Improving Reliability of LLMs using GEPA for Fact Checking and Code Generation

Team: Amaan Mansuri (am14419), Ninad Chaudhari (nac8810),
Shravan Khunti (ssk10036), Harshit Bhargava (hb2976), Vishwa Raval (vmr7988)

Literature Survey

GEPA. GEPA (Genetic-Pareto) proposes a reflection-first prompt optimization loop where an LLM critiques its own failed outputs and turns those critiques into candidate prompt edits. A multi-objective (Pareto) search keeps several high-performing prompts to avoid premature convergence. The authors show GEPA can outperform GRPO and MIPROv2 on reasoning / fact-verification tasks while using far fewer rollouts. [1]

MBPP. MBPP (Mostly Basic Programming Problems) is another unit-test–based program synthesis benchmark, but with entry-level problems that exercise basic Python, data structures, and simple algorithms. It complements HumanEval by giving us a broader mix of task types. If GEPA helps on both HumanEval and MBPP, we can argue the gains are not tied to a single code dataset. [2]

HumanEval. HumanEval is an execution-based code benchmark: each problem comes with a function signature and unit tests, and we measure Pass@k over generated solutions. It is widely used to compare LLMs trained on code because correctness is objective and automatic. We use it to see whether GEPA's reflective edits help on short, self-contained coding tasks. [3]

HoVer. HoVer is a multi-hop fact verification dataset where a claim must be supported or refuted using evidence drawn from multiple Wikipedia pages. Because many examples require 3–4 hops and cross-sentence reasoning, performance drops sharply for models that only do shallow retrieval. It's a good testbed for GEPA because the original paper already reports gains on HoVer, so we can reproduce and compare. [4]

SWE-bench. SWE-bench evaluates LLMs in realistic software-engineering settings: the model must read a GitHub issue, modify a real repository, and produce a patch that applies and passes tests. This is harder than single-function code benchmarks because it involves multi-file reasoning and tool-like behavior. In our project this is the "last" milestone to see if GEPA scales to agentic / multi-prompt pipelines. [5]

Extended Project Description

The goal of this project is to test whether **reflective, language-space prompt optimization**—specifically GEPA (Genetic–Pareto prompt evolution)—can reliably improve large language model (LLM) performance on tasks that go beyond the ones evaluated in the original paper. The original GEPA work shows that a model can (i) look at its own execution trace or output, (ii) produce a natural language reflection about what went wrong, and (iii) evolve the original prompt using a multi-objective search, outperforming reinforcement-learning approaches while using up to 35× fewer rollouts. Our project extends this to two families of tasks:

- **Reasoning / verification**: we reproduce GEPA on HoVer to make sure our implementation and data formatting match the reported behavior;
- **Code / software engineering**: we will then apply the same reflective loop to execution-based code benchmarks (HumanEval, MBPP) and finally to repository-level SWE-bench, which is considerably harder than the datasets in the GEPA paper.

A small extension that is *specific to our framework* is that we also experiment with a **few-shot–aware GEPA variant**: during training, for each HoVer example we generate and attach a small set of in-context demonstrations (3 shots) and let the reflective LLM optimize *under that richer input.* At test time we can then compare (i) pure zero-shot GEPA, (ii) GEPA with example-tied 3-shot inference, and (iii) GEPA trained with 3-shot conditioning. This few-shot conditioning inside the GEPA loop was *not* part of the original paper, and it is the piece we are probing in our intermediate results.

The high-level pipeline is:

1. Run a *task LLM* (here: **gpt-4.1-mini**) with a seed prompt on a train subset and collect success / failure signals.
2. Feed those traces to a *reflective LLM* (here: **gpt-5**) that generates critique / edit proposals.
3. Use GEPA's Pareto-style search to maintain several improved prompts instead of a single one.
4. Re-evaluate the improved prompts on a validation split and pick the best one(s) for test.

For this check-in, we ran the entire loop on HoVer with a small but realistic budget: train set = 100, validation set = 30, test set = 500, maximum metric calls = 200, and reflection mini-batch size = 5. This gives us enough signal to compare seed vs. GEPA-optimized prompts, and to see how few-shot conditioning interacts with HoVer-style inputs.

Evaluation / Analysis Plan Our evaluation plan is intentionally aligned with community-standard benchmarks so results are comparable:

- **HoVer**: accuracy on 500 claims (support / not-support). This is our reproduction / sanity benchmark.
- **HumanEval, MBPP (next milestone)**: Pass@k using unit tests; here we will also log compile / runtime errors so the reflector has richer feedback.
- **SWE-bench (later milestone)**: percentage of issues resolved after patch application and tests.

The analysis we will do at each stage:

1. Compare seed prompt vs. GEPA-optimized prompt (same inference setting).
2. Compare zero-shot inference vs. 3-shot (example-tied) inference.
3. Compare standard GEPA vs. our few-shot–aware GEPA variant to see whether conditioning the optimizer on demonstrations actually transfers to test-time.

Intermediate Results

We ran HoVer twice on the same 500-example test split, once in a pure **zero-shot** regime and once in an **FS-style** regime (3-shot inference tied to the example). In the zero-shot run, the seed prompt with no training reached 62.8% accuracy. After we applied GEPA—i.e., GEPA-optimized training on zero-shot outputs, and then zero-shot inference on the improved prompt—accuracy increased to 65.4%, a +2.6 point absolute gain, showing that reflective prompt evolution helps even without in-context examples. A third variant, where we trained per-example with 3 generated shots and GEPA but still did zero-shot inference, dropped to 62.2%, suggesting that the current automatic few-shot construction for HoVer is not aligned with the inference condition.

In the FS-style run, where inference *does* receive 3 example-specific shots, absolute numbers are higher across the board. The seed prompt (no training) with 3-shot inference reached 65.4%; replacing it with the GEPA-optimized prompt (trained zero-shot, evaluated with 3-shot inference) improved accuracy to 69.2%, which is our best result so far. This also shows that *adding few-shot inference on top of a GEPA-optimized* *prompt* is strictly better than running the same GEPA prompt in zero-shot mode (GEPA zero-shot was 65.4%), so the optimization and the in-context demonstrations are complementary. When we also made training few-shot–aware (GEPA trained with 3-shot conditioning) and then evaluated with 3 shots, accuracy dropped to 64.0%. Across both runs the pattern is consistent: **GEPA > seed** in the same inference mode, **GEPA + 3-shot inference > GEPA zero-shot**, and **the current few-shot–optimized training recipe is weaker** and needs rebalancing before we apply this pipeline to HumanEval and MBPP.

Variant	Training setup	Inference setup	Accuracy
Zero-shot run			
Baseline	Zero training (seed prompt)	Zero-shot on seed prompt	62.8%
GEPA	GEPA-optimized training (zero-shot)	Zero-shot on trained prompt	65.4%
Few-shot opt.	Per-example 3-shot + GEPA training	Zero-shot on trained prompt	62.2%
FS-style run			
Baseline	Zero training (seed prompt)	3-shot inference (example-specific)	65.4%
GEPA	GEPA-optimized training (zero-shot)	3-shot inference (example-specific)	69.2%
Few-shot opt.	Per-example 3-shot + GEPA training	3-shot inference (example-specific)	64.0%

Table 1: HoVer intermediate results on 500 test examples under two inference regimes. GEPA improves over the seed in both zero-shot and 3-shot settings; the current few-shot–optimized training variant underperforms.

Milestones and Current Progress

Milestone 1: Reproduce GEPA on an NLP/verification task (HoVer) with our own codebase and small data splits.

Milestone 2: Port the same pipeline to execution-based code benchmarks (HumanEval, MBPP) and verify that the reflector can use tracebacks as feedback.

Milestone 3: Compose GEPA with a multi-prompt / multi-agent setup for SWE-bench (planner $\rightarrow$ coder $\rightarrow$ debugger) and optimize prompts across the pipeline.

For this intermediate check-in, **Milestone 1 is achieved**. We have:

- a working training script (`train_hover.py`) that sets up GEPA for HoVer with the budget above,
- a structured evaluation script (`evaluate_test_set.py`) that runs seed, GEPA-optimized, and few-shot–optimized prompts on the same 500-example test split.

Team Contribution Plan

- **Literature & related work: Shravan Khunti** and **Amaan Mansuri** reviewed GEPA / prompt-optimization work, the HoVer paper, and the code-benchmark lines (HumanEval, MBPP, SWE-bench), and kept the literature section aligned with what we were actually running.
- **Modeling / coding: Harshit Bhargava** and **Ninad Chaudhari** set up and ran the GEPA loop with the chosen models (task LLM = gpt-4.1-mini, reflective LLM = gpt-5) under the current budget (train = 100, val = 30, test = 500, max metric calls = 200, reflection batch = 5), and produced the zero-shot and FS-style runs reported above.
- **Evaluation & writeup integration: Vishwa Raval** consolidated the run outputs (zero-shot vs. 3-shot), organized them into the intermediate-results table, checked that the prompts in the Appendix matched the experiments, and integrated the findings into this check-in writeup.

References

[1] A. Agrawal et al. Gepa: Reflective prompt evolution can outperform reinforcement learning. *arXiv preprint arXiv:2507.19457*, 2025.

[2] Jacob Austin et al. Program synthesis with large language models. In *NeurIPS Datasets and Benchmarks*, 2021.

[3] Mark Chen et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[4] Zhengnan Jiang et al. Hover: A dataset for many-hop fact extraction and claim verification. In *Findings of EMNLP*, 2020.

[5] C. Jimenez et al. Swe-bench: Can language models resolve real-world github issues? In *Proceedings of ICLR*, 2024.

APPENDIX: Prompts

Prompt 1: Training setup: Zero training (seed prompt)

```
Given a claim and an evidence Answer SUPPORTED or NOT_SUPPORTED.
```

Prompt 2: Training setup: GEPA-optimized training (zero-shot)

```
Task: Given a Claim and a set of Context statements, output exactly one label: SUPPORTED
or NOT_SUPPORTED.

How to decide:
- Use only the provided Context. Do not use outside knowledge or assumptions.
- Decompose the claim into its atomic facts. For SUPPORTED, every part must be explicitly
supported by the Context (directly or by safely combining multiple Context lines). If any
part is missing, not stated, or contradicted, answer NOT_SUPPORTED.
- Safe combination: You may link facts across Context lines through a clearly identified
common entity (e.g., \the team he joined in 2006" = West Ham United in one line; \the club
relocated in 2016" in another → SUPPORTED).
- Do not accept added qualifiers or specifics unless they are explicitly stated:
  - Time qualifiers (e.g., \on the morning of...") must be explicitly present. If only the
  date is given, time-of-day is NOT_SUPPORTED.
  - Extra roles/titles (e.g., calling someone a \detective" when Context lists other
  occupations) make the claim NOT_SUPPORTED.
  - Causal/attribution claims (e.g., \was the inspiration for...") must be explicitly in
  Context.
  - Descriptors like \request-stop" or \commonly used" must be directly supported; do not
  infer them from related entities unless explicitly tied.
- You may treat \starred/appeared/acted in" as satisfied if the Context shows the person
had a role in the film, unless the Context explicitly contradicts it.
- If the claim stitches together multiple facts but the relationship between them is not
clearly established in Context (e.g., assuming a connection implies completion date or
usage pattern), answer NOT_SUPPORTED.
- If any ambiguity remains about an essential part of the claim, answer NOT_SUPPORTED.
Output format:
- Return exactly one of: SUPPORTED or NOT_SUPPORTED
- No explanations, punctuation, or extra text.
```

Prompt 3: Training setup: Per-example 3-shot + GEPA training

```
Task: Given a Claim and a Context, output a single label: SUPPORTED or NOT_SUPPORTED.

Input format:
- Claim: <text>
- Context: <text>

Output format:
- Exactly one of: SUPPORTED or NOT_SUPPORTED
- Do not prepend "Label:", do not add punctuation or extra text.

Decision rule (optimized for short, synthetic texts with a small vocabulary such as: fox,
dog, brown, quick, lazy, ai, model, verify, evidence, claim, jumps, data, over):
1) Normalize:
   - Lowercase both Claim and Context.
   - Remove punctuation.
2) Token handling:
   - Tokenize into words.
   - For the Claim, keep unique content words; ignore duplicates in the Claim (e.g., "ai
   ai" counts once).
   - You may ignore common stopwords (e.g., the, a, an, is, are, was, were, be, been,
   being, to, of, in, on, at, by, for, with, and, or, but, if, then, so, than, that, this,
   these, those, as, from). Treat task-specific words (ai, verify, evidence, claim, data,
   model, fox, dog, quick, brown, lazy, jumps, over) as content, not stopwords.
3) Overlap check (bag-of-words entailment):
   - If all Claim content words appear at least once in the Context, output SUPPORTED.
   - Otherwise, if at least 60% of the Claim's content words appear in the Context and
   there are at least 3 overlapping content words, output SUPPORTED.
   - Otherwise, output NOT_SUPPORTED.
   - You do not need to match duplicate counts; one occurrence in Context is sufficient.
4) Negation/contradiction:
   - If the Context clearly negates the Claim (e.g., contains direct negation like "not",
   "no", "never", "without" tied to the key claim content), prefer NOT_SUPPORTED even if
   overlap is high. This is rare in this task.
5) Keep it simple:
   - Do not perform deep reasoning or seek synonyms; exact token overlap drives the
   decision.
   - In ambiguous cases without substantial overlap, choose NOT_SUPPORTED.

Important:
- Output only SUPPORTED or NOT_SUPPORTED on a single line.
- Be concise; do not include explanations.
```