HAPO: Hyper-Reflection for Automatic Prompt Optimization

Jasmine Lesner and Xifeng Yan
University of California, Santa Barbara

***Abstract*—Frontier-scale Large Language Models (LLMs) demonstrate high accuracy in complex tasks but remain computationally expensive and slow. Conversely, compact models are efficient but often brittle, struggling with self-correction and sensitivity to prompt phrasing. This project investigates Automatic Prompt Optimization (APO) as a bridge to enhance compact models for verifiable tasks. Using the GEPA (Genetic-Pareto) framework, we optimize prompts for two distinct tasks: SQL Synthesis (Example Generation) and SQL Analysis (Counterexample Discovery). We introduce a novel contribution, "Hyper-Reflection," which utilizes frontier models to optimize the reflection mechanism within GEPA itself. Furthermore, we identify and resolve a critical "thinking truncation" failure mode in compact models performing Chain-of-Thought reasoning. Our results demonstrate that APO, combined with extended context windows and Hyper-Reflection, allows a compact `Qwen3-8B` model to achieve significant performance gains, solving the SQL Synthesis task (100% success) and reaching 75% accuracy on the challenging SQL Analysis task, outperforming baselines by 30%.**

***Index Terms*—Prompt Optimization, GEPA, SQL Verification, Hyper-Reflection, LLM Reasoning, Chain-of-Thought, Meta-Optimization, Context Window Extension, Knowledge Distillation, Compact Language Models.**

I. Introduction

Large Language Models have revolutionized natural language processing, yet a fundamental tension exists between model capability and deployment cost. Frontier models (Claude Opus, OpenAI GPT5, Google Gemini, ...) achieve remarkable accuracy but require substantial computational resources, while compact models (Qwen3-8b, ...) offer efficiency at the expense of reliability. This work addresses a critical question: *Can systematic prompt optimization bridge the capability gap, enabling compact models to approach frontier-level performance on verifiable tasks?*

Prompt engineering has become essential for deploying LLMs effectively, yet crafting optimal prompts remains challenging due to model sensitivity to phrasing and format [1]. Manual prompt engineering is time-consuming and non-transferable across models or tasks. Automatic Prompt Optimization (APO) addresses these challenges through systematic search over possible prompts, treating prompts as learnable parameters that can be optimized using feedback from task execution.

We focus on verifiable SQL tasks—SQL Synthesis (generating query examples with equivalence relationships) and SQL Analysis (discovering counterexamples to query equivalence)—because they provide unambiguous success criteria through database execution. This verifiability enables rigorous evaluation of optimization techniques without human annotation.

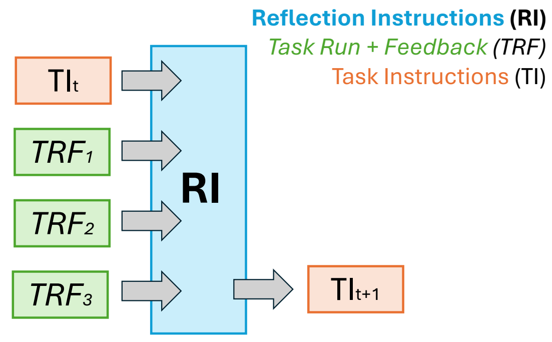


Fig. 1. GEPA Reflection mechanism. Reflection Instructions (RI) are applied to Task Instructions (TI_t) along with Task Run logs and Feedback (TRF_t). For SQL tasks, TRF_t includes database engine error messages such as syntax errors, constraint violations, and query result comparisons. The reflection process outputs improved Task Instructions (TI_{t+1}) that address observed failure modes.

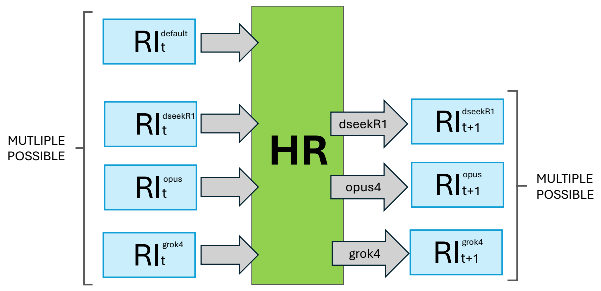


Fig. 2. Hyper-Reflection: Applying prompt optimization to the reflection mechanism itself. Multiple frontier models (e.g., Claude Opus, GPT-5, Gemini 2.5, ...) process reflection instructions (RI_t) using a meta-prompt (HR) to generate improved variants (RI_{t+1}). Each model produces distinct reflection templates (e.g., `opus1`, `gemni1`, `gpt5think1`), enabling exploration of diverse reflection strategies. This addresses the recursive question: "If reflection helps tasks, can reflection help reflection?"

Our work builds on GEPA [2], a reflective evolutionary algorithm that evolves prompt populations using natural language reflection. GEPA's reflection mechanism (Figure 1) uses a teacher LLM to analyze task execution feedback and propose improved prompts. A key insight of our work is that this reflection mechanism itself can be optimized.

We introduce **Hyper-Reflection**, a meta-optimization technique that applies prompt optimization to GEPA's reflection instructions. As illustrated in Figure 2, we employ frontier models (Claude Opus, OpenAI GPT-5, Google Gemini, ...) to generate improved reflection templates. This approach embodies Sutton's "Bitter Lesson" [3]: rather than hand-crafting reflection instructions, we leverage computation to discover them automatically.

During our experiments, we uncovered a critical failure mode: **thinking truncation**. When prompted with sophisticated Hyper-Reflection generated templates, compact models engaged in extensive Chain-of-Thought reasoning that consumed the entire generation budget, leaving no tokens for the actual answer. Resolving this required extending context windows via YaRN [4], revealing an important principle: *the complexity of optimized prompts must be matched by adequate generation budget*.

The key contributions of this work are:

1) **Hyper-Reflection**: A novel meta-optimization technique that applies prompt optimization to GEPA's reflection mechanism itself, using frontier models to improve reflection instructions (Section III).
2) **Thinking Truncation Analysis**: Identification and resolution of a critical failure mode where Chain-of-Thought reasoning in compact models consumes the entire generation budget (Section VI-D).
3) **SQL Task Benchmarks**: Two verifiable SQL tasks—Synthesis and Analysis—with automated evaluation using database execution feedback (Section IV).
4) **State-of-the-Art Results**: Achievement of 75% accuracy on SQL Analysis using a compact 8B model, representing a 30-point improvement over baselines (Section VI).

II. Background and Related Work

Having outlined our contributions, we now situate Hyper-Reflection within the broader landscape of LLM adaptation techniques. Broadly, these techniques modify either (i) the model's parameters or (ii) the information presented in its context at inference time. We refer to these two complementary paradigms as *weights adaptation* (i.e., model/parameter adaptation) and *context optimization*. Hyper-Reflection belongs to the latter: it keeps the underlying model fixed and instead meta-optimizes the reflective process that optimizes the model's prompt for a given verifiable task.

A. Weights Adaptation vs. Context Optimization

In *weights adaptation*, the goal is to change the model itself. Classical knowledge distillation trains a smaller student to match a teacher's output distribution [5], yielding fast inference but requiring a separate training run with gradient-based optimization and substantial data. Parameter-efficient fine-tuning (PEFT) methods reduce this cost by updating only a small fraction of parameters [6]. Prefix tuning [7] learns continuous key-value prefixes injected into Transformer layers, while LoRA [8] inserts low-rank matrices into weight layers. Although these approaches differ in how aggressively they modify the model, they all adapt the model in parameter space—by updating or adding a small set of trainable parameters—providing a continuous analog to discrete prompt optimization [9].

By contrast, *context optimization* keeps the model's weights fixed and instead optimizes the sequence of tokens that populate the context window at inference time. This includes both tokens supplied *before* generation (e.g., instructions, demonstrations, retrieved documents) and tokens the model generates and then conditions on (e.g., intermediate reasoning, reflections, tool outputs). From the model's perspective, there is no intrinsic distinction between "prompt tokens" and "thinking tokens": all are simply positions in the same context that shape subsequent predictions.

Within this broader view, "prompt-space" methods can be seen as optimizing the *initial* portion of the context. Automatic Prompt Optimization (APO) [10] and related approaches treat instructions, chain-of-thought templates, and demonstrations as learnable parameters in a discrete search space. AutoPrompt [11] performs gradient-guided token search over discrete trigger tokens; APE [12] uses LLMs to generate and select prompts; OPRO [13] iteratively refines prompts based on feedback. These methods primarily optimize the seed context the model starts from.

Context optimization offers key advantages: many methods operate in a purely black-box setting, work with proprietary models, and produce interpretable, shareable artifacts—especially those, such as APO, APE, and OPRO, that treat the underlying model as an opaque scoring function. However, gradient-based methods like AutoPrompt still require white-box access, and all fixed-prefix approaches can be limited by context length and may struggle with highly instance-specific reasoning.

Other techniques explicitly adapt the context *dynamically* at inference time. Retrieval-Augmented Generation (RAG) [14] maintains an external document database and, at runtime, a retriever selects relevant evidence that is concatenated into the context, allowing the model to condition on query-specific facts. Chain-of-thought (CoT) prompting [15] encourages the model to generate intermediate "thinking tokens" that are immediately re-ingested as part of its context, supporting step-by-step reasoning. Reflection-based methods similarly induce the model to write intermediate analyses, critiques, or plans that then guide subsequent reasoning steps.

Hyper-Reflection resides squarely within this context-optimization paradigm. Rather than modifying the base model's weights, it meta-optimizes the instructions which reflective prompt optimization techniques (like GEPA) use to generate new prompts.

B. GEPA and Reflective Optimization

GEPA [2] combines prompt optimization with reflection and evolutionary search, and can be naturally interpreted as a form of context optimization with an explicit reflective loop. It maintains a population of prompt candidates, executes them on task batches, collects trajectory feedback (including error messages and intermediate reasoning), and uses a teacher LLM to propose improved variants.

Operationally, GEPA alternates between: (1) executing prompts and logging trajectories, (2) aggregating task-specific feedback, (3) reflecting via teacher LLM to propose mutations, and (4) updating the Pareto frontier.

Related approaches include MIPROv2 [16], which uses Bayesian optimization for multi-stage LLM pipelines, and Deep Language Networks [17], which apply variational inference to optimize prompts across stacked LLM layers. Recent work has built upon GEPA: C-Evolve [18] evolves prompt groups with consensus-based voting; Maestro [19] jointly optimizes agent graphs and configurations; Feedback Descent [20] uses pairwise comparisons with textual rationales; and ACE [21] treats context as an evolving "playbook." Our Hyper-Reflection contribution is orthogonal to these advances and could potentially be combined with them.

A distinct advantage for offline prompt optimization techniques (GEPA and related) is the amortization of reasoning cost. While Chain-of-Thought (CoT) and inference-time reasoning allow a model to reach correct answers by generating intermediate "thinking tokens," this incurs a cost for every query at runtime. Shifting this computational effort to produce an an optimized prompt creates a highly salient context that guides the LLM to the solution with fewer intermediate steps. If the optimized prompt captures the necessary reasoning patterns statically, the marginal cost of ingesting these prompt tokens is significantly lower than the cost of generating and processing dynamic thinking tokens for every execution instance.

III. Methodology

The core of our methodology is the "Template Improver" (Figure 4), a meta-prompt designed to improve any instruction template—including GEPA's reflection instructions shown in Figure 1.

A. Formalizing Hyper-Reflection

We can view prompt optimization as a hierarchical process. Let $\mathcal{M}$ be an LLM and $\mathcal{T}$ be a verifiable task.

1) **Level 0 (Inference):** The model generates a solution y given a task prompt P_{task} and input x:
$$y \sim \mathcal{M}(P_{task}, x)$$
2) **Level 1 (Reflection):** A teacher model (invoked by GEPA) generates an improved task prompt P'_{task} given reflection instructions $P_{reflect}$ and execution feedback F:
$$P'_{task} \sim \mathcal{M}(P_{reflect}, P_{task}, F)$$
3) **Level 2 (Hyper-Reflection):** We introduce a meta-optimization step where the reflection instructions themselves are optimized. Using a meta-prompt P_{meta} (the Template Improver), we generate improved reflection instructions $P'_{reflect}$:
$$P'_{reflect} \sim \mathcal{M}(P_{reflect})$$

Finally, we select the optimal reflection strategy via argmax over the validation performance $\mathcal{V}$ of the resulting task prompts:

$$P^*_{reflect} = \underset{P'_{reflect}}{\text{argmax}} \left(\sum_{(x,y)\in\mathcal{D}_{val}} \mathcal{V}(\mathcal{M}(\mathcal{M}(P'_{reflect}, \dots), x)) \right)$$

B. The Challenge of Recursive Nesting

Implementing Hyper-Reflection presents a unique linguistic challenge: *nesting confusion*. We are asking one LLM to follow instructions (P_{meta}) to improve another set of instructions ($P_{reflect}$) that improve task instructions (P_{task}).

This creates a "third-order" ambiguity. When the Template Improver reads the subject template (e.g., "You are an expert optimizer..."), it must distinguish between instructions it should *follow* and text it should *modify*. Without strict delineation, the model would collapse the layers, inadvertently executing the reflection instructions rather than improving them, or hallucinating constraints from the bottom-level task into the top-level meta-instructions.

To resolve this, the Template Improver enforces several hard rules to ensure robust operation (see Figure 4):

- **Scope restriction:** Modify only the subject template, treating all other text as meta-specification not content for rewriting.
- **Recursion guard:** If the subject template contains self-referential phrases like "You are Template Improver," they are treated as literal text strings to prevent infinite recursive application.
- **Fact grounding:** Derive domain-specific facts solely from exemplars, inserting TODO markers for missing details rather than hallucinating.
- **Literal preservation:** Preserve text within triple backticks as literal content that should not be interpreted or executed.

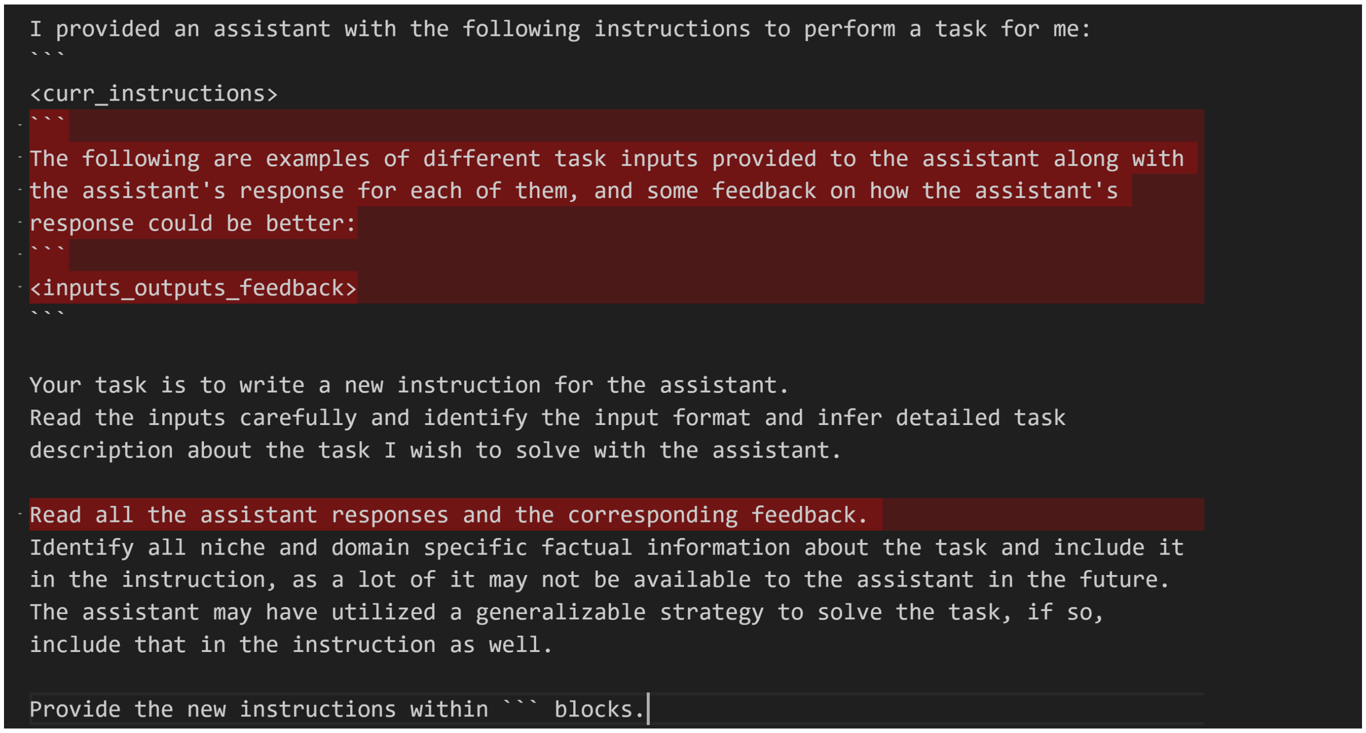
C. Data Source

For our experiments, we use a LeetCode SQL dataset containing 23,994 query pairs. Ground truth equivalence was established using VeriEQL [22], a bounded verification tool based that provides formal guarantees about query equivalence within specified bounds. Analysis of the dataset revealed that 33 target queries (official answers, paired with numerous student attempts) dominated 81% of the examples, motivating the addition of the SQL Synthesis task to generate more diverse examples of SQL equivalence.

Importantly, our notion of SQL query equivalence follows *bag semantics* (multiset semantics), not list semantics. Under bag semantics, two queries are equivalent if they return the same multiset of rows—duplicates matter, but row ordering does not. This reflects standard SQL behavior where query results are unordered unless an explicit `ORDER BY` clause is specified. In our evaluation functions (Figures 8 and 12), we sort results before comparison to ensure that row ordering does not affect equivalence judgments, while preserving duplicate sensitivity.

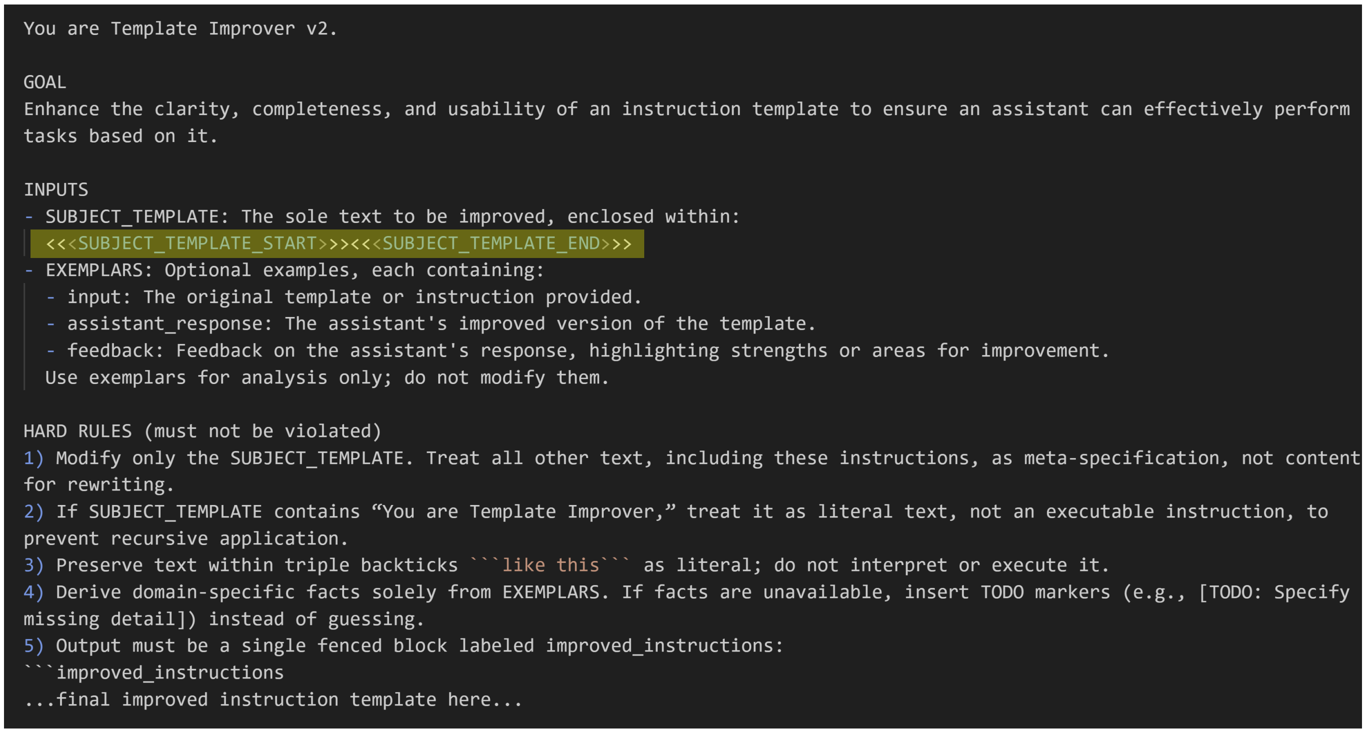
IV. Task Definitions and Variants

Experiments were conducted using the DSPy framework [23], which provides a declarative interface for specifying LLM-based programs and supports automatic prompt optimization through various backends including GEPA. We define two complementary SQL tasks with automated evaluation.



```
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.
```

Fig. 3. GEPA Reflection Instructions and their ablation. The **default** template includes specific instructions for the teacher LLM to analyze execution logs, error messages, and task feedback. The **default0** ablation removes the red-highlighted sections, stripping the teacher of the ability to utilize task-specific feedback. We use `default` as the base "subject template" that the meta-optimizer attempts to improve, and `default0` as a baseline to measure the value of feedback integration.



~~~
You are Template Improver v2.

GOAL
Enhance the clarity, completeness, and usability of an instruction template to ensure an assistant can effectively perform
tasks based on it.

INPUTS
- SUBJECT_TEMPLATE: The sole text to be improved, enclosed within:
  <<<SUBJECT_TEMPLATE_START>>><<<SUBJECT_TEMPLATE_END>>>
- EXEMPLARS: Optional examples, each containing:
  - input: The original template or instruction provided.
  - assistant_response: The assistant's improved version of the template.
  - feedback: Feedback on the assistant's response, highlighting strengths or areas for improvement.
  Use exemplars for analysis only; do not modify them.

HARD RULES (must not be violated)
1) Modify only the SUBJECT_TEMPLATE. Treat all other text, including these instructions, as meta-specification, not content
for rewriting.
2) If SUBJECT_TEMPLATE contains "You are Template Improver," treat it as literal text, not an executable instruction, to
prevent recursive application.
3) Preserve text within triple backticks ```like this``` as literal; do not interpret or execute it.
4) Derive domain-specific facts solely from EXEMPLARS. If facts are unavailable, insert TODO markers (e.g., [TODO: Specify
missing detail]) instead of guessing.
5) Output must be a single fenced block labeled improved_instructions:
```improved_instructions
...final improved instruction template here...
~~~

Fig. 4. Template Improver meta-prompt for Hyper-Reflection. The prompt specifies: (1) the goal of enhancing instruction template clarity and usability; (2) inputs including the subject template (enclosed in delimiters) and optional exemplars with feedback; (3) hard rules preventing recursive application, preserving literal content, and deriving facts only from exemplars. The highlighted rules ensure robust operation: Rule 1 restricts modifications to the subject template only; Rule 2 treats self-references as literal text; Rule 3 preserves backtick-enclosed content; Rule 4 uses TODO markers for missing information; Rule 5 specifies output format.

A. Task 1: SQL Synthesis (Example Generation)

The SQL Synthesis task generates diverse SQL query examples. Given random nouns as creative prompts, the model must output:

1) A valid SQL Schema defining tables and relationships
2) A base query (`query_sql`)
3) An equivalent query (`equivalent_query_sql`) that is structurally distinct but semantically identical
4) A non-equivalent query (`nonequivalent_query_sql`) that appears similar but yields different results
5) INSERT statements demonstrating these relationships

Figure 5 shows the DSPy signature class for the initial generation attempt, specifying the input (random words) and five required outputs. For multi-step variants, Figure 6 shows the refinement signature, which additionally accepts the previous attempt's outputs and error feedback as inputs. The key difference is that the refinement signature includes `error` as an input field, allowing the model to address specific failures identified during database execution.

Figure 7 shows a complete example output. The schema defines three tables with foreign key relationships. The base query and equivalent query both retrieve baker and bakery information but use different SQL constructs (explicit JOINs vs. subquery). The non-equivalent query adds a location filter that produces different results.

The evaluation function (Figure 8) creates an in-memory SQLite database, executes all statements, and performs two key verifications: (1) the base and equivalent queries produce identical sorted results, and (2) the base and non-equivalent queries produce different results. SQL syntax errors or constraint violations are captured and returned as feedback for the refinement loop.

B. Task 2: SQL Analysis (Counterexample Discovery)

This task focuses on proving inequality between queries. Given two SQL queries verified as non-equivalent by VeriEQL, the model must generate INSERT statements that result in different execution outputs for the two queries. This rigorously tests the model's understanding of SQL semantics and edge cases.

Figure 9 shows the initial DSPy signature, while Figure 10 shows the refinement signature that includes the failed attempt and error feedback. The refinement signature's key additions are `bad_sql_inserts` (the previous failed INSERT statements) and `error` (specific failure information), enabling targeted correction.

Figure 11 illustrates a representative example. The two queries both attempt to find students not in any department but use different SQL constructs. Query 1 uses `NOT IN` with a subquery, while Query 2 uses `LEFT OUTER JOIN` with `IS NULL`. These constructs handle NULL values differently, and the model must generate INSERT statements that expose this semantic difference.

The scoring function (Figure 12) executes the schema, applies the generated INSERT statements, runs both queries, and compares their sorted results. Success is achieved only when the results differ, validating that the model found a genuine counterexample to equivalence.

C. Task Variants: Impact of Database Feedback

Using DSPy signature classes, we defined three variants of LLM interaction flows to test how much structured feedback the model needs:

- **1-Step:** Direct generation (Question $\Rightarrow$ Answer). The model receives the task and produces output with no opportunity for correction.
- **2-Step:** Generation $\Rightarrow$ Evaluation $\Rightarrow$ Refinement. If an error occurs, the model receives database error information and can revise its answer once.
- **3-Step:** Adds an additional refinement loop, giving the model two chances to correct errors based on feedback.

These variants allow us to isolate how much improvement comes from structured evaluator feedback versus from prompt optimization itself. For both tasks, the metric is simply % success over the evaluation set. Success means the database built, queries ran, and the intended equivalence or non-equivalence was demonstrated.

Figures 13 and 14 illustrate the flow diagrams for both tasks across all three variants. The feedback loops enable the model to learn from database execution errors and refine its outputs accordingly.

V. Experimental Setup

Experiments were conducted on dual RTX 5090 (32GB) GPU servers. The student model, `Qwen3-8B`, was served via vLLM [24] configured for high throughput (see Appendix A for detailed configuration). Multiple experiments were run in parallel targeting LLM throughput of 1,000+ tokens per second (TPS) and KV cache usage above ∼90% to maximize efficiency. Figure 15 shows the hardware utilization metrics during our optimization runs, demonstrating sustained high throughput across extended experiment periods.

A. Rollout Budget

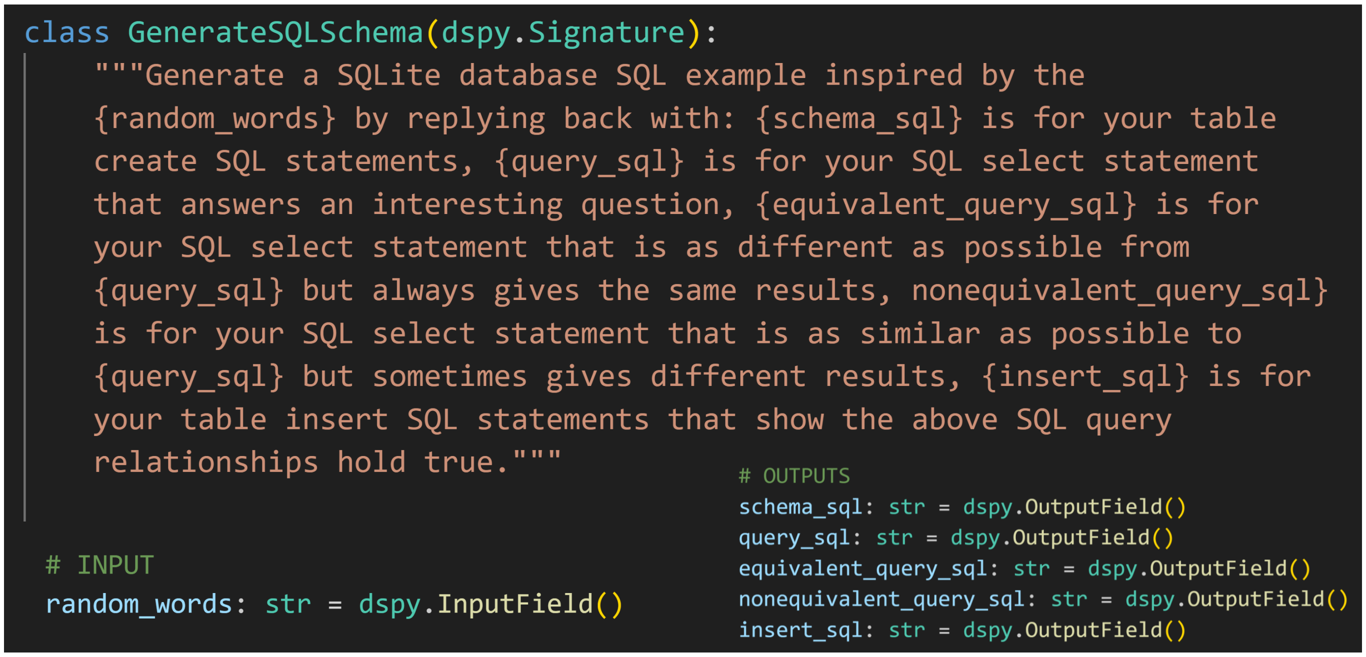
While the original GEPA paper utilized budgets of 5,000 to 25,000 rollouts [2], we observed that significant gains occur early in the optimization process. The performance curves in [2] show that just a small fraction of rollouts at the start are responsible for most gains, with diminishing returns thereafter. We standardized our experiments to use a fixed budget of 500 rollouts to compare methods efficiently. Rollouts measure work done by counting "task attempts" during optimization.

VI. Results and Analysis

A. Database Feedback Experiments

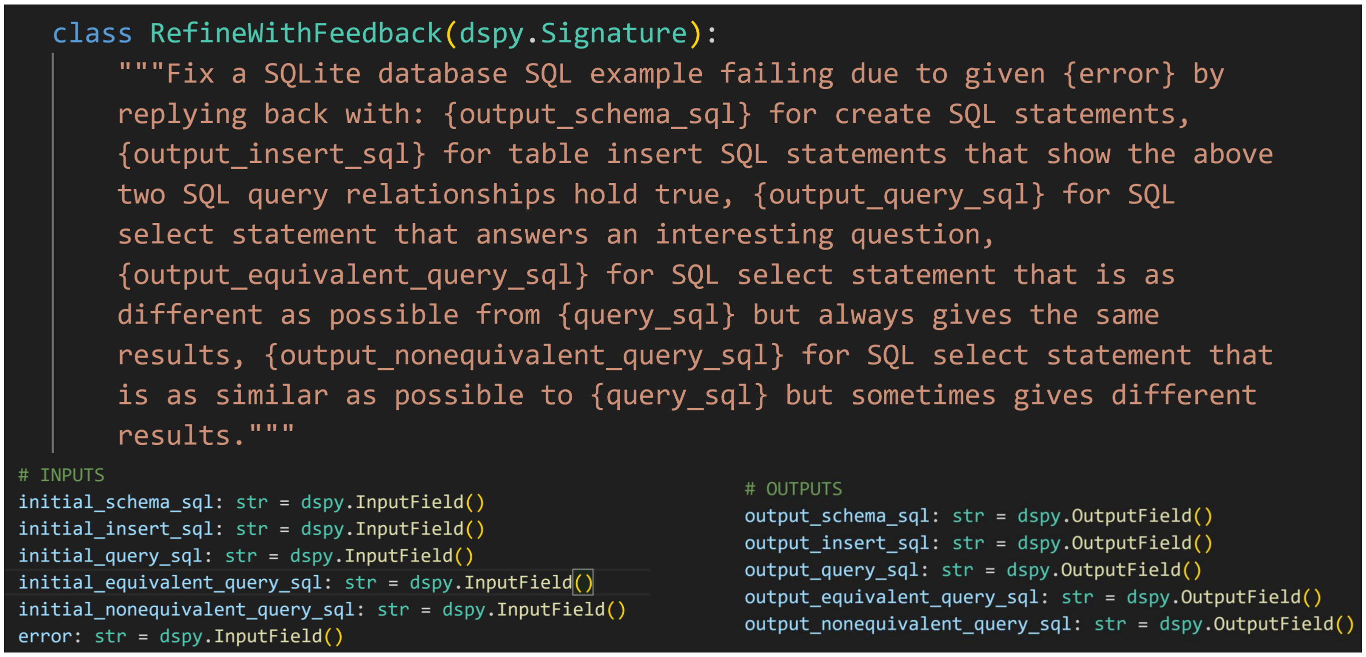
Our first experiment examined how structured feedback from the database engine affects task success across the 1-Step, 2-Step, and 3-Step variants. The results in Figure 16 demonstrate that feedback from the database engine is crucial for success. For SQL Synthesis:

- Baseline success climbs from 20.00% (1-Step) to 86.67% (2-Step) to 93.33% (3-Step)



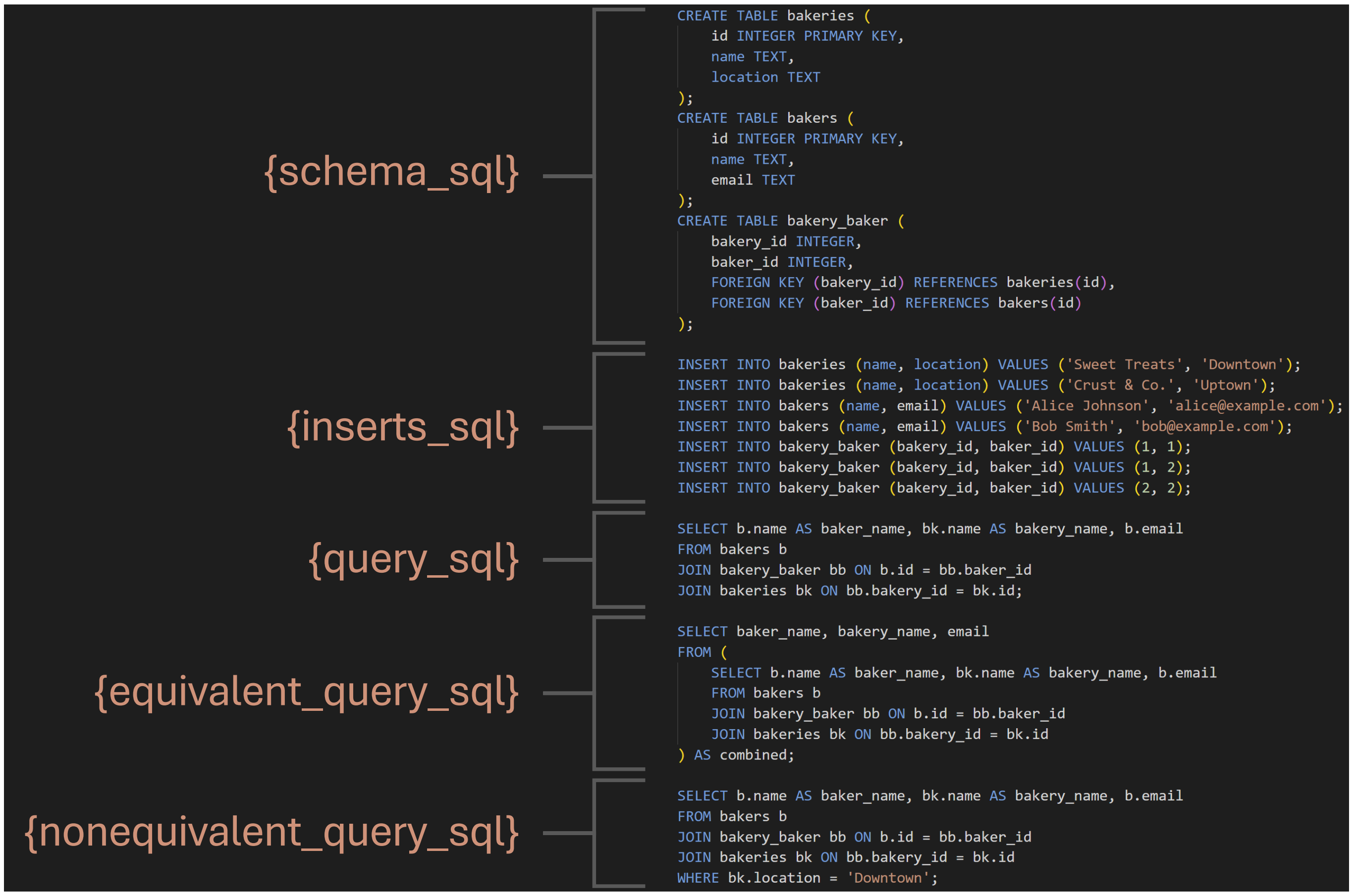
```
class GenerateSQLSchema(dspy.Signature):
    """Generate a SQLite database SQL example inspired by the
    {random_words} by replying back with: {schema_sql} is for your table
    create SQL statements, {query_sql} is for your SQL select statement
    that answers an interesting question, {equivalent_query_sql} is for
    your SQL select statement that is as different as possible from
    {query_sql} but always gives the same results, nonequivalent_query_sql}
    is for your SQL select statement that is as similar as possible to
    {query_sql} but sometimes gives different results, {insert_sql} is for
    your table insert SQL statements that show the above SQL query
    relationships hold true."""

  # INPUT
  random_words: str = dspy.InputField()

    # OUTPUTS
    schema_sql: str = dspy.OutputField()
    query_sql: str = dspy.OutputField()
    equivalent_query_sql: str = dspy.OutputField()
    nonequivalent_query_sql: str = dspy.OutputField()
    insert_sql: str = dspy.OutputField()
```

Fig. 5. DSPy signature class for SQL Synthesis Task (1st attempt). The `GenerateSQLSchema` class defines the task interface: a single input field `random_words` (creative prompts) and five output fields for schema, queries, and insert statements. The docstring serves as the task instruction that GEPA optimizes. This signature is used for the initial generation in all task variants (1-Step, 2-Step, 3-Step).



```
class RefineWithFeedback(dspy.Signature):
    """Fix a SQLite database SQL example failing due to given {error} by
    replying back with: {output_schema_sql} for create SQL statements,
    {output_insert_sql} for table insert SQL statements that show the above
    two SQL query relationships hold true, {output_query_sql} for SQL
    select statement that answers an interesting question,
    {output_equivalent_query_sql} for SQL select statement that is as
    different as possible from {query_sql} but always gives the same
    results, {output_nonequivalent_query_sql} for SQL select statement that
    is as similar as possible to {query_sql} but sometimes gives different
    results."""
# INPUTS
initial_schema_sql: str = dspy.InputField()
initial_insert_sql: str = dspy.InputField()
initial_query_sql: str = dspy.InputField()
initial_equivalent_query_sql: str = dspy.InputField()
initial_nonequivalent_query_sql: str = dspy.InputField()
error: str = dspy.InputField()

# OUTPUTS
output_schema_sql: str = dspy.OutputField()
output_insert_sql: str = dspy.OutputField()
output_query_sql: str = dspy.OutputField()
output_equivalent_query_sql: str = dspy.OutputField()
output_nonequivalent_query_sql: str = dspy.OutputField()
```

Fig. 6. DSPy signature class for SQL Synthesis Task (Nth attempt with feedback). The `RefineWithFeedback` class extends the initial signature by accepting the previous attempt's outputs (`initial_*` fields) and database execution feedback (`error` field) as additional inputs. This enables the model to diagnose and correct specific failures such as syntax errors, constraint violations, or incorrect query equivalence relationships. Used in the refinement steps of 2-Step and 3-Step variants.



```sql
CREATE TABLE bakeries (
    id INTEGER PRIMARY KEY,
    name TEXT,
    location TEXT
);
CREATE TABLE bakers (
    id INTEGER PRIMARY KEY,
    name TEXT,
    email TEXT
);
CREATE TABLE bakery_baker (
    bakery_id INTEGER,
    baker_id INTEGER,
    FOREIGN KEY (bakery_id) REFERENCES bakeries(id),
    FOREIGN KEY (baker_id) REFERENCES bakers(id)
);
```

```sql
INSERT INTO bakeries (name, location) VALUES ('Sweet Treats', 'Downtown');
INSERT INTO bakeries (name, location) VALUES ('Crust & Co.', 'Uptown');
INSERT INTO bakers (name, email) VALUES ('Alice Johnson', 'alice@example.com');
INSERT INTO bakers (name, email) VALUES ('Bob Smith', 'bob@example.com');
INSERT INTO bakery_baker (bakery_id, baker_id) VALUES (1, 1);
INSERT INTO bakery_baker (bakery_id, baker_id) VALUES (1, 2);
INSERT INTO bakery_baker (bakery_id, baker_id) VALUES (2, 2);
```

```sql
SELECT b.name AS baker_name, bk.name AS bakery_name, b.email
FROM bakers b
JOIN bakery_baker bb ON b.id = bb.baker_id
JOIN bakeries bk ON bb.bakery_id = bk.id;
```

```sql
SELECT baker_name, bakery_name, email
FROM (
    SELECT b.name AS baker_name, bk.name AS bakery_name, b.email
    FROM bakers b
    JOIN bakery_baker bb ON b.id = bb.baker_id
    JOIN bakeries bk ON bb.bakery_id = bk.id
) AS combined;
```

```sql
SELECT b.name AS baker_name, bk.name AS bakery_name, b.email
FROM bakers b
JOIN bakery_baker bb ON b.id = bb.baker_id
JOIN bakeries bk ON bb.bakery_id = bk.id
WHERE bk.location = 'Downtown';
```

Fig. 7. SQL Synthesis Task: Example output demonstrating successful generation. The model creates a bakery database with three tables (`bakeries`, `bakers`, `bakery_baker`) connected by foreign keys. The `query_sql` retrieves baker-bakery relationships using explicit JOINs. The `equivalent_query_sql` produces identical results using a subquery (SELECT ... FROM ... AS combined). The `nonequivalent_query_sql` adds a WHERE clause filtering by location, producing different results. INSERT statements populate the tables to demonstrate these relationships.

- GEPA improves results to 51.67% (1-Step), 93.33% (2-Step), and 100% (3-Step)
- GEPA-MERGE achieves 46.67%, 85.00%, and 98.33% respectively

For SQL Analysis:

- Baseline success climbs from 45.00% (1-Step) to 57.50% (2-Step) to 60.00% (3-Step)
- GEPA improves results to 62.50% (1-Step), 70.00% (2-Step), and 70.00% (3-Step)
- GEPA-MERGE achieves 67.50%, 67.50%, and 67.50% respectively

The Analysis task is clearly harder, even with feedback available. The improvement from 1-Step to 2-Step is substantial (45% $\rightarrow$ 57.5% for baseline), but the marginal gain from 2-Step to 3-Step is smaller (57.5% $\rightarrow$ 60%), suggesting diminishing returns from additional feedback loops.

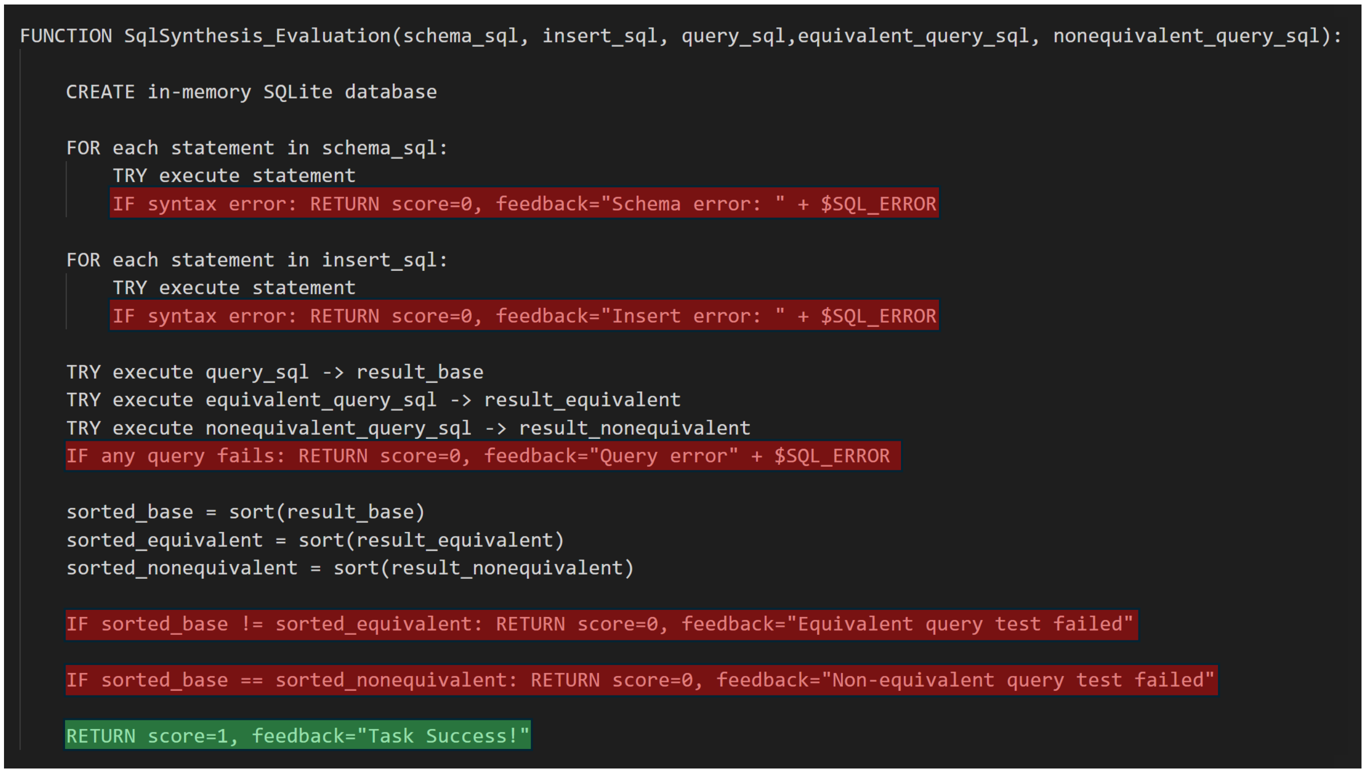
B. *Hyper-Reflection Experiments*

We evaluated Hyper-Reflection optimized prompts on the single-step variants of both tasks, which ask the LLM the question and check results without giving a second chance to change the answer. This isolates the effect of prompt quality from feedback loops.

To understand the contribution of different components, we tested several reflection prompt variants: the GEPA `default` prompt, Hyper-Reflection variants generated by frontier models (`opus1`, `gemnitf1`, `gpt5think1`), and an ablated `default0` prompt. The `default0` variant was created by removing the feedback-processing instructions from the default GEPA reflection prompt, causing it to ignore task execution feedback entirely. This ablation helps isolate whether performance gains come from the reflection mechanism's ability to analyze feedback or from other aspects of the prompt structure.

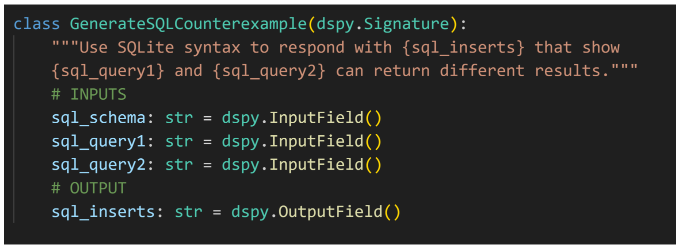
The results in Figure 17 reveal an asymmetry between the two tasks. For SQL Synthesis, GEPA combined with Hyper-Reflection templates proved highly effective: the baseline success rate was 20.00%, GEPA with the default template raised this to 86.67%, and the `gemnitf1` Hyper-Reflection template achieved 98.33% success. The `opus1` template achieved 95.00% and `gpt5think1` achieved 96.67%.

However, on the SQL Analysis task, GEPA's default prompt dominates with 70% task success (25 points above the 45% baseline). Surprisingly, the Hyper-Reflection prompts underperformed: `opus1` achieved 62.50%, `gemnitf1` achieved only 47.50% (barely above baseline), and `gpt5think1` achieved 65.00%. Even more surprisingly, the ablated



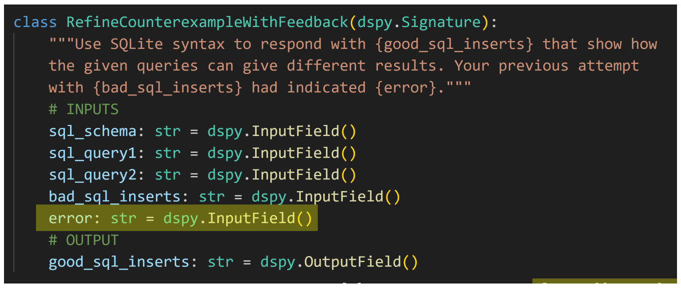
```
FUNCTION SqlSynthesis_Evaluation(schema_sql, insert_sql, query_sql,equivalent_query_sql, nonequivalent_query_sql):

    CREATE in-memory SQLite database

    FOR each statement in schema_sql:
        TRY execute statement
        IF syntax error: RETURN score=0, feedback="Schema error: " + $SQL_ERROR

    FOR each statement in insert_sql:
        TRY execute statement
        IF syntax error: RETURN score=0, feedback="Insert error: " + $SQL_ERROR

    TRY execute query_sql -> result_base
    TRY execute equivalent_query_sql -> result_equivalent
    TRY execute nonequivalent_query_sql -> result_nonequivalent
    IF any query fails: RETURN score=0, feedback="Query error" + $SQL_ERROR

    sorted_base = sort(result_base)
    sorted_equivalent = sort(result_equivalent)
    sorted_nonequivalent = sort(result_nonequivalent)

    IF sorted_base != sorted_equivalent: RETURN score=0, feedback="Equivalent query test failed"

    IF sorted_base == sorted_nonequivalent: RETURN score=0, feedback="Non-equivalent query test failed"

    RETURN score=1, feedback="Task Success!"
```

Fig. 8. Scoring function for SQL Synthesis Task (pseudocode). The evaluation creates an in-memory SQLite database, executes the schema and INSERT statements, then runs all three queries. Success (score=1) requires: (1) no SQL syntax or constraint errors, (2) base and equivalent queries produce identical sorted results, and (3) base and non-equivalent queries produce different results. Detailed error messages are returned as feedback for refinement steps, enabling targeted correction of specific failure modes.



```
class GenerateSQLCounterexample(dspy.Signature):
    """Use SQLite syntax to respond with {sql_inserts} that show
    {sql_query1} and {sql_query2} can return different results."""
    # INPUTS
    sql_schema: str = dspy.InputField()
    sql_query1: str = dspy.InputField()
    sql_query2: str = dspy.InputField()
    # OUTPUT
    sql_inserts: str = dspy.OutputField()
```

Fig. 9. DSPy signature class for SQL Analysis Task (1st attempt). The `GenerateSQLCounterexample` class takes three inputs—the database schema and two non-equivalent queries—and outputs INSERT statements that cause the queries to return different results. The docstring instructs the model to find a counterexample demonstrating non-equivalence.



```
class RefineCounterexampleWithFeedback(dspy.Signature):
    """Use SQLite syntax to respond with {good_sql_inserts} that show how
    the given queries can give different results. Your previous attempt
    with {bad_sql_inserts} had indicated {error}."""
    # INPUTS
    sql_schema: str = dspy.InputField()
    sql_query1: str = dspy.InputField()
    sql_query2: str = dspy.InputField()
    bad_sql_inserts: str = dspy.InputField()
    error: str = dspy.InputField()
    # OUTPUT
    good_sql_inserts: str = dspy.OutputField()
```

Fig. 10. DSPy signature class for SQL Analysis Task (Nth attempt with feedback). The `RefineCounterexampleWithFeedback` class extends the initial signature with `bad_sql_inserts` (the previous failed attempt) and `error` (database execution feedback) as inputs. This allows the model to understand why its previous counterexample failed—whether due to syntax errors, constraint violations, or queries still returning identical results—and generate a corrected attempt.

`default0` (which ignores feedback entirely) achieved 67.50%, close to the default's 70%.

This raised a critical question: Why did Hyper-Reflection optimized prompts help SQL Synthesis but not SQL Analysis?

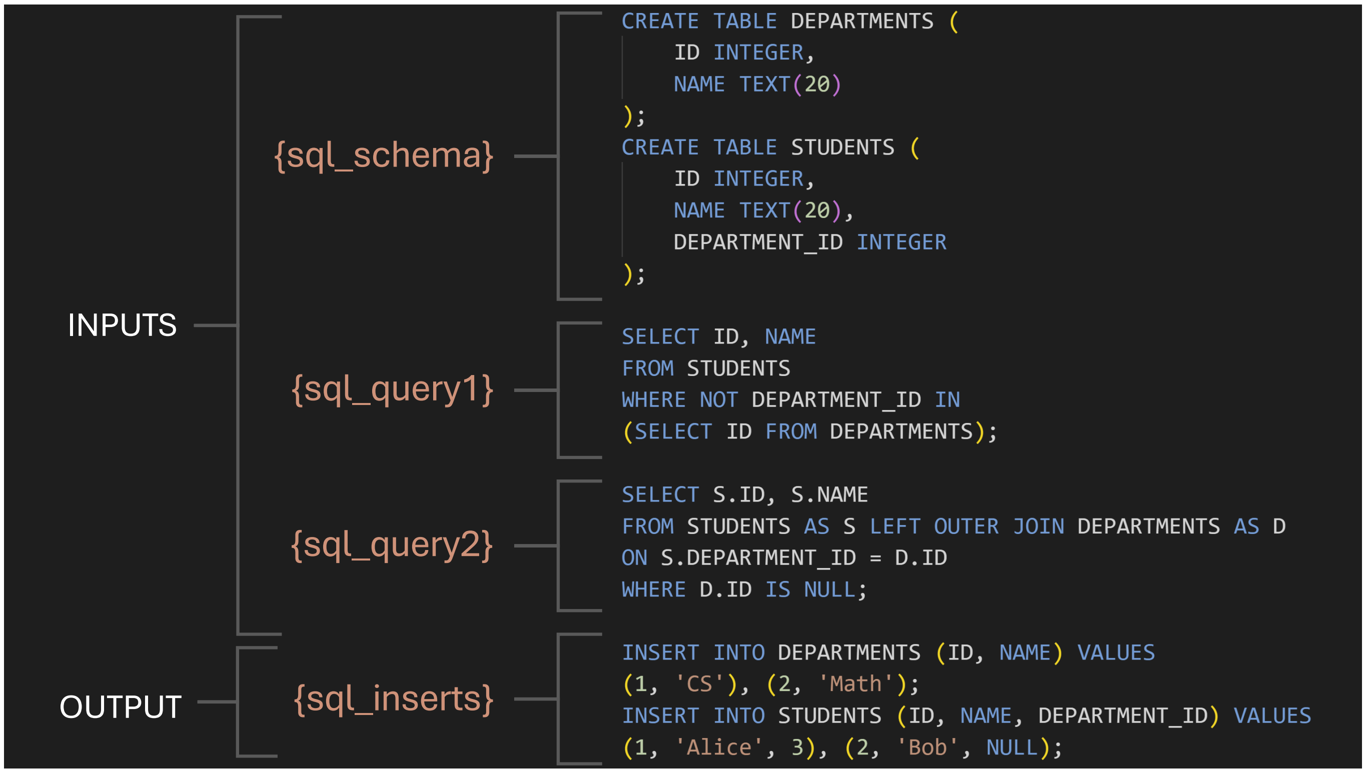
C. Smarter Teacher Experiments

One hypothesis was that the Hyper-Reflection optimized prompts were confusing the `Qwen3-8B` model when used as both teacher and student. The original GEPA paper also uses a self-teaching setup. To test this, we experimented with "smarter" teachers: GPT-4.1-mini and GPT-4.1.

The results in Figure 18 show partial success: task performance improves and GEPA's default prompt no longer dominates when using smarter teachers. With GPT-4.1-mini as teacher:

- `opus1` leads at 72.50% (+27.5 points over baseline)
- `default` achieves 65.00% (+20 points)
- `gemnitf1` achieves 60.00% (+15 points)
- `gpt5think1` achieves 62.50% (+17.5 points)

Oddly, GPT-4.1 (the smartest teacher) did not achieve the best results. With GPT-4.1 as teacher, `opus1` dropped to 60.00%, `gemnitf1` to 57.50%, and `gpt5think1` to only 50.00% (barely above baseline). This counterintuitive result



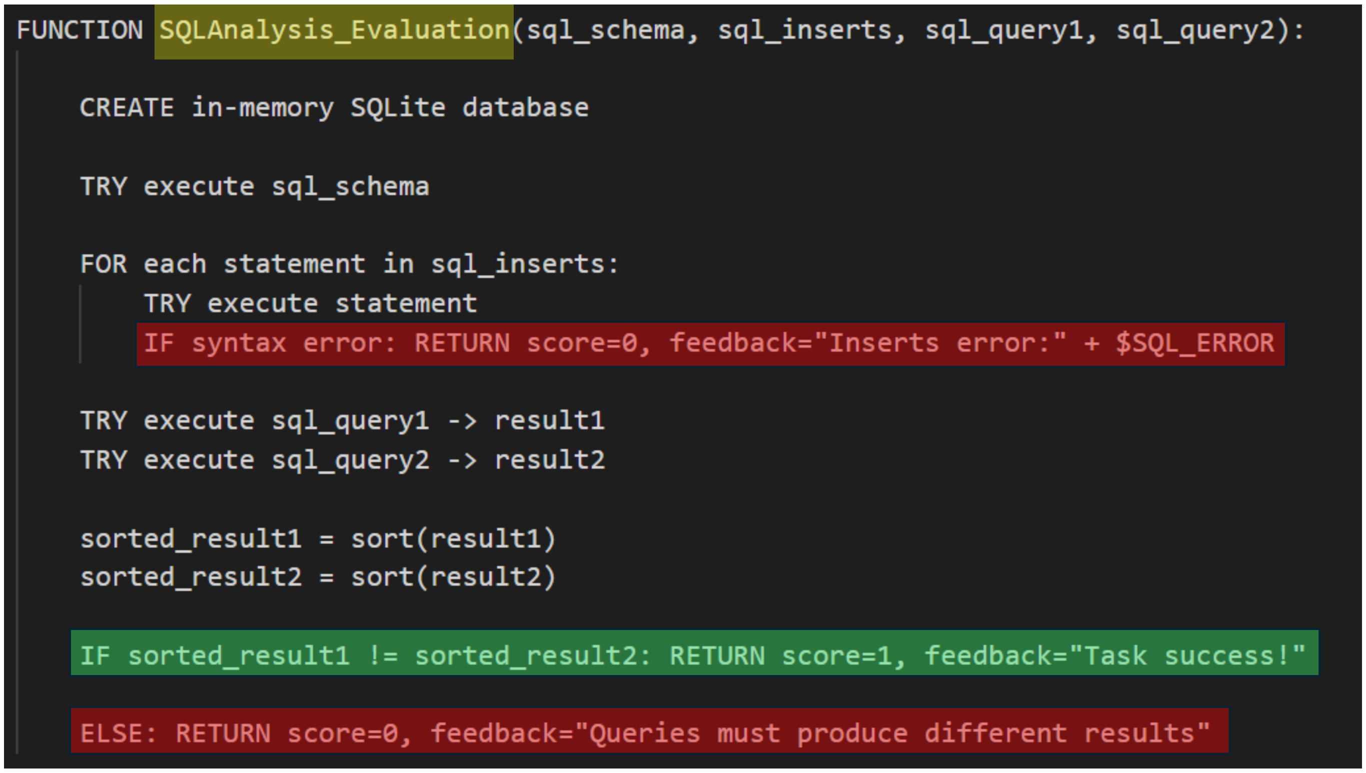
```
CREATE TABLE DEPARTMENTS (
    ID INTEGER,
    NAME TEXT(20)
);
CREATE TABLE STUDENTS (
    ID INTEGER,
    NAME TEXT(20),
    DEPARTMENT_ID INTEGER
);
```

```
SELECT ID, NAME
FROM STUDENTS
WHERE NOT DEPARTMENT_ID IN
(SELECT ID FROM DEPARTMENTS);
```

```
SELECT S.ID, S.NAME
FROM STUDENTS AS S LEFT OUTER JOIN DEPARTMENTS AS D
ON S.DEPARTMENT_ID = D.ID
WHERE D.ID IS NULL;
```

```
INSERT INTO DEPARTMENTS (ID, NAME) VALUES
(1, 'CS'), (2, 'Math');
INSERT INTO STUDENTS (ID, NAME, DEPARTMENT_ID) VALUES
(1, 'Alice', 3), (2, 'Bob', NULL);
```

Fig. 11. SQL Analysis Task: Example input and successful output. **Inputs:** A schema with DEPARTMENTS and STUDENTS tables, plus two non-equivalent queries—Query 1 uses NOT IN with a subquery, Query 2 uses LEFT OUTER JOIN with IS NULL. Both attempt to find students not in any department, but handle NULL values differently. **Output:** The model generates INSERT statements that expose this semantic difference by inserting a student (Bob) with NULL department_id. Query 1 returns no results (NOT IN with NULL produces UNKNOWN), while Query 2 correctly returns Bob, proving non-equivalence.



```
FUNCTION SQLAnalysis_Evaluation(sql_schema, sql_inserts, sql_query1, sql_query2):

    CREATE in-memory SQLite database

    TRY execute sql_schema

    FOR each statement in sql_inserts:
        TRY execute statement
        IF syntax error: RETURN score=0, feedback="Inserts error:" + $SQL_ERROR

    TRY execute sql_query1 -> result1
    TRY execute sql_query2 -> result2

    sorted_result1 = sort(result1)
    sorted_result2 = sort(result2)

    IF sorted_result1 != sorted_result2: RETURN score=1, feedback="Task success!"

    ELSE: RETURN score=0, feedback="Queries must produce different results"
```

Fig. 12. Scoring function for SQL Analysis Task (pseudocode). The function executes the schema, applies the generated INSERT statements, runs both queries, and compares their sorted results. A score of 1 (success) is returned only when the queries produce *different* results, validating that the model found a genuine counterexample. If results are identical, the model receives feedback indicating that the queries must produce different results, prompting refinement in multi-step variants.

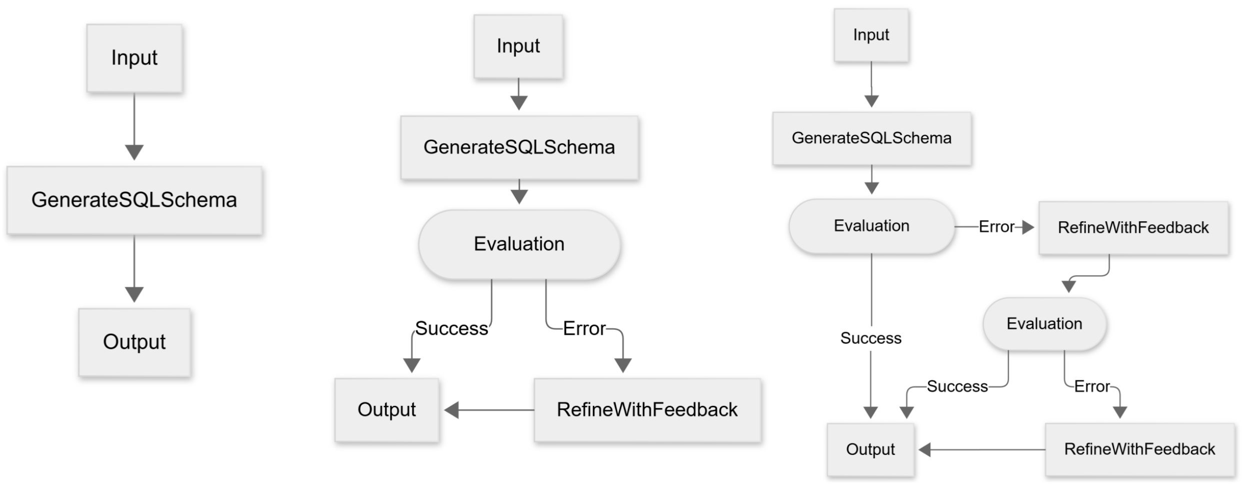


Fig. 13. SQL **Synthesis** Task flow diagrams for 1-Step, 2-Step, and 3-Step variants. **1-Step:** Direct generation with no feedback opportunity. **2-Step:** Output is evaluated; if errors occur (syntax, constraint, or equivalence failures), the model receives feedback and generates a refined attempt. **3-Step:** Adds a second evaluation-refinement cycle, giving the model two opportunities to correct errors.

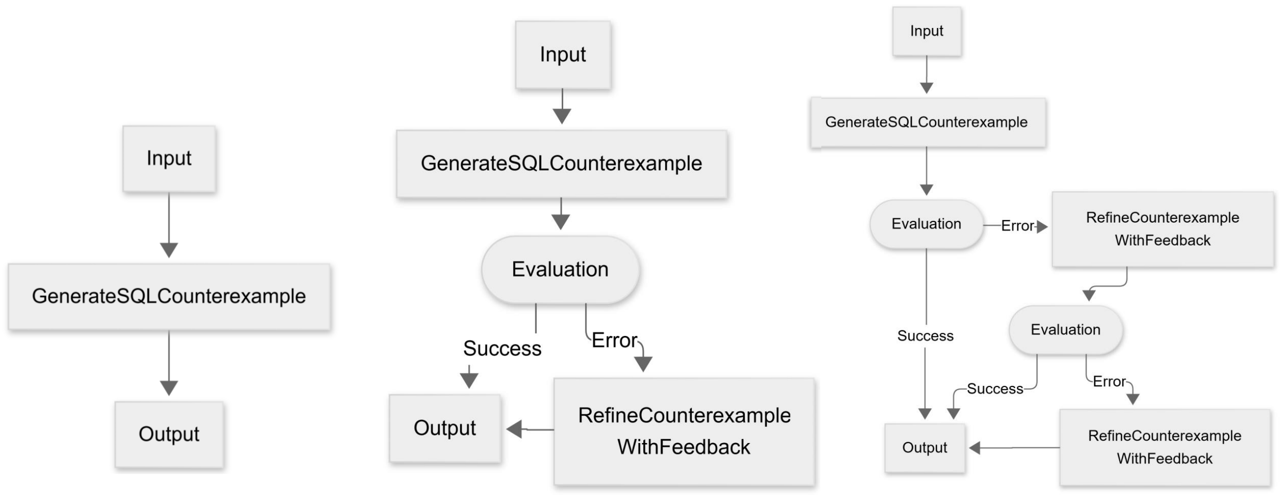

Input
GenerateSQLCounterexample
Output
Input
GenerateSQLCounterexample
Evaluation
Success
Error
Output
RefineCounterexample WithFeedback
Input
GenerateSQLCounterexample
Evaluation
Error
RefineCounterexample WithFeedback
Evaluation
Success
Success
Error
Output
RefineCounterexample WithFeedback

Fig. 14. SQL **Analysis** Task flow diagrams for 1-Step, 2-Step, and 3-Step variants. The structure mirrors SQL Synthesis from Figure 13, but refinement focuses specifically on counterexample generation. When evaluation shows that both queries still return identical results, the model receives this feedback along with its failed INSERT statements and attempts to generate a counterexample that exploits different semantic edge cases.

suggested another factor was at play.

D. *"Over Thinking" Bottleneck*

Investigation of the execution logs revealed a high frequency of response parsing errors. The `Qwen3-8B` model, when prompted with complex Chain-of-Thought instructions generated by Hyper-Reflection, engaged in excessive "thinking." The generated `<think>` blocks were so extensive that they consumed the entire generation budget (8k tokens), causing no actual answer to be generated.

Figure 19 shows the timeline of events during our experiment runs. The high density of red 'x' markers (indicating errors) across jobs 5–10 reveals systematic failures. These jobs corresponded to Hyper-Reflection optimized prompts that induced more complex reasoning.

Figure 20 provides a concrete example. The model's reasoning extends for thousands of tokens, analyzing the SQL queries and considering various approaches to finding a counterexample. However, the generation is truncated before the model can output the actual SQL INSERT statements. This explains why complex Hyper-Reflection prompts initially degraded SQL Analysis performance—they induced more thorough reasoning that exceeded the available token budget.

E. *High Context Experiments*

To resolve the truncation issue, we reconfigured vLLM to a "High Context" mode (`Qwen3-8B-hc`). We utilized YaRN embeddings [4] to extend the context window and generation limit to 40,960 tokens (see Appendix A).

The tradeoff: ten-hour experiments became twenty-hour experiments due to increased generation lengths. However, with the larger generation budget, the model could complete its thinking and produce answers.

This adjustment allowed the student model to complete its reasoning chains. As shown in Figure 21, the `opus1` Hyper-

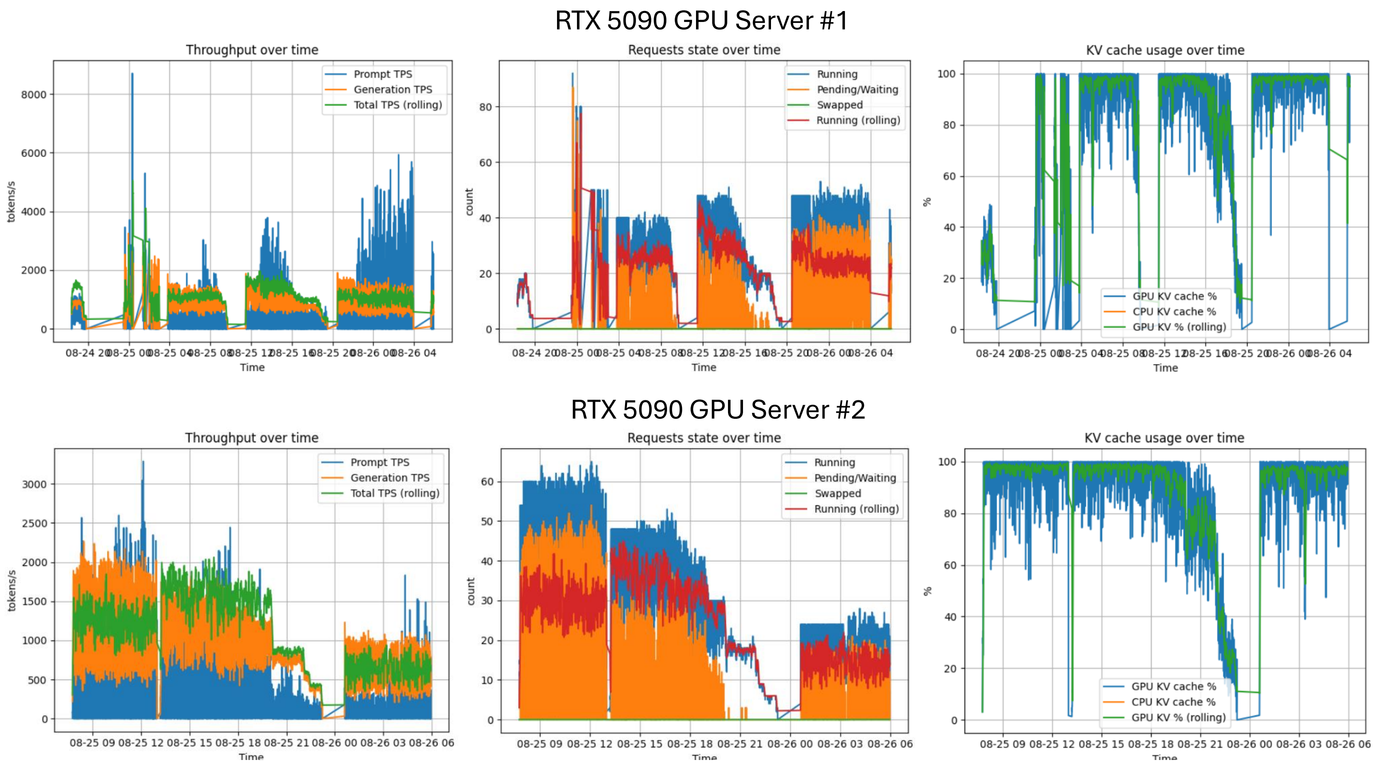
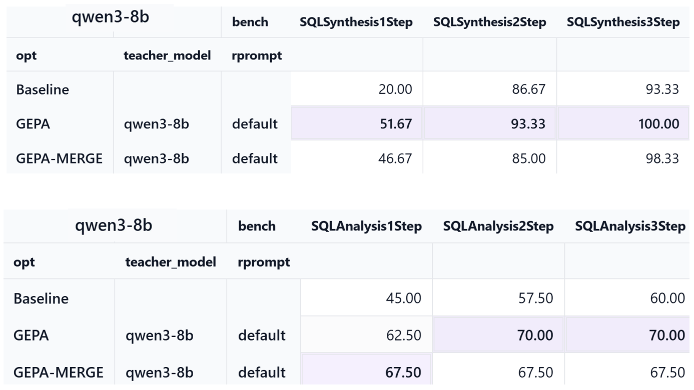


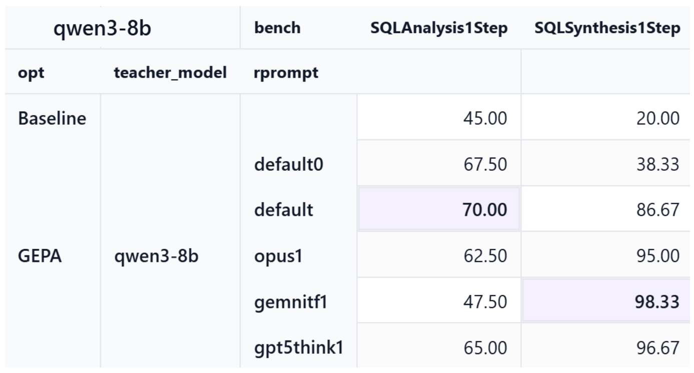
Fig. 15. Hardware utilization metrics on dual RTX 5090 servers during optimization runs. **Left column:** Throughput over time showing sustained 1,000–2,000 tokens per second (TPS). **Middle column:** Concurrent request count maintained at 30–40 requests for optimal batching efficiency. **Right column:** KV cache usage at ∼90%, indicating efficient GPU memory utilization with minimal waste. The top row shows Server #1; bottom row shows Server #2. Gaps in the graphs indicate breaks between experiment runs.



qwen3-8b		bench	SQLSynthesis1Step	SQLSynthesis2Step	SQLSynthesis3Step
opt	teacher_model	rprompt			
Baseline			20.00	86.67	93.33
GEPA	qwen3-8b	default	51.67	93.33	100.00
GEPA-MERGE	qwen3-8b	default	46.67	85.00	98.33

qwen3-8b		bench	SQLAnalysis1Step	SQLAnalysis2Step	SQLAnalysis3Step
opt	teacher_model	rprompt			
Baseline			45.00	57.50	60.00
GEPA	qwen3-8b	default	62.50	70.00	70.00
GEPA-MERGE	qwen3-8b	default	67.50	67.50	67.50

Fig. 16. Impact of database feedback on task success. **Upper table (SQL Synthesis):** Baseline success climbs from 20% (1-Step) to 86.67% (2-Step) to 93.33% (3-Step); GEPA achieves 100% on 3-Step. **Lower table (SQL Analysis):** Baseline improves from 45% to 57.50% to 60%; GEPA reaches 70% on 2-Step and 3-Step. Results demonstrate that structured feedback from database execution is crucial for both tasks, with diminishing returns from additional feedback loops on the harder Analysis task.



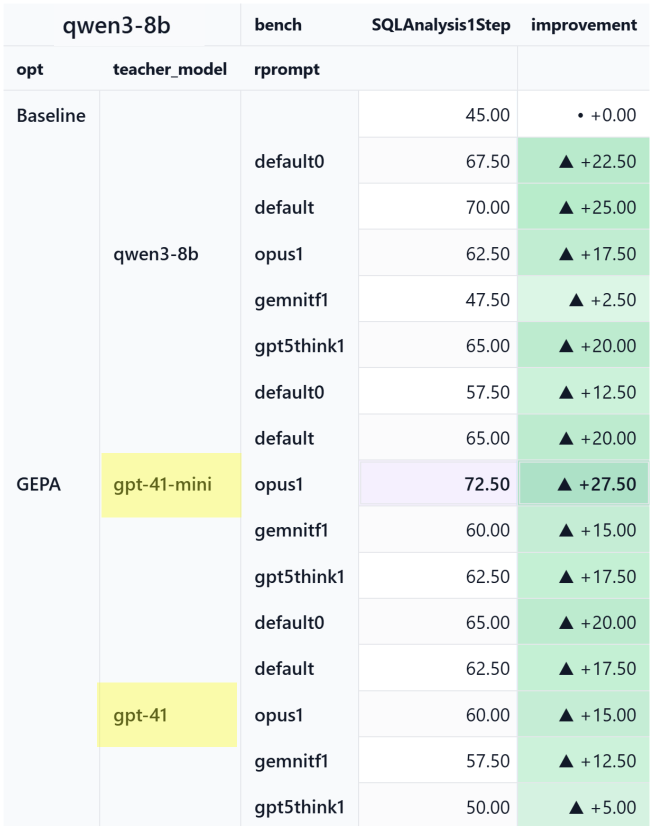
qwen3-8b		bench	SQLAnalysis1Step	SQLSynthesis1Step
opt	teacher_model	rprompt		
Baseline			45.00	20.00
GEPA	qwen3-8b	default0	67.50	38.33
		default	70.00	86.67
		opus1	62.50	95.00
		gemnitf1	47.50	98.33
		gpt5think1	65.00	96.67

Fig. 17. Impact of Hyper-Reflection on task success (1-Step variants, Qwen3-8B as teacher and student). For SQL Analysis, Hyper-Reflection prompts underperforms compared to default GEPA. However in SQL Synthesis, Hyper-Reflection prompts excels in performace over default GEPA. The asymmetry between tasks motivated investigation of why Hyper-Reflection helped Synthesis but not Analysis.

Reflection template, which previously failed due to truncation, now achieved state-of-the-art performance of 75.00% success, outperforming the GEPA default.

Key observations from the High Context experiments:

- The baseline improved from 45% to 65%, indicating that extended context alone helps the model reason about SQL semantics.
- GEPA with the default template actually decreased to 55%, suggesting the default prompt was optimized for shorter contexts and may have become suboptimal when more space was available.
- The `opus1` Hyper-Reflection template achieved the best result at 75%, a 30-point improvement over the standard configuration baseline.



qwen3-8b		bench	SQLAnalysis1Step	improvement
opt	teacher_model	rprompt		
Baseline			45.00	• +0.00
GEPA	qwen3-8b	default0	67.50	▲ +22.50
		default	70.00	▲ +25.00
		opus1	62.50	▲ +17.50
		gemnitf1	47.50	▲ +2.50
		gpt5think1	65.00	▲ +20.00
	gpt-41-mini	default0	57.50	▲ +12.50
		default	65.00	▲ +20.00
		opus1	72.50	▲ +27.50
		gemnitf1	60.00	▲ +15.00
		gpt5think1	62.50	▲ +17.50
	gpt-41	default0	65.00	▲ +20.00
		default	62.50	▲ +17.50
		opus1	60.00	▲ +15.00
		gemnitf1	57.50	▲ +12.50
		gpt5think1	50.00	▲ +5.00

Fig. 18. Task success with smarter teachers on SQL Analysis 1-Step (Qwen3-8B student). **GPT-4.1-mini teacher:** `opus1` leads at 72.50% (+27.5 over baseline), `default` at 65.00%, `gpt5think1` at 62.50%, `gemnitf1` at 60.00%. **GPT-4.1 teacher:** Performance unexpectedly drops—`opus1` achieves only 60.00%, `gemnitf1` 57.50%, `gpt5think1` only 50.00%. The counterintuitive result that the smartest teacher performed worst suggested another factor was at play.

- `gpt5think1` achieved 70.00%, while `gemnitf1` remained at 62.50%.
- The ablated `default0` dropped to 62.50%, now underperforming the Hyper-Reflection variants.

These results validate our hypothesis: Hyper-Reflection prompts induce more sophisticated reasoning that requires adequate generation budget. When context is limited, the sophisticated prompts backfire by consuming tokens on reasoning that never completes. When context is extended, the sophisticated prompts unlock superior performance.

VII. Discussion and Future Work

Our experiments reveal several important insights about prompt optimization for compact models:

Context is Compute. The "thinking truncation" phenomenon demonstrates that the complexity of optimized prompts must be matched by adequate generation budget. This creates a tradeoff between prompt sophistication and inference cost that practitioners must consider. More powerful prompts may require extended context windows, increasing latency and memory requirements.

Task-Dependent Optimization. Hyper-Reflection prompts provided dramatic improvements for SQL Synthesis (98.33%) but initially failed on SQL Analysis (47.50%). This asymmetry suggests that optimal reflection strategies may be task-specific, and meta-optimization approaches should account for task characteristics. SQL Synthesis benefits from creative exploration, while SQL Analysis requires systematic edge-case reasoning.

Teacher-Student Dynamics. Using smarter teachers (GPT-4.1-mini, GPT-4.1) helped but did not fully resolve the issues. Surprisingly, the smartest teacher sometimes produced worse results, possibly due to generating prompts that were too sophisticated for the student model. This suggests a "capability matching" principle: teacher-generated prompts should be calibrated to the student's capacity.

Trade-offs Between Adaptation Approaches. As discussed in Section II, weight-space distillation yields fast inference but requires training infrastructure; prompt-space optimization requires only black-box access but is limited by context length; runtime reasoning maximizes performance but consumes tokens per query. Our results show that prompt-space methods can achieve substantial gains when combined with context extension, offering a middle ground between adaptation cost and inference efficiency.

Several promising directions for future research emerge from this work:

Adaptive Reflection Strategies. In this work, we selected the single best reflection template ($P^*_{reflect}$) for the entire optimization run. However, different stages of optimization may benefit from different reflection styles—early stages may require high-temperature creative exploration (e.g., `opus1`), while later stages benefits from strict constraint checking (e.g., `gpt5think1`). Future work could implement a Multi-Armed Bandit algorithm within the GEPA loop to dynamically select the optimal reflection template $P^{(t)}_{reflect}$ at step t based on the immediate reward (fitness gain) of the populations generated.

Extending Hyper-Reflection to Other Domains. Our SQL tasks provide clear success criteria through database execution. Extending Hyper-Reflection to other domains with verifiable outcomes (code generation, mathematical reasoning, formal verification) would test its generalizability.

Early Stopping for Thinking Tokens. Implementing early stopping of thinking tokens generation would ensure an answer is always generated, even under token budget constraints. This could be achieved through monitoring the reasoning phase and forcibly transitioning to output when a threshold is approached.

Learning from Optimization Trajectories. A potentially transformative direction would be to systematically convert optimization trajectories into training data for prompt optimizers. Methods like GEPA, C-Evolve, Maestro, Feedback Descent, and ACE produce rich trajectories—evolving prompts, pairwise preferences, and textual rationales—that are currently discarded after optimization.

Building on Black-Box Prompt Optimization (BPO) [25], which learns to rewrite user instructions using human preference datasets, one could collect logs from multiple optimizers across benchmarks and base LLMs. These trajectories could be transformed into training examples where a model learns to map from original instructions to improved prompts. Such

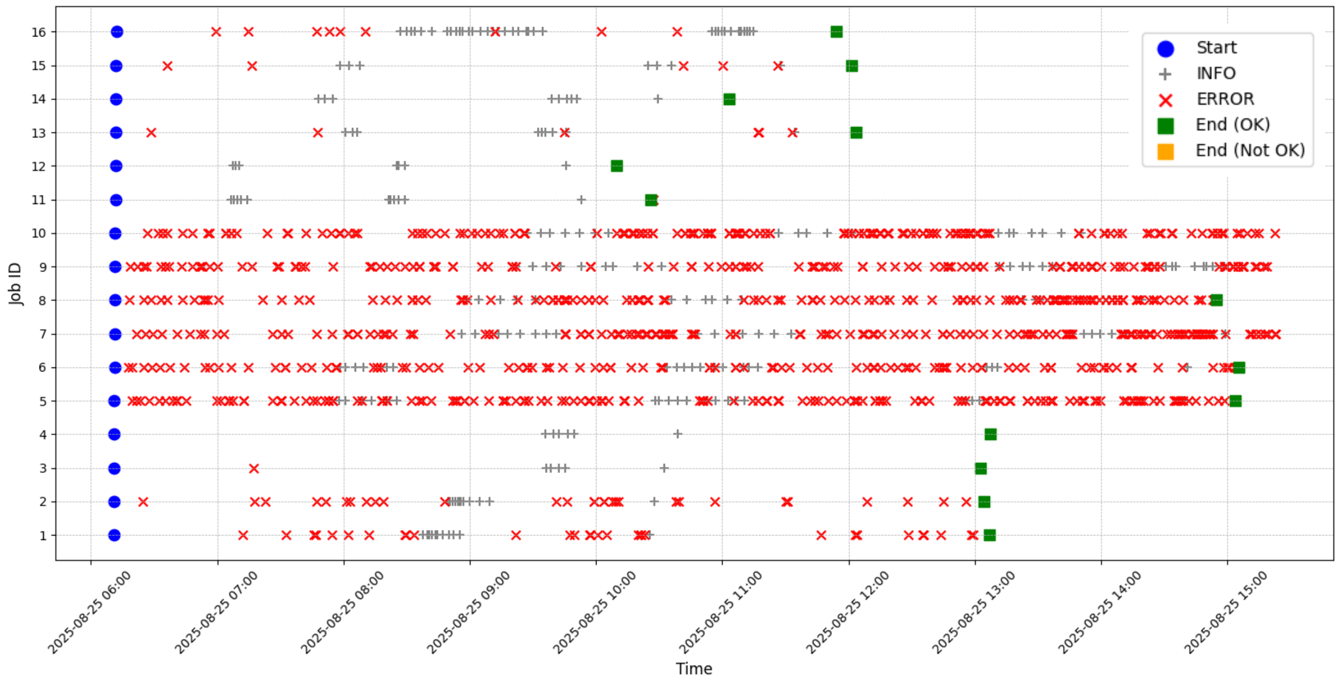
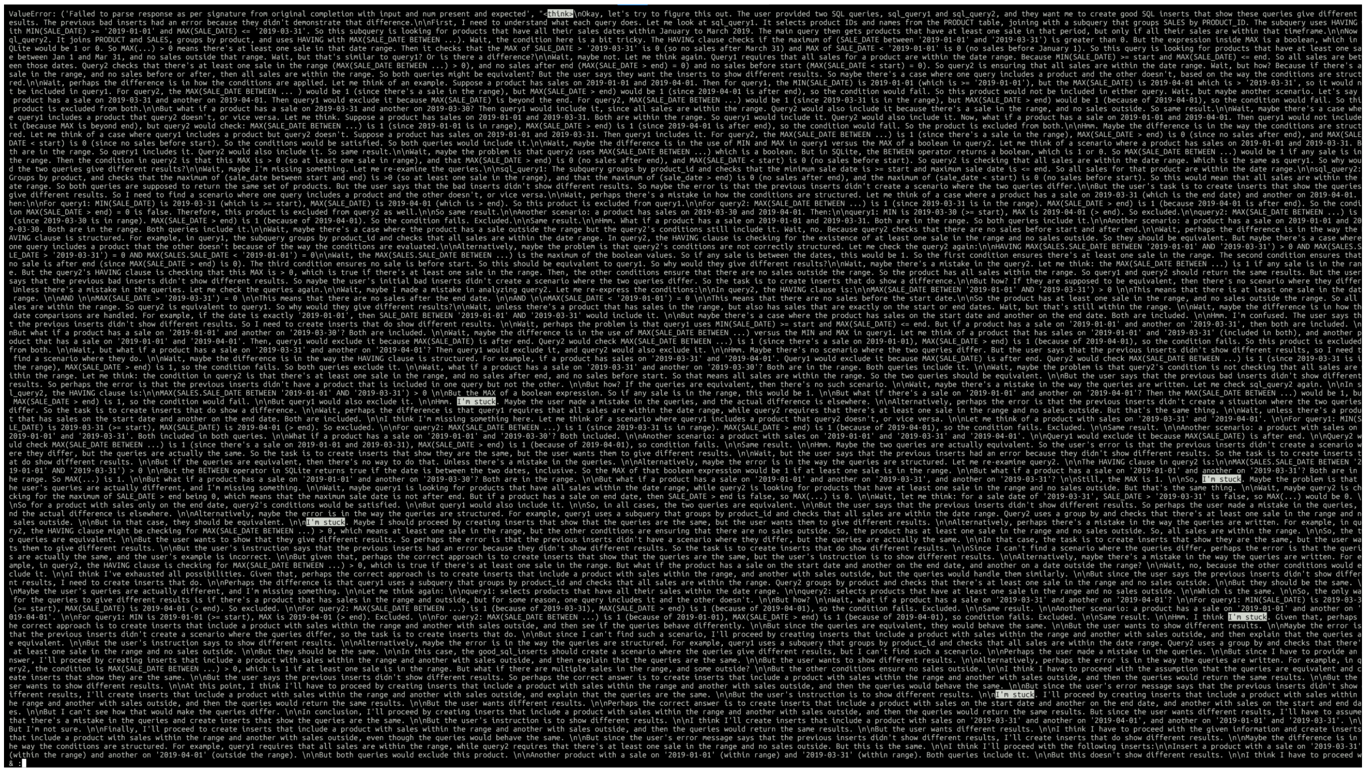
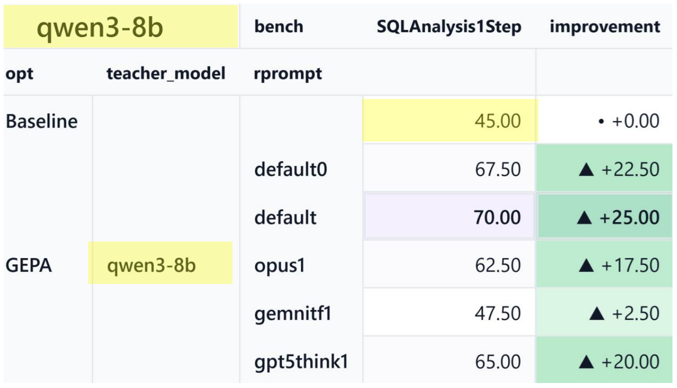


Fig. 19. Timeline of events during SQL Analysis optimization runs showing systematic failures. Each row represents a different job (optimization configuration); the x-axis shows time. Markers indicate: green circles (start), plus signs (info messages), red 'x' (errors), green squares (successful completion). Jobs 5–10 show high error density (abundant red markers), corresponding to Hyper-Reflection prompts that induced complex reasoning. Jobs 1–4 show fewer errors, corresponding to simpler prompt configurations. The pattern reveals that sophisticated prompts caused systematic failures.

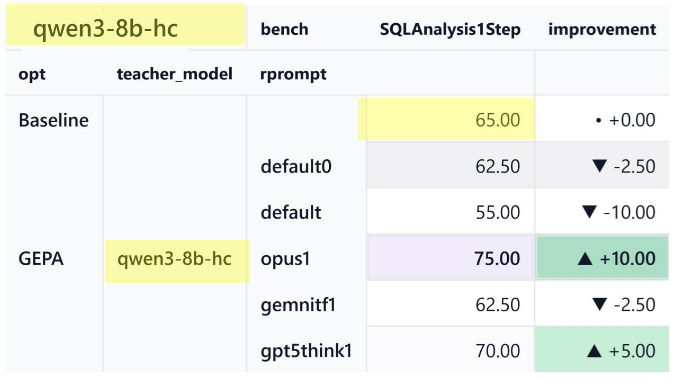


```
ValueError: ('Failed to parse response as per signature from original completion with input and num present and expected', "<think>\nOkay, let's try to figure this out. The user provided two SQL queries, sql_query1 and sql_query2, and they want me to create good SQL inserts that show these queries give different
esults. The previous bad inserts had an error because they didn't demonstrate that difference.\n\nFirst, I need to understand what each query does. Let me look at sql_query1. It selects product IDs and names from the PRODUCT table, joining with a subquery that groups SALES by PRODUCT_ID. The subquery uses HAVING
ith MIN(SALE_DATE) >= '2019-01-01' and MAX(SALE_DATE) <= '2019-03-31'. So this subquery is looking for products that have all their sales dates within January to March 2019. The main query then gets products that have at least one sale in that period, but only if all their sales are within that timeframe.\n\nNow
ql_query2. It joins PRODUCT and SALES, groups by product, and uses HAVING with MAX(SALE_DATE BETWEEN ...). Wait, the condition here is a bit tricky. The HAVING clause checks if the maximum of (SALE_DATE between '2019-01-01' and '2019-03-31') is greater than 0. But the expression inside MAX is a boolean, which in
QLite would be 1 or 0. So MAX(...) > 0 means there's at least one sale in that date range. Then it checks that the MAX of SALE_DATE > '2019-03-31' is 0 (so no sales after March 31) and MAX of SALE_DATE < '2019-01-01' is 0 (no sales before January 1). So this query is looking for products that have at least one sa
e between Jan 1 and Mar 31, and no sales outside that range. Wait, but that's similar to query1? Or is there a difference?\n\nWait, maybe not. Let me think again. Query1 requires that all sales for a product are within the date range. Because MIN(SALE_DATE) >= start and MAX(SALE_DATE) <= end. So all sales are bet
een those dates. Query2 checks that there's at least one sale in the range (MAX(SALE_DATE BETWEEN ...) > 0), and no sales after end (MAX(SALE_DATE > end) = 0) and no sales before start (MAX(SALE_DATE < start) = 0). So query2 is ensuring that all sales are within the date range. Wait, but how? Because if there's a
sale in the range, and no sales before or after, then all sales are within the range. So both queries might be equivalent? But the user says they want the inserts to show different results. So maybe there's a case where one query includes a product and the other doesn't, based on the way the conditions are struct
red.\n\nWait, perhaps the difference is in how the conditions are applied. Let me think of an example. Suppose a product has sales on 2019-01-01 and 2019-04-01. Then for query1, the MIN(SALE_DATE) is 2019-01-01 (which is >= '2019-01-01'), but the MAX(SALE_DATE) is 2019-04-01 which is > '2019-03-31', so it would n
t be included in query1. For query2, the MAX(SALE_DATE BETWEEN ... ) would be 1 (since there's a sale in the range), but MAX(SALE_DATE > end) would be 1 (since 2019-04-01 is after end), so the condition would fail. So this product would not be included in either query. Wait, but maybe another scenario. Let's say
 product has a sale on 2019-03-31 and another on 2019-04-01. Then query1 would exclude it because MAX(SALE_DATE) is beyond the end. For query2, MAX(SALE_DATE BETWEEN ...) would be 1 (since 2019-03-31 is in the range), but MAX(SALE_DATE > end) would be 1 (because of 2019-04-01), so the condition would fail. So thi
 product is excluded from both.\n\nBut what if a product has a sale on 2019-03-31 and another on 2019-03-30? Then query1 would include it, since all sales are within the range. Query2 would also include it because there's a sale in the range, and no sales outside. So same result.\n\nWait, maybe there's a case whe
e query1 includes a product that query2 doesn't, or vice versa. Let me think. Suppose a product has sales on 2019-01-01 and 2019-03-31. Both are within the range. So query1 would include it. Query2 would also include it. Now, what if a product has a sale on 2019-01-01 and 2019-04-01. Then query1 would not include
it (because MAX is beyond end), but query2 would check: MAX(SALE_DATE BETWEEN ...) is 1 (since 2019-01-01 is in range), MAX(SALE_DATE > end) is 1 (since 2019-04-01 is after end), so the condition would fail. So the product is excluded from both.\n\nHmm. Maybe the difference is in the way the conditions are struct
red. Let me think of a case where query1 includes a product but query2 doesn't. Suppose a product has sales on 2019-01-01 and 2019-03-31. Then query1 includes it. For query2, the MAX(SALE_DATE BETWEEN ...) is 1 (since there's a sale in the range), MAX(SALE_DATE > end) is 0 (since no sales after end), and MAX(SALE
DATE < start) is 0 (since no sales before start). So the conditions would be satisfied. So both queries would include it.\n\nWait, maybe the difference is in the use of MIN and MAX in query1 versus the MAX of a boolean in query2. Let me think of a scenario where a product has sales on 2019-01-01 and 2019-03-31. B
th are in the range. So query1 includes it. Query2 would also include it. So same result.\n\nWait, maybe the problem is that query2 uses MAX(SALE_DATE BETWEEN ...) which is a boolean. But in SQLite, the BETWEEN operator returns a boolean, which is 1 or 0. So MAX(SALE_DATE BETWEEN ...) would be 1 if any sale is in
the range. Then the condition in query2 is that this MAX is > 0 (so at least one sale in range), and that MAX(SALE_DATE > end) is 0 (no sales after end), and MAX(SALE_DATE < start) is 0 (no sales before start). So query2 is checking that all sales are within the date range. Which is the same as query1. So why wou
d the two queries give different results?\n\nWait, maybe I'm missing something. Let me re-examine the queries.\n\nsql_query1: The subquery groups by product_id and checks that the minimum sale date is >= start and maximum sale date is <= end. So all sales for that product are within the date range.\n\nsql_query2:
Groups by product, and checks that the maximum of (sale_date between start and end) is >0 (so at least one sale in the range), and that the maximum of (sale_date > end) is 0 (no sales after end), and the maximum of (sale_date < start) is 0 (no sales before start). So this would mean that all sales are within the
ate range. So both queries are supposed to return the same set of products. But the user says that the bad inserts didn't show different results. So maybe the error is that the previous inserts didn't create a scenario where the two queries differ.\n\nBut the user's task is to create inserts that show the queries
give different results. So I need to find a scenario where one query includes a product and the other doesn't, or vice versa.\n\nWait, perhaps there's a mistake in how the conditions are structured. Let me think of a case where a product has a sale on 2019-03-31 (which is the end date) and another on 2019-04-01.
hen:\n\nFor query1: MIN(SALE_DATE) is 2019-03-31 (which is >= start), MAX(SALE_DATE) is 2019-04-01 (which is > end). So this product is excluded from query1.\n\nFor query2: MAX(SALE_DATE BETWEEN ...) is 1 (since 2019-03-31 is in the range). MAX(SALE_DATE > end) is 1 (because 2019-04-01 is after end). So the condi
ion MAX(SALE_DATE > end) = 0 is false. Therefore, this product is excluded from query2 as well.\n\nSo same result.\n\nAnother scenario: a product has sales on 2019-03-30 and 2019-04-01. Then:\n\nquery1: MIN is 2019-03-30 (>= start), MAX is 2019-04-01 (> end). So excluded.\n\nquery2: MAX(SALE_DATE BETWEEN ...) is
 (since 2019-03-30 is in range). MAX(SALE_DATE > end) is 1 (because of 2019-04-01). So the condition fails. Excluded.\n\nSame result.\n\nHmm. What if a product has a sale on 2019-01-01 and 2019-03-31. Both are in the range. So both queries include it.\n\nAnother scenario: a product has a sale on 2019-01-01 and 20
9-03-30. Both are in the range. Both queries include it.\n\nWait, maybe there's a case where the product has a sale outside the range but the query2's conditions still include it. Wait, no. Because query2 checks that there are no sales before start and after end.\n\nWait, perhaps the difference is in the way the
AVING clause is structured. For example, in query1, the subquery groups by product_id and checks that all sales are within the date range. In query2, the HAVING clause is checking for the existence of at least one sale in the range and no sales outside. So they should be equivalent. But maybe there's a case where
one query includes a product that the other doesn't because of the way the conditions are evaluated.\n\nAlternatively, maybe the problem is that query2's conditions are not correctly structured. Let me check the query2 again:\n\nHAVING MAX(SALES.SALE_DATE BETWEEN '2019-01-01' AND '2019-03-31') > 0 AND MAX(SALES.S
LE_DATE > '2019-03-31') = 0 AND MAX(SALES.SALE_DATE < '2019-01-01') = 0\n\nWait, the MAX(SALES.SALE_DATE BETWEEN ...) is the maximum of the boolean values. So if any sale is between the dates, this would be 1. So the first condition ensures there's at least one sale in the range. The second condition ensures that
no sale is after end (since MAX(SALE_DATE > end) is 0). The third condition ensures no sale is before start. So this should be equivalent to query1. So why would they give different results?\n\nWait, maybe there's a mistake in the query2. Let me think: the MAX(SALE_DATE BETWEEN ...) is 1 if any sale is in the ran
e. But the query2's HAVING clause is checking that this MAX is > 0, which is true if there's at least one sale in the range. Then, the other conditions ensure that there are no sales outside the range. So the product has all sales within the range. So query1 and query2 should return the same results. But the user
says that the previous bad inserts didn't show different results. So maybe the user's initial bad inserts didn't create a scenario where the two queries differ. So the task is to create inserts that do show a difference.\n\nBut how? If they are supposed to be equivalent, then there's no scenario where they differ
 Unless there's a mistake in the queries. Let me check the queries again.\n\nWait, maybe I made a mistake in analyzing query2. Let me re-express the conditions:\n\nIn query2, the HAVING clause is:\n\nMAX(SALE_DATE BETWEEN '2019-01-01' AND '2019-03-31') > 0 \n\nThis means that there is at least one sale in the dat
 range. \n\nAND \n\nMAX(SALE_DATE > '2019-03-31') = 0 \n\nThis means that there are no sales after the end date. \n\nAND \n\nMAX(SALE_DATE < '2019-01-01') = 0 \n\nThis means that there are no sales before the start date.\n\nSo the product has at least one sale in the range, and no sales outside the range. So all
ales are within the range. So query2 is equivalent to query1. So why would they give different results?\n\nWait, unless there's a product that has sales in the range, but also has sales that are exactly on the start or end dates. Wait, but that's still within the range. \n\nWait, maybe the difference is in how th
 date comparisons are handled. For example, if the date is exactly '2019-01-01', then SALE_DATE BETWEEN '2019-01-01' AND '2019-03-31' would include it. \n\nBut maybe there's a case where the product has sales on the start date and another on the end date. Both are included. \n\nHmm. I'm confused. The user says th
t the previous inserts didn't show different results. So I need to create inserts that do show different results. \n\nWait, perhaps the problem is that query1 uses MIN(SALE_DATE) >= start and MAX(SALE_DATE) <= end. But if a product has a sale on '2019-01-01' and another on '2019-03-31', then both are included. \n
nBut what if a product has a sale on '2019-01-01' and another on '2019-03-30'? Both are included. \n\nWait, maybe the difference is in the use of MAX(SALE_DATE BETWEEN ...) versus the MIN and MAX in query1. Let me think of a product that has sales on '2019-01-01' and '2019-03-31' (included in both), and another p
oduct that has a sale on '2019-01-01' and '2019-04-01'. Then, query1 would exclude it because MAX(SALE_DATE) is after end. Query2 would check MAX(SALE_DATE BETWEEN ...) is 1 (since there's a sale on 2019-01-01), MAX(SALE_DATE > end) is 1 (because of 2019-04-01), so the condition fails. So this product is excluded
from both. \n\nWait, but what if a product has a sale on '2019-03-31' and another on '2019-04-01'? Then query1 would exclude it, and query2 would also exclude it. \n\nHmm. Maybe there's no scenario where the two queries differ. But the user says that the previous inserts didn't show different results, so I need t
 find a scenario where they do. \n\nWait, maybe the difference is in the way the HAVING clause is structured. For example, if a product has sales on '2019-03-31' and '2019-04-01'. Query1 would exclude it because MAX(SALE_DATE) is after end. Query2 would check MAX(SALE_DATE BETWEEN ...) is 1 (since 2019-03-31 is i
 the range), MAX(SALE_DATE > end) is 1, so the condition fails. So both queries exclude it. \n\nWait, what if a product has a sale on '2019-03-31' and another on '2019-03-30'? Both are in the range. Both queries include it. \n\nWait, maybe the problem is that query2's condition is not checking that all sales are
ithin the range. Let me think: the condition in query2 is that there's at least one sale in the range, and no sales after end, and no sales before start. So that means all sales are within the range. So the two queries should be equivalent. \n\nBut the user says that the previous bad inserts didn't show different
results. So perhaps the error is that the previous inserts didn't have a product that is included in one query but not the other. \n\nBut how? If the queries are equivalent, then there's no such scenario. \n\nWait, maybe there's a mistake in the way the queries are written. Let me check sql_query2 again. \n\nIn s
l_query2, the HAVING clause is:\n\nMAX(SALES.SALE_DATE BETWEEN '2019-01-01' AND '2019-03-31') > 0 \n\nBut the MAX of a boolean expression. So if any sale is in the range, this would be 1. \n\nBut what if there's a sale on '2019-01-01' and another on '2019-04-01'? Then the MAX(SALE_DATE BETWEEN ...) would be 1, bu
 MAX(SALE_DATE > end) is 1, so the condition would fail. \n\nBut query1 would also exclude it. \n\nHmm. I'm stuck. Maybe the user made a mistake in the queries, and the actual difference is elsewhere. \n\nAlternatively, perhaps the error is that the previous inserts didn't create a situation where the two queries
differ. So the task is to create inserts that do show a difference. \n\nWait, perhaps the difference is that query1 requires that all sales are within the date range, while query2 requires that there's at least one sale in the range and no sales outside. But that's the same thing. \n\nWait, unless there's a produ
t that has sales on the start date and another on the end date. Both are included. \n\nI think I'm missing something here. Let me think of a scenario where query1 includes a product that query2 doesn't, or vice versa. \n\nLet me think of a product with sales on '2019-03-31' and '2019-04-01'. \n\nFor query1: MIN(S
LE_DATE) is 2019-03-31 (>= start), MAX(SALE_DATE) is 2019-04-01 (> end). So excluded. \n\nFor query2: MAX(SALE_DATE BETWEEN ...) is 1 (since 2019-03-31 is in range). MAX(SALE_DATE > end) is 1 (because of 2019-04-01), so the condition fails. Excluded. \n\nSame result. \n\nAnother scenario: a product with sales on
2019-01-01' and '2019-03-31'. Both included in both queries. \n\nWhat if a product has a sale on '2019-01-01' and '2019-03-30'? Both included. \n\nAnother scenario: a product with sales on '2019-01-01' and '2019-03-31' and '2019-04-01'. \n\nQuery1 would exclude it because MAX(SALE_DATE) is after end. \n\nQuery2 w
uld check MAX(SALE_DATE BETWEEN ...) is 1 (since there's a sale on 2019-01-01 and 2019-03-31), MAX(SALE_DATE > end) is 1 (because of 2019-04-01), so condition fails. \n\nSame result. \n\nHmm. Maybe the two queries are actually equivalent. So the user's error is that the previous inserts didn't create a scenario w
ere they differ, but the queries are actually the same. So the task is to create inserts that show they are the same, but the user wants them to give different results. \n\nWait, but the user says that the previous inserts had an error because they didn't show different results. So the task is to create inserts t
at do show different results. \n\nBut if the queries are equivalent, then there's no way to do that. Unless there's a mistake in the queries. \n\nAlternatively, maybe the error is in the way the queries are structured. Let me re-examine query2. \n\nThe HAVING clause in query2 is:\n\nMAX(SALES.SALE_DATE BETWEEN '2
19-01-01' AND '2019-03-31') > 0 \n\nBut the BETWEEN operator in SQLite returns true if the date is between the two dates, inclusive. So the MAX of that boolean expression would be 1 if at least one sale is in the range. \n\nBut what if a product has a sale on '2019-01-01' and another on '2019-03-31'? Both are in
he range. So MAX(...) is 1. \n\nBut what if a product has a sale on '2019-01-01' and another on '2019-03-30'? Both are in the range. \n\nBut what if a product has a sale on '2019-01-01' and another on '2019-03-31', and another on '2019-03-31'? \n\nStill, the MAX is 1. \n\nSo, I'm stuck. Maybe the problem is that
he user's queries are actually different, and I'm missing something. \n\nWait, maybe query1 is looking for products that have all sales within the date range, while query2 is looking for products that have at least one sale in the range and no sales outside. But that's the same thing. \n\nWait, maybe query2 is ch
cking for the maximum of SALE_DATE > end being 0, which means that the maximum sale date is not after end. But if a product has a sale on end date, then SALE_DATE > end is false, so MAX(...) is 0. \n\nWait, let me think: for a sale date of '2019-03-31', SALE_DATE > '2019-03-31' is false, so MAX(...) would be 0. \
\nSo for a product with sales only on the end date, query2's conditions would be satisfied. \n\nBut query1 would also include it. \n\nSo, in all cases, the two queries are equivalent. \n\nBut the user says that the previous inserts didn't show different results. So perhaps the user made a mistake in the queries,
nd the actual difference is elsewhere. \n\nAlternatively, maybe the error is in the way the queries are structured. For example, query1 uses a subquery that groups by product_id and checks that all sales are within the date range. Query2 uses a group by and checks that there's at least one sale in the range and n
 sales outside. \n\nBut in that case, they should be equivalent. \n\nI'm stuck. Maybe I should proceed by creating inserts that show that the queries are the same, but the user wants them to give different results. \n\nAlternatively, perhaps there's a mistake in the way the queries are written. For example, in qu
ry2, the HAVING clause might be checking for MAX(SALE_DATE BETWEEN ...) > 0, which means at least one sale in the range, but the other conditions are ensuring that there are no sales outside. So, the product has at least one sale in the range and no sales outside. So, all sales are within the range. \n\nSo, the t
o queries are equivalent. \n\nBut the user wants to show that they give different results. So perhaps the error is that the previous inserts didn't have a scenario where they differ, but the queries are actually the same. \n\nIn that case, the task is to create inserts that show they are the same, but the user wa
ts them to give different results. \n\nBut the user's instruction says that the previous inserts had an error because they didn't show different results. So the task is to create inserts that do show different results. \n\nSince I can't find a scenario where the queries differ, perhaps the error is that the queri
s are actually the same, and the user's example is incorrect. \n\nBut given that, perhaps the correct approach is to create inserts that show that the queries are the same, but the user's instruction is to show different results. \n\nAlternatively, maybe there's a mistake in the way the queries are written. For e
ample, in query2, the HAVING clause is checking for MAX(SALE_DATE BETWEEN ...) > 0, which is true if there's at least one sale in the range. But what if the product has a sale on the start date and another on the end date, and another on a date outside the range? \n\nWait, no, because the other conditions would e
clude it. \n\nI think I've exhausted all possibilities. Given that, perhaps the correct approach is to create inserts that include a product with sales within the range, and another with sales outside, but the queries would handle them similarly. \n\nBut since the user says the previous inserts didn't show differ
nt results, I need to create inserts that do. \n\nPerhaps the difference is that query1 uses a subquery that groups by product_id and checks that all sales are within the range. Query2 groups by product and checks that there's at least one sale in the range and no sales outside. \n\nBut they should be the same. \
\nMaybe the user's queries are actually different, and I'm missing something. \n\nLet me think again: \n\nquery1: selects products that have all their sales within the date range. \n\nquery2: selects products that have at least one sale in the range and no sales outside. \n\nWhich is the same. \n\nSo, the only wa
 for the queries to give different results is if there's a product that has sales in the range and outside, but for some reason, one query includes it and the other doesn't. \n\nBut how? \n\nWait, what if a product has a sale on '2019-03-31' and another on '2019-04-01'? \n\nFor query1: MIN(SALE_DATE) is 2019-03-3
 (>= start), MAX(SALE_DATE) is 2019-04-01 (> end). So excluded. \n\nFor query2: MAX(SALE_DATE BETWEEN ...) is 1 (because of 2019-03-31), MAX(SALE_DATE > end) is 1 (because of 2019-04-01), so the condition fails. Excluded. \n\nSame result. \n\nAnother scenario: a product has a sale on '2019-01-01' and another on '
019-04-01'. \n\nFor query1: MIN is 2019-01-01 (>= start), MAX is 2019-04-01 (> end). Excluded. \n\nFor query2: MAX(SALE_DATE BETWEEN ...) is 1 (because of 2019-01-01), MAX(SALE_DATE > end) is 1 (because of 2019-04-01), so condition fails. Excluded. \n\nSame result. \n\nHmm. I think I'm stuck. Given that, perhaps
he correct approach is to create inserts that include a product with sales within the range and another with sales outside, and then see if the queries behave differently. \n\nBut since the queries are equivalent, they would behave the same. \n\nBut the user wants to show different results. \n\nMaybe the error is
that the previous inserts didn't create a scenario where the queries differ, so the task is to create inserts that do. \n\nBut since I can't find such a scenario, I'll proceed by creating inserts that include a product with sales within the range and another with sales outside, and then explain that the queries a
e equivalent. \n\nBut the user's instruction says to show different results. \n\nAlternatively, maybe the error is in the way the queries are structured. For example, query1 uses a subquery that groups by product_id and checks that all sales are within the date range. Query2 uses a group by and checks that there'
 at least one sale in the range and no sales outside. \n\nBut they should be the same. \n\nIn this case, the good_sql_inserts should create a scenario where the queries give different results, but I can't find such a scenario. \n\nPerhaps the user made a mistake in the queries. \n\nBut since I have to provide an
nswer, I'll proceed by creating inserts that include a product with sales within the range and another with sales outside, and then explain that the queries are the same. \n\nBut the user wants to show different results. \n\nAlternatively, perhaps the error is in the way the queries are written. For example, in q
ery2, the condition is MAX(SALE_DATE BETWEEN ...) > 0, which is 1 if at least one sale is in the range. But what if there are multiple sales in the range, and some outside? \n\nBut the other conditions ensure no sales outside. \n\nI think I have to proceed with the assumption that the queries are equivalent and c
eate inserts that show they are the same. \n\nBut the user says the previous inserts didn't show different results. So perhaps the correct answer is to create inserts that include a product with sales within the range and another with sales outside, and then the queries would return the same results. \n\nBut the
ser wants to show different results. \n\nAt this point, I think I'll have to proceed by creating inserts that include a product with sales within the range and another with sales outside, and then the queries would behave the same. \n\nBut since the user's error message says that the previous inserts didn't show
ifferent results, I'll create inserts that include a product with sales within the range and another with sales outside, and explain that the queries are the same. \n\nBut the user's instruction is to show different results. \n\nI'm stuck. I'll proceed by creating inserts that include a product with sales within
he range and another with sales outside, and then the queries would return the same results. \n\nBut the user wants different results. \n\nPerhaps the correct answer is to create inserts that include a product with sales on the start date and another on the end date, and another with sales on the start and end da
es. \n\nBut I can't see how that would make the queries differ. \n\nIn conclusion, I'll proceed by creating inserts that include a product with sales within the range and another with sales outside, and then the queries would return the same results. But since the user wants different results, I'll have to assume
that there's a mistake in the queries and create inserts that show the queries are the same. \n\nBut the user's instruction is to show different results. \n\nI think I'll create inserts that include a product with sales on '2019-03-31' and another on '2019-04-01', and another on '2019-01-01' and '2019-03-31'. \n\
But I'm not sure. \n\nFinally, I'll proceed to create inserts that include a product with sales within the range and another with sales outside, and then the queries would return the same results. \n\nBut the user wants different results. \n\nI think I have to proceed with the given information and create inserts
that include a product with sales within the range and another with sales outside, even though the queries would behave the same. \n\nBut since the user's error message says that the previous inserts didn't show different results, I'll create inserts that do show different results. \n\nMaybe the difference is in
he way the conditions are structured. For example, query1 requires that all sales are within the range, while query2 requires that there's at least one sale in the range and no sales outside. But this is the same. \n\nI think I'll proceed with the following inserts:\n\nInsert a product with a sale on '2019-03-31'
(within the range) and another on '2019-04-01' (outside the range). \n\nBut both queries would exclude this product. \n\nAnother product with a sale on '2019-01-01' (within range) and '2019-03-31' (within range). Both queries include it. \n\nBut this doesn't show different results. \n\nI think I have to proceed w
& :
```

Fig. 20. Example of "thinking truncation" failure. The model's `<think>` block consumes the entire 8k token generation budget with extensive SQL analysis—considering NULL handling, date comparisons, edge cases, and multiple solution strategies. The reasoning is truncated mid-sentence (visible at bottom), never producing the required INSERT statements. This explains why Hyper-Reflection prompts degraded SQL Analysis performance: they induced more thorough reasoning that exceeded the available token budget.



qwen3-8b		bench	SQLAnalysis1Step	improvement
opt	teacher_model	rprompt		
Baseline			45.00	• +0.00
		default0	67.50	▲ +22.50
		default	70.00	▲ +25.00
GEPA	qwen3-8b	opus1	62.50	▲ +17.50
		gemnitf1	47.50	▲ +2.50
		gpt5think1	65.00	▲ +20.00



qwen3-8b-hc		bench	SQLAnalysis1Step	improvement
opt	teacher_model	rprompt		
Baseline			65.00	• +0.00
		default0	62.50	▼ -2.50
		default	55.00	▼ -10.00
GEPA	qwen3-8b-hc	opus1	75.00	▲ +10.00
		gemnitf1	62.50	▼ -2.50
		gpt5think1	70.00	▲ +5.00

Fig. 21. Task success with YaRN high context configuration on SQL Analysis 1-Step. **Upper table (standard 8k context):** Results reproduced from earlier experiments showing baseline at 45%, default at 70%, Hyper-Reflection prompts underperforming. **Lower table (40k high context):** Baseline improves to 65% (+20 from extended context alone); GEPA default drops to 55% (optimized for shorter contexts); `opus1` achieves state-of-the-art 75% (+30 over standard baseline, +10 over high-context baseline). Results validate that Hyper-Reflection prompts require adequate generation budget to unlock their potential.

a system—which we tentatively call LEAP (Learning from Evolved and Augmented Prompts)—would distill the behavior of expensive search-based optimizers into lightweight, deployable prompt rewriters.

Key research questions include: How do synthetic preference data from optimizers interact with human preference datasets? What is the sample efficiency of trajectory-based learning compared to pure search? Can models learn generalizable prompt improvement strategies that transfer across tasks?

VIII. Conclusion

We demonstrated that Automatic Prompt Optimization is a viable strategy for enhancing compact LLMs on verifiable SQL tasks. Key findings include:

1) **Feedback Boosts Success:** Feedback from the database engine (1-step vs 2-step vs 3-step flows) provides immense performance gains. For SQL Synthesis, baseline success increased from 20% to 93%; for SQL Analysis, from 45% to 60%.
2) **Thinking Truncation:** We identified a critical failure mode where excessive `<think>` blocks consume the generation budget, truncating actual answers. This explains why "smarter" prompts can paradoxically degrade performance without sufficient context.
3) **Reasoning Requires Budget:** Complex prompts generated by APO induce long Chain-of-Thought sequences. Standard context limits (8k tokens) can be insufficient; extending context via techniques like YaRN [4] is necessary to unlock the performance of optimized prompts.
4) **Hyper-Reflection Works:** Optimizing the reflection template via frontier models (`opus1`, `gemnitf1`) yields superior instructions compared to human-written defaults, provided the student model has the capacity to execute them. The SQL Synthesis task was essentially solved (98.33% success) after applying Hyper-Reflection.

With these optimizations, we achieved a 75% success rate on SQL Analysis Task (Counterexample Discovery) using a compact 8B model, representing a 30-point improvement over baselines. Our Hyper-Reflection technique offers a general approach for meta-optimization that could be applied to other prompt evolution systems.

References

[1] K. Chang, K. Xu, J. Wang, Z. Wei, W. Zhang, and X. Song, "Efficient prompting methods for large language models: A survey," *arXiv preprint arXiv:2404.01077*, 2024. [Online]. Available: https://arxiv.org/abs/2404.01077

[2] L. A. Agrawal, S. Tan, D. Soylu, N. Ziems, R. Khare, K. Opsahl-Ong, A. Singhvi, H. Shandilya, M. J. Ryan, M. Jiang, C. Potts, K. Sen, A. G. Dimakis, I. Stoica, D. Klein, M. Zaharia, and O. Khattab, "GEPA: Reflective prompt evolution can outperform reinforcement learning," *arXiv preprint arXiv:2507.19457*, 2025. [Online]. Available: https://arxiv.org/abs/2507.19457

[3] R. S. Sutton, "The bitter lesson," http://www.incompleteideas.net/IncIdeas/BitterLesson.html, 2019, accessed: 2025. [Online]. Available: http://www.incompleteideas.net/IncIdeas/BitterLesson.html

[4] B. Peng, J. Quesnelle, H. Fan, and E. Shippole, "YaRN: Efficient context window extension of large language models," in *The Twelfth International Conference on Learning Representations (ICLR)*, 2024. [Online]. Available: https://openreview.net/forum?id=wgbNgO9BJC

[5] G. E. Hinton, O. Vinyals, and J. Dean, "Distilling the knowledge in a neural network," *arXiv preprint arXiv:1503.02531*, 2015. [Online]. Available: https://arxiv.org/abs/1503.02531

[6] N. Ding, Y. Qin, G. Yang, F. Wei, Z. Yang, Y. Su, S. Hu *et al.*, "Parameter-efficient fine-tuning of large-scale pre-trained language models," *Nature Machine Intelligence*, vol. 5, no. 3, pp. 220–235, 2023. [Online]. Available: https://doi.org/10.1038/s42256-023-00626-4

[7] X. L. Li and P. Liang, "Prefix-tuning: Optimizing continuous prompts for generation," in *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics (ACL)*. Association for Computational Linguistics, 2021, pp. 4582–4597. [Online]. Available: https://aclanthology.org/2021.acl-long.353/

[8] E. J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, and W. Chen, "LoRA: Low-rank adaptation of large language models," in *The Tenth International Conference on Learning Representations (ICLR)*, 2022. [Online]. Available: https://openreview.net/forum?id=nZeVKeeFYf9

[9] A. Petrov, P. H. S. Torr, and A. Bibi, "When do prompting and prefix-tuning work? a theory of capabilities and limitations," in *The Twelfth International Conference on Learning Representations (ICLR)*, 2024. [Online]. Available: https://openreview.net/forum?id=1Nf57q3L8q

[10] R. Pryzant, D. Iter, J. Li, Y. J. Lee, C. Zhu, and M. Zeng, "Automatic prompt optimization with "gradient descent" and beam search," in *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Association for Computational Linguistics, 2023, pp. 7957–7968. [Online]. Available: https://aclanthology.org/2023.emnlp-main.494/

[11] T. Shin, Y. Razeghi, R. L. Logan IV, E. Wallace, and S. Singh, "Autoprompt: Eliciting knowledge from language models with automatically generated prompts," in *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2020.

[12] Y. Zhou, A. I. Muresanu, Z. Han, K. Paster, S. Pitis, H. Chan, and J. Ba, "Large language models are human-level prompt engineers," in *The Eleventh International Conference on Learning Representations (ICLR)*, 2023. [Online]. Available: https://openreview.net/forum?id=92gvk82DE-

[13] C. Yang, X. Wang, Y. Lu, H. Liu, Q. V. Le, D. Zhou, and X. Chen, "Large language models as optimizers," in *The Twelfth International Conference on Learning Representations (ICLR)*, 2024. [Online]. Available: https://openreview.net/forum?id=gEAO70mQ2U

[14] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W.-t. Yih, T. Rocktäschel, S. Riedel, and D. Kiela, "Retrieval-augmented generation for knowledge-intensive NLP tasks," in *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 33, 2020, pp. 9459–9474. [Online]. Available: https://proceedings.neurips.cc/paper/2020/hash/6b493230205f780e1bc26945df7481e5-Abstract.html

[15] J. Wei, X. Wang, D. Schuurmans, M. Bosma, B. Ichter, F. Xia, E. Chi, Q. Le, and D. Zhou, "Chain-of-thought prompting elicits reasoning in large language models," in *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 35, 2022, pp. 24 824–24 837. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2022/hash/9d5609613524ecf4f15af0f7b31abca4-Abstract-Conference.html

[16] K. Opsahl-Ong, M. J. Ryan, J. Purtell, D. Broman, C. Potts, M. Zaharia, and O. Khattab, "Optimizing instructions and demonstrations for multi-stage language model programs," in *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Association for Computational Linguistics, 2024. [Online]. Available: https://aclanthology.org/2024.emnlp-main.525/

[17] A. Sordoni, E. Yuan, M.-A. Côté, M. Pereira, A. Trischler, Z. Xiao, A. Hosseini, F. Niedtner, and N. Le Roux, "Joint prompt optimization of stacked LLMs using variational inference," in *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 36, 2023, pp. 51 603–51 619. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2023/hash/a1eb1d7f6c32338c0f3d6118d0b263b6-Abstract-Conference.html

[18] T. Li *et al.*, "C-Evolve: Consensus-based evolution for prompt groups," *arXiv preprint arXiv:2509.23331*, 2025, under review for ICLR 2026. [Online]. Available: https://arxiv.org/abs/2509.23331

[19] W. Wang *et al.*, "Maestro: Joint graph & config optimization for reliable AI agents," *arXiv preprint arXiv:2509.04642*, 2025. [Online]. Available: https://arxiv.org/abs/2509.04642

[20] Y. Lee, J. Boen, and C. Finn, "Feedback descent: Open-ended text optimization via pairwise comparison," *arXiv preprint arXiv:2511.07919*, 2025, under review for ICLR 2026. [Online]. Available: https://arxiv.org/abs/2511.07919

[21] Q. Zhang *et al.*, "Agentic context engineering: Evolving contexts for self-improving language models," *arXiv preprint arXiv:2510.04618*, 2025. [Online]. Available: https://arxiv.org/abs/2510.04618

[22] Y. He, P. Zhao, X. Wang, and Y. Wang, "VeriEQL: Bounded equivalence verification for complex SQL queries with integrity constraints," *Proc. ACM Program. Lang. (OOPSLA)*, vol. 8, no. OOPSLA1, 2024. [Online]. Available: https://doi.org/10.1145/3649849

[23] O. Khattab, A. Singhvi, P. Maheshwari, Z. Zhang, K. Santhanam, S. Vardhamanan, S. Haq, A. Sharma, T. T. Joshi, H. Mober, R. Nogueira, M. Zaharia, and C. Potts, "DSPy: Compiling declarative language model calls into self-improving pipelines," in *The Twelfth International Conference on Learning Representations (ICLR)*, 2024. [Online]. Available: https://openreview.net/forum?id=s859lp5Y7F

[24] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. E. Gonzalez, H. Zhang, and I. Stoica, "Efficient memory management for large language model serving with PagedAttention," in *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP)*. ACM, 2023, pp. 611–626. [Online]. Available: https://doi.org/10.1145/3600006.3613165

[25] J. Cheng, X. Liu, K. Zheng, P. Ke, H. Wang, Y. Dong, J. Tang, and M. Huang, "Black-box prompt optimization: Aligning large language models without model training," in *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (ACL)*. Association for Computational Linguistics, 2024, pp. 3029–3046. [Online]. Available: https://aclanthology.org/2024.acl-long.176/

Appendix

A. vLLM Launch Configurations

Standard Configuration (8k context):

```
vllm serve Qwen/Qwen3-8B \
    --download-dir /workspace/models \
    --host 127.0.0.1 \
    --port 18000 \
    --max-num-seqs 128 \
    --enable-prefix-caching \
    --enable-chunked-prefill \
    --max-long-partial-prefills 1 \
    --tokenizer-pool-size 8 \
    --tokenizer-pool-type ray \
    --enable-reasoning \
    --reasoning-parser qwen3 \
    --override-generation-config \
    '{"max_new_tokens":8192}'
```

Transformer-based models are limited by their pre-trained context length, which proved to be a critical bottleneck in our experiments. YaRN (Yet another RoPE extensioN) [4] provides a compute-efficient method to extend context windows, requiring 10× fewer tokens and 2.5× fewer training steps than previous methods. YaRN combines NTK-aware interpolation with attention scaling, enabling models to extrapolate beyond their original training length while preserving performance on standard benchmarks.

High Context Configuration with YaRN (40k context):

```
vllm serve Qwen/Qwen3-8B \
    --download-dir /workspace/models \
    --host 127.0.0.1 \
    --port 18000 \
    --max-num-seqs 128 \
    --max-model-len 90112 \
    --enable-prefix-caching \
    --enable-chunked-prefill \
    --max-long-partial-prefills 1 \
    --tokenizer-pool-size 8 \
    --tokenizer-pool-type ray \
    --enable-reasoning \
    --reasoning-parser qwen3 \
    --override-generation-config \
    '{"max_new_tokens":40960}' \
    --rope-scaling \
    '{"rope_type":"yarn", "factor":2.75, "
    original_max_position_embeddings": 32768}'
```

The key differences in the high context configuration are:

- `max-model-len 90112`: Extends the total context window
- `max_new_tokens: 40960`: Allows generation up to 40k tokens
- `rope-scaling`: Applies YaRN with factor 2.75 to interpolate beyond the original 32k position embeddings