Large Language Models are Zero-Shot Multi-Tool Users

Luca Beurer-Kellner [*] [1] **Marc Fischer** [*] [1] **Martin Vechev** [1]

Abstract

We introduce ACTIONS, a framework and programming environment to facilitate the implementation of tool-augmented language models (LMs). Concretely, we augment LMs with the ability to call actions (arbitrary Python functions), and experiment with different ways of tool discovery and invocation. We find that, while previous works heavily rely on few-shot prompting to teach tool use, a zero-shot, instruction-only approach is enough to achieve competitive performance. At the same time, ACTIONS zero-shot approach also offers a much simpler programming interface, not requiring any involved demonstrations. Building on this, we show how ACTIONS enables LLMs to automatically discover and combine multiple tools to solve complex tasks. Overall, we find that inline tool use as enabled by ACTIONS, outperforms existing tool augmentation approaches, both in arithmetic reasoning tasks and text-based question answering. Our implementation extends the open source `LMQL` programming language for LM interaction (Beurer-Kellner et al., 2023) and is available at ANONYMIZED (upon publication).

1. Introduction

State-of-the-art Large Language Models (Large LMs - LLMs) offer powerful conversational abilities (Brown et al., 2020; Rae et al., 2021; Chowdhery et al., 2022; Touvron et al., 2023; Bommasani et al., 2021). However, they have been shown to struggle with symbolic reasoning, information look-up and consistency (Gao et al., 2023).

Tool-Augemented LMs To address this, researchers took inspiration from the idea of System-1-System-2 dichotomy in human cognition (Kahneman, 2011), where System 1 is fast intuitive reasoning, prone to errors, and System 2 is slow, structured logical reasoning. By treating the LM as a System 1 reasoning agent, and augmenting it with System 2 tools, researchers hope to improve LLM reasoning abilities (Mialon et al., 2023; OpenAI; Schick et al., 2023; Gao et al., 2023; Chen et al., 2022; Shuster et al., 2022; Nakano et al., 2021; Thoppilan et al., 2022; Komeili et al., 2022; Lazaridou et al., 2022; Cobbe et al., 2021; Yao et al., 2022), using a wide range of different techniques and approaches.

```
argmax
    """Q: What is the population of
          Germany and the US combined?
       A: Let's think step by step.[REASONING]
          Therefore the answer is[ANSWER]"""
from
    "openai/text-davinci-003"
where
    inline_use(REASONING, [wiki, calc])
```

Model Output

```
Q: What is the population of Germany and the US combined?
A: Let's think step by step.
First, let's find the population of Germany and the US. We can
use the wiki tool to search for the population of each country.
For Germany, <<wiki("Population of Germany") | The
demography of Germany is monitored by the Statistisches
Bundesamt (Federal Statistical Office of Germany).
According to the most recent data, Germany's population is
84,270,625...>>
For the US, <<wiki("Population of the United States") | The
United States had an official estimated resident
population of 333,287,557 ...>>
Now, let's use the calc tool to add the two populations
together. The population of Germany and the US combined is
<<calc("84270625 + 333287557") | 417558182>>.
Therefore the answer is 417558182.
```

Figure 1: Example of ACTIONS usage within a `LMQL` query (top). In the output (bottom) we show the full output and highlight tool usage in orange boxes and input in the blue.

This Work In this work we systemize these existing approaches and discover two crucial aspects to all of them: (i) tool discovery, i.e. teaching the LM which tools are available and what their interfaces are, and (ii) tool execution, i.e. how the LM invokes the tool and incorporates its output.

Based on this, this work introduces ACTIONS, a framework

[*]Equal contribution [1]ETH Zurich. Correspondence to: Luca Beurer-Kellner <luca.beurer-kellner@inf.ethz.ch>.

ICML Workshop on Knowledge and Logical Reasoning in the Era of Data-Driven Learning Hybrid Workshop at ICML 2023.Copyright 2023 by the author(s).

for tool-augmented LMs supporting multiple strategies for tool discovery and execution. ACTIONS provides an accessible interface, allowing users to easily expose any Python function to the LM reasoning process, without the need for manual guidance or few-shot demonstrations. We implement ACTIONS on top of the LMQL language for scripted prompting (Beurer-Kellner et al., 2023) (see §2).

We showcase ACTIONS in Fig. 1. Given a simple natural language prompt, we can use ACTIONS to expose the Python functions `wiki` and `calc` to the LM reasoning process. During generation, the LM can then simply invoke the functions using a demonstrated calling convention, where execution is handled transparently by ACTIONS. We refer to this tool usage pattern as inline usage, as the LM can invoke tools at any time during the generation process. While ACTIONS supports multiple tool discovery paradigms, it focuses on a zero-shot instruction-based approach. To this end, internally, ACTIONS uses a dynamically generated instructive prompt to teach the LM about tools and handles all tool execution by constraining and guiding LLM generation. Users can also specify different tool use paradigms, such as program-aided (Gao et al., 2023) or ReAct (Yao et al., 2022), simply by relying on respective ACTIONS functions like `inline_use` in Fig. 1. For this, no further adaptation of the tool implementations or the user prompt is required.

We demonstrate the utility of ACTIONS in several case studies and carry out a quantitative comparison of different tool use paradigms. We find that inline tool use as shown in Fig. 1 outperforms existing tool augmentation approaches, both on arithmetic reasoning tasks and text-based question answering. ACTIONS is competitive with few-shot prompted tool usage approaches, while being much easier to use and implement, not requiring any few-shot demonstrations or training. This greatly facilitates multi-tool usage, where (domain-specific) tools can be added and removed as required and model guidance and tool execution is handled transparently by ACTIONS.

Key Contributions To summarize, our contributions are:

- A review and systematization of existing tool-augmented LMs, identifying the tool discovery and tool execution phases as crucial to tool use (§3).
- ACTIONS, a novel framework for tool-augmented LMs, implementing both the tool discovery and tool execution phases. ACTIONS focuses on zero-shot inline tool use, which is effective with state-of-the-art LLMs, but also easy to use and implement (§4).
- A thorough evaluation showcasing use cases for ACTIONS and a quantitative comparison of different tool use paradigms in and outside of ACTIONS (§5).

```
argmax
   "Hey!"
   "2 + 2 = [ANSWER]"
from
   "jeffwan/vicuna-13b"
where
    INT(ANSWER)
```

(a) Constrained decoding with integers.

```
argmax
   "Q: From which countries did the Norse
    originate?\n"
   "Action: Let's search Wikipedia for the
    term '[TERM]\n"
   result = await wikipedia(TERM)
   "Result: {result} \n"
   "Final Answer:[ANSWER]"
from
   "openai/text-davinci-003"
where
   STOPS_AT(TERM, "'")
```

(b) Searching Wikipedia during LM reasoning.

Figure 2: Two examples of simple LMQL programs.

2. Background: Scripted Prompting

Recently, Beurer-Kellner et al. (2023) introduced the idea of scripted prompting, which extends natural language prompts with (Python) scripting capabilities and logical constraints.

We provide a brief demonstration of LMQL in Fig. 2, with two example queries. The first, Fig. 2a, asks a locally hosted Vicuna model (Chiang et al., 2023) to perform a simple mathematical operation while constraining the output to be a number. Here, **`argmax`** denotes the decoding strategy (others would be **`sample`** or **`beam`**). The subsequent block is an arbitrary Python program. Each top-level string is part of the prompt fed to the LM, where `[VARIABLE]` denotes a variable to be filled in. All text up to the variable is given to the LM as a prompt. The model then completes the sequence and assigns the result to the variable in the context of the query program. In addition to this prompting mechanic, the **`where`** block specifies constraints on the variables. Shown here is a constraint `INT(ANWER)`, which restricts the model to only generate tokens that form a valid integer number. For a more detailed disucssion of constraints we refer to Beurer-Kellner et al. (2023).

Fig. 2b showcases a more advanced query program where an (OpenAI) LM responds to a general knowledge question, but scripting logic first looks up an LM-determined term on Wikipedia and adds the result to the prompt. To prevent the running on issue Beurer-Kellner et al. (2023) when generating `TERM`, we enforce a stopping phrase on `TERM`. Since LMQL is a superset of Python, `wikipedia` can be a standard Python function that will automatically be called during generation.

```
Thought: Lets find the population of German.
Action: wiki("Population of Germany")
Observation: The demography of Germany is ...
```

Figure 3: Example of the thought-action-observation template in ReAct.

3. Tool Augmentation

The idea of augmenting an LM with tools has been explored in several recent works. Here we provide a brief overview of these approaches and and systematize them.

ChatGPT Plugins (OpenAI) is a proprietary system by OpenAI for models accessed through their API. The LLM is provided with textual descriptions and an HTTP API endpoint for a tool. These tools are executed server-side and only the result is visible to the user.

PAL (Gao et al., 2023) and Chen et al. (2022), both, rather than augmenting an LM with inline tools propose to let the LM generate a python program (with text and reasoning as comments) to answer the prompt rather than reply in natural language. This program is then executed and the result is returned to the user. PAL has also been extended to use symbolic reasoning libraries (He-Yueya et al., 2023).

Schick et al. (2023) formalize tool-usage for language models, in a way similar to what we consider inline use in Actions, however, they require tool-specific training. The trained models then output special syntax that stops the decoder, executes a tool and inserts the result into the currently decoded sequence.

ReAct (Yao et al., 2022) prompts a model with a specific template interleaving reasoning ("thoughts" by the LLM) and actions and observations from a task-specific action space. The common `ReAct` template with example actions and observations is shown in Fig. 3.

With respect to tools, the most common use cases are variants of search, text or knowledge retrieval, usually in dialogue systems or for question answering (Shuster et al., 2022; Nakano et al., 2021; Thoppilan et al., 2022; Komeili et al., 2022; Lazaridou et al., 2022; Cobbe et al., 2021).

All these methods share two key components: (i) a way for the LM to discover the tool and (ii) a way for the LM to use the tool. We discuss them in more detail in in the following.

3.1. Tool Discovery

Before an LM can use a tool, it needs to know about its availability and how to use it.

Training or fine-tuning with a dataset can be used to train an LM aware of the set of available tools, as used in Schick et al. (2023)'s Toolformer. This approach requires a dataset of tool usage examples, which can be expensive to collect, though Schick et al. (2023) also provide an approach to automatically generate such datasets. A downside, however, is that the set of tools is fixed at training time and cannot be changed at a later point without retraining.

Few-Shot Prompting Here the syntax for using tools is shown in a few-shot prompt before the actual LM reasoning begins. This allows the LM to discover the invocation and expected output of tools. By seeing which tools have been used in the few-shot prompts the LM can learn and adapt to a broad set of tools. While reasonably flexible, this approach can become problematic as the number of tools increases and few-shot prompts have to be very long and comprehensive to cover all tools and combinations.

Instruction As an alternative to few-shot prompting, ChatGPT Plugins (OpenAI) use a set of instructions to describe the available tools. This approach is very flexible and allows to add new tools at inference time. However, to be effective it requires a large-scale model trained to follow and understand complex instructions.

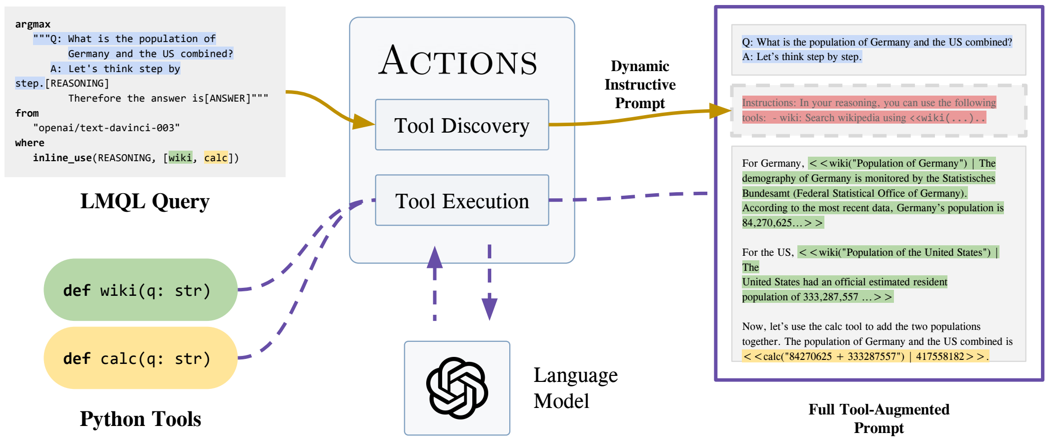
3.2. Tool Usage

Next, we briefly summarize the different ways LLM tool use can be implemented.

Post Hoc PAL (Gao et al., 2023) and Chen et al. (2022) let the LM generate a Python program and evaluate the resulting snippet of code with a python interpreter. This approach is flexible and enables powerful, symbolic reasoning, however, the LM must possess significant planning capabilities to generate a correct program. Further, deferring tool use until after the program was generated means that the LM cannot use the tool output or intermediate results to guide its generation process.

Template As an alternative, ReAct (Yao et al., 2022) prompts an LM with a specific template where tool use is admitted only in designated syntactic locations. To enable multiple consecutive tool uses, ReAct relies on an interpretation loop that alternates LLM *thoughts* and *actions* (i.e., tool use). This approach is flexible and allows to use multiple tools in sequence. However, the template syntax is limited and does not allow to use tools in arbitrary locations.

Ad Hoc or *inline* tool usage is the most flexible and powerful way to use tools in LMs. Once a specific syntax for tool use is emitted by the model during generation, the specified tool is evoked, the result inserted into the prompt, and text generation continues. This approach is used by Schick et al. (2023) and ChatGPT Plugins (OpenAI).



```
argmax
   """Q: What is the population of
         Germany and the US combined?
      A: Let's think step by
step.[REASONING]
         Therefore the answer is[ANSWER]"""
from
   "openai/text-davinci-003"
where
   inline_use(REASONING, [wiki, calc])
```

```
def wiki(q: str)
```

```
def calc(q: str)
```

Q: What is the population of Germany and the US combined?
A: Let's think step by step.

Instructions: In your reasoning, you can use the following tools: - wiki: Search wikipedia using <<wiki(...)..

For Germany, < <wiki("Population of Germany") | The demography of Germany is monitored by the Statistisches Bundesamt (Federal Statistical Office of Germany). According to the most recent data, Germany's population is 84,270,625…> >

For the US, < <wiki("Population of the United States") | The
United States had an official estimated resident population of 333,287,557 …> >

Now, let's use the calc tool to add the two populations together. The population of Germany and the US combined is < <calc("84270625 + 333287557") | 417558182> >.

Figure 4: ACTIONS implements a framework handling both tool discovery via dynamic instructive prompt as well as tool invocation, issueing multiple consecutive LLM calls to interleave tool use with LM reasoning.

4. ACTIONS: A framework for tool-augmentated LM reasoning

Based on existing tool augmentation approaches, we introduce ACTIONS, a programmatic framework for tool augmentation with LMs. By default, ACTIONS is an instruction-based, zero-shot approach, that allows ad-hoc tool use in arbitrary locations. It is implemented on top of the `LMQL` query language, and offers a lightweight and simple interface to augment existing LMs with tools (OpenAI and `transformers` models).

We provide an overview of the ACTIONS framework in Fig. 4. In ACTIONS, any ordinary Python function can be exposed as a tool. For tool discovery, ACTIONS relies on a dynamically constructed instructive prompt, that is inserted right before the model generates output in response to a user prompt (cf. Fig. 4, right). To use ACTIONS, users employ designated constraints in the **`where`** clause of an `LMQL` query, to expose functions as LLM tools. This enables precise scoping of tool use, allowing users to limit tool access only to specific parts of the LM output or use different tools across different parts of a prompt.

Tool Discovery ACTIONS automatically derives the description and a simple demonstration from the docstring[1] of a tool function, as shown in Fig. 5. This allows users to expose any Python function as a tool, without requiring any additional annotations. If desired, users can provide additional few-shot samples to aid discovery, disabling dynamic instructions if needed. ACTIONS is also compatible with LMs trained for tool use, as it operates during inference only. All these options are non-exclusive and can be combined.

[1] `https://peps.python.org/pep-0257/`

Transient Instructions In `LMQL` queries with more than one variable, the variable-specific, dynamic tool discovery prompt is automatically removed, once tool usage in a given variable has completed. This keeps tool-specific instructive prompts local to the tool use, and avoids interference with the reasoning process on other variables. For instance, in Fig. 4, when query execution has finished tool use in `REASONING` and moves on to decode `ANSWER` next, the instructive prompt of `REASONING` is removed and the model is only shown tool results interleaved with its reasoning, without any tool instructions (full opacity parts in Fig. 4). This also allows users to use different tools in different variables, without any interference on the instruction level.

Tool Use in ACTIONS On the tool execution level, ACTIONS provides support for three core modes, but can also be extended to new forms. The core modes are:

- `inline_use(VAR, actions)` refers to inline action use as showcased in Fig. 1. Here the LM is allowed to call functions provided as `actions` at any point during reasoning, simply by emitting `<<` followed by the function name and the arguments. Based on this calling convention, the corresponding tool is then invoked, where the pipe character `|`, result and `>>` are inserted by the ACTIONS runtime.

- `inline_code(VAR)` Inline code execution treats every line emitted by the model (as part of `VAR`) as Python code, evaluates it, and appends the evaluation result at the end of the line. This provides the LM with real-time feedback during code generation, and can be seen as a special case of `inline_use(VAR, actions)` where the only action is a python interpreter.

- `react(VAR, actions)` refers to REACT-style action use where the model is instructed to reason based on a template with designated lines for `Thought`, `Action` and `Observation`, as introduced by Yao et al. (2022).

We note that the simple post-hoc tool use like PAL (Gao et al., 2023) is not supported explicitly, as it can be trivially written in standard LMQL and does not need any additional support. We showcase this in App. A.

Summary To summarize, ACTIONS is a way to enable the simple use of tools in LMs. In particular, it aims to be an instruction-centered inline-use framework but also supports other paradigms, like different forms of tool execution and few-shot demonstrations.

The only other instruction based, inline-use currently is OpenAI, which is only available through a commercial interface with few models. In contrast, ACTIONS aims to support this paradigm for any model and tool within the realm of the existing LMQL query language, is open source and research-friendly, allowing for easy experimentation and extension.

5. Experimental Evaluation

We evaluate ACTIONS with case studies in §5.1 and §5.2 and additionally carry out a quantitative comparison of different tool use paradigms in §5.3, comparing unassisted chain-of-thought, program-aided reasoning (Gao et al., 2023) and the inline modes of ACTIONS. We first describe the experimental setup and then present the results.

Model We use OpenAI's `text-davinci-003` InstructGPT model (Wei et al., 2022b) for all our experiments, because of its strong zero-shot capabilities (Kojima et al., 2022). We also experimented with the smaller `text-curie-001` model, but, overall, find that its instruction following capabilities are insufficient for zero-shot tool use.

Datasets To assess the effectiveness of tool use during reasoning, we focus on object tracking, factual knowledge tasks and arithmetic reasoning. More specifically, we asses performance with the Tracking Shuffled Objects benchmark (Srivastava et al., 2022), the fact checking benchmark Fever (Thorne et al., 2018), and the hard arithmetic benchmarks GSM-Hard (Gao et al., 2023) and the ALGEBRA (He-Yueya et al., 2023). We specifically choose the latter two over GSM8K, which has been found out to be of low complexity with respect to the numbers in use (Gao et al., 2023; Madaan & Yazdanbakhsh, 2022).

Sample Size In our quantitative evaluation, we choose to limit our evaluation to 50 samples per dataset, and allocate our compute budget more towards a comparison of different prompting and tool use strategies.

```
room = [
    Person("Alice", 28, "data scientist"),
    Person("Bob", 81, "student"),
    Person("Aunty", 45, "CEO"),
]

def people():
    """
    Returns a list of people

    Example: people()
    Result: "Peter", "Bob", "Alice"
    """
    return ", ".join([p.name for p in room])

def attr(person: str, attribute: str):
    """
    Get an attribute of a person.

    Example: attr("Alice", "age")
    Result: 27
    """
    assert attribute == "age"
    p = [p for p in room if p.name == person][0]
    return getattr(p, attribute)

argmax
    "Q: What is the average age of people
    in the room?\n"
    "A: Let's think step by step\n"
    "[REASONING]"
from
    "openai/text-davinci-003"
where
    inline_use(REASONING, [people, attr, calc])
```

Model Output

Q: What is the average age of people in the room?

A: Let's think step by step

First, let's get a list of all the people in the room using the people() tool. `<<people() | Alice, Aunty, Bob>>` Alice, Aunty, Bob are the people in the room.

Next, let's get the age of each person using the attr() tool.
`<<attr("Alice", "age") | 28>>` 28 is Alice's age.
`<<attr("Aunty", "age") | 45>>` 45 is Aunty's age.
`<<attr("Bob", "age") | 81>>` 81 is Bob's age.

Finally, let's calculate the average age of the people in the room using the calc() tool. `<<calc("(28+45+81)/3") | 51.33>>` 51.33 ✓ is the average age of the people in the room.

Figure 5: Using multiple tools at once with ACTIONS and `text-davinci-003`. A room of people of different age is exposed purely via a tool-based interface, where the LLM successfully operates across all tools, to answer a question about the average age of people in the room.

Few-Shot Prompts Although ACTIONS is a zero-shot framework by default, we also compare with few-shot prompting instead of using an instructive prompt. For this, we use the few-shot examples provided in Gao et al. (2023).

Q: What is the average age of people in the room?

A: Let's think step by step

Instructions: In your reasoning, you can use the following tools:

- people: Returns a list of people Usage: <<people() | "Peter", "Bob", "Alice">>
- attr: Get an attribute of a person. Usage: <<attr("Alice", "age") | 27>>
- calc: Evaluate the provided arithmetic expression in Python syntax. Usage: <<calc("1+2*3") | 7>>

You can also use the tools multiple times in one reasoning step.

Reasoning with Tools: [...]

Figure 6: The dynamically generated instructive prompt for tool discovery in Fig. 5.

5.1. Case Study: Multi-Tool Use

ACTIONS allows users to easily expose arbitrary Python functions to the LLM. To demonstrate this, consider the example given in Fig. 5. There, we define two simple actions `people` and `attr` as Python functions of the same name. The former returns a list of people in the room, the latter returns attributes of a given person (e.g., age). Exposing these two actions and the previously shown `calc` via ACTIONS, we can equip an LM to reason about people in the room and their attributes. Note, how `attr` in Fig. 5 even has multiple arguments to specify person and attribute name separately.

Instructive Prompt To enable tool discovery, we rely on inline action usage (cf. `inline_use`), which automatically inserts a corresponding instructive prompt before the `REASONING` step. The dynamically generated prompt is shown in Fig. 6. As shown in the snippet, the instructive prompt includes descriptions of all available tools, including the standard action `calc` and the user actions `people` and `attr`.

Model Behavior We ask the model to compute the average age of all people in the room. As shown in Fig. 5, the successfully discovers and uses the provided actions, first obtaining a list of all people in the room via `people` and then determining the age of each person via `attr(<p>, "age")`, to finally calculate their average via `calc`.

Baseline Comparison We also construct an analogous chain-of-thought prompt, first listing all `Person` objects, then asking the model to compute the average age. Even though the model is able to parse age information from the data objects, it fails to compute the average age correctly, as it cannot resort to a robust arithmetic tool. Further, more complex data with more attributes could also be challenging for the model to parse. Including all information in the prompt in this way, may also not be feasible with large amounts of data. In contrast, tool use and ACTIONS allow the LLM to guide the data selection process, actively deciding which information to include and which to ignore. Overall, this lowers token usage and enables more focused reasoning.

```
kvstore = {}
def store(key, value):
    """
    Store a key-value mapping.

    Example: store("A", "has B")
    Result: A -> has B
    """
    kvstore[key] = value
    return f"{key} -> {value}"
def swap(key1, key2):
    """
    Swap the values of key1, key2.

    Example: swap("A", "B")
    Result: None
    """
    temp = kvstore[key1]
    kvstore[key1] = kvstore[key2]
    kvstore[key2] = temp
def load(key):
    """
    Get the stored value of a key-value mapping.,

    Example: load("A")
    Result: "has B"
    """
    return kvstore[key]

argmax
    "Q: Alice, Bob, and Claire are playing a game.
    At the start of the game, they are each holding
    a ball: Alice has a black ball, Bob has a brown
    ball, and Claire has a blue ball. \n\nAs the game
    progresses, pairs of players trade balls. First,
    Bob and Claire swap balls. Then, Alice and Bob
    swap balls. Finally, Claire and Bob swap balls.
    At the end of the game, what ball does Alice have?"
    "A: Let's think step by step\n"
    "[REASONING]"
from
    "openai/text-davinci-003"
where
    inline_use(REASONING, [store, load, swap])
```

Model Output

...

A: Let's think step by step

At the start of the game, Alice has a black ball, Bob has a brown ball, and Claire has a blue ball. `<<store("Alice", "black") | Alice -> black>>` Alice -> black, `<<store("Bob", "brown") | Bob -> brown>>` Bob -> brown, `<<store("Claire", "blue") | Claire -> blue>>` Claire -> blue.

Bob and Claire swap balls. `<<swap("Bob", "Claire") | None>>` None, `<<load("Bob") | blue>>` blue, `<<load("Claire") | brown>>` brown.

Alice and Bob swap balls. `<<swap("Alice", "Bob") | None>>` None, `<<load("Alice") | blue>>` blue, `<<load("Bob") | black>>` black.

Claire and Bob swap balls. `<<swap("Claire", "Bob") | None>>` None, `<<load("Claire") | black>>` black, `<<load("Bob") | brown>>` brown.

At the end of the game, Alice has a blue ball. ✓.

Figure 7: A simple key-value store for LM use via `inline_use`. Using the operators `store`, `load` and `swap`, the LLM can manage state for improved object tracking.

5.2. Case Study: Stateful Tool Use

Since ACTIONS tools are standard Python functions, the surrounding program context can be leveraged to enable

Table 1: Comparing standard and tool-assisted LLM task accuracy with ACTIONS-assisted task accuracy.

	Unassisted	Post-Hoc	**ACTIONS (ours)**		
Task	*Chain-Of-Thought*	Program-Aided	`reAct`	`inline_use`	`inline_code`
GSM-Hard (Gao et al., 2023)					
Zero-Shot	0.12	0.56	0.24	0.40	**0.58**
3-Shot	0.26	**0.68**	-	0.56	**0.68**
ALGEBRA (He-Yueya et al., 2023)					
Zero-Shot	0.50	0.56	0.46	0.54	**0.62**
3-Shot	0.64	**0.70**	-	0.58	0.66
FEVER (Thorne et al., 2018)					
Zero-Shot	0.82	-	0.84	**0.92**	-

stateful behavior. To demonstrate this, we consider the object tracking task, as included in the BIG benchmark collection for LLMs (Srivastava et al., 2022). An example task is shown in line 31 in Fig. 7. Here, the model is tasked with tracking a number of objects that are exchanged among a group of people.

To support this with ACTIONS, we implement a simple key-value storage data structure, with three operators. `store` to save a key-value mapping, `load` to recover the value of a given key, and `swap` to swap the value between entries. We show the full implementation in Fig. 7.

As shown, stateful behavior can simply be implemented using a global `kvstore` variable, keeping track of state across different tool calls. We then prompt the model to solve a simple object tracking task and show the model output and tool usage over time at the bottom of snippet Fig. 7. As shown in the output, the model successfully uses the provided tools to track object state across several transactions.

Baseline Comparison We again run the same sample with unassisted Chain-Of-Thought reasoning, and find that the model again fails to solve the task correctly. Further inspection reveals that already after the first object swap, the model loses track of object state, and is unable to recover. In contrast, with stateful ACTIONS as implemented here, we observe robust state tracking across all transactions, eventually resulting in the correct solution.

5.3. Quantitative Evaluation

We also evaluate how tool use with ACTIONS relates to traditional prompting and non-interleaved tool assistance. For this, wee compare to the following baselines:

- **Unassisted Chain-Of-Thought (CoT)** As the fundemantal baseline, we consider unassisted chain-of-thought prompting Wei et al. (2022a). This baselines helps us assess the effectiveness of tool use in general, i.e. the benefit external tools provide on top of purely LLM-based reasoning.

- **Program-Aided Language Models (pal)** As a second baseline, we consider program-aided language models, i.e. the idea of prompting an LLM to generate a program which on execution computes the solution to the task (Gao et al., 2023; He-Yueya et al., 2023). This baseline helps us assess the effectiveness of inline tool use, i.e. the benefit of interleaving tool use with reasoning, as supposed to using a tool, here a program interpreter, only after the LLM has finished reasoning.

We evaluate CoT, pal and ACTIONS both in a zero-shot and 3-shot setting. When providing ACTIONS with few-shot samples, we do not provide any tool usage instructions. We prompt `text-davinci-003` to solve 50 samples each from the GSM-Hard, the ALGEBRA and Fever benchmark datasets. We report our results in Table 1.

Main Results Overall we find, that ACTIONS clearly outperforms unassisted CoT reasoning for all datasets. In particular code-based ACTIONS (i.e. line-by-line execution of LLM-generated code), appears to be beneficial to the LLM reasoning process. With fact checking dataset Fever, we even observe a 10 percentage point increase in accuracy, where we only provide a simple `wikipedia` action to the model using ACTIONS.

Tool Use Paradigms We also compare the different paradigms of tool use, namely post-hoc PAL, inline `ReAct`, inline action usage and inline code execution. We find that inline `ReAct` performs the weakest across all datasets, even though it outperforms Cot. Inline code execution performs best on artihmetic reasoning task, whereas inline action usage performs best on the more text-based fact checking task. Manual inspection reveals that code reasoning as in PAL and inline code execution elicits more formal reasoning in the form of intermediate variables and formulas, whereas inline action usage does not provoke such behavior with arithmetic reasoning. However, inline actions still exhibit the best performance on Fever, also outperforming REACT, where code-based reasoning is not applicable. We compare the concrete reasoning styles of CoT, `ReAct` and `inline_use` with factual data in App. C.

Zero-Shot, Few-Shot and PAL In a zero-shot setting, ad-hoc tool use with ACTIONS outperforms post-hoc PAL, especially with inline code execution. This is not surprising, as inline code execution is essentially an eagerly evaluated variant of PAL, which provides the LLM with intermediate results during code generation.

In a few-shot setting PAL closes this performance gap, and we observe simlar performance as inline code execution. More detailed inspection shows that PAL is a very strong baseline on arithmetic reasoning tasks, as it can leverage loops and branching to generate programmatic solutions. Inline code execution on the other hand only works with code that is executable on a line-per-line basis, which does not work well with loops, as each line may produce multiple outputs. Nonetheless we observe that ACTIONS still remains largely competitive with PAL. We compare concrete model output with PAL, `inline_use` and `inline_code` in App. B.

Discussion Overall, ACTIONS strong zero-shot performance suggests that in practice, it may not be necessary to provide few-shot samples for specific domains and tools, and that a simple zero-shot instructive prompt may be sufficient with modern LLMs. This can be a benefit in practice, where collecting or constructing suitable few-shot samples for specific domains can be cumbersome and expensive. Further, this directly enables LMs to utilize existing (well documented) Python functions.

6. Future Work

ACTIONS offers a simple and flexible framework for tool-augmented language models. Our results show that ACTIONS can be used to solve complex tasks, such as arithmetic reasoning and text-based question answering, by combining multiple tools and that a zero-shot, instruction-only approach is enough to achieve competitive performance. Nonetheless there are several limitations that we aim to focus on in future work on ACTIONS.

Typed Actions While ACTIONS supports multiple function arguments, parameter types are currently limited to primitive **`str`** and **`int`** values. Future LLM tool use, however, may entail actions with more complex parameter types like lists or entire object hierarchies. To accommodate this, we plan to extend ACTIONS to support more complex parameter types, e.g. by using type annotations in the action definition and by relying on `LMQL`'s constrained decoding (Beurer-Kellner et al., 2023) abilities, to strictly enforce parameter types. Particularly constrained decoding offers an elegant way to enforce parameter types, as it allows to restrict the LM to only generate tokens that are valid for a given type, again, without having to provide involved demonstrations, which will would be even more involved when dealing with complex parameter types.

Safety and Scope Another concern with tool-augmented LLMs is safety and action scope. State-of-the-art LLMs can still be prone to hallucinations, which in some cases can lead to an attempt to call an action that is not defined. To mitigate this, we plan to extend ACTIONS to constrain the scope of actions to a predefined scope of actions using constrained decoding. This would enable safe interfacing but also offers a much more robust and safe framework, when it comes to e.g. prompt injection (Greshake et al., 2023), as the LM can only call actions that are explicitly exposed.

Programming Language Primitives Our evaluation demonstrates how ACTIONS can be used to implement stateful tool behavior, e.g. by updating a key-value storage. Building on this, we plan to investigate object state, exposing simple object-oriented programming primitives to the LLM in addition to just function calls. This would enable more complex tool behavior, allowing the LLM to track and separate state in a semantically meaningful way, e.g., by instantiating several `Person` objects and tracking their individual state. Going beyond this, LLM-based interfacing with programming systems may also require the community to revisit the design of programming languages in general, e.g. by introducing language abstractions more suitable for LLMs and less focused on human readability.

Automatic Action Discovery LMs have shown remarkable ability in code synthesis, particularly for popular languages such as Pythons. Thus a direction we are excited to explore with ACTIONS is to equip the LM with the ability to generate further tools.

7. Conclusion

We introduced ACTIONS, a framework and programming environment to facilitate the implementation of tool-augmented language models (LMs). Our evaluation shows that our zero-shot, instruction-only approach is competitive with previous few-shot tool use paradigms, while offering a much simpler interface. In future work, we want to extend ACTIONS further, adding types, constrained tool use and by exposing more programmatic primitives to the LM, going beyond simple function calls.

References

Beurer-Kellner, L., Fischer, M., and Vechev, M. Prompting is programming. *Proceedings of the 44rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, 2023.

Bommasani, R., Hudson, D. A., Adeli, E., Altman, R. B., Arora, S., von Arx, S., Bernstein, M. S., Bohg, J., Bosselut, A., Brunskill, E., Brynjolfsson, E., Buch, S., Card, D., Castellon, R., Chatterji, N. S., Chen, A. S., Creel, K., Davis, J. Q., Demszky, D., Donahue, C., Doumbouya, M., Durmus, E., Ermon, S., Etchemendy, J., Ethayarajh, K., Fei-Fei, L., Finn, C., Gale, T., Gillespie, L., Goel, K., Goodman, N. D., Grossman, S., Guha, N., Hashimoto, T., Henderson, P., Hewitt, J., Ho, D. E., Hong, J., Hsu, K., Huang, J., Icard, T., Jain, S., Jurafsky, D., Kalluri, P., Karamcheti, S., Keeling, G., Khani, F., Khattab, O., Koh, P. W., Krass, M. S., Krishna, R., Kuditipudi, R., and et al. On the opportunities and risks of foundation models. *CoRR*, abs/2108.07258, 2021. URL https://arxiv.org/abs/2108.07258.

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. Language models are few-shot learners. In Larochelle, H., Ranzato, M., Hadsell, R., Balcan, M., and Lin, H. (eds.), *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020. URL https://proceedings.neurips.cc/paper/2020/hash/1457c0d6bfcb4967418bfb8ac142f64a-Abstract.html.

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

Chiang, W.-L., Li, Z., Lin, Z., Sheng, Y., Wu, Z., Zhang, H., Zheng, L., Zhuang, S., Zhuang, Y., Gonzalez, J. E., Stoica, I., and Xing, E. P. Vicuna: An open-source chatbot impressing gpt-4 with 90%* chatgpt quality, March 2023. URL https://lmsys.org/blog/2023-03-30-vicuna/.

Chowdhery, A., Narang, S., Devlin, J., Bosma, M., Mishra, G., Roberts, A., Barham, P., Chung, H. W., Sutton, C., Gehrmann, S., Schuh, P., Shi, K., Tsvyashchenko, S., Maynez, J., Rao, A., Barnes, P., Tay, Y., Shazeer, N., Prabhakaran, V., Reif, E., Du, N., Hutchinson, B., Pope, R., Bradbury, J., Austin, J., Isard, M., Gur-Ari, G., Yin, P., Duke, T., Levskaya, A., Ghemawat, S., Dev, S., Michalewski, H., Garcia, X., Misra, V., Robinson, K., Fedus, L., Zhou, D., Ippolito, D., Luan, D., Lim, H., Zoph, B., Spiridonov, A., Sepassi, R., Dohan, D., Agrawal, S., Omernick, M., Dai, A. M., Pillai, T. S., Pellat, M., Lewkowycz, A., Moreira, E., Child, R., Polozov, O., Lee, K., Zhou, Z., Wang, X., Saeta, B., Diaz, M., Firat, O., Catasta, M., Wei, J., Meier-Hellstern, K., Eck, D., Dean, J., Petrov, S., and Fiedel, N. Palm: Scaling language modeling with pathways. *CoRR*, abs/2204.02311, 2022. doi: 10.48550/arXiv.2204.02311. URL https://doi.org/10.48550/arXiv.2204.02311.

Cobbe, K., Kosaraju, V., Bavarian, M., Hilton, J., Nakano, R., Hesse, C., and Schulman, J. Training verifiers to solve math word problems. *CoRR*, abs/2110.14168, 2021. URL https://arxiv.org/abs/2110.14168.

Gao, L., Madaan, A., Zhou, S., Alon, U., Liu, P., Yang, Y., Callan, J., and Neubig, G. Pal: Program-aided language models. 2023.

Greshake, K., Abdelnabi, S., Mishra, S., Endres, C., Holz, T., and Fritz, M. More than you've asked for: A comprehensive analysis of novel prompt injection threats to application-integrated large language models. *arXiv preprint arXiv:2302.12173*, 2023.

He-Yueya, J., Poesia, G., Wang, R. E., and Goodman, N. D. Solving math word problems by combining language models with symbolic solvers. *arXiv preprint arXiv:2304.09102*, 2023.

Kahneman, D. *Thinking, fast and slow*. Farrar, Straus and Giroux, New York, 2011. ISBN 9780374275631 0374275637. URL https://www.amazon.de/Thinking-Fast-Slow-Daniel-Kahneman/dp/0374275637/ref=wl_it_dp_o_pdT1_nS_nC?ie=UTF8&colid=151193SNGKJT9&coliid=I3OCESLZCVDFL7.

Kojima, T., Gu, S. S., Reid, M., Matsuo, Y., and Iwasawa, Y. Large Language Models are Zero-Shot Reasoners. *ArXiv preprint*, abs/2205.11916, 2022.

Komeili, M., Shuster, K., and Weston, J. Internet-augmented dialogue generation. In Muresan, S., Nakov, P., and Villavicencio, A. (eds.), *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2022, Dublin, Ireland, May 22-27, 2022*, pp. 8460–8478. Association for Computational Linguistics, 2022. doi: 10.18653/v1/2022.acl-long.579. URL https://doi.org/10.18653/v1/2022.acl-long.579.

Lazaridou, A., Gribovskaya, E., Stokowiec, W., and Grigorev, N. Internet-augmented language models through few-shot prompting for open-domain question answering. *CoRR*, abs/2203.05115, 2022. doi: 10.48550/arXiv.2203.

05115. URL https://doi.org/10.48550/arXiv.2203.05115.

Madaan, A. and Yazdanbakhsh, A. Text and patterns: For effective chain of thought, it takes two to tango. *arXiv preprint arXiv:2209.07686*, 2022.

Mialon, G., Dessì, R., Lomeli, M., Nalmpantis, C., Pasunuru, R., Raileanu, R., Rozière, B., Schick, T., Dwivedi-Yu, J., Celikyilmaz, A., Grave, E., LeCun, Y., and Scialom, T. Augmented language models: a survey, 2023.

Nakano, R., Hilton, J., Balaji, S., Wu, J., Ouyang, L., Kim, C., Hesse, C., Jain, S., Kosaraju, V., Saunders, W., Jiang, X., Cobbe, K., Eloundou, T., Krueger, G., Button, K., Knight, M., Chess, B., and Schulman, J. Webgpt: Browser-assisted question-answering with human feedback. *CoRR*, abs/2112.09332, 2021. URL https://arxiv.org/abs/2112.09332.

OpenAI. Chatgpt plugins. URL https://openai.com/blog/chatgpt-plugins.

Rae, J. W., Borgeaud, S., Cai, T., Millican, K., Hoffmann, J., Song, H. F., Aslanides, J., Henderson, S., Ring, R., Young, S., Rutherford, E., Hennigan, T., Menick, J., Cassirer, A., Powell, R., van den Driessche, G., Hendricks, L. A., Rauh, M., Huang, P., Glaese, A., Welbl, J., Dathathri, S., Huang, S., Uesato, J., Mellor, J., Higgins, I., Creswell, A., McAleese, N., Wu, A., Elsen, E., Jayakumar, S. M., Buchatskaya, E., Budden, D., Sutherland, E., Simonyan, K., Paganini, M., Sifre, L., Martens, L., Li, X. L., Kuncoro, A., Nematzadeh, A., Gribovskaya, E., Donato, D., Lazaridou, A., Mensch, A., Lespiau, J., Tsimpoukelli, M., Grigorev, N., Fritz, D., Sottiaux, T., Pajarskas, M., Pohlen, T., Gong, Z., Toyama, D., de Masson d'Autume, C., Li, Y., Terzi, T., Mikulik, V., Babuschkin, I., Clark, A., de Las Casas, D., Guy, A., Jones, C., Bradbury, J., Johnson, M. J., Hechtman, B. A., Weidinger, L., Gabriel, I., Isaac, W., Lockhart, E., Osindero, S., Rimell, L., Dyer, C., Vinyals, O., Ayoub, K., Stanway, J., Bennett, L., Hassabis, D., Kavukcuoglu, K., and Irving, G. Scaling language models: Methods, analysis & insights from training gopher. *CoRR*, abs/2112.11446, 2021. URL https://arxiv.org/abs/2112.11446.

Schick, T., Dwivedi-Yu, J., Dessì, R., Raileanu, R., Lomeli, M., Zettlemoyer, L., Cancedda, N., and Scialom, T. Toolformer: Language models can teach themselves to use tools. *CoRR*, abs/2302.04761, 2023. doi: 10.48550/arXiv.2302.04761. URL https://doi.org/10.48550/arXiv.2302.04761.

Shuster, K., Xu, J., Komeili, M., Ju, D., Smith, E. M., Roller, S., Ung, M., Chen, M., Arora, K., Lane, J., Behrooz, M., Ngan, W., Poff, S., Goyal, N., Szlam, A., Boureau, Y., Kambadur, M., and Weston, J. Blenderbot 3: a deployed conversational agent that continually learns to responsibly engage. *CoRR*, abs/2208.03188, 2022. doi: 10.48550/arXiv.2208.03188. URL https://doi.org/10.48550/arXiv.2208.03188.

Srivastava, A., Rastogi, A., Rao, A., Shoeb, A. A. M., Abid, A., Fisch, A., Brown, A. R., Santoro, A., Gupta, A., Garriga-Alonso, A., Kluska, A., Lewkowycz, A., Agarwal, A., Power, A., Ray, A., Warstadt, A., Kocurek, A. W., Safaya, A., Tazarv, A., Xiang, A., Parrish, A., Nie, A., Hussain, A., Askell, A., Dsouza, A., Rahane, A., Iyer, A. S., Andreassen, A., Santilli, A., Stuhlmüller, A., Dai, A. M., La, A., Lampinen, A. K., Zou, A., Jiang, A., Chen, A., Vuong, A., Gupta, A., Gottardi, A., Norelli, A., Venkatesh, A., Gholamidavoodi, A., Tabassum, A., Menezes, A., Kirubarajan, A., Mullokandov, A., Sabharwal, A., Herrick, A., Efrat, A., Erdem, A., Karakas, A., and et al. Beyond the imitation game: Quantifying and extrapolating the capabilities. *ArXiv preprint*, abs/2206.04615, 2022.

Thoppilan, R., Freitas, D. D., Hall, J., Shazeer, N., Kulshreshtha, A., Cheng, H., Jin, A., Bos, T., Baker, L., Du, Y., Li, Y., Lee, H., Zheng, H. S., Ghafouri, A., Menegali, M., Huang, Y., Krikun, M., Lepikhin, D., Qin, J., Chen, D., Xu, Y., Chen, Z., Roberts, A., Bosma, M., Zhou, Y., Chang, C., Krivokon, I., Rusch, W., Pickett, M., Meier-Hellstern, K. S., Morris, M. R., Doshi, T., Santos, R. D., Duke, T., Soraker, J., Zevenbergen, B., Prabhakaran, V., Diaz, M., Hutchinson, B., Olson, K., Molina, A., Hoffman-John, E., Lee, J., Aroyo, L., Rajakumar, R., Butryna, A., Lamm, M., Kuzmina, V., Fenton, J., Cohen, A., Bernstein, R., Kurzweil, R., Aguera-Arcas, B., Cui, C., Croak, M., Chi, E. H., and Le, Q. Lamda: Language models for dialog applications. *CoRR*, abs/2201.08239, 2022. URL https://arxiv.org/abs/2201.08239.

Thorne, J., Vlachos, A., Christodoulopoulos, C., and Mittal, A. Fever: a large-scale dataset for fact extraction and verification. *arXiv preprint arXiv:1803.05355*, 2018.

Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M., Lacroix, T., Rozière, B., Goyal, N., Hambro, E., Azhar, F., Rodriguez, A., Joulin, A., Grave, E., and Lample, G. Llama: Open and efficient foundation language models. *CoRR*, abs/2302.13971, 2023. doi: 10.48550/arXiv.2302.13971. URL https://doi.org/10.48550/arXiv.2302.13971.

Wei, J., Wang, X., Schuurmans, D., Bosma, M., Chi, E., Le, Q., and Zhou, D. Chain of thought prompting elicits reasoning in large language models. *arXiv preprint arXiv:2201.11903*, 2022a.

Wei, J., Wang, X., Schuurmans, D., Bosma, M., Chi, E. H., Le, Q., and Zhou, D. Training language models to follow instructions with human feedback. *ArXiv preprint*, abs/2203.02155, 2022b.

Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K., and Cao, Y. React: Synergizing reasoning and acting in language models. *CoRR*, abs/2210.03629, 2022. doi: 10.48550/arXiv.2210.03629. URL `https://doi.org/10.48550/arXiv.2210.03629`.

A. Program-Aided Language Models (PALs) in `LMQL`

Since `LMQL` is a superset of Python, we can easily implement any post-hoc tool use like PAL (Gao et al., 2023) using the `exec` function, as shown in Listing 1.

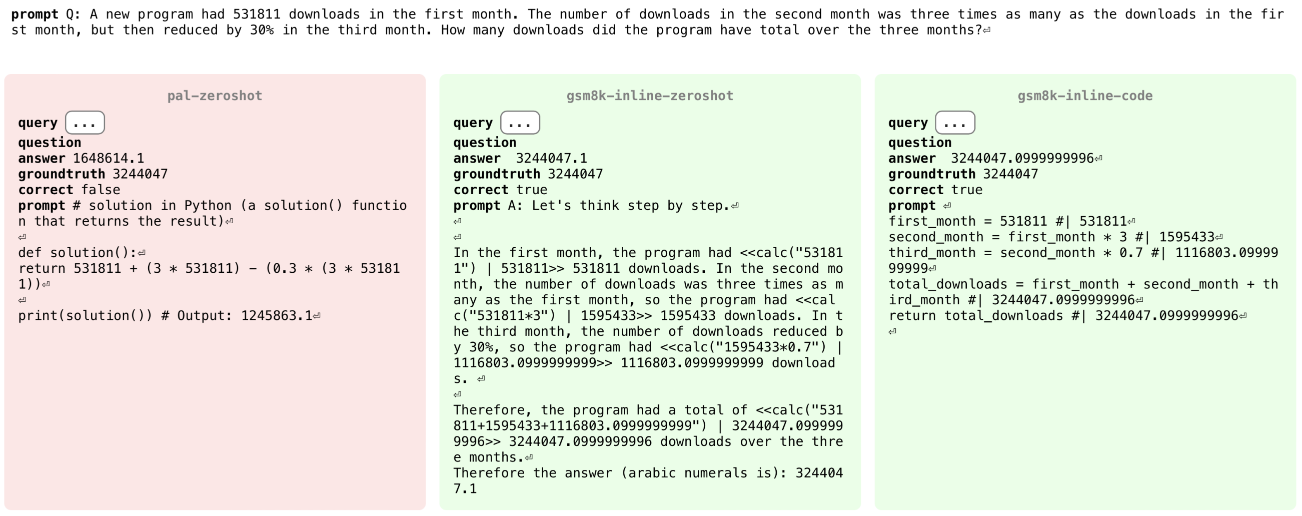
```
argmax
    """Q: Eliza's rate per hour for the first 40 hours she works each week is USD 1616598. She also receives
    an overtime pay of 1.2 times her regular hourly rate. If Eliza worked for 45 hours this week, how much
    are her earnings for this week?

    # solution in Python (a solution() function that returns the result)
    [CODE]
    """
    interpreter_environment = {}
    exec(CODE, interpreter_environment)
    solution_fct = interpreter_environment["solution"]
    "Execution Result: {solution_fct()}"
from
    "openai/text-davinci-003"
where
    STOPS_AT(CODE, "\n\n")
```

Listing 1: Implementing program-aided LLM reasoning in `LMQL`.

B. Comparing PAL, `inline_use` and `inline_code` with arithmetic reasoning

Below we compare the model output for the same question with PAL, `inline_use` and `inline_code`. We can see that both ACTIONS based methods leverage tools and code to obtain correct intermediate results and eventually the correct answer.

Post-hoc PAL on the other hand produces a full program without any intermediate results, which in this case leads to the wrong answer. In PAL we can also observe that the model attempts to perform inline execution of the code, but fails to do so correctly (the model predicted output is very different from the actual output when the `solution` function is executed). This examples demonstrates how model reasoning can benefit greatly from inline tool use and that intermediate results indeed can help the model to reason better.



```
prompt Q: A new program had 531811 downloads in the first month. The number of downloads in the second month was three times as many as the downloads in the fir
st month, but then reduced by 30% in the third month. How many downloads did the program have total over the three months?⏎
```

pal-zeroshot

```
query ...
question
answer 1648614.1
groundtruth 3244047
correct false
prompt # solution in Python (a solution() functio
n that returns the result)⏎
⏎
def solution():⏎
return 531811 + (3 * 531811) - (0.3 * (3 * 53181
1))⏎
⏎
print(solution()) # Output: 1245863.1⏎
```

gsm8k-inline-zeroshot

```
query ...
question
answer  3244047.1
groundtruth 3244047
correct true
prompt A: Let's think step by step.⏎
⏎
⏎
In the first month, the program had <<calc("53181
1") | 531811>> 531811 downloads. In the second mo
nth, the number of downloads was three times as m
any as the first month, so the program had <<cal
c("531811*3") | 1595433>> 1595433 downloads. In t
he third month, the number of downloads reduced b
y 30%, so the program had <<calc("1595433*0.7") |
1116803.0999999999>> 1116803.0999999999 download
s. ⏎
⏎
Therefore, the program had a total of <<calc("531
811+1595433+1116803.0999999999") | 3244047.099999
9996>> 3244047.0999999996 downloads over the thre
e months.⏎
Therefore the answer (arabic numerals is): 324404
7.1
```

gsm8k-inline-code

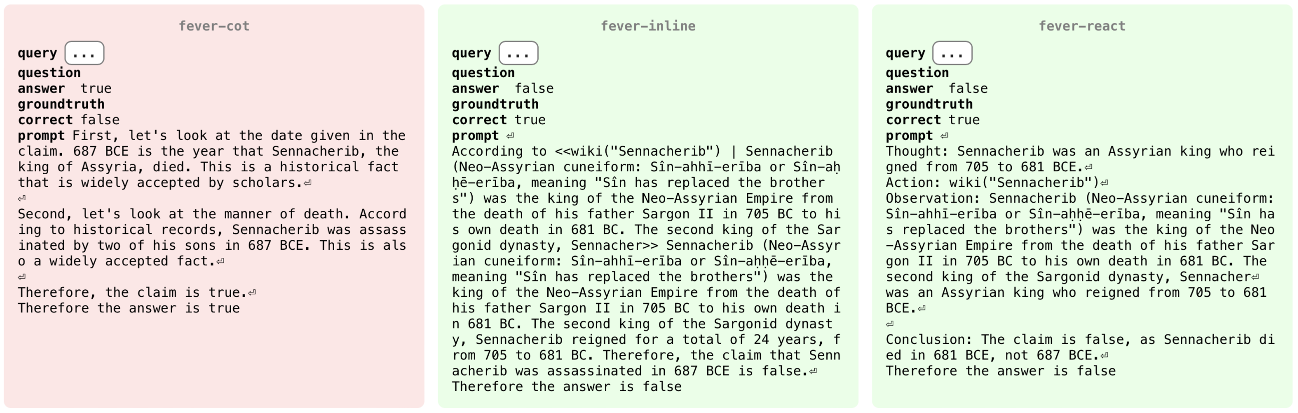
```
query ...
question
answer  3244047.0999999996⏎
groundtruth 3244047
correct true
prompt ⏎
first_month = 531811 #| 531811⏎
second_month = first_month * 3 #| 1595433⏎
third_month = second_month * 0.7 #| 1116803.09999
99999⏎
total_downloads = first_month + second_month + th
ird_month #| 3244047.0999999996⏎
return total_downloads #| 3244047.0999999996⏎
⏎
```

C. Comparing Chain-Of-Thought, `ReAct` and `inline_use` with text-based question answering

Below, we compare the model output for the same question with Chain-Of-Thought, `ReAct` and `inline_use` on the fact checking dataset Fever (Thorne et al., 2018). We can see that both ACTIONS based methods leverage tools and code to obtain correct intermediate results and eventually the correct answer.

Chain-Of-Thought prompting on the other hand hallucinates background information on the provided claim, simply assuming the correctness of the evidently wrong time frame of 687 BCE, as opposed to the correct time frame of 681 BCE.

```
question On June 2017, the following claim was made: Sennacherib was assassinated in 687 BCE.⏎
Q: Was this claim true or false?
groundtruth false
```



```
fever-cot

query ...
question
answer  true
groundtruth
correct false
prompt First, let's look at the date given in the
claim. 687 BCE is the year that Sennacherib, the
king of Assyria, died. This is a historical fact
that is widely accepted by scholars.⏎
⏎
Second, let's look at the manner of death. Accord
ing to historical records, Sennacherib was assass
inated by two of his sons in 687 BCE. This is als
o a widely accepted fact.⏎
⏎
Therefore, the claim is true.⏎
Therefore the answer is true
```

```
fever-inline

query ...
question
answer  false
groundtruth
correct true
prompt ⏎
According to <<wiki("Sennacherib") | Sennacherib
(Neo-Assyrian cuneiform: Sîn-ahhī-erība or Sîn-aḥ
ḥē-erība, meaning "Sîn has replaced the brother
s") was the king of the Neo-Assyrian Empire from
the death of his father Sargon II in 705 BC to hi
s own death in 681 BC. The second king of the Sar
gonid dynasty, Sennacher>> Sennacherib (Neo-Assyr
ian cuneiform: Sîn-ahhī-erība or Sîn-aḥḥē-erība,
meaning "Sîn has replaced the brothers") was the
king of the Neo-Assyrian Empire from the death of
his father Sargon II in 705 BC to his own death i
n 681 BC. The second king of the Sargonid dynast
y, Sennacherib reigned for a total of 24 years, f
rom 705 to 681 BC. Therefore, the claim that Senn
acherib was assassinated in 687 BCE is false.⏎
Therefore the answer is false
```

```
fever-react

query ...
question
answer  false
groundtruth
correct true
prompt ⏎
Thought: Sennacherib was an Assyrian king who rei
gned from 705 to 681 BCE.⏎
Action: wiki("Sennacherib")⏎
Observation: Sennacherib (Neo-Assyrian cuneiform:
Sîn-ahhī-erība or Sîn-aḥḥē-erība, meaning "Sîn ha
s replaced the brothers") was the king of the Neo
-Assyrian Empire from the death of his father Sar
gon II in 705 BC to his own death in 681 BC. The
second king of the Sargonid dynasty, Sennacher⏎
was an Assyrian king who reigned from 705 to 681
BCE.⏎
⏎
Conclusion: The claim is false, as Sennacherib di
ed in 681 BCE, not 687 BCE.⏎
Therefore the answer is false
```