Extended Abstract

Motivation Improving LLM performance across various tasks is essential as models are increasingly deployed in complex real-world applications. Inference-time prompt optimization has emerged as a key approach for improving performance without modifying model weights, demonstrated by DSPy, a framework for robust LLM programming. However, existing techniques like random search across prompts and instructions often lack generalizability and are inefficient. Existing optimizers treat each prompt candidate independently, failing to learn from past strategy successes and failures, a gap that reinforcement learning could fill and help improve performance.

Method We introduce ARPO (Adaptive Reward-driven Prompt Optimization), a RL-based optimization technique that applies adaptive policy learning and credit assignment within the prompt optimization loop. The ARPO approach has three variants: ARPO-Policy, which uses a learned RL policy to dynamically select between prompt-editing strategies (in-context learning or instruction tuning), ARPO-Ledger, which assigns and accumulates credit to individual prompt components (demonstrations and rules) based on their observed utility, and ARPO-Hybrid, which combines both strategies. The ARPO pipeline iteratively samples and evaluates candidate prompts, updating strategy and component values based on observed rewards. This process converges toward high-performing prompt configurations by leveraging cumulative feedback signals.

Implementation ARPO is implemented within the DSPy framework, building upon the existing SIMBA optimizer and benchmarked against a true unoptimized baseline prompt with no prompt optimization and then existing DSPy optimization techniques (FewShotRandomSearch, MIPROv2, SIMBA) as additional baseline comparisons. We evaluate the ARPO variants on the LangProBe benchmark, spanning tasks in classification, coding, QA, medical, math, and reasoning, and across multiple Llama model sizes (3B, 8B, 70B) and student-teacher (3B-70B, 8B-70B) setups. Evaluation metrics include exact match and retrieval recall, as well as token usage to assess computational cost.

Results Empirical evaluation shows that ARPO variants achieve competitive performance across a wide range of tasks, with ARPO-Hybrid attaining the highest upper-bound improvement of 95.7% and all ARPO methods maintaining strong median gains. ARPO outperforms existing optimizers by 11–25% across the evaluation suite and consumes 50% fewer tokens during optimization, placing it on the favorable end of the performance-cost Pareto frontier. While different ARPO optimizers perform well on different tasks (ARPO-Ledger best on biomedical QA, classification, ARPO-Policy best on reasoning, ARPO-Hybrid best on coding and classification), the ARPO approach overall does not dominate all benchmarks, and gains from including larger teacher models for optimization remain insignificant compared to other optimizers.

Discussion ARPO's adaptive approach enables strong gains in structured tasks and even in complex prompting pipelines like retrieval-augmented generation and multi-hop QA, outperforming SIMBA and approaching existing state-of-the-art optimizers' performance with fewer token consumption. However, ARPO has some drawbacks in having limited benefit when using teacher models which generate edits that student models struggle to leverage compared to the other optimizers. These challenges highlight the need for representation-aware credit assignment and more structured exploration strategies. Overall, ARPO shows that adaptive optimization can be both effective and efficient, and further exploration of adding more prompt edit strategies and budget-aware evaluation comparisons can highlight where ARPO performs best on the axes of LLM optimization performance comparisons.

Conclusion ARPO provides a novel approach to leveraging RL in automated prompt optimization to drive adaptive, reward-driven methods that are both computationally efficient and broadly applicable to LLM benchmark tasks. While the results show that ARPO is not the universally best, we demonstrate how ARPO is still competitive in optimal and near-optimal performance with other optimizers and indicate that no single optimizer was universally best either. We demonstrate how ARPO is positioned high on the Pareto frontier, showing its computationally feasible applications while maintaining competitive performance. Overall, ARPO's employment of RL reflects a novel approach into adaptive automated prompt optimization, which can be further explored with the addition and integration of other RL-based strategies to improve LLM prompting pipelines.

ARPO: Adaptive Reward-driven Prompt Optimization

Arnav Singhvi
Department of Computer Science
Stanford University
arnavs11@stanford.edu

Shreyas Agarwal
Department of Computer Science
Stanford University
shrey15@stanford.edu

Abstract

Improving large language models' performance commonly relies on manual prompt engineering, which lacks generalizability to real-world applications. Automated prompt optimization techniques for improving LLM behavior have emerged as effective strategies, yet can often be computationally-expensive, relying on random search explorations to determine optimal prompts. We introduce ARPO (Adaptive Reward-driven Prompt Optimization), a reinforcement learning-based method for optimizing prompts in LLM pipelines through adaptive policy learning and credit assignment that leverage the reward signal to iteratively converge to an optimized prompt. We develop and evaluate three variants: ARPO-Policy learns which element (few-shot example and instruction rule) to tweak, ARPO-Ledger tracks component-level credit and prunes low-impact pieces, and ARPO-Hybrid combines both signals. ARPO is built within the DSPy optimization framework to compare against existing optimization techniques and demonstrates competitive performance across LLM benchmarks in the LangProBe evaluation suite, outperforming existing optimizers by 11-25% and achieving upper-bound gains of up to 95.7%. Additionally, ARPO uses 50% fewer tokens, positioning it highly on the Pareto frontier as a computationally-efficient optimization approach.

1 Introduction

Large language models (LLMs) have increasingly become integral components of AI systems applied on natural language tasks that span complex decision-making Cui et al. (2025), domain-specific reasoning Chen et al. (2024), and interactive dialogue systems Hong et al. (2024). Effectively applying LLMs fundamentally depends on the quality of information flow communicated between the user-defined tasks and the model's capabilities, which currently is done through prompting Sahoo et al. (2025). While existing techniques of prompt engineering serve to enhance these capabilities, achieving optimal performance relies on trial-and-error and manual prompt edits that are not generalizable to new domains, tasks, or even changing models with evolving capabilities Cao et al. (2024). These gaps highlight the inherent limitations of manually engineered prompting techniques, leaving such systems brittle and unscalable for newer models and capabilities.

To address this gap, recent research has explored ways to automate prompt improvement through prompt optimization. Frameworks like OPRO Yang et al. (2023); Zhang et al. (2024) and QPO Kong et al. (2024b) guide model behavior with iterative prompt refinement to improve downstream task performance. DSPy Khattab et al. (2023) has emerged as a framework that supports building robust AI systems through modules such as *Predict*, *CoT*, and *Baleen* Stanford-NLP-Group ([n. d.]); Wei et al. (2022); Khattab et al. (2021) and improving these systems through optimizers like FewShotRandomSearch, MIPROv2 (Multiprompt Instruction PRoposal Optimizer) Opsahl-Ong et al. (2024) and the forthcoming SIMBA (Stochastic Introspective Mini-Batch Ascent) that apply common prompt engineering techniques like in-context learning and instruction tuning. While existing DSPy optimizers reflect state-of-the-art performance on open-source benchmarks and closed-source production use cases with significant improved performance compared to unoptimized baselines, these techniques predominantly rely on static, heuristic-based search strategies that independently evaluate prompt candidates without dynamically leveraging accumulated feedback from prior optimization rounds, resulting in inefficient exploration with repeated evaluation of redundant candidates. This results in such prompt optimizers being limited by shallow search depth and at times, the inability for optimization to generalize improvements across iterations.

In this work, we propose a novel optimization approach called **A**daptive **R**eward-driven **P**rompt **O**ptimization (ARPO), which integrates reinforcement learning strategies into the DSPy optimization process to improve LLM workflows. ARPO differs from the existing heuristic-based methods by dynamically adapting its prompt optimization strategy using reward-driven signals accumulated over iterative optimization steps. This enables ARPO to identify patterns in model trajectories during optimization to determine which prompt components work better and perform smarter exploration and generalization within the iteration before outputting a final optimized system. ARPO comprises of three adaptive configurations designed to intelligently manage and refine prompt components, such as few-shot demonstrations and induced guiding rules based on observed utility:

- **ARPO-Policy**: Employs an RL-based adaptive policy to guide prompt edits by dynamically learning the optimal selection strategy between demonstrations and rules.
- **ARPO-Ledger**: Maintains a credit assignment ledger that tracks individual prompt component performance, enabling pruning of lower-impact demonstrations or rules.
- **ARPO-Hybrid**: Combines the strengths of both adaptive policy guidance and ledger-based credit assignments to simultaneously select and prune prompt components.

We assess ARPO's effectiveness using the LangProBe benchmark suite Tan et al. (2025), a comprehensive evaluation framework within DSPy that spans diverse LLM benchmarks including classification, coding, QA, medicine, math, and reasoning tasks. We compare ARPO against the existing DSPy optimizers, demonstrating ARPO's capability to continuously learn from cumulative feedback signals and effectively manage complex prompt optimization decisions using reinforcement learning . Our experiments indicate that ARPO consistently achieves significant improvements in downstream task performance metrics, outperforming its baseline SIMBA optimizer by up to 11% and other existing optimizers by up to 24% with around 50% less token consumption.

2 Related Work

Recent advances in prompt optimization techniques have explored the paradigm of optimizing language models themselves. Approaches like Optimization by PROmpting (OPRO) Yang et al. (2023); Zhang et al. (2024) leverage LLMs to iteratively refine prompts through candidate scoring, reducing the need for manual prompt engineering. However, this approach has limitations, including degraded performance with smaller LLMs and the absence of sequential decision-making, hindering adaptation based on cumulative feedback. Similarly, Multi-prompt Instruction Proposal Optimizer (MIPROv2) Opsahl-Ong et al. (2024) enhances this concept by jointly optimizing instruction templates and few-shot examples using Bayesian optimization, effectively improving zero and few-shot configurations. However, MIPROv2 is constrained by batch evaluations of static prompts and lacking adaptive policies for managing prompt components dynamically over successive iterations.

Applying reinforcement learning to prompt optimization is another approach being explored. Methods such as Prompt Rewriting with Reinforcement Learning (PRewrite) Kong et al. (2024a) utilize RL, particularly Proximal Policy Optimization (PPO), to train models that iteratively refine prompts for improved task performance. This method effectively automates prompt improvements but is primarily limited to single-stage prompts, lacking mechanisms for handling combinatorial complexity in multi-module pipelines. Query-dependent Prompt Optimization (QPO) Kong et al. (2024b) addresses query-specific adaptations through offline RL policies, which generate optimized prompts incorporating context-specific few-shot examples. Although effective at the single-module level, QPO has yet to extend to complex, multi-stage pipelines. Similarly, ParetoPrompt Zhao (2025) employs RL to optimize prompts across multiple objectives simultaneously, using pairwise comparisons to balance trade-offs. While this approach successfully generates prompts beneficial to diverse objectives, it falls short in providing holistic evaluations of different prompt components like instructions and few-shot examples to yield generalizable optimizations.

3 Method

3.1 Backbone Framework: DSPy and SIMBA optimizer

We adopt the DSPy SIMBA (Stochastic Introspective Mini-Batch Ascent) optimizer as the foundational backbone for developing our approach due to its modular, strategy-driven design for prompt optimization. SIMBA optimizes prompts stochastically choosing between (1) appending few-shot demonstrations from training data or (2) injecting guiding rules that are induced from program components like the data, task objective and passing and negative examples into the baseline prompt, and does so for a possible candidate which is then evaluated in batches until a final canidtdae is determined as the final optimized program. This makes SIMBA an ideal starting point for introducing reinforcement learning, as its decision points of which strategy to apply and when naturally lend themselves to policy learning. Since SIMBA explores candidates purely through random exploration, the optimizer currently fails to leverage feedback from previous optimization steps, limiting its ability to learn effective strategies cumulatively over time.

3.2 ARPO Methodology

To address SIMBA's limitations, we introduce Adaptive Reward-driven Prompt Optimization (ARPO), a RL-based methodology that dynamically adjusts prompt optimization strategies based on cumulative reward signals, layering in variations of RL-based techniques onto SIMBA's decision structure. This helps improve prompt configurations by explicitly learning optimal strategies for managing demonstrations and rules through adaptive feedback.

The ARPO optimizer comprises 3 variants:

- **ARPO-Policy: Adaptive Prompt Mutation via Learned Policies** Candidate prompts undergo modifications guided by an adaptive RL policy, dynamically selecting editing strategies based on their historical effectiveness. This learned policy continuously assesses the relative performance of demonstrations and rules, thus optimizing their selection.
- **ARPO-Ledger: Credit Assignment and Continuous Refinement** ARPO employs a credit assignment mechanism, scoring each demonstration and rule individually based on observed performance improvements. These cumulative credits facilitate intelligent pruning, systematically removing lower-impact components while retaining consistently beneficial ones.
- **ARPO-Hybrid: Integrated Policy and Ledger Approach** The ARPO-Hybrid configuration combines policy-driven adaptive selection with ledger-based credit tracking. This integration ensures robust and precise optimization by simultaneously employing learned strategies for component selection and continuously refining component choices based on accumulated credits.

3.3 ARPO Pipeline

ARPO optimization iteratively refines candidate prompts through RL-driven updates. (Algorithm Figure in Appendix B)

1. Trajectory Sampling: Each iteration begins by sampling candidate prompt configurations, generated by cloning the unoptimized downstream task with varying random seeds and temperatures. Candidates are then selected probabilistically using a softmax over their average historical rewards. ARPO-Policy and ARPO-Ledger base this sampling on raw reward values, while ARPO-Hybrid further integrates ledger-based credit scores to prioritize high-utility candidates during selection.

2. Continuous Strategy and Component Updates After evaluating candidate prompts, ARPO performs targeted updates to both strategy-level decisions and component-level utility scores:

- **Strategy Value Updates:** In ARPO-Policy and ARPO-Hybrid, the strategy preferences are adjusted based on observed performance gains over baselines, enabling adaptive exploration.
- **Credit Assignment Updates:** In ARPO-Ledger and ARPO-Hybrid, individual demonstrations and rules are assigned cumulative credits based on their contribution to downstream performance, facilitating ongoing refinement and selective pruning.

3. Reinforcement Learning Updates: After evaluating each candidate on mini-batches, ARPO updates its internal learning mechanisms. For ARPO-Policy, this involves updating strategy selection probabilities using observed rewards. For ARPO-Ledger, credit scores are incrementally refined based on the performance impact of individual components. ARPO-Hybrid combines both strategies.

4. Convergence to Final Optimized Program: Over successive iterations over the candidate pool, ARPO converges towards an optimal prompt configuration with the highest achieved overall performance and returns this for evaluation on the test data.

4 Experimental Setup

We use the LangProBe evaluation suite of LLM benchmark tasks. For datasets with over 1000 examples, we use default randomized splits: 150 for train, 300 for val (during optimization), and 500 for test. For smaller datasets, we apply custom splits listed below:

Category	Dataset Description	DSPy Program(s)
Classification	**HeartDisease**Detrano et al. (1989): Predict heart disease from patient data - binary classification(Dataset: 303 total, 15 train, 136 val, 152 test).	*Predict*, *CoT*
	IrisFisher (1936): Classify flower type species (setosa, virginica, versicolor) by measurements (Dataset: 150 total, 15 train, 60 val, 75 test)	*Predict*, *CoT*
Coding	**HumanEval**OpenAI (2021): Generate code for small programming tasks (Dataset: 164 total, 15 train, 67 val, 82 test)	*Predict*, *CoT*
	SWEBenchVerifiedNLP (2023): Code fixes for real GitHub issues.	*Predict*, *CoT*
Knowledge-based QA	**HoVeR**Jiang et al. (2020): Claim verification with multi-document evidence.	*Predict*, *CoT*, *Baleen*
	HotPotQAYang et al. (2018): Multi-hop questions over Wikipedia content.	*Predict*, *CoT*, *RAG*, *Baleen*
	IReRaD'Oosterlinck et al. (2024): Label job descriptions from 14k skills (multi-label classification).	*Predict*, *CoT*
	MMLUHendrycks et al. (2021): MCQs (A-D) spanning general knowledge.	*Predict*, *CoT*, *RAG*, *Baleen*
Medical	**MedMCQA**Pal et al. (2022): MCQs from Indian medical entrance exams.	*Predict*, *CoT*
	MedQAJin et al. (2021): MCQs from US medical board exams.	*Predict*, *CoT*
	PubMedQAJin et al. (2019): Yes/no/maybe QA on biomedical abstracts.	*Predict*, *CoT*
Math	**GSM8K**Cobbe et al. (2021): Word problems requiring numeric reasoning.	*Predict*, *CoT*
Reasoning	**JudgeBench**Tan et al. (2024): Compare pairwise responses for factual accuracy.	*Predict*, *CoT*
	SconeNLIShe et al. (2023): Binary QA on negation-sensitive sentence pairs.	*Predict*, *CoT*

4.1 Evaluation Method

We evaluate methods on four canonical prompting modules and pipelines written in DSPy:

1. *Predict* Stanford-NLP-Group ([n. d.]): sends instructions and inputs to LLM, returns generation.
2. *ChainOfThought* Wei et al. (2022): appends "Think step by step" to the prompt so that the LLM produces a reasoning trace before providing the final answer.
3. *Retrieval-Augmented Generation (RAG)* Lewis et al. (2020): uses an external retriever to retrieve evidence passages so the LLM has relevant information when producing output
4. *Multi-Hop Retrieval* Khattab et al. (2021): first asks the LM to write one or more search queries, retrieves the resulting passages with the external retriever, and then applies RAG.

Each task pipeline is run once with a specified default baseline prompt (**Unoptimized**) and then with all six optimizers, using the train and validation sets for optimization and the test set for final evaluation, which is used for all reported performance comparisons (hyperparameters Appendix B):

- **BootstrapFewShotRandomSearch** *(existing)*
  Produces few-shot examples that pass the task metric through bootstrapped LLM runs, then performs random search over subsets and returns the best-performing combination.
- **MIPROv2 - Multiprompt Instruction PRoposal Optimizer)** *(existing)*
  Uses an LLM to rewrite the downstream task instructions and considers various candidates that are iteratively optimized with Bayesian optimization while also including few-shots.
- **SIMBA - Stochastic Introspective Mini-Batch Ascent** *(existing)*
  Performs multi-step stochastic optimization by applying random edit strategies (add demo or rule) to prompt configurations and evaluating over mini-batches
- **ARPO - Policy, Ledger & Hybrid** *(new)*

4.2 Metrics and Analysis

We evaluate task performance using **Exact Match**, except for HoVeR which uses retrieval recall. To measure **performance**, we track how often each optimizer achieves the best or near-best result (within 3% of the maximum), and summarize relative performance gains at the 10th, 50th, and 90th percentiles across tasks to capture worst-case, typical, and best-case improvements. For **cost**, we log total input/output tokens used during optimization and final evaluation to account for optimization cost. Finally, we assess **performance-cost tradeoffs** by plotting Pareto curves showing overall performance gain against token cost, considered over all configurations of tasks and models tested for the datasets for each optimizer.

4.3 Experimental Details

We conduct all experiments on a 4xNVIDIA A100 GPU cluster, using open-source Llama models accessed through the SGLang framework Zheng et al. (2024). The models are of varying parameter sizes: Llama3.2-**3B** AI (2024b), Llama3.1-**8B** AI (2024a), Llama3.3-**70B** AI (2024c)

All models are called using `DSPy.LM` Stanford-NLP-Group (2024) using LiteLLM LiteLLM (2024), with endpoints such as `openai/meta-llama/Meta-Llama-3.1-8b-Instruct`. For consistency, we fix the temperature at 0.0 and limit outputs to 1000 tokens.

Student–Teacher Configuration. We evaluate five model configurations: `3B-3B`, `3B-70B`, `8B-8B`, `8B-70B`, and `70B-70B`. For the differing-model setups, the larger model acts as a "teacher" performing optimization, while the smaller student model learns this behavior and applies at inference. We denote these cases with the optimizer suffix `-T` (e.g., `ARPO-T`). For `70B-70B`, no teacher is used.

5 Results

Our analysis covers all optimizers' performance across all tasks and model setups. First, we track how often each optimizer achieves either the top score or lands within 3% of it (Figure 1). Next, we highlight the spread of performance improvements relative to the unoptimized baseline, highlighting outcomes at the 10th, 50th, and 90th percentiles (Figure 2). Finally, we examine token efficiency by plotting each method's performance gain against its associated token usage, revealing cost-performance tradeoffs across model sizes and optimizer types (Figure 3). Additionally, we include model-specific breakdowns to better highlight different axes and configurations on where which optimizers succeed (A) in the appendices.

5.1 Quantitative Evaluation

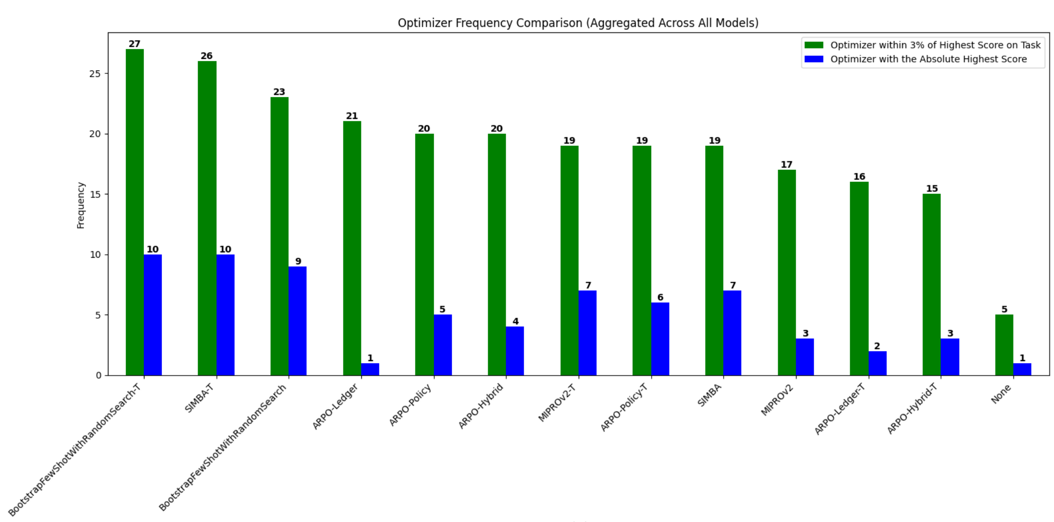

Figure 1: Global Optimizer Frequency Comparison

5.1.1 No Single Optimizer Dominates, and ARPO is a Competitive Optimizer

The global optimizer frequency comparison (Figure 1) demonstrates that no single optimization approach works best across all tasks and model configurations. We do find that ARPO variants outperform the direct baseline SIMBA. Specifically, ARPO-Ledger achieves performance within 3% of the highest score on 21 tasks, compared to SIMBA's 19 tasks, representing a 10.5% improvement with ARPO-Policy and ARPO-Hybrid also achieving near-optimal performance on 20 tasks.

When examining absolute highest scores on the tasks, ARPO-Policy achieves the best performance on 5 tasks compared to SIMBA's 7 tasks, while ARPO-Ledger and ARPO-Hybrid achieve peak performance on 1 and 4 tasks, respectively. This suggests that while ARPO variants may not always achieve the absolute optimum, they consistently achieve near-optimal performance, which is often more valuable in practical applications.

When comparing ARPO to the broader optimizer landscape, we find that ARPO falls short of BootstrapFewShotWithRandomSearch (23 tasks) but does outperform MIPROv2 (17 tasks) and naturally no optimizer (5). However, ARPO doesn't demonstrate gains with the teacher model, whereas optimizers like BootstrapFewShotWithRandomSearch-T (27 tasks) and SIMBA-T (26 tasks) achieve the highest frequency of near-optimal performance. Overall, this competitive ranking validates ARPO's adaptive RL approach as a viable alternative to existing heuristic strategies.

5.1.2 ARPO-Hybrid Emerges as the Most Reliable Variant within ARPO optimizers

Among the three ARPO variants, ARPO-Hybrid performed the best across measured dimensions. The percentile distribution analysis (Figure 2) reveals that ARPO-Hybrid achieves significant upper-bound performance with a 90th percentile gain of 95.7%, significantly outperforming other variants in upper-bound gains, while also maintaining a relatively low lower-bound degradation (as at times prompt optimizers can result in regressions and potential overfitting that lower task performance) compared to other ARPO variants and other existing optimizers.

ARPO-Ledger also achieves competitive upper-bound gain at 75.7% and 9.7% median performance, but both are actually enhanced with introducing a larger teacher model at 88.9% and 11.3%, respectively, while ARPO-Hybrid did not benefit from including a teacher. ARPO-Policy performed the worst out of the variants with upper-bound gain of 58.5% and lower-bound loss of -2.7%, suggesting that the adaptive policy on its own struggled and led to greater volatility, while only improving the upper-bound gain with the teacher model, achieving 71.3%.

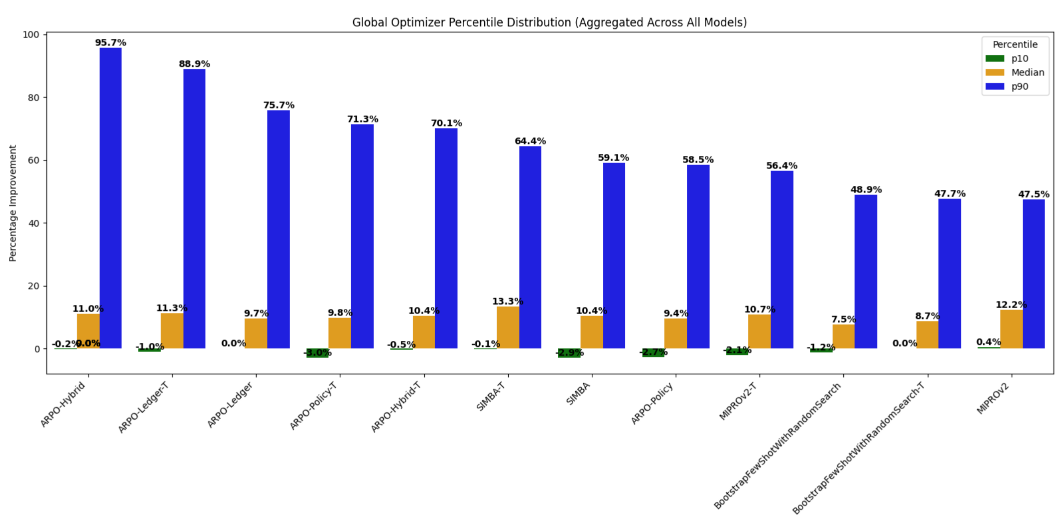

Figure 2: Global Optimizer Percentile Distribution

5.1.3 Teacher Models Improve Absolute Scores, but not Consistently

While we expected teacher models to always improve smaller student models performance, our results show that their benefits are more nuanced. Across ARPO variants, using a 70B teacher (-T) does not consistently increase (Figure 1) the likelihood of being near-optimal ARPO-Ledger, ARPO-Policy, and ARPO-Hybrid rank have 21, 20, and 20 tasks, respectively, within top 3% of best task scores whereas their teacher versions drop to 16, 19, and 15, respectively. However, teacher models do slightly improve the frequency of achieving the absolute best score, particularly for ARPO-Policy ($5 \rightarrow 6$) and ARPO-Ledger ($1 \rightarrow 2$). In percentile-based gains (Figure 2), we observe modest improvements for ARPO-Policy and ARPO-Ledger in both upper-bound and median gains when using a teacher, whereas ARPO-Hybrid sees declines in both as mentioned in the previous section.

These mixed results suggest that teacher models may help amplify peak performance but often at the cost of consistency. While larger LLMs can generate more sophisticated prompts during optimization, these improvements don't always transfer well during adaptive learning, potentially overfitting and failing to cleanly distill into smaller student models during evaluation. ARPO-Hybrid is especially sensitive to prompt structure, which can be undermined due to teacher-generated modifications that introduce conflicting signals from the policy or credit-assignment learning, leading to lower performance. In contrast, ARPO-Ledger and ARPO-Policy benefit more from teacher models due to their simpler yet structured optimization, which helps learn from complex prompts more effectively and retain generalizable gains, further highlighted in Appendix C.

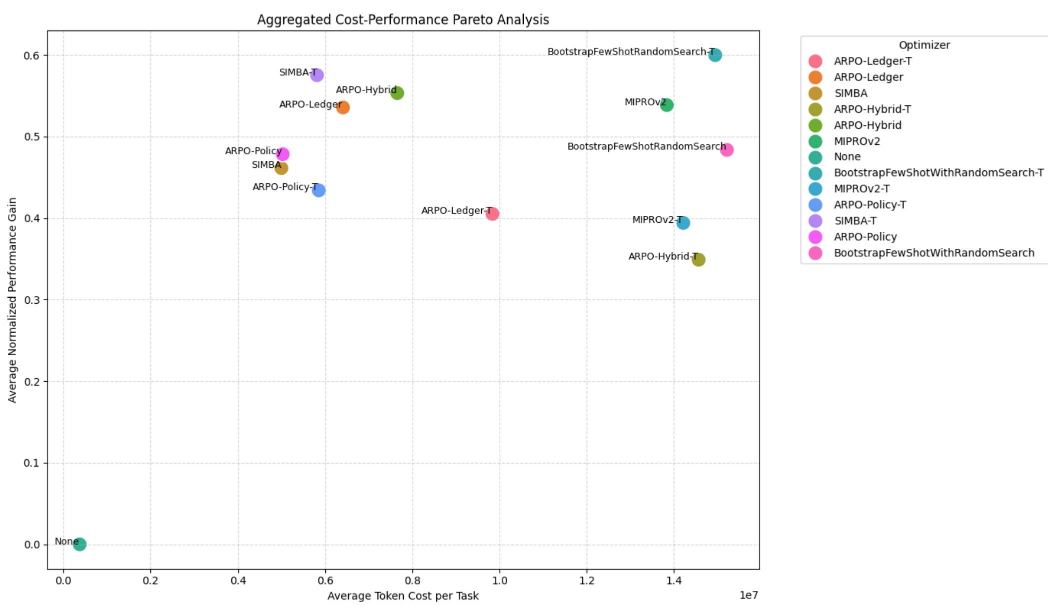

Figure 3: Pareto Frontier of Optimizer Cost vs. Performance

5.1.4 ARPO is Competitive on the Pareto Frontier

When plotting aggregate task metric scores (performance) against model average input and output token consumption (cost), (Figure 3) we find that ARPO variants are competitive on the Pareto frontier, achieving high performance gains while maintaining low token costs compared to other optimizers. ARPO-Ledger, ARPO-Policy, and ARPO-Hybrid generally cluster in the low-cost, moderate-to-high performance region with token costs ranging from approximately 5×10^6 to 7×10^6 tokens per task. This positions ARPO as the most efficient optimizers, requiring substantially fewer tokens than BootstrapFewShotWithRandomSearch (1.6×10^7 tokens) and MIPROv2 variants (1.4×10^7 tokens) while achieving comparable or superior performance gains.

ARPO's computational efficiency supports our hypothesis that RL techniques can outperform existing search-based heuristics by enabling more targeted exploration of the prompt space, converging faster to optimal prompt configurations. This is especially important when ARPO learns that using rules is more effective than few-shots, unlike SIMBA, which randomly samples between token-heavy demonstrations and lightweight rules, or FewShotRandomSearch and MIPROv2, which always include demonstrations regardless of utility. This context-aware cost-effectiveness positions ARPO as a strong candidate for efficient prompt optimization, especially in constrained budget environments.

5.2 Qualitative Analysis

Our qualitative analysis provides further insights into how the RL-based prompt editing and reward-based pruning in ARPO behaves across various task types.

5.2.1 Task-Specific Analysis Across ARPO Variants

ARPO-Ledger performs best on structured, deterministic tasks including biomedical QA (PubMedQA, MedQA, MedMCQA), basic classification (HeartDisease, Iris, IReRa), and factual reasoning tasks (JudgeBench, Scone), where the credit assignment isolated high-utility domain-specific few-shots and rules that improved performance. ARPO-Policy performed better on more complex multi-hop reasoning tasks (HotPotQA, MMLU, HoVeR), mathematical reasoning (GSM8K), and pairwise judgement (JudgeBench) as the adaptive learning was able to change the prompt editing approach for more dynamic tasks with less deterministic reasoning. ARPO-Hybrid naturally achieved optimal performance on a combination of such tasks, particularly excelling in coding tasks (HumanEval, SWEVerifiedAnnotationTask), mixed-reasoning QA scenarios (MMLU, HotPotQA), and classification tasks (IReRa, HeartDisease, Iris) with the ability to systematically filter and dynamically select prompt components that contribute to optimization success (Appendix C).

5.2.2 ARPO is Effective Across Complex Pipelines

ARPO performs well across most tasks in our evaluation suite, including complex prompting pipelines that require multiple LLM or external data source calls. It is particularly effective on multi-stage pipelines like Retrieval-Augmented Generation and multi-hop RAG pipelines like Baleen, where its adaptive strategies optimize each stage and combine to improve overall system performance. In some cases, such as HotPotQA, the generated rules were not especially helpful, ARPO could not produce better rules than existing optimizers. However, it successfully identified that few-shot examples using relevant passages led to better performance, demonstrating the benefits of adaptive prompt selection.

5.2.3 ARPO with Larger Teachers Only Improves Performance Slightly

Including a larger teacher model for optimization (e.g. 3B–70B) over the same model (e.g. 3B-3B) only led to marginal gains for ARPO. The larger model doesn't propose significantly better few-shot examples, and although its chain-of-thought traces are sometimes clearer, the reward signal that drives ARPO updates remains mostly unchanged compared to the same model performing optimization, offering no additional learning signal. In some cases, the larger model produces more grounded but overly general rules, while the smaller model offers more specific strategies that, counterintuitively, improve performance even if potentially overfitting (Appendix C).

Other optimizers benefit more from a stronger teacher because they use the teacher in different ways. MIPRO samples a diverse set of instruction candidates directly using the teacher, and Few-Shot Random-Search bootstrapping relies on the teacher to sample and evaluate in-context example subsets. SIMBA also improves with the larger teacher, but its gains come mainly from the random sampling choosing potentially better few-shots, rather than proposing more elaborate rules. ARPO's structure currently underutilizes the teacher model as it doesn't incorporate its capabilities fully into the reward function, so its impact on optimization is limited compared to other optimizers.

6 Discussion

Optimizing an LLM pipeline is not always just using the "best optimizer." Evaluating optimizers on different axes of LLM components highlights where which optimizer works best, which this report highlights while introducing a new optimizer. ARPO demonstrates a new approach of shifting from random search across fixed optimization strategies into a latent space of adaptive exploration on what techniques work best and in what ways and eventually applying this cumulative knowledge gained into a final optimized prompt.

While this open-endedness allows for fine-grained edits, the reward signal can be noisy and lead to high upper-bound gains or potentially unwanted degradations as the addition of certain few-shots or rules can guide the downstream LLM in different ways. Hence, when applying RL techniques like strategy search and credit assignment, it is important to keep the reward signal representation-aware, ensuring that the meta-learning is truly adaptive through ways like grouping near-duplicate rules or tracking clusters of related few-shots that are helpful.

Another lesson was that stacking strategies do not always guarantee gains. Giving ARPO a larger teacher and at times layering RL techniques together didn't always lead to improved performance. Hence, it's important to recognize that added complexity to prompt optimization does not necessitate improved performance, and it's important to measure these factors through evaluation suites as diverse as LangProBe to understand this signal clearly.

Future work with ARPO would consider an enhanced search space rather than more complex policies. This would include adding actions like deleting rules and few-shots that aren't helpful, reordering prompt components to account for positional bias, replacing these parts as a selective strategy and various other possible prompt-space edits. Additionally, incorporating a fixed optimization budget could ensure for more accurate comparisons to existing optimizers as budget-constrained evaluation sweeps would reveal where each optimizer performs best at a fairer Pareto frontier measurement.

7 Conclusion

ARPO offers a novel adaptive approach to prompt optimization, demonstrating competitive performance across a wide range of LLM tasks and prompting pipelines. While no single optimizer consistently outperforms others in all scenarios, ARPO's RL mechanisms enable nuanced prompt refinements that are particularly effective and are especially computationally efficient as ARPO is highly positioned on the Pareto frontier. These results highlight the benefits of guided cumulative learning optimization over random search heuristics, while also demonstrating the need to explore and align optimization strategies with specific task requirements and model capabilities, rather than relying on a one-size-fits-all solution. Overall, ARPO offers a high-performing and cost-efficient approach to automated prompt optimization using reward-driven RL methods to improve LLM prompting pipelines.

8 Team Contributions

- **Arnav Singhvi:** Arnav worked on coding up each of the three ARPO optimizer variants and integrating with the DSPy library, testing for compatibility and ensuring aligned implementation to get ready for benchmarking. In changes from the proposal, Arnav had to also take over the evaluation responsibilities for all the existing optimizers and ARPO optimizers on LangProBe, launching the cluster and required SGLang servers to host all the open-source models for experimentation for all the 5 model teacher-student configurations across the 7 total experiment configurations for the 15 datasets, and totalling to about 1200 experiments ran total. Arnav also took over the visualizations component for demonstrative analysis of the experiments, developing the performance-wise, cost-wise and Pareto frontier graphs. Regarding the project milestone, Arnav worked on the "Changes to Research Hypothesis" and "Next Steps" sections, while outlining necessary content for the Experiments and Potential Challenges section but still had to do heavy proofreading for the final version. For the project poster, Arnav worked on the Results, Conclusion and Experiments section, while outlining the necessary content required to fill out the rest of the poster for the Problem Statement, Background, Methodology, Pipeline, and Algorithm components and had to do heavy proofreading for the final version. Arnav also printed out the poster and presented individually. For the final report, Arnav worked on the Introduction, Methodology, Experimental Setup, and Appendix, provided graphs for the necessary Results section and outlined the necessary content for the other main sections including the Related Work, Results, and Discussion sections. For all of these sections, Arnav had to do heavy rewriting and proofreading due to lack of communication. Overall, from Arnav's vantage point, he had to do a significant portion of work that was not outlined in the proposal to be his contributions which he still did so that the project could be submitted. [written by Arnav Singhvi]

- **Shreyas Agarwal:** Shreyas worked on configuring the LangProbe repository and creating the initial data runs for integrating all existing and new ARPO optimizers across 13 benchmark tasks for 3 model configurations. For over a week, he ran first experiments across multiple teacher-student configurations using 3B, 8B, and 70B models and for all benchmark tasks, debugging issues with inconsistent results and broken runs. After Shreyas spent considerable time debugging and coordinating with his teammate to get the full set of optimizers and models running correctly, the teammate took over the remaining evaluation runs, citing challenges in progressing due to "back and forth" communication. Shreyas agreed, deferring to his teammate's direct access to the compute cluster and ability to run unsupervised, extended experiments.

  Following this change, Shreyas took primary responsibility for background research and writing. For the project milestone, he wrote Experiments, Initial Results, Changes to Research Hypothesis or Objective, and Next Steps sections (except Potential Challenges) using result data and plots provided by his teammate and incorporated his teammate's feedback afterward.

  For the poster, Shreyas wrote the Problem Statement, Background, Methodology, Pipeline, and Algorithm sections. Shreyas was unable to make it in person to the poster presentation due to scheduling conflicts, and therefore he handled more of the written components.

  For the final report, Shreyas wrote the Extended Abstract, Abstract, Related Work, Results, Discussion, and Conclusion sections, as well as contributed references for the Introduction.

  Overall, from Shreyas' viewpoint, after having let go of the evaluation responsibility in the proposal, he took on significant writing tasks. Despite lack of communication regarding need for heavy proofreading, Shreyas completed all tasks he was responsible for with the new contribution split.

  [written by Shreyas Agarwal]

Changes from Proposal From the milestone, the project shifted from proposing a standalone RL-based optimizer to enhancing the existing SIMBA framework with RL-driven policies, producing the ARPO variants. This revised focus enables more targeted exploration and credit assignment within SIMBA's modular prompt optimization loop. By leveraging SIMBA's existing structure, we integrated RL techniques to show improved performance and efficiency over existing random search techniques, offering a new optimizer approach for the prompt optimization techniques in the DSPy framework.

Team contributions also shifted from the original proposal due to coordination and scheduling challenges. Arnav took on the remaining evaluation runs and visualizations and contributed to and reviewed the writing portions. Shreyas took more writing tasks. [written by Shreyas Agarwal]

References

Meta AI. 2024a. Introducing Llama 3.1: Our Most Capable Models to Date. `https://ai.meta.com/blog/meta-llama-3-1/`.

Meta AI. 2024b. Llama 3.2: Revolutionizing Edge AI and Vision with Open Models. `https://ai.meta.com/blog/llama-3-2-connect-2024-vision-edge-mobile-devices/`.

Meta AI. 2024c. Llama 3.3: Advancing Multimodal AI. `https://ollama.com/library/llama3.3`.

Bowen Cao, Deng Cai, Zhisong Zhang, Yuexian Zou, and Wai Lam. 2024. On the Worst Prompt Performance of Large Language Models. arXiv:2406.10248 [cs.CL] `https://arxiv.org/abs/2406.10248`

Junying Chen, Zhenyang Cai, Ke Ji, Xidong Wang, Wanlong Liu, Rongsheng Wang, Jianye Hou, and Benyou Wang. 2024. HuatuoGPT-o1, Towards Medical Complex Reasoning with LLMs. arXiv:2412.18925 [cs.CL] `https://arxiv.org/abs/2412.18925`

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, et al. 2021. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168* (2021).

Can Cui, Yunsheng Ma, Zichong Yang, Yupeng Zhou, Peiran Liu, Juanwu Lu, Lingxi Li, Yaobin Chen, Jitesh H. Panchal, Amr Abdelraouf, Rohit Gupta, Kyungtae Han, and Ziran Wang. 2025. Large Language Models for Autonomous Driving (LLM4AD): Concept, Benchmark, Experiments, and Challenges. arXiv:2410.15281 [cs.RO] `https://arxiv.org/abs/2410.15281`

Robert C. Detrano, András Jánosi, Walter Steinbrunn, Matthias Emil Pfisterer, Johann-Jakob Schmid, Sarbjit Sandhu, Kern Guppy, Stella Lee, and Victor Froelicher. 1989. International application of a new probability algorithm for the diagnosis of coronary artery disease. *The American Journal of Cardiology* 64, 5 (1989), 304–310.

Karel D'Oosterlinck, Omar Khattab, François Remy, Thomas Demeester, Chris Develder, and Christopher Potts. 2024. In-Context Learning for Extreme Multi-Label Classification. arXiv:2401.12178 [cs.CL] `https://arxiv.org/abs/2401.12178`

Rory A. Fisher. 1936. The use of multiple measurements in taxonomic problems. *Annals of Human Genetics* 7 (1936), 179–188.

Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. 2021. Measuring Massive Multitask Language Understanding. arXiv:2009.03300 [cs.CY] `https://arxiv.org/abs/2009.03300`

Joey Hong, Jessica Lin, Anca Dragan, and Sergey Levine. 2024. Interactive Dialogue Agents via Reinforcement Learning on Hindsight Regenerations. arXiv:2411.05194 [cs.LG] `https://arxiv.org/abs/2411.05194`

Yichen Jiang, Shikha Bordia, Zheng Zhong, Charles Dognin, Maneesh Singh, and Mohit Bansal. 2020. HoVer: A dataset for many-hop fact extraction and claim verification. In *Findings of the Association for Computational Linguistics: EMNLP 2020* (Online). Association for Computational Linguistics, 3441–3460.

Di Jin, Eileen Pan, Nassim Oufattole, Wei-Hung Weng, Hanyi Fang, and Peter Szolovits. 2021. What disease does this patient have? a large-scale open domain question answering dataset from medical exams. *Applied Sciences* 11, 14 (2021), 6421.

Qiao Jin, Bhuwan Dhingra, Zhengping Liu, William W Cohen, and Xinghua Lu. 2019. Pubmedqa: A dataset for biomedical research question answering. *arXiv preprint arXiv:1909.06146* (2019).

Omar Khattab, Christopher Potts, and Matei Zaharia. 2021. Baleen: Robust Multi-Hop Reasoning at Scale via Condensed Retrieval. *CoRR* abs/2101.00436 (2021). arXiv:2101.00436 `https://arxiv.org/abs/2101.00436`

Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan, Saiful Haq, Ashutosh Sharma, Thomas T. Joshi, Hanna Moazam, Heather Miller, Matei Zaharia, and Christopher Potts. 2023. DSPy: Compiling Declarative Language Model Calls into Self-Improving Pipelines. arXiv:2310.03714 [cs.CL] `https://arxiv.org/abs/2310.03714`

Weize Kong, Spurthi Amba Hombaiah, Mingyang Zhang, Qiaozhu Mei, and Michael Bendersky. 2024a. Prewrite: Prompt rewriting with reinforcement learning. *arXiv preprint arXiv:2401.08189* (2024).

Yilun Kong, Hangyu Mao, Qi Zhao, Bin Zhang, Jingqing Ruan, Li Shen, Yongzhe Chang, Xueqian Wang, Rui Zhao, and Dacheng Tao. 2024b. QPO: Query-dependent Prompt Optimization via Multi-Loop Offline Reinforcement Learning. *arXiv preprint arXiv:2408.10504* (2024).

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in neural information processing systems* 33 (2020), 9459–9474.

LiteLLM. 2024. LiteLLM: Simplifying Access to Multiple LLMs. `https://docs.litellm.ai/`.

Princeton NLP. 2023. SWE-bench Verified Dataset. `https://huggingface.co/datasets/princeton-nlp/SWE-bench_Verified`

OpenAI. 2021. OpenAI HumanEval Dataset. `https://huggingface.co/datasets/openai/openai_humaneval`

Krista Opsahl-Ong, Michael J Ryan, Josh Purtell, David Broman, Christopher Potts, Matei Zaharia, and Omar Khattab. 2024. Optimizing Instructions and Demonstrations for Multi-Stage Language Model Programs. arXiv:2406.11695 [cs.CL] `https://arxiv.org/abs/2406.11695`

Ankit Pal, Logesh Kumar Umapathi, and Malaikannan Sankarasubbu. 2022. Medmcqa: A large-scale multi-subject multi-choice dataset for medical domain question answering. In *Conference on health, inference, and learning*. PMLR, 248–260.

Pranab Sahoo, Ayush Kumar Singh, Sriparna Saha, Vinija Jain, Samrat Mondal, and Aman Chadha. 2025. A Systematic Survey of Prompt Engineering in Large Language Models: Techniques and Applications. arXiv:2402.07927 [cs.AI] `https://arxiv.org/abs/2402.07927`

Jingyuan S. She, Christopher Potts, Samuel R. Bowman, and Atticus Geiger. 2023. ScoNe: Benchmarking negation reasoning in language models with finetuning and in-context learning. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)* (Toronto, Canada). Association for Computational Linguistics, 1803–1821.

Stanford-NLP-Group. [n. d.]. dspy.Predict. `https://dspy.ai/api/modules/Predict/`.

Stanford-NLP-Group. 2024. DSPy.LM: Language Model Integration Module. `https://dspy.ai/learn/programming/language_models/`.

Shangyin Tan, Lakshya A. Agrawal, Arnav Singhvi, Liheng Lai, Michael J. Ryan, Dan Klein, Omar Khattab, Koushik Sen, and Matei Zaharia. 2025. LangProBe: A Language Programs Benchmark. *arXiv preprint arXiv:2502.20315* (2025).

Sijun Tan, Siyuan Zhuang, Kyle Montgomery, William Y. Tang, Alejandro Cuadron, Chenguang Wang, Raluca Ada Popa, and Ion Stoica. 2024. JudgeBench: A Benchmark for Evaluating LLM-based Judges. arXiv:2410.12784 [cs.AI] `https://arxiv.org/abs/2410.12784`

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems* 35 (2022), 24824–24837.

Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V. Le, Denny Zhou, and Xinyun Chen. 2023. Large Language Models as Optimizers. *arXiv preprint* arXiv:2309.03409 (2023). `https://arxiv.org/abs/2309.03409`

Zhilin Yang, Peng Qi, Saizheng Zhang, Yoshua Bengio, William Cohen, Ruslan Salakhutdinov, and Christopher D. Manning. 2018. HotpotQA: A dataset for diverse, explainable multi-hop question answering. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing* (Brussels, Belgium). Association for Computational Linguistics, 2369–2380.

Tuo Zhang, Jinyue Yuan, and Salman Avestimehr. 2024. Revisiting opro: The limitations of small-scale llms as optimizers. *arXiv preprint arXiv:2405.10276* (2024).

Guang Zhao. 2025. *Pareto Prompt Optimization*. Technical Report. Brookhaven National Laboratory (BNL), Upton, NY (United States).

Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. 2024. SGLang: Efficient Execution of Structured Language Model Programs. arXiv:2312.07104 [cs.AI] `https://arxiv.org/abs/2312.07104`

A Additional Experiments

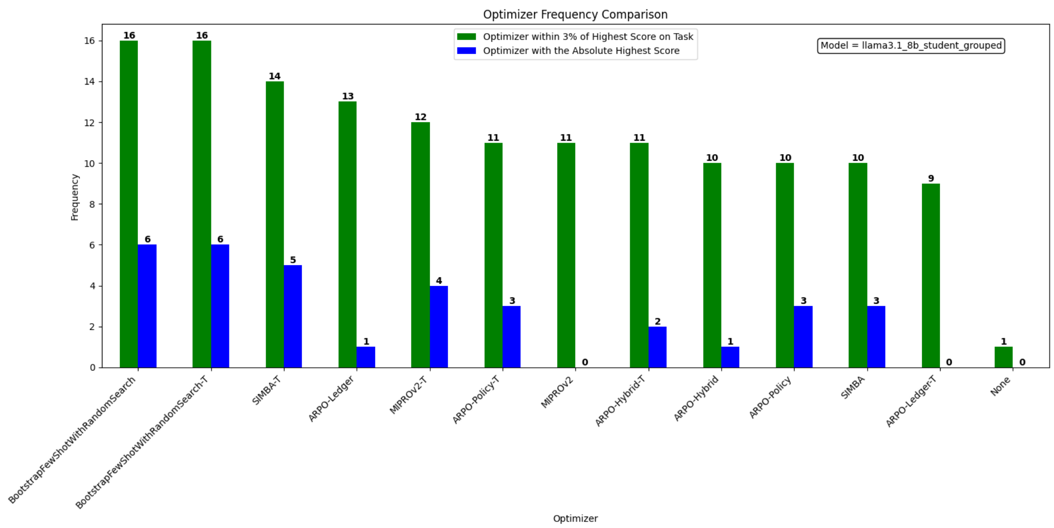

Figure 4: Comparing Optimizers on Llama-3.1-8B.

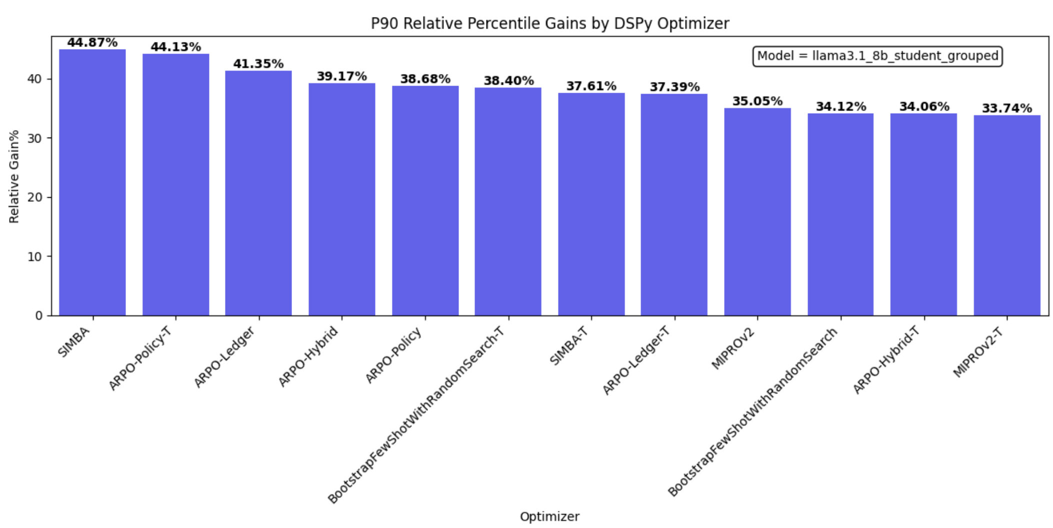

Figure 5: P90 Relative Gains for Optimizers on Llama-3.1-8B.

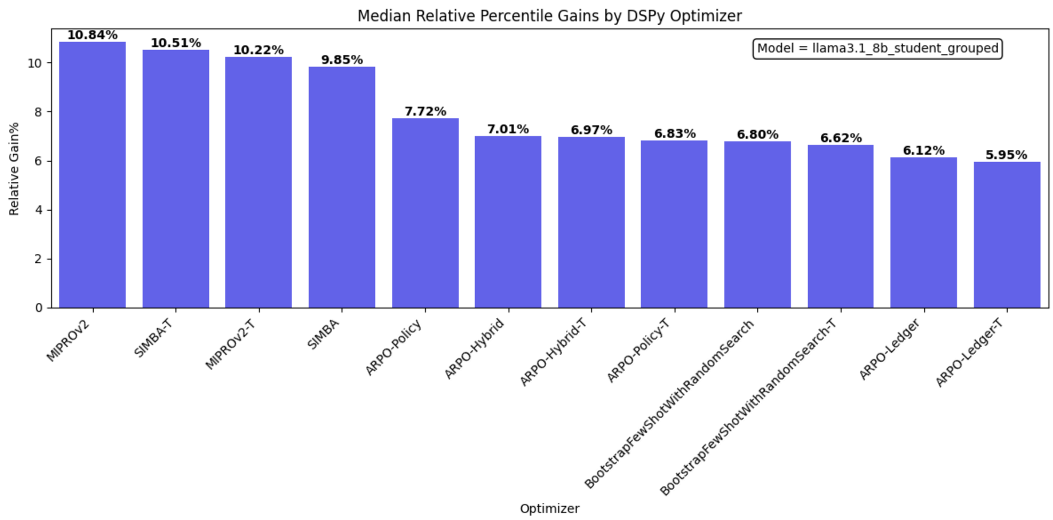

Figure 6: Median Relative Gains for Optimizers on Llama-3.1-8B.

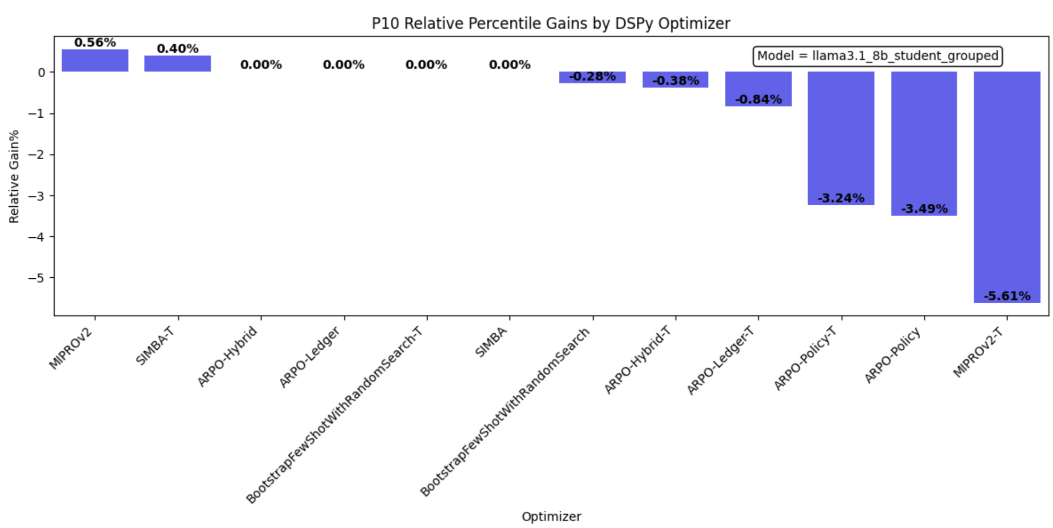

Figure 7: P10 Relative Gains for Optimizers on Llama-3.1-8B.

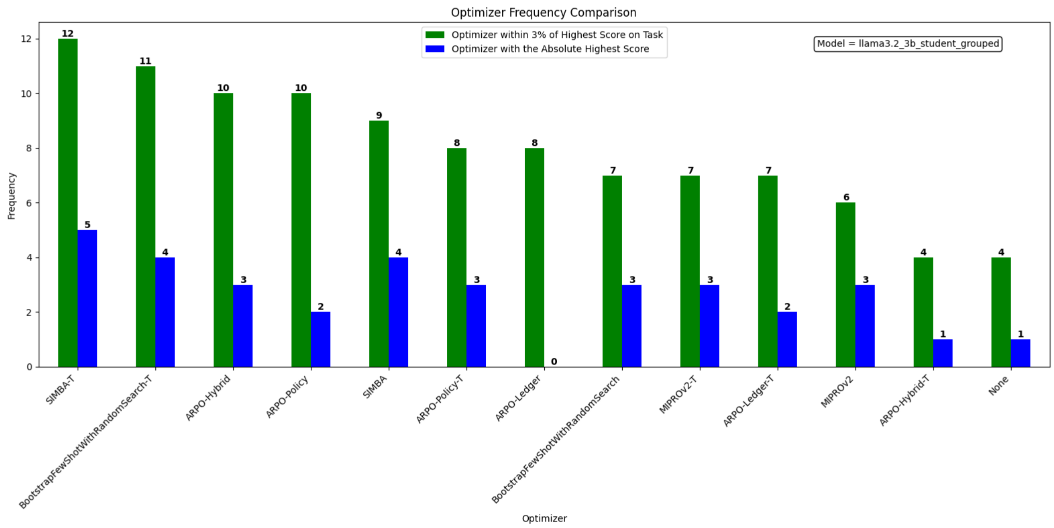

Figure 8: Comparing Optimizers on Llama-3.2-3B.

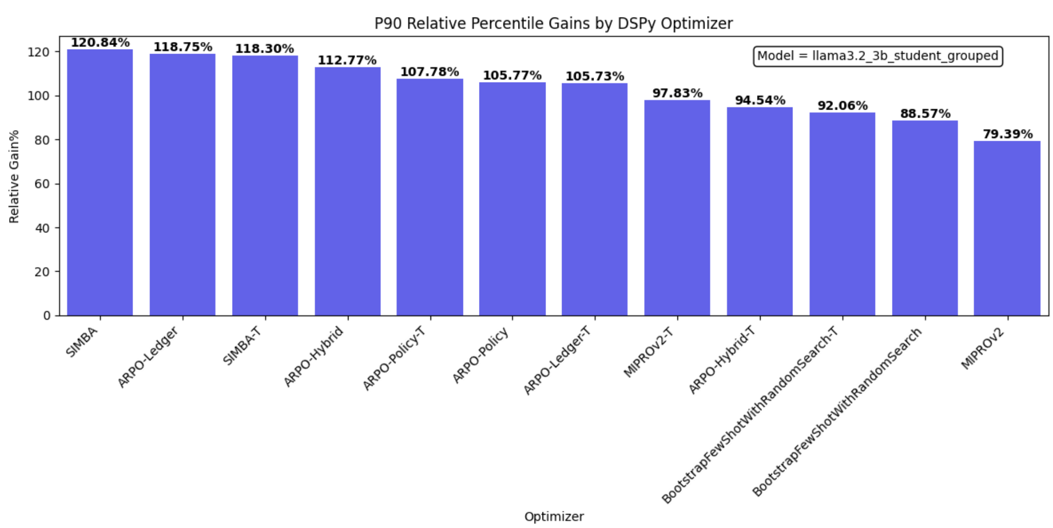

Figure 9: P90 Relative Gains for Optimizers on Llama-3.2-3B.

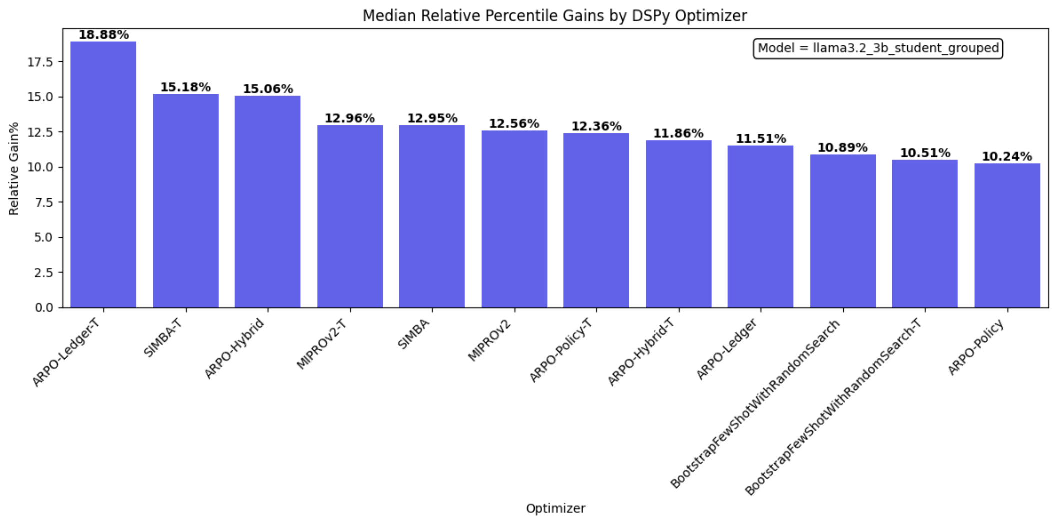

Figure 10: Median Relative Gains for Optimizers on Llama-3.2-3B.

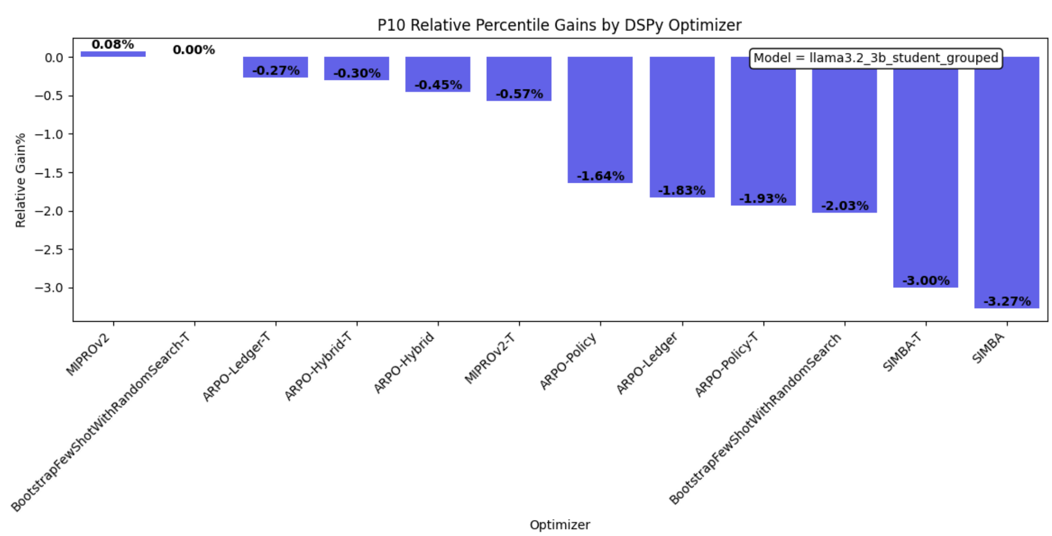

Figure 11: P10 Relative Gains for Optimizers on Llama-3.2-3B.

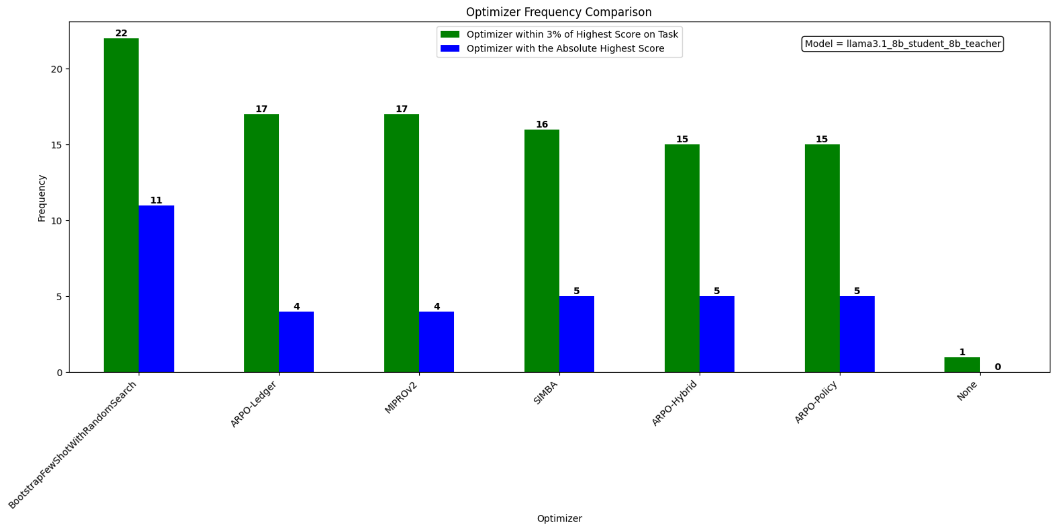

Figure 12: Comparing Optimizers on Llama-3.1-8B (Student Only).

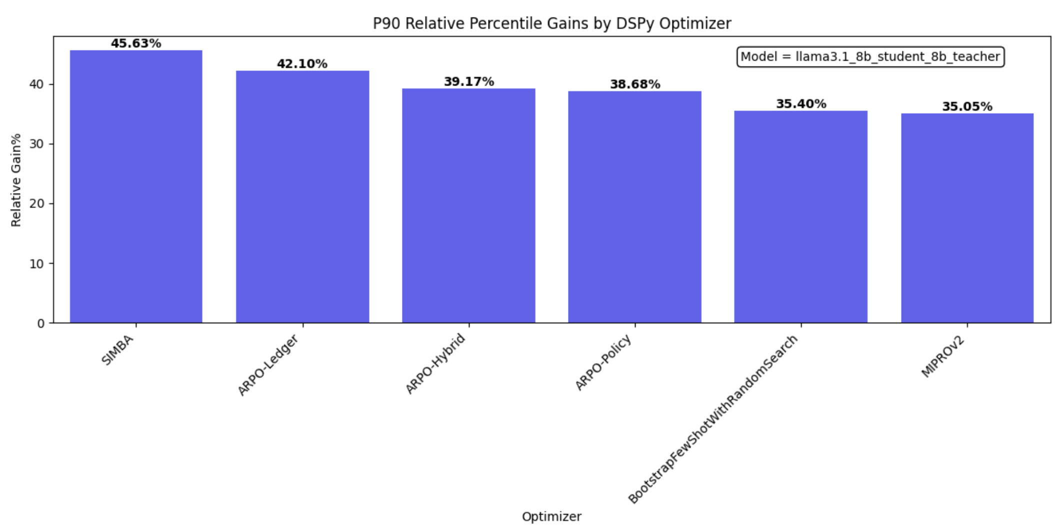

Figure 13: P90 Relative Gains for Optimizers on Llama-3.1-8B (Student Only).

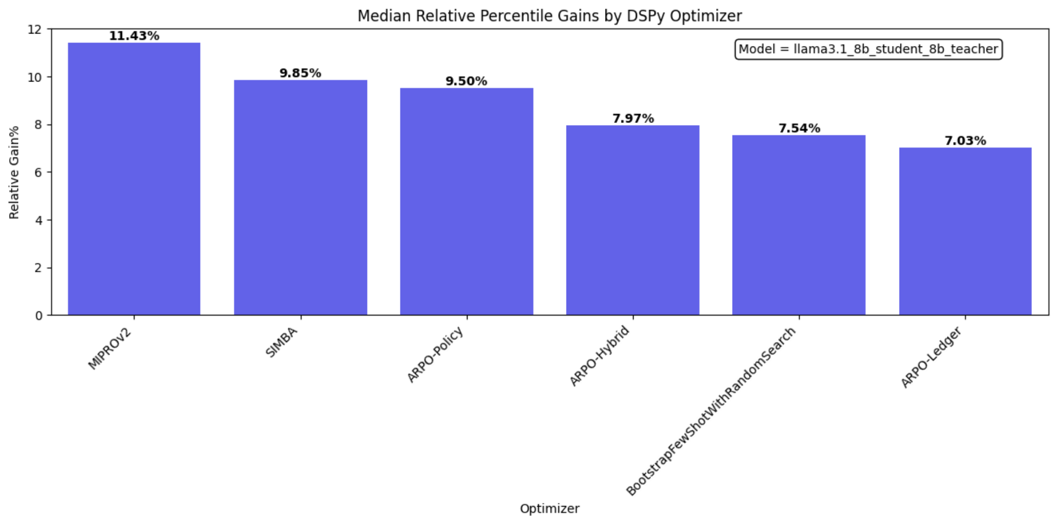

Figure 14: Median Relative Gains for Optimizers on Llama-3.1-8B (Student Only).

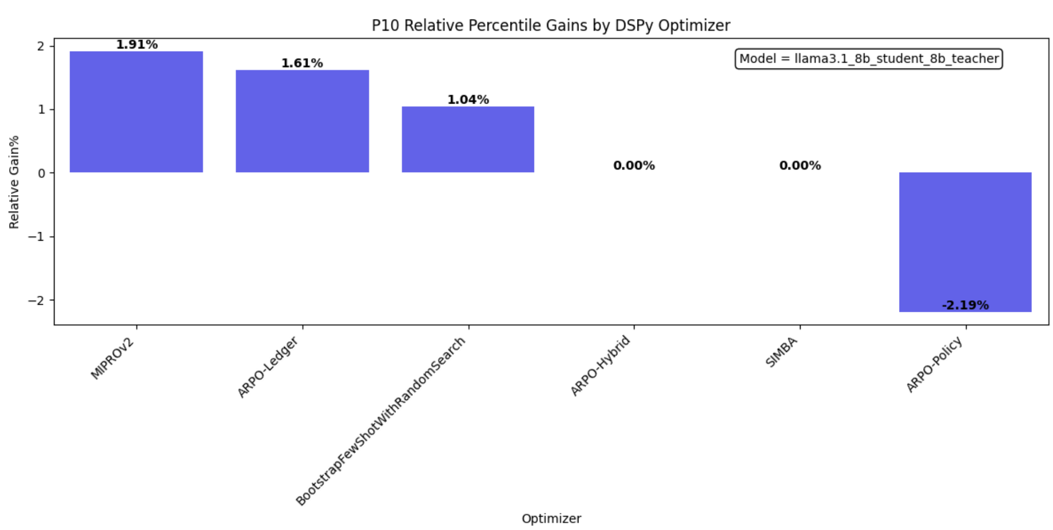

Figure 15: P10 Relative Gains for Optimizers on Llama-3.1-8B (Student Only).

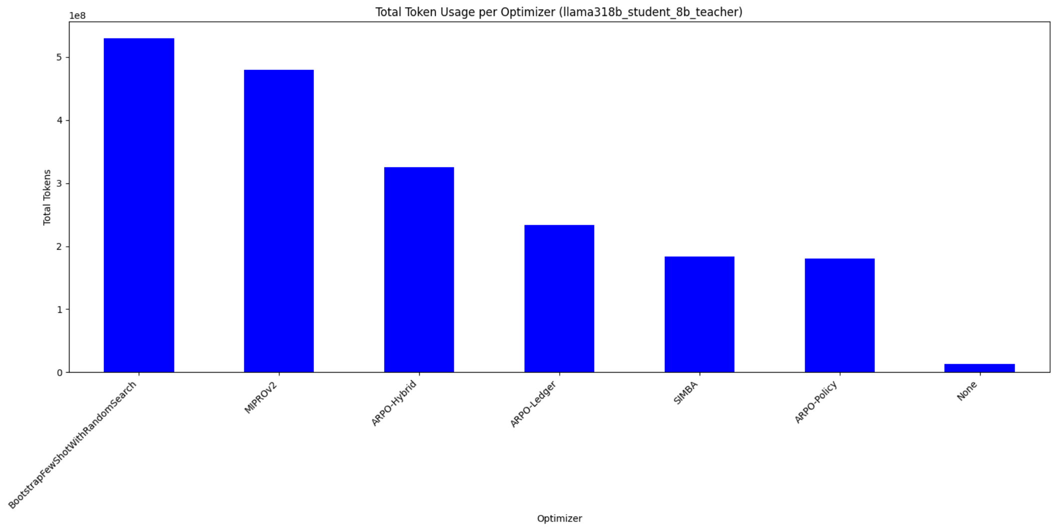

Figure 16: Total Token Usage For Optimizers on Llama-3.1-8B (Student Only).

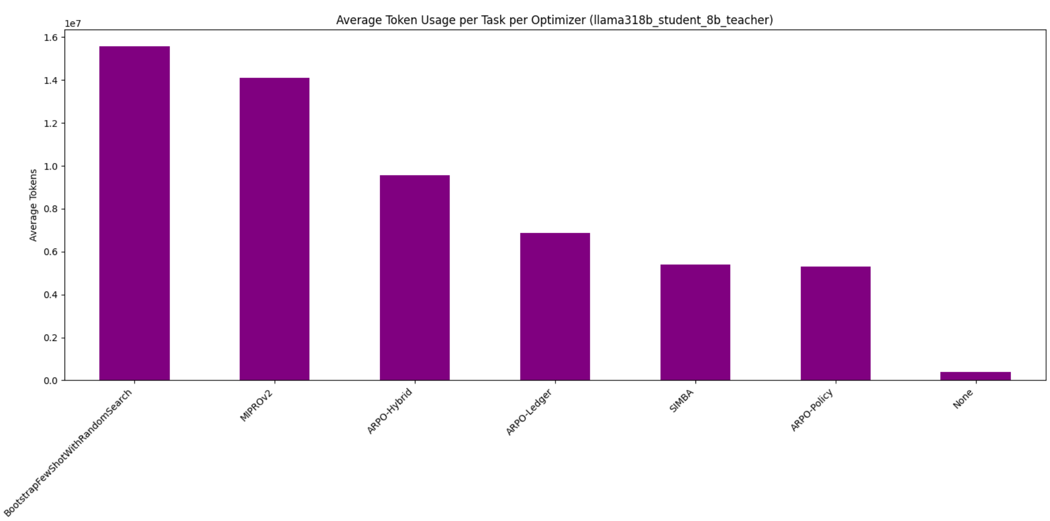

Figure 17: Average Token Usage For Optimizers on Llama-3.1-8B (Student Only).

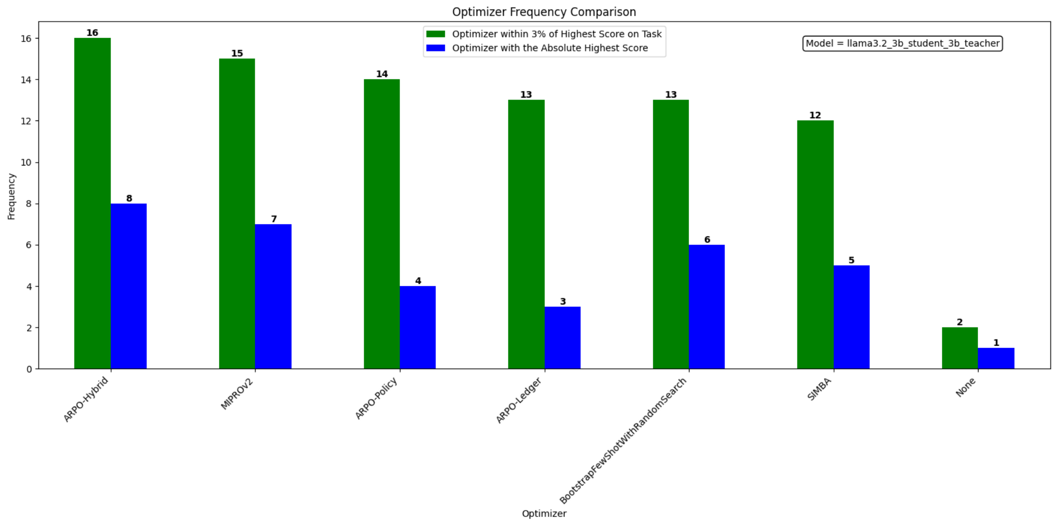

Figure 18: Comparing Optimizers on Llama-3.2-3B (Student Only).

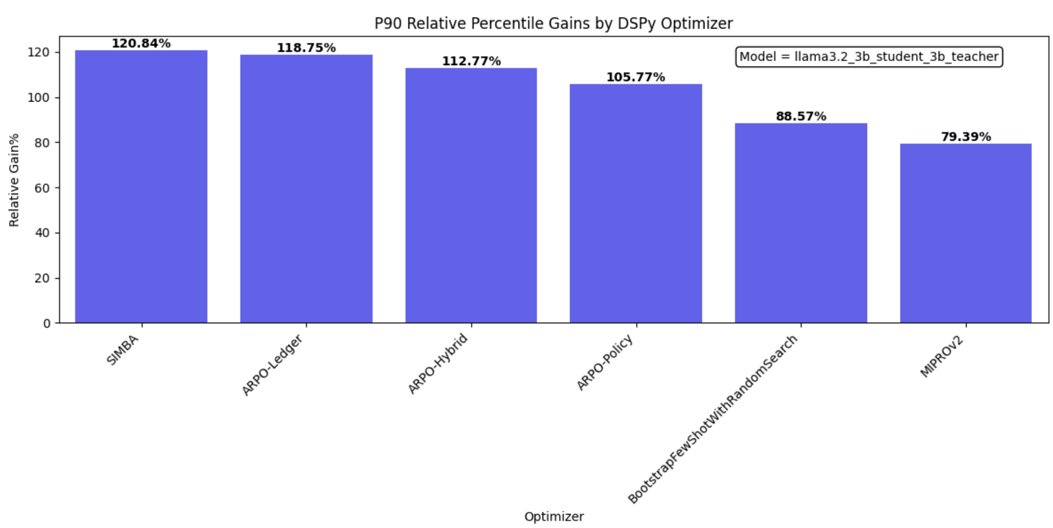

Figure 19: P90 Relative Gains for Optimizers on Llama-3.2-3B (Student Only).

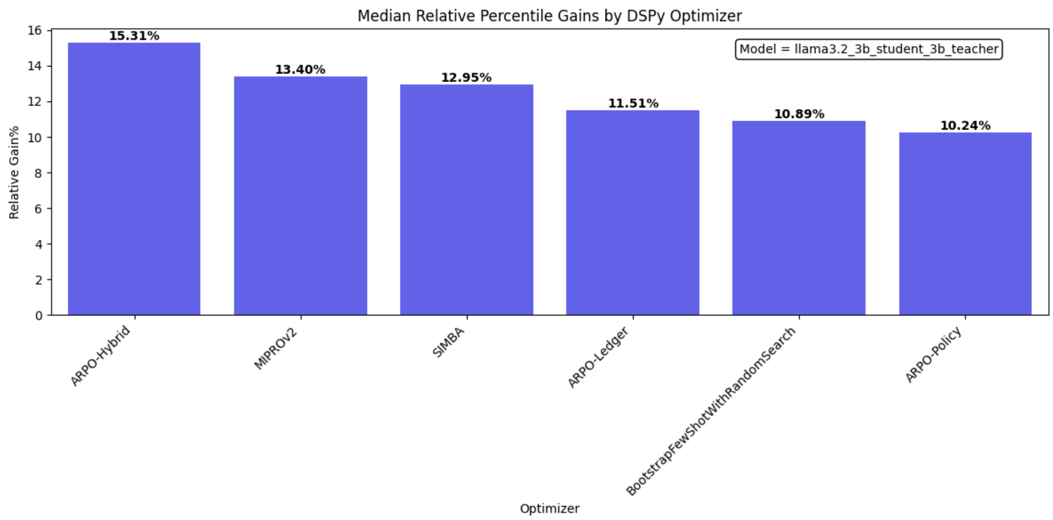

Figure 20: Median Relative Gains for Optimizers on Llama-3.2-3B (Student Only).

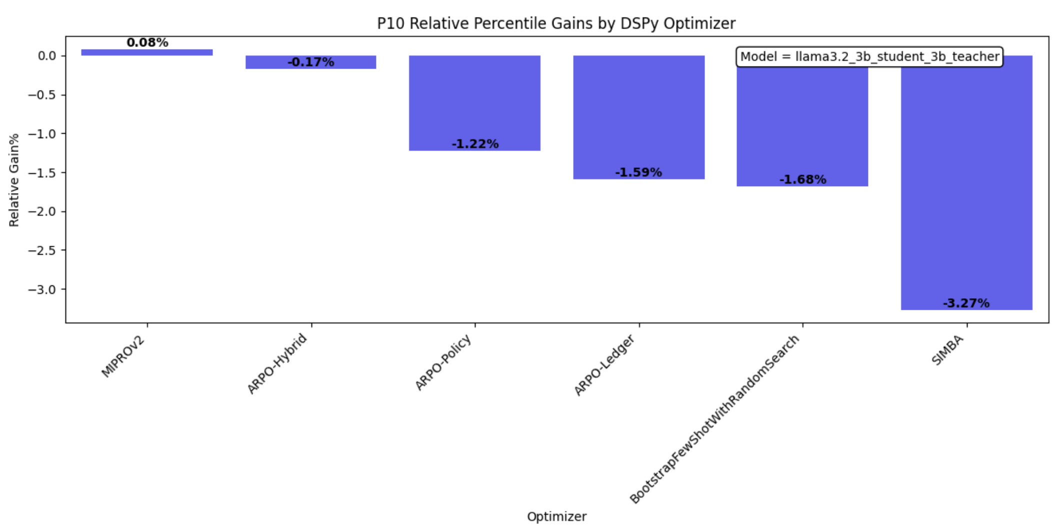

Figure 21: P10 Relative Gains for Optimizers on Llama-3.2-3B (Student Only).

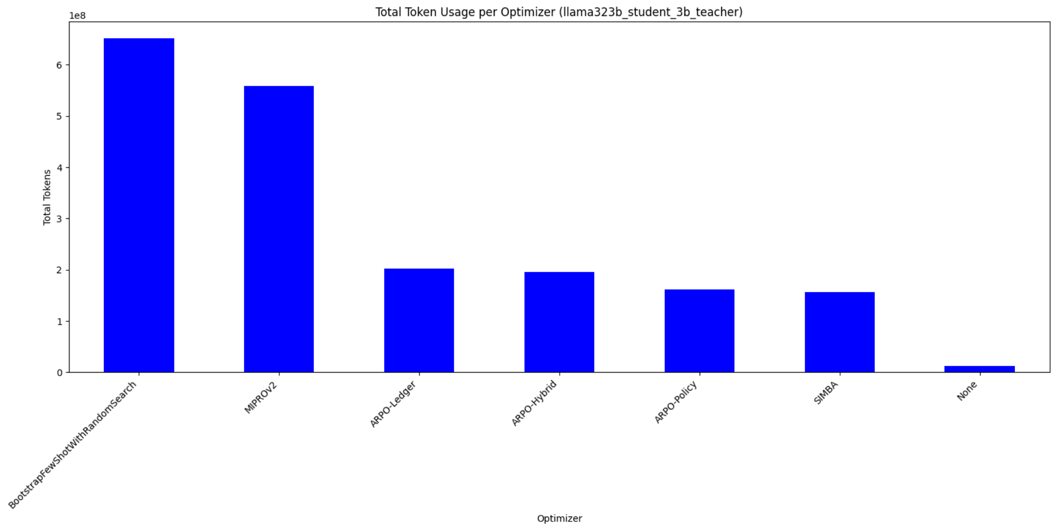

Figure 22: Total Token Usage For Optimizers on Llama-3.2-3B (Student Only).

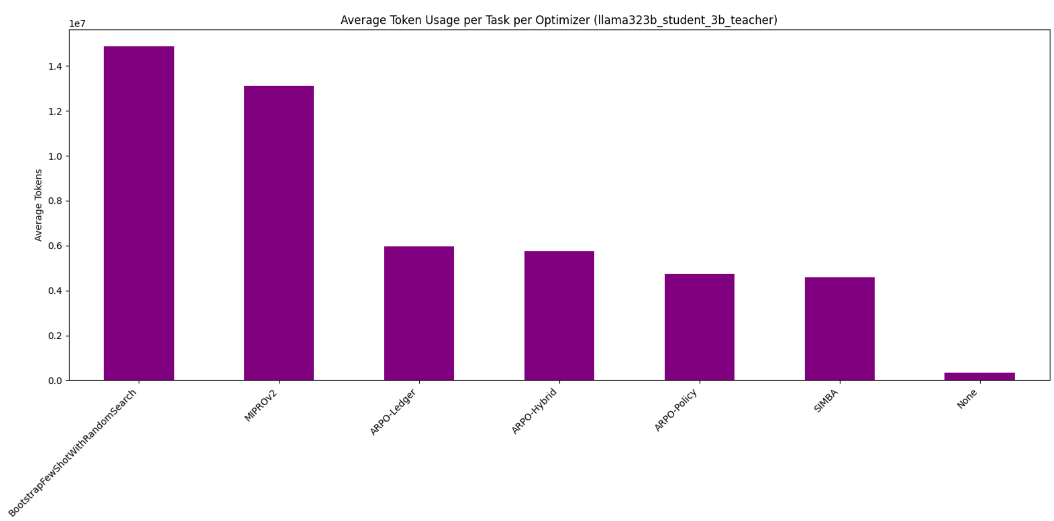

Figure 23: Average Token Usage For Optimizers on Llama-3.2-3B (Student Only).

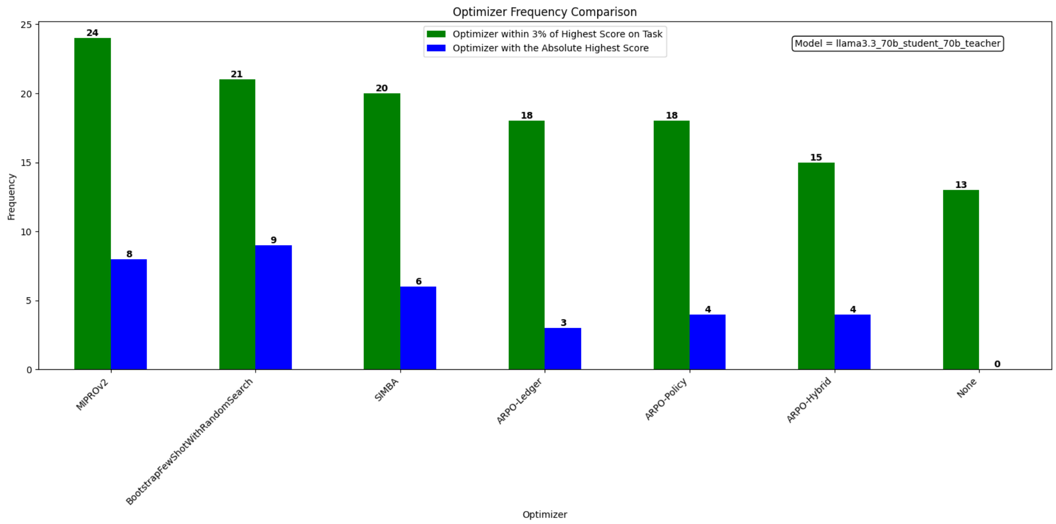

Figure 24: Comparing Optimizers on Llama-3.3-70B (Student Only).

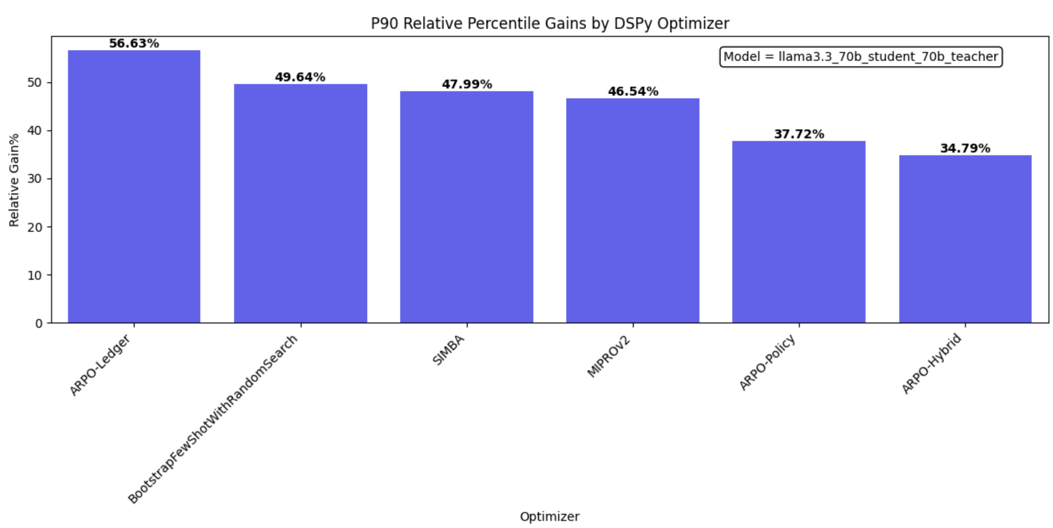

Figure 25: P90 Relative Gains for Optimizers on Llama-3.3-70B (Student Only).

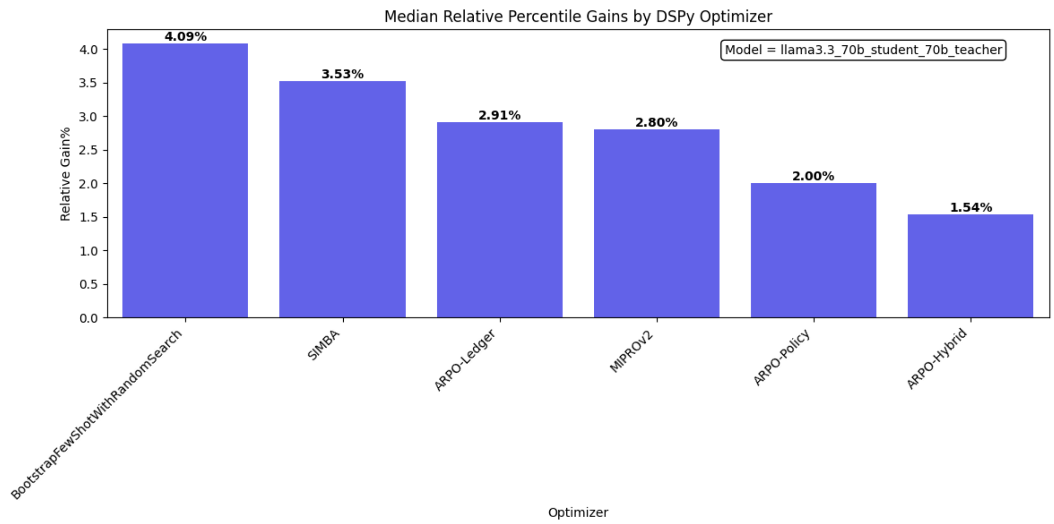

Figure 26: Median Relative Gains for Optimizers on Llama-3.3-70B (Student Only).

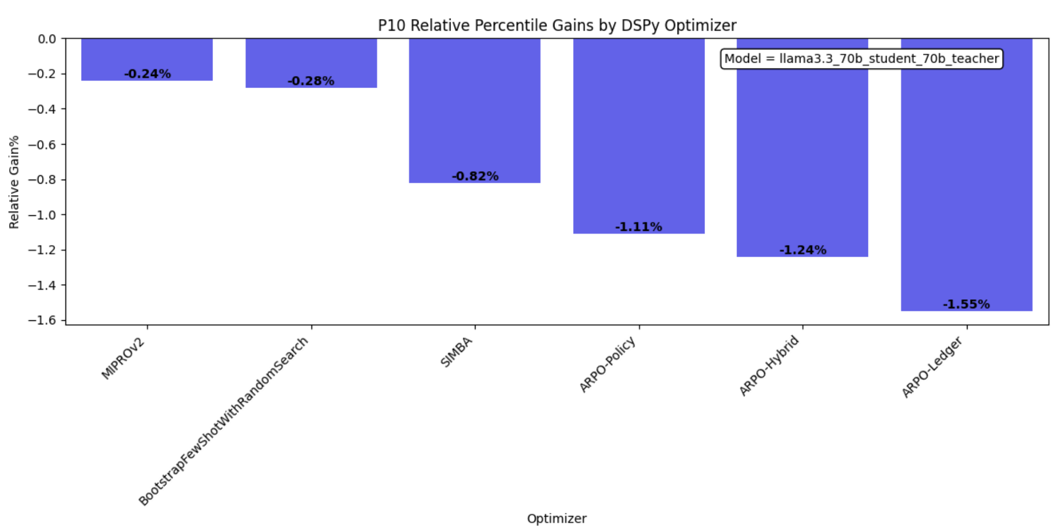

Figure 27: P10 Relative Gains for Optimizers on Llama-3.3-70B (Student Only).

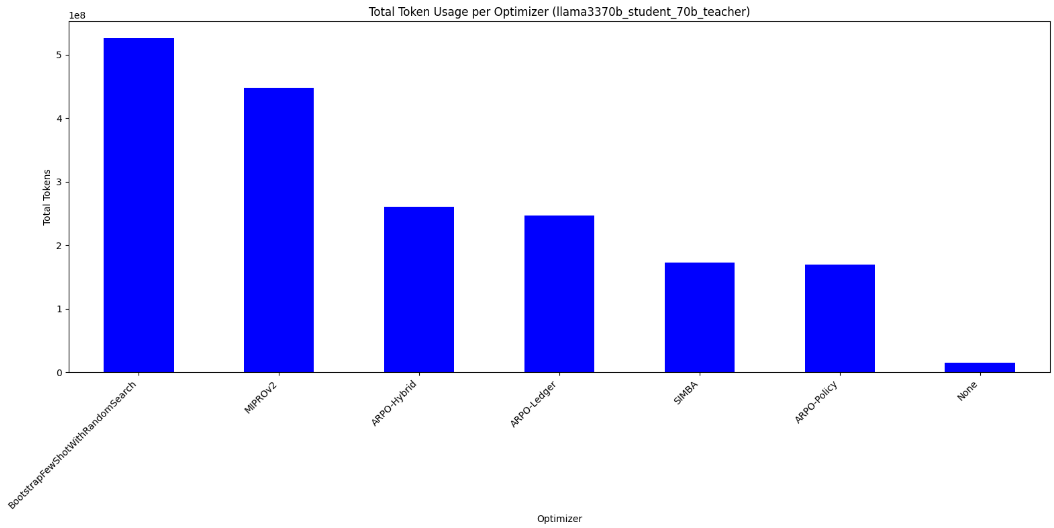

Figure 28: Total Token Usage For Optimizers on Llama-3.3-70B (Student Only).

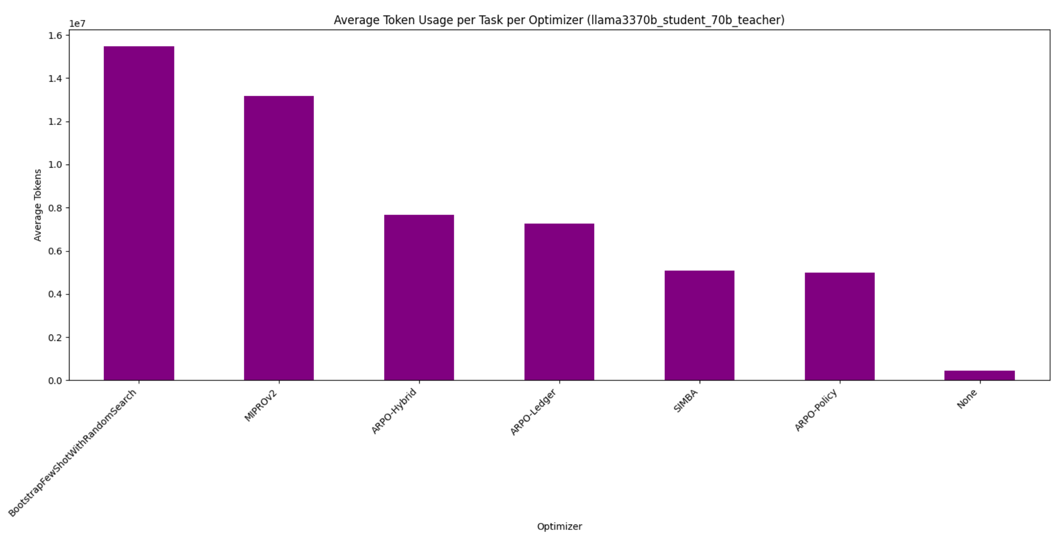

Figure 29: Average Token Usage For Optimizers on Llama-3.3-70B (Student Only).

B Implementation Details

Algorithm 1: ARPO

Input: Training set $\mathcal{E}$, student program S_0, metric function $\mathcal{M}(y, \hat{y})$
Output: Optimized program S^*
Initialize pool of candidate programs $\mathcal{P} \leftarrow \{S_0\}$
for $t = 1$ **to** *max_steps* **do**
  Sample mini-batch $B \subset \mathcal{E}$
  foreach *model* m *and example* $e \in B$ **do**
    $S \sim \begin{cases} \text{credit\_weighted\_softmax}(\mathcal{P}) & \text{if Hybrid} \\ \text{softmax}(\mathcal{P}) & \text{all others} \end{cases}$
    Run S with m on e and collect output
  Group outputs by example and sort buckets by *(i) max–min score gap, (ii) max score, (iii) max–avg score gap*
  foreach *bucket in top_buckets* **do**
    Select source program S_{src} via softmax
    if *Credit Ledger or Hybrid* **then**
      Prune worst demos/rules, replace with high-credit alternatives
    else
      Drop a Poisson-sampled subset of existing demos
    if *RL Policy or Hybrid* **then**
      strategy $\sim$ policy_softmax(V_{strat})
    else
      strategy $\sim$ uniform(strategies)
    Apply strategy to create candidate S_{cand}
  foreach *candidate* S_{cand} *with strategy* s **do**
    $r \leftarrow (\text{score}(S_{\text{cand}}) - \text{baseline}) - \text{decay\_penalty} + \text{edit\_bonus}$
    if *RL Policy or Hybrid* **then**
      $V_{\text{strat}}[s] \leftarrow V_{\text{strat}}[s] + \alpha\big(r - V_{\text{strat}}[s]\big)$
    if *Credit Ledger or Hybrid* **then**
      Update demo and rule credits: $c \leftarrow \gamma c + \eta r$
  Add all S_{cand} to $\mathcal{P}$
$S^* \leftarrow \arg\max_{S \in \mathcal{P}} \text{score}(S, \mathcal{E})$
return S^*

Figure 30: ARPO Algorithm Overview

Optimizer	Parameters	Description

BootstrapFewShotWithRandomSearch	`max_bootstrapped_demos`
	`max_labeled_demos`
	`teacher_lm`
		Teacher-LLM bootstrapping of input/output examples from trainset – 4.
		Labeled examples from train data as demos – 16.
		LLM that generates bootstrapped demonstrations.

MIPROv2	`num_candidates`
	`num_trials`
	`minibatch_size`
	`minibatch_full_eval_steps`
	`max_bootstrapped_demos`
	`max_labeled_demos`
	`teacher_lm`
		Instruction-set hypotheses explored per optimization round – 10.
		Total Bayesian-optimization iterations across rounds – 50.
		Examples processed in each inner-loop mini-batch – 35.
		After this many mini-batches, trigger a full-set evaluation – 5.
		Teacher-LLM bootstrapped demos per predictor – 4.
		Labeled demos per predictor – 2.
		LLM that generates both instruction proposals and bootstrapped demos.

SIMBA	`bsize`
	`num_candidates`
	`max_steps`
	`max_demos`
	`temperature_for_sampling`
	`temperature_for_candidates`
		Mini-batch size per optimization step – 32.
		New candidate programs generated each iteration – 6.
		Total optimization iterations – 8.
		Maximum few-shot demos kept per predictor – 4.
		Softmax temperature for program sampling during trajectory generation – 0.2.
		Softmax temperature when choosing a source program to mutate – 0.2.

ARPO-Policy	`bsize`
	`num_candidates`
	`max_steps`
	`max_demos`
	`temperature_for_sampling`
	`temperature_for_candidates`
	`strategy_lr`
	`strategy_temperature`
		Mini-batch size – 32.
		Candidate programs per step – 6.
		Optimization steps – 8.
		Demo cap per predictor – 4.
		Softmax temperature for program sampling – 0.2.
		Softmax temperature for source-program selection – 0.2.
		Learning-rate of the RL policy that selects edit strategies – 0.1.
		Softmax temperature for strategy selection – 1.0.

ARPO-Ledger	`bsize`
	`num_candidates`
	`max_steps`
	`max_demos`
	`temperature_for_sampling`
	`temperature_for_candidates`
	`ledger_lr`
	`ledger_decay`
		Mini-batch size – 32.
		Candidate programs per step – 6.
		Optimization steps – 8.
		Demo cap per predictor – 4.
		Temperature for program sampling – 0.2.
		Temperature for source-program selection – 0.2.
		Learning-rate for updating demo/rule credit scores – 0.05.
		Exponential decay factor applied to accumulated credit – 0.9.

ARPO-Hybrid	`bsize`
	`num_candidates`
	`max_steps`
	`max_demos`
	`temperature_for_sampling`
	`temperature_for_candidates`
	`ledger_lr`
	`ledger_decay`
	`strat_lr`
	`strat_temp`
	`credit_weight`
	`ablate_examples`
		Mini-batch size – 32.
		Candidate programs per step – 6.
		Optimization steps – 8.
		Demo cap per predictor – 4.
		Temperature for program sampling – 0.2.
		Temperature for source-program selection – 0.2.
		Learning-rate for credit-ledger updates – 0.05.
		Decay factor for ledger credit averaging – 0.9.
		Learning-rate for strategy-policy updates – 0.1.
		Strategy-policy softmax temperature – 1.0.
		Weight given to credit-ledger signal when ranking programs – 0.1.
		Examples used for fine-grained ablation after each batch – 16.

Table 1: Parameters for existing DSPy optimizers and the three ARPO variants.

C Sample Prompts

Classification Task – Iris

Baseline:
Given the petal and sepal dimensions in cm, predict the iris species.

SIMBA (no rule additions - only fewshot)

```
"petal_length": "1.5",
"petal_width": "0.2",
"sepal_length": "5.4",
"sepal_width": "3.7",
"answer": "versicolor"
```

ARPO-Policy (only rules - no fewshots)

```
"instructions": "Given the petal and sepal dimensions in cm, predict the
iris species.

Rules:

If the input dimensions are for a species with a lower sepal width, the
module should return a species with a lower sepal width. For example,
if the input dimensions are
for 'setosa', the module should return 'setosa' instead of 'virginica'.

If the input dimensions are for a species with a lower sepal width,
return a species with a lower sepal width. For example, if the input
dimensions are for 'setosa', return 'setosa' instead of 'virginica'.

If the input dimensions are for a species with a lower sepal width,
return a species with a lower sepal width. For example, if the input
dimensions are for 'setosa', return 'setosa' instead of 'virginica'."
```

ARPO-Ledger (both rule and few-shot)

```
"petal_length": "1.7",
"petal_width": "0.4",
"sepal_length": "5.4",
"sepal_width": "3.9",
"answer": "setosa"

Given the petal and sepal dimensions in cm, predict the iris species.
Rules ->

When given input dimensions, the module should focus on providing
instance-based reasoning that is specific to the given input. This can be
achieved by using the input dimensions
to identify the most common species associated with the given dimensions.
For example, if the input dimensions fall within the ranges most
commonly associated with the species Iris setosa, the module should
output 'setosa' as the answer.
```

ARPO-Hybrid (only few-shot) - Performs slightly worse than ARPO-Ledger.

SWEBench

Baseline:
"You are considering an issue from an open source Python repository. The issue has an associated test_patch that provides a set of tests to check whether the issue is resolved. The issue also has a corresponding gold_patch that provides a fix for the issue. Your task is to evaluate if this issue should be included in a question bank to benchmark coding ability. To evaluate an engineer's coding ability, the issue will be provided to them without the gold_patch and they will be tasked with writing code to resolve the issue. The test_patch will then be used to check whether the engineer's solution passes the tests in the test_patch. However, to be a fair evaluation, the issue must have a well-specified issue description that is clear and unambiguous about what is required for a successful solution. The engineer should have complete information about what is required to resolve the issue, just by reading the issue description. Please read the issue description and the test_patch carefully."

SIMBA (few-shot and rules)

"repo": "sympy/sympy",
"issue_description": "sylow_subgroup() IndexError \nI use sympy 1..... ...",
"gold_patch": "diff --git a\..."
"test_patch": "diff --git a/sympy/combinatorics/tests/test_perm_groups.py
b/sympy/combinatorics/tests/test_perm_groups.py\n--- ...",
"test_names": ["test_sylow_subgroup"],
"reasoning": "The issue description provides a clear example of the problem,
and the gold patch makes the necessary changes to fix the ‘IndexError‘
that occurs when calling ‘sylow_subgroup()‘.
The changes made to the ‘perm_groups.py‘ file ensure that the function
handles cases where the input group has multiple Sylow p-subgroups correctly.
The test patch provides additional test cases to verify that the changes make
the correct fixes to the ‘sylow_subgroup()‘ function. Overall, the issue
description, gold patch, and test patch are well-written and provide a clear
understanding of the problem and its solution.",
"score": "0"

Rules →
Imagine that you are an experienced software engineer who has been instructed to solve the issue. You have full access to the codebase, and can see the issue description as it is below. But you are not able to ask for clarification and would need to work exclusively from this information. Is the issue description well-specified enough for a meaningful attempt at a solution?
Provide a score from 0 to 3 based on the following criteria:
0: Well-specified and clearly solvable.
1: Some blanks but sensible interpretation possible.
2: Vague and ambiguous.
3: Nearly impossible to understand without more info.
Reason carefully about specific functions and classes before giving your answer.

Best ARPO (ARPO-Policy)

```
"repo": "sympy/sympy",
"issue_description": "ceiling(pos) > 0 should be true\nAlso, shouldn’t
‘floor(neg) < 0‘, ‘floor(pos) >= 0‘ and ‘ceiling(neg) <=0‘ evaluate to
True, too?",
"gold_patch": "diff --git a\...",
"test_patch": "diff --git a/sympy/functions/...",
"test_names": ["test_floor", "test_ceiling"],
"reasoning": "The issue description is well-specified enough for a
meaningful attempt at a solution. The description clearly states the
expected behavior of the floor and ceiling functions, including the cases
where the input is negative, non-negative, positive, or zero.
The description also explicitly states that the issue is with the behavior
of the floor and ceiling functions when the input is negative, non-negative,
positive, or zero.
This provides a clear understanding of what needs to be fixed.
However, the description does not provide enough information about
the specific functions and classes that are involved in the printing process,
which could lead to ambiguity in the solution.

Additionally, the gold patch appears to have fixed the issue by adding
specific tests for the behavior of the floor and ceiling functions
when the input is negative, non-negative, positive, or zero. This
suggests that the issue was indeed related to the behavior of these
functions in these cases.",
"score": "1"
```

Rules →
Imagine that you are an experienced software engineer who has been instructed to solve the issue. You have full access to the codebase, and can see the issue description as it is below. But you are not able to ask for clarification and would need to work exclusively from this information. Is the issue description well-specified enough for a meaningful attempt at a solution?
Provide a score from 0 to 3 based on the following criteria:
0: Well-specified and clearly solvable.
1: Some blanks but sensible interpretation possible.
2: Vague and ambiguous.
3: Nearly impossible to understand without more info.
Reason carefully about specific functions and classes before giving your answer.
The module should explicitly state that underscores should be ignored when converting a string to a float. It should also consider the context of the upstream bug in mpmath. To achieve this, the module should provide a clear definition of the required behavior for a successful solution. The module should also ensure that the test patch provides sufficient tests to determine whether the issue is resolved. For example, the module could provide test cases that cover different scenarios, such as converting strings with underscores to floats, and verify that the underscores are correctly ignored. Additionally, the module could provide a mechanism for the engineer to specify the context of the upstream bug in mpmath, such as by providing a parameter that allows the engineer to choose whether to ignore underscores or not. This would enable the engineer to tailor the solution to their specific needs and ensure that the issue is resolved correctly.
When formatting LaTeX expressions, ensure that superscripted symbols are correctly parenthesized. Use the `parenthesize_super` function to achieve this. For example, if the input is `x^{*2}`, the output should be `x^{(*2)}` instead of `x^{*2}`.

QA Task – HotPotQA

Baseline:
"Answer multiple choice questions"

SIMBA (both demos and rules)

```
"context": [
"Semichi Islands | The Semichi Islands (Samiyan in Aleut) are ..."
],
"question": "The Semichi Islands are located southeast of which island
that became uninhabited in 2010?",
"reasoning": "The context provides information about the Semichi Islands,
which are a group of small islands in the Aleutian Islands, Alaska. To
answer the question, we need to look for information about the island
that became uninhabited in 2010.
The context mentions that Nizki Island is the middle island of the
Semichi Islands group and is periodically
joined to Alaid Island by a sand spit. However, it does not mention
any information about the island becoming
uninhabited in 2010. We can look at the information about the other
islands, but there is no direct information about the island becoming
uninhabited in 2010. However, we can look at the information about the
island's name, which is said to derive from the Russian \"nizkiy\",
meaning \"low\". This information does not provide any clues about
the island becoming uninhabited in 2010.",
"answer": "Attu Island"
```

Rules →
When analyzing the context and question, ensure that you are not mistakenly identifying the floor space of one building as the other. Instead, carefully consider the information provided and provide the correct answer based on the context. For example, in this case, the context mentions that 15 Penn Plaza would have 2,050,000 square feet of floor space, so the correct answer is 15 Penn Plaza, not Manhattan West.

ARPO (Policy and Ledger only include demos

Instructions:
Answer multiple choice questions.

Rules →
When processing context information, consider leveraging general knowledge and external sources to supplement the provided information. This can help to improve the accuracy of the answer and provide a more comprehensive understanding of the topic.

Student–Teacher Prompt Comparisons

3B-Student, 3B-Teacher

Prompt:
Given a snippet from a job vacancy, identify all the ESCO job skills mentioned. Always return skills.

Rules →
When receiving a snippet from a job vacancy, consider the following patterns: keywords related to cloud technologies, job roles, and industry-specific terms. If the input text contains phrases like "public cloud offerings" or "private cloud services", consider returning the ESCO job skills related to cloud technologies, such as "cloud technologies" or "cloud infrastructure". Avoid returning the input text itself, and instead focus on extracting the relevant ESCO job skills.

When receiving a snippet from a job vacancy, consider the following patterns: keywords related to cloud technologies, job roles, and industry-specific terms. If the input text contains phrases like "public cloud offerings" or "private cloud services", consider returning the ESCO job skills related to cloud technologies, such as "cloud technologies" or "cloud infrastructure". Avoid returning the input text itself, and instead focus on extracting the relevant ESCO job skills.

3B-Student, 70B-Teacher

Prompt:
Given a snippet from a job vacancy, identify all the ESCO job skills mentioned. Always return skills.

Rules →
When identifying ESCO skills, the module should consider the input text more carefully to avoid missing relevant skills. For example, if the input text mentions contributing to open source projects, the module should consider skills related to open source software development, such as operating open source software.