TroVE: Inducing Verifiable and Efficient Toolboxes for Solving Programmatic Tasks

Zora Zhiruo Wang[1] **Graham Neubig**[1] **Daniel Fried**[1]

Abstract

Language models (LMs) can solve tasks such as answering questions about tables or images by writing programs. However, using primitive functions often leads to verbose and error-prone programs, and higher-level functions require expert design. To enable better solutions without human labor, we ask code LMs to curate reusable high-level functions, and use them to write solutions. We present TroVE, a *training-free* method of inducing a *verifiable and efficient toolbox* of functions, by generating via using, growing, and periodically trimming the toolbox. On 11 datasets from math, table question answering, and image reasoning tasks, TroVE consistently yields *simpler solutions* with *higher accuracy* than baselines using CodeLLaMa and previous methods using GPT, while using 79-98% *smaller* toolboxes. TroVE further enables 31% *faster* and 13% *more accurate human verification* than baselines. With the same pipeline, it creates *diverse functions* for varied tasks and datasets, providing insights into their individual characteristics. Code and data are available at https://github.com/zorazrw/trove.

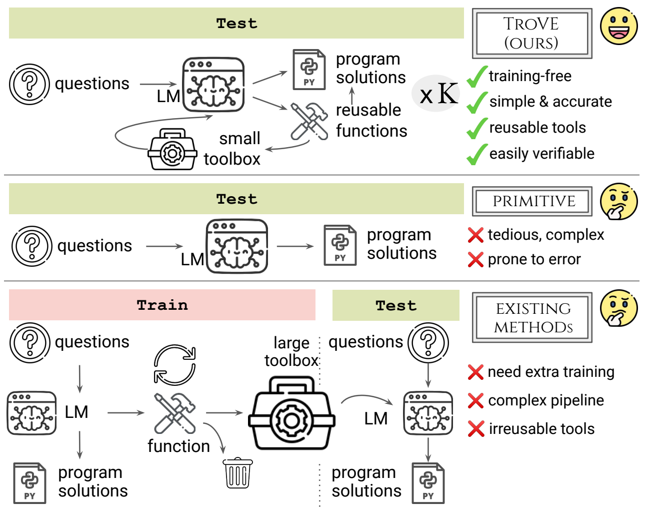


Figure 1. Our TroVE induces reusable functions to produce better program solutions than the Primitive setting, without training, supervision, or iterations required by existing methods.

1. Introduction

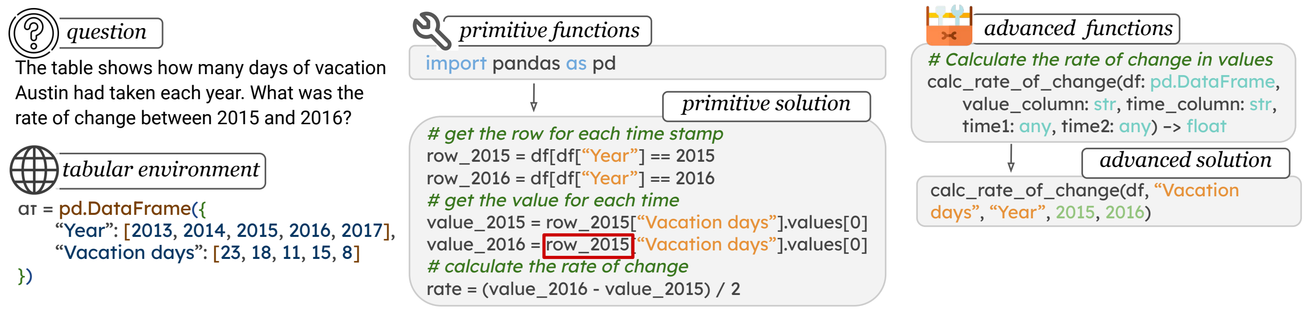
Generating code from natural language commands has long been a method of choice for solving tasks such as question answering (Zettlemoyer & Collins, 2007; Liang et al., 2011) or agent navigation (Artzi & Zettlemoyer, 2013). Recently, language models (LMs) have been used to write programs in general-purpose languages such as Python, further expanding code generation's applicability (Yin & Neubig, 2017; Li et al., 2022b; Cheng et al., 2023). These programs generally rely on multiple function calls to Python built-in functions or libraries such as `pandas`, as in the example in Figure 2. However, in many cases relying on these primitive functions can lead to programs that are tedious, complex, and error-prone (Cai et al., 2023; Majumder et al., 2023). These programs can also be difficult to verify, as the users may need to check every operation as well as their combinations and interactions across the entire program (e.g., the primitive solution in Figure 2).

When human developers are faced with an analogous situation, they *create application-specific functions*, i.e. tools, composing primitive functions that are often used together. For instance, in Figure 2 (right), the `calc_rate_of_change` tool is easier to understand and less error-prone to use, hence enabling a more concise and accurate solution.

A few recent works have attempted to use LMs to *automatically induce tools* in a similar way (existing methods in Figure 1). However, existing methods tend to either induce large and ponderous toolboxes and/or have added complexity and data requirements. For instance, Qian et al. (2023) propose CREATOR to disentangle planning (tool making)

[1]Language Technologies Institute, Carnegie Mellon University. Correspondence to: Zora Zhiruo Wang <zhiruow@cs.cmu.edu>, Graham Neubig <gneubig@cs.cmu.edu>, Daniel Fried <dfried@cs.cmu.edu>.

Proceedings of the 41st International Conference on Machine Learning, Vienna, Austria. PMLR 235, 2024. Copyright 2024 by the author(s).



The table shows how many days of vacation Austin had taken each year. What was the rate of change between 2015 and 2016?

```
df = pd.DataFrame({
    "Year": [2013, 2014, 2015, 2016, 2017],
    "Vacation days": [23, 18, 11, 15, 8]
})
```

```
import pandas as pd
```

```
# get the row for each time stamp
row_2015 = df[df["Year"] == 2015
row_2016 = df[df["Year"] == 2016
# get the value for each time
value_2015 = row_2015["Vacation days"].values[0]
value_2016 = row_2015["Vacation days"].values[0]
# calculate the rate of change
rate = (value_2016 - value_2015) / 2
```

```
# Calculate the rate of change in values
calc_rate_of_change(df: pd.DataFrame,
    value_column: str, time_column: str,
    time1: any, time2: any) -> float
```

```
calc_rate_of_change(df, "Vacation
days", "Year", 2015, 2016)
```

Figure 2. Function design affect solutions. Using primitive functions results in complex, error-prone solutions (middle), while using abstract functions leads to more concise and accurate solutions (right).

from execution, but at the cost of producing hundreds or more tools that are challenging for models to reuse or humans to verify. Further, compared to the standard setting (PRIMITIVE in Figure 1) that only needs test data to produce their solutions, Wang et al. (2023a) propose life-long learning via an automatic curriculum, equipped with iterative self-verification. Cai et al. (2023); Yuan et al. (2023) require additional training and validation datasets to create tools ahead to be used by solutions, plus auxiliary modules such as self-verification or toolset deduplication.

In this paper, we propose TROVE, a training-free method of inducing a verifiable and efficient function toolbox (§3), and using these functions to write solutions. TROVE features three major components: using and growing a toolbox maintained over time, execution agreement-based selection, and periodic toolbox trimming. Notably, our method *requires zero additional training or supervision*, and selects programs only by their inter-*execution agreement*. Given a stream of questions, TROVE produces their solutions, along with inducing a handy toolbox to solve the questions.

We experiment on 11 datasets from three real-world tasks (§4): (1) mathematical problems with the MATH dataset, (2) table question answering on TabMWP, WTQ, and HiTab, and (3) compositional visual reasoning with GQA. Compared to baselines using CODELLAMA as well as previous state-of-the-art methods using GPT, our TROVE consistently produces solutions with higher accuracy and reduced complexity, while maintaining a significantly smaller, efficient function library (§5). We further show that via human study (§6), verifying solutions generated by TROVE is 31% faster and 13% more accurate, than solutions generated by baseline methods.

2. Problem Statement & Baseline Methods

We formally define the task of problem solving via programs (§2.1) and introduce corresponding baseline methods (§2.2).

2.1. Problem Solving via Programs

We focus on problems that are describable in natural language (NL) and solvable using programs. Concretely, given an example x, i.e., an NL query q grounded on an environment e, we ask a language model LM to write a programmatic solution s by composing multiple functions $F = \{f_1, \cdots, f_n\}$. This process is denoted as $P_{LM}(q, e, f)$. f and hence s can be executed on e to obtain the final answer. We use Python as the programming language for our experiments, since it is general-purpose thus allowing flexible functions to be created for most tasks. Each solution s is a Python program, and each function f is a Python function.

It is crucial to note that the difficulty of solution generation is greatly affected by the usability of the functions in the set F. Relatively speaking, we can categorize functions into two types: (i) *primitive functions*, which only support basic operations such as Python standard libraries, and (ii) *composed functions*, which perform more complex operations by composing multiple basic operations.

Primitive Functions Primitive functions are atomic, low-level operations on the task environment, such as subtraction `-` and division `/` in the *primitive solution* in Figure 2 (middle). They are often easy to obtain without expert knowledge about the application domain. Yet often, to solve a question as in Figure 2 (left), it requires complex compositions of numerous functions and hence becomes extremely error-prone. In this example, due only to a tiny $\boxed{\text{mistake}}$, which calls the wrong `row_2015` instead of `row_2016`, the output goes wrong despite all the other steps being correct.

Composed Functions Composed functions, such as the `calc_rate_of_change` in Figure 2 (right), combine various primitive functions. Conceptually, they are *easier to use*, as they align better with the actions asked in the question. In this example, it is easier to associate "What is the rate of change ..." with a function named `calc_rate_of_change`, compared to certain compositions of data slicing `df[·]`, check equality `==`, get value `cell.values[index]`, subtraction `-`, and division `/`. Practically, composed functions are *less error-prone*, by only showing an API interface and abstracting intricate details inside.

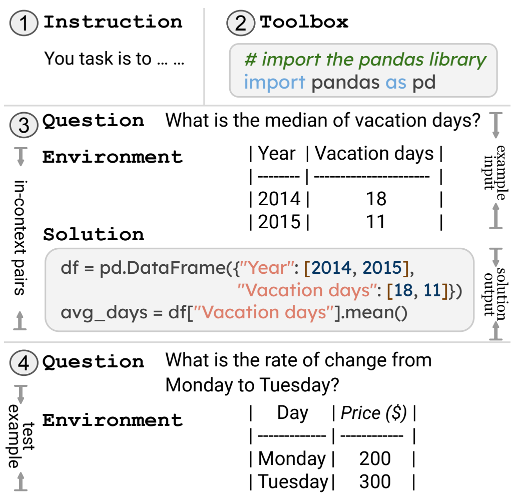
Composed functions are often crafted by human experts, by recognizing and generalizing shared functionality. However, this process is costly and hardly scalable to new domains. Therefore, it is crucial to create such functions automatically, to enhance problem solving while saving human labor.

2.2. Baseline Methods

We introduce two baselines that generate programs using primitive and composed functions. All methods, including our main method later in §3, operate solely by prompting an LM, and do not update the parameters of the LM itself.

Using Primitive Functions Our first baseline, PRIMITIVE, asks models to generate programs using primitive functions, which is the de facto approach for program-aided problem solving without tool induction (Cheng et al., 2023; Gao et al., 2023b).

As exemplified in Figure 3, our prompt inputs consist of four components: (1) an NL instruction specifying the task, (2) the function signature and textual docstring of primitive functions,[1] (3) c (example, solution) pairs to demonstrate the usage of primitive functions, and lastly (4) the query and environment (q, e) of the current testing example. We collect the code snippets from model responses as s and execute on e to obtain the final result and evaluate it. Please find more detailed examples in §A.



```
# import the pandas library
import pandas as pd
```

```
| Year  | Vacation days |
| -------- | -------------------- |
| 2014 |        18         |
| 2015 |        11         |
```

```
df = pd.DataFrame({"Year": [2014, 2015],
                   "Vacation days": [18, 11]})
avg_days = df["Vacation days"].mean()
```

```
|    Day     | Price ($) |
| ------------ | ----------- |
| Monday |    200    |
| Tuesday|    300    |
```

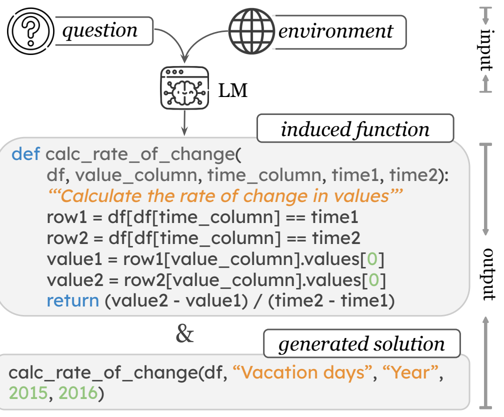
Figure 3. Example input prompt on the tabular environment. We textualize tables by markdown format in the prompt, and ask LMs to parse tables into pandas DataFrame in program solutions.

[1]We do not show the built-in functions since LMs have already been trained on them extensively. We only have 1-2 primitives in addition (§4.3, §4.4), so they can easily fit into the prompt.

Abstracting Functions Example Wise Our second baseline INSTANCE asks models to create tools for each example, and use them in the solution for that example. Qian et al. (2023) found this two-step process helpful for model reasoning by disentangling tool abstraction (planning) from example-wise decision (execution).

In preliminary studies, we compared generating functions and solutions in two sequential responses or prompting the model to generate both in one response. They performed comparably, so we adopted the latter approach since it is simpler. Concretely, given a set of primitive functions P, for each example (q, e), the model needs to generate the functions F by composing operations in P, as well as the solution s that uses F. Grounding on Figure 4, F is the *induced function* snippet, and s is the *generated solution*.

We include the same four prompt components used in PRIMITIVE, but alter the instruction and example outputs according to the $F\&s$ generation format. We query each example independently to allow example-wise function induction. However, it is not possible to share these functions across examples, even if they have similar functions.



```
def calc_rate_of_change(
    df, value_column, time_column, time1, time2):
    """Calculate the rate of change in values"""
    row1 = df[df[time_column] == time1
    row2 = df[df[time_column] == time2
    value1 = row1[value_column].values[0]
    value2 = row2[value_column].values[0]
    return (value2 - value1) / (time2 - time1)
```

```
calc_rate_of_change(df, "Vacation days", "Year",
2015, 2016)
```

Figure 4. An illustration of a model inducing a composed function and using it to generate the solution at the same time.

3. Inducing Reusable Functions On-the-fly

Now, we present our main method TroVE that: uses and grows a toolbox iteratively over time (§3.1), selects optimal outputs based on execution agreement (§3.2), and periodically trims low-utility functions from the toolbox (§3.3).

3.1. Using and Growing the Toolbox

To learn functions that can be reused across examples, we maintain a shared function library F over time. To keep our method running in linear time, we process all examples *online* in a streaming fashion. We start with $F = \varnothing$ and gradually add or remove functions from it. Figure 5 illustrates the processing of example x^t at time t.

First, we define 3 modes with which LMs can interact with the current toolbox F^t, represented as the docstrings and signature of all tool functions in it. ① IMPORT : In this mode, the LM is instructed to import defined functions from the toolbox and apply them to solve x^t. ② CREATE : In this mode, the LM is instructed to create a new function f^t_{new} and add it to the toolbox. ③ SKIP : In this mode, the LM is instructed to only use primitive functions just as in the PRIMITIVE setting.

For each example, for each of the three modes we sample from the LM to generate K responses, for a total of $3K$ responses per example.[2] Each response contains a solution s and function f as shown in Figure 4, except for the SKIP mode that only contains s. We denote these candidate responses as (f^{ti}_m, s^{ti}_m), where $m \in$ {IMPORT, CREATE, SKIP} and $i \in \{1, \cdots, K\}$.

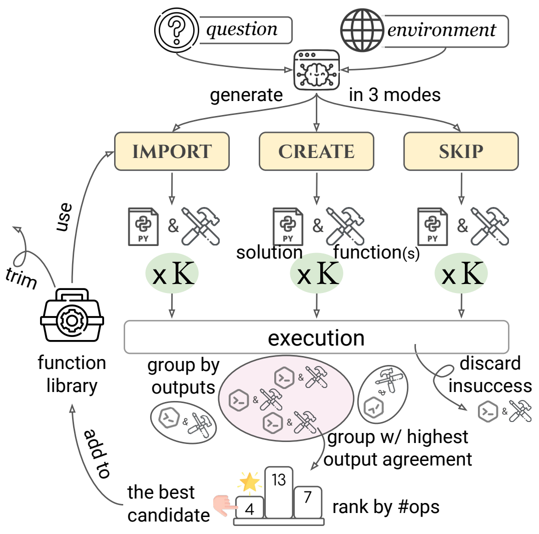


Figure 5. TROVE illustration. Top: generate solutions while using and growing the toolbox. Bottom: select the best response by execution agreement. Left: periodically trim low-utility functions.

3.2. Agreement-Based Selection

From the $3K$ sampled (f, s) pairs, we select one to use via self-consistency (Shi et al., 2022; Li et al., 2022a; Wang et al., 2023b) and solution complexity. We first execute all solutions and remove those that cannot execute successfully. Next, we select an answer based on the consistency of the execution outputs, keeping the solutions that produce the most frequently occurring answer. Next, if there are multiple solutions with the same frequency, we rank solutions by the number of operations they require, and pick the one with the least operations (preferring simple solutions). Finally, if multiple candidates remain, we break the tie by arbitrarily choosing the one that appears first in the model prediction list. We add the function in this best response to the toolbox, and adopt its solution as the solution for the current example.

[2]In preliminary studies, we tried to let models choose one mode and only generate in that mode, but results degraded significantly.

3.3. Periodic Toolbox Trimming

Not all the functions induced by models are highly reusable. Hence, we also propose to periodically trim the toolbox to effectively remove low-utility functions.

Periodically during testing, we remove functions that have been used less than λ times. By observing that function-usage frequency has a logarithmic relation with data size, we set $\lambda = C \times \log_{10}(n)$, where $C = \frac{1}{2}$, n is the number of examples processed so far.[3]

4. Testbeds: Program-Solvable Tasks

We now introduce the three programmatic tasks for experiments: math problems, table question answering, and visual reasoning. We first state the default primitive functions (§4.1), then describe the specialized functions for each task.

4.1. The Default Primitive Functions

For all experiments, we instruct models to generate solutions as Python programs, so that default primitive functions are built-in Python functions.[4] We use these default primitives for MATH (§4.2), and add other data-related functions for TableQA (§4.3) and VisualQA (§4.4).

Task	Dataset	Size	Primitive Functions
MATH	algebra	881	built-in functions
	count & prob.	291	
	geometry	237	
	inter. algebra	503	
	number theory	497	
	prealgebra	636	
	precalculus	156	
TABLEQA	TabMWP	5,376	+ pandas
	WTQ	4,344	+ pandas
	HiTab	1,574	+ pandas + parse_table
VISUALQA	GQA	12,578	+ PIL.Image + locate_objects + visual_qa + crop_region

Table 1. Statistics and primitives for three tasks.

[3]To ensure all examples can be solved by functions available in the library, we update the solutions for examples previously using trimmed functions (< 5% of the dataset), by re-generating solutions under IMPORT & SKIP modes.

[4]https://docs.python.org/3/library/functions.html

Method	Metric	MATH alg	count	geo	inte	num	prealg	precal	TABLEQA TabMWP	WTQ	HiTab	VISUAL GQA
PRIMITIVE	acc ↑	0.15	0.14	0.06	0.05	0.16	0.21	0.10	0.43	0.20	0.09	0.37
	# ops ↓	**15.4**	10.9	**15.1**	**17.0**	12.3	12.1	20.8	17.4	24.3	16.5	24.8
	# lib ↓				—					—		—
INSTANCE	acc ↑	0.22	0.23	0.07	0.06	0.23	0.26	**0.17**	0.36	0.17	0.12	0.16
	# ops ↓	18.4	10.2	26.8	28.2	14.3	**10.6**	26.9	**8.3**	**8.4**	14.1	**18.8**
	# lib ↓	39	7	36	82	5	16	36	3,175	537	31	395
TROVE	acc ↑	**0.25**	**0.26**	**0.08**	**0.11**	**0.25**	**0.29**	**0.17**	**0.47**	**0.21**	**0.18**	**0.44**
	# ops ↓	18.8	**10.0**	25.4	23.9	**11.2**	11.7	**19.6**	10.9	9.2	9.3	20.3
	# lib ↓	10	1	7	8	8	4	7	10	11	5	7

Table 2. CODELLAMA-7B-INSTRUCT results on MATH, TABLEQA, and VISUAL tasks.

4.2. Math Problems

To test model abilities in solving math problems, we use the MATH (Hendrycks et al., 2021) dataset that covers questions from seven subjects: algebra, counting and probability, geometry, intermediate algebra, number theory, prealgebra, and precalculus. Table 1 lists the number of test examples, since our methods only need test data. We only use the default primitives, i.e., built-in Python functions.

4.3. Table Question Answering

We adopt three table question answering datasets: TabMWP (Lu et al., 2023), WTQ (Pasupat & Liang, 2015), and Hitab (Cheng et al., 2022). They cover a diverse range of question types and table structures. We represent tables as `pandas` DataFrame objects since this is a standard table library to use with Python; we accordingly add the `pandas` library into the set of primitive functions for the table QA task.

TabMWP The TabMWP dataset (Lu et al., 2023) includes math word problems on relational tables. Questions in TabMWP are relatively simple, such as performing numerical calculations (e.g., "What is the mean of the numbers?") and argument selection ("Who has the most OBJECT?"). We directly input the serialized tables in markdown format in model prompts, because they are relatively small.

WTQ The WikiTableQuestions (WTQ) dataset (Pasupat & Liang, 2015) contains questions about semi-structured Wikipedia tables, which feature un-normalized cell values, thus requires string processing (e.g., parse the number from "$ 100.00") and external knowledge retrieval (e.g., find the country name of "Franco Pellizotti (ITA)") operations.

Because WTQ tables can be too long to input limits, we put DataFrame previews in the prompt, and instruct models to use `pandas.read_table` to load tables from CSV files.

HiTab The Hitab dataset (Cheng et al., 2022) contains questions about hierarchical matrix tables. HiTab tables have more complex structures and require special operations such as multi-hop selection along a bi-dimensional header hierarchy. We similarly load HiTab tables from the source JSON files using the `parse_table` function used by Cao et al. (2023), and add it as the primitive functions.

4.4. Visual Reasoning

We use the GQA dataset (Hudson & Manning, 2019) that contains real-world images and compositional questions about them. However, the skills required to solve GQA questions are more advanced than the previous two tasks. Although image processing libraries such as `PIL` or `cv2` are available, our preliminary experiments show that using these libraries alone is extremely hard for models to generate viable solutions (only achieving 1% accuracy), not to mention further inducing advanced functions.

Therefore, we adopt three main primitive actions from VisProg (Gupta & Kembhavi, 2022) of two types: (1) neural modules: `visual_qa`, `locate_objects`; and (2) processing modules: `crop_region`; along with the image loading module `PIL.Image`. To reduce the extent of expert engineering, we did not adopt the other six actions used in VisProg, since they may be tailored to GQA queries or crafted shortcuts to assist models (e.g., `check_exists`). Removing these actions slightly degrades model performance.

5. Experiments

We introduce the experiment setup (§5.1) and evaluation metrics (§5.2), then report and analyze the results (§5.3).

5.1. Experimental Setup

We compare our method TROVE (§3) to the two baselines PRIMITIVE and INSTANCE (§2.2). We mainly use

CODELLAMA2-7B-INSTRUCT for experiments,[5] but also use GPT-4 to fairly compare with existing SOTA methods. We include $c = 2$ examples in prompts, sampled $K = 5$ responses in each mode, and trim the toolbox every 200 steps. By default, we set the decoding temperature to 0.6 and use top-p 0.95. We limit the model to generate at most 512 tokens to prevent excessive hallucination and save computational cost. To accommodate for randomness in the sampling result, we run each experiment five times and report the best-performing run. For all methods, we evaluate the results on test examples.

5.2. Evaluation Metrics

We propose three metrics to comprehensively evaluate generated solutions and induced functions.

Answer Correctness (acc ↑) The most practically important aspect of solutions is correctness. We measure if the execution outcome of the solution program exactly matches the ground-truth answer(s).

Solution Complexity (# ops ↓) We also measure the program complexity by counting the number of function calls involved. Solutions with fewer functions are easier and quicker to understand and verify.

Library Size (# lib ↓) It is important to control the library size and encourage function sharing across examples. Compared to multiple tools performing similar operations, fewer tools with distinct functionaly enable easier tool selection during solution generation. Since the number of primitive functions is always the same on a given dataset, we only report the number of additionally induced functions.

5.3. Model Performance

Table 2 shows the results on all datasets. TROVE produces the most accurate solutions with generally lower complexity, while maintaining a small, efficient function library.

Math Problems Compared to PRIMITIVE, TROVE substantially improves answer correctness by 30–120% across 7 datasets. Comparing to INSTANCE, TROVE yields 8.7–83.3% higher correctness using 60.0–90.2% fewer tools.

Table Question Answering Notably, INSTANCE yields lower correctness than PRIMITIVE on most datasets, while generating hundreds even thousands of functions. We conjecture the reason to be increased task difficulty and dataset size (than MATH), driving up the number of functions and confusing the model with too many low-utility options.

[5]We are the first to show that open-source LMs can make tools.

While TROVE, by reusing and trimming functions, alleviates this distraction and gives the best results.

Visual Question Answering TROVE still scores the best, but INSTANCE performs substantially worse than other methods. with a correctness drop of 0.21 compared to PRIMITIVE. Similarly to TABLEQA, a larger number of functions (i.e., 395) are created, which greatly challenges the solution generation process. Based on our result analysis, we conjecture that it is difficult to create many valid reusable functions for GQA, hence most induced functions are invalid and impair solution generation.

Please find more detailed ablation studies on example ordering and toolbox trimming in §8.

5.4. Comparing with Other Tool-Making Methods

In addition, we compare TROVE with three existing methods that perform tool making, namely LATM (Cai et al., 2023), CREATOR (Qian et al., 2023), and CRAFT (Yuan et al., 2023). Notably, all three methods include extra modules or supervision not required by our method, and were only demonstrated effective on *closed-source* GPT models. Specifically, LATM requires an extra training set to induce tools in advance, and a validation set to verify the tools. CREATOR runs multiple iterations of decision, execution, and rectification. CRAFT requires extra training data and tool retrieval modules, and necessitates GPT models.

To make a fair comparison, we also use GPT-4 with our TROVE approach, and test on three datasets — algebra from MATH, TabMWP from TABLEQA, and GQA from VISUALQA task — that overlap with these works. Due to resource limitations, we do not experiment on all 11 datasets. We believe the result differences in these 2 datasets are representative to demonstrate the superiority of our method.

In Table 3, TROVE outperforms these existing state-of-the-art methods on all datasets and most evaluation aspects, while having a much simpler pipeline. TROVE not only works with closed-source GPTs, but also open-source CODELLAMA (Table 2), with which CRAFT reported near-random performance using their method.

Training Advantage of Seen Primitive Functions While GPT-4 outperforms CODELLAMA on most tasks, it is impressive to see that they perform comparably on the GQA task, as shown in Table 4. We conjecture that this difference between GQA and other tasks comes from the advantage of models using corresponding primitive functions. For example, `pandas`, as a primitive in TABLEQA, may appear frequently in the training data, so models may be more proficient in or inclined to write solutions with this library.

In contrast, models may have never seen any data using

Method	MATH$_{algebra}$ acc ↑	MATH$_{algebra}$ # lib ↓	TabMWP acc ↑	TabMWP # lib ↓	GQA acc ↑	GQA # lib ↓
w/ additional supervision						
LATM	0.30	-	0.09	-	0.29	-
CRAFT	0.68	282	0.88	181	**0.45**	525
w/ additional rectification & iteration						
Creator	0.65	875	0.81	4,595	0.34	-
TroVE: *w/o supervision, rectification, or iteration*						
GPT-3.5	**0.68**	**17**	**0.89**	**25**	0.44	**10**
GPT-4	**0.72**	**16**	**0.92**	**38**	0.44	**8**

Table 3. Comparing with existing methods using GPT-4. We adopt the baseline results as reported in Yuan et al. (2023). We do not report the *complexity* metric since none of these methods report it (our results in Table 2).

GQA primitives since no large-scale data are annotated with these hand-crafted functions. The difficulty of models learning these primitives in context is similar to that of learning the induced functions. While GQA reduces GPT-4's advantage in using primitives, it is intriguing to observe that 7B CodeLLaMa2 performs on par with GPT-4 by making and (re-)using tools.

Model	Method	Evaluation Metrics acc ↑	# ops ↓	# lib ↓
CodeLLaMa	Primitive	0.37	24.6	-
	TroVE	0.44	20.3	7
GPT-4	Primitive	0.40	27.4	-
	TroVE	0.44	20.2	8

Table 4. 7B CodeLLaMa2 and GPT-4 perform comparably on the GQA task without training advantage.

6. Efficient Verification by Humans

Model-produced solutions may not be reliable, so we investigate if TroVE facilitates more efficient solution verification by humans. To test this hypothesis, we randomly selected 100 examples and asked 6 human evaluators to verify the correctness of solutions generated by Primitive, Instance, and TroVE on the WTQ dataset.[6]

We evaluate human performance from two aspects. (1) Detection accuracy: if they can accurately predict whether or not the solution is correct; the higher the better, and (2) Time used: how many seconds they need to verify an average example; the lower the better.

Table 5 shows the results. For *accuracy*, using tools in solutions (Instance and TroVE) improves detection accuracy by 13.0–14.3%, compared to using Primitive functions only. For the *time used*, our method reduces the average time by 31.4% compared to primitive, and 43.0% compared to instance. However, using irreusable tools (i.e., the instance setting) actually increases the time by 20.4%. Overall, TroVE substantially speeds up the verification process, while achieving similar detection accuracy. See §C for the test results of individual participants.

[6]We chose WTQ from the TableQA task, because TableQA functions are more complex than those in other tasks, and WTQ is the fairest representative of the task with mid-level difficulty.

Method	Accuracy ↑ avg	Accuracy ↑ std	Time (s) ↓ avg	Time (s) ↓ std
Primitive	0.77	0.109	25.5	6.671
Instance	0.88	0.024	30.7	12.750
TroVE	0.87	0.057	17.5	4.855

Table 5. Human accuracy and time in verifying model-produced solutions with three methods experimented.

7. Inducing Specialized Functions

TroVE demonstrates its generality on multiple tasks and datasets. In this section, we further show that TroVE can produce specialized functions that (1) differ in forms across tasks, and (2) differ in functions across datasets. We use the CodeLLaMa2-7B results as an example.

Different Function Forms Across Tasks We compare the three tasks and list a few exemplar functions in Table 6.

For MATH, the model often imports external libraries (`sympy`) to enable using advanced functions, or creates functions targeting certain problems (`calculate_remainder`). TableQA tasks induce more complex functions comprising many primitive functions (e.g., `get_match_after_condition`). VisualQA functions involve fewer primitives, for example, `get_image_region` is a chain of two primitives: `locate_objects` and `crop_region`.

Varied Functionalities Across Datasets For MATH and TableQA that have multiple datasets, we further analyze the variance in functions between the datasets.

Among MATH datasets, as shown in Figure 6, core functions such as `math` and `sympy` overlap. Meanwhile, some functions are particularly useful for certain questions, such as the self-defined `calculuate_remainder` for *number theory* questions, and `sympy.Polygen` for *geometry* questions.

Figure 7 shows some functions in three TableQA datasets. Due to the greater variance in question types and table structures, most functions differ except for the basic `pandas`.

Overall, TroVE can effectively propose both functions that are (1) generic to the task, and (2) specific to each domain. These induced functions not only help solve the problems, but also characterize their functional distribution.

Task	Example Functions

MATH	`from sympy import solve`
	
	`def calculate_remainder(numbers, modulus):`
	`    product = 1`
	`    for number in numbers: product *= number`
	`    return produce % modulus`

TABLEQA	`def get_match_after_condition(`
	`    df, condition_column: str, condition: any,`
	`    value_column: str) -> any:`
	`  """Get the match that comes after the match that`
	`  satisfies a condition in the specified column."""`
	`  row = df[df[condition_column] == condition]`
	`  index = row.index[0] + 1`
	`  if index < len(df):`
	`    return df.iloc[index][value_column]`
	`  else:`
	`    return None`

VISUALQA	`from PIL import Image`
	`from toolbox import crop_region, locate_objects`
	
	`def get_object_region(`
	`    image: Image.Image, object_name: str`
	`) -> Image.Image:`
	`  """Locate the crop the image of the object."""`
	`  boxes = locate_objects(image, object_name)`
	`  object_image = crop_region(image, boxes)`
	`  return object_image`

Table 6. Example functions induced by TROVE on three tasks.

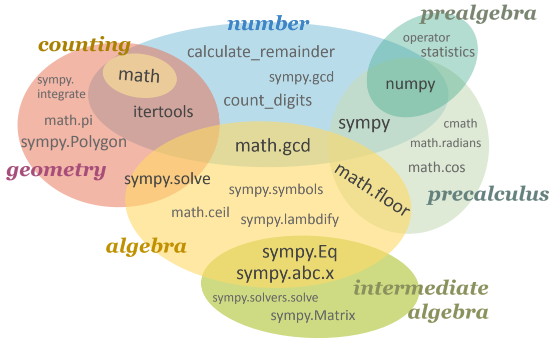


Figure 6. Illustration of MATH libraries for seven subjects.

8. Ablation Studies

We conduct ablation studies to test the robustness to ordering (§8.1) and the importance of toolbox trimming (§8.2).

8.1. Robustness to Ordering

Our method inputs examples in a streaming fashion and orders examples as they appear in the original dataset. However, it is important to study if variations in example ordering would affect final results. We select one dataset from each task (MATH$_{algebra}$, HiTab, GQA) as representatives for this examination.

We shuffle the examples five times with different random seeds and run TROVE on each of the five orderings. We report the range of metric values and their standard deviation in Table 7. As a reference, we denote results (from §3) using the original dataset order as *original*.

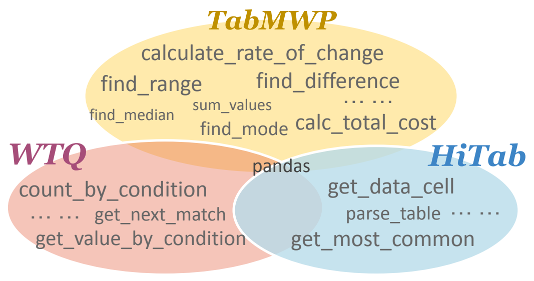


Figure 7. Illustration of three TABLEQA function libraries.

Method / Value	acc ↑	# ops ↓	# lib ↓
MATH$_{algebra}$			
original	0.25	18.8	10
value range	0.23–0.24	17.3–19.0	5–9
std.dev.	0.000	0.879	1.924
HiTab			
original	0.18	9.3	5
value range	0.17–0.18	9.0–9.9	8–10
std.dev.	0.003	0.358	0.837
GQA			
original	0.43	20.6	6
value range	0.43–0.44	20.4–20.6	6–8
std.dev.	0.005	0.150	0.957

Table 7. CODELLAMA results with alternative orders.

For all three datasets, no significant variance exists between the *original* and *randomized* ordering — the *original* results well within the *randomized* value range, and standard deviations are small — showing that TROVE is robust to example ordering. While the datasets may not be ordered in a way that optimizes function induction, the *original* ordering may just be another instance of somewhat *randomized* ordering.

8.2. Without Toolbox Trimming

Periodic function trimming is crucial to ensure the efficiency of TROVE. To demonstrate this point, we compare to a TROVE version without toolbox trimming. In Figure 8, when including the trimming mechanism, the size of function libraries significantly decreases by 74% - 90%. The accuracy and complexity in Table 8 also slightly degraded. While the trimming threshold and time interval are easily adjustable, one can flexibly keep more functions to explore more diverse functions, or fewer due to certain constraints.

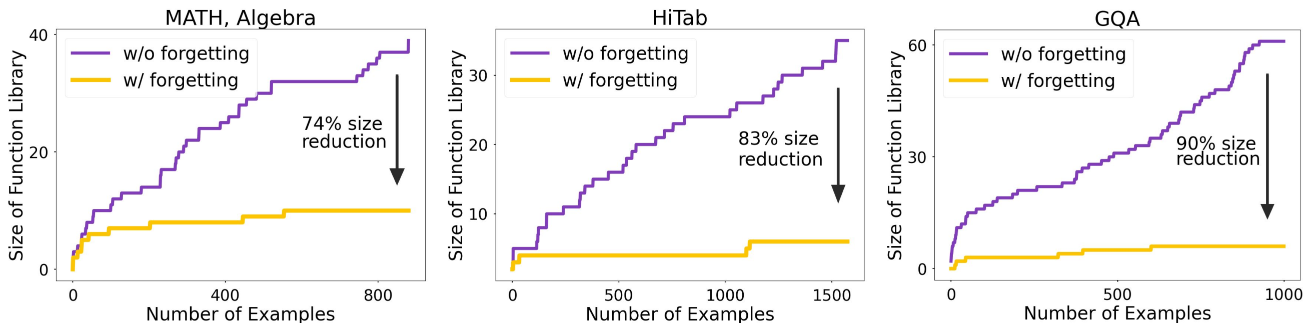


Figure 8. Library size without toolbox trimming.

Method	**MATH$_{algebra}$** acc↑	# ops↓	**HiTab** acc↑	# ops↓	**GQA** acc↑	# ops↓
without trim	0.25	19.6	0.15	10.5	0.39	21.1
with trim	0.25	18.8	0.18	9.3	0.44	20.3

Table 8. TROVE results with and without toolbox trimming.

9. Related Work

Generating Program Solutions Many works focus on generating Python programs to solve problems, such as math (Ni et al., 2023; Li et al., 2022b; Gao et al., 2023b; Chen et al., 2022) and table QA (Cao et al., 2023; Cheng et al., 2023). Yet most programs are built with basic operations or libraries (e.g., `sum`, `pandas`), and may be tedious and erroneous. Gupta & Kembhavi (2022); Subramanian et al. (2023); Surís et al. (2023) generate image-executable programs by hand-crafting task-specific functions, Gao et al. (2023a); Yang et al. (2023) extend this to audio and video modalities, but still require expert designs from humans. In contrast, our work enjoys the benefit of advanced functions with reduced human labor by inducing functions using LMs.

Domain-Specific Library Abstraction Shin et al. (2019) mine common code idioms and utilize them for program synthesis. Ellis et al. (2023) propose to induce functions bottom-up from a large corpus via a wake-sleep Bayesian process. Wong et al. (2021) improve the search efficiency, and Bowers et al. (2023) proposed a top-down method STITCH to save memory. Most recently, LILO (Grand et al., 2023) integrates LLMs into STITCH and abstract libraries with auto-documentation. While these methods all work on domain-specific logical forms, running them with general-purpose languages may vastly enlarge the search space, thus have limited applicability on many real-world tasks. Instead, our method generates general-purpose Python programs and can readily extend to new tasks.

Making Program Tools Using LLMs With the advances of LLMs, many works explore using LLMs to build tools. Cai et al. (2023) examine homogenous BigBench tasks, where in each task all examples use a single tool. Qian et al. (2023) work on math and table QA tasks but create numerous tools that are not re-used across tasks – this serves as our INSTANCE baseline. Yuan et al. (2023) increase tool sharing via additional training but still yield redundant tools. Xin et al. (2023) enables a growing lemma library for math theorem proving, but requires external supervision from the theorem prover and expert heuristics. Wang et al. (2023a) can build and learn skills in the embodied Minecraft world, yet requires self-verification and iterative refinement. In comparison, our method leverages execution agreement without any training or supervision.

programming language	DreamCoder	PATOIS	STITCH	LILO	TROVE
domain-specific	✓	✓	✓	✓	
general purpose					✓
tool making modules	LATM	Creator	CRAFT	Voyager	TROVE
training / curriculum	✓		✓	✓	
self-verification	✓	✓	✓	✓	
iterative refine		✓	✓	✓	
self-consistency					✓

Table 9. Modules required by existing methods and our TROVE.

10. Conclusion

We proposed TROVE, a method for inducing a toolbox of reusable functions to use in solving programmatic tasks. TROVE produces simpler and more accurate solutions than existing methods, using sufficiently smaller function libraries. Moreover, it facilitates human program verification to be 31% faster and 13% more accurate. Finally, TROVE can induce diverse functions across tasks and datasets, shedding insights on data-specific characteristics.

Acknowledgements

We thank Shuyan Zhou and Zhoujun Cheng for the insightful discussions about this work, Saujas Vaduguru for providing feedback about the draft, and all human participants of the verification study. Zora Zhiruo Wang is supported by the Carnegie Mellon University Presidential Fellowship.

Impact Statement

Our work proposes a self-adaptive system to solve programmatic tasks, which can produce or adopt programs without full human supervision. We encourage users to manually verify model-generated solutions to avoid potential safety issues, as the verification process exemplified in Section 6.

References

Artzi, Y. and Zettlemoyer, L. Weakly Supervised Learning of Semantic Parsers for Mapping Instructions to Actions. *Transactions of the Association for Computational Linguistics*, 2013. URL https://doi.org/10.1162/tacl_a_00209.

Bowers, M., Olausson, T. X., Wong, L., Grand, G., Tenenbaum, J. B., Ellis, K., and Solar-Lezama, A. Top-down synthesis for library learning. *Proc. ACM Program. Lang.*, 7(POPL), jan 2023. doi: 10.1145/3571234. URL https://doi.org/10.1145/3571234.

Cai, T., Wang, X., Ma, T., Chen, X., and Zhou, D. Large language models as tool makers. *arXiv preprint arXiv:2305.17126*, 2023. URL https://arxiv.org/pdf/2305.17126.

Cao, Y., Chen, S., Liu, R., Wang, Z., and Fried, D. Api-assisted code generation for question answering on varied table structures. *arXiv preprint arXiv:2310.14687*, 2023. URL https://arxiv.org/pdf/2310.14687.

Chen, W., Ma, X., Wang, X., and Cohen, W. W. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks. *arXiv preprint arXiv:2211.12588*, 2022.

Cheng, Z., Dong, H., Wang, Z., Jia, R., Guo, J., Gao, Y., Han, S., Lou, J.-G., and Zhang, D. HiTab: A hierarchical table dataset for question answering and natural language generation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, May 2022.

Cheng, Z., Xie, T., Shi, P., Li, C., Nadkarni, R., Hu, Y., Xiong, C., Radev, D., Ostendorf, M., Zettlemoyer, L., Smith, N. A., and Yu, T. Binding language models in symbolic languages. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=lH1PV42cbF.

Ellis, K., Wong, L., Nye, M., Sable-Meyer, M., Cary, L., Anaya Pozo, L., Hewitt, L., Solar-Lezama, A., and Tenenbaum, J. B. Dreamcoder: growing generalizable, interpretable knowledge with wake–sleep bayesian program learning. *Philosophical Transactions of the Royal Society A*, 381(2251):20220050, 2023.

Gao, D., Ji, L., Zhou, L., Lin, K. Q., Chen, J., Fan, Z., and Shou, M. Z. Assistgpt: A general multi-modal assistant that can plan, execute, inspect, and learn. *arXiv preprint arXiv:2306.08640*, 2023a.

Gao, L., Madaan, A., Zhou, S., Alon, U., Liu, P., Yang, Y., Callan, J., and Neubig, G. Pal: Program-aided language models. In *International Conference on Machine Learning*, pp. 10764–10799. PMLR, 2023b.

Grand, G., Wong, L., Bowers, M., Olausson, T. X., Liu, M., Tenenbaum, J. B., and Andreas, J. Lilo: Learning interpretable libraries by compressing and documenting code. *arXiv preprint arXiv:2310.19791*, 2023.

Gupta, T. and Kembhavi, A. Visual programming: Compositional visual reasoning without training. *arXiv preprint arXiv:2211.11559*, 2022. URL https://arxiv.org/pdf/2211.11559.

Hendrycks, D., Burns, C., Kadavath, S., Arora, A., Basart, S., Tang, E., Song, D., and Steinhardt, J. Measuring mathematical problem solving with the math dataset. *arXiv preprint arXiv:2103.03874*, 2021. URL https://arxiv.org/pdf/2103.03874.

Hudson, D. A. and Manning, C. D. Gqa: A new dataset for real-world visual reasoning and compositional question answering. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pp. 6700–6709, 2019.

Li, Y., Choi, D., Chung, J., Kushman, N., Schrittwieser, J., Leblond, R., Eccles, T., Keeling, J., Gimeno, F., Dal Lago, A., et al. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097, 2022a.

Li, Y., Choi, D., Chung, J., Kushman, N., Schrittwieser, J., Leblond, R., Eccles, T., Keeling, J., Gimeno, F., Dal Lago, A., et al. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097, 2022b.

Liang, P., Tripp, O., and Naik, M. Learning minimal abstractions. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 31–42, 2011.

Lu, P., Qiu, L., Chang, K.-W., Wu, Y. N., Zhu, S.-C., Rajpurohit, T., Clark, P., and Kalyan, A. Dynamic prompt learning via policy gradient for semi-structured mathematical reasoning. *arXiv preprint arXiv:2209.14610*, 2023. URL https://arxiv.org/pdf/2209.14610.

Majumder, B. P., Mishra, B. D., Jansen, P., Tafjord, O., Tandon, N., Zhang, L., Callison-Burch, C., and Clark, P. Clin: A continually learning language agent for rapid task adaptation and generalization. *arXiv preprint arXiv:2310.10134*, 2023.

Ni, A., Iyer, S., Radev, D., Stoyanov, V., Yih, W.-t., Wang, S., and Lin, X. V. Lever: Learning to verify language-to-code generation with execution. In *International Conference on Machine Learning*, pp. 26106–26128. PMLR, 2023.

Pasupat, P. and Liang, P. Compositional semantic parsing on semi-structured tables. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*. Association for Computational Linguistics, July 2015. URL https://aclanthology.org/P15-1142.

Qian, C., Han, C., Fung, Y. R., Qin, Y., Liu, Z., and Ji, H. Creator: Disentangling abstract and concrete reasonings of large language models through tool creation. *arXiv preprint arXiv:2305.14318*, 2023. URL https://arxiv.org/pdf/2305.14318.

Shi, F., Fried, D., Ghazvininejad, M., Zettlemoyer, L., and Wang, S. I. Natural language to code translation with execution. In *Proceedings of EMNLP*. Association for Computational Linguistics, December 2022. URL https://aclanthology.org/2022.emnlp-main.231.

Shin, R., Allamanis, M., Brockschmidt, M., and Polozov, O. Program synthesis and semantic parsing with learned code idioms, 2019.

Subramanian, S., Narasimhan, M., Khangaonkar, K., Yang, K., Nagrani, A., Schmid, C., Zeng, A., Darrell, T., and Klein, D. Modular visual question answering via code generation. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, July 2023.

Surís, D., Menon, S., and Vondrick, C. Vipergpt: Visual inference via python execution for reasoning. *arXiv preprint arXiv:2303.08128*, 2023.

Wang, G., Xie, Y., Jiang, Y., Mandlekar, A., Xiao, C., Zhu, Y., Fan, L., and Anandkumar, A. Voyager: An open-ended embodied agent with large language models. *arXiv preprint arXiv:2305.16291*, 2023a.

Wang, X., Wei, J., Schuurmans, D., Le, Q. V., Chi, E. H., Narang, S., Chowdhery, A., and Zhou, D. Self-consistency improves chain of thought reasoning in language models. In *The Eleventh International Conference on Learning Representations*, 2023b. URL https://openreview.net/forum?id=1PL1NIMMrw.

Wong, C., Ellis, K. M., Tenenbaum, J., and Andreas, J. Leveraging language to learn program abstractions and search heuristics. In Meila, M. and Zhang, T. (eds.), *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pp. 11193–11204. PMLR, 18–24 Jul 2021. URL https://proceedings.mlr.press/v139/wong21a.html.

Xin, H., Wang, H., Zheng, C., Li, L., Liu, Z., Cao, Q., Huang, Y., Xiong, J., Shi, H., Xie, E., et al. Lego-prover: Neural theorem proving with growing libraries. *arXiv preprint arXiv:2310.00656*, 2023.

Yang, Z., Li, L., Wang, J., Lin, K., Azarnasab, E., Ahmed, F., Liu, Z., Liu, C., Zeng, M., and Wang, L. Mm-react: Prompting chatgpt for multimodal reasoning and action. *arXiv preprint arXiv:2303.11381*, 2023.

Yin, P. and Neubig, G. A syntactic neural model for general-purpose code generation. *arXiv preprint arXiv:1704.01696*, 2017.

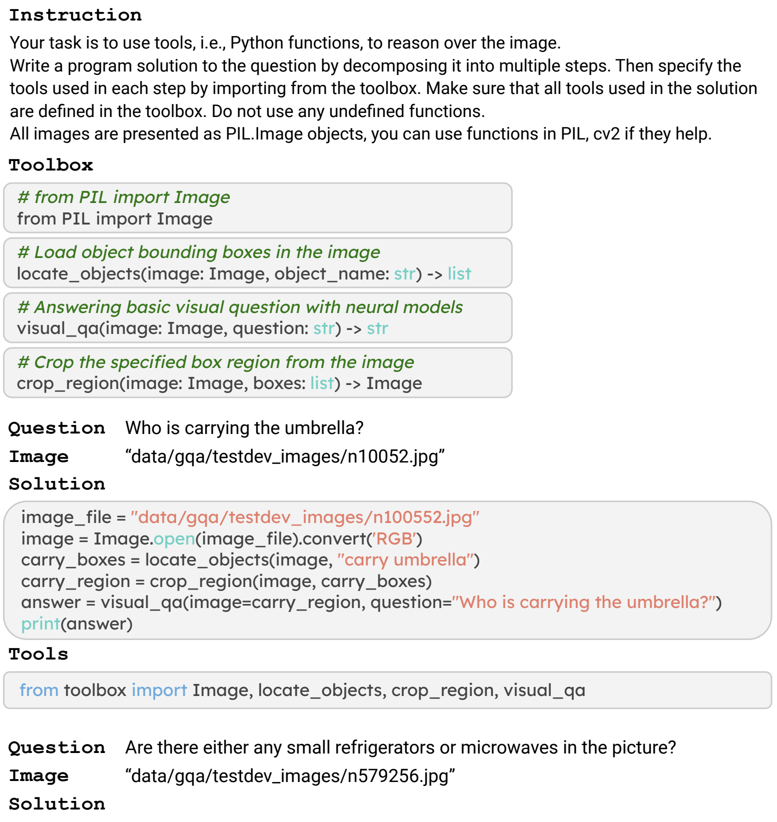
Yuan, L., Chen, Y., Wang, X., Fung, Y. R., Peng, H., and Ji, H. Craft: Customizing llms by creating and retrieving from specialized toolsets. *arXiv preprint arXiv:2309.17428*, 2023.

Zettlemoyer, L. and Collins, M. Online learning of relaxed ccg grammars for parsing to logical form. In *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL)*, pp. 678–687, 2007.

A. Prompt Example

We introduced baseline methods (PRIMITIVE and INSTANCE) in §2.2 and our main method in §3. To more concretely illustrate the prompts beyond textual description, we provide figure examples.

Figure 9 is an example prompt used in the PRIMITIVE setting, where on the bottom is the current test example and *Solution* is expected to be filled by the model.



Instruction

Your task is to use tools, i.e., Python functions, to reason over the image.
Write a program solution to the question by decomposing it into multiple steps. Then specify the tools used in each step by importing from the toolbox. Make sure that all tools used in the solution are defined in the toolbox. Do not use any undefined functions.
All images are presented as PIL.Image objects, you can use functions in PIL, cv2 if they help.

Toolbox

```
# from PIL import Image
from PIL import Image
```

```
# Load object bounding boxes in the image
locate_objects(image: Image, object_name: str) -> list
```

```
# Answering basic visual question with neural models
visual_qa(image: Image, question: str) -> str
```

```
# Crop the specified box region from the image
crop_region(image: Image, boxes: list) -> Image
```

Question Who is carrying the umbrella?
Image "data/gqa/testdev_images/n10052.jpg"
Solution

```
image_file = "data/gqa/testdev_images/n100552.jpg"
image = Image.open(image_file).convert('RGB')
carry_boxes = locate_objects(image, "carry umbrella")
carry_region = crop_region(image, carry_boxes)
answer = visual_qa(image=carry_region, question="Who is carrying the umbrella?")
print(answer)
```

Tools

```
from toolbox import Image, locate_objects, crop_region, visual_qa
```

Question Are there either any small refrigerators or microwaves in the picture?
Image "data/gqa/testdev_images/n579256.jpg"
Solution

Figure 9. Example prompt in the primitive setting.

We use the same format in the INSTANCE setting, but changed the *Instruction* to be:

> Your task is to write the solution with high-level tools, i.e., Python functions, to reason over the image.
>
> Think about the potential program solution for this example, you can create high-level functions, that could be used to solve this example. For example, if the solution involves multiple actions that are always used together, it is more efficient to create and use the tool.

Similarly, in the ONLINE setting, respectively for the CREATE, IMPORT, and SKIP modes, the *Instruction*s reads:

> Your task is to write Python program solutions to reason over images. You should also create Python functions that can be used by your solution, if you believe the function can be reused to solve other questions.

> Your task is to write Python program solutions to reason over images. The toolbox section lists all the available functions that can be used in your solution.

> Your task is to write Python program solutions to reason over images.

B. Evaluation Metric Details

Answer Correctness We measure if the solution execution result matches the annotated answers. For answers that are expected in a textual format (e.g., "brown"), we use the Exact Match (EM) metric; for numerical answers, we convert it to float type and round to two decimals, then measure if the values match (with a difference less than $1e{-}6$).

Program Complexity We quantify the complexity of programs in their number of operations. Concretely, we separate solutions into multiple expressions, and parse each expression into abstract syntax trees (AST). We take the depth of each AST as the number of operations conducted in the corresponding expression. We sum this value of all expressions and denote it as the complexity (i.e., number of operations) of the entire program.

C. Human Study: Individual Results

In the verification human study, performance between people may vary due to their programming expertise. Therefore, in this section, we present more detailed results of individual participants, and justify the significance of our findings.

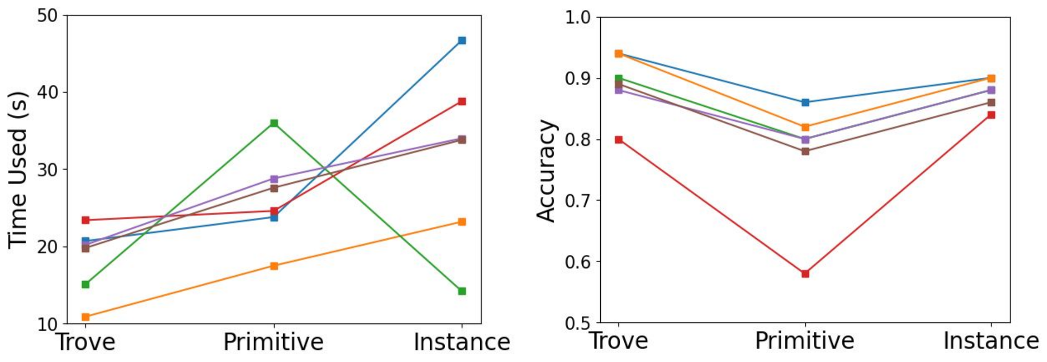


Figure 10. Individual verification accuracy and time used.

As shown in Figure 10 (left), verification accuracy is higher for tool-involved methods (TROVE and INSTANCE) compared to using primitive functions only (PRIMITIVE). Most people perform verification more accurately on programs produced by TROVE except one person (the red line), which also has lower detection accuracy in general.

Their times used for verification are shown on the right, with the same line color identifying each human. Most people find TROVE the fastest, PRIMITIVE taking 8.2 more seconds on average, and INSTANCE requires another 10.8 seconds. However, one person responded differently and personally found INSTANCE more efficient than PRIMITIVE, which is the major contribution of the relatively large variance of INSTANCE reported in §6.

D. Ablation Studies

We conduct several ablation studies regarding tool ordering (§D.1), tool selection (§D.2), tool rectification (§D.3), and tool frequency estimation (§D.4).

D.1. Problem-Specific Tool Listing Order

To keep our method simple and efficient, we always rank tools by their usage frequency for every example. To investigate if example-specific tool listing order would further improve the performance, we perform an ablation study by introducing an extra tool-ranking module.

Specifically, to list tools by their relevance to the example, we experimented with two methods: (i) rank tools by their lexical overlap (unigram F1 score) with the question, and (ii) rank tools using the experimented LM with the prompt "Please rank the tool functions based on their relevance to the question: \${question} \${tools}" alone with one in-context example.

As shown in Table 10, compared to our original example-agnostic frequency-based ranking, changing the ranking of tools does not substantially improve task correctness. This result demonstrates the general effectiveness of frequency-based

Method	MATH							TABLEQA			VISUAL
	alg	count	geo	inte	num	prealg	precal	TabMWP	WTQ	HiTab	GQA
freq (ours)	0.25	**0.26**	**0.08**	**0.11**	**0.25**	**0.29**	0.17	**0.47**	**0.21**	**0.18**	**0.44**
lexical	**0.26**	0.24	0.07	0.10	0.25	0.27	0.14	0.31	0.17	0.15	0.42
lm-based	0.24	0.25	**0.08**	0.10	**0.25**	0.28	**0.18**	0.43	0.18	0.17	0.43

Table 10. Comparing general frequency-based and example-specific tool listing methods.

ranking. In short, problem-specific tool listing does not improve performance, yet adds extra computation cost. We thus did not include this as a main module in TROVE.

D.2. Selection by Self-Consistency

While self-consistency (Wang et al., 2023b) has been widely adopted and proven effective, we further conduct an ablation study to prove its effectiveness particularly in our experimental setup. Specifically, we compare the original agreement-based selection (§3.2) with two other alternatives: (i) no sampling, only generating one response and always selecting it, and (ii) random selection over the sampled generations.

Method	MATH							TABLEQA			VISUAL
	alg	count	geo	inte	num	prealg	precal	TabMWP	WTQ	HiTab	GQA
self-consistency (ours)	**0.25**	**0.26**	**0.08**	**0.11**	**0.25**	**0.29**	**0.17**	**0.47**	**0.21**	**0.18**	**0.44**
no sampling	**0.25**	0.23	0.05	0.10	0.22	0.28	0.16	0.39	0.17	0.16	0.39
random selection	0.12	0.10	0.02	0.01	0.06	0.08	0.03	0.18	0.00	0.00	0.41

Table 11. Comparing the original self-consistency-based response selection to two alternatives, namely no sampling and random selection.

As shown by Table 11, removing sampling decreases the results slightly on all datasets; further, randomly selecting solutions without using the self-consistency heuristic, causes substantial degradation of results on all datasets. Results of both ablated settings effectively demonstrate the effectiveness of self-consistency in our experiments.

D.3. Influence of tool rectification

While some of the baseline methods (Qian et al., 2023) include a tool rectification module to improve the generation quality, we also study the influence of this module on TROVE. Rectification in our setting, more concretely, could be concretized into two modules: (i) right after creating the tool, rectify the implementation before adding it to the toolbox; or (ii) when encountering a new example, import an existing tool and rectify its implementation based on the current example.

To realize this, we add another generation step for tool and solution rectification: given the instruction "Your task is to update the program solution with the provided math tools, i.e., Python functions. For each example, based on the initial solution, select the suitable tool(s), you need to rewrite the solution using the selected tools. Do not use any undefined functions or unspecified tools." We input the current toolbox preview, the current example, tools, and solutions generated in the last (creation/import) steps, then ask models to output an updated version of the tools and solutions. We always adopt the tools and solutions after rectification.

Method	MATH							TABLEQA			VISUAL
	alg	count	geo	inte	num	prealg	precal	TabMWP	WTQ	HiTab	GQA
no rectification (ours)	**0.25**	**0.26**	**0.08**	**0.11**	**0.25**	**0.29**	**0.17**	**0.47**	**0.21**	**0.18**	**0.44**
rec. after creation	0.17	0.15	0.03	**0.11**	0.24	0.24	0.10	0.26	0.00	0.04	**0.44**
rec. after import	0.23	0.25	0.05	0.09	0.24	0.23	0.16	0.22	0.00	0.11	0.41

Table 12. Results after applying tool rectification after tool creation or importing.

We experiment with both settings and show their results in Table 12. For (i), rectification after creation barely helps task performance but adds extra compute. For (ii), rectification after import hurts performance while costing extra compute. We manually analyzed the outputs and conjecture the reason to be that LMs tend to over-optimize the tool implementation to the context of the current example, thus making the tools less generic in functionality and non-reusable for future generations.

D.4. Tool Freuquency Estimation

We estimate the tool usage to follow a logarithmic relation with data size in §3.3. To further validate the appropriateness of this choice, as well as its effectiveness in downstream performance, we compare it with another trimming threshold — a fixed-value threshold $\lambda = 2$ selected by manually observing tool-using frequency. The result in Table 13 shows that using fixed-value thresholds underperforms log-based thresholds by a large margin on all datasets.

Method	MATH							TableQA			Visual
	alg	count	geo	inte	num	prealg	precal	TabMWP	WTQ	HiTab	GQA
logarithmic (ours)	**0.25**	**0.26**	**0.08**	**0.11**	**0.25**	**0.29**	**0.17**	**0.47**	**0.21**	**0.18**	**0.44**
fixed value	0.21	0.10	0.04	0.08	0.22	0.27	0.15	0.23	0.00	0.11	0.43

Table 13. Results after applying tool rectification after tool creation or importing.