test),
      - full valset: mean(scores) for tracking and Pareto-front selection.
      Ensure your metric is calibrated accordingly or normalized to a consistent scale.
    - trajectories: opaque to GEPA (the engine never inspects them). They must be
      consumable by your own make_reflective_dataset implementation to extract the
      minimal context needed to produce meaningful feedback for every component of
      the system under optimization.
    - error handling: Never raise for individual example failures. Instead:
      - Return a valid `EvaluationBatch` with per-example failure scores (e.g., 0.0)
        when formatting/parsing fails. Even better if the trajectories are also populated
        with the failed example, including the error message, identifying the reason for
the failure.
      - Reserve exceptions for unrecoverable, systemic failures (e.g., missing model,
        misconfigured program, schema mismatch).
      - If an exception is raised, the engine will log the error and proceed to the next
iteration.
    """

    def evaluate(
        self,
        batch: list[DataInst],
        candidate: dict[str, str],
```

```
        capture_traces: bool = False,
    ) -> EvaluationBatch[Trajectory, RolloutOutput]:
        """
        Run the program defined by `candidate` on a batch of data.

        Parameters
        - batch: list of task-specific inputs (DataInst).
        - candidate: mapping from component name -> component text. You must instantiate
          your full system with the component text for each component, and execute it on
the batch.
        - capture_traces: when True, you must populate `EvaluationBatch.trajectories`
          with a per-example trajectory object that your `make_reflective_dataset` can
          later consume. When False, you may set trajectories=None to save time/memory.
          capture_traces=True is used by the reflective mutation proposer to build a
reflective dataset.

        Returns
        - EvaluationBatch with:
          - outputs: raw per-example outputs (opaque to GEPA).
          - scores: per-example floats, length == len(batch). Higher is better.
          - trajectories:
              - if capture_traces=True: list[Trajectory] with length == len(batch).
              - if capture_traces=False: None.

        Scoring semantics
        - The engine uses sum(scores) on minibatches to decide whether to accept a
          candidate mutation and average(scores) over the full valset for tracking.
        - Prefer to return per-example scores, that can be aggregated via summation.
        - If an example fails (e.g., parse error), use a fallback score (e.g., 0.0).

        Correctness constraints
        - len(outputs) == len(scores) == len(batch)
        - If capture_traces=True: trajectories must be provided and len(trajectories) ==
len(batch)
        - Do not mutate `batch` or `candidate` in-place. Construct a fresh program
          instance or deep-copy as needed.
        """
        ...

    def make_reflective_dataset(
        self,
        candidate: dict[str, str],
        eval_batch: EvaluationBatch[Trajectory, RolloutOutput],
        components_to_update: list[str],
    ) -> Mapping[str, Sequence[Mapping[str, Any]]]:
        """
        Build a small, JSON-serializable dataset (per component) to drive instruction
        refinement by a teacher LLM.

        Parameters
        - candidate: the same candidate evaluated in evaluate().
        - eval_batch: The result of evaluate(..., capture_traces=True) on
          the same batch. You should extract everything you need from
eval_batch.trajectories
          (and optionally outputs/scores) to assemble concise, high-signal examples.
        - components_to_update: subset of component names for which the proposer has
          requested updates. At a time, GEPA identifies a subset of components to update.

        Returns
        - A dict: component_name -> list of dict records (the "reflective dataset").
          Each record should be JSON-serializable and is passed verbatim to the
          instruction proposal prompt. A recommended schema is:
            {
              "Inputs": Dict[str, str],             # Minimal, clean view of the inputs to
the component
              "Generated Outputs": Dict[str, str] | str,  # Model outputs or raw text
              "Feedback": str                       # Feedback on the component's
performance, including correct answer, error messages, etc.
```

```
            }
          You may include additional keys (e.g., "score", "rationale", "trace_id") if
useful.

        Determinism
        - If you subsample trace instances, use a seeded RNG to keep runs reproducible.
        """
        ...

    propose_new_texts: ProposalFn | None = None
```

### src/gepa/core/data_loader.py

```
"""Data loader protocols and concrete helpers."""

from __future__ import annotations

from typing import Any, Hashable, Protocol, Sequence, TypeVar, cast, runtime_checkable

from gepa.core.adapter import DataInst


class ComparableHashable(Hashable, Protocol):
    """Protocol requiring hashing and rich comparison support."""

    def __lt__(self, other: Any, /) -> bool: ...

    def __gt__(self, other: Any, /) -> bool: ...

    def __le__(self, other: Any, /) -> bool: ...

    def __ge__(self, other: Any, /) -> bool: ...


DataId = TypeVar("DataId", bound=ComparableHashable)
""" Generic for the identifier for data examples """


@runtime_checkable
class DataLoader(Protocol[DataId, DataInst]):
    """Minimal interface for retrieving validation examples keyed by opaque ids."""

    def all_ids(self) -> Sequence[DataId]:
        """Return the ordered universe of ids currently available. This may change over
time."""
        ...

    def fetch(self, ids: Sequence[DataId]) -> list[DataInst]:
        """Materialise the payloads corresponding to `ids`, preserving order."""
        ...

    def __len__(self) -> int:
        """Return current number of items in the loader."""
        ...


class MutableDataLoader(DataLoader[DataId, DataInst], Protocol):
    """A data loader that can be mutated."""

    def add_items(self, items: list[DataInst]) -> None:
        """Add items to the loader."""


class ListDataLoader(MutableDataLoader[int, DataInst]):
```

```
    """In-memory reference implementation backed by a list."""

    def __init__(self, items: Sequence[DataInst]):
        self.items = list(items)

    def all_ids(self) -> Sequence[int]:
        return list(range(len(self.items)))

    def fetch(self, ids: Sequence[int]) -> list[DataInst]:
        return [self.items[data_id] for data_id in ids]

    def __len__(self) -> int:
        return len(self.items)

    def add_items(self, items: Sequence[DataInst]) -> None:
        self.items.extend(items)


def ensure_loader(data_or_loader: Sequence[DataInst] | DataLoader[DataId, DataInst]) ->
DataLoader[DataId, DataInst]:
    if isinstance(data_or_loader, DataLoader):
        return data_or_loader
    if isinstance(data_or_loader, Sequence):
        return cast(DataLoader[DataId, DataInst], ListDataLoader(data_or_loader))
    raise TypeError(f"Unable to cast to a DataLoader type: {type(data_or_loader)}")
```

### src/gepa/core/engine.py

```
# Copyright (c) 2025 Lakshya A Agrawal and the GEPA contributors
# https://github.com/gepa-ai/gepa

import traceback
from typing import Generic

from gepa.core.adapter import DataInst, EvaluatorFn, RolloutOutput, Trajectory
from gepa.core.data_loader import DataId, DataLoader, ensure_loader
from gepa.core.state import GEPAState, ProgramIdx, initialize_gepa_state
from gepa.logging.experiment_tracker import ExperimentTracker
from gepa.logging.logger import LoggerProtocol
from gepa.logging.utils import log_detailed_metrics_after_discovering_new_program
from gepa.proposer.merge import MergeProposer
from gepa.proposer.reflective_mutation.reflective_mutation import
ReflectiveMutationProposer
from gepa.strategies.eval_policy import EvaluationPolicy, FullEvaluationPolicy
from gepa.utils import StopperProtocol

# Import tqdm for progress bar functionality
try:
    from tqdm import tqdm
except ImportError:
    tqdm = None


class GEPAEngine(Generic[DataId, DataInst, Trajectory, RolloutOutput]):
    """Orchestrates the optimization loop using pluggable candidate proposers."""

    def __init__(
        self,
        run_dir: str | None,
        evaluator: EvaluatorFn,
        valset: list[DataInst] | DataLoader[DataId, DataInst] | None,
        seed_candidate: dict[str, str],
        # Controls
        perfect_score: float,
```

```
        seed: int,
        # Strategies and helpers
        reflective_proposer: ReflectiveMutationProposer,
        merge_proposer: MergeProposer | None,
        # Logging
        logger: LoggerProtocol,
        experiment_tracker: ExperimentTracker,
        # Optional parameters
        track_best_outputs: bool = False,
        display_progress_bar: bool = False,
        raise_on_exception: bool = True,
        use_cloudpickle: bool = False,
        # Budget and Stop Condition
        stop_callback: StopperProtocol | None = None,
        val_evaluation_policy: EvaluationPolicy[DataId, DataInst] | None = None,
    ):
        self.logger = logger
        self.run_dir = run_dir

        # Graceful stopping mechanism
        self._stop_requested = False

        # Set up stopping mechanism
        self.stop_callback = stop_callback
        self.evaluator = evaluator
        self.valset = ensure_loader(valset) if valset is not None else None
        self.seed_candidate = seed_candidate

        self.perfect_score = perfect_score
        self.seed = seed
        self.experiment_tracker = experiment_tracker

        self.reflective_proposer = reflective_proposer
        self.merge_proposer = merge_proposer
        if self.merge_proposer is not None:
            self.merge_proposer.last_iter_found_new_program = False

        self.track_best_outputs = track_best_outputs
        self.display_progress_bar = display_progress_bar
        self.use_cloudpickle = use_cloudpickle

        self.raise_on_exception = raise_on_exception
        self.val_evaluation_policy: EvaluationPolicy[DataId, DataInst] = (
            val_evaluation_policy if val_evaluation_policy is not None else
FullEvaluationPolicy()
        )

    def _evaluate_on_valset(
        self,
        program: dict[str, str],
        state: GEPAState[RolloutOutput, DataId],
    ) -> tuple[dict[DataId, RolloutOutput], dict[DataId, float]]:
        valset = self.valset
        assert valset is not None

        val_ids = self.val_evaluation_policy.get_eval_batch(valset, state)
        batch = valset.fetch(val_ids)
        outputs, scores = self.evaluator(batch, program)
        assert len(outputs) == len(val_ids), "Eval outputs should match length of selected
validation indices"

        outputs_by_val_idx = dict(zip(val_ids, outputs, strict=False))
        scores_by_val_idx = dict(zip(val_ids, scores, strict=False))
        return outputs_by_val_idx, scores_by_val_idx

    def _get_pareto_front_programs(self, state: GEPAState[RolloutOutput, DataId]) ->
dict[DataId, set[ProgramIdx]]:
        return state.program_at_pareto_front_valset
```

```
    def _run_full_eval_and_add(
        self,
        new_program: dict[str, str],
        state: GEPAState[RolloutOutput, DataId],
        parent_program_idx: list[int],
    ) -> tuple[int, int]:
        num_metric_calls_by_discovery = state.total_num_evals

        valset_outputs, valset_subscores = self._evaluate_on_valset(new_program, state)

        state.num_full_ds_evals += 1
        state.total_num_evals += len(valset_subscores)

        new_program_idx = state.update_state_with_new_program(
            parent_program_idx=parent_program_idx,
            new_program=new_program,
            valset_outputs=valset_outputs,
            valset_subscores=valset_subscores,
            run_dir=self.run_dir,
            num_metric_calls_by_discovery_of_new_program=num_metric_calls_by_discovery,
        )
        state.full_program_trace[-1]["new_program_idx"] = new_program_idx
        state.full_program_trace[-1]["evaluated_val_indices"] =
sorted(valset_subscores.keys())

        valset_score = self.val_evaluation_policy.get_valset_score(new_program_idx, state)

        linear_pareto_front_program_idx =
self.val_evaluation_policy.get_best_program(state)
        if new_program_idx == linear_pareto_front_program_idx:
            self.logger.log(f"Iteration {state.i + 1}: Found a better program on the
valset with score {valset_score}.")

        valset = self.valset
        assert valset is not None

        log_detailed_metrics_after_discovering_new_program(
            logger=self.logger,
            gepa_state=state,
            new_program_idx=new_program_idx,
            valset_subscores=valset_subscores,
            experiment_tracker=self.experiment_tracker,
            linear_pareto_front_program_idx=linear_pareto_front_program_idx,
            valset_size=len(valset),
            val_evaluation_policy=self.val_evaluation_policy,
        )
        return new_program_idx, linear_pareto_front_program_idx

    def run(self) -> GEPAState[RolloutOutput, DataId]:
        # Check tqdm availability if progress bar is enabled
        progress_bar = None
        if self.display_progress_bar:
            if tqdm is None:
                raise ImportError("tqdm must be installed when display_progress_bar is
enabled")

            # Check if stop_callback contains MaxMetricCallsStopper
            total_calls: int | None = None
            stop_cb = self.stop_callback
            if stop_cb is not None:
                max_calls_attr = getattr(stop_cb, "max_metric_calls", None)
                if isinstance(max_calls_attr, int):
                    # Direct MaxMetricCallsStopper
                    total_calls = max_calls_attr
                else:
                    stoppers = getattr(stop_cb, "stoppers", None)
                    if stoppers is not None:
```

```
                        # CompositeStopper - iterate to find MaxMetricCallsStopper
                        for stopper in stoppers:
                            stopper_max = getattr(stopper, "max_metric_calls", None)
                            if isinstance(stopper_max, int):
                                total_calls = stopper_max
                                break

            if total_calls is not None:
                progress_bar = tqdm(total=total_calls, desc="GEPA Optimization",
unit="rollouts")
            else:
                progress_bar = tqdm(desc="GEPA Optimization", unit="rollouts")
            progress_bar.update(0)

        # Prepare valset
        valset = self.valset
        if valset is None:
            raise ValueError("valset must be provided to GEPAEngine.run()")

        def valset_evaluator(program: dict[str, str]) -> tuple[dict[DataId,
RolloutOutput], dict[DataId, float]]:
            all_ids = list(valset.all_ids())
            all_outputs, all_scores = self.evaluator(valset.fetch(all_ids), program)
            return (
                dict(zip(all_ids, all_outputs, strict=False)),
                dict(zip(all_ids, all_scores, strict=False)),
            )

        # Initialize state
        state = initialize_gepa_state(
            run_dir=self.run_dir,
            logger=self.logger,
            seed_candidate=self.seed_candidate,
            valset_evaluator=valset_evaluator,
            track_best_outputs=self.track_best_outputs,
        )

        # Log base program score
        base_val_avg, base_val_coverage = state.get_program_average_val_subset(0)
        self.experiment_tracker.log_metrics(
            {
                "base_program_full_valset_score": base_val_avg,
                "base_program_val_coverage": base_val_coverage,
                "iteration": state.i + 1,
            },
            step=state.i + 1,
        )

        self.logger.log(
            f"Iteration {state.i + 1}: Base program full valset score: {base_val_avg} "
            f"over {base_val_coverage} / {len(valset)} examples"
        )

        # Merge scheduling
        if self.merge_proposer is not None:
            self.merge_proposer.last_iter_found_new_program = False

        # Main loop
        last_pbar_val = 0
        while not self._should_stop(state):
            if self.display_progress_bar and progress_bar is not None:
                delta = state.total_num_evals - last_pbar_val
                progress_bar.update(delta)
                last_pbar_val = state.total_num_evals

            assert state.is_consistent()
            try:
                state.save(self.run_dir, use_cloudpickle=self.use_cloudpickle)
```

```
                state.i += 1
                state.full_program_trace.append({"i": state.i})

                # 1) Attempt merge first if scheduled and last iter found new program
                if self.merge_proposer is not None and self.merge_proposer.use_merge:
                    if self.merge_proposer.merges_due > 0 and
self.merge_proposer.last_iter_found_new_program:
                        proposal = self.merge_proposer.propose(state)
                        self.merge_proposer.last_iter_found_new_program = False  # old
behavior

                        if proposal is not None and proposal.tag == "merge":
                            parent_sums = proposal.subsample_scores_before or [float("-
inf"), float("-inf")]
                            new_sum = sum(proposal.subsample_scores_after or [])

                            if new_sum >= max(parent_sums):
                                # ACCEPTED: consume one merge attempt and record it
                                self._run_full_eval_and_add(
                                    new_program=proposal.candidate,
                                    state=state,
                                    parent_program_idx=proposal.parent_program_ids,
                                )
                                self.merge_proposer.merges_due -= 1
                                self.merge_proposer.total_merges_tested += 1
                                continue  # skip reflective this iteration
                            else:
                                # REJECTED: do NOT consume merges_due or
total_merges_tested
                                self.logger.log(
                                    f"Iteration {state.i + 1}: New program subsample score
{new_sum} "
                                    f"is worse than both parents {parent_sums}, skipping
merge"
                                )
                                # Skip reflective this iteration (old behavior)
                                continue

                    # Old behavior: regardless of whether we attempted, clear the flag
before reflective
                    self.merge_proposer.last_iter_found_new_program = False

                # 2) Reflective mutation proposer
                proposal = self.reflective_proposer.propose(state)
                if proposal is None:
                    self.logger.log(f"Iteration {state.i + 1}: Reflective mutation did not
propose a new candidate")
                    continue

                # Acceptance: require strict improvement on subsample
                old_sum = sum(proposal.subsample_scores_before or [])
                new_sum = sum(proposal.subsample_scores_after or [])
                if new_sum <= old_sum:
                    self.logger.log(
                        f"Iteration {state.i + 1}: New subsample score {new_sum} is not
better than old score {old_sum}, skipping"
                    )
                    continue
                else:
                    self.logger.log(
                        f"Iteration {state.i + 1}: New subsample score {new_sum} is better
than old score {old_sum}. Continue to full eval and add to candidate pool."
                    )

                # Accept: full eval + add
                self._run_full_eval_and_add(
                    new_program=proposal.candidate,
                    state=state,
```

```
                    parent_program_idx=proposal.parent_program_ids,
                )

                # Schedule merge attempts like original behavior
                if self.merge_proposer is not None:
                    self.merge_proposer.last_iter_found_new_program = True
                    if self.merge_proposer.total_merges_tested <
self.merge_proposer.max_merge_invocations:
                        self.merge_proposer.merges_due += 1

            except Exception as e:
                self.logger.log(f"Iteration {state.i + 1}: Exception during optimization:
{e}")
                self.logger.log(traceback.format_exc())
                if self.raise_on_exception:
                    raise e
                else:
                    continue

        # Close progress bar if it exists
        if self.display_progress_bar and progress_bar is not None:
            progress_bar.close()

        state.save(self.run_dir)
        return state

    def _should_stop(self, state: GEPAState[RolloutOutput, DataId]) -> bool:
        """Check if the optimization should stop."""
        if self._stop_requested:
            return True
        if self.stop_callback and self.stop_callback(state):
            return True
        return False

    def request_stop(self) -> None:
        """Manually request the optimization to stop gracefully."""
        self.logger.log("Stop requested manually. Initiating graceful shutdown...")
        self._stop_requested = True
```

## src/gepa/core/result.py

```
# Copyright (c) 2025 Lakshya A Agrawal and the GEPA contributors
# https://github.com/gepa-ai/gepa

from dataclasses import dataclass
from typing import TYPE_CHECKING, Any, ClassVar, Generic

from gepa.core.adapter import RolloutOutput
from gepa.core.data_loader import DataId
from gepa.core.state import ProgramIdx

if TYPE_CHECKING:
    from gepa.core.state import GEPAState


@dataclass(frozen=True)
class GEPAResult(Generic[RolloutOutput, DataId]):
    """
    Immutable snapshot of a GEPA run with convenience accessors.

    - candidates: list of proposed candidates (component_name -> component_text)
    - parents: lineage info; for each candidate i, parents[i] is a list of parent indices
or None
    - val_aggregate_scores: per-candidate aggregate score on the validation set (higher is
```

```
better)
    - val_subscores: per-candidate mapping from validation id to score on the validation
set (sparse dict)
    - per_val_instance_best_candidates: for each val instance t, a set of candidate
indices achieving the current best score on t
    - discovery_eval_counts: number of metric calls accumulated up to the discovery of
each candidate

    Optional fields:
    - best_outputs_valset: per-task best outputs on the validation set. [task_idx ->
[(program_idx_1, output_1), (program_idx_2, output_2), ...]]

    Run-level metadata:
    - total_metric_calls: total number of metric calls made across the run
    - num_full_val_evals: number of full validation evaluations performed
    - run_dir: where artifacts were written (if any)
    - seed: RNG seed for reproducibility (if known)
    - tracked_scores: optional tracked aggregate scores (if different from
val_aggregate_scores)

    Convenience:
    - best_idx: candidate index with the highest val_aggregate_scores
    - best_candidate: the program text mapping for best_idx
    - non_dominated_indices(): candidate indices that are not dominated across per-
instance pareto fronts
    - lineage(idx): parent chain from base to idx
    - diff(parent_idx, child_idx, only_changed=True): component-wise diff between two
candidates
    - best_k(k): top-k candidates by aggregate val score
    - instance_winners(t): set of candidates on the pareto front for val instance t
    - to_dict(...), save_json(...): serialization helpers
    """

    # Core data
    candidates: list[dict[str, str]]
    parents: list[list[ProgramIdx | None]]
    val_aggregate_scores: list[float]
    val_subscores: list[dict[DataId, float]]
    per_val_instance_best_candidates: dict[DataId, set[ProgramIdx]]
    discovery_eval_counts: list[int]

    # Optional data
    best_outputs_valset: dict[DataId, list[tuple[ProgramIdx, RolloutOutput]]] | None =
None

    # Run metadata (optional)
    total_metric_calls: int | None = None
    num_full_val_evals: int | None = None
    run_dir: str | None = None
    seed: int | None = None

    _VALIDATION_SCHEMA_VERSION: ClassVar[int] = 2

    # -------- Convenience properties --------
    @property
    def num_candidates(self) -> int:
        return len(self.candidates)

    @property
    def num_val_instances(self) -> int:
        return len(self.per_val_instance_best_candidates)

    @property
    def best_idx(self) -> int:
        scores = self.val_aggregate_scores
        return max(range(len(scores)), key=lambda i: scores[i])

    @property
```

```
    def best_candidate(self) -> dict[str, str]:
        return self.candidates[self.best_idx]

    def to_dict(self) -> dict[str, Any]:
        cands = [dict(cand.items()) for cand in self.candidates]

        return {
            "candidates": cands,
            "parents": self.parents,
            "val_aggregate_scores": self.val_aggregate_scores,
            "val_subscores": self.val_subscores,
            "best_outputs_valset": self.best_outputs_valset,
            "per_val_instance_best_candidates": {
                val_id: list(front) for val_id, front in
self.per_val_instance_best_candidates.items()
            },
            "discovery_eval_counts": self.discovery_eval_counts,
            "total_metric_calls": self.total_metric_calls,
            "num_full_val_evals": self.num_full_val_evals,
            "run_dir": self.run_dir,
            "seed": self.seed,
            "best_idx": self.best_idx,
            "validation_schema_version": GEPAResult._VALIDATION_SCHEMA_VERSION,
        }

    @staticmethod
    def from_dict(d: dict[str, Any]) -> "GEPAResult[RolloutOutput, DataId]":
        version = d.get("validation_schema_version") or 0
        if version > GEPAResult._VALIDATION_SCHEMA_VERSION:
            raise ValueError(
                f"Unsupported GEPAResult validation schema version {version}; "
                f"max supported is {GEPAResult._VALIDATION_SCHEMA_VERSION}"
            )

        if version <= 1:
            return GEPAResult._migrate_from_dict_v0(d)

        return GEPAResult._from_dict_v2(d)

    @staticmethod
    def _common_kwargs_from_dict(d: dict[str, Any]) -> dict[str, Any]:
        return {
            "candidates": [dict(candidate) for candidate in d.get("candidates", [])],
            "parents": [list(parent_row) for parent_row in d.get("parents", [])],
            "val_aggregate_scores": list(d.get("val_aggregate_scores", [])),
            "discovery_eval_counts": list(d.get("discovery_eval_counts", [])),
            "total_metric_calls": d.get("total_metric_calls"),
            "num_full_val_evals": d.get("num_full_val_evals"),
            "run_dir": d.get("run_dir"),
            "seed": d.get("seed"),
        }

    @staticmethod
    def _migrate_from_dict_v0(d: dict[str, Any]) -> "GEPAResult[RolloutOutput, DataId]":
        kwargs = GEPAResult._common_kwargs_from_dict(d)
        kwargs["val_subscores"] = [
            {idx: score for idx, score in enumerate(scores)} for scores in
d.get("val_subscores", [])
        ]
        kwargs["per_val_instance_best_candidates"] = {
            idx: set(front) for idx, front in
enumerate(d.get("per_val_instance_best_candidates", []))
        }

        best_outputs_valset = d.get("best_outputs_valset")
        if best_outputs_valset is not None:
            kwargs["best_outputs_valset"] = {
                idx: [(program_idx, output) for program_idx, output in outputs]
```

```
                for idx, outputs in enumerate(best_outputs_valset)
            }
        else:
            kwargs["best_outputs_valset"] = None
        return GEPAResult(**kwargs)

    @staticmethod
    def _from_dict_v2(d: dict[str, Any]) -> "GEPAResult[RolloutOutput, DataId]":
        kwargs = GEPAResult._common_kwargs_from_dict(d)
        kwargs["val_subscores"] = [dict(scores) for scores in d.get("val_subscores", [])]
        per_val_instance_best_candidates_data = d.get("per_val_instance_best_candidates",
{})
        kwargs["per_val_instance_best_candidates"] = {
            val_id: set(candidates_on_front)
            for val_id, candidates_on_front in
per_val_instance_best_candidates_data.items()
        }

        best_outputs_valset = d.get("best_outputs_valset")
        if best_outputs_valset is not None:
            kwargs["best_outputs_valset"] = {
                val_id: [(program_idx, output) for program_idx, output in outputs]
                for val_id, outputs in best_outputs_valset.items()
            }
        else:
            kwargs["best_outputs_valset"] = None

        return GEPAResult(**kwargs)

    @staticmethod
    def from_state(
        state: "GEPAState[RolloutOutput, DataId]",
        run_dir: str | None = None,
        seed: int | None = None,
    ) -> "GEPAResult[RolloutOutput, DataId]":
        """Build a GEPAResult from a GEPAState."""

        return GEPAResult(
            candidates=list(state.program_candidates),
            parents=list(state.parent_program_for_candidate),
            val_aggregate_scores=list(state.program_full_scores_val_set),
            best_outputs_valset=getattr(state, "best_outputs_valset", None),
            val_subscores=[dict(scores) for scores in state.prog_candidate_val_subscores],
            per_val_instance_best_candidates={
                val_id: set(front) for val_id, front in
state.program_at_pareto_front_valset.items()
            },
            discovery_eval_counts=list(state.num_metric_calls_by_discovery),
            total_metric_calls=getattr(state, "total_num_evals", None),
            num_full_val_evals=getattr(state, "num_full_ds_evals", None),
            run_dir=run_dir,
            seed=seed,
        )
```

### src/gepa/core/state.py

```
# Copyright (c) 2025 Lakshya A Agrawal and the GEPA contributors
# https://github.com/gepa-ai/gepa

import json
import os
from collections import defaultdict
from collections.abc import Callable
from typing import Any, ClassVar, Generic
```

```
from gepa.core.adapter import RolloutOutput
from gepa.core.data_loader import DataId
from gepa.gepa_utils import json_default
from gepa.logging.logger import LoggerProtocol

# Types for GEPAState
ProgramIdx = int
"""Opaque identifier for program candidates."""


class GEPAState(Generic[RolloutOutput, DataId]):
    """Persistent optimizer state tracking candidates, sparse validation coverage, and
execution metadata."""

    _VALIDATION_SCHEMA_VERSION: ClassVar[int] = 2

    program_candidates: list[dict[str, str]]
    parent_program_for_candidate: list[list[ProgramIdx | None]]
    prog_candidate_val_subscores: list[dict[DataId, float]]

    pareto_front_valset: dict[DataId, float]
    program_at_pareto_front_valset: dict[DataId, set[ProgramIdx]]

    list_of_named_predictors: list[str]
    named_predictor_id_to_update_next_for_program_candidate: list[int]

    i: int
    num_full_ds_evals: int

    total_num_evals: int

    num_metric_calls_by_discovery: list[int]

    full_program_trace: list[dict[str, Any]]
    best_outputs_valset: dict[DataId, list[tuple[ProgramIdx, RolloutOutput]]] | None

    validation_schema_version: int

    def __init__(
        self,
        seed_candidate: dict[str, str],
        base_valset_eval_output: tuple[dict[DataId, RolloutOutput], dict[DataId, float]],
        track_best_outputs: bool = False,
    ):
        base_outputs, base_scores = base_valset_eval_output
        self.program_candidates = [seed_candidate]
        self.prog_candidate_val_subscores = [base_scores]

        self.pareto_front_valset = {val_id: score for val_id, score in
base_scores.items()}
        self.parent_program_for_candidate = [[None]]
        self.program_at_pareto_front_valset = {val_id: {0} for val_id in
base_scores.keys()}

        self.list_of_named_predictors = list(seed_candidate.keys())
        self.named_predictor_id_to_update_next_for_program_candidate = [0]
        self.i = -1

        self.num_metric_calls_by_discovery = [0]

        if track_best_outputs:
            self.best_outputs_valset = {
                val_id: [(0, output)] for val_id, output in base_outputs.items()
            }
        else:
            self.best_outputs_valset = None
```

```
        self.full_program_trace = []
        self.validation_schema_version = self._VALIDATION_SCHEMA_VERSION

    def is_consistent(self) -> bool:
        assert len(self.program_candidates) == len(self.parent_program_for_candidate)
        assert len(self.program_candidates) ==
len(self.named_predictor_id_to_update_next_for_program_candidate)

        assert len(self.prog_candidate_val_subscores) == len(self.program_candidates)
        assert len(self.pareto_front_valset) == len(self.program_at_pareto_front_valset)
        assert len(self.program_candidates) == len(self.num_metric_calls_by_discovery)

        for front in self.program_at_pareto_front_valset.values():
            for prog_idx in front:
                assert prog_idx < len(self.program_candidates), (
                    "Program index in valset pareto front exceeds number of program
candidates"
                )

        assert set(self.pareto_front_valset.keys()) ==
set(self.program_at_pareto_front_valset.keys())

        return True

    def save(self, run_dir: str | None, *, use_cloudpickle: bool = False) -> None:
        if run_dir is None:
            return
        with open(os.path.join(run_dir, "gepa_state.bin"), "wb") as f:
            if use_cloudpickle:
                import cloudpickle as pickle  # pragma: no cover - optional dependency
            else:
                import pickle
            serialized = dict(self.__dict__.items())
            serialized["validation_schema_version"] = GEPAState._VALIDATION_SCHEMA_VERSION
            pickle.dump(serialized, f)

    @staticmethod
    def load(run_dir: str) -> "GEPAState[RolloutOutput, DataId]":
        with open(os.path.join(run_dir, "gepa_state.bin"), "rb") as f:
            import pickle

            data = pickle.load(f)

        # handle schema migration
        version = data.get("validation_schema_version")
        if version is None or version == 1:
            GEPAState._migrate_from_legacy_state_v0(data)

        state = GEPAState.__new__(GEPAState)
        state.__dict__.update(data)

        assert len(state.program_candidates) == len(state.program_full_scores_val_set)
        assert len(state.pareto_front_valset) == len(state.program_at_pareto_front_valset)

        assert len(state.program_candidates) == len(state.parent_program_for_candidate)
        assert len(state.program_candidates) ==
len(state.named_predictor_id_to_update_next_for_program_candidate)
        return state

    @staticmethod
    def _migrate_from_legacy_state_v0(d: dict[str, Any]) -> None:
        assert isinstance(d, dict)
        assert "prog_candidate_val_subscores" in d
        assert isinstance(d["prog_candidate_val_subscores"], list)
        assert all(isinstance(scores, list) for scores in
d["prog_candidate_val_subscores"])
        legacy_scores: list[list[float]] = d.pop("prog_candidate_val_subscores", [])
        # convert to sparse val subscores
```

```
        d["prog_candidate_val_subscores"] = [
            {idx: score for idx, score in enumerate(scores)} for scores in legacy_scores
        ]

        pareto_front = d.get("pareto_front_valset")
        if isinstance(pareto_front, list):
            d["pareto_front_valset"] = {idx: score for idx, score in
enumerate(pareto_front)}

        program_at_front = d.get("program_at_pareto_front_valset")
        if isinstance(program_at_front, list):
            d["program_at_pareto_front_valset"] = {idx: set(front) for idx, front in
enumerate(program_at_front)}

        best_outputs = d.get("best_outputs_valset")
        if isinstance(best_outputs, list):
            d["best_outputs_valset"] = {idx: list(outputs) for idx, outputs in
enumerate(best_outputs)}

        d["validation_schema_version"] = GEPAState._VALIDATION_SCHEMA_VERSION

    def get_program_average_val_subset(self, program_idx: int) -> tuple[float, int]:
        # TODO: This should be only used/handled by the val_evaluation_policy, and never
used directly.
        scores = self.prog_candidate_val_subscores[program_idx]
        if not scores:
            return float("-inf"), 0
        num_samples = len(scores)
        avg = sum(scores.values()) / num_samples
        return avg, num_samples

    @property
    def valset_evaluations(self) -> dict[DataId, list[ProgramIdx]]:
        """
        Valset examples by id and programs that have evaluated them. Keys include only
validation
        ids that have been scored at least once.
        """
        result: dict[DataId, list[ProgramIdx]] = defaultdict(list)
        for program_idx, val_scores in enumerate(self.prog_candidate_val_subscores):
            for val_id in val_scores.keys():
                result[val_id].append(program_idx)
        return result

    @property
    def program_full_scores_val_set(self) -> list[float]:
        # TODO: This should be using the val_evaluation_policy instead of the
get_program_average_val_subset method to calculate the scores.
        return [
            self.get_program_average_val_subset(program_idx)[0]
            for program_idx in range(len(self.prog_candidate_val_subscores))
        ]

    def _update_pareto_front_for_val_id(
        self,
        val_id: DataId,
        score: float,
        program_idx: ProgramIdx,
        output: RolloutOutput | None,
        run_dir: str | None,
        iteration: int,
    ) -> None:
        prev_score = self.pareto_front_valset.get(val_id, float("-inf"))
        if score > prev_score:
            self.pareto_front_valset[val_id] = score
            self.program_at_pareto_front_valset[val_id] = {program_idx}
            if self.best_outputs_valset is not None and output is not None:
                self.best_outputs_valset[val_id] = [(program_idx, output)]
```

```
                if run_dir is not None:
                    task_dir = os.path.join(run_dir, "generated_best_outputs_valset",
f"task_{val_id}")
                    os.makedirs(task_dir, exist_ok=True)
                    with open(os.path.join(task_dir,
f"iter_{iteration}_prog_{program_idx}.json"), "w") as fout:
                        json.dump(output, fout, indent=4, default=json_default)
        elif score == prev_score:
            assert self.program_at_pareto_front_valset.get(val_id), (
                f"Program at pareto front for val_id {val_id} should be non-empty"
            )
            pareto_front = self.program_at_pareto_front_valset[val_id]
            pareto_front.add(program_idx)
            if self.best_outputs_valset is not None and output is not None:
                self.best_outputs_valset[val_id].append((program_idx, output))

    def update_state_with_new_program(
        self,
        parent_program_idx: list[ProgramIdx],
        new_program: dict[str, str],
        valset_subscores: dict[DataId, float],
        valset_outputs: dict[DataId, RolloutOutput] | None,
        run_dir: str | None,
        num_metric_calls_by_discovery_of_new_program: int,
    ) -> ProgramIdx:
        new_program_idx = len(self.program_candidates)
        self.program_candidates.append(new_program)

self.num_metric_calls_by_discovery.append(num_metric_calls_by_discovery_of_new_program)

        max_predictor_id = max(
            [self.named_predictor_id_to_update_next_for_program_candidate[p] for p in
parent_program_idx],
            default=0,
        )

self.named_predictor_id_to_update_next_for_program_candidate.append(max_predictor_id)
        self.parent_program_for_candidate.append(list(parent_program_idx))

        self.prog_candidate_val_subscores.append(valset_subscores)
        for val_id, score in valset_subscores.items():
            valset_output = valset_outputs.get(val_id) if valset_outputs else None
            self._update_pareto_front_for_val_id(val_id, score, new_program_idx,
valset_output, run_dir, self.i + 1)
        return new_program_idx


def write_eval_scores_to_directory(scores: dict[DataId, float], output_dir: str) -> None:
    for val_id, score in scores.items():
        task_dir = os.path.join(output_dir, f"task_{val_id}")
        os.makedirs(task_dir, exist_ok=True)
        with open(os.path.join(task_dir, f"iter_{0}_prog_0.json"), "w") as f:
            json.dump(score, f, indent=4, default=json_default)


def initialize_gepa_state(
    run_dir: str | None,
    logger: LoggerProtocol,
    seed_candidate: dict[str, str],
    valset_evaluator: Callable[[dict[str, str]], tuple[dict[DataId, RolloutOutput],
dict[DataId, float]]],
    track_best_outputs: bool = False,
) -> GEPAState[RolloutOutput, DataId]:
    if run_dir is not None and os.path.exists(os.path.join(run_dir, "gepa_state.bin")):
        logger.log("Loading gepa state from run dir")
        gepa_state = GEPAState.load(run_dir)
    else:
        num_evals_run = 0
```

```
        seed_val_outputs, seed_val_scores = valset_evaluator(seed_candidate)
        if run_dir is not None:
            write_eval_scores_to_directory(seed_val_scores, os.path.join(run_dir,
"generated_best_outputs_valset"))
        num_evals_run += len(seed_val_scores)

        gepa_state = GEPAState(
            seed_candidate,
            (seed_val_outputs, seed_val_scores),
            track_best_outputs=track_best_outputs,
        )

        gepa_state.num_full_ds_evals = 1
        gepa_state.total_num_evals = num_evals_run

    return gepa_state
```

### src/gepa/examples/__init__.py

### src/gepa/examples/aime.py

```
# Copyright (c) 2025 Lakshya A Agrawal and the GEPA contributors
# https://github.com/gepa-ai/gepa


def init_dataset():
    import random

    from datasets import load_dataset

    train_split = [
        {"input": x["problem"], "additional_context": {"solution": x["solution"]},
"answer": "### " + str(x["answer"])}
        for x in load_dataset("AI-MO/aimo-validation-aime")["train"]
    ]
    random.Random(0).shuffle(train_split)
    test_split = [
        {"input": x["problem"], "answer": "### " + str(x["answer"])}
        for x in load_dataset("MathArena/aime_2025")["train"]
    ]

    trainset = train_split[: len(train_split) // 2]
    valset = train_split[len(train_split) // 2 :]
    testset = test_split * 5

    return trainset, valset, testset
```

### src/gepa/examples/anymaths-bench/eval_default.py

```
from train_anymaths import init_dataset

from gepa.adapters.anymaths_adapter.anymaths_adapter import AnyMathsStructuredOutput
```

```
if __name__ == "__main__":
    import argparse
    import ast
    from pathlib import Path

    import litellm
    from tqdm import tqdm

    parser = argparse.ArgumentParser()
    parser.add_argument("--anymaths_dset_name", type=str, default="openai/gsm8k")
    parser.add_argument("--model", type=str, default="ollama/qwen3:4b", help="The model to
evaluate.")
    parser.add_argument("--use_api_url", action="store_true", help="Whether to use the API
URL.")
    parser.add_argument("--api_url", type=str, default="http://localhost:11434", help="The
API URL to use.")
    parser.add_argument("--batch_size", type=int, default=8, help="The batch size for
evaluation.")
    parser.add_argument(
        "--max_litellm_workers", type=int, default=1, help="The maximum number of LiteLLM
workers to use."
    )
    parser.add_argument(
        "--which_prompt",
        type=str,
        default="seed",
        choices=["seed", "optimized"],
        help="The prompt to use for evaluation.",
    )

    args = parser.parse_args()

    dataset = args.anymaths_dset_name

    use_api_url = args.use_api_url
    if not use_api_url:
        api_url = ""
    else:
        api_url = args.api_url

    model = args.model
    max_litellm_workers = args.max_litellm_workers

    _, _, testset = init_dataset(dataset)

    if args.which_prompt == "seed":
        INSTRUCTION_PROMPT_PATH = Path(__file__).parent / "prompt-
templates/instruction_prompt.txt"
    else:
        INSTRUCTION_PROMPT_PATH = Path(__file__).parent / "prompt-
templates/optimal_prompt.txt"

    instruction = INSTRUCTION_PROMPT_PATH.read_text()

    batched_testset = []
    batch_size = args.batch_size

    for i in range(0, len(testset), batch_size):
        batched_testset.append(testset[i : i + batch_size])

    total_score = 0.0

    print("-" * 100)
    print(f"Evaluating model: {model}")
    print(f"Using API URL: {api_url if api_url else 'No API URL'}")
    print(f"Batch size: {batch_size}")
    print(f"Max LiteLLM workers: {max_litellm_workers}")
```

```
    print(f"Using prompt: {args.which_prompt}")
    print("-" * 100)

    with tqdm(total=len(testset), desc="Evaluating") as pbar:
        for batch in batched_testset:
            litellm_requests = []

            for item in batch:
                user_content = f"{item['input']}"
                messages = [{"role": "system", "content": instruction}, {"role": "user",
"content": user_content}]

                litellm_requests.append(messages)

            try:
                responses = litellm.batch_completion(
                    model=model,
                    messages=litellm_requests,
                    api_base=api_url,
                    max_workers=max_litellm_workers,
                    format=AnyMathsStructuredOutput.model_json_schema(),
                    response_format={
                        "type": "json_object",
                        "response_schema": AnyMathsStructuredOutput.model_json_schema(),
                        "enforce_validation": True,
                    },
                )
            except litellm.exceptions.JSONSchemaValidationError as e:
                raise e

            for response, item in zip(responses, batch, strict=False):
                correct_output_format = True
                try:
                    assistant_response =
ast.literal_eval(response.choices[0].message.content.strip())
                    assistant_final_answer = assistant_response["final_answer"]
                    ground_truth = item["answer"]
                    score = 1.0 if ground_truth in assistant_final_answer else 0.0
                    total_score += score
                except Exception:
                    correct_output_format = False
                    continue

            pbar.update(len(batch))
            pbar.set_postfix({"Score": f"{total_score} / {len(testset):.4f}"})

    print("-" * 100)
    print(f"Final score >> {total_score} / {len(testset):.4f}")
    print("-" * 100)
```

### src/gepa/examples/anymaths-bench/train_anymaths.py

```
def init_dataset(anymaths_dset_name: str = "openai/gsm8k"):
    import random

    from datasets import load_dataset

    train_split = []
    test_split = []
    match anymaths_dset_name:
        case "openai/gsm8k":
            train_load_dataset = load_dataset(anymaths_dset_name, "main", split="train")
            for item in train_load_dataset:
                answer = item["answer"].split("####")[-1].strip()
```

```
                solution = item["answer"].split("####")[0].strip()
                question = item["question"]

                train_split.append({"input": question, "additional_context": {"solution":
solution}, "answer": answer})

            random.Random(0).shuffle(train_split)

            test_load_dataset = load_dataset(anymaths_dset_name, "main", split="test")
            for item in test_load_dataset:
                answer = item["answer"].split("####")[-1].strip()
                solution = item["answer"].split("####")[0].strip()
                question = item["question"]

                test_split.append({"input": question, "answer": answer})

        case "MathArena/aime_2025":
            train_load_dataset = load_dataset("AI-MO/aimo-validation-aime", "default",
split="train")
            for item in train_load_dataset:
                question = item["problem"]
                solution = item["solution"]
                answer = item["answer"]

                train_split.append({"input": question, "additional_context": {"solution":
solution}, "answer": answer})

            random.Random(0).shuffle(train_split)

            test_load_dataset = load_dataset("MathArena/aime_2025", "default",
split="train")
            for item in test_load_dataset:
                question = item["problem"]
                answer = item["answer"]

                test_split.append({"input": question, "answer": answer})
        case _:
            raise ValueError(f"Unknown dataset name: {anymaths_dset_name}")

    trainset = train_split[: len(train_split) // 2]
    valset = train_split[len(train_split) // 2 :]
    testset = test_split

    return trainset, valset, testset


if __name__ == "__main__":
    import argparse
    from functools import partial
    from pathlib import Path

    import litellm

    from gepa import optimize
    from gepa.adapters.anymaths_adapter import AnyMathsAdapter

    parser = argparse.ArgumentParser()
    parser.add_argument("--anymaths_dset_name", type=str, default="openai/gsm8k")
    parser.add_argument("--train_size", type=int, default=1, help="The size of the
training set to use.")
    parser.add_argument("--val_size", type=int, default=1, help="The size of the
validation set to use.")
    parser.add_argument("--test_size", type=int, default=1, help="The size of the test set
to use.")
    parser.add_argument("--base_lm", type=str, default="ollama/qwen3:4b")
    parser.add_argument("--use_api_base", action="store_true", help="Use API base URL")
    parser.add_argument("--api_base_url", type=str, default="http://localhost:11434")
    parser.add_argument(
```

```
        "--reflection_lm", type=str, default="ollama/qwen3:8b", help="The name of the
reflection LM to use."
    )
    parser.add_argument("--use_api_reflection", action="store_true", help="Use API
reflection URL")
    parser.add_argument(
        "--api_reflection_url",
        type=str,
        default="http://localhost:11434",
        help="The API base URL for the reflection LM.",
    )
    parser.add_argument(
        "--reflection_minibatch_size", type=int, default=8, help="The size of the
minibatch for the reflection LM."
    )
    parser.add_argument("--max_litellm_workers", type=int, default=10)
    parser.add_argument("--budget", type=int, default=500, help="The budget for the
optimization process.")
    parser.add_argument(
        "--seed", type=int, default=0, help="The seed for the random number generator for
reproducibility."
    )
    args = parser.parse_args()

    INSTRUCTION_PROMPT_PATH = Path(__file__).parent / "prompt-
templates/instruction_prompt.txt"

    seed_instruction = INSTRUCTION_PROMPT_PATH.read_text()

    trainset, valset, testset = init_dataset(args.anymaths_dset_name)

    train_size = args.train_size
    val_size = args.val_size
    test_size = args.test_size

    for size in map(int, [train_size, val_size, test_size]):
        if size <= 0:
            raise ValueError("Train, val, and test sizes must be positive integers.")

    trainset = trainset[:train_size]
    valset = valset[:val_size]
    testset = testset[:test_size]

    print("-" * 100)
    print(f"Using dataset: {args.anymaths_dset_name}")
    print(f"Training set size: {len(trainset)}")
    print(f"Validation set size: {len(valset)}")
    print(f"Test set size: {len(testset)}")
    print("-" * 100)

    base_lm = args.base_lm

    reflection_lm_name = args.reflection_lm

    _reflection = {"model": reflection_lm_name}

    use_api_base = args.use_api_base
    use_api_reflection = args.use_api_reflection

    if use_api_base:
        api_base = args.api_base_url
    else:
        api_base = None

    if use_api_reflection:
        api_reflection = args.api_reflection_url
        _reflection["base_url"] = api_reflection
    else:
```

```
        api_reflection = None

    _reflection_completion = partial(litellm.completion, **_reflection)

    def reflection_lm(prompt: str):
        """Call the reflection language model with the given prompt and return its content
string."""
        response = _reflection_completion(messages=[{"role": "user", "content": prompt}])
        return response.choices[0].message.content

    max_litellm_workers = args.max_litellm_workers
    budget = args.budget
    reflection_minibatch_size = args.reflection_minibatch_size
    seed = args.seed

    print(f"Using base LM: {base_lm}")
    print(f"Using reflection LM: {reflection_lm_name}")
    print(f"Using API base URL: {api_base}")
    print(f"Using API reflection URL: {api_reflection}")
    print(f"Reflection minibatch size: {reflection_minibatch_size}")
    print(f"Max LiteLLM workers: {max_litellm_workers}")
    print(f"Budget: {budget}")
    print(f"Seed: {seed}")
    print("-" * 100)

    optimized_results = optimize(
        seed_candidate={"instruction_prompt": seed_instruction},
        trainset=trainset,
        valset=valset,
        adapter=AnyMathsAdapter(model=base_lm, api_base=api_base,
max_litellm_workers=max_litellm_workers),
        reflection_lm=reflection_lm,
        reflection_minibatch_size=reflection_minibatch_size,
        perfect_score=1,
        skip_perfect_score=False,
        use_wandb=False,
        max_metric_calls=budget,
        seed=seed,
        display_progress_bar=True,
    )

    print("-" * 100)
    print(f"Best prompt >>> {optimized_results.best_candidate}")
    print("-" * 100)
```

### src/gepa/examples/rag_adapter/RAG_GUIDE.md

```
# GEPA Generic RAG Adapter Guide

This guide demonstrates how to use GEPA's Generic RAG Adapter with the new **unified
`rag_optimization.py`** script that supports multiple vector stores. This consolidated
approach makes it easy to test and compare different vector databases with a single
command.

## 🆕 Unified Script: `rag_optimization.py`

**One script, multiple vector stores!** We've consolidated all individual optimization
scripts into a single, powerful `rag_optimization.py` that supports all vector stores
through command-line arguments.

### 📁 Supported Vector Stores

| Vector Store | Docker Required | Key Features | Usage |
```

```
|--------------|----------------|--------------|--------|
| **ChromaDB** (default) | ❌ No | Local storage, simple setup, semantic search | `--vector-store chromadb` |
| **LanceDB** | ❌ No | Serverless, columnar format, developer-friendly | `--vector-store lancedb` |
| **Milvus** | ❌ No* | Cloud-native, scalable, Milvus Lite for local dev | `--vector-store milvus` |
| **Qdrant** | ❌ No* | Advanced filtering, payload search, high performance | `--vector-store qdrant` |
| **Weaviate** | ✅ Yes | Hybrid search, production-ready, advanced features | `--vector-store weaviate` |

*Docker optional for production deployments

### ✨ Benefits of the Unified Approach

- **🔄 Easy Switching**: Test different vector stores with just a flag change
- **🔧 Consistent Interface**: Same commands work across all databases
- **📊 Fair Comparison**: Identical test conditions for comparing performance
- **🛠️ Less Maintenance**: Single file to maintain instead of 5 separate scripts

## 🚀 Quick Start Guide

### Prerequisites

1. **Install Dependencies:**

   **Install GEPA Core:**
   ```bash
   pip install gepa
   ```

   **Install RAG Adapter Dependencies:**

   You can either install all vector store dependencies or specific ones:

   ```bash
   # Option A: Install all dependencies (recommended for exploration)
   pip install -r requirements-rag.txt

   # Option B: Install specific vector store dependencies
   # ChromaDB (easiest to start with)
   pip install litellm chromadb

   # LanceDB (serverless, no Docker needed)
   pip install litellm lancedb pyarrow

   # Milvus (local Lite mode)
   pip install litellm pymilvus

   # Qdrant (in-memory mode)
   pip install litellm qdrant-client

   # Weaviate (requires Docker)
   pip install litellm weaviate-client
   ```

   **Note:** Vector store dependencies are now separate from the core GEPA package and must be installed manually based on which vector stores you want to use. For specific version requirements, see `requirements-rag.txt`.

2. **For Local Models (Ollama):**
   ```bash
   # Install Ollama
```
```

```
   curl -fsSL https://ollama.com/install.sh | sh

   # Pull models used in examples
   ollama pull qwen3:8b
   ollama pull llama3.1:8b
   ollama pull nomic-embed-text:latest
   ```

3. **For Cloud Models:**
   ```bash
   export OPENAI_API_KEY="your-api-key"
   export ANTHROPIC_API_KEY="your-api-key"
   ```

4. **Docker Requirements:**

   | Database | Docker Required | Notes |
   |----------|----------------|-------|
   | **ChromaDB** | ❌ No | Runs locally, no external services |
   | **LanceDB** | ❌ No | Serverless, creates local files |
   | **Milvus** | ❌ No (default) | Uses Milvus Lite (local SQLite) |
   | **Qdrant** | ❌ No (default) | Uses in-memory mode by default |
   | **Weaviate** | ✅ Yes | Requires Docker or cloud instance |

   **Docker Setup (only for Weaviate):**
   ```bash
   # Start Weaviate with Docker
   docker run -p 8080:8080 -p 50051:50051 cr.weaviate.io/semitechnologies/weaviate:1.26.1
   ```

   **Optional Docker Setup:**
   ```bash
   # For production Milvus (optional)
   docker run -d -p 19530:19530 milvusdb/milvus:latest standalone

   # For production Qdrant (optional)
   docker run -p 6333:6333 qdrant/qdrant
   ```

## 🚀 Using the Unified RAG Optimization Script

### Basic Usage

```bash
# Navigate to the examples directory
cd src/gepa/examples/rag_adapter

# 🔵 ChromaDB (Default - No Docker Required)
python rag_optimization.py --vector-store chromadb

# 🟢 LanceDB (Serverless - No Docker Required)
python rag_optimization.py --vector-store lancedb

# 🔵 Milvus (Local Lite Mode - No Docker Required)
python rag_optimization.py --vector-store milvus

# 🟡 Qdrant (In-Memory - No Docker Required)
python rag_optimization.py --vector-store qdrant

# 🟠 Weaviate (Requires Docker)
python rag_optimization.py --vector-store weaviate
```

### Quick Test (No Optimization)

```bash
# Test setup without running full optimization
python rag_optimization.py --vector-store chromadb --max-iterations 0
python rag_optimization.py --vector-store lancedb --max-iterations 0
python rag_optimization.py --vector-store qdrant --max-iterations 0
```

### Full Optimization Runs

```bash
# ChromaDB with 10 iterations
python rag_optimization.py --vector-store chromadb --max-iterations 10

# LanceDB with 20 iterations
python rag_optimization.py --vector-store lancedb --max-iterations 20

# Qdrant with 15 iterations
python rag_optimization.py --vector-store qdrant --max-iterations 15
```

### Different Models

```bash
# Use different Ollama models
python rag_optimization.py --vector-store chromadb --model ollama/llama3.1:8b

# Use cloud models (requires API key)
python rag_optimization.py --vector-store lancedb --model gpt-4o-mini --max-iterations 10

# Use Anthropic models
python rag_optimization.py --vector-store qdrant --model claude-3-haiku-20240307
```

## 📋 Vector Store Specific Instructions

### 🔵 ChromaDB (Default & Easiest)

ChromaDB is perfect for getting started - lightweight, runs locally, no external services needed.

**✅ No Docker Required**

```bash
# Basic usage (default vector store)
python rag_optimization.py --vector-store chromadb

# Or simply (chromadb is the default)
python rag_optimization.py

# Quick test
python rag_optimization.py --max-iterations 0

# Full optimization run
python rag_optimization.py --max-iterations 20
```

**Key Features:**
- Local persistent storage
- Simple setup with no configuration
- Semantic similarity search
- Built-in embedding functions

### 🟢 LanceDB (Serverless & Developer-Friendly)

````
LanceDB is a serverless vector database built on the Lance columnar format, perfect for
local development.

**✅ No Docker Required**

```bash
# Basic usage
python rag_optimization.py --vector-store lancedb

# With different models
python rag_optimization.py --vector-store lancedb --model ollama/qwen3:8b

# Full optimization run
python rag_optimization.py --vector-store lancedb --max-iterations 20
```

**Key Features:**
- Serverless architecture (no external services)
- Built on Apache Arrow/Lance for performance
- Creates local database files (./lancedb_demo)
- Developer-friendly with simple setup


### 🔵 Milvus (Cloud-Native & Scalable)

Milvus is a cloud-native vector database designed for large-scale AI applications. Uses
Milvus Lite for local development.

**✅ No Docker Required (uses Milvus Lite locally)**

```bash
# Basic usage (uses local SQLite-based Milvus Lite)
python rag_optimization.py --vector-store milvus

# With different models
python rag_optimization.py --vector-store milvus --model gpt-4o-mini --max-iterations 10

# Full optimization run
python rag_optimization.py --vector-store milvus --max-iterations 15
```

**Key Features:**
- Milvus Lite (local SQLite, no Docker needed)
- Creates local ./milvus_demo.db file automatically
- Cloud-native design for production scaling
- Advanced indexing and search capabilities


### 🟡 Qdrant (High-Performance & Advanced Filtering)

Qdrant is a high-performance vector database with advanced filtering and payload search
capabilities.

**✅ No Docker Required (uses in-memory mode by default)**

```bash
# Basic usage (in-memory mode)
python rag_optimization.py --vector-store qdrant

# With different models
python rag_optimization.py --vector-store qdrant --model gpt-4o-mini --max-iterations 10

# Full optimization run
python rag_optimization.py --vector-store qdrant --max-iterations 15
```
````

```
**Key Features:**
- In-memory mode (no external services needed)
- Advanced metadata filtering capabilities
- Payload search (vector + metadata combined)
- High-performance optimized for speed and scale
- Optional persistent storage or remote server


### 🟠 Weaviate (Hybrid Search)

Weaviate offers advanced features like hybrid search (semantic + keyword) and is
production-ready.

**⚠️ Docker Required**

```bash
# Setup: Start Weaviate with Docker (required)
docker run -p 8080:8080 -p 50051:50051 cr.weaviate.io/semitechnologies/weaviate:1.26.1

# Verify Weaviate is running
curl http://localhost:8080/v1/meta

# Basic usage (requires Docker setup above)
python rag_optimization.py --vector-store weaviate

# With cloud models
python rag_optimization.py --vector-store weaviate --model gpt-4o-mini --max-iterations 10

# Full optimization run
python rag_optimization.py --vector-store weaviate --max-iterations 15
```

**Key Features:**
- Hybrid search (semantic + keyword/BM25 combined)
- Production-ready with clustering support
- Rich GraphQL and RESTful APIs
- Advanced schema management
- Built-in vectorization modules


## ⚙️ Configuration Options

### Command Line Arguments

The unified `rag_optimization.py` script supports these arguments:

| Argument | Default | Description |
|----------|---------|-------------|
| `--vector-store` | `chromadb` | Choose vector store: `chromadb`, `lancedb`, `milvus`,
`qdrant`, `weaviate` |
| `--model` | `ollama/qwen3:8b` | LLM model to use for generation |
| `--embedding-model` | `ollama/nomic-embed-text:latest` | Embedding model for vector
search |
| `--max-iterations` | `5` | GEPA optimization iterations (use 0 to skip optimization) |
| `--verbose` | `False` | Enable detailed logging and debugging |

### Complete Help

```bash
# See all available options
python rag_optimization.py --help
```

### Vector Store Selection

```bash
# Available vector stores (choose one)
```
```

```
--vector-store chromadb   # Default: Local, no Docker
--vector-store lancedb    # Serverless, no Docker
--vector-store milvus     # Local Lite mode, no Docker
--vector-store qdrant     # In-memory mode, no Docker
--vector-store weaviate   # Requires Docker
```

### Model Recommendations

| Model | Size | Speed | Quality | Use Case |
|---|---|---|---|---|
| `ollama/qwen3:8b` | Large | Medium | Excellent | Default for ChromaDB/Weaviate/Qdrant |
| `ollama/llama3.1:8b` | Large | Medium | Excellent | Default for LanceDB/Milvus |
| `gpt-4o-mini` | Cloud | Fast | Excellent | Production (cloud) |
| `claude-3-haiku-20240307` | Cloud | Fast | Excellent | Production (cloud) |

### Embedding Models

| Model | Provider | Use Case |
|---|---|---|
| `ollama/nomic-embed-text:latest` | Local | Offline, privacy |
| `text-embedding-3-small` | OpenAI | Fast, cost-effective |
| `text-embedding-3-large` | OpenAI | High quality |

## 🧪 Testing Your Setup

### Quick Health Check

```bash
# Test all vector stores (no optimization, just setup verification)
python rag_optimization.py --vector-store chromadb --max-iterations 0
python rag_optimization.py --vector-store lancedb --max-iterations 0
python rag_optimization.py --vector-store milvus --max-iterations 0
python rag_optimization.py --vector-store qdrant --max-iterations 0
python rag_optimization.py --vector-store weaviate --max-iterations 0  # Requires Docker

# Test external services (if using)
curl http://localhost:8080/v1/meta  # Weaviate (if using Docker)
ollama list                         # Check available Ollama models
```

### Compare Vector Stores

```bash
# Run same optimization across all vector stores for comparison
python rag_optimization.py --vector-store chromadb --max-iterations 5
python rag_optimization.py --vector-store lancedb --max-iterations 5
python rag_optimization.py --vector-store milvus --max-iterations 5
python rag_optimization.py --vector-store qdrant --max-iterations 5
```

## 🔧 Troubleshooting

### Common Issues

#### Import Errors
```bash
# Make sure you're in the right directory
cd /path/to/gepa/src/gepa/examples/rag_adapter
python rag_optimization.py --vector-store chromadb

# If you get import errors, install missing dependencies using requirements-rag.txt
pip install -r requirements-rag.txt

# Or install specific vector store dependencies:
pip install litellm chromadb                    # For ChromaDB
pip install litellm lancedb pyarrow             # For LanceDB
```

```
pip install litellm pymilvus                    # For Milvus
pip install litellm qdrant-client               # For Qdrant
pip install litellm weaviate-client             # For Weaviate
```

#### Ollama Issues
```bash
# Check Ollama is running
ollama list

# Pull required models
ollama pull qwen3:8b
ollama pull llama3.1:8b
ollama pull nomic-embed-text:latest

# Test models
ollama run qwen3:8b "Hello"
ollama run llama3.1:8b "Hello"
```

#### Weaviate Issues
```bash
# Check Weaviate is accessible
curl http://localhost:8080/v1/meta

# Start Weaviate with Docker
docker run -d -p 8080:8080 -p 50051:50051 cr.weaviate.io/semitechnologies/weaviate:1.26.1

# Check Docker container
docker ps
```

#### LanceDB Issues
```bash
# Check LanceDB installation
python -c "import lancedb; print('LanceDB installed')"

# Check PyArrow installation
python -c "import pyarrow; print('PyArrow installed')"

# Install missing dependencies
pip install litellm lancedb pyarrow

# Check sentence-transformers for embeddings
python -c "import sentence_transformers; print('sentence-transformers installed')"
pip install sentence-transformers
```

#### Milvus Issues
```bash
# Check Milvus Lite installation
python -c "import pymilvus; print('PyMilvus installed')"

# Install missing dependencies
pip install litellm pymilvus

# Check if milvus_demo.db file exists
ls -la milvus_demo.db

# For full Milvus server issues
docker run -d -p 19530:19530 milvusdb/milvus:latest standalone
curl http://localhost:19530/health
```

#### Qdrant Issues
```bash
# Check Qdrant client installation
python -c "import qdrant_client; print('Qdrant client installed')"
```

```
# Install missing dependencies
pip install litellm qdrant-client

# Test Qdrant server connection
curl http://localhost:6333/health

# Start Qdrant with Docker
docker run -p 6333:6333 qdrant/qdrant

# Check Qdrant container
docker ps | grep qdrant
```

#### Memory Issues
```bash
# Use cloud model instead of local
python rag_optimization.py --vector-store chromadb --model gpt-4o-mini

# Reduce iterations
python rag_optimization.py --vector-store chromadb --max-iterations 2

# Test without optimization first
python rag_optimization.py --vector-store chromadb --max-iterations 0
```

#### Vector Store Specific Issues

```bash
# ChromaDB - No common issues, very stable

# LanceDB - Check PyArrow installation
python -c "import pyarrow; print('PyArrow OK')"
pip install litellm lancedb pyarrow

# Milvus - Check PyMilvus installation
python -c "import pymilvus; print('PyMilvus OK')"
pip install litellm pymilvus

# Qdrant - Check client installation
python -c "import qdrant_client; print('Qdrant client OK')"
pip install litellm qdrant-client

# Weaviate - Ensure Docker is running
curl http://localhost:8080/v1/meta
docker run -p 8080:8080 -p 50051:50051 cr.weaviate.io/semitechnologies/weaviate:1.26.1
```

### Getting Help

If you encounter issues:

1. **Check Prerequisites**: Ensure all dependencies are installed
2. **Start Simple**: Use `--max-iterations 0` to test setup without optimization
3. **Use Cloud Models**: Try `gpt-4o-mini` for faster testing with less memory
4. **Enable Verbose Mode**: Add `--verbose` for detailed error information
5. **Check Resources**: Ensure sufficient memory and disk space

## 📈 Understanding Results

### Evaluation Metrics

- **Retrieval Quality**: How well relevant documents are retrieved
- **Generation Quality**: How accurate and helpful the generated answers are
- **Combined Score**: Weighted combination optimized by GEPA (higher is better)

### Optimization Process

````
GEPA uses evolutionary search to improve prompts:

1. **Baseline**: Test initial prompts
2. **Mutation**: Generate variations of prompts
3. **Selection**: Keep best performing versions
4. **Iteration**: Repeat until convergence or max iterations

### Expected Improvements

Typical score improvements with GEPA:
- **Initial Score**: 0.3-0.5 (basic prompts)
- **After Optimization**: 0.6-0.8 (optimized prompts)
- **Improvement Range**: +0.1 to +0.4 points

## 🎯 Next Steps

1. **Scale Up**: Use larger models and more iterations for production
2. **Custom Data**: Replace example data with your domain-specific knowledge
3. **Advanced Features**: Explore metadata filtering and custom prompts
4. **Production Setup**: Configure persistent storage and monitoring
5. **Integration**: Incorporate optimized prompts into your applications

---

## 📊 Real Optimization Results

### 🔬 ChromaDB + GEPA Optimization Example

**Configuration:**
- **Vector Database**: ChromaDB (Local, No Docker)
- **LLM Model**: Ollama Qwen3:8b (Local)
- **Embedding Model**: Ollama nomic-embed-text:latest
- **Max Iterations**: 10
- **Knowledge Base**: 6 AI/ML articles
- **Training Examples**: 3
- **Validation Examples**: 2
- **Search Strategy**: Semantic similarity search

**Performance Results:**

| Metric | Initial Score | Final Score | Improvement | Total Iterations |
|--------|---------------|-------------|-------------|------------------|
| **Validation Score** | 0.388 | 0.388 | **+0.014** | 14 iterations |
| **Training Score** | 0.374 | - | **+3.7%** | - |

**Setup Commands:**
```bash
# No Docker required for ChromaDB!
# Run optimization directly with unified script
PYTHONPATH=src python src/gepa/examples/rag_adapter/rag_optimization.py \
    --vector-store chromadb \
    --max-iterations 10 \
    --model ollama/qwen3:8b \
    --verbose
```

**Key Observations:**
- ✅ **Successful improvement**: +0.014 score increase (+3.7% improvement)
- ChromaDB's simple setup makes it ideal for quick optimization experiments
- Local Ollama models integrated seamlessly with GEPA
- Semantic similarity search provided good retrieval quality
- GEPA's evolutionary optimization found better prompt variants

**Sample Output Evolution:**

*Initial Answer (0.374 score):*
````

```
### Answer:
Computer vision is a field of artificial intelligence (AI) focused on enabling computers
to interpret and understand the visual world. It leverages digital images and videos as
input and employs deep learning models—a subset of machine learning—to analyze and
classify visual data...
```

*Optimized Answer (0.388 score):*
```
**Answer:**
Computer vision is a field of artificial intelligence (AI) focused on enabling computers
to interpret and understand the visual world. It leverages **digital images and videos**
as input and employs **deep learning models**—a subset of machine learning—to analyze and
classify visual data. These models, inspired by biological neural networks, excel at
processing **unstructured or unlabeled data**...
```

**Improvement Analysis:**
- Better formatting with bold headings and key terms
- More structured presentation of technical concepts
- Enhanced readability through strategic emphasis
- Maintained technical accuracy while improving clarity

---

## src/gepa/examples/rag_adapter/__init__.py

```
# Copyright (c) 2025 Lakshya A Agrawal and the GEPA contributors
# https://github.com/gepa-ai/gepa

"""
GEPA Generic RAG Adapter Examples

This package contains examples demonstrating how to use GEPA's Generic RAG Adapter
with different vector stores and both local and cloud-based language models.
"""
```

## src/gepa/examples/rag_adapter/rag_optimization.py

```
#!/usr/bin/env python3
"""
GEPA RAG Optimization Example with Multiple Vector Stores

This example demonstrates how to use GEPA to optimize a RAG system using various
vector stores, showcasing their unique capabilities and search methods.

Supported Vector Stores:
- ChromaDB: Local/persistent vector store with simple setup
- LanceDB: Developer-friendly serverless vector database
- Milvus: Cloud-native vector database with Lite mode
- Qdrant: High-performance vector database with advanced filtering
- Weaviate: Vector database with hybrid search capabilities

Usage:
    # ChromaDB (default, no external dependencies)
    python rag_optimization.py --vector-store chromadb

    # LanceDB (local, no Docker required)
```

```
    python rag_optimization.py --vector-store lancedb

    # Milvus Lite (local SQLite-based)
    python rag_optimization.py --vector-store milvus

    # Qdrant (in-memory or with Docker)
    python rag_optimization.py --vector-store qdrant

    # Weaviate (requires Docker)
    python rag_optimization.py --vector-store weaviate

    # With specific models
    python rag_optimization.py --vector-store chromadb --model ollama/llama3.1:8b

    # Full optimization run
    python rag_optimization.py --vector-store qdrant --max-iterations 20

Requirements:
    Base: pip install gepa[rag]
    ChromaDB: pip install chromadb
    LanceDB: pip install lancedb pyarrow sentence-transformers
    Milvus: pip install pymilvus sentence-transformers
    Qdrant: pip install qdrant-client
    Weaviate: pip install weaviate-client

Prerequisites:
    - For Ollama: ollama pull qwen3:8b && ollama pull nomic-embed-text:latest
    - For Weaviate: docker run -p 8080:8080 -p 50051:50051
cr.weaviate.io/semitechnologies/weaviate:1.26.1
    - For Qdrant (optional): docker run -p 6333:6333 qdrant/qdrant
"""

import argparse
import os
import sys
import tempfile
import warnings
from pathlib import Path
from typing import Any

# Suppress all warnings for clean output
warnings.filterwarnings("ignore")
os.environ["PYTHONWARNINGS"] = "ignore"

# Add parent directory to path for imports
sys.path.insert(0, str(Path(__file__).parent.parent.parent.parent))

import gepa  # noqa: E402
from gepa.adapters.generic_rag_adapter import GenericRAGAdapter, RAGDataInst  # noqa: E402

# Vector store imports (lazy loaded)
_vector_stores = {}


def lazy_import_vector_store(store_name: str):
    """Lazy import vector store classes to avoid dependency issues."""
    global _vector_stores

    if store_name in _vector_stores:
        return _vector_stores[store_name]

    try:
        if store_name == "chromadb":
            from gepa.adapters.generic_rag_adapter import ChromaVectorStore

            _vector_stores[store_name] = ChromaVectorStore
            return ChromaVectorStore
        elif store_name == "lancedb":
```

```
            from gepa.adapters.generic_rag_adapter import LanceDBVectorStore

            _vector_stores[store_name] = LanceDBVectorStore
            return LanceDBVectorStore
        elif store_name == "milvus":
            from gepa.adapters.generic_rag_adapter import MilvusVectorStore

            _vector_stores[store_name] = MilvusVectorStore
            return MilvusVectorStore
        elif store_name == "qdrant":
            from gepa.adapters.generic_rag_adapter import QdrantVectorStore

            _vector_stores[store_name] = QdrantVectorStore
            return QdrantVectorStore
        elif store_name == "weaviate":
            from gepa.adapters.generic_rag_adapter import WeaviateVectorStore

            _vector_stores[store_name] = WeaviateVectorStore
            return WeaviateVectorStore
        else:
            raise ValueError(f"Unknown vector store: {store_name}")
    except ImportError as e:
        raise ImportError(
            f"Failed to import {store_name} dependencies: {e}\n"
            f"Install with: pip install {get_install_command(store_name)}"
        )


def get_install_command(store_name: str) -> str:
    """Get pip install command for vector store dependencies."""
    commands = {
        "chromadb": "chromadb",
        "lancedb": "lancedb pyarrow sentence-transformers",
        "milvus": "pymilvus sentence-transformers",
        "qdrant": "qdrant-client",
        "weaviate": "weaviate-client",
    }
    return commands.get(store_name, "unknown")


def create_llm_client(model_name: str):
    """Create LLM client supporting both Ollama and cloud models."""
    try:
        import litellm

        litellm.drop_params = True
        litellm.set_verbose = False
    except ImportError:
        raise ImportError("LiteLLM is required. Install with: pip install litellm")

    def llm_client(messages_or_prompt, **kwargs):
        try:
            # Handle both string prompts and message lists
            if isinstance(messages_or_prompt, str):
                messages = [{"role": "user", "content": messages_or_prompt}]
            else:
                messages = messages_or_prompt

            params = {
                "model": model_name,
                "messages": messages,
                "max_tokens": kwargs.get("max_tokens", 400),
                "temperature": kwargs.get("temperature", 0.1),
            }

            if "ollama/" in model_name:
                params["request_timeout"] = 120
```

```
            response = litellm.completion(**params)
            return response.choices[0].message.content.strip()

        except Exception as e:
            return f"Error: Unable to generate response ({e})"

    return llm_client


def create_embedding_function():
    """Create embedding function using sentence-transformers as fallback."""
    try:
        from sentence_transformers import SentenceTransformer

        model = SentenceTransformer("all-MiniLM-L6-v2")
        return lambda text: model.encode(text)
    except ImportError:
        # Fallback to litellm for embedding
        try:
            import litellm

            def embed_text(text: str):
                try:
                    response = litellm.embedding(model="ollama/nomic-embed-text:latest",
input=text)
                    if hasattr(response, "data") and response.data:
                        if hasattr(response.data[0], "embedding"):
                            return response.data[0].embedding
                        elif isinstance(response.data[0], dict) and "embedding" in
response.data[0]:
                            return response.data[0]["embedding"]
                    elif isinstance(response, dict):
                        if response.get("data"):
                            return response["data"][0]["embedding"]
                        elif "embedding" in response:
                            return response["embedding"]
                    raise ValueError(f"Unknown response format: {type(response)}")
                except Exception as e:
                    raise RuntimeError(
                        f"Embedding failed: {e}. Please check your embedding model setup
(sentence-transformers or litellm) and ensure all dependencies are installed."
                    )

            return embed_text
        except ImportError:
            raise ImportError("Either sentence-transformers or litellm is required for
embeddings")


def setup_chromadb_store():
    """Set up ChromaDB vector store with sample data."""
    print("🗄️ Setting up ChromaDB vector store...")

    try:
        from chromadb.utils import embedding_functions
    except ImportError:
        raise ImportError("ChromaDB is required. Install with: pip install chromadb")

    # Create temporary directory
    temp_dir = tempfile.mkdtemp()
    print(f"   📁 ChromaDB directory: {temp_dir}")

    # Initialize ChromaDB
    embedding_function = embedding_functions.DefaultEmbeddingFunction()
    chroma_vector_store = lazy_import_vector_store("chromadb")
    vector_store = chroma_vector_store.create_local(
        persist_directory=temp_dir, collection_name="ai_ml_knowledge",
```

```
embedding_function=embedding_function
    )

    documents = get_sample_documents()
    vector_store.collection.add(
        documents=[doc["content"] for doc in documents],
        metadatas=[doc["metadata"] for doc in documents],
        ids=[doc["metadata"]["doc_id"] for doc in documents],
    )

    print(f"   ✅ Created ChromaDB knowledge base with {len(documents)} articles")
    return vector_store


def setup_lancedb_store():
    """Set up LanceDB vector store with sample data."""
    print("🗄️ Setting up LanceDB vector store...")

    try:
        embedding_function = create_embedding_function()
        lancedb_vector_store = lazy_import_vector_store("lancedb")

        vector_store = lancedb_vector_store.create_local(
            table_name="rag_demo",
            embedding_function=embedding_function,
            db_path="./lancedb_demo",
            vector_size=384,
        )

        documents = get_sample_documents_simple()
        embeddings = [embedding_function(doc["content"]) for doc in documents]
        ids = vector_store.add_documents(documents, embeddings)

        print(f"   ✅ Added {len(ids)} documents to LanceDB table")
        return vector_store

    except ImportError as e:
        raise ImportError(
            f"LanceDB dependencies missing: {e}\nInstall with: pip install lancedb pyarrow
sentence-transformers"
        )


def setup_milvus_store():
    """Set up Milvus vector store with sample data."""
    print("🗄️ Setting up Milvus vector store...")

    try:
        embedding_function = create_embedding_function()
        milvus_vector_store = lazy_import_vector_store("milvus")

        vector_store = milvus_vector_store.create_local(
            collection_name="rag_demo",
            embedding_function=embedding_function,
            vector_size=384,
            uri="./milvus_demo.db",
        )

        documents = get_sample_documents_simple()
        embeddings = [embedding_function(doc["content"]) for doc in documents]
        ids = vector_store.add_documents(documents, embeddings)

        print(f"   ✅ Added {len(ids)} documents to Milvus collection")
        return vector_store

    except ImportError as e:
```

```
        raise ImportError(f"Milvus dependencies missing: {e}\nInstall with: pip install
pymilvus sentence-transformers")


def setup_qdrant_store():
    """Set up Qdrant vector store with sample data."""
    print("🗄️ Setting up Qdrant vector store...")

    try:
        from qdrant_client import QdrantClient
        from qdrant_client.http import models
    except ImportError:
        raise ImportError("Qdrant client required. Install with: pip install qdrant-
client")

    # Connect to in-memory Qdrant
    client = QdrantClient(path=":memory:")
    print("   ✅ Connected to in-memory Qdrant")

    collection_name = "AIKnowledge"

    # Delete existing collection if it exists
    try:
        client.delete_collection(collection_name)
    except Exception:
        pass

    # Create embedding function and determine vector size
    embedding_fn = create_embedding_function()
    sample_vector = embedding_fn("test")
    vector_size = len(sample_vector)

    client.create_collection(
        collection_name=collection_name,
        vectors_config=models.VectorParams(
            size=vector_size,
            distance=models.Distance.COSINE,
        ),
    )

    # Add documents
    documents = get_sample_documents_for_qdrant()

    points = []
    for i, doc in enumerate(documents):
        doc_vector = embedding_fn(doc["content"])
        payload = dict(doc)
        payload["original_id"] = f"doc_{i + 1}"

        point = models.PointStruct(
            id=i + 1,
            vector=doc_vector,
            payload=payload,
        )
        points.append(point)

    client.upsert(collection_name=collection_name, points=points, wait=True)
    print(f"   ✅ Created Qdrant knowledge base with {len(documents)} articles")

    qdrant_vector_store = lazy_import_vector_store("qdrant")
    vector_store = qdrant_vector_store(client, collection_name, embedding_fn)
    return vector_store


def setup_weaviate_store():
    """Set up Weaviate vector store with sample data."""
    print("🗄️ Setting up Weaviate vector store...")
```

```
    try:
        import weaviate
        import weaviate.classes as wvc
    except ImportError:
        raise ImportError("Weaviate client required. Install with: pip install weaviate-
client")

    # Connect to local Weaviate
    try:
        client = weaviate.connect_to_local()
        print("   ✅ Connected to local Weaviate")
    except Exception as e:
        print(f"   ❌ Failed to connect to Weaviate: {e}")
        print("   💡 Make sure Weaviate is running:")
        print("      docker run -p 8080:8080 -p 50051:50051
cr.weaviate.io/semitechnologies/weaviate:1.26.1")
        raise

    collection_name = "AIKnowledge"

    # Delete existing collection if it exists
    try:
        client.collections.delete(collection_name)
        print(f"   🗑️ Removed existing collection: {collection_name}")
    except Exception:
        pass

    # Create collection
    collection = client.collections.create(
        name=collection_name,
        properties=[
            wvc.config.Property(name="content", data_type=wvc.config.DataType.TEXT,
description="Document content"),
            wvc.config.Property(name="topic", data_type=wvc.config.DataType.TEXT,
description="Topic category"),
            wvc.config.Property(name="difficulty", data_type=wvc.config.DataType.TEXT,
description="Difficulty level"),
        ],
        vectorizer_config=wvc.config.Configure.Vectorizer.none(),
        inverted_index_config=wvc.config.Configure.inverted_index(
            bm25_b=0.75,
            bm25_k1=1.2,
        ),
    )

    # Create embedding function and add documents
    embedding_fn = create_embedding_function()
    documents = get_sample_documents_for_weaviate()

    with collection.batch.dynamic() as batch:
        for doc in documents:
            doc_vector = embedding_fn(doc["content"])
            batch.add_object(properties=doc, vector=doc_vector)

    client.close()
    print(f"   ✅ Created Weaviate knowledge base with {len(documents)} articles")

    # Reconnect and create vector store wrapper
    client_for_store = weaviate.connect_to_local()
    weaviate_vector_store = lazy_import_vector_store("weaviate")
    vector_store = weaviate_vector_store(client_for_store, collection_name, embedding_fn)
    return vector_store


def get_sample_documents() -> list[dict[str, Any]]:
```

```
    """Get sample documents for ChromaDB (with nested metadata structure)."""
    return [
        {
            "content": "Machine Learning is a subset of artificial intelligence that
enables computers to learn and improve from experience without being explicitly
programmed. It focuses on the development of computer programs that can access data and
use it to learn for themselves.",
            "metadata": {"doc_id": "ml_basics", "topic": "machine_learning", "difficulty":
"beginner"},
        },
        {
            "content": "Deep Learning is a subset of machine learning based on artificial
neural networks with representation learning. It can learn from data that is unstructured
or unlabeled. Deep learning models are inspired by information processing patterns found
in biological neural networks.",
            "metadata": {"doc_id": "dl_basics", "topic": "deep_learning", "difficulty":
"intermediate"},
        },
        {
            "content": "Natural Language Processing (NLP) is a branch of artificial
intelligence that helps computers understand, interpret and manipulate human language. NLP
draws from many disciplines, including computer science and computational linguistics.",
            "metadata": {"doc_id": "nlp_basics", "topic": "nlp", "difficulty":
"intermediate"},
        },
        {
            "content": "Computer Vision is a field of artificial intelligence that trains
computers to interpret and understand the visual world. Using digital images from cameras
and videos and deep learning models, machines can accurately identify and classify
objects.",
            "metadata": {"doc_id": "cv_basics", "topic": "computer_vision", "difficulty":
"intermediate"},
        },
        {
            "content": "Reinforcement Learning is an area of machine learning where an
agent learns to behave in an environment by performing actions and seeing the results. The
agent receives rewards by performing correctly and penalties for performing incorrectly.",
            "metadata": {"doc_id": "rl_basics", "topic": "reinforcement_learning",
"difficulty": "advanced"},
        },
        {
            "content": "Large Language Models (LLMs) are a type of artificial intelligence
model designed to understand and generate human-like text. They are trained on vast
amounts of text data and can perform various natural language tasks such as translation,
summarization, and question answering.",
            "metadata": {"doc_id": "llm_basics", "topic": "large_language_models",
"difficulty": "advanced"},
        },
    ]


def get_sample_documents_simple() -> list[dict[str, str]]:
    """Get sample documents for LanceDB/Milvus (flat structure)."""
    return [
        {"content": "Machine learning is a method of data analysis that automates
analytical model building."},
        {"content": "It is a branch of artificial intelligence based on the idea that
systems can learn from data."},
        {"content": "Machine learning algorithms build a model based on training data to
make predictions."},
        {
            "content": "Deep learning is part of a broader family of machine learning
methods based on artificial neural networks."
        },
        {"content": "It uses multiple layers to progressively extract higher-level
features from raw input."},
        {
            "content": "Deep learning models can automatically learn representations of
```

```
data with multiple levels of abstraction."
        },
        {
            "content": "Natural language processing (NLP) is a subfield of linguistics,
computer science, and artificial intelligence."
        },
        {"content": "It deals with the interaction between computers and human
language."},
        {"content": "NLP techniques enable computers to process and analyze large amounts
of natural language data."},
    ]


def get_sample_documents_for_qdrant() -> list[dict[str, Any]]:
    """Get sample documents for Qdrant (with flat metadata)."""
    return [
        {
            "content": "Artificial Intelligence (AI) is the simulation of human
intelligence in machines that are programmed to think and learn like humans. The term may
also be applied to any machine that exhibits traits associated with a human mind such as
learning and problem-solving.",
            "topic": "artificial_intelligence",
            "difficulty": "beginner",
            "category": "definition",
        },
        {
            "content": "Machine Learning is a method of data analysis that automates
analytical model building. It is a branch of artificial intelligence based on the idea
that systems can learn from data, identify patterns and make decisions with minimal human
intervention.",
            "topic": "machine_learning",
            "difficulty": "beginner",
            "category": "definition",
        },
        {
            "content": "Deep Learning is part of a broader family of machine learning
methods based on artificial neural networks with representation learning. Learning can be
supervised, semi-supervised or unsupervised. Deep learning architectures such as deep
neural networks have been applied to computer vision, speech recognition, and natural
language processing.",
            "topic": "deep_learning",
            "difficulty": "intermediate",
            "category": "technical",
        },
        {
            "content": "Natural Language Processing (NLP) is a subfield of linguistics,
computer science, and artificial intelligence concerned with the interactions between
computers and human language. The goal is to program computers to process and analyze
large amounts of natural language data.",
            "topic": "nlp",
            "difficulty": "intermediate",
            "category": "technical",
        },
        {
            "content": "Computer Vision is a field of artificial intelligence (AI) that
enables computers and systems to derive meaningful information from digital images, videos
and other visual inputs. It uses machine learning models to analyze and interpret visual
data.",
            "topic": "computer_vision",
            "difficulty": "intermediate",
            "category": "application",
        },
        {
            "content": "Transformers are a deep learning architecture that has
revolutionized natural language processing. They rely entirely on self-attention
mechanisms to draw global dependencies between input and output, dispensing with
recurrence and convolutions entirely.",
            "topic": "transformers",
```

```
            "difficulty": "advanced",
            "category": "architecture",
        },
    ]


def get_sample_documents_for_weaviate() -> list[dict[str, str]]:
    """Get sample documents for Weaviate (flat string properties)."""
    return [
        {
            "content": "Artificial Intelligence (AI) is the simulation of human
intelligence in machines that are programmed to think and learn like humans. The term may
also be applied to any machine that exhibits traits associated with a human mind such as
learning and problem-solving.",
            "topic": "artificial_intelligence",
            "difficulty": "beginner",
        },
        {
            "content": "Machine Learning is a method of data analysis that automates
analytical model building. It is a branch of artificial intelligence based on the idea
that systems can learn from data, identify patterns and make decisions with minimal human
intervention.",
            "topic": "machine_learning",
            "difficulty": "beginner",
        },
        {
            "content": "Deep Learning is part of a broader family of machine learning
methods based on artificial neural networks with representation learning. Learning can be
supervised, semi-supervised or unsupervised. Deep learning architectures such as deep
neural networks have been applied to computer vision, speech recognition, and natural
language processing.",
            "topic": "deep_learning",
            "difficulty": "intermediate",
        },
        {
            "content": "Natural Language Processing (NLP) is a subfield of linguistics,
computer science, and artificial intelligence concerned with the interactions between
computers and human language. The goal is to program computers to process and analyze
large amounts of natural language data.",
            "topic": "nlp",
            "difficulty": "intermediate",
        },
        {
            "content": "Computer Vision is an interdisciplinary scientific field that
deals with how computers can gain high-level understanding from digital images or videos.
From an engineering perspective, it seeks to understand and automate tasks that the human
visual system can do.",
            "topic": "computer_vision",
            "difficulty": "intermediate",
        },
        {
            "content": "Transformers are a deep learning architecture that has
revolutionized natural language processing. They rely entirely on self-attention
mechanisms to draw global dependencies between input and output, dispensing with
recurrence and convolutions entirely.",
            "topic": "transformers",
            "difficulty": "advanced",
        },
    ]


def create_training_data() -> tuple[list[RAGDataInst], list[RAGDataInst]]:
    """Create training and validation datasets for RAG optimization."""
    # Training examples
    train_data = [
        RAGDataInst(
            query="What is machine learning?",
            ground_truth_answer="Machine Learning is a method of data analysis that
```

```
automates analytical model building. It is a branch of artificial intelligence based on
the idea that systems can learn from data, identify patterns and make decisions with
minimal human intervention.",
            relevant_doc_ids=["ml_basics"],
            metadata={"category": "definition", "difficulty": "beginner"},
        ),
        RAGDataInst(
            query="How does deep learning work?",
            ground_truth_answer="Deep Learning is a subset of machine learning based on
artificial neural networks with representation learning. It can learn from data that is
unstructured or unlabeled. Deep learning models are inspired by information processing
patterns found in biological neural networks.",
            relevant_doc_ids=["dl_basics"],
            metadata={"category": "explanation", "difficulty": "intermediate"},
        ),
        RAGDataInst(
            query="What is natural language processing?",
            ground_truth_answer="Natural Language Processing (NLP) is a branch of
artificial intelligence that helps computers understand, interpret and manipulate human
language. NLP draws from many disciplines, including computer science and computational
linguistics.",
            relevant_doc_ids=["nlp_basics"],
            metadata={"category": "definition", "difficulty": "intermediate"},
        ),
    ]

    # Validation examples
    val_data = [
        RAGDataInst(
            query="Explain computer vision in AI",
            ground_truth_answer="Computer Vision is a field of artificial intelligence
that trains computers to interpret and understand the visual world. Using digital images
from cameras and videos and deep learning models, machines can accurately identify and
classify objects.",
            relevant_doc_ids=["cv_basics"],
            metadata={"category": "explanation", "difficulty": "intermediate"},
        ),
        RAGDataInst(
            query="What are large language models?",
            ground_truth_answer="Large Language Models (LLMs) are a type of artificial
intelligence model designed to understand and generate human-like text. They are trained
on vast amounts of text data and can perform various natural language tasks such as
translation, summarization, and question answering.",
            relevant_doc_ids=["llm_basics"],
            metadata={"category": "definition", "difficulty": "advanced"},
        ),
    ]

    return train_data, val_data


def clean_answer(answer: str) -> str:
    """Clean up LLM answer by removing thinking tokens and truncating appropriately."""
    import re

    cleaned = re.sub(r"<think>.*?</think>", "", answer, flags=re.DOTALL)
    cleaned = cleaned.strip()

    # If still empty or starts with <think> without closing tag, try to find content after
    if not cleaned or cleaned.startswith("<think>"):
        lines = answer.split("\n")
        content_lines = []
        skip_thinking = False

        for line in lines:
            if "<think>" in line:
                skip_thinking = True
                continue
```

```
            if "</think>" in line:
                skip_thinking = False
                continue
            if not skip_thinking and line.strip():
                content_lines.append(line.strip())

        cleaned = " ".join(content_lines)

    # Show more of the answer - increase limit significantly
    if len(cleaned) > 500:
        return cleaned[:500] + "..."
    return cleaned or answer[:500] + ("..." if len(answer) > 500 else "")


def create_initial_prompts() -> dict[str, str]:
    """Create initial prompt templates for optimization."""
    return {
        "answer_generation": """You are an AI expert providing accurate technical
explanations.

Based on the retrieved context, provide a clear and informative answer to the user's
question.

Guidelines:
- Use information from the provided context
- Be accurate and concise
- Include key technical details
- Structure your response clearly

Context: {context}

Question: {query}

Answer:"""
    }


def setup_vector_store(store_name: str):
    """Factory function to set up the specified vector store."""
    setup_functions = {
        "chromadb": setup_chromadb_store,
        "lancedb": setup_lancedb_store,
        "milvus": setup_milvus_store,
        "qdrant": setup_qdrant_store,
        "weaviate": setup_weaviate_store,
    }

    if store_name not in setup_functions:
        raise ValueError(f"Unknown vector store: {store_name}. Supported:
{list(setup_functions.keys())}")

    return setup_functions[store_name]()


def parse_arguments():
    """Parse command line arguments."""
    parser = argparse.ArgumentParser(
        description="GEPA RAG Optimization Example with Multiple Vector Stores",
        formatter_class=argparse.RawDescriptionHelpFormatter,
        epilog="""
Examples:
  python rag_optimization.py --vector-store chromadb
  python rag_optimization.py --vector-store lancedb --model ollama/llama3.1:8b
  python rag_optimization.py --vector-store qdrant --max-iterations 10
  python rag_optimization.py --vector-store weaviate --model gpt-4o-mini

Supported Vector Stores:
  chromadb  - Local/persistent, simple setup (default)
```

```
    lancedb   - Serverless, no Docker required
    milvus    - Cloud-native, uses Lite mode locally
    qdrant    - High-performance, advanced filtering
    weaviate  - Hybrid search capabilities (requires Docker)
          """,
      )

      parser.add_argument(
          "--vector-store",
          type=str,
          default="chromadb",
          choices=["chromadb", "lancedb", "milvus", "qdrant", "weaviate"],
          help="Vector store to use (default: chromadb)",
      )
      parser.add_argument("--model", type=str, default="ollama/qwen3:8b", help="LLM model
(default: ollama/qwen3:8b)")
      parser.add_argument(
          "--embedding-model",
          type=str,
          default="ollama/nomic-embed-text:latest",
          help="Embedding model (default: ollama/nomic-embed-text:latest)",
      )
      parser.add_argument(
          "--max-iterations",
          type=int,
          default=5,
          help="GEPA optimization iterations (default: 5, use 0 to skip optimization)",
      )
      parser.add_argument("--verbose", action="store_true", help="Enable verbose output")

      return parser.parse_args()


  def main():
      """Main function demonstrating RAG optimization with multiple vector stores."""
      args = parse_arguments()

      print("🚀 GEPA RAG Optimization with Multiple Vector Stores")
      print("=" * 60)
      print(f"🗄️ Vector Store: {args.vector_store}")
      print(f"📊 Model: {args.model}")
      print(f"🔗 Embeddings: {args.embedding_model}")
      print(f"🔄 Max Iterations: {args.max_iterations}")

      try:
          # Step 1: Setup vector store
          print(f"\n1️⃣ Setting up {args.vector_store} vector store...")
          vector_store = setup_vector_store(args.vector_store)

          # Step 2: Create datasets
          print("\n2️⃣ Creating training and validation datasets...")
          train_data, val_data = create_training_data()
          print(f"   📚 Training examples: {len(train_data)}")
          print(f"   📝 Validation examples: {len(val_data)}")

          # Step 3: Initialize LLM client
          print(f"\n3️⃣ Initializing LLM client ({args.model})...")
          llm_client = create_llm_client(args.model)

          # Test LLM
          test_response = llm_client([{"role": "user", "content": "Say 'OK' only."}])
          if "Error:" not in test_response:
              print(f"   ✅ LLM connected: {test_response[:30]}...")
          else:
```

```
            print(f"   ⚠️ LLM issue: {test_response}")

        # Step 4: Initialize RAG adapter
        print("\n4️⃣ Initializing GenericRAGAdapter...")
        rag_config = {
            "retrieval_strategy": "similarity",
            "top_k": 3,
            "retrieval_weight": 0.3,
            "generation_weight": 0.7,
        }

        # Add hybrid search for Weaviate
        if args.vector_store == "weaviate":
            rag_config["retrieval_strategy"] = "hybrid"
            rag_config["hybrid_alpha"] = 0.7

        rag_adapter = GenericRAGAdapter(
            vector_store=vector_store,
            llm_model=llm_client,
            embedding_model=args.embedding_model,
            rag_config=rag_config,
        )

        # Step 5: Create initial prompts
        print("\n5️⃣ Creating initial prompts...")
        initial_prompts = create_initial_prompts()

        # Step 6: Test initial performance
        print("\n6️⃣ Testing initial performance...")
        eval_result = rag_adapter.evaluate(batch=val_data[:1], candidate=initial_prompts,
capture_traces=True)

        initial_score = eval_result.scores[0]
        print(f"   📊 Initial score: {initial_score:.3f}")
        print(f"   💬 Sample answer: {clean_answer(eval_result.outputs[0]
['final_answer'])}")

        # Step 7: Run GEPA optimization
        if args.max_iterations > 0:
            print(f"\n7️⃣ Running GEPA optimization ({args.max_iterations} iterations)...")

            result = gepa.optimize(
                seed_candidate=initial_prompts,
                trainset=train_data,
                valset=val_data,
                adapter=rag_adapter,
                reflection_lm=llm_client,
                max_metric_calls=args.max_iterations,
            )

            best_score = result.val_aggregate_scores[result.best_idx]
            print("   🎉 Optimization complete!")
            print(f"   🏆 Best score: {best_score:.3f}")
            print(f"   📈 Improvement: {best_score - initial_score:+.3f}")
            print(f"   🔄 Total iterations: {result.total_metric_calls or 0}")

            # Test optimized prompts
            print("\n   Testing optimized prompts...")
            optimized_result = rag_adapter.evaluate(
                batch=val_data[:1], candidate=result.best_candidate, capture_traces=False
            )
            print(f"   💬 Optimized answer: {clean_answer(optimized_result.outputs[0]
['final_answer'])}")
```

```
        else:
            print("\n7️⃣ Skipping optimization (use --max-iterations > 0 to enable)")

        print(f"\n✅ {args.vector_store.title()} RAG optimization completed
successfully!")

        # Clean up connections
        try:
            if hasattr(vector_store, "client") and hasattr(vector_store.client, "close"):
                vector_store.client.close()
        except Exception:
            pass

    except Exception as e:
        print(f"\n❌ Error: {e}")
        if args.verbose:
            import traceback

            traceback.print_exc()

        print("\n🔧 Troubleshooting tips:")
        if args.vector_store == "weaviate":
            print("  • Ensure Weaviate is running: curl http://localhost:8080/v1/meta")
            print(
                "  • Start Weaviate: docker run -p 8080:8080 -p 50051:50051
cr.weaviate.io/semitechnologies/weaviate:1.26.1"
            )
        elif args.vector_store == "qdrant":
            print("  • For external Qdrant: docker run -p 6333:6333 qdrant/qdrant")

        print("  • Ensure Ollama is running: ollama list")
        print("  • Check models are available: ollama pull qwen3:8b")
        print("  • For cloud models: set API keys (OPENAI_API_KEY, etc.)")
        print(f"  • Install dependencies: pip install
{get_install_command(args.vector_store)}")

        return 1

    return 0


if __name__ == "__main__":
    sys.exit(main())
```

## src/gepa/examples/terminal-bench/train_terminus.py

```
import argparse
import json
from pathlib import Path

import litellm
from terminal_bench.agents.terminus_1 import AgentResult, Chat, FailureMode, Terminus
from terminal_bench.dataset.dataset import Dataset
from terminal_bench.terminal.tmux_session import TmuxSession

from gepa import optimize
from gepa.adapters.terminal_bench_adapter.terminal_bench_adapter import (
    TerminalBenchTask,
    TerminusAdapter,
)
```

```
INSTRUCTION_PROMPT_PATH = Path(__file__).parent / "prompt-
templates/instruction_prompt.txt"


class TerminusWrapper(Terminus):
    def __init__(
        self,
        model_name: str,
        max_episodes: int = 50,
        api_base: str | None = None,
        **kwargs,
    ):
        self.PROMPT_TEMPLATE_PATH = Path(__file__).parent / "prompt-
templates/terminus.txt"
        self.instruction_prompt = INSTRUCTION_PROMPT_PATH.read_text()
        super().__init__(model_name, max_episodes, api_base, **kwargs)

    def perform_task(
        self,
        instruction: str,
        session: TmuxSession,
        logging_dir: Path | None = None,
    ):
        chat = Chat(self._llm)

        initial_prompt = self.instruction_prompt + self._prompt_template.format(
            response_schema=self._response_schema,
            instruction=instruction,
            history="",
            terminal_state=session.capture_pane(),
        )

        self._run_agent_loop(initial_prompt, session, chat, logging_dir)

        return AgentResult(
            total_input_tokens=chat.total_input_tokens,
            total_output_tokens=chat.total_output_tokens,
            failure_mode=FailureMode.NONE,
            timestamped_markers=self._timestamped_markers,
        )


if __name__ == "__main__":
    parser = argparse.ArgumentParser()
    parser.add_argument("--model_name", type=str, default="gpt-4o-mini")
    parser.add_argument("--n_concurrent", type=int, default=6)
    args = parser.parse_args()

    initial_prompt_from_terminus = """
You are an AI assistant tasked with solving command-line tasks in a Linux environment. You
will be given a task instruction and the output from previously executed commands. Your
goal is to solve the task by providing batches of shell commands.

For each response:
1. Analyze the current state based on any terminal output provided
2. Determine the next set of commands needed to make progress
3. Decide if you need to see the output of these commands before proceeding

Don't include markdown formatting.

Note that you operate directly on the terminal from inside a tmux session. Use tmux
keystrokes like `C-x` or `Escape` to interactively navigate the terminal. If you would
like to execute a command that you have written you will need to append a newline
character to the end of your command.

For example, if you write "ls -la" you will need to append a newline character to the end
of your command like this: `ls -la\n`.
```

```
One thing to be very careful about is handling interactive sessions like less, vim, or git
diff. In these cases, you should not wait for the output of the command. Instead, you
should send the keystrokes to the terminal as if you were typing them.
"""

    terminal_bench_dataset = Dataset(name="terminal-bench-core", version="head")
    terminal_bench_dataset.sort_by_duration()

    terminal_bench_tasks = terminal_bench_dataset._tasks[::-1]

    trainset = [
        TerminalBenchTask(task_id=task.name, model_name=args.model_name) for task in
terminal_bench_tasks[30:50]
    ]
    valset = [TerminalBenchTask(task_id=task.name, model_name=args.model_name) for task in
terminal_bench_tasks[:30]]

    testset = [
        TerminalBenchTask(task_id=task.name, model_name=args.model_name)
        for task in terminal_bench_tasks[50:]
        if task.name != "chem-rf"
    ]

    reflection_lm_name = "openai/gpt-5"
    reflection_lm = (
        lambda prompt: litellm.completion(
            model=reflection_lm_name,
            messages=[{"role": "user", "content": prompt}],
            reasoning_effort="high",
        )
        .choices[0]
        .message.content
    )

    adapter = TerminusAdapter(n_concurrent=args.n_concurrent,
instruction_prompt_path=INSTRUCTION_PROMPT_PATH)
    testset_results_no_prompt = adapter.evaluate(testset, {"instruction_prompt": ""},
capture_traces=True)
    testset_results_before_opt = adapter.evaluate(
        testset,
        {"instruction_prompt": initial_prompt_from_terminus},
        capture_traces=True,
    )

    with open("gepa_terminus/testset_results_no_prompt.json", "w") as f:
        json.dump(
            {
                "score": sum(trajectory["success"] for trajectory in
testset_results_no_prompt.trajectories),
                "trajectories": testset_results_no_prompt.trajectories,
            },
            f,
            indent=4,
        )
    with open("gepa_terminus/testset_results_before_opt.json", "w") as f:
        json.dump(
            {
                "score": sum(trajectory["success"] for trajectory in
testset_results_before_opt.trajectories),
                "trajectories": testset_results_before_opt.trajectories,
            },
            f,
            indent=4,
        )

    optimized_results = optimize(
        seed_candidate={"instruction_prompt": initial_prompt_from_terminus},
        trainset=trainset,
```

```
        valset=valset,
        adapter=adapter,
        reflection_lm=reflection_lm,
        use_wandb=True,
        max_metric_calls=400,
        reflection_minibatch_size=3,
        perfect_score=1,
        skip_perfect_score=False,
        run_dir="gepa_terminus",
    )

    testset_results_after_opt = adapter.evaluate(
        testset,
        {"instruction_prompt": optimized_results.best_candidate["instruction_prompt"]},
        capture_traces=True,
    )

    with open("gepa_terminus/optimized_results.json", "w") as f:
        json.dump(
            {
                "score": sum(trajectory["success"] for trajectory in
testset_results_after_opt.trajectories),
                "trajectories": testset_results_after_opt.trajectories,
            },
            f,
            indent=4,
        )
```

### src/gepa/gepa_utils.py

```
# Copyright (c) 2025 Lakshya A Agrawal and the GEPA contributors
# https://github.com/gepa-ai/gepa


import random
from typing import Any, Mapping


def json_default(x):
    """Default JSON encoder for objects that are not serializable by default."""
    try:
        return {**x}
    except Exception:
        return repr(x)


def idxmax(lst: list[float]) -> int:
    """Return the index of the maximum value in a list."""
    max_val = max(lst)
    return lst.index(max_val)


def is_dominated(y, programs, program_at_pareto_front_valset):
    y_fronts = [front for front in program_at_pareto_front_valset.values() if y in front]
    for front in y_fronts:
        found_dominator_in_front = False
        for other_prog in front:
            if other_prog in programs:
                found_dominator_in_front = True
                break
        if not found_dominator_in_front:
            return False

    return True
```

```
def remove_dominated_programs(program_at_pareto_front_valset, scores=None):
    freq = {}
    for front in program_at_pareto_front_valset.values():
        for p in front:
            freq[p] = freq.get(p, 0) + 1

    dominated = set()
    programs = list(freq.keys())

    if scores is None:
        scores = dict.fromkeys(programs, 1)

    programs = sorted(programs, key=lambda x: scores[x], reverse=False)

    found_to_remove = True
    while found_to_remove:
        found_to_remove = False
        for y in programs:
            if y in dominated:
                continue
            if is_dominated(y, set(programs).difference({y}).difference(dominated),
program_at_pareto_front_valset):
                dominated.add(y)
                found_to_remove = True
                break

    dominators = [p for p in programs if p not in dominated]
    for front in program_at_pareto_front_valset.values():
        if not front:
            continue
        assert any(p in front for p in dominators)

    new_program_at_pareto_front_valset = {
        val_id: {prog_idx for prog_idx in front if prog_idx in dominators}
        for val_id, front in program_at_pareto_front_valset.items()
    }
    for val_id, front_new in new_program_at_pareto_front_valset.items():
        assert front_new.issubset(program_at_pareto_front_valset[val_id])

    return new_program_at_pareto_front_valset


def find_dominator_programs(pareto_front_programs,
train_val_weighted_agg_scores_for_all_programs):
    train_val_pareto_front_programs = pareto_front_programs
    new_program_at_pareto_front_valset = remove_dominated_programs(
        train_val_pareto_front_programs,
scores=train_val_weighted_agg_scores_for_all_programs
    )
    uniq_progs = []
    for front in new_program_at_pareto_front_valset.values():
        uniq_progs.extend(front)
    uniq_progs = set(uniq_progs)
    return list(uniq_progs)


def select_program_candidate_from_pareto_front(
    pareto_front_programs: Mapping[Any, set[int]],
    train_val_weighted_agg_scores_for_all_programs: list[float],
    rng: random.Random,
) -> int:
    train_val_pareto_front_programs = pareto_front_programs
    new_program_at_pareto_front_valset = remove_dominated_programs(
        train_val_pareto_front_programs,
scores=train_val_weighted_agg_scores_for_all_programs
    )
```

```
    program_frequency_in_validation_pareto_front = {}
    for testcase_pareto_front in new_program_at_pareto_front_valset.values():
        for prog_idx in testcase_pareto_front:
            if prog_idx not in program_frequency_in_validation_pareto_front:
                program_frequency_in_validation_pareto_front[prog_idx] = 0
            program_frequency_in_validation_pareto_front[prog_idx] += 1

    sampling_list = [
        prog_idx for prog_idx, freq in
program_frequency_in_validation_pareto_front.items() for _ in range(freq)
    ]

    # TODO: Determine if we need this fallback
    # if not sampling_list:
    #     # No Pareto programs survived; fall back to the globally highest-scoring
program.
    #     return idxmax(train_val_weighted_agg_scores_for_all_programs)
    assert len(sampling_list) > 0

    curr_prog_id = rng.choice(sampling_list)
    return curr_prog_id
```

## src/gepa/logging/__init__.py

## src/gepa/logging/experiment_tracker.py

```
# Copyright (c) 2025 Lakshya A Agrawal and the GEPA contributors
# https://github.com/gepa-ai/gepa

from typing import Any


class ExperimentTracker:
    """
    Unified experiment tracking that supports both wandb and mlflow.
    """

    def __enter__(self):
        """Context manager entry."""
        self.initialize()
        self.start_run()
        return self

    def __exit__(self, exc_type, exc_val, exc_tb):
        """Context manager exit - always end the run."""
        self.end_run()
        return False  # Don't suppress exceptions

    def __init__(
        self,
        use_wandb: bool = False,
        wandb_api_key: str | None = None,
        wandb_init_kwargs: dict[str, Any] | None = None,
        use_mlflow: bool = False,
        mlflow_tracking_uri: str | None = None,
        mlflow_experiment_name: str | None = None,
    ):
        self.use_wandb = use_wandb
```

```
        self.use_mlflow = use_mlflow

        self.wandb_api_key = wandb_api_key
        self.wandb_init_kwargs = wandb_init_kwargs or {}
        self.mlflow_tracking_uri = mlflow_tracking_uri
        self.mlflow_experiment_name = mlflow_experiment_name

        self._created_mlflow_run = False

    def initialize(self):
        """Initialize the logging backends."""
        if self.use_wandb:
            self._initialize_wandb()
        if self.use_mlflow:
            self._initialize_mlflow()

    def _initialize_wandb(self):
        """Initialize wandb."""
        try:
            import wandb  # type: ignore

            if self.wandb_api_key:
                wandb.login(key=self.wandb_api_key, verify=True)
            else:
                wandb.login()
        except ImportError:
            raise ImportError("wandb is not installed. Please install it or set
backend='mlflow' or 'none'.")
        except Exception as e:
            raise RuntimeError(f"Error logging into wandb: {e}")

    def _initialize_mlflow(self):
        """Initialize mlflow."""
        try:
            import mlflow  # type: ignore

            if self.mlflow_tracking_uri:
                mlflow.set_tracking_uri(self.mlflow_tracking_uri)
            if self.mlflow_experiment_name:
                mlflow.set_experiment(self.mlflow_experiment_name)
        except ImportError:
            raise ImportError("mlflow is not installed. Please install it or set
backend='wandb' or 'none'.")
        except Exception as e:
            raise RuntimeError(f"Error setting up mlflow: {e}")

    def start_run(self):
        """Start a new run."""
        if self.use_wandb:
            import wandb  # type: ignore

            wandb.init(**self.wandb_init_kwargs)
        if self.use_mlflow:
            import mlflow  # type: ignore

            # Only start a new run if there's no active run
            if mlflow.active_run() is None:
                mlflow.start_run()
                self._created_mlflow_run = True
            else:
                self._created_mlflow_run = False

    def log_metrics(self, metrics: dict[str, Any], step: int | None = None):
        """Log metrics to the active backends."""
        if self.use_wandb:
            try:
                import wandb  # type: ignore
```

```
                wandb.log(metrics, step=step)
            except Exception as e:
                print(f"Warning: Failed to log to wandb: {e}")

        if self.use_mlflow:
            try:
                import mlflow  # type: ignore

                mlflow.log_metrics(metrics, step=step)
            except Exception as e:
                print(f"Warning: Failed to log to mlflow: {e}")

    def end_run(self):
        """End the current run."""
        if self.use_wandb:
            try:
                import wandb  # type: ignore

                if wandb.run is not None:
                    wandb.finish()
            except Exception as e:
                print(f"Warning: Failed to end wandb run: {e}")

        if self.use_mlflow:
            try:
                import mlflow  # type: ignore

                if self._created_mlflow_run and mlflow.active_run() is not None:
                    mlflow.end_run()
                    self._created_mlflow_run = False
            except Exception as e:
                print(f"Warning: Failed to end mlflow run: {e}")

    def is_active(self) -> bool:
        """Check if any backend has an active run."""
        if self.use_wandb:
            try:
                import wandb  # type: ignore

                if wandb.run is not None:
                    return True
            except Exception:
                pass

        if self.use_mlflow:
            try:
                import mlflow  # type: ignore

                if mlflow.active_run() is not None:
                    return True
            except Exception:
                pass

        return False


def create_experiment_tracker(
    use_wandb: bool = False,
    wandb_api_key: str | None = None,
    wandb_init_kwargs: dict[str, Any] | None = None,
    use_mlflow: bool = False,
    mlflow_tracking_uri: str | None = None,
    mlflow_experiment_name: str | None = None,
) -> ExperimentTracker:
    """
    Create an experiment tracker based on the specified backends.

    Args:
```

```
        use_wandb: Whether to use wandb
        use_mlflow: Whether to use mlflow
        wandb_api_key: API key for wandb
        wandb_init_kwargs: Additional kwargs for wandb.init()
        mlflow_tracking_uri: Tracking URI for mlflow
        mlflow_experiment_name: Experiment name for mlflow

    Returns:
        ExperimentTracker instance

    Note:
        Both wandb and mlflow can be used simultaneously if desired.
    """
    return ExperimentTracker(
        use_wandb=use_wandb,
        wandb_api_key=wandb_api_key,
        wandb_init_kwargs=wandb_init_kwargs,
        use_mlflow=use_mlflow,
        mlflow_tracking_uri=mlflow_tracking_uri,
        mlflow_experiment_name=mlflow_experiment_name,
    )
```

### src/gepa/logging/logger.py

```
# Copyright (c) 2025 Lakshya A Agrawal and the GEPA contributors
# https://github.com/gepa-ai/gepa

import sys
from typing import Protocol


class LoggerProtocol(Protocol):
    def log(self, message: str): ...


class StdOutLogger(LoggerProtocol):
    def log(self, message: str):
        print(message)


class Tee:
    def __init__(self, *files):
        self.files = files

    def write(self, obj):
        for f in self.files:
            f.write(obj)

    def flush(self):
        for f in self.files:
            if hasattr(f, "flush"):
                f.flush()

    def isatty(self):
        # True if any of the files is a terminal
        return any(hasattr(f, "isatty") and f.isatty() for f in self.files)

    def close(self):
        for f in self.files:
            if hasattr(f, "close"):
                f.close()

    def fileno(self):
        for f in self.files:
```

```
            if hasattr(f, "fileno"):
                return f.fileno()
        raise OSError("No underlying file object with fileno")


class Logger(LoggerProtocol):
    def __init__(self, filename, mode="a"):
        self.file_handle = open(filename, mode)
        self.file_handle_stderr = open(filename.replace("run_log.", "run_log_stderr."),
mode)
        self.modified_sys = False

    def __enter__(self):
        self.original_stdout = sys.stdout
        self.original_stderr = sys.stderr
        sys.stdout = Tee(sys.stdout, self.file_handle)
        sys.stderr = Tee(sys.stderr, self.file_handle_stderr)
        self.modified_sys = True
        return self

    def __exit__(self, exc_type, exc_value, traceback):
        sys.stdout = self.original_stdout
        sys.stderr = self.original_stderr
        self.file_handle.close()
        self.file_handle_stderr.close()
        self.modified_sys = False

    def log(self, *args, **kwargs):
        if self.modified_sys:
            print(*args, **kwargs)
        else:
            # Emulate print(*args, **kwargs) behavior but write to the file
            print(*args, **kwargs)
            print(*args, file=self.file_handle_stderr, **kwargs)
        self.file_handle.flush()
        self.file_handle_stderr.flush()
```

### src/gepa/logging/utils.py

```
# Copyright (c) 2025 Lakshya A Agrawal and the GEPA contributors
# https://github.com/gepa-ai/gepa


from gepa.core.adapter import DataInst
from gepa.core.data_loader import DataId
from gepa.core.state import GEPAState
from gepa.strategies.eval_policy import EvaluationPolicy


def log_detailed_metrics_after_discovering_new_program(
    logger,
    gepa_state: GEPAState,
    new_program_idx,
    valset_subscores,
    experiment_tracker,
    linear_pareto_front_program_idx,
    valset_size: int,
    val_evaluation_policy: EvaluationPolicy[DataId, DataInst],
    log_individual_valset_scores_and_programs: bool = False
):
    # best_prog_per_agg_val_score = idxmax(gepa_state.program_full_scores_val_set)
    best_prog_per_agg_val_score = val_evaluation_policy.get_best_program(gepa_state)
    best_score_on_valset =
val_evaluation_policy.get_valset_score(best_prog_per_agg_val_score, gepa_state)
```

```
    # avg, coverage = gepa_state.get_program_average_val_subset(new_program_idx)
    valset_score = val_evaluation_policy.get_valset_score(new_program_idx, gepa_state)
    coverage = len(valset_subscores)
    logger.log(
        f"Iteration {gepa_state.i + 1}: Valset score for new program: {valset_score}"
        f" (coverage {coverage} / {valset_size})"
    )

    agg_valset_score_new_program = val_evaluation_policy.get_valset_score(new_program_idx,
gepa_state)

    logger.log(f"Iteration {gepa_state.i + 1}: Val aggregate for new program:
{agg_valset_score_new_program}")
    logger.log(f"Iteration {gepa_state.i + 1}: Individual valset scores for new program:
{valset_subscores}")
    logger.log(f"Iteration {gepa_state.i + 1}: New valset pareto front scores:
{gepa_state.pareto_front_valset}")

    pareto_scores = list(gepa_state.pareto_front_valset.values())
    assert all(score > float("-inf") for score in pareto_scores), (
        "Should have at least one valid score per validation example"
    )
    assert len(pareto_scores) > 0
    pareto_avg = sum(pareto_scores) / len(pareto_scores)

    logger.log(f"Iteration {gepa_state.i + 1}: Valset pareto front aggregate score:
{pareto_avg}")
    logger.log(
        f"Iteration {gepa_state.i + 1}: Updated valset pareto front programs:
{gepa_state.program_at_pareto_front_valset}"
    )
    logger.log(
        f"Iteration {gepa_state.i + 1}: Best valset aggregate score so far:
{max(gepa_state.program_full_scores_val_set)}"
    )
    logger.log(
        f"Iteration {gepa_state.i + 1}: Best program as per aggregate score on valset:
{best_prog_per_agg_val_score}"
    )
    logger.log(f"Iteration {gepa_state.i + 1}: Best score on valset:
{best_score_on_valset}")
    logger.log(f"Iteration {gepa_state.i + 1}: Linear pareto front program index:
{linear_pareto_front_program_idx}")
    logger.log(f"Iteration {gepa_state.i + 1}: New program candidate index:
{new_program_idx}")

    metrics = {
        "iteration": gepa_state.i + 1,
        "new_program_idx": new_program_idx,
        "valset_pareto_front_agg": pareto_avg,
        "valset_pareto_front_programs": {k: list(v) for k, v in
gepa_state.program_at_pareto_front_valset.items()},
        "best_valset_agg_score": best_score_on_valset,
        "linear_pareto_front_program_idx": linear_pareto_front_program_idx,
        "best_program_as_per_agg_score_valset": best_prog_per_agg_val_score,
        "best_score_on_valset": best_score_on_valset,
        "val_evaluated_count_new_program": coverage,
        "val_total_count": valset_size,
        "val_program_average": valset_score,
    }
    if log_individual_valset_scores_and_programs:
        metrics.update({
            "valset_pareto_front_scores": dict(gepa_state.pareto_front_valset),
            "individual_valset_score_new_program": dict(valset_subscores),
        })

    experiment_tracker.log_metrics(metrics, step=gepa_state.i + 1)
```

## src/gepa/proposer/__init__.py

## src/gepa/proposer/base.py

```
# Copyright (c) 2025 Lakshya A Agrawal and the GEPA contributors
# https://github.com/gepa-ai/gepa

from dataclasses import dataclass, field
from typing import Any, Generic, Protocol

from gepa.core.data_loader import DataId
from gepa.core.state import GEPAState


@dataclass
class CandidateProposal(Generic[DataId]):
    candidate: dict[str, str]
    parent_program_ids: list[int]
    # Optional mini-batch / subsample info
    subsample_indices: list[DataId] | None = None
    subsample_scores_before: list[float] | None = None
    subsample_scores_after: list[float] | None = None
    # Free-form metadata for logging/trace
    tag: str = ""
    metadata: dict[str, Any] = field(default_factory=dict)


class ProposeNewCandidate(Protocol[DataId]):
    """
    Strategy that receives the current optimizer state and proposes a new candidate or
returns None.
    It may compute subsample evaluations, set trace fields in state, etc.
    The engine will handle acceptance and full eval unless the strategy already did those
and encoded in metadata.
    """

    def propose(self, state: GEPAState[Any, DataId]) -> CandidateProposal | None: ...
```

## src/gepa/proposer/merge.py

```
# Copyright (c) 2025 Lakshya A Agrawal and the GEPA contributors
# https://github.com/gepa-ai/gepa

import math
import random
from collections.abc import Callable, Iterable, Sequence
from copy import deepcopy

from gepa.core.adapter import Candidate, DataInst, EvaluatorFn, RolloutOutput
from gepa.core.data_loader import DataId, DataLoader
from gepa.core.state import GEPAState, ProgramIdx
from gepa.gepa_utils import find_dominator_programs
from gepa.logging.logger import LoggerProtocol
```

```
from gepa.proposer.base import CandidateProposal, ProposeNewCandidate

AncestorLog = tuple[int, int, int]
MergeDescription = tuple[int, int, tuple[int, ...]]
MergeAttempt = tuple[Candidate, ProgramIdx, ProgramIdx, ProgramIdx] | None


def does_triplet_have_desirable_predictors(
    program_candidates: Sequence[Candidate],
    ancestor: ProgramIdx,
    id1: ProgramIdx,
    id2: ProgramIdx,
) -> bool:
    found_predictors: list[tuple[int, int]] = []
    pred_names = list(program_candidates[ancestor].keys())
    for pred_idx, pred_name in enumerate(pred_names):
        pred_anc = program_candidates[ancestor][pred_name]
        pred_id1 = program_candidates[id1][pred_name]
        pred_id2 = program_candidates[id2][pred_name]
        if (pred_anc == pred_id1 or pred_anc == pred_id2) and pred_id1 != pred_id2:
            same_as_ancestor_id = 1 if pred_anc == pred_id1 else 2
            found_predictors.append((pred_idx, same_as_ancestor_id))

    return len(found_predictors) > 0


def filter_ancestors(
    i: ProgramIdx,
    j: ProgramIdx,
    common_ancestors: Iterable[ProgramIdx],
    merges_performed: tuple[list[AncestorLog], list[MergeDescription]],
    agg_scores: Sequence[float],
    program_candidates: Sequence[Candidate],
) -> list[ProgramIdx]:
    filtered_ancestors: list[ProgramIdx] = []
    for ancestor in common_ancestors:
        if (i, j, ancestor) in merges_performed[0]:
            continue

        if agg_scores[ancestor] > agg_scores[i] or agg_scores[ancestor] > agg_scores[j]:
            continue

        if not does_triplet_have_desirable_predictors(program_candidates, ancestor, i, j):
            continue

        filtered_ancestors.append(ancestor)
    return filtered_ancestors


def find_common_ancestor_pair(
    rng: random.Random,
    parent_list: Sequence[Sequence[int | None]],
    program_indexes: Sequence[int],
    merges_performed: tuple[list[AncestorLog], list[MergeDescription]],
    agg_scores: Sequence[float],
    program_candidates: Sequence[Candidate],
    max_attempts: int = 10,
) -> tuple[int, int, int] | None:
    def get_ancestors(node: int, ancestors_found: set[int]) -> list[int]:
        parents = parent_list[node]
        for parent in parents:
            if parent is not None and parent not in ancestors_found:
                ancestors_found.add(parent)
                get_ancestors(parent, ancestors_found)

        return list(ancestors_found)

    for _ in range(max_attempts):
```

```
        if len(program_indexes) < 2:
            return None
        i, j = rng.sample(list(program_indexes), 2)
        if i == j:
            continue

        if j < i:
            i, j = j, i

        ancestors_i = get_ancestors(i, set())
        ancestors_j = get_ancestors(j, set())

        if j in ancestors_i or i in ancestors_j:
            # If one is an ancestor of the other, we cannot merge them
            continue

        common_ancestors = set(ancestors_i) & set(ancestors_j)
        common_ancestors = filter_ancestors(i, j, common_ancestors, merges_performed,
agg_scores, program_candidates)
        if common_ancestors:
            # Select a random common ancestor
            common_ancestor = rng.choices(
                list(common_ancestors),
                k=1,
                weights=[agg_scores[ancestor] for ancestor in common_ancestors],
            )[0]
            return (i, j, common_ancestor)

    return None


def sample_and_attempt_merge_programs_by_common_predictors(
    agg_scores: Sequence[float],
    rng: random.Random,
    merge_candidates: Sequence[int],
    merges_performed: tuple[list[AncestorLog], list[MergeDescription]],
    program_candidates: Sequence[Candidate],
    parent_program_for_candidate: Sequence[Sequence[int | None]],
    has_val_support_overlap: Callable[[ProgramIdx, ProgramIdx], bool] | None = None,
    max_attempts: int = 10,
) -> MergeAttempt:
    if len(merge_candidates) < 2:
        return None
    if len(parent_program_for_candidate) < 3:
        return None

    for _ in range(max_attempts):
        ids_to_merge = find_common_ancestor_pair(
            rng,
            parent_program_for_candidate,
            list(merge_candidates),
            merges_performed=merges_performed,
            agg_scores=agg_scores,
            program_candidates=program_candidates,
            max_attempts=max_attempts,
        )
        if ids_to_merge is None:
            continue
        id1, id2, ancestor = ids_to_merge

        assert (id1, id2, ancestor) not in merges_performed, "This pair has already been
merged"
        assert agg_scores[ancestor] <= agg_scores[id1], "Ancestor should not be better
than its descendants"
        assert agg_scores[ancestor] <= agg_scores[id2], "Ancestor should not be better
than its descendants"
        assert id1 != id2, "Cannot merge the same program"
```

```
        # Now we have a common ancestor, which is outperformed by both its descendants

        new_program: Candidate = deepcopy(program_candidates[ancestor])

        new_prog_desc: tuple[int, ...] = ()

        pred_names = set(program_candidates[ancestor].keys())
        assert pred_names == set(program_candidates[id1].keys()) ==
set(program_candidates[id2].keys()), (
            "Predictors should be the same across all programs"
        )
        for pred_name in pred_names:
            pred_anc = program_candidates[ancestor][pred_name]
            pred_id1 = program_candidates[id1][pred_name]
            pred_id2 = program_candidates[id2][pred_name]
            if (pred_anc == pred_id1 or pred_anc == pred_id2) and pred_id1 != pred_id2:
                # We have a predictor that is the same as one of its ancestors, so we can
update it with the other
                same_as_ancestor_id = 1 if pred_anc == pred_id1 else 2
                new_value_idx = id2 if same_as_ancestor_id == 1 else id1
                new_program[pred_name] = program_candidates[new_value_idx][pred_name]
                new_prog_desc = (*new_prog_desc, new_value_idx)
            elif pred_anc != pred_id1 and pred_anc != pred_id2:
                # Both predictors are different from  the ancestor, and it is difficult to
decide which one gives the benefits
                # We randomly select one of the descendants to update the predictor
                # The probability of selecting is proportional to the agg_scores of the
descendants
                # prog_to_get_instruction_from = id1 if (rng.random() < (agg_scores[id1] /
(agg_scores[id1] + agg_scores[id2]))) else id2
                prog_to_get_instruction_from = (
                    id1
                    if agg_scores[id1] > agg_scores[id2]
                    else (id2 if agg_scores[id2] > agg_scores[id1] else rng.choice([id1,
id2]))
                )
                new_program[pred_name] = program_candidates[prog_to_get_instruction_from]
[pred_name]
                new_prog_desc = (*new_prog_desc, prog_to_get_instruction_from)
            elif pred_id1 == pred_id2:
                # Either both predictors are the same, or both are different from the
ancestor
                # If both are different from the ancestor, we should use the new
predictor, so selecting either one of the descendants is fine
                # If both are same as the ancesor, again selecting any one of the
descendants is fine
                # So let's select id1
                new_program[pred_name] = program_candidates[id1][pred_name]
                new_prog_desc = (*new_prog_desc, id1)
            else:  # pragma: no cover - defensive
                raise AssertionError("Unexpected case in predictor merging logic")

        if (id1, id2, new_prog_desc) in merges_performed[1]:
            # This triplet has already been merged, so we skip it
            continue

        if has_val_support_overlap and not has_val_support_overlap(id1, id2):
            # not enough overlapping validation support for candidates
            continue

        merges_performed[1].append((id1, id2, new_prog_desc))

        return (new_program, id1, id2, ancestor)

    return None


class MergeProposer(ProposeNewCandidate[DataId]):
```

```
    """
    Implements merge flow that combines compatible descendants of a common ancestor.

    - Find merge candidates among Pareto front dominators
    - Attempt a merge via sample_and_attempt_merge_programs_by_common_predictors
    - Subsample eval on valset-driven selected indices
    - Return proposal if merge's subsample score >= max(parents)
    The engine handles full eval + adding to state.
    """

    def __init__(
        self,
        logger: LoggerProtocol,
        valset: DataLoader[DataId, DataInst],
        evaluator: EvaluatorFn,
        use_merge: bool,
        max_merge_invocations: int,
        val_overlap_floor: int = 5,
        rng: random.Random | None = None,
    ):
        self.logger = logger
        self.valset = valset
        self.evaluator = evaluator
        self.use_merge = use_merge
        self.max_merge_invocations = max_merge_invocations
        self.rng = rng if rng is not None else random.Random(0)

        if val_overlap_floor <= 0:
            raise ValueError("val_overlap_floor should be a positive integer")
        self.val_overlap_floor = val_overlap_floor
        # Internal counters matching original behavior
        self.merges_due = 0
        self.total_merges_tested = 0
        self.merges_performed: tuple[list[AncestorLog], list[MergeDescription]] = ([], [])

        # Toggle controlled by engine: set True when last iter found new program
        self.last_iter_found_new_program = False

    def schedule_if_needed(self) -> None:
        if self.use_merge and self.total_merges_tested < self.max_merge_invocations:
            self.merges_due += 1

    def select_eval_subsample_for_merged_program(
        self,
        scores1: dict[DataId, float],
        scores2: dict[DataId, float],
        num_subsample_ids: int = 5,
    ) -> list[DataId]:
        common_ids = list(set(scores1.keys()) & set(scores2.keys()))

        p1 = [idx for idx in common_ids if scores1[idx] > scores2[idx]]
        p2 = [idx for idx in common_ids if scores2[idx] > scores1[idx]]
        p3 = [idx for idx in common_ids if idx not in p1 and idx not in p2]

        n_each = max(1, math.ceil(num_subsample_ids / 3))
        selected: list[DataId] = []
        for bucket in (p1, p2, p3):
            if len(selected) >= num_subsample_ids:
                break
            available = [idx for idx in bucket if idx not in selected]
            take = min(len(available), n_each, num_subsample_ids - len(selected))
            if take > 0:
                selected += self.rng.sample(available, k=take)

        remaining = num_subsample_ids - len(selected)
        if remaining > 0:
            unused = [idx for idx in common_ids if idx not in selected]
            if len(unused) >= remaining:
```

```
                selected += self.rng.sample(unused, k=remaining)
            else:
                selected += self.rng.choices(common_ids, k=remaining)

        return selected[:num_subsample_ids]

    def propose(self, state: GEPAState[RolloutOutput, DataId]) ->
CandidateProposal[DataId] | None:
        i = state.i + 1
        state.full_program_trace[-1]["invoked_merge"] = True

        # Only attempt when scheduled by engine and after a new program in last iteration
        if not (self.use_merge and self.last_iter_found_new_program and self.merges_due >
0):
            self.logger.log(f"Iteration {i}: No merge candidates scheduled")
            return None

        def has_val_support_overlap(id1: ProgramIdx, id2: ProgramIdx) -> bool:
            common_ids = set(state.prog_candidate_val_subscores[id1].keys()) & set(
                state.prog_candidate_val_subscores[id2].keys()
            )
            return len(common_ids) >= self.val_overlap_floor

        pareto_front_programs = state.program_at_pareto_front_valset
        merge_candidates = find_dominator_programs(pareto_front_programs,
state.program_full_scores_val_set)
        merge_output = sample_and_attempt_merge_programs_by_common_predictors(
            agg_scores=state.program_full_scores_val_set,
            rng=self.rng,
            merge_candidates=merge_candidates,
            merges_performed=self.merges_performed,
            program_candidates=state.program_candidates,
            parent_program_for_candidate=state.parent_program_for_candidate,
            has_val_support_overlap=has_val_support_overlap,
        )

        if merge_output is None:
            self.logger.log(f"Iteration {i}: No merge candidates found")
            return None

        # success, new_program, id1, id2, ancestor
        new_program, id1, id2, ancestor = merge_output
        state.full_program_trace[-1]["merged"] = True
        state.full_program_trace[-1]["merged_entities"] = (id1, id2, ancestor)
        self.merges_performed[0].append((id1, id2, ancestor))
        self.logger.log(f"Iteration {i}: Merged programs {id1} and {id2} via ancestor
{ancestor}")

        subsample_ids = self.select_eval_subsample_for_merged_program(
            state.prog_candidate_val_subscores[id1],
            state.prog_candidate_val_subscores[id2],
        )
        mini_devset = self.valset.fetch(subsample_ids)
        # below is a post condition of `select_eval_subsample_for_merged_program`
        assert set(subsample_ids).issubset(state.prog_candidate_val_subscores[id1].keys())
        assert set(subsample_ids).issubset(state.prog_candidate_val_subscores[id2].keys())
        id1_sub_scores = [state.prog_candidate_val_subscores[id1][k] for k in
subsample_ids]
        id2_sub_scores = [state.prog_candidate_val_subscores[id2][k] for k in
subsample_ids]
        state.full_program_trace[-1]["subsample_ids"] = subsample_ids

        _, new_sub_scores = self.evaluator(mini_devset, new_program)

        state.full_program_trace[-1]["id1_subsample_scores"] = id1_sub_scores
        state.full_program_trace[-1]["id2_subsample_scores"] = id2_sub_scores
        state.full_program_trace[-1]["new_program_subsample_scores"] = new_sub_scores
```

```
        # Count evals
        state.total_num_evals += len(subsample_ids)

        # Acceptance will be evaluated by engine (>= max(parents))
        return CandidateProposal(
            candidate=new_program,
            parent_program_ids=[id1, id2],
            subsample_indices=subsample_ids,
            subsample_scores_before=[sum(id1_sub_scores), sum(id2_sub_scores)],
            subsample_scores_after=new_sub_scores,
            tag="merge",
            metadata={"ancestor": ancestor},
        )
```

### src/gepa/proposer/reflective_mutation/__init__.py

### src/gepa/proposer/reflective_mutation/base.py

```
# Copyright (c) 2025 Lakshya A Agrawal and the GEPA contributors
# https://github.com/gepa-ai/gepa

from dataclasses import dataclass
from typing import Any, ClassVar, Mapping, Protocol, runtime_checkable

from gepa.core.adapter import Trajectory
from gepa.core.state import GEPAState


@runtime_checkable
class CandidateSelector(Protocol):
    def select_candidate_idx(self, state: GEPAState) -> int: ...


class ReflectionComponentSelector(Protocol):
    def __call__(
        self,
        state: GEPAState,
        trajectories: list[Trajectory],
        subsample_scores: list[float],
        candidate_idx: int,
        candidate: dict[str, str],
    ) -> list[str]: ...


class LanguageModel(Protocol):
    def __call__(self, prompt: str) -> str: ...


@dataclass
class Signature:
    prompt_template: ClassVar[str]
    input_keys: ClassVar[list[str]]
    output_keys: ClassVar[list[str]]

    @classmethod
    def prompt_renderer(cls, input_dict: Mapping[str, Any]) -> str:
        raise NotImplementedError
```

```
    @classmethod
    def output_extractor(cls, lm_out: str) -> dict[str, str]:
        raise NotImplementedError

    @classmethod
    def run(cls, lm: LanguageModel, input_dict: Mapping[str, Any]) -> dict[str, str]:
        full_prompt = cls.prompt_renderer(input_dict)
        lm_out = lm(full_prompt).strip()
        return cls.output_extractor(lm_out)
```

### src/gepa/proposer/reflective_mutation/reflective_mutation.py

```
# Copyright (c) 2025 Lakshya A Agrawal and the GEPA contributors
# https://github.com/gepa-ai/gepa

from collections.abc import Mapping, Sequence
from typing import Any

from gepa.core.adapter import DataInst, GEPAAdapter, RolloutOutput, Trajectory
from gepa.core.data_loader import DataId, DataLoader, ensure_loader
from gepa.core.state import GEPAState
from gepa.proposer.base import CandidateProposal, ProposeNewCandidate
from gepa.proposer.reflective_mutation.base import (
    CandidateSelector,
    LanguageModel,
    ReflectionComponentSelector,
)
from gepa.strategies.batch_sampler import BatchSampler
from gepa.strategies.instruction_proposal import InstructionProposalSignature


class ReflectiveMutationProposer(ProposeNewCandidate[DataId]):
    """
    Implements current reflective mutation flow:
    - Select candidate via selector
    - Select minibatch via sampler
    - capture_traces_and_eval -> trajectories, subsample_scores
    - skip if all scores==perfect and skip_perfect_score
    - reflection + mutate -> new candidate
    - evaluate new candidate on same minibatch -> new_subsample_scores
    - Return proposal if improved; else None
    """

    def __init__(
        self,
        logger: Any,
        trainset: list[DataInst] | DataLoader[DataId, DataInst],
        adapter: GEPAAdapter[DataInst, Trajectory, RolloutOutput],
        candidate_selector: CandidateSelector,
        module_selector: ReflectionComponentSelector,
        batch_sampler: BatchSampler[DataId, DataInst],
        perfect_score: float,
        skip_perfect_score: bool,
        experiment_tracker: Any,
        reflection_lm: LanguageModel | None = None,
        reflection_prompt_template: str | None = None,
    ):
        self.logger = logger
        self.trainset = ensure_loader(trainset)
        self.adapter = adapter
        self.candidate_selector = candidate_selector
        self.module_selector = module_selector
        self.batch_sampler = batch_sampler
        self.perfect_score = perfect_score
```

```
        self.skip_perfect_score = skip_perfect_score
        self.experiment_tracker = experiment_tracker
        self.reflection_lm = reflection_lm

        InstructionProposalSignature.validate_prompt_template(reflection_prompt_template)
        self.reflection_prompt_template = reflection_prompt_template

    def propose_new_texts(
        self,
        candidate: dict[str, str],
        reflective_dataset: Mapping[str, Sequence[Mapping[str, Any]]],
        components_to_update: list[str],
    ) -> dict[str, str]:
        if self.adapter.propose_new_texts is not None:
            return self.adapter.propose_new_texts(candidate, reflective_dataset,
components_to_update)

        if self.reflection_lm is None:
            raise ValueError("reflection_lm must be provided when
adapter.propose_new_texts is None.")
        new_texts: dict[str, str] = {}
        for name in components_to_update:
            # Gracefully handle cases where a selected component has no data in
reflective_dataset
            if name not in reflective_dataset or not reflective_dataset.get(name):
                self.logger.log(
                    f"Component '{name}' is not in reflective dataset. Skipping."
                )
                continue

            base_instruction = candidate[name]
            dataset_with_feedback = reflective_dataset[name]
            new_texts[name] = InstructionProposalSignature.run(
                lm=self.reflection_lm,
                input_dict={
                    "current_instruction_doc": base_instruction,
                    "dataset_with_feedback": dataset_with_feedback,
                    "prompt_template": self.reflection_prompt_template,
                },
            )["new_instruction"]
        return new_texts

    def propose(self, state: GEPAState) -> CandidateProposal | None:
        i = state.i + 1

        curr_prog_id = self.candidate_selector.select_candidate_idx(state)
        curr_prog = state.program_candidates[curr_prog_id]
        state.full_program_trace[-1]["selected_program_candidate"] = curr_prog_id
        self.logger.log(
            f"Iteration {i}: Selected program {curr_prog_id} score:
{state.program_full_scores_val_set[curr_prog_id]}"
        )

        self.experiment_tracker.log_metrics({"iteration": i, "selected_program_candidate":
curr_prog_id}, step=i)

        subsample_ids = self.batch_sampler.next_minibatch_ids(self.trainset, state)
        state.full_program_trace[-1]["subsample_ids"] = subsample_ids
        minibatch = self.trainset.fetch(subsample_ids)

        # 1) Evaluate current program with traces
        eval_curr = self.adapter.evaluate(minibatch, curr_prog, capture_traces=True)
        state.total_num_evals += len(subsample_ids)
        state.full_program_trace[-1]["subsample_scores"] = eval_curr.scores

        if not eval_curr.trajectories or len(eval_curr.trajectories) == 0:
            self.logger.log(f"Iteration {i}: No trajectories captured. Skipping.")
            return None
```

```
        if self.skip_perfect_score and all(s >= self.perfect_score for s in
eval_curr.scores):
            self.logger.log(f"Iteration {i}: All subsample scores perfect. Skipping.")
            return None

        self.experiment_tracker.log_metrics({"subsample_score": sum(eval_curr.scores)},
step=i)

        # 2) Decide which predictors to update
        predictor_names_to_update = self.module_selector(
            state, eval_curr.trajectories, eval_curr.scores, curr_prog_id, curr_prog
        )

        # 3) Build reflective dataset and propose texts
        try:
            reflective_dataset = self.adapter.make_reflective_dataset(curr_prog,
eval_curr, predictor_names_to_update)
            new_texts = self.propose_new_texts(curr_prog, reflective_dataset,
predictor_names_to_update)
            for pname, text in new_texts.items():
                self.logger.log(f"Iteration {i}: Proposed new text for {pname}: {text}")
            self.experiment_tracker.log_metrics(
                {f"new_instruction_{pname}": text for pname, text in new_texts.items()},
step=i
            )
        except Exception as e:
            self.logger.log(f"Iteration {i}: Exception during reflection/proposal: {e}")
            import traceback

            self.logger.log(traceback.format_exc())
            return None

        # 4) Create candidate, evaluate on same minibatch (no need to capture traces)
        new_candidate = curr_prog.copy()
        for pname, text in new_texts.items():
            assert pname in new_candidate, f"{pname} missing in candidate"
            new_candidate[pname] = text

        eval_new = self.adapter.evaluate(minibatch, new_candidate, capture_traces=False)
        state.total_num_evals += len(subsample_ids)
        state.full_program_trace[-1]["new_subsample_scores"] = eval_new.scores

        new_sum = sum(eval_new.scores)
        self.experiment_tracker.log_metrics({"new_subsample_score": new_sum}, step=i)

        return CandidateProposal(
            candidate=new_candidate,
            parent_program_ids=[curr_prog_id],
            subsample_indices=subsample_ids,
            subsample_scores_before=eval_curr.scores,
            subsample_scores_after=eval_new.scores,
            tag="reflective_mutation",
        )
```

### src/gepa/strategies/__init__.py

### src/gepa/strategies/batch_sampler.py

```
# Copyright (c) 2025 Lakshya A Agrawal and the GEPA contributors
# https://github.com/gepa-ai/gepa

import random
from collections import Counter
from typing import Protocol

from gepa.core.adapter import DataInst
from gepa.core.data_loader import DataId, DataLoader
from gepa.core.state import GEPAState


class BatchSampler(Protocol[DataId, DataInst]):
    def next_minibatch_ids(self, loader: DataLoader[DataId, DataInst], state: GEPAState) -
> list[DataId]: ...


class EpochShuffledBatchSampler(BatchSampler[DataId, DataInst]):
    """
    Mirrors the original batching logic:
    - Shuffle ids each epoch
    - Pad to minibatch size with least frequent ids
    - Deterministic via state.rng1
    """

    def __init__(self, minibatch_size: int, rng: random.Random | None = None):
        self.minibatch_size = minibatch_size
        self.shuffled_ids: list[DataId] = []
        self.epoch = -1
        self.id_freqs = Counter()
        self.last_trainset_size = 0
        if rng is None:
            self.rng = random.Random(0)
        else:
            self.rng = rng

    def _update_shuffled(self, loader: DataLoader[DataId, DataInst]):
        all_ids = list(loader.all_ids())
        trainset_size = len(loader)
        self.last_trainset_size = trainset_size

        if trainset_size == 0:
            self.shuffled_ids = []
            self.id_freqs = Counter()
            return

        self.shuffled_ids = list(all_ids)
        self.rng.shuffle(self.shuffled_ids)
        self.id_freqs = Counter(self.shuffled_ids)

        mod = trainset_size % self.minibatch_size
        num_to_pad = (self.minibatch_size - mod) if mod != 0 else 0
        if num_to_pad > 0:
            for _ in range(num_to_pad):
                selected_id = self.id_freqs.most_common()[::-1][0][0]
                self.shuffled_ids.append(selected_id)
                self.id_freqs[selected_id] += 1

    def next_minibatch_ids(self, loader: DataLoader[DataId, DataInst], state: GEPAState) -
> list[DataId]:
        trainset_size = len(loader)
        if trainset_size == 0:
            raise ValueError("Cannot sample a minibatch from an empty loader.")

        base_idx = state.i * self.minibatch_size
        curr_epoch = 0 if self.epoch == -1 else base_idx // max(len(self.shuffled_ids), 1)
```

```
        needs_refresh = not self.shuffled_ids or trainset_size != self.last_trainset_size
or curr_epoch > self.epoch
        if needs_refresh:
            self.epoch = curr_epoch
            self._update_shuffled(loader)

        assert len(self.shuffled_ids) >= self.minibatch_size
        assert len(self.shuffled_ids) % self.minibatch_size == 0

        base_idx = base_idx % len(self.shuffled_ids)
        end_idx = base_idx + self.minibatch_size
        assert end_idx <= len(self.shuffled_ids)
        return self.shuffled_ids[base_idx:end_idx]
```

## src/gepa/strategies/candidate_selector.py

```
# Copyright (c) 2025 Lakshya A Agrawal and the GEPA contributors
# https://github.com/gepa-ai/gepa

import random

from gepa.core.state import GEPAState
from gepa.gepa_utils import idxmax, select_program_candidate_from_pareto_front
from gepa.proposer.reflective_mutation.base import CandidateSelector


class ParetoCandidateSelector(CandidateSelector):
    def __init__(self, rng: random.Random | None):
        if rng is None:
            self.rng = random.Random(0)
        else:
            self.rng = rng

    def select_candidate_idx(self, state: GEPAState) -> int:
        assert len(state.program_full_scores_val_set) == len(state.program_candidates)
        return select_program_candidate_from_pareto_front(
            state.program_at_pareto_front_valset,
            state.program_full_scores_val_set,
            self.rng,
        )


class CurrentBestCandidateSelector(CandidateSelector):
    def __init__(self):
        pass

    def select_candidate_idx(self, state: GEPAState) -> int:
        assert len(state.program_full_scores_val_set) == len(state.program_candidates)
        return idxmax(state.program_full_scores_val_set)


class EpsilonGreedyCandidateSelector(CandidateSelector):
    def __init__(self, epsilon: float, rng: random.Random | None):
        assert 0.0 <= epsilon <= 1.0
        self.epsilon = epsilon
        if rng is None:
            self.rng = random.Random(0)
        else:
            self.rng = rng

    def select_candidate_idx(self, state: GEPAState) -> int:
        assert len(state.program_full_scores_val_set) == len(state.program_candidates)
        if self.rng.random() < self.epsilon:
            return self.rng.randint(0, len(state.program_candidates) - 1)
```

```
        else:
            return idxmax(state.program_full_scores_val_set)
```

### src/gepa/strategies/component_selector.py

```
# Copyright (c) 2025 Lakshya A Agrawal and the GEPA contributors
# https://github.com/gepa-ai/gepa


from gepa.core.adapter import Trajectory
from gepa.core.state import GEPAState
from gepa.proposer.reflective_mutation.base import ReflectionComponentSelector


class RoundRobinReflectionComponentSelector(ReflectionComponentSelector):
    def __call__(
        self,
        state: GEPAState,
        trajectories: list[Trajectory],
        subsample_scores: list[float],
        candidate_idx: int,
        candidate: dict[str, str],
    ) -> list[str]:
        pid = state.named_predictor_id_to_update_next_for_program_candidate[candidate_idx]
        state.named_predictor_id_to_update_next_for_program_candidate[candidate_idx] =
(pid + 1) % len(
            state.list_of_named_predictors
        )
        name = state.list_of_named_predictors[pid]
        return [name]


class AllReflectionComponentSelector(ReflectionComponentSelector):
    def __call__(
        self,
        state: GEPAState,
        trajectories: list[Trajectory],
        subsample_scores: list[float],
        candidate_idx: int,
        candidate: dict[str, str],
    ) -> list[str]:
        return list(candidate.keys())
```

### src/gepa/strategies/eval_policy.py

```
"""Validation evaluation policy protocols and helpers."""

from __future__ import annotations

from abc import abstractmethod
from typing import Protocol, runtime_checkable

from gepa.core.data_loader import DataId, DataInst, DataLoader
from gepa.core.state import GEPAState, ProgramIdx


@runtime_checkable
class EvaluationPolicy(Protocol[DataId, DataInst]):  # type: ignore
    """Strategy for choosing validation ids to evaluate and identifying best programs for
```

```
validation instances."""

    @abstractmethod
    def get_eval_batch(
        self, loader: DataLoader[DataId, DataInst], state: GEPAState, target_program_idx:
ProgramIdx | None = None
    ) -> list[DataId]:
        """Select examples for evaluation for a program"""
        ...

    @abstractmethod
    def get_best_program(self, state: GEPAState) -> ProgramIdx:
        """Return "best" program given all validation results so far across candidates"""
        ...

    @abstractmethod
    def get_valset_score(self, program_idx: ProgramIdx, state: GEPAState) -> float:
        """Return the score of the program on the valset"""
        ...


class FullEvaluationPolicy(EvaluationPolicy[DataId, DataInst]):
    """Policy that evaluates all validation instances every time."""

    def get_eval_batch(
        self, loader: DataLoader[DataId, DataInst], state: GEPAState, target_program_idx:
ProgramIdx | None = None
    ) -> list[DataId]:
        """Always return the full ordered list of validation ids."""
        return list(loader.all_ids())

    def get_best_program(self, state: GEPAState) -> ProgramIdx:
        """Pick the program whose evaluated validation scores achieve the highest
average."""
        best_idx, best_score, best_coverage = -1, float("-inf"), -1
        for program_idx, scores in enumerate(state.prog_candidate_val_subscores):
            coverage = len(scores)
            avg = sum(scores.values()) / coverage if coverage else float("-inf")
            if avg > best_score or (avg == best_score and coverage > best_coverage):
                best_score = avg
                best_idx = program_idx
                best_coverage = coverage
        return best_idx

    def get_valset_score(self, program_idx: ProgramIdx, state: GEPAState) -> float:
        """Return the score of the program on the valset"""
        return state.get_program_average_val_subset(program_idx)[0]


__all__ = [
    "DataLoader",
    "EvaluationPolicy",
    "FullEvaluationPolicy",
]
```

### src/gepa/strategies/instruction_proposal.py

```
# Copyright (c) 2025 Lakshya A Agrawal and the GEPA contributors
# https://github.com/gepa-ai/gepa

import re
from collections.abc import Mapping, Sequence
from typing import Any, ClassVar
```

```
from gepa.proposer.reflective_mutation.base import Signature


class InstructionProposalSignature(Signature):
    default_prompt_template = """I provided an assistant with the following instructions
to perform a task for me:
```
<curr_instructions>
```

The following are examples of different task inputs provided to the assistant along with
the assistant's response for each of them, and some feedback on how the assistant's
response could be better:
```
<inputs_outputs_feedback>
```

Your task is to write a new instruction for the assistant.

Read the inputs carefully and identify the input format and infer detailed task
description about the task I wish to solve with the assistant.

Read all the assistant responses and the corresponding feedback. Identify all niche and
domain specific factual information about the task and include it in the instruction, as a
lot of it may not be available to the assistant in the future. The assistant may have
utilized a generalizable strategy to solve the task, if so, include that in the
instruction as well.

Provide the new instructions within ``` blocks."""

    input_keys: ClassVar[list[str]] = ["current_instruction_doc", "dataset_with_feedback",
"prompt_template"]
    output_keys: ClassVar[list[str]] = ["new_instruction"]

    @classmethod
    def validate_prompt_template(cls, prompt_template: str | None) -> None:
        if prompt_template is None:
            return
        missing_placeholders = [
            placeholder
            for placeholder in ("<curr_instructions>", "<inputs_outputs_feedback>")
            if placeholder not in prompt_template
        ]
        if missing_placeholders:
            raise ValueError(
                f"Missing placeholder(s) in prompt template: {',
'.join(missing_placeholders)}"
            )

    @classmethod
    def prompt_renderer(cls, input_dict: Mapping[str, Any]) -> str:
        current_instruction = input_dict.get("current_instruction_doc")
        if not isinstance(current_instruction, str):
            raise TypeError("current_instruction_doc must be a string")

        dataset = input_dict.get("dataset_with_feedback")
        if not isinstance(dataset, Sequence) or isinstance(dataset, (str, bytes)):
            raise TypeError("dataset_with_feedback must be a sequence of records")
        def format_samples(samples):
            def render_value(value, level=3):
                # level controls markdown header depth (###, ####, etc.)
                if isinstance(value, dict):
                    s = ""
                    for k, v in value.items():
                        s += f"{'#' * level} {k}\n"
                        s += render_value(v, min(level + 1, 6))
                    if not value:
                        s += "\n"
```

```
                    return s
                elif isinstance(value, list | tuple):
                    s = ""
                    for i, item in enumerate(value):
                        s += f"{'#' * level} Item {i + 1}\n"
                        s += render_value(item, min(level + 1, 6))
                    if not value:
                        s += "\n"
                    return s
                else:
                    return f"{str(value).strip()}\n\n"

            def convert_sample_to_markdown(sample, examplenum):
                s = f"# Example {examplenum}\n"
                for key, val in sample.items():
                    s += f"## {key}\n"
                    s += render_value(val, level=3)
                return s

            return "\n\n".join(convert_sample_to_markdown(sample, i + 1) for i, sample in
enumerate(samples))

        prompt_template = input_dict.get("prompt_template")
        if prompt_template is None:
            prompt_template = cls.default_prompt_template

        cls.validate_prompt_template(prompt_template)

        prompt = prompt_template.replace("<curr_instructions>", current_instruction)
        prompt = prompt.replace("<inputs_outputs_feedback>", format_samples(dataset))

        return prompt

    @classmethod
    def output_extractor(cls, lm_out: str) -> dict[str, str]:
        def extract_instruction_text() -> str:
            # Find the first and last backtick positions (if any)
            start = lm_out.find("```") + 3
            end = lm_out.rfind("```")

            # Handle if the first and last backticks are the same or overlap
            if start >= end:
                # Handle incomplete blocks
                stripped = lm_out.strip()
                if stripped.startswith("```"):
                    # Remove opening ``` and optional language specifier
                    match = re.match(r"^```\S*\n?", lm_out)
                    if match:
                        return lm_out[match.end() :].strip()
                elif stripped.endswith("```"):
                    # Remove closing ```
                    return stripped[:-3].strip()
                return stripped

            # Skip optional language specifier
            content = lm_out[start:end]
            match = re.match(r"^\S*\n", content)
            if match:
                content = content[match.end() :]

            return content.strip()

        return {"new_instruction": extract_instruction_text()}
```

**src/gepa/utils/__init__.py**

```
from .stop_condition import (
    CompositeStopper,
    FileStopper,
    MaxMetricCallsStopper,
    NoImprovementStopper,
    ScoreThresholdStopper,
    SignalStopper,
    StopperProtocol,
    TimeoutStopCondition,
)
```

### src/gepa/utils/stop_condition.py

```
"""
Utility functions for graceful stopping of GEPA runs.
"""

import os
import signal
import time
from typing import Literal, Protocol, runtime_checkable

from gepa.core.state import GEPAState


@runtime_checkable
class StopperProtocol(Protocol):
    """
    Protocol for stop condition objects.

    A stopper is a callable object that returns True when the optimization should stop.
    """

    def __call__(self, gepa_state: GEPAState) -> bool:
        """
        Check if the optimization should stop.

        Args:
            gepa_state: The current GEPA state containing optimization information

        Returns:
            True if the optimization should stop, False otherwise.
        """
        ...


class TimeoutStopCondition(StopperProtocol):
    """Stop callback that stops after a specified timeout."""

    def __init__(self, timeout_seconds: float):
        self.timeout_seconds = timeout_seconds
        self.start_time = time.time()

    def __call__(self, gepa_state: GEPAState) -> bool:
        # return true if timeout has been reached
        return time.time() - self.start_time > self.timeout_seconds


class FileStopper(StopperProtocol):
    """
    Stop callback that stops when a specific file exists.
    """
```

```
    def __init__(self, stop_file_path: str):
        self.stop_file_path = stop_file_path

    def __call__(self, gepa_state: GEPAState) -> bool:
        # returns true if stop file exists
        return os.path.exists(self.stop_file_path)

    def remove_stop_file(self):
        # remove the stop file
        if os.path.exists(self.stop_file_path):
            os.remove(self.stop_file_path)


class ScoreThresholdStopper(StopperProtocol):
    """
    Stop callback that stops when a score threshold is reached.
    """

    def __init__(self, threshold: float):
        self.threshold = threshold

    def __call__(self, gepa_state: GEPAState) -> bool:
        # return true if score threshold is reached
        try:
            current_best_score = (
                max(gepa_state.program_full_scores_val_set) if
gepa_state.program_full_scores_val_set else 0.0
            )
            return current_best_score >= self.threshold
        except Exception:
            return False


class NoImprovementStopper(StopperProtocol):
    """
    Stop callback that stops after a specified number of iterations without improvement.
    """

    def __init__(self, max_iterations_without_improvement: int):
        self.max_iterations_without_improvement = max_iterations_without_improvement
        self.best_score = float("-inf")
        self.iterations_without_improvement = 0

    def __call__(self, gepa_state: GEPAState) -> bool:
        # return true if max iterations without improvement reached
        try:
            current_score = (
                max(gepa_state.program_full_scores_val_set) if
gepa_state.program_full_scores_val_set else 0.0
            )
            if current_score > self.best_score:
                self.best_score = current_score
                self.iterations_without_improvement = 0
            else:
                self.iterations_without_improvement += 1

            return self.iterations_without_improvement >=
self.max_iterations_without_improvement
        except Exception:
            return False

    def reset(self):
        """Reset the counter (useful when manually improving the score)."""
        self.iterations_without_improvement = 0


class SignalStopper(StopperProtocol):
    """Stop callback that stops when a signal is received."""
```

```
    def __init__(self, signals=None):
        self.signals = signals or [signal.SIGINT, signal.SIGTERM]
        self._stop_requested = False
        self._original_handlers = {}
        self._setup_signal_handlers()

    def _setup_signal_handlers(self):
        """Set up signal handlers for graceful shutdown."""

        def signal_handler(signum, frame):
            self._stop_requested = True

        # Store original handlers and set new ones
        for sig in self.signals:
            try:
                self._original_handlers[sig] = signal.signal(sig, signal_handler)
            except (OSError, ValueError):
                # Signal not available on this platform
                pass

    def __call__(self, gepa_state: GEPAState) -> bool:
        # return true if a signal was received
        return self._stop_requested

    def cleanup(self):
        """Restore original signal handlers."""
        for sig, handler in self._original_handlers.items():
            try:
                signal.signal(sig, handler)
            except (OSError, ValueError):
                pass


class MaxTrackedCandidatesStopper(StopperProtocol):
    """
    Stop callback that stops after a maximum number of tracked candidates.
    """

    def __init__(self, max_tracked_candidates: int):
        self.max_tracked_candidates = max_tracked_candidates

    def __call__(self, gepa_state: GEPAState) -> bool:
        # return true if max tracked candidates reached
        return len(gepa_state.program_candidates) >= self.max_tracked_candidates


class MaxMetricCallsStopper(StopperProtocol):
    """
    Stop callback that stops after a maximum number of metric calls.
    """

    def __init__(self, max_metric_calls: int):
        self.max_metric_calls = max_metric_calls

    def __call__(self, gepa_state: GEPAState) -> bool:
        # return true if max metric calls reached
        return gepa_state.total_num_evals >= self.max_metric_calls


class CompositeStopper(StopperProtocol):
    """
    Stop callback that combines multiple stopping conditions.

    Allows combining several stoppers and stopping when any or all of them are triggered.
    """

    def __init__(self, *stoppers: StopperProtocol, mode: Literal["any", "all"] = "any"):
```

```
        # initialize composite stopper

        self.stoppers = stoppers
        self.mode = mode

    def __call__(self, gepa_state: GEPAState) -> bool:
        # return true if stopping condition is met
        if self.mode == "any":
            return any(stopper(gepa_state) for stopper in self.stoppers)
        elif self.mode == "all":
            return all(stopper(gepa_state) for stopper in self.stoppers)
        else:
            raise ValueError(f"Unknown mode: {self.mode}")
```