LangProBe: a Language Programs Benchmark

Shangyin Tan[1], Lakshya A Agrawal[1], Arnav Singhvi[2], Liheng Lai[1], Michael J. Ryan[2], Dan Klein[1], Omar Khattab[3]*, Koushik Sen[1], Matei Zaharia[1,4]

[1]University of California, Berkeley, [2]Stanford University
[3]Massachusetts Institute of Technology, [4]Databricks

shangyin@berkeley.edu

Abstract

Composing language models (LMs) into multi-step *language programs* and automatically optimizing their modular prompts is now a mainstream paradigm for building AI systems, but the tradeoffs in this space have only scarcely been studied before. We introduce LangProBe, the first large-scale benchmark for evaluating the architectures and optimization strategies for language programs, with over 2000 combinations of tasks, architectures, optimizers, and choices of LMs. Using LangProBe, we are the first to study the impact of program architectures and optimizers (and their compositions together and with different models) on tradeoffs of quality and cost. We find that optimized language programs offer strong cost-quality Pareto improvement over raw calls to models, but simultaneously demonstrate that human judgment (or empirical decisions) about which compositions to pursue is still necessary for best performance.We have open-sourced LangProBe at https://github.com/Shangyint/langProBe

1 Introduction

Language models are now routinely used to build modular natural-language software systems that process data or serve bespoke applications. Previous work (Khattab et al., 2024; Opsahl-Ong et al., 2024; Schlag et al., 2023; Soylu et al., 2024) refer to such systems as *language programs*[1], where language model calls are wrapped into highly structured modules and pipelined with external tool calls. Language programs often compose better-scoped LM capabilities (Zaharia et al., 2024), offer structure for models to access tools or information (Lewis et al., 2020; Khattab et al., 2021; Lazaridou et al., 2022), and systematically scale

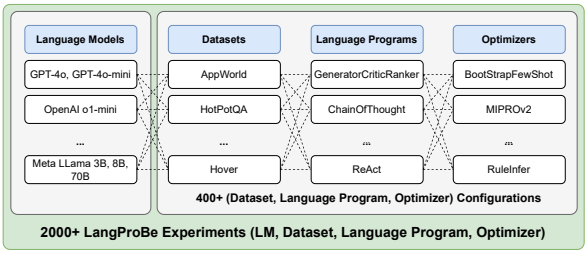


Figure 1: LangProBe includes 15 datasets, 4 optimizers, and more than 10 language programs, creating more than 400 configurations for evaluating language programs with different tasks and optimizers.

planning and search at inference time (Snell et al., 2024; Saad-Falcon et al., 2024).

To support such programming and permit portability across models and tasks, recent work has introduced declarative languages for expressing these systems and automating prompting (or fine-tuning) for their modules. For example, given a task-specific objective, frameworks like DSPy (Khattab et al., 2022, 2024) and TextGrad (Yuksekgonul et al., 2024) offer *optimizers* for language programs. By composing techniques like bootstrapping few-shot examples, refining free-form instructions, or fine-tuning, the optimizers transform the baseline language program into a new, optimized language program with updated parameters (including both LM prompts and LM weights), which often align better with a distribution of inputs or a nuanced task.

Despite the interest in this space, it remains unclear which problems actually need modular programs, especially as models continue to improve, or which types of architectures and optimizers will work best for different problems. To study this, we introduce **LangProBe** (Language Program Benchmark) (Figure 1), a benchmark for evaluating combinations of language models, program architectures, and their optimizers. For many datasets, LangProBe implements multiple language

*Work done while at Databricks.

[1]For a complete definition and examples of language programs, refer to Appendix A

Findings of the Association for Computational Linguistics: EMNLP 2025, pages 21489–21509

programs (Section 3.2), ranging from a single LM call to sophisticated and modular systems. Using LangProBe, we run over 2000 experiments to investigate several research questions.

> Do language programs and optimizers for them improve cost-performance compared to using raw model calls? (Section 5)

On average, we find that language programs show non-trivial improvement over raw model prediction baselines even while considering language model costs. For example, the best optimized program running on gpt-4o-mini performs better than both the gpt-4o and o1-mini raw model prediction baselines at a significantly cheaper cost. However, this is far from uniform, as many self-contained problems (e.g., MMLU) can be straightforwardly tackled by powerful models without composition or optimization.

> What language program architectures work best on different problems? (Section 6)

We find that modular programs are, perhaps expectedly, indispensable for tasks whose specification demands or strongly encourages access to external information or other tools. For example, tasks that require composing long-tail world knowledge benefit from retrieval-augmented generation (RAG) programs and multi-hop retrieval. While inference-time scaling programs are known to be helpful in certain cases, we find that they can fail to exceed baseline systems in certain applications, e.g. when error compounds across different modules. This illustrates a general theme we find, in which human judgment (or empirical decisions) about which compositions to pursue is still necessary for best performance.

> Which optimizers perform best and where? (Section 7)

All optimizers can provide quality gains for many—though not all—combinations of models, tasks, and programs, but different optimizers create rich tradeoffs in terms of optimization and inference costs. On average, we find that MIPRO (Opsahl-Ong et al., 2024), an optimizer that constructs and explores combinations of instructions and few-shot examples through Bayesian search, performs best overall. As in prior work, we find that searching over combinations of *self-bootstrapped* few-shot examples (BootstrapFewShotRandomSearch) is a highly competitive simple strategy, though the benefits of (bootstrapped) demonstrations heavily depend on the model–task pair. While this lies outside our scope, we believe that LangProBe defines a general methodology and large headroom for new prompt optimizers and also for finetuning- or RL-based optimizers to boost quality even further.

In summary, we contribute a **new benchmark for Language Programs** (Section 3), the **first systematic investigation of the cost–quality tradeoffs of Language Programs** (Section 5), and new **empirical insights** showing that employing appropriate language programs with cheaper models for some tasks is both cost- and performance-optimal compared to using the raw prediction from a stronger model and offering guidance on which language program architecture and optimizer choices work best (Section 6, Section 7).

2 Related Work

Composing language model calls with tool usage into complex applications is now popular, with the help of established language model programming libraries such as DSPy (Khattab et al., 2022, 2024), LangChain (Chase, 2022), or TextGrad (Yuksekgonul et al., 2024). Language programs compose and integrate knowledge and information flow and are often equipped with additional reasoning capabilities. For instance, RAG (Lewis et al., 2020) combines a language model with external retrieval from a corpus for knowledge-intensive tasks; Multi-Agent Debate (Du et al., 2023) harnesses multiple models debating each other to sharpen mathematical and strategic reasoning; Self-Refine (Madaan et al., 2023) iterates on its own outputs for continuous improvement; and ReAct (Yao et al., 2023) integrates step-by-step reasoning with actions to facilitate interactions with external environments. Recent research also focuses on scaling inference-time computation by generating multiple responses and verifying them with specialized models (Snell et al., 2024; Chen et al., 2024).

Recent work has introduced agentic benchmarks

[2]Cost for OpenAI's models are from (OpenAI, 2025) and for Meta's Llama models are from (Amazon Web Services, 2025). Inference costs are likely to vary across time and providers, but provide a reasonable relative comparison between models by the same provider.

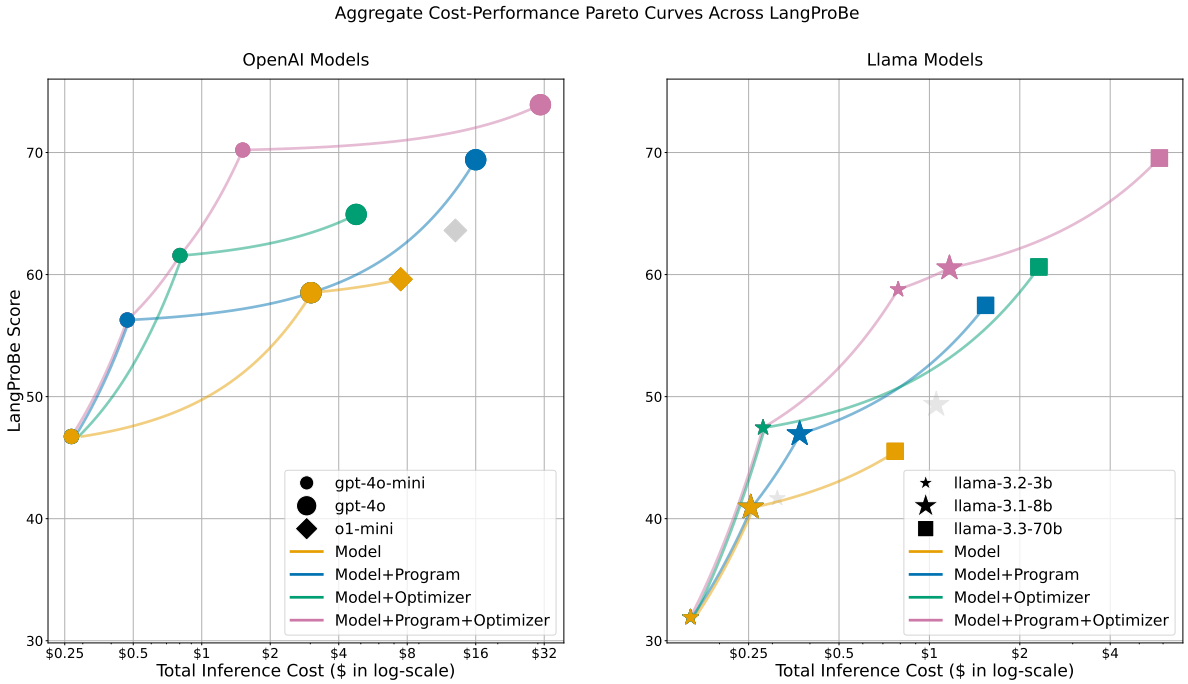


Figure 2: This figure shows stark cost-performance trade-offs across various configurations of (LM, Language Program, Optimizer), aggregated over multiple datasets in LangProBe. The Pareto curves represent the upper-left convex hull of achievable configurations. Piece-wise linear Pareto curve segments appear curved due to the log scale, but all points on the Pareto front are achievable via weighted (randomized) choice between the two endpoints. Four configurations are compared: 1) Model: Performance of baseline program (e.g., raw model predictions) without optimizers. 2) Model+Program: Performance with language programs applied, without optimizers. 3) Model+Optimizer: Performance with optimizers applied to the baseline program. 4) Model+Program+Optimizer: Performance of combined use of both language programs and optimizers. **Key Takeaway:** For both model families, the Model+Program+Optimizer Pareto curve deliver cost and quality improvements against Model+Program and Model+Optimizer Pareto curves, which in turn improve over the Model Pareto curve, implying that using language programs and optimizing them can offer considerable gains not only with respect to quality, but also cost[2].

(Kapoor et al., 2024; Zhou et al., 2024; Yang et al., 2024; Wang et al., 2024), which are closely related to LangProBe. As agents are a form of complex language programs, these benchmarks can help examine some design choices in language programs and we intend to add some of them to future iterations of LangProBe. Unfortunately, such agentic tasks leave plenty of unstudied problem types and are somewhat hard to adapt for asking questions about broader categories of program architectures (Stroebl et al., 2024). For example, SWE-bench (Yang et al., 2024) is helpful in evaluating software engineer-like agents, but it can be difficult to rely on it for testing general inference-time scaling or retrieval-augmented generation methods.

3 The LangProBe Benchmark

To offer an overview of LangProBe, we define different categories of benchmarks and then describe the selected programs and optimizers that we will study in this work. A detailed list of descriptions can be found in Appendix B and Appendix C.

3.1 Dataset Categories

We choose datasets across diverse categories. In contrast to evaluations focused on comparing *model capabilities*, e.g. HELM (Liang et al., 2023) or `lm-evaluation-harness` (Gao et al., 2024), for each category, we recast existing datasets into a uniform testbed with metrics and data splits, seeking to establish a proxy for applications that people program. Our categories include agentic tasks (AppWorld; Trivedi et al. 2024), coding and software engineering tasks (SweBench annotation tasks; OpenAI 2024, HumanEval; Chen et al. 2021), mathematical and reasoning tasks (MATH; Hendrycks et al. 2021b, GSM8K; Cobbe et al. 2021), domain-specific classification tasks (Iris; Fisher (1936) and Heart Disease; Janosi et al. (1989)), and question-answering problems (HotpotQA; Yang et al. 2018,

MMLU; Hendrycks et al. 2021a), and more.[3] While these serve an insightful starting point, we believe that future work must increasingly push LangProBe past the general capability datasets that model providers benchmark against and closer to the composite downstream problems that LM programmers seek to solve.

3.2 Language Programs

We adopt and design a diverse set of language program designs for the different tasks in LangProBe. By nature, language programs are much more structured, declarative, and compositional than free-form conversations with language models or community evaluation harnesses comparing model capabilities on self-contained tasks. This paradigm necessitates a uniform framework for posing our research questions. For this, we build our evaluation testbed in the DSPy framework, leveraging its offering of such language program composability and optimization of such systems, although in principle, future declarative languages for programming language models can support the same research questions we ask about programs, models, and optimizers.

General language programs Many programs architectures are general: with little to no changes, they can be used for *all* benchmarks. The simplest such program is to use a single DSPy `Predict` module, which translates to directly calling the language model in a structured manner with the task description and inputs and parsing its outputs. Similarly, we also include a DSPy chain-of-thought (CoT; Wei et al. 2023) program that uses CoT prompting, requiring the language model to output both reasoning and answer. We also adapt Archon's modular structure (Saad-Falcon et al., 2024) for designing more complex general language programs. Specifically, we use generators (generate a list of responses given a single query), critics (provide feedback for a list of responses), fusers (compile a list of responses into a single one), and rankers (rank the list of responses).

From these basic building blocks, we build two general language model programs: GenCriticFuser and GenCriticRanker. Namely, both pipelines first employ a generator module to generate a list of responses given inputs from the language model, then a critic module to provide strengths and weaknesses to the list of responses; finally, both a fuser and ranker module is used respectively to get the singular final result.

Specialized programs We also adopt problem-specific program architectures for certain tasks. For example, we define two retrieval-augmented generation (RAG) programs for some knowledge-intensive and classification tasks. We define the RAG program with a module that queries the retriever with the task input and then leverages a CoT module to generate the final response with the retrieved documents. Another RAG system, a highly simplified version of Baleen (Khattab et al., 2021), makes LM calls in a multi-hop design, systematically generating queries for the retriever, performing the retrieval and composing the set of retrieved documents to generate the final answer.

For benchmarks that require interactions within a closed-space environment (agent benchmarks), we build a ReAct (Yao et al., 2023) program, which reasons and generates actions simultaneously.

3.3 Optimizers

To measure the performances of these compound language programs, it is also important to measure how they perform under various prompt optimization techniques. LangProBe provides four general prompt optimization techniques for all language model programs.

BootstrapFewShot (Khattab et al., 2022, 2024) is a simple heuristic for building few-shot examples for all modules in arbitrary programs. Using a metric and a teacher model, it bootstraps "successful" demonstrations from the training set that pass the metric. The process iteratively refines predictions until the maximum number of bootstrapped demonstrations is reached and then included in the final prompt. The **BootstrapFewShotRandomSearch** optimizer searches through different combinations of bootstrapped few-shot demonstrations using a validation set, applying multiple optimization rounds with random search to select the best-performing set of few-shot examples to include in the final prompt.

MIPROv2 (Opsahl-Ong et al., 2024) optimizer produces sets of few-shot examples like BootstrapFewShot and then generates instruction candidates for each predictor in the program using a separate LM proposer program. To search for the best combination of few-shot examples and instruc-

[3] We borrow a few programs and datasets like HotPotQA, HoVer, Iris, and Heart Disease from the open-source DSPy repository, contributed by Khattab et al. (2024) and Opsahl-Ong et al. (2024), and adapt them to our more general setting.

Dataset Category	Datasets & Tasks	Specialized Programs
Code	HumanEval, SWEUnderspecified, SWEValidity	GeneratorCriticRanker, GeneratorCriticFuser
Reasoning	Judge, Scone	GeneratorCriticRanker, GeneratorCriticFuser
Agent	AppWorld	ReActBaseline, ReActAugmented
Knowledge	MMLU, HoVer, IReRa, HotpotQA, HotpotQAConditional, RAGQAArena	RAGBasedRank, RAG, MultiHopSummarize, SimplifiedBaleen
Classification	HeartDisease, Iris	CoTBasedVote, GeneratorCriticRanker, GeneratorCriticFuser
Math	MATH, GSM8K	GeneratorCriticRanker, GeneratorCriticFuser

Table 1: Dataset categories, the datasets associated with them, and the specialized language programs evaluated on each different category. We also provide a detailed description for each dataset, task, and language program in Appendix B and Appendix C.

tions, the optimizer uses Bayesian Optimization to identify the optimal instruction-examples set for each module in the program.

Finally, as part of this work, we introduce the **RuleInfer** optimizer, which runs the program through a single iteration of the BootstrapFewShot optimizer, allowing candidates to induce rules based on the successful proposed few-shot demonstrations, tasks, and instructions. These rules are then appended to the original instruction, generating refined instructions directly from the best-selected few-shot demonstrations.

4 Benchmarking Language Programs

Evaluating language programs with LangProBe are simply Cartesian products of dataset tasks, language programs, optimizers, and language models. While some language programs obviously do not work with specific types of datasets, we evaluate all such possible combinations.

To this end, we evaluate 16 distinct tasks from various open-source datasets with six different language models (*gpt-4o*, *gpt-4o-mini*, *o1-mini*, *Llama3.1-8B-Instruct*, *Llama3.2-3B-Instruct*, and *Llama3.3-70B-Instruct*) and a total of more than 10 different language model programs. We also apply four prompt optimization techniques described in Section 3.3 to all language model programs with different hyperparameters (Table 2). In total, we run over 2000 different combinations of language programs and dataset experiments.

5 Costs of Programs and Optimizers

In this section, we analyze the cost-performance trade-offs of using language models with varying inference costs and capabilities when combined with language programs and optimizers. Figure 2 visualizes these trade-offs as Pareto curves aggregated over multiple datasets in LangProBe. The x-axis uses a logarithmic scale for inference costs to accommodate the wide range of costs across different configurations.

We make the following observations from the Pareto curves: 1) **Every point on the Pareto-optimal curve is instantiable**: The Pareto-curves are calculated as an upper-left convex hull over linearly scaled performance-cost axes, and hence, every point on or below the curves is instantiable by cost-aware load balancing between the configurations at either end of a Pareto segment. 2) **Use of language programs or optimizers enables achieving better performance at lower cost:** the Model+Program+Optimizer Pareto curve dominates (achieve better performance at lower cost) the Model+Program and Model+Optimizer Pareto curves, which in turn dominate the Model Pareto curve, implying that using language programs and optimizing them has the capacity to support significant improvements not only with respect to performance, but also cost. 3) **Smaller models with effective programs and optimizers outperform larger models, at a lower cost**: We observe that often a smaller or cheaper LM with language program or optimization (or both) outperform configurations with larger or expensive LMs lacking language programs or optimization both in terms

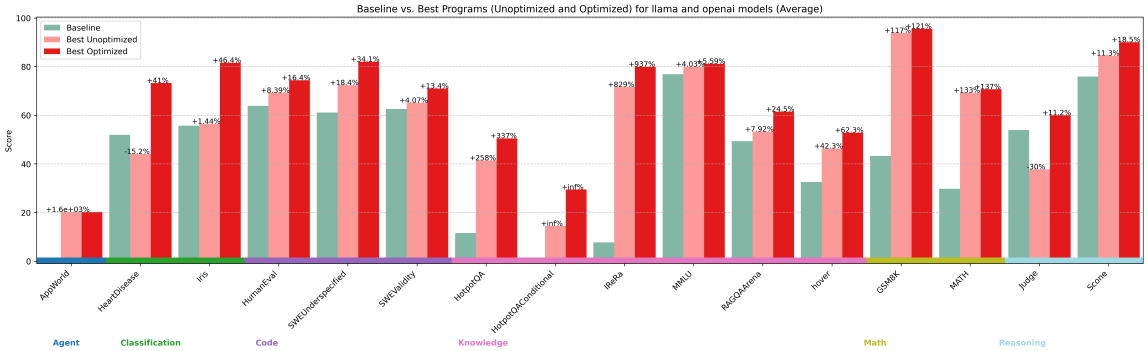


Figure 3: Performance comparison between best-performing programs and the baseline. In most cases, the baseline is a zero-shot call to the LM with task description and task inputs. Unoptimized programs are original, unchanged language programs, while optimized language programs are trained with updated prompts to adapt to the specific tasks. Scores are averaged from all language models we evaluated, including both OpenAI models and Llama models. We also include the same comparison of gpt-4o-mini only in Appendix F to reduce possible scattering of averaging results from different model families. Figure 8 delivers a similar conclusion as this plot: **in almost all tasks, both optimized and unoptimized programs perform better than the raw model prediction baselines.**

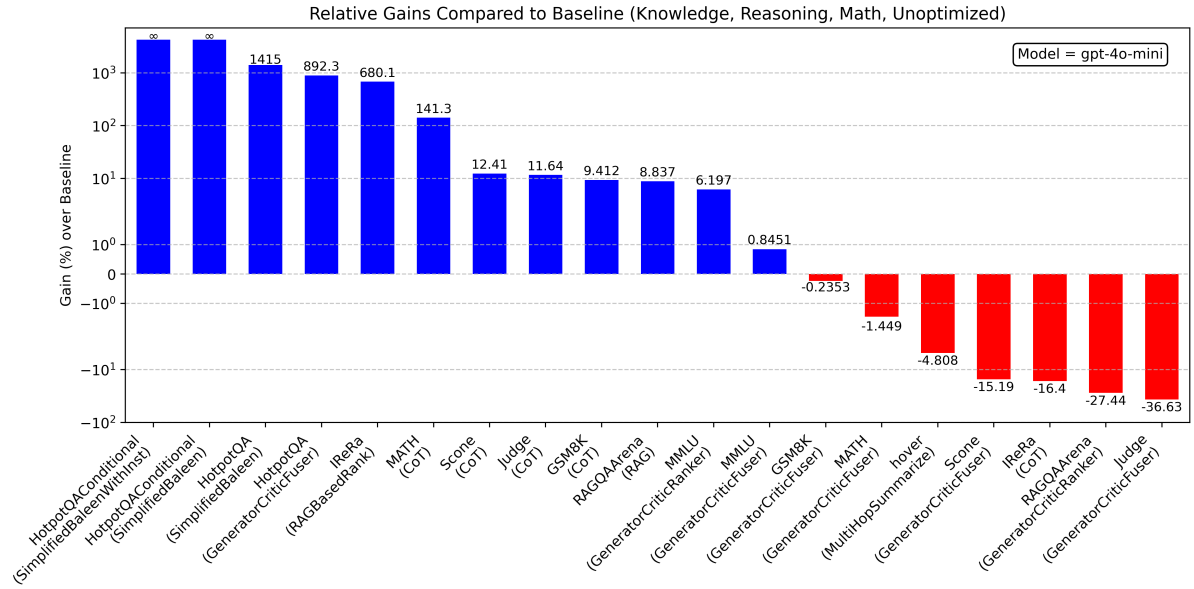


Figure 4: Performance comparison to Baseline, with *best* performing and *worst* performing programs for all Knowledge, Reasoning, and Math tasks. All programs are unoptimized. **On the same dataset, different language programs' performances vary. Similarly, the same language program's performance varies on different datasets.**

of cost and absolute performance. On aggregate, in Figure 2, we observe that gpt-4o-mini with language programs and optimization achieves 11.68% higher score than gpt-4o's baseline at just 50% of the cost, and gets slightly better performance than gpt-4o with language programs at 10% of the cost.

Among individual datasets, in Figure 9c, we see that gpt-4o-mini with program composition and optimization achieves 33.2% better result than gpt-4o without program composition and optimization at 18% lesser cost. These instances reflect the importance of composing compound systems to prompt language models and optimizing these systems, which can lead to increased downstream task performance at fractional costs.

In some instances, we observed that using an optimizer can actually reduce the inference cost for the same program while boosting performance. For instance, in GSM8K (Figure 11b), we see that gpt-4o-mini with optimization (BootstrapFewShotWithRandomSearch) achieves 8% better result at 5% lesser inference cost than gpt-4o-mini without optimization. Since optimizing the program with BootstrapFewShotWithRandomSearch

includes few-shot examples in the prompt, which increases the number of input-context tokens presented to the model but excludes the generation of lengthy reasoning traces in the output, and the notion that LM input token costs are cheaper compared to output token costs, we observe such cases.

In conclusion, our results demonstrate that language programs and optimizers can significantly enhance performance and cost-efficiency across tasks compared to raw model predictions on the aggregate.

6 Language Program Design

We now address the research question of what language program architecture achieves higher quality, e.g. whether structured language programs consistently outperform direct model prediction baselines and which programs contribute the most to performance gains. Additionally, we examine whether the effectiveness of different language programs varies across tasks, highlighting the conditions under which they provide the greatest benefit.

First, we show an overview of performance comparison between best language programs and raw model prediction baseline in Figure 3. In almost all cases, using **some** selected language programs, either unoptimized or optimized, gives a better performance than the raw model prediction baseline.

6.1 Can language program achieve higher quality compared to raw language model baseline?

We present the overall result of language program performance in Figure 3. In this figure, we observe constant improvement compared to the raw model prediction baseline.

On some knowledge-intensive benchmarks like HotPotQA or IReRa, the improvement is significant, with relative gains of +258% and +829%, respectively. In these benchmarks, the raw model lacks certain knowledge to directly answer complex questions or classify complex labels. However, with the addition of using a retriever as a tool, the structured language program is able to fetch additional information from the retriever and integrate the information to the context. This capability allows the program to address knowledge gaps that a raw model would struggle with, enabling accurate and context-aware responses.

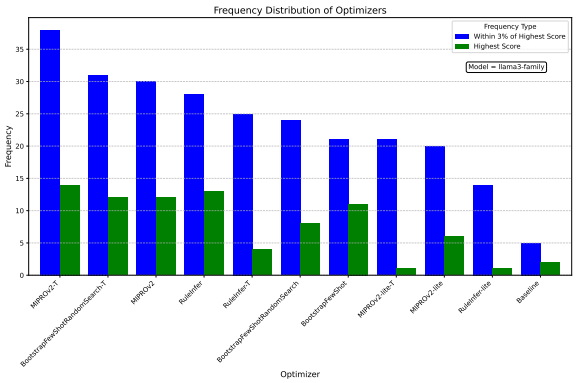


Figure 5: Frequency Distribution for individual optimizer performance, ranked by the number of times that an optimizer applied on a program is within 3% of that program's highest score (blue bar). We also note the number of the highest-performing optimizer as the top score (green bar). From the plot, **MIPROv2-T, which uses a stronger model for optimization to propose better instructions combined with corresponding few-shot examples through Bayesian search, works the best.**

6.2 Which language program designs provide more improvements?

Although Figure 3 provides a clear overview of the best-performing language programs on each dataset, it does not easily show the relative improvements achieved by different programs. To address this, we present Figure 4, which highlights, for each dataset in the Knowledge, Reasoning, and Math categories, both the best-performing and worst-performing programs and their performance relative to the baseline.

From Figure 4, the performance of different language programs varies significantly when compared to the baseline. On the left side, we observe significant improvement over the baseline by mostly retriever-augmented generation programs. Conversely, other programs, including GeneratorCriticRanker for RAGQAArena and GeneratorCriticFuser for JudgeBench perform worse than the baseline. Inspecting the evaluation traces, we find that when the program involves multiple modules, errors (like wrong format, factual errors, or hallucination) cascade from one module flow into other modules, causing errors to aggregate. Luckily, the error aggregation patterns, especially parsing errors, are mitigated through carefully curated few-shot examples and instructions from different optimization techniques as highlighted by the positive performance with both optimized variances.

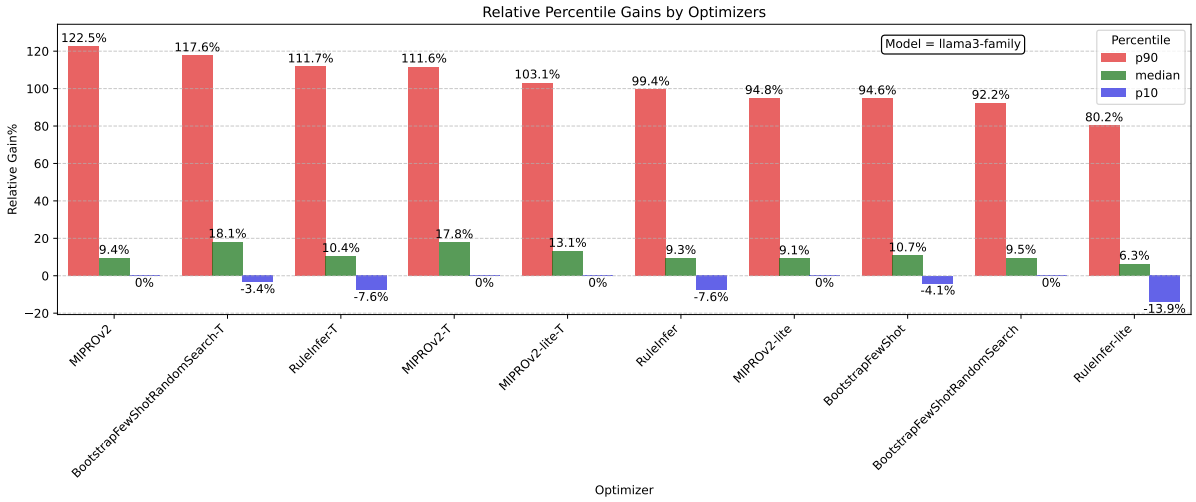


Figure 6: For all the experiments with Llama models, the relative gains by the optimizer compared to the same unoptimized program. We report 90th percentile, median, and 10th percentile. Optimizers with "lite" suffixes are configured with less compute resources, and optimizers with "T" are configured with a stronger optimizer model (gpt-4o-mini). From this plot, **in best case scenarios, all optimizers provide large performance gains of more than 80%. The median performance gains are from 6.3% to 18.1%. However, in some cases, optimizers do degrade the performance of the unoptimized program.**

Overall, these results demonstrate various gains due to compositional program architectures, but they also show that human judgment about which compositions to pursue (or a well-scoped search for tasks with enough data) is still required for best performance. In other words, there is no general "set it and forget it" strategy for program composition within the scope we considered.

6.3 When do language programs provide more improvements?

From both Figure 4 and the individual Pareto curves on each dataset in Appendix D, we observe larger improvement from language programs often associated with datasets or tasks that require *additional* information to complete. For example, HotPotQAConditional requires answering questions in a specific format that is unknown to the language model; IReRa is a classification task that needs additional information about the labels, etc. Language programs often show significant improvement over these datasets, which resembles a large class of real-world applications. On the other hand, tasks that language models are trained to perform, like MMLU or HumanEval, often see little to no benefit from unoptimized language programs. Figure 9f plots an overlapping Pareto curve for Model+Program and the baseline, demonstrating no performance benefits from using unoptimized language programs for HumanEval.

7 Optimizers

As in Section 6, programs are important for most datasets, as they provide additional context or more opportunities to reason with the language model. In addition to program design and architecture, prompt-based optimizations are also crucial to the performance of a compound AI system. In this section, we discuss whether the existing optimization techniques are effective by answering the following research questions.

7.1 What optimization techniques are more effective than others?

We rank the performance of each individual optimizer in Figure 5. From the plot, we observe that general optimizers that perform optimal instruction finding (MIPROv2 and its variants) and in-context learning (BootstrapFewShotRandomSearch and its variants) are leading in general, both having more experiments that appear within 3% of the highest score for the same dataset and program among all optimizer settings. The rule induction optimizers "RuleInfer" are good at obtaining the best scores but perform slightly worse to generalize for more tasks. This is due to its specialized nature, which allows RuleInfer to only work for tasks with obvious and conclusive rules from the examples.

Additionally, optimizers using a stronger model internally (suffices with "T") generally perform

better with an exception on RuleInfer. Because MIPRO and BootstrapFewshot all depend on good few-shot demonstrations, a stronger internal model provides better performance by generating better traces at a relatively lower cost.

7.2 How do performance gains from different optimizations vary across the distribution?

We list the relative gain for all optimizers in Figure 6. From the plot, at the 90th percentile, all optimizers demonstrate strong performance, ranging from 80.2% up to 122.5%. This illustrates that if used carefully, optimizers can provide substantial performance gains. For example, for the HeartDisease dataset, the MIPROv2 optimizer finds instructions like "*using the provided patient information, predict whether the patient has heart disease by analyzing the clinical parameters step by step and providing a rationale for your conclusion*" and a few good few-shot examples with detailed LM-generated reasoning. These optimized prompts increase the performance for Llama-3.2-3B from 26.32 to 76.32, resulting in an 189.97% increase.

The median of performance gains suggests the common effect of using optimizers. For most tasks, we expect a 5% to 20% from the optimizers. Finally, in some rare cases, performance degradation happens. Because most optimizers employ a separate validation set to control the quality of optimizations, the optimized language program may overfit the validation set, causing a performance drop on the final test set. RuleInfer and its variants have the most performance degradation due to non-transferable rules. Although the rules induced by the optimizer work well for the validation set, some may not directly help with the test set performance.

8 Conclusion

We presented LangProBe, a benchmark for evaluating language programs across a range of tasks. We used it to investigate far more comprehensively than prior work the interactions between architectures, optimization strategies, language models, and the resulting quality and cost tradeoffs. We hope that this starts a line of work on studying this new category of AI systems and that our current findings can already begin to offer practical guidance for researchers and practitioners designing and optimizing modular AI systems. Lastly, one of the primary goals of LangProBe is to facilitate the development and comparison of new language program architectures and optimization strategies.

Limitations

Due to compute constraints, LangProBe is not able to include more language programs (e.g., program-of-thought) and datasets (e.g., SWE-bench), which could allow more insightful observations. LangProBe does not run full evaluation on the newest reasoning models, like DeepSeek-R1 or OpenAI o3-mini, also due to budget constraint.

References

Amazon Web Services. 2025. Amazon bedrock pricing. Accessed on February 05, 2025.

Harrison Chase. 2022. LangChain.

Lingjiao Chen, Jared Quincy Davis, Boris Hanin, Peter Bailis, Ion Stoica, Matei Zaharia, and James Zou. 2024. Are more llm calls all you need? towards scaling laws of compound inference systems. *Preprint*, arXiv:2403.02419.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating large language models trained on code. *Preprint*, arXiv:2107.03374.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. 2021. Training verifiers to solve math word problems. *Preprint*, arXiv:2110.14168.

Karel D'Oosterlinck, Omar Khattab, François Remy, Thomas Demeester, Chris Develder, and Christopher Potts. 2024. In-context learning for extreme multi-label classification. *Preprint*, arXiv:2401.12178.

Yilun Du, Shuang Li, Antonio Torralba, Joshua B. Tenenbaum, and Igor Mordatch. 2023. Improving

factuality and reasoning in language models through multiagent debate. *Preprint*, arXiv:2305.14325.

R. A. Fisher. 1936. Iris. UCI Machine Learning Repository. DOI: https://doi.org/10.24432/C56C76.

Leo Gao, Jonathan Tow, Baber Abbasi, Stella Biderman, Sid Black, Anthony DiPofi, Charles Foster, Laurence Golding, Jeffrey Hsu, Alain Le Noac'h, Haonan Li, Kyle McDonell, Niklas Muennighoff, Chris Ociepa, Jason Phang, Laria Reynolds, Hailey Schoelkopf, Aviya Skowron, Lintang Sutawika, Eric Tang, Anish Thite, Ben Wang, Kevin Wang, and Andy Zou. 2024. A framework for few-shot language model evaluation.

Rujun Han, Yuhao Zhang, Peng Qi, Yumo Xu, Jenyuan Wang, Lan Liu, William Yang Wang, Bonan Min, and Vittorio Castelli. 2024. Rag-qa arena: Evaluating domain robustness for long-form retrieval augmented question answering. *Preprint*, arXiv:2407.13998.

Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. 2021a. Measuring massive multitask language understanding. *Preprint*, arXiv:2009.03300.

Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. 2021b. Measuring mathematical problem solving with the MATH dataset. In *NeurIPS Datasets and Benchmarks*.

Andras Janosi, William Steinbrunn, Matthias Pfisterer, and Robert Detrano. 1989. Heart disease. UCI Machine Learning Repository. https://doi.org/10.24432/C52P4X.

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. *Preprint*, arXiv:2011.03088.

Sayash Kapoor, Benedikt Stroebl, Zachary S. Siegel, Nitya Nadgir, and Arvind Narayanan. 2024. Ai agents that matter. *Preprint*, arXiv:2407.01502.

Omar Khattab, Christopher Potts, and Matei A. Zaharia. 2021. Baleen: Robust multi-hop reasoning at scale via condensed retrieval. In *NeurIPS*, pages 27670–27682.

Omar Khattab, Keshav Santhanam, Xiang Lisa Li, David Hall, Percy Liang, Christopher Potts, and Matei Zaharia. 2022. Demonstrate-search-predict: Composing retrieval and language models for knowledge-intensive NLP. *CoRR*, abs/2212.14024.

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. 2024. Dspy: Compiling declarative language model calls into self-improving pipelines. In *ICLR*.

Hynek Kydlicek, Alina Lozovskaya, Nathan Habib, and Clémentine Fourrier. 2025. Fixing open llm leaderboard with math-verify. https://huggingface.co/blog/math_verify_leaderboard. Accessed: 2025-02-21.

Angeliki Lazaridou, Elena Gribovskaya, Wojciech Stokowiec, and Nikolai Grigorev. 2022. Internet-augmented language models through few-shot prompting for open-domain question answering. *CoRR*, abs/2203.05115.

Patrick S. H. Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. 2020. Retrieval-augmented generation for knowledge-intensive NLP tasks. In *NeurIPS*.

Percy Liang, Rishi Bommasani, Tony Lee, Dimitris Tsipras, Dilara Soylu, Michihiro Yasunaga, Yian Zhang, Deepak Narayanan, Yuhuai Wu, Ananya Kumar, Benjamin Newman, Binhang Yuan, Bobby Yan, Ce Zhang, Christian Alexander Cosgrove, Christopher D Manning, Christopher Re, Diana Acosta-Navas, Drew Arad Hudson, Eric Zelikman, Esin Durmus, Faisal Ladhak, Frieda Rong, Hongyu Ren, Huaxiu Yao, Jue WANG, Keshav Santhanam, Laurel Orr, Lucia Zheng, Mert Yuksekgonul, Mirac Suzgun, Nathan Kim, Neel Guha, Niladri S. Chatterji, Omar Khattab, Peter Henderson, Qian Huang, Ryan Andrew Chi, Sang Michael Xie, Shibani Santurkar, Surya Ganguli, Tatsunori Hashimoto, Thomas Icard, Tianyi Zhang, Vishrav Chaudhary, William Wang, Xuechen Li, Yifan Mai, Yuhui Zhang, and Yuta Koreeda. 2023. Holistic evaluation of language models. *Transactions on Machine Learning Research*. Featured Certification, Expert Certification.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. 2023. Self-refine: Iterative refinement with self-feedback. *Preprint*, arXiv:2303.17651.

Princeton NLP. 2024. Swe-bench verified dataset. Accessed: 2025-02-15.

OpenAI. 2024. Introducing swe-bench verified.

OpenAI. 2025. Openai api pricing. Accessed on February 5, 2025.

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. In *EMNLP*, pages 9340–9366. Association for Computational Linguistics.

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. *Preprint*, arXiv:2406.11695.

Jon Saad-Falcon, Adrian Gamarra Lafuente, Shlok Natarajan, Nahum Maru, Hristo Todorov, Etash Guha, E. Kelly Buchanan, Mayee Chen, Neel Guha, Christopher Ré, and Azalia Mirhoseini. 2024. Archon: An architecture search framework for inference-time techniques. *Preprint*, arXiv:2409.15254.

Imanol Schlag, Sainbayar Sukhbaatar, Asli Celikyilmaz, Wen tau Yih, Jason Weston, Jürgen Schmidhuber, and Xian Li. 2023. Large language model programs. *Preprint*, arXiv:2305.05364.

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

Charlie Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. 2024. Scaling llm test-time compute optimally can be more effective than scaling model parameters. *Preprint*, arXiv:2408.03314.

Dilara Soylu, Christopher Potts, and Omar Khattab. 2024. Fine-tuning and prompt optimization: Two great steps that work better together. *Preprint*, arXiv:2407.10930.

Benedikt Stroebl, Sayash Kapoor, and Arvind Narayanan. 2024. Inference scaling flaws: The limits of llm resampling with imperfect verifiers. *Preprint*, arXiv:2411.17501.

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. *Preprint*, arXiv:2410.12784.

Harsh Trivedi, Tushar Khot, Mareike Hartmann, Ruskin Manku, Vinty Dong, Edward Li, Shashank Gupta, Ashish Sabharwal, and Niranjan Balasubramanian. 2024. Appworld: A controllable world of apps and people for benchmarking interactive coding agents. In *ACL (1)*, pages 16022–16076. Association for Computational Linguistics.

Cunxiang Wang, Ruoxi Ning, Boqi Pan, Tonghui Wu, Qipeng Guo, Cheng Deng, Guangsheng Bao, Xiangkun Hu, Zheng Zhang, Qian Wang, and Yue Zhang. 2024. Novelqa: Benchmarking question answering on documents exceeding 200k tokens. *Preprint*, arXiv:2403.12766.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. 2023. Chain-of-thought prompting elicits reasoning in large language models. *Preprint*, arXiv:2201.11903.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. Swe-agent: Agent-computer interfaces enable automated software engineering. *Preprint*, arXiv:2405.15793.

Zhilin Yang, Peng Qi, Saizheng Zhang, Yoshua Bengio, William W. Cohen, Ruslan Salakhutdinov, and Christopher D. Manning. 2018. Hotpotqa: A dataset for diverse, explainable multi-hop question answering. In *EMNLP*, pages 2369–2380. Association for Computational Linguistics.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. React: Synergizing reasoning and acting in language models. *Preprint*, arXiv:2210.03629.

Mert Yuksekgonul, Federico Bianchi, Joseph Boen, Sheng Liu, Zhi Huang, Carlos Guestrin, and James Zou. 2024. Textgrad: Automatic "differentiation" via text. *Preprint*, arXiv:2406.07496.

Matei Zaharia, Omar Khattab, Lingjiao Chen, Jared Quincy Davis, Heather Miller, Chris Potts, James Zou, Michael Carbin, Jonathan Frankle, Naveen Rao, and Ali Ghodsi. 2024. The shift from models to compound ai systems.

Shuyan Zhou, Frank F. Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, Uri Alon, and Graham Neubig. 2024. Webarena: A realistic web environment for building autonomous agents. *Preprint*, arXiv:2307.13854.

A Language Programs

Language programs are modular systems around structured LLM calls. Soylu et al. (2024) defines language program as a program $\Phi : X \Rightarrow Y$, whose execution breaks down inputs into calls for a set of language model modules $M =< M\_1, ..., M\_m >$, each representing an LM invocation, declaratively defined in terms of its desired input/output behavior. Note that here, the LM modules are parameterized with prompts or language model weights, which are tuned by optimizers.

We show a common language programs (RAG) in two popular frameworks, DSPy and LangChain, in Figure 7. While the two programs are different in syntactical forms, they both contains abstractions around building and invoking language model modules, together with allowing tool calling (retriever) in-between.

We list dataset categories and specialized programs for them in Table 1.

```python
class RAG(dspy.Module):
    def __init__(self, num_passages=3):

        self.retrieve = dspy.Retrieve(k=num_passages)
        self.generate_query = dspy.ChainOfThought("question -> search_query")
        self.generate_answer = dspy.ChainOfThought("context, question -> answer")

    def forward(self, question):
        search_query = self.generate_query(question=question).search_query
        passages = self.retrieve(search_query).passages

        return self.generate_answer(context=passages, question=question)
```

```python
query_prompt = ChatPromptTemplate.from_template(
    "Generate a concise Wikipedia search query for the question: {question}"
)

generate_query_chain = (
    RunnablePassthrough.assign(question=lambda x: x["question"])
    | query_prompt | llm | StrOutputParser()
)

retriever = WikipediaRetriever(top_k_results=3, lang="en")

def format_docs(docs: list[Document]) -> str:
    return "\n\n".join(doc.page_content for doc in docs)

answer_prompt = ChatPromptTemplate.from_template(
    """
    Answer the question based only on the context provided.
    Context: {context}
    Question: {question}
    """
)

chain = (
    { "question": RunnablePassthrough(), "query": generate_query_chain, }
    | RunnableMap({
      "question": lambda x: x["question"],
      "context": lambda x: retriever.invoke(x["query"]) | format_docs,
    })
    | answer_prompt | llm | StrOutputParser()
)
```

Figure 7: We show a RAG langauge program in two different popular frameworks, DSPy (up) and LangChain (bottom). Although two frameworks provide different abstractions, i.e., different declarative language for writing language programs, they both contain structured ways to interact with LLMs. DSPy provide signatures, while LangChain uses prompt templates. In LangProBe, we use DSPy for easier integration and evaluation of existing optimization techniques. We open-sourced our evaluation suite and encourage language program contributions from all frameworks.

B Dataset Descriptions

LangProBe uses the following open-sourced datasets for research purposes only.

AppWorld (Trivedi et al., 2024)
Task: AppWorld databases start in some initial state set by the benchmark. The agent should take certain actions during the execution to change the database into another state.
Input: A question about mobile applications, along with its supervisor's information like name, phone number, and email.
Output: python code that will be executed by AppWorld server.
Evaluation metrics: the output python code will be executed by the AppWorld server and check if the database after execution is the same as the ideal final state. Success is marked by passing all the unit tests for state check and no unwanted actions are taken. Otherwise, it's a failure.
License: Apache 2.0

SweBenchVerifiedAnnotation (NLP, 2024)
Task: to examine each sample in the SWE-bench test set for properly defined issue descriptions and unit tests with appropriate scope by giving them scores.
Input: the repository name containing the issue, the issue description, gold patch, test patch, and the names of the tests in the test patch that will be used to evaluate the solution.
Output: a score from 0 to 3 based on certain 4 criteria.
Evaluation metrics: the ground truth is also a score based on the same criteria. We calculate the string equivalence between the predicted score and the ground truth score.
License: MIT

MATH (Hendrycks et al., 2021b)
Task: The Mathematics Aptitude Test of Heuristics (MATH) dataset consists of problems from mathematics competitions, including the AMC 10, AMC 12, AIME, and more. Each problem in MATH has a full step-by-step solution, which can be used to teach models to generate answer derivations and explanations.
Input: a math question written in LaTeX and natural language.
Ground truth: step-by-step solution written in LaTeX and natural language with the final answer enclosed in LaTeX's \boxed tag.
Evaluation metrics: string equivalence between the predicted mathematical answer and the gold solution extracted from the LaTeX's \boxed tag. We adopt Hendrycks et al. (2021b)'s evaluation. Arguably, a more prominent evaluator like MathVerify (Kydlicek et al., 2025) would report fairer (and higher) scores for all programs and optimizers, which is left as future work.
License: MIT

GSM8K (Cobbe et al., 2021)
Task: solving high-school-level mathematics problems across various topics. These problems are presented in natural language, and the model is required to produce correct solutions.
Input: The question to a grade school math problem in natural language
Output: The solution to this problem with step by step reasoning in natural language. The numerical solution is always at the end of the solution string.
Evaluation metrics: the ground truth is the solution to this problem with step-by-step reasoning in natural language. We calculate the integer equivalence between the predicted mathematical answer and the gold solution.
License: MIT

HotPotQA (Yang et al., 2018)
Task: HotpotQA is a new dataset with 113k Wikipedia-based question-answer pairs. Our task is to answer questions that require reasoning across multiple supporting documents.
Input: question in natural language
Output: answer to the question in natural language
Evaluation metrics: ground truth is the answer to the question in natural language. We evaluate the string equivalence between the predicted answer and the ground truth answer.
License: CC BY-SA 4.0

HumanEval (Chen et al., 2021)
Task: The HumanEval dataset released by OpenAI includes 164 programming problems with a function signature, docstring, body, and several unit tests. The task is to produce reliable and executable code that passes the unit tests given.
Input: the prompt for function specification that includes necessary import statements, function signatures, and docstring of unit tests.
Ground truth: generated code snippet following the function specification.

Evaluation metrics: binary indicator that specifies whether generated code passes the test cases defined in the ground truth.
License: MIT

MMLU (Hendrycks et al., 2021a)
Task: answer multiple-choice questions across a broad range of knowledge domains.
Input: description of the question along with its four options in natural language.
Ground truth: the correct option for this question
Evaluation metrics: ground truth is provided as the correct option in natural language ("A", "B", "C", or "D"). We evaluate the string equivalence between the predicted option and the ground truth option.
License: MIT

IReRa (D'Oosterlinck et al., 2024)
Task: solve multi-label classification tasks with an extreme number of classes
Inputs: a textual description
Outputs: all the ESCO job skill labels mentioned in natural language.
Evaluation metrics: rank-precision (RP) of the produced rankings, calculated from a list of true relevant items and a list containing the predicted items in ranked order.
License: MIT

Heart Disease (Janosi et al., 1989)
Task: classify the patient's heart disease status based on the information provided
Inputs: textual description of patient heart disease attributes, including age, slope, chol, ca, thal, restecg, exang, trestbps, cp, thalach, fbs, oldpeak, and sex.
Outputs: A binary classification ("yes" or "no") indicating whether the patient has heart disease.
Evaluation metrics: we evaluate the string equivalence between the predicted diagnosis and the ground truth diagnosis ("yes" or "no").
License: CC BY 4.0

HoVer (Jiang et al., 2020)
Task: perform multi-hop evidence retrieval of a claim to determine whether it's supported or not.
Inputs: a claim about a fact in natural language.
Outputs: documents being retrieved concatenated as a single string.
Evaluation metrics: The ground truth is the fact the model is required to retrieve. We calculate the proportion of examples in which the model is required to retrieve at least one supporting fact from each supporting document and accurately predict the correct label.
License: CC BY-SA 4.0

Iris (Fisher, 1936)
Task: predict the species of an iris flower based on its sepal and petal measurements.
Inputs: petal and sepal dimensions in cm about the iris species in natural language.
Outputs: the class of iris plant (setosa, versicolor, or virginica).
Evaluation metrics: the ground truth is the iris class in natural language. We evaluate the string equivalence between the predicted class and the ground truth class.
License: CC BY 4.0

Scone (Scoped Negation Benchmark) (She et al., 2023)
Task: ScoNe-NLI contains contrast sets of six examples where entailment relations are impacted by the scope of one or two negations. The main task is to test how well models understand and reason about negation in natural language.
Inputs: a context dependent question and its context in natural language
Outputs: "yes" or "no"
Evaluation metrics: ground truth answer to the question is "yes" or "no". We evaluate the string equivalence between the predicted answer and the ground truth answer.
License: CC0 1.0

RAG-QA Arena "Technology" (Han et al., 2024)
Task: generate quality answers for technology questions.
Inputs: question in natural language.
Outputs: response to the question in natural language.
Evaluation metrics: ground truth is a response in natural language. We calculate the semantic F1 score between the model response and the ground truth response. Note here the language model used to evaluate F1 is the same as the model being evaluated.
License: Apache 2.0

JudgeBench (Tan et al., 2024)
Task: evaluating LLM-based judges for objective correctness on challenging response pairs.

Inputs: a question, response A and response B in natural language
Outputs: which response is better. Either A>B or B>A.
Evaluation metrics: ground truth is which one is better, either "A>B" or "B>A". string equivalence between the predicted answer and the ground truth answer.
License: MIT

C Language Program Descriptions

CoT
Given an instruction prompt, Chain of thought produces a step-by-step explanation leading to the final answer.
Number of LLM calls: 1

RAG
Retrieval Augmented Generation first retrieves top-k most relevant passages using a retriever. These retrieved passages are then integrated as context into the model's input and passed through a Chain-of-Thought reasoning process to generates a final response.
Number of LLM calls: 1

ReActBaseline
Re-Act is a combination of reasoning and action in a step-by-step manner. Usually, the action involves retrieving relevant information from external sources and then integrates reasoning and action-based decision-making into the model's process. In our case of AppWorld, the action involves generating code that will be executed by the AppWorld server.
Number of LLM calls: equivalent to the number of reasoning steps (the number of reasoning steps defaults to 40 in the case of AppWorld).

ReActAugmented
Same as ReActBaseline but with few-shot demonstration added.
Number of LLM calls: same as ReActBaseline

SimplifiedBaleen
First generates multiple search queries and iterate through each one of them in multiple "hops," where each hop retrieves relevant passages from a knowledge base using a retriever. These retrieved passages are then aggregated as contexts and are passed through a Chain-of-Thought reasoning process to generate the final response.
Number of LLM calls: equivalent to the number of hops (defaults to be 2).

SimplifiedBaleenWithInst The setup is the same as SimplifiedBaleen except for a manually written prompt. In our case of HotpotQA conditional, the prompt is the following: When the answer is a person, respond entirely in lowercase. When the answer is a place, ensure your response contains no punctuation. When the answer is a date, end your response with "Peace!". Never end your response with "Peace!" under other circumstances. When the answer is none of the above categories respond in all caps.
Number of LLM calls: same as SimplifiedBaleen.

GeneratorCriticRanker
Produce a list of candidate responses from a given instruction prompt, then for each of these responses, it identifies the strengths/weaknesses for each candidate response. Then it returns a ranked list of top-K candidate responses.
Number of LLM calls: equivalent to the number of candidate * 2 + 1 (the number of candidates defaults to be 5).

GeneratorCriticFuser
Produce a list of candidate responses from a given instruction prompt, then for each of these responses, it identifies the strengths/weaknesses for each candidate response. Then it returns a single, high-quality response.
Number of LLM calls: equivalent to the number of candidate * 2 + 1 (the number of candidates defaults to be 5).

RAGBasedRank
It's an in-context learning framework designed for multi-label classification with an extremely large number of classes (IReRa). The process begins by generating queries based on the input and retrieving documents from a fixed retriever. Then the retrieved documents are re-ranked by an LM.
Number of LLM calls: 2. One for generating the query for retrieval and one for re-ranking the top k retrieved documents.

MultiHopSummarize
It begins by retrieving and summarizing the top k relevant passages for a given claim. Subsequent

hops generate refined queries based on previous summaries, retrieving additional passages. In total of 3 iteration performed to gather information about the initial question.
Number of LLM calls: equivalent to the number of hops (the number of hops defaults to 7 in the case of HoVer).

CoTBasedVote
It first applies multiple Chain-of-Thought classifiers to generate independent predictions (votes) that includes both the reasoning and the answer to the question. These votes are then assessed and consolidated through critical evaluation of these "opinions".
Number of LLM calls: equivalent to the number of voters + 1. (the number of voters defaults to be 3 in the case of Heart Disease).

D Cost-Performance Pareto Curves for all benchmarks

Figure 2 visualizes the Performance-Cost Pareto optimal configuration aggregated over 7 datasets in LangProBe (Scone, HotpotQA, HumanEval, HeartDisease, Judge, Iris, HotpotQAConditional) and Figures 11, 9, and 10 and visualize the Performance-Cost Pareto optimal configurations for individuals datasets in LangProBe.

E Hyperparameter Settings for Optimizers

Table 2 summarizes the optimizer configurations used in LangProBe.

F Program Comparison for `gpt-4o-mini`

To show that results from Appendix F are not scattered or biased averaging result from different model families, we also show the Baseline vs. Best Programs plot using data from `gpt-4o-mini` in Figure 8. This plot delivers the same message as Figure 3. In most cases, optimized programs perform better than unoptimized programs, which all are much better than language model call baselines.

G Introducing RuleInfer Optimizer

We introduce RuleInfer, a prompt optimization approach that identifies actionable rules to optimize the DSPy program performance. We provide a diagram of RuleInfer in Figure 12. Building upon the BootstrapFewShot optimizer, RuleInfer leverages an LLM to perform rule induction on the generated successful few-shot demonstrations, ascertaining specific insights grounded by the "positive" behavior from the examples to improve model performance. The optimizer then appends these sets of rules to the original task instructions and produces candidates of optimized prompts with natural language guidelines that are validated iteratively with the final output of a best-performing optimized program.

In comparison to optimizers like BootstrapFewShot (designed mainly for producing examples) and MIPRO (which proposes both instructions and few-shots), RuleInfer offers a distinct way of incorporating grounded, actionable rules derived from existing few-shots. RuleInfer excels in tasks with clear, discrete constraints, such as classification (HeartDisease and Iris) or coding domains (HumanEval and SWEBench), where well-defined rules can be leveraged to create structured decision boundaries for the language model to consider and hence reinforce both consistency and accuracy in performance. This quality makes RuleInfer particularly effective at aligning model behavior to task requirements without extensive manual engineering of the prompt instructions specifically, as the LLM performing rule induction strengthens this alignment. However, the optimizer lacks benefits on tasks are less domain-specific like question-answering tasks (HotPotQA, MMLU) as the induced rules tend can be too broad for open-ended general knowledge queries.

Algorithm 1 RULEINFER: Inducing Rules from Few-Shot Demonstrations to Optimize LM Programs

Require: Initial program Φ, Training set X, Validation set X', Candidates N

$\Phi^* \leftarrow \text{ApplyFewShots}(\Phi, X)$
$\mu^* \leftarrow \text{Evaluate}(\Phi^*, X')$ ▷ Compute initial score
for $n = 1, \ldots, N$ **do**
  $\Phi_n \leftarrow$ Apply rule induction to Φ^* using X
  $\mu_n \leftarrow \text{Evaluate}(\Phi_n, X')$
  if $\mu_n > \mu^*$ **then**
    $\Phi^* \leftarrow \Phi_n$ ▷ Update best program
    $\mu^* \leftarrow \mu_n$ ▷ Update best score
  end if
end for
return Φ^* ▷ Final optimized program

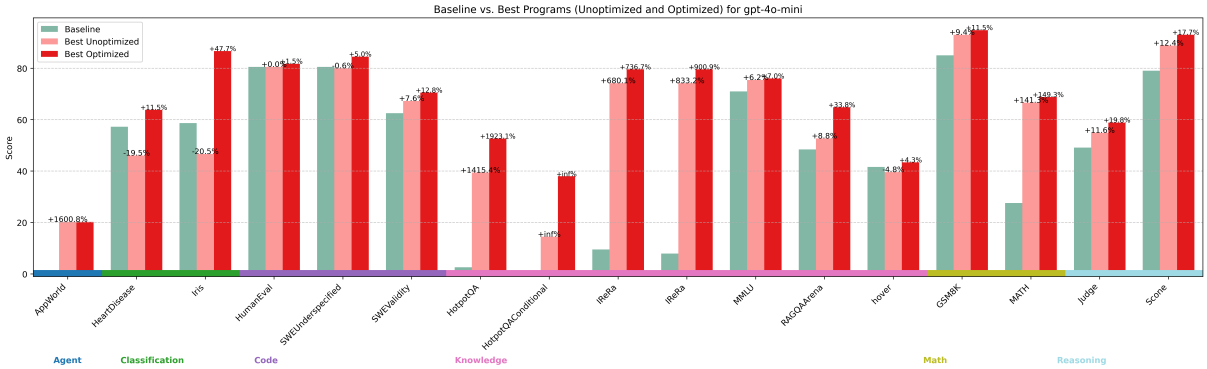


Figure 8: Performance comparison between best-performing programs and the baseline for *gpt-4o-mini*, to reduce result scattering between different models. This plot deliver the same message as Figure 3.

BootstrapFewShot	
Init Args	`max_errors=5000 max_labeled_demos=2`
Compile Args	`(none)`
BootstrapFewShotWithRandomSearch	
Init Args	`max_errors=5000 max_labeled_demos=2 num_threads=16`
Compile Args	`(none)`
MIPROv2-lite	
Init Args	`max_errors=5000 auto="medium" num_threads=16`

Compile Args	`num_trials=20`
	`max_bootstrapped_demos=4`
	`max_labeled_demos=2`

MIPROv2	
Init Args	`max_errors=5000 num_threads=16 num_candidates=12`

Compile Args	`num_trials=50`
	`max_bootstrapped_demos=4`
	`max_labeled_demos=2`
	`batch_size=35`
	`batch_full_eval_steps=5`

RuleInfer-lite	
Init Args	`max_errors=5000 num_candidates=10 num_rules=10 num_threads=8`
Compile Args	`(none)`
RuleInfer	
Init Args	`max_errors=5000 num_candidates=10 num_rules=20 num_threads=8`
Compile Args	`(none)`

Table 2: Summary of default optimizer configurations.

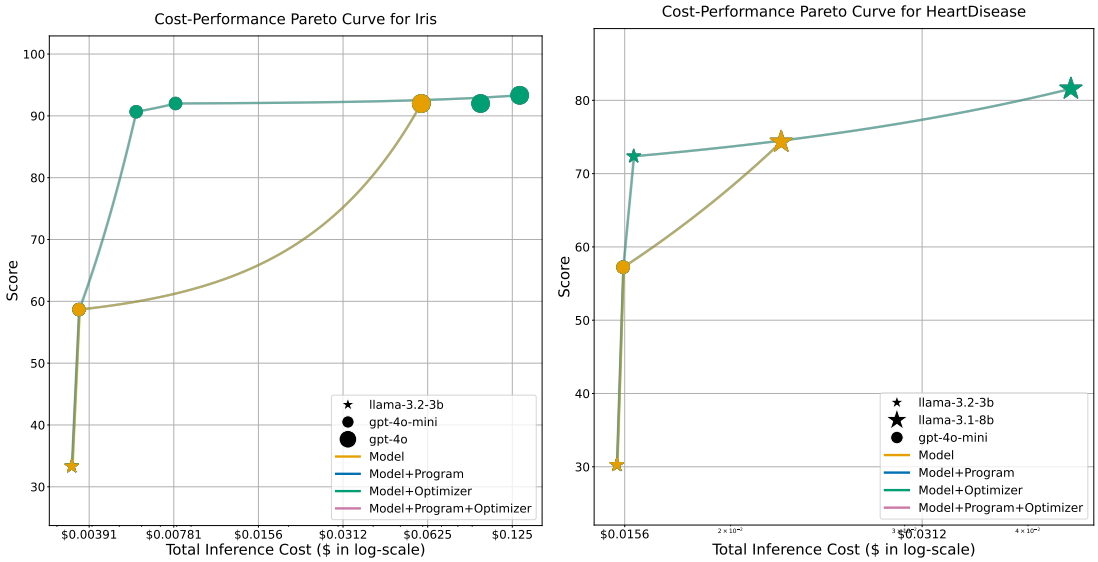
RuleInfer demonstrates how the LangProBe benchmark can be leveraged to introduce and validate new prompt optimization techniques across various tasks. By comparing RuleInfer across existing DSPy optimizers across multiple tasks, programs and models, we provide a streamlined analysis to easily pinpoint instances where rule-based induction excels versus where it underperforms. This will greatly benefit both prompt optimization developers in benchmarking novel techniques against previous ones and prompt workflow developers in understanding when to use what kinds of optimizers based on their task and program.

H Models Evaluated

We include all models we evaluated on LangProBe. OpenAI models are available through https://platform.openai.com/ and Llama models are available on https://huggingface.co/ with the handle below.

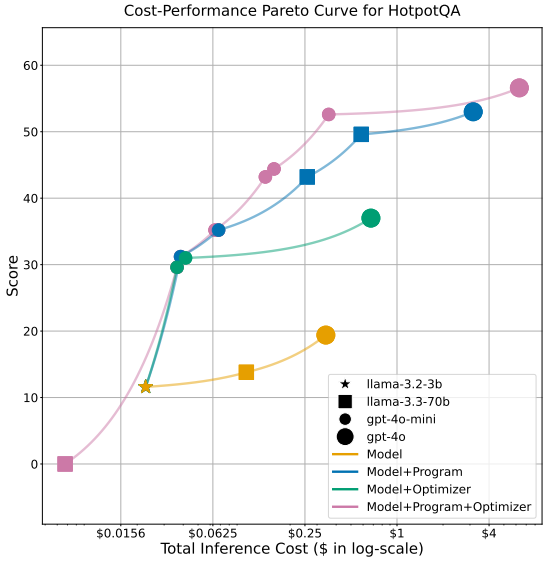
- `meta-llama/Llama-3.1-8B-Instruct`

- meta-llama/Llama-3.2-3B-Instruct
- meta-llama/Llama-3.3-70B-Instruct
- gpt-4o-mini-2024-07-18
- gpt-4o-2024-08-06
- o1-mini-2024-09-12

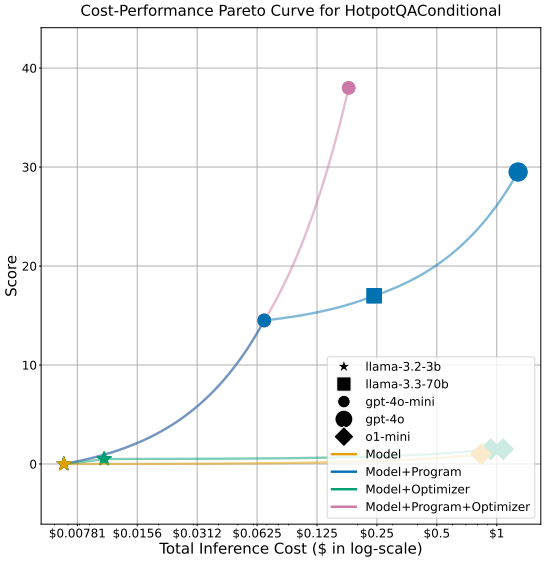


(a) Iris

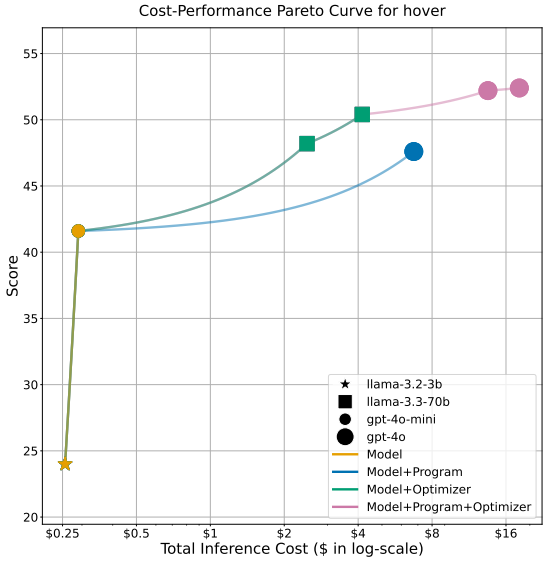
(b) HeartDisease



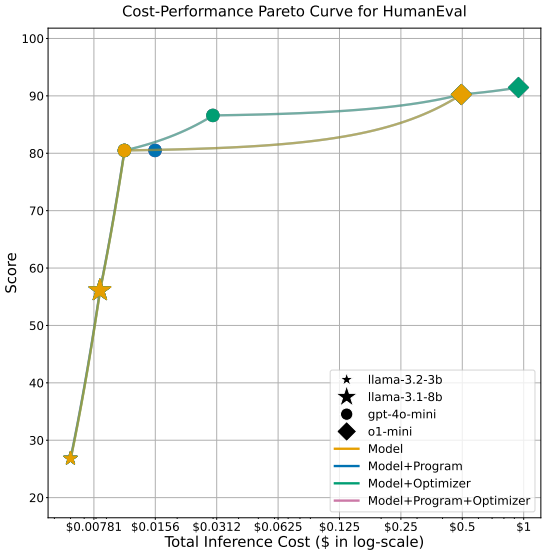
(c) HotpotQA



(d) HotpotQAConditional

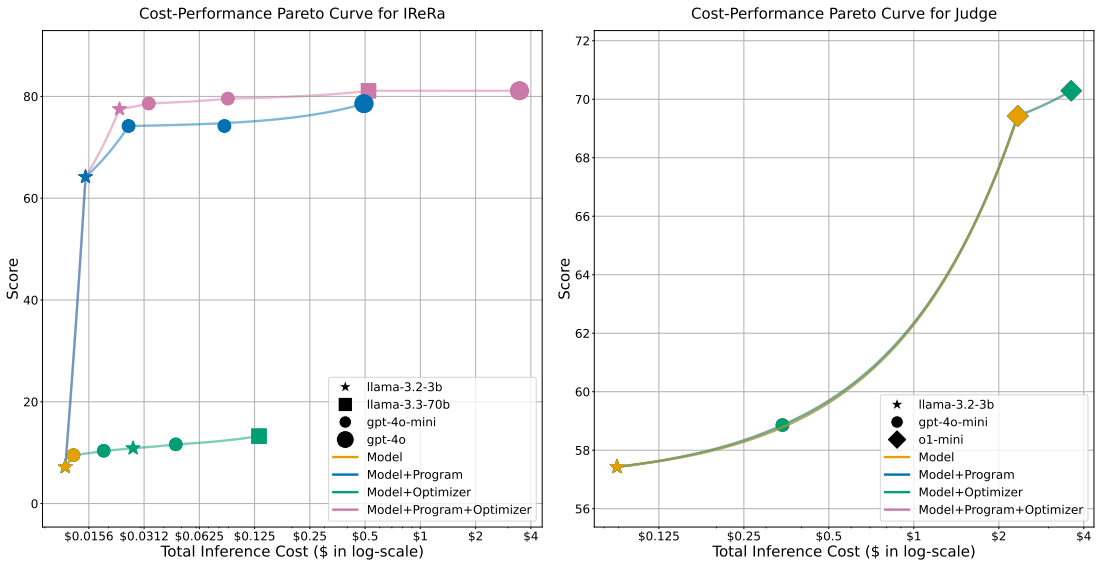


(e) HoVer



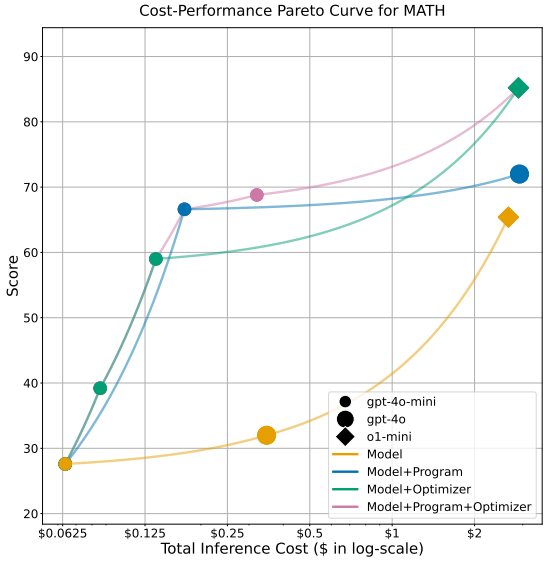
(f) HumanEval

Figure 9: Performance (Y) vs. Cost (X) Graph for different Benchmarks, Language Programs and Optimizers

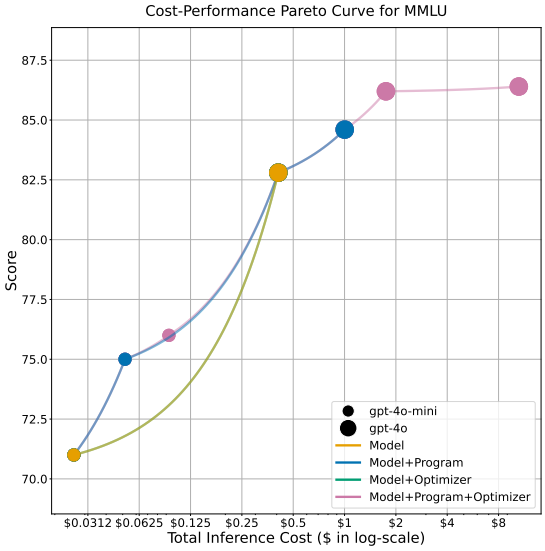


(a) IReRa

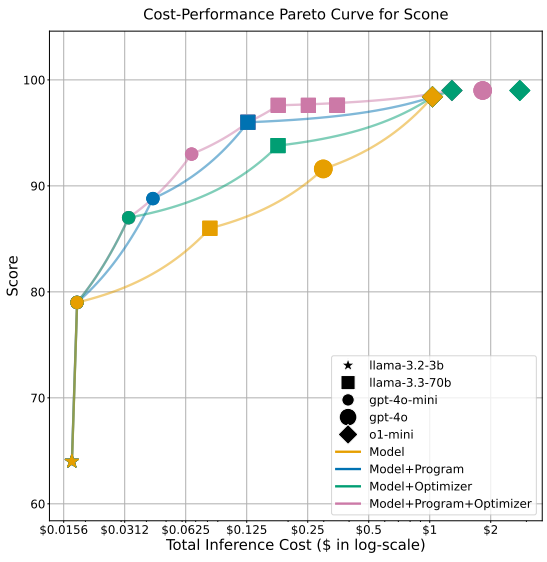
(b) Judge



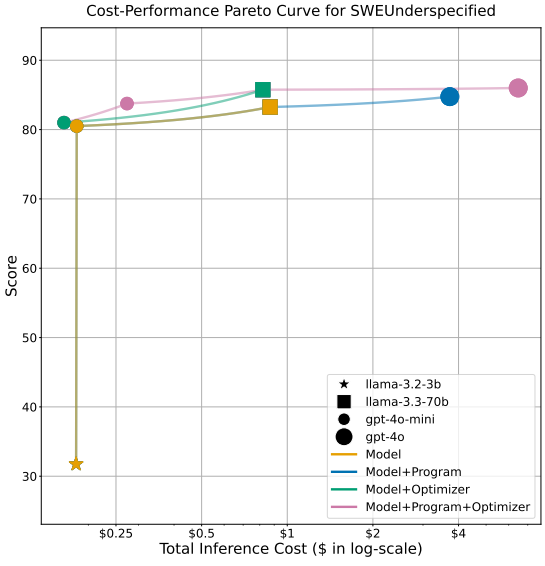
(c) MATH



(d) MMLU

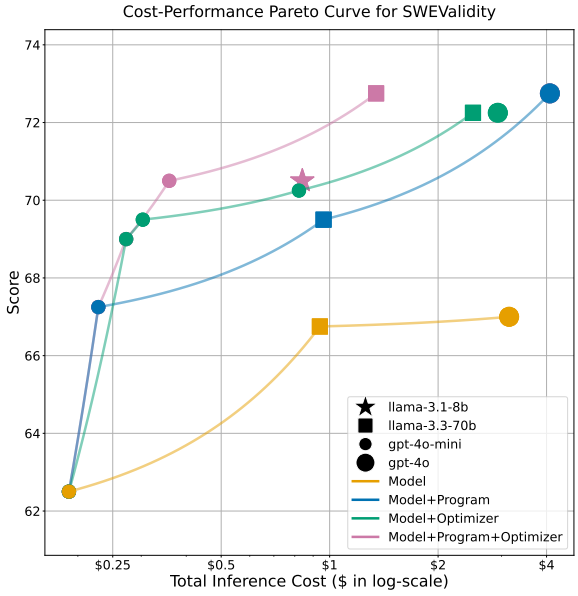


(e) Scone

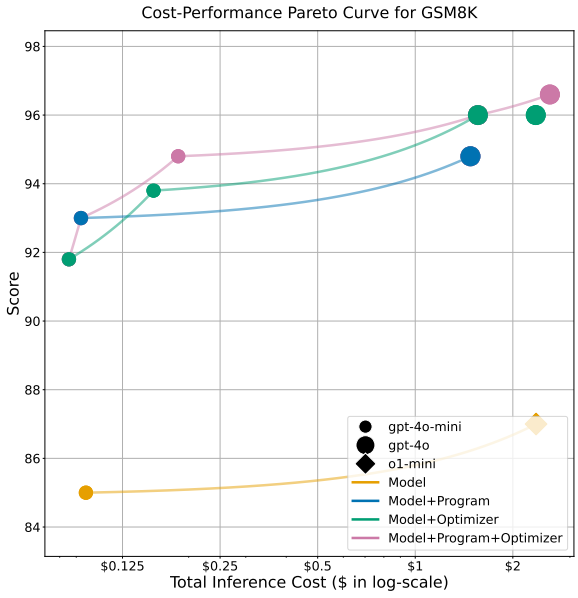


(f) SWEUnderspecified

Figure 10: Performance (Y) vs. Cost (X) Graph for different Benchmarks, Language Programs and Optimizers



(a) SWEValidity



(b) GSM8K

Figure 11: Performance (Y) vs. Cost (X) Graph for different Benchmarks, Language Programs and Optimizers

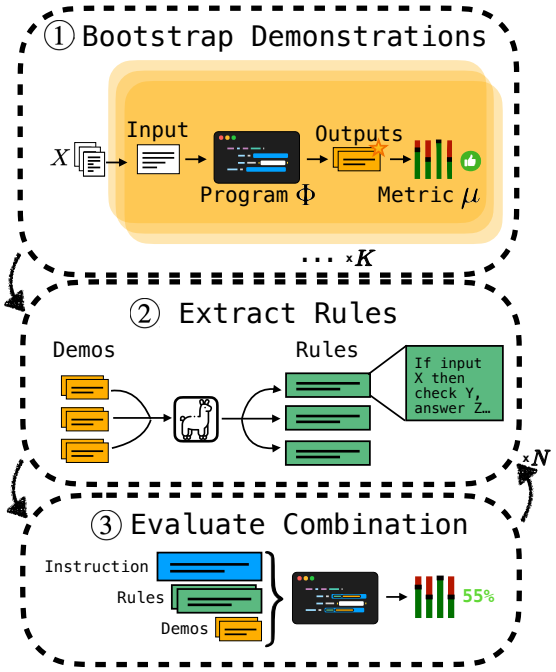


Figure 12: RuleInfer Optimizer Diagram. RuleInfer Bootstraps demonstrations, extracts rules, and finally finds useful combinations of the rules and demonstrations that work well on the validation set. Optimizer diagram styles from (Opsahl-Ong et al., 2024)