AutoPDL: Automatic Prompt Optimization for LLM Agents

Anonymous[1]

[1]Anonymous Institution

Abstract The performance of large language models (LLMs) depends on how they are prompted, with choices spanning both the high-level prompting pattern (e.g., Zero-Shot, CoT, ReAct, ReWOO) and the specific prompt content (instructions and few-shot demonstrations). Manually tuning this combination is tedious, error-prone, and non-transferable across LLMs or tasks. Therefore, this paper proposes AutoPDL, an automated approach to discover good LLM agent configurations. Our method frames this as a structured AutoML problem over a combinatorial space of agentic and non-agentic prompting patterns and demonstrations, using successive halving to efficiently navigate this space. We introduce a library implementing common prompting patterns using the PDL prompt programming language. AutoPDL solutions are human-readable, editable, and executable PDL programs that use this library. This approach also enables source-to-source optimization, allowing human-in-the-loop refinement and reuse. Evaluations across three tasks and six LLMs (ranging from 8B to 70B parameters) show consistent accuracy gains (9.5 ± 17.5 percentage points), up to 68.9pp, and reveal that selected prompting strategies vary across models and tasks.

1 Introduction

Large language models (LLMs) and LLM-based agents excel at a variety of tasks, including question answering, math word problems, and programming. The performance of an LLM depends heavily on how it is prompted, and there are a variety of popular prompting patterns. These include zero-shot or few-shot (Brown et al. 2020) prompting with chain-of-thought (CoT) (Wei et al. 2022), as well as agentic patterns such as ReAct (Yao et al. 2023) or ReWOO (Xu et al. 2023). However, given a dataset D_{test} with a loss function $\mathcal{L}$, *e.g.*, error rate, it is not clear which pattern will do best. Furthermore, besides the pattern A, performance also depends on the prompt p, including few-shot samples and instructions. The problem is thus to find a combination A_p^* of a pattern along with an optimized prompt that minimizes $\mathcal{L}$. This is usually done via manual prompt engineering, but that is tedious and has to be repeated if a new LLM comes along. Therefore, this paper explores how to find A_p^* using automated machine learning (AutoML). And for users to trust the result or to tweak it further, A_p^* itself should be easy to read and edit.

Agent frameworks, such as CrewAI (Moura 2023) or AutoGen (Wu et al. 2023), contain pre-built agent patterns with prompts optimized for proprietary frontier models and common tasks. Unfortunately, their prompts are deeply buried (Schluntz et al. 2024), making them hard to modify and adapt to non-frontier models or novel tasks. Moreover, the prompting pattern is fixed to a variation of ReAct, limiting flexibility in customizing the prompt structure. Prompt optimizers, such as DSPy (Khattab et al. 2023), optimize few-shot samples for in-context learning (ICL) and/or instructions in the prompt p. Unfortunately, they do not automatically select the prompting pattern A and do not return human-readable code.

We formulate the problem of finding a good pattern and corresponding prompt by defining and then exploring a combined search space. We were inspired by the AutoML literature on combined search spaces of machine-learning algorithms and their hyperparameters (Thornton et al. 2013), except that (i) instead of discrete or continuous hyperparameters, we explore textual ICL samples, instructions, and prompting patterns; (ii) instead of classification or regression tasks, we tackle generative tasks; and (iii) instead of model training or fine-tuning, we focus on in-context learning. We assume a dataset with a validation set D_{valid}, test set D_{test}, as well as an example bank D_{train} for few-shot samples. As usual, to avoid over-fitting, we assume these are disjoint from each other. The problem statement is to find $A_p^* = \underset{A_p \in \mathcal{A}_\mathcal{P}}{\operatorname{argmin}} \mathcal{L}(A_p, D_{\text{valid}})$, where:

- $A \in \mathcal{A} = \{\text{Zero-Shot, CoT, ReWOO, ReAct}\}$ is the prompting pattern, and
- $p = \langle n, d_{\text{train}}, \text{instr}\rangle \in \mathcal{P}$ is the prompt, comprising a number $n \leq |D_{\text{train}}|$ of few-shot samples, the actual few-shot samples $d_{\text{train}} \in (D_{\text{train}})^n$, and an instruction $\text{instr} \in \mathcal{I}$.

To avoid getting stuck in local minima while saving compute and finding a solution with a low loss, we explore the search space $\mathcal{A}_\mathcal{P}$ using successive halving (Jamieson et al. 2016). To make the initial search space $\mathcal{A}_\mathcal{P}$ user-interpretable, and the final solution A_p^* both human readable and executable, we express them in a YAML-based prompting language, PDL (Vaziri et al. 2024). PDL's structured format makes it easy to modify both the initial search space and the optimized program, and ensures the final solution remains directly executable. We introduce a library for PDL that implements each of the common prompting patterns in $\mathcal{A}$. The initial search space $\mathcal{A}_\mathcal{P}$ is a YAML file with various choice points for AutoML to decide. And the solution A_p^* is a custom-tailored PDL program optimized for the given task, as given by the dataset and loss function. The developer can read or even tweak either or both as desired.

We evaluate our optimizer on three tasks (question answering, math, and programming), using six LLMs sized between 8B and 70B parameters. We find that the optimizer often gives accuracy boosts in the 6–30% range, in some cases higher. Given the same task, different patterns $A \in \mathcal{A}$ do best for different models. Conversely, given the same model, different patterns do best for different tasks. Besides this variability in the chosen pattern A, our experiments also revealed variability in the optimized prompts $p = \langle n, d_{\text{train}}, \text{instr}\rangle$. We also found that when training data for a task is missing, data from a related but different dataset can help. Also, while most of our experiments use moderate-sized open models, we also show our optimized solutions can benefit frontier models.

This paper makes three primary contributions:

1. Jointly searching pattern and prompt: prior work in prompt optimization has not investigated searching *joint* search spaces, including agentic patterns.
2. No one size fits all: we find that different models sometimes have differing optimal prompt patterns for the *same* benchmark, suggesting that there is not one single optimal prompt pattern.
3. Source-to-source optimization: we propose the first source-to-source optimizer for LLM prompt programs, where both the initial search space and the final solution are prompt programs in the same language, making the final solution both human-readable and executable.

Overall, this paper shows how to apply AutoML to automatically discover agentic or non-agentic LLM prompts and patterns optimized for a given task.

2 Background

This paper uses PDL (Vaziri et al. 2024) as a representation for exploring the search space of programs. PDL programs are declarative and combine human readability with ease of execution. They represent the composition of calls to LLMs and tools, abstracting away the plumbing necessary for such compositions. The output of the optimizer is also a PDL program, rather than simple textual prompts, so it is fully executable and could be further refined by a developer.

Figure 1 shows a simple PDL program that uses a tool to answer a query. PDL is based on the premise that interactions with an LLM are mainly for the purpose of generating data. So, it allows users to specify the shape of data to be generated in a declarative way (in YAML), and is agnostic

```
1  text:
2  - role: tools
3    text: ${ tools }
4  - "Out of 1400 participants, 400 passed the test. What percentage is that?\n"
5  - def: actions
6    model: replicate/ibm-granite/granite-3.1-8b-instruct
7    parser: json
8    spec: [{ name: str, arguments: { expr: str }}]
9  - if: ${ actions[0].name == "calc" }
10   then:
11     lang: python
12     code: result = ${ actions[0].arguments.expr }
```

Figure 1: Basic example of a PDL program.

of any programming language. The first line of Figure 1 specifies that we are creating some text. The next block in the itemized list defines the tools prompt. Line 3 contains a use of variable *tools*, expressed as a Jinja expression. This variable is defined as the JSON Schema specification of a calculator tool (not shown in this figure, for the full program see Appendix A.1). Line 4 is the user query prompt. We do not specify the role explicitly as this is the default role for prompts. Lines 5 through 8 show a model call. In PDL, the background context is accumulated implicitly, so the output of all blocks executed so far will be passed to the LLM as a list of input messages. The result of the model call is assigned to the variable *actions* (line 5). The model to be called is specified on line 6 (PDL is based on LiteLLM,[1] so this is a LiteLLM model id). Finally, lines 7 and 8 say that we parse the output of the model as JSON and type-check it according to the type on line 8. Furthermore, when the inferencing server supports it, model calls with a schema use constrained decoding (Willard et al. 2023), enforcing syntactically correct JSON.[2]

On line 9, an if-statement checks whether the output of the LLM asks for the calculator tool. If so, we use a Python code block to compute the requested tool call (lines 11 and 12). When we execute this program using the PDL interpreter, we obtain all the model inputs, followed by the model output, and finally the output of the tool call. PDL has a rich set of control structures to allow writing a variety of prompting patterns, as well as functions to support libraries. For instance, Figure 3 shows a function call on line 4. In this paper, we consider the problem of automatically tuning prompts and choosing prompting patterns for a given dataset. The following section explains our approach in further detail.

3 AutoPDL Approach

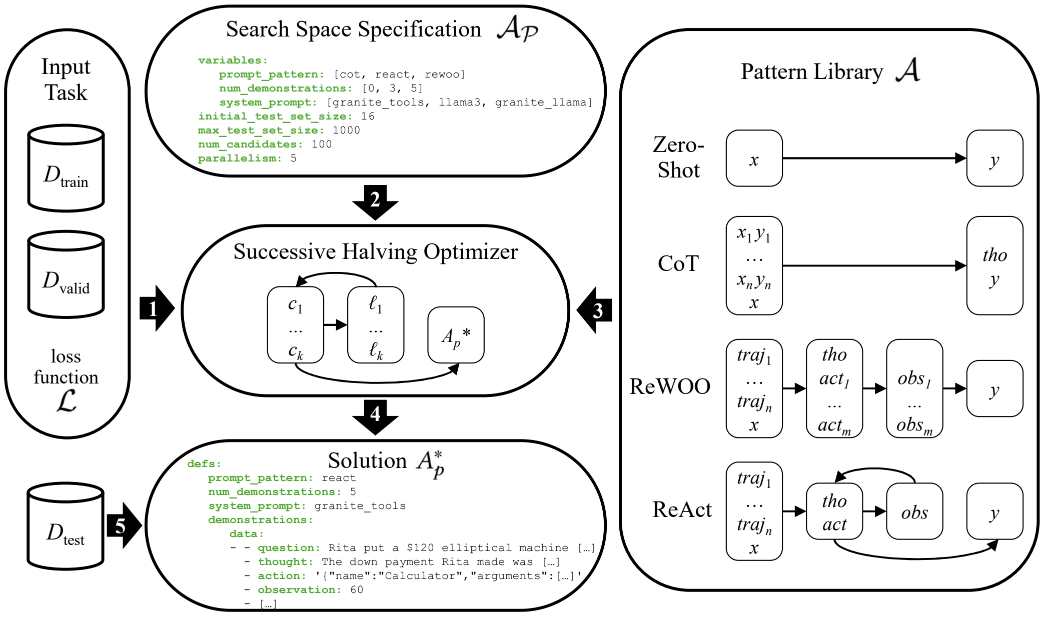
Figure 2 gives an overview of our approach. Referring to the numbers in the arrows:

(1) The input task is given by two disjoint datasets D_{train} and D_{valid} and a loss function $\mathcal{L}$. The datasets comprise $\langle x, y \rangle$ instances, where x is a question, y is the corresponding answer, and both are text strings. The loss function evaluates the quality of an answer y. (2) The search space specification $\mathcal{A}_{\mathcal{P}}$ is a YAML file with the optimization variables and their possible values, along with some hyperparameters. For example, `num_demonstrations: [0, 3, 5]` indicates that each candidate will have zero, three, or five ICL samples randomly drawn from D_{train}. In the case of zero demonstrations, this is equivalent to the zero-shot baseline. If zero is an option, we bias our candidate sampling to always include one zero-shot candidate, just in case that baseline turns out to be the best-performing configuration.

(3) The pattern library consists of four PDL functions. Zero-shot is a baseline that simply prompts the LLM with x and expects it to return y. CoT refers to chain-of-thought (Wei et al. 2022) with in-context learning (Brown et al. 2020): the input includes a few $x_i y_i$ pairs before the actual question x, and the output includes some reasoning thought *tho* before the actual answer y.

[1] https://github.com/BerriAI/litellm

[2] PDL additionally makes use of the heuristic `json-repair` package.



```
variables:
   prompt_pattern: [cot, react, rewoo]
   num_demonstrations: [0, 3, 5]
   system_prompt: [granite_tools, llama3, granite_llama]
initial_test_set_size: 16
max_test_set_size: 1000
num_candidates: 100
parallelism: 5
```

```
defs:
   prompt_pattern: react
   num_demonstrations: 5
   system_prompt: granite_tools
   demonstrations:
      data:
      - - question: Rita put a $120 elliptical machine […]
        - thought: The down payment Rita made was […]
        - action: '{"name":"Calculator","arguments":[…]'
        - observation: 60
        - […]
```

Figure 2: Overview of our approach.

ReWOO (Xu et al. 2023) refers to reasoning without observations. Here, the few-shot samples are trajectories $traj_i$, and the first LLM call generates reasoning thought tho and multiple actions act_i. The PDL code executes each of the actions as a tool call to obtain the corresponding observations obs_i. A final model call generates the answer y based on the observations. Finally, the ReAct pattern (Yao et al. 2023) starts with few-shot trajectory examples $traj_i$ and the question x, then enters a TAO (thought, action, observation) loop. In each loop iteration, the LLM generates tho and act, then the PDL code executes the action as a tool call, and feeds the tool output back as an observation obs. A special Finish action breaks out of the loop to return the answer y.

Once inputs (1), (2), and (3) are in place, the successive halving optimizer runs in a loop. The algorithm is given § A.2. It starts with a small subset $D_v \subset D_{\text{valid}}$ and many candidates $\mathcal{C} = \{c_1, \ldots, c_k\} \subseteq \mathcal{A}_{\mathcal{P}}$ sampled from the search space. Each iteration uses D_v to evaluate the corresponding losses $\ell_1, \ldots, \ell_k$. Then each iteration keeps the $\frac{k}{2}$ candidates with the smallest $\mathcal{L}$ while doubling the size of the validation subset D_v. (4) After the last iteration, the best remaining candidate is the solution A_p^*. This solution is a set of PDL definitions with concrete values for the optimization variables, e.g., in Figure 2, `num_demonstrations: 5` and `demonstrations: …` a list of ReAct trajectories. (5) This program can be used on the test set D_{test}. For instance, Figure 3 shows a call to the ReAct function that passes `${demonstrations}` from A_p^* as an argument.

```
text:
  - include: ../../tools.pdl
  - include: ../../ReAct.pdl
  - call: ${ react }
    def: ANSWER
    args:
      task: "Question: ${ question }"
      model: ${ model }
      tools: ${ tools }
      trajectories: ${ demonstrations }
  - "\nThe answer is ${ ANSWER.answer }"
```

Figure 3: Basic example of PDL program using the ReAct pattern.

4 Methodology

This section describes the datasets used, the tools available to agents, and our experimental setup, including how we construct agent trajectories that demonstrate tool use for each dataset.

4.1 Datasets

We selected datasets that are widely used in the literature, span diverse tools and domains, and are representative of tool categories frequently studied in prior work (*e.g.*, calculator, search, code execution). In our experiments, each dataset has three disjoint splits: D_{train} to sample few-shot samples from, D_{valid} to evaluate candidates during optimization, and D_{test} to evaluate the final chosen solution upon completion of optimization.

GSM8K. The Grade School Math (Cobbe et al. 2021) dataset consists of over eight thousand grade school math problems. We sample 1,000 problems without replacement from the train set to use as our D_{valid} set, and 1,000 from the test set to use as our D_{test}. This leaves a D_{train} of 6,449 problems. Each problem consists of a word problem x such as *"What is fifteen more than a quarter of 48?"*, a sequence *tho* of reasoning steps, and finally a plain numeric answer y following a special delimiter. For the CoT prompt pattern, we include these reasoning steps directly in the demonstrations.

GSM-Hard. Gao et al. (2023) introduce a derivative of GSM8K with variables randomly changed to large numbers. We split the single set into equally sized D_{valid} and D_{test} ($n = 594$), and use the GSM8K training set described above for D_{train} (*cross transfer*). Unfortunately, GSM-Hard had 132 samples where the ground truth was incorrect, and hence we excluded those samples.

FEVER. FEVER (Thorne et al. 2018) is a question-answering dataset structured around fact verification. The original dataset contained 185,445 claims that are true, false, or unverifiable, and associated with human annotated supporting, refuting, or neutral sentences and their Wikipedia article of origin. We follow the widely used derivative of this benchmark in BIG-bench (Srivastava et al. 2023), which reformulates it into a true-or-false task by removing unverifiable claims. We sample 1,000 claims from the train set as D_{valid} and 1,000 from the test set as D_{test}. This leaves a D_{train} of 5,676 claims. BIG-bench also does not include the supporting or refuting sentences, which we recover by joining on the original dataset, for use *e.g.*, as CoT demonstrations.

MBPP+. MBPP (Austin et al. 2021) is a dataset of mostly basic Python problems, with each example consisting of a natural language problem specification x for a self-contained function y, along with a single test case. Each problem has an extended set of test cases used for evaluation, which are not shown to the model. Liu et al. (2023) found that MBPP test cases are incomplete, allowing proposed solutions to pass as correct, despite not matching the problem specification. Therefore, our experiments are based on MBPP+, which contains a subset of the problems, but a more complete set of test cases for each problem. Our ReAct implementation follows Wang et al. (2024). We do not implement ReWOO as it is not reactive, *i.e.*, cannot include execution feedback.

4.2 Tools

Our prompt library represents actions, *i.e.*, tool calls, following the JSON tool calling schema (Abdelaziz et al. 2024). An action is represented as a JSON object with `name` and `arguments` mapping. For example, `{"action": "Calc", "arguments": {"expr": "48/4"}}` represents a call to a calculator with the expression to evaluate. The PDL functions implementing the patterns (see Figure 2) accept a list of tool definitions as an argument. Each element of that list is itself a `PDL` function. As both the agents and tools are implemented in PDL, the set of tools for a given task could itself be made a search space dimension, which we leave to future work.

Calculator. For math datasets, we give the agentic approaches access to the *Calc* tool. This tool evaluates a cleaned (*e.g.*, replacement of ^ with $**$) expression with SymPy, returning the result. In case of error, the function returns a warning that the expression was invalid, which may help the agent recover from invalid input.

Search. For fact verification, we provide access to the *Search* tool, which returns the summary of the first Wikipedia search result for the query. If no results are found, a hint to try again is returned, or if the title is too ambiguous, a list of possible disambiguations.

Execute. We implement a programming agent for the code generation task, which can execute arbitrary code surrounded in XML-style `<execute>` tags. This tool executes the code in a Python shell, which returns the result of the final expression. This allows the agent to test its proposed solution against the given test case before submitting its solution.

Finish. The most basic action is the *Finish* action, or `<solution>` tag for the coding agent. This ends the agent's trajectory and results in the agent returning the value as the solution.

4.3 Experimental Setup

We evaluate the efficacy of our approach by running an optimization process to completion for each model & dataset pair, and subsequently comparing the task accuracy. As a baseline, we evaluate each model in a zero-shot setting. This setting reflects the minimal effort approach by a user or developer, where they do not include any demonstrations with their query or task. As no model we investigate was specifically trained to create agentic trajectories in a zero-shot setting, it is not feasible to create a zero-shot setting under ReAct or ReWOO.

For the optimization process, we used an initial candidate set $\mathcal{C}$ of 100 candidates per experiment. We fixed the size of the initial validation subset $D_v \subset D_{\text{valid}}$ to 16, and the reduction factor η to 2. We define $\mathcal{L}(c_i, D_v) = -\text{Accuracy}(c_i, D_v)$ for a given candidate c_i. Additionally, a variable **Demonstrations** that holds the demonstrations or trajectories, defined as $D_t \subset D_{\text{train}}$ where $|D_t| =$ number of demonstrations, is sampled with replacement for each candidate. The number of possible values for **Demonstrations** is combinatorially large and depends on the dataset D_{train} used. Finally, upon completion of an optimization process, the optimal candidate is evaluated on D_{test}.

Constructing Trajectories. As we are also optimizing over agentic trajectories, we also need a set of trajectories to sample few-shot samples from. To achieve this, we create a basic agentic trajectory $traj_i$ for each training example $\langle x_i, y_i \rangle$, following a rule-based transformation. We design and apply a template to each dataset, which is relatively simple and easy to implement for other datasets (details are provided in § A.4). Prior work has introduced approaches to bootstrapping trajectories *e.g.*, in software engineering (Pan et al. 2024), tool use demonstrations (Li et al. 2025), and reasoning paths (Zelikman et al. 2022), which could be applied to this problem. While we acknowledge the shortcomings of manual template construction, we argue this approach has two strengths: it is generalizable in the sense that templates can be mixed and matched, and that the trajectories are directly based on the datasets used. Additionally, we wanted to work with commonly used datasets that cover a variety of tools and domains, rather than emerging datasets containing trajectories or tool use demonstrations.

Models. We aim to study models of various abilities, *e.g.*, natural language or code, various creators, and various sizes, ranging from single digit billions of parameters, up to the edge of feasibility on consumer hardware. We include six models available on inference service *IBM watsonx* in our study. We select two generalist natural language models, Llama 3.1 8B and Llama 3.1 70B, from the open-source and widely studied LLaMa family (Dubey et al. 2024). We further select three models from Mishra et al. (2024), which predate Dubey et al. (2024) by approximately 3 months. We select Granite 13B Instruct V2 as a generalist model, and Granite 20B and 34B Code Instruct as code models. The number of models we evaluate is limited by cost in $/token terms and execution time. By studying various models, we demonstrate the generalizability of our approach.

Alternative Setups. We evaluate two alternative experimental setups. First, to investigate low-resource scenarios, we examine whether performance on one dataset can be improved by using demonstrations D_{train} from a similar dataset, while optimizing w.r.t. D_{valid}. For this experiment, we investigate whether performance on GSM-Hard can be improved by using demonstrations from *GSM8K*, while optimizing w.r.t. GSM-Hard D_{valid}. Second, to explore saving optimization costs, we assess whether optimized prompt programs of one model can transfer well to a frontier model. The intuition is that while that might not be the best program for the frontier model, it might at least improve somewhat over the baseline. To this end, we evaluate the optimized PDL programs of LLaMa 3.1 70B on OpenAI's `gpt-4o-mini-2024-07-18`, for each dataset.

5 Results

This section describes the results of our empirical study to evaluate our AutoPDL approach and answer the following research questions:

RQ 1: To what extent does AutoPDL improve accuracy, and how much does the best solution vary by task and model?

RQ1 asks to what degree our AutoPDL approach can improve model performance over their zero-shot baseline across a variety of commonly used benchmarks. We also seek to identify trends, if any, in optimal configurations, *e.g.*, whether more few-shots is always better, or whether certain prompt patterns are particularly suited to certain problem domains.

RQ 2: Can AutoPDL make up for a missing few-shot example bank for a given task by reusing the example bank from a similar task?

RQ2 investigates whether optimizing on one dataset using demonstrations from another, related, dataset can result in higher performance than using no demonstrations (zero-shot). This RQ addresses a low-resource scenario in which a limited pool of demonstrations exists in one dataset, but a dataset from a similar domain has a large pool.

RQ 3: Do solutions found by AutoPDL improve performance on frontier models, even when optimized for open-source models?

It can be expensive to run optimization against commercial frontier-model APIs. RQ3 assesses whether optimized prompt programs are transferable to different (and likely stronger) models than those they were optimized with.

Table 1 reports the results of our optimization and evaluation procedure. Across models and datasets, we generally find some improvement over the zero-shot baseline with few-shot chain-of-thought, or agentic patterns ReAct or ReWOO.

FEVER. We observed the minimum improvement in Granite 3.1 8B, with a 0.7 percentage point (pp) improvement, and a maximal improvement of 68.9pp for Granite 13B Instruct V2. In terms of prompt pattern, CoT and ReWOO are represented, with CoT being the most frequent. ReAct was not the optimal for any of the models. Interestingly, the largest model (LLaMa 3.1 70B) benefited by 56.6pp from 3-shot CoT. FEVER runtimes are generally higher than the other benchmarks, likely due to the large number of tokens involved by including Wikipedia content.

GSM8K. The highest improvement recorded (12.7pp) was for LLaMa 3.1 70B using 5-shot CoT, while the minimum improvement of 1.3pp was in Granite 3.1 8B using 5-shot ReAct. ReWOO was not the optimal for any model. For Granite 20B Code and Granite 34B Code, no improvement over the zero-shot baseline was identified. This was somewhat surprising, as generally including even some few-shot samples improves performance in LLMs.

MBPP+. The majority of models benefited from execution feedback, as 4 out of 6 had ReAct as the optimal prompt pattern (ReWOO was excluded as described in § 4.3). The greatest improvement of 8pp was in Granite 13B Instruct V2, likely due to its poor programming performance as a generalist, non-code model. In contrast, the smaller LLaMa 3.1 8B model had high zero-shot performance of 61.2%, yet still improved by 6.2pp with ReAct. No improvement was observed for Granite 3.1 8B and LLaMa 3.1 70B.

Dataset	Model	Accuracy			Pattern	Runtime (HH:mm)
		Zero-Shot	Optimized	Delta		
FEVER	Granite 3.1 8B	78.3%	79.0%	+0.7pp	ReWOO (5 shot)	08:55
	Granite 13B Instruct V2	6.5%	75.4%	+68.9pp	ReWOO (3 shot)	08:12
	Granite 20B Code	39.7%	64.2%	+24.5pp	CoT (3 shot)	05:06
	Granite 34B Code	56.4%	65.6%	+9.2pp	CoT (3 shot)	03:47
	LLaMA 3.1 8B	68.5%	78.0%	+9.5pp	CoT (3 shot)	05:24
	LLaMA 3.1 70B	29.7%	86.3%	+56.6pp	CoT (3 shot)	04:57
GSM8K	Granite 3.1 8B	74.5%	75.8%	+1.3pp	ReAct (5 shot, Granite Tools)	01:29
	Granite 13B Instruct V2	23.2%	30.3%	+7.1pp	CoT (5 shot)	02:24
	Granite 20B Code	68.8%	68.8%	+0.0pp	Zero-Shot (Baseline)	05:06
	Granite 34B Code	72.3%	72.3%	+0.0pp	Zero-Shot (Baseline)	03:19
	LLaMA 3.1 8B	78.4%	84.8%	+6.4pp	CoT (3 shot)	03:24
	LLaMA 3.1 70B	82.1%	94.8%	+12.7pp	CoT (5 shot)	04:09
MBPP+	Granite 3.1 8B	68.8%	68.8%	+0.0pp	Zero-Shot (Baseline)	02:07
	Granite 13B Instruct V2	10.7%	18.8%	+8.0pp	ReAct (3 shot)	02:55
	Granite 20B Code	57.6%	60.7%	+3.1pp	ReAct (5 shot)	02:57
	Granite 34B Code	58.9%	59.8%	+0.9pp	ReAct (3 shot)	04:52
	LLaMA 3.1 8B	61.2%	67.4%	+6.2pp	ReAct (5 shot)	01:25
	LLaMA 3.1 70B	73.2%	73.2%	+0.0pp	Zero-Shot (Baseline)	01:38

Table 1: Model accuracies across datasets for baseline (zero-shot) and optimized versions.

Dataset	Model	Accuracy			Pattern	Runtime (HH:mm)
		Zero-Shot	Optimized	Delta		
GSM-Hard	Granite 3.1 8B	44.0%	44.0%	+0.0pp	Zero-Shot (Baseline)	04:57
	Granite 13B Instruct V2	4.4%	5.6%	+1.2pp	CoT (3 shot)	03:30
	Granite 20B Code	28.8%	28.8%	+0.0pp	Zero-Shot (Baseline)	08:26
	Granite 34B Code	27.9%	30.0%	+2.0pp	ReWOO (5 shot)	05:49
	LLaMA 3.1 8B	31.6%	32.3%	+0.7pp	ReWOO (5 shot)	04:44
	LLaMA 3.1 70B	46.6%	56.6%	+9.9pp	ReAct (5 shot, Granite LLaMa)	06:10

Table 2: Model accuracies on GSM-Hard for cross optimization experiment.

Missing Few-Shot Example Bank. We optimized the PDL program for GSM-Hard, using GSM8K demonstrations, and report results in Table 2. We found that in most cases, GSM8K demonstrations were at least not harmful for models on GSM-Hard, with up to 9.9pp improvement for LLaMA 3.1 70B using 5-shot ReAct (Granite LLaMa instructions). As the largest evaluated model, LLaMA 3.1 70B likely generalizes best and makes effective use of the calculator tool.

Dataset	Model	Accuracy			Pattern
		Zero-Shot	Optimized	Delta	
FEVER	GPT-4o-mini	83.7%	87.7%	+4.0pp	CoT (3 shot)
GSM-Hard	GPT-4o-mini	45.6%	54.9%	+9.3pp	ReAct (5 shot, Granite LLaMa)
GSM8K	GPT-4o-mini	77.8%	90.9%	+13.1pp	CoT (5 shot)
MBPP+	GPT-4o-mini	72.3%	72.3%	+0.0pp	Zero-Shot (Baseline)

Table 3: Model Accuracy for GPT-4o-mini cross experiment results

Commercial Frontier Model. To assess whether performance gains in one model can be achieved in another, we evaluate the optimized PDL programs of LLaMA 3.1 70B on OpenAI's `gpt-4o-mini-2024-07-18` and report results in Table 3. For all dataset/prompt pattern pairs which resulted in improvement for LLaMA 3.1 70B, we found a surprising improvement in GPT4o-mini of at least 4pp on FEVER using 3-shot CoT, 9.3pp on GSM-Hard (using GSM8K demonstrations) with 5-shot ReAct (Granite LLaMa instructions), and up to 13.1pp on GSM8K using 5-shot CoT. This suggests that optimizing for an open-source model can also benefit a closed-source model.

6 Related Work

The closest related work is on **prompt optimization**. APE starts with an LLM-generated set of candidate prompts, then performs rejection sampling based on evaluation on a subset of data (Zhou et al. 2023). Unlike our approach, APE does not optimize few-shot samples. CEDAR uses a demonstration pool, from which it retrieves few-shot examples at query time (Nashid et al. 2023). Unlike our approach, these few-shot samples are retrieved on a per-inference basis, not optimized ahead-of-time. DSPy optimizes instructions and few-shot samples for a chain of LLM calls (Khattab et al. 2023) (not just a single call like APE or CEDAR). Also, DSPy takes away control over the exact prompt from the programmer, which our approach preserves. Similarly to DSPy, TextGrad also optimizes a chain of LLM calls, by using LLMs to back-propagate modifications to instructions in prompts (Yuksekgonul et al. 2024). However, unlike our approach, neither of these optimize agentic patterns. EvoAgent optimizes the instructions of a population of agents via crossover, mutation, and selection (Yuan et al. 2024). It then forms an ensemble from the final, fittest, population. GPTSwarm represents each agent as a graph, then freezes intra-agent edges and optimizes the placement of additional inter-agent edges (Zhuge et al. 2024). Unlike our approach, neither EvoAgent nor GPTSwarm optimize the agentic pattern inside individual agents, nor do they optimize few-shot samples.

Another closely related field of study is **AutoML**. Auto-sklearn (Feurer et al. 2015) used Bayesian optimization to jointly perform both algorithm selection and hyperparameters of a scikit-learn pipeline (Buitinck et al. 2013). While different, we see some analogy between algorithms and agentic patterns, and between hyperparameters and few-shot samples. DAUB first evaluates many candidate models on a small amount of data, then successively reduces candidates and increases data to ultimately pick a strong model (Sabharwal et al. 2016). While both randomized search and Bayesian optimization are popular in AutoML, there are also more intricate approaches. For instance, TPOT uses genetic algorithms (Olson et al. 2016), and AlphaD3M uses Monte-Carlo tree search (Drori et al. 2018). We chose to start with a simpler technique that depends less on a well-behaved optimization space.

7 Conclusion

We present our AutoPDL approach for jointly optimizing prompting patterns and textual prompts for large language models, addressing the challenges associated with manual prompt engineering. By formulating the optimization as a discrete search over both agentic and non-agentic patterns, combined with instructions and few-shot samples, we leveraged successive halving to efficiently navigate this search space. Our evaluation across various datasets (FEVER, GSM8K, GSM-Hard, and MBPP+) and multiple models (LLaMA, Granite, GPT4o-mini) demonstrates substantial accuracy improvements, up to 68.9 percentage points, and affirms that no single prompting strategy universally outperforms others across tasks and models. Additionally, generating code in a YAML-based prompt programming language (PDL) makes it executable, easy to modify, and readable by humans, supporting practical adoption and adaptation.

References

Abdelaziz, I., Basu, K., Agarwal, M., Kumaravel, S., Stallone, M., Panda, R., Rizk, Y., Bhargav, G., Crouse, M., Gunasekara, C., Ikbal, S., Joshi, S., Karanam, H., Kumar, V., Munawar, A., Neelam, S., Raghu, D., Sharma, U., Soria, A. M., Sreedhar, D., Venkateswaran, P., Unuvar, M., Cox, D., Roukos, S., Lastras, L., and Kapanipathi, P. (2024). *Granite-Function Calling Model: Introducing Function Calling Abilities via Multi-task Learning of Granular Tasks.* URL: https://arxiv.org/abs/2407.00121 (cit. on p. 5).

Austin, J., Odena, A., Nye, M., Bosma, M., Michalewski, H., Dohan, D., Jiang, E., Cai, C., Terry, M., Le, Q., and Sutton, C. (15, 2021). *Program Synthesis with Large Language Models.* URL: http://arxiv.org/abs/2108.07732 (visited on 09/25/2022). Pre-published (cit. on p. 5).

Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D. M., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., and Amodei, D. (2020). *Language Models are Few-Shot Learners.* URL: https://arxiv.org/abs/2005.14165 (cit. on pp. 1, 3).

Buitinck, L., Louppe, G., Blondel, M., Pedregosa, F., Mueller, A., Grisel, O., Niculae, V., Prettenhofer, P., Gramfort, A., Grobler, J., Layton, R., VanderPlas, J., Joly, A., Holt, B., and Varoquaux, G. (2013). *API Design for Machine Learning Software: Experiences from the scikit-learn Project.* URL: https://arxiv.org/abs/1309.0238 (cit. on p. 9).

Cobbe, K., Kosaraju, V., Bavarian, M., Chen, M., Jun, H., Kaiser, L., Plappert, M., Tworek, J., Hilton, J., Nakano, R., Hesse, C., and Schulman, J. (17, 2021). *Training Verifiers to Solve Math Word Problems.* URL: http://arxiv.org/abs/2110.14168 (visited on 10/02/2022). Pre-published (cit. on p. 5).

Drori, I., Krishnamurthy, Y., Rampin, R., Lourenco, R. d. P., Ono, J. P., Cho, K., Silva, C., and Freire, J. (2018). “AlphaD3M: Machine Learning Pipeline Synthesis”. In: *Workshop on Automatic Machine Learning (AutoML)* (cit. on p. 9).

Dubey, A. et al. (15, 2024). *The Llama 3 Herd of Models.* URL: http://arxiv.org/abs/2407.21783. Pre-published (cit. on p. 6).

Feurer, M., Klein, A., Eggensperger, K., Springenberg, J., Blum, M., and Hutter, F. (2015). “Efficient and Robust Automated Machine Learning”. In: *Conference on Neural Information Processing Systems (NIPS)*, pp. 2962–2970 (cit. on p. 9).

Gao, L., Madaan, A., Zhou, S., Alon, U., Liu, P., Yang, Y., Callan, J., and Neubig, G. (2023). “PAL: Program-aided Language Models”. In: *International Conference on Machine Learning (ICML)*, pp. 10764–10799 (cit. on p. 5).

Jamieson, K. and Talwalkar, A. (2016). “Non-stochastic Best Arm Identification and Hyperparameter Optimization”. In: *Conference on Artificial Intelligence and Statistics (AISTATS)*, pp. 240–248 (cit. on pp. 2, 14).

Khattab, O., Singhvi, A., Maheshwari, P., Zhang, Z., Santhanam, K., Vardhamanan, S., Haq, S., Sharma, A., Joshi, T. T., Moazam, H., Miller, H., Zaharia, M., and Potts, C. (2023). *DSPy: Compiling Declarative Language Model Calls into Self-Improving Pipelines.* URL: https://arxiv.org/abs/2310.03714 (cit. on pp. 1, 9).

Li, C., Xue, M., Zhang, Z., Yang, J., Zhang, B., Wang, X., Yu, B., Hui, B., Lin, J., and Liu, D. (2025). *START: Self-taught Reasoner with Tools.* URL: https://arxiv.org/abs/2503.04625 (cit. on pp. 6, 15).

Liu, J., Xia, C. S., Wang, Y., and Zhang, L. (15, 2023). “Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation”. *Advances in Neural Information Processing Systems*, 36, pp. 21558–21572 (cit. on p. 5).

Mishra, M., Stallone, M., Zhang, G., Shen, Y., Prasad, A., Soria, A. M., Merler, M., Selvam, P., Surendran, S., Singh, S., Sethi, M., Dang, X.-H., Li, P., Wu, K.-L., Zawad, S., Coleman, A., White, M., Lewis, M., Pavuluri, R., Koyfman, Y., Lublinsky, B., de Bayser, M., Abdelaziz, I., Basu, K., Agarwal, M., Zhou, Y., Johnson, C., Goyal, A., Patel, H., Shah, Y., Zerfos, P., Ludwig, H., Munawar, A., Crouse, M., Kapanipathi, P., Salaria, S., Calio, B., Wen, S., Seelam, S., Belgodere, B., Fonseca, C., Singhee, A., Desai, N., Cox, D. D., Puri, R., and Panda, R. (7, 2024). *Granite Code Models: A Family of Open Foundation Models for Code Intelligence.* Version 1. URL: http://arxiv.org/abs/2405.04324. Pre-published (cit. on p. 6).

Moura, J. (2023). *CrewAI: Framework for orchestrating role-playing, autonomous AI agents.* URL: https://github.com/crewAIInc/crewAI (cit. on p. 1).

Nashid, N., Sintaha, M., and Mesbah, A. (2023). "Retrieval-Based Prompt Selection for Code-Related Few-Shot Learning". In: *International Conference on Software Engineering (ICSE)*, pp. 2450–2462 (cit. on p. 9).

Olson, R. S., Urbanowicz, R. J., Andrews, P. C., Lavender, N. A., Kidd, L. C., and Moore, J. H. (2016). "Automating Biomedical Data Science Through Tree-Based Pipeline Optimization". In: *European Conference on the Applications of Evolutionary Computation (EvoApplications)*, pp. 123–137 (cit. on p. 9).

Pan, J., Wang, X., Neubig, G., Jaitly, N., Ji, H., Suhr, A., and Zhang, Y. (30, 2024). *Training Software Engineering Agents and Verifiers with SWE-Gym.* URL: http://arxiv.org/abs/2412.21139 (visited on 03/28/2025). Pre-published (cit. on p. 6).

Sabharwal, A., Samulowitz, H., and Tesauro, G. (2016). "Selecting Near-Optimal Learners via Incremental Data Allocation". In: *Conference on Artificial Intelligence (AAAI)*, pp. 2007–2015 (cit. on p. 9).

Schluntz, E. and Zhang, B. (2024). *Building effective agents.* URL: https://www.anthropic.com/research/building-effective-agents (cit. on p. 1).

Srivastava, A. et al. (19, 2023). "Beyond the Imitation Game: Quantifying and Extrapolating the Capabilities of Language Models". *Transactions on Machine Learning Research* (cit. on p. 5).

Thorne, J., Vlachos, A., Christodoulopoulos, C., and Mittal, A. (18, 2018). *FEVER: A Large-Scale Dataset for Fact Extraction and VERification.* URL: http://arxiv.org/abs/1803.05355 (visited on 06/20/2024). Pre-published (cit. on p. 5).

Thornton, C., Hutter, F., Hoos, H. H., and Leyton-Brown, K. (2013). "Auto-WEKA: Combined Selection and Hyperparameter Optimization of Classification Algorithms". In: *Conference on Knowledge Discovery and Data Mining (KDD)*, pp. 847–855 (cit. on p. 1).

Vaziri, M., Mandel, L., Spiess, C., and Hirzel, M. (24, 2024). *PDL: A Declarative Prompt Programming Language.* URL: http://arxiv.org/abs/2410.19135. Pre-published (cit. on p. 2).

Wang, X., Wang, Z., Liu, J., Chen, Y., Yuan, L., Peng, H., and Ji, H. (12, 2024). *MINT: Evaluating LLMs in Multi-turn Interaction with Tools and Language Feedback.* URL: http://arxiv.org/abs/2309.10691 (visited on 08/27/2024). Pre-published (cit. on pp. 5, 14, 15).

Wei, J., Wang, X., Schuurmans, D., Bosma, M., Ichter, B., Xia, F., Chi, E., Le, Q., and Zhou, D. (2022). "Chain-of-Thought Prompting Elicits Reasoning in Large Language Models". In: *Advances in Neural Information Processing Systems (NeurIPS)*, pp. 24824–24837 (cit. on pp. 1, 3).

Willard, B. T. and Louf, R. (2023). *Efficient Guided Generation for Large Language Models.* URL: https://arxiv.org/abs/2307.09702 (cit. on p. 3).

Wu, Q., Bansal, G., Zhang, J., Wu, Y., Li, B., Zhu, E., Jiang, L., Zhang, X., Zhang, S., Liu, J., Awadallah, A. H., White, R. W., Burger, D., and Wang, C. (2023). *AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation.* URL: https://arxiv.org/abs/2308.08155 (cit. on p. 1).

Xu, B., Peng, Z., Lei, B., Mukherjee, S., and Xu, D. (2023). *ReWOO: Decoupling Reasoning from Observations for Efficient Augmented Language Models*. URL: https://arxiv.org/abs/2305.18323 (cit. on pp. 1, 4).

Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K. R., and Cao, Y. (2023). “ReAct: Synergizing Reasoning and Acting in Language Models”. In: *International Conference on Learning Representations (ICLR)* (cit. on pp. 1, 4).

Yuan, S., Song, K., Chen, J., Tan, X., Li, D., and Yang, D. (2024). *EvoAgent: Towards Automatic Multi-Agent Generation via Evolutionary Algorithms*. URL: https://arxiv.org/abs/2406.14228 (cit. on p. 9).

Yuksekgonul, M., Bianchi, F., Boen, J., Liu, S., Huang, Z., Guestrin, C., and Zou, J. (2024). *TextGrad: Automatic “Differentiation” via Text*. URL: http://arxiv.org/abs/2406.07496 (cit. on p. 9).

Zelikman, E., Wu, Y., Mu, J., and Goodman, N. D. (20, 2022). *STaR: Bootstrapping Reasoning With Reasoning*. URL: http://arxiv.org/abs/2203.14465 (visited on 06/26/2024). Pre-published (cit. on p. 6).

Zhou, Y., Muresanu, A. I., Han, Z., Paster, K., Pitis, S., Chan, H., and Ba, J. (2023). “Large Language Models are Human-Level Prompt Engineers”. In: *International Conference on Learning Representations (ICLR)* (cit. on p. 9).

Zhuge, M., Wang, W., Kirsch, L., Faccio, F., Khizbullin, D., and Schmidhuber, J. (2024). “GPTSwarm: Language Agents as Optimizable Graphs”. In: *International Conference on Machine Learning (ICML)* (cit. on p. 9).

A Supplemental Material 445

A.1 Tool Calling Code 446

```
description: tool use
defs:
  tools:
    data:
    - name: calc
      description: Calculator function
      arguments:
        expr:
          type: string
          description: Arithmetic expression to calculate
text:
- role: system
  text: You are Granite, developed by IBM. You are a helpful AI assistant
  with access to the following tools. When a tool is required to answer
  the user's query, respond with <|tool_call|> followed by a JSON list of
  tools used. If a tool does not exist in the provided list of tools,
  notify the user that you do not have the ability to fulfill the request.
  contribute: [context]
- role: tools
  text: ${ tools }
  contribute: [context]
- "Out of 1400 participants, 400 passed the test. What percentage is that?\n"
- def: actions
  model: replicate/ibm-granite/granite-3.1-8b-instruct
  parser: json
  spec: [{ name: str, arguments: { expr: str }}]
- "\n"
- if: ${ actions[0].name == "calc" }
  then:
    lang: python
    code: result = ${ actions[0].arguments.expr }
```

Figure 4: Basic example of a PDL program.

A.2 Optimization 447

Algorithm 1 Successive Halving for PDL Optimization

```
Require: Program candidate set $\mathcal{C}$, validation dataset $D_{\text{valid}}$, initial validation subset size $v_{\min}$, reduction factor $\eta > 1$
1: $v \longleftarrow v_{\min}$
2: while $|\mathcal{C}| > 1$ and $v \leq |D_{\text{valid}}|$ do
3:     for each candidate $c_i \in \mathcal{C}$ do
4:         Compute loss $\ell_i \longleftarrow \mathcal{L}(c_i, D_v)$, where $D_v \subset D_{\text{valid}}$ and $|D_v| = v$
5:     end for
6:     $\mathcal{C} \longleftarrow$ top $\lceil |\mathcal{C}|/\eta \rceil$ candidates with lowest loss
7:     $v \longleftarrow 2 \cdot v$
8: end while
9: return Candidate in $\mathcal{C}$ with lowest loss
```

Figure 5: Illustration of the Successive Halving algorithm used to optimize the PDL program by pruning poor candidates on progressively larger validation subsets.

Figure 5 describes our optimization algorithm, based on successive halving (Jamieson et al. 2016). The algorithm accepts a candidate set sampled from possible configurations and demonstrations, a validation dataset to optimize against, an initial validation subset size, and a reduction factor. AutoPDL allows the user to specify these options in a YAML configuration file, and ultimately saves its result as a PDL program. This source-to-source transformation enables the user to modify both the search space, and the resulting optimized PDL program, allowing further modification and execution.

A.3 Search Space

The search space is the Cartesian product of the following discrete variables, each taking one value per candidate:

(1) $A \in \mathcal{A} = \{\text{Zero-Shot, CoT, ReWOO, ReAct}\}$, *i.e.*, the overall prompting pattern to apply.

(2) **Number of demonstrations** $\in$ 0, 3, 5. We selected these options as a representative sweep across no supervision, moderate few-shot use, and an upper-end case (in terms of token window). We limited the search space to three options to avoid combinatorial explosion and limit experiment cost.

(3) If A = ReAct, **System prompt** $\in$ `Granite Tools`, `LLaMa 3`, `Granite LLaMa`. As the system prompt instructs the model how to format tool calls, it only has an effect on benchmarks with JSON tool calling (FEVER, GSM8K, and GSM-Hard) for candidates with the ReAct prompt pattern. We note that only for MBPP+, ReWOO is not included as a prompt pattern, and that we always include two trajectories displaying iterative refinement, *i.e.*, an example of an solution failing the example test case, followed by a passing solution, in line with Wang et al. (2024). This effectively increases the number of trajectories to $|traj| + 2$.

A.4 Agent Trajectory Construction

To achieve this, we create a basic agentic trajectory $traj_i$ for each training example $\langle x_i, y_i \rangle$, following a rule-based transformation outlined below.

GSM8K. To demonstrate tool use in ReAct, we instead derive a trajectory *traj* as follows. We exploit the fact that there is at most one expression per reasoning step, by iterating through the steps. At each step, we append a 'thought' to the trajectory, consisting of the text leading up to the math expression, concatenated with a reflection 'I need to calculate'. We append a calculator tool call with the expression, and an 'observation', *i.e.*, the result of the expression. Finally, we append a thought 'The answer is ...', containing the ground truth answer, followed by the *finish* action with the answer. We follow the same procedure to create ReWOO trajectories, except we use slightly different wording, *e.g.*, 'Calculate xyz' in place of 'I need to calculate xyz', and omit the final thought and action. Additionally, we use string substitution to replace any assumed expression results in the trajectory with the corresponding variable.

FEVER. To produce agent trajectories, we iterate over each article associated with a claim, append a thought 'I need to search for ...', followed by the action, an observation containing the article summary, and finally a thought containing all the relevant sentences associated with that article for that claim, which we repeat for each article associated with a claim. This procedure is not ideal as there is no inherent order to the articles or sentences, even though there may be a natural ordering following the annotator's Wikipedia navigation. Finally, we append a thought 'The claim is true/false' and the finish action, both with the ground truth answer. For chain-of-thought, we perform the same procedure except we only include the concatenated evidence sentences, as there is no tool use.

MBPP+. To generate sample agent trajectories from the training set, we follow the agent pattern (without feedback) in-context examples by Wang et al. (2024), which consists of the problem x, a thought such as "The intersection is elements that are in both lists", an execute action that contains proposed code *and* an assertion calling the proposed method with the test case input from the prompt and comparing its output. This is then followed by an observation containing the execution result, *i.e.*, either "`[Executed Successfully with No Output]`" or a stack traceback. This allows the agent to iterate on solutions (up to five times in our implementation). We use the full MBPP train set of 374 problems as D_{train}, and split the MBPP+ dataset into D_{valid} and D_{test} based on problem id membership in MBPP, leaving 39 and 224 validation and test problems respectively.

To generate synthetic trajectories from the training set, we start with the natural language specification and single test case (the prompt), append the thought "I should run a solution on the test case before proposing a solution.", followed by the ground truth solution and substitute in the prompt test case following the pattern `[solution]res = ...; assert res == ..., "Expected ... but got ".format(res)`. Subsequently, we append the observation "`[Executed Successfully with No Output]`", the thought "`There is no more AssertionError. I can now submit the solution.`", and finally the solution action with the ground truth solution. This naive approach allows us to provide demonstration trajectories, albeit simplistic ones that assume the first solution is correct. Sampling a reflection or thought from a strong model may be beneficial (Li et al. 2025), but we restrict our trajectories to rule based transformations. As ReWOO is not reactive, *i.e.*, without execution feedback, it does not make sense for MBPP. Hence, we exclude it from our experiments.

A.5 Results Plot

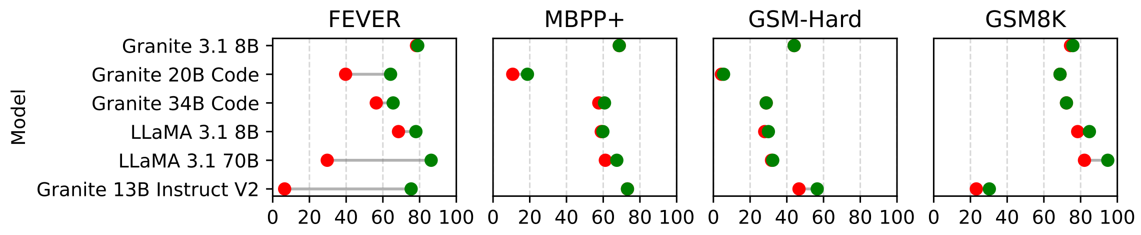


Figure 6: Comparison of optimized prompt program performance across models and datasets. Each barbell shows the accuracy improvement, if any, over the zero-shot baseline.

In Figure 6, we visualize the results from Table 1 and Table 2.

A.6 Accuracy *vs.* iterations

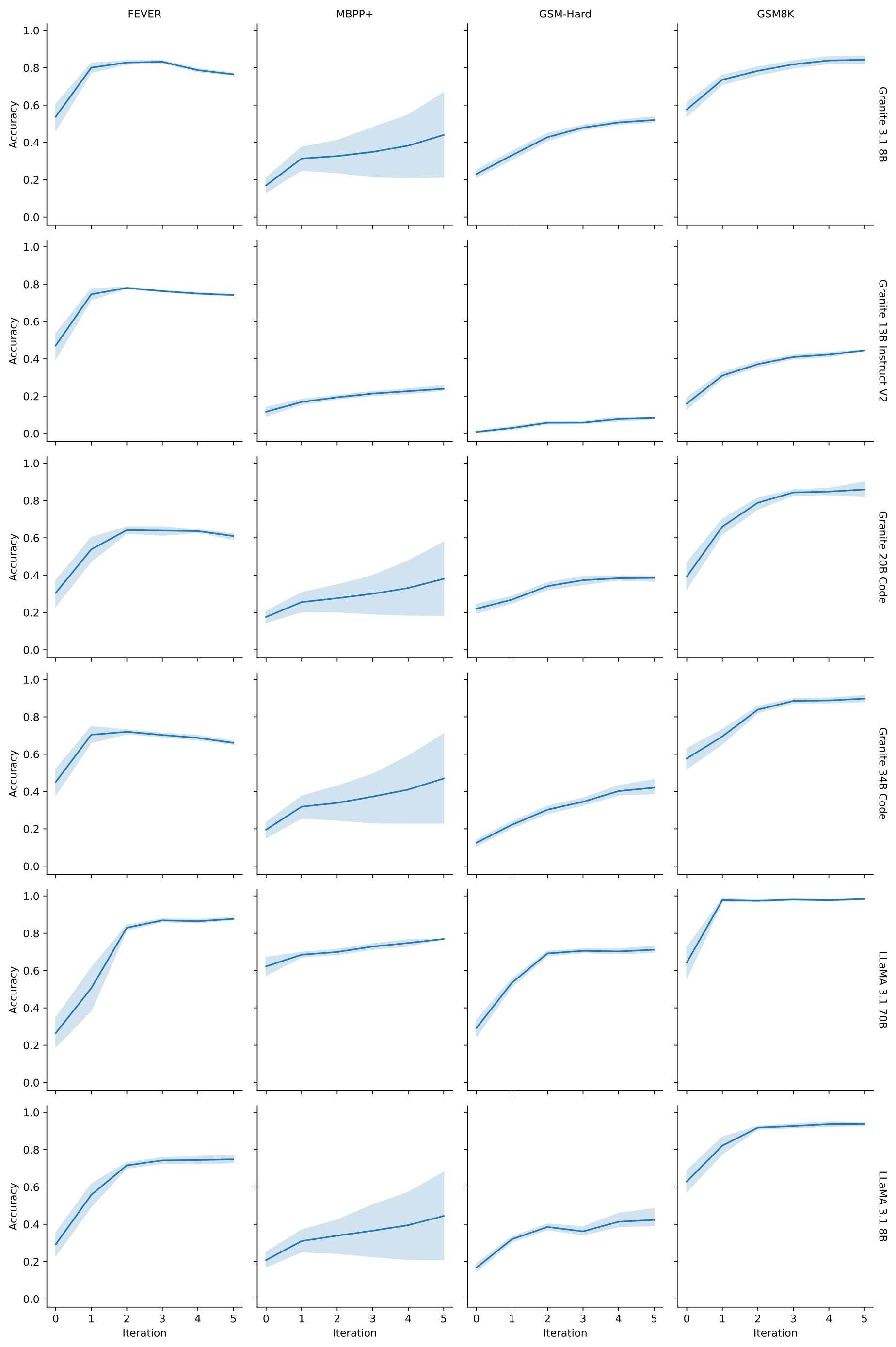


Figure 7: Accuracy *vs.* iterations with 95% CI.

In Figure 7, we visualize the accuracy across candidates versus the iterations of the optimization process, including a 95% confidence interval depicting the spread in accuracy across candidates. As the iterations increase, the number of candidates decreases, while the size of the validation set D_v increases.