EXECUTABLE CODE ACTIONS ELICIT BETTER LLM AGENTS

Xingyao Wang[1], **Yangyi Chen**[1], **Lifan Yuan**[1], **Yizhe Zhang**[2], **Yunzhu Li**[1], **Hao Peng**[1], **Heng Ji**[1]
[1]University of Illinois Urbana-Champaign, [2]Apple
[1]`{xingyao6,yangyic3,yunzhuli,haopeng,hengji}@illinois.edu`
[2]`yizhe_zhang@apple.com`

ABSTRACT

Large Language Model (LLM) agents, capable of performing a broad range of actions, such as invoking tools and controlling robots, show great potential in tackling real-world challenges. LLM agents are typically prompted to produce actions by generating JSON or text in a pre-defined format, which is usually limited by constrained action space (e.g., the scope of pre-defined tools) and restricted flexibility (e.g., inability to compose multiple tools). This work proposes to use executable Python **code** to consolidate LLM agents' **act**ions into a unified action space (`CodeAct`). Integrated with a Python interpreter, `CodeAct` can execute code actions and dynamically revise prior actions or emit new actions upon new observations through multi-turn interactions. Our extensive analysis of 17 LLMs on API-Bank and a newly curated benchmark shows that `CodeAct` outperforms widely used alternatives (up to 20% higher success rate). The encouraging performance of `CodeAct` motivates us to build an open-source LLM agent that interacts with environments by executing interpretable code and collaborates with users using natural language. To this end, we collect an instruction-tuning dataset CodeActInstruct that consists of 7k multi-turn interactions using `CodeAct`. We show that it can be used with existing data to improve models in agent-oriented tasks without compromising their general capability. CodeActAgent, finetuned from Llama2 and Mistral, is integrated with Python interpreter and uniquely tailored to perform sophisticated tasks (e.g., model training) using existing libraries and autonomously self-debug. [1]

1 INTRODUCTION

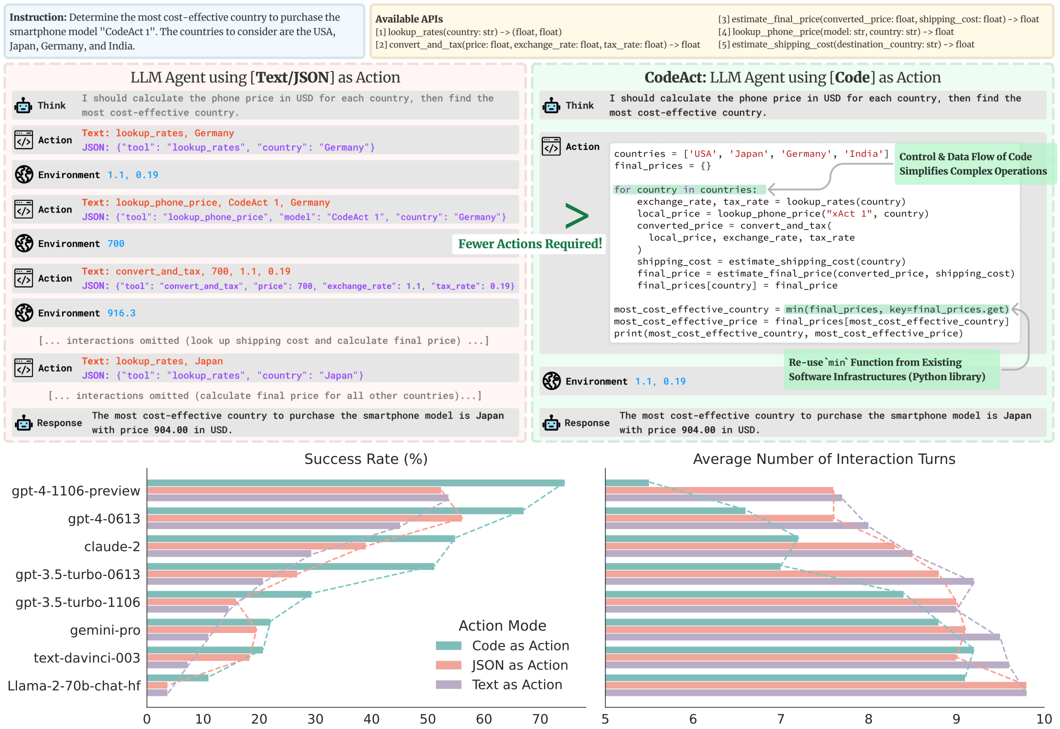
Large Language Models (LLMs) have emerged as a pivotal breakthrough in natural language processing (NLP). When augmented with *action* modules that allow access to APIs, their action space expands beyond conventional text processing, allowing LLMs to acquire capabilities such as tool invocation and memory management (Mialon et al., 2023; Schick et al., 2023) and venture into real-world tasks such as controlling robots (Ahn et al., 2022; Huang et al., 2023; Ma et al., 2023) and performing scientific experiments (Bran et al., 2023).

We inquire: *how to effectively expand LLM agents' action space for solving complex real-world problems?* Much existing research has examined using text (Yao et al., 2022b; Park et al., 2023, *inter alia*) or JSON (Qin et al., 2023b; Chase, 2022, *inter alia*) to produce actions (e.g., tool uses in Fig. 1 top left). However, both methods typically suffer from constrained scope of action spaces (actions are usually tailored for specific tasks) and restricted flexibility (e.g., inability to compose multiple tools in a single action). As an alternative approach, several work (Liang et al., 2022; Singh et al., 2023; Wang et al., 2023a) demonstrate the potential of using LLMs to generate code to control robots or game characters. However, they typically rely on pre-specified control primitives and hand-engineered prompts and, more importantly, struggle to dynamically adjust or emit actions based on new environmental observation and feedback.

This work proposes `CodeAct`, a general-purpose framework that allows LLMs to generate executable Python **code** as **act**ions (Fig. 1 top right). `CodeAct` is designed to handle a variety of applications and comes with unique advantages:

---

[1] The code, data, model, and demo are available at `https://github.com/xingyaoww/code-act`.



Instruction: Determine the most cost-effective country to purchase the smartphone model "CodeAct 1". The countries to consider are the USA, Japan, Germany, and India.

Available APIs
[1] lookup_rates(country: str) -> (float, float)
[2] convert_and_tax(price: float, exchange_rate: float, tax_rate: float) -> float
[3] estimate_final_price(converted_price: float, shipping_cost: float) -> float
[4] lookup_phone_price(model: str, country: str) -> float
[5] estimate_shipping_cost(destination_country: str) -> float

LLM Agent using [**Text/JSON**] as Action

```
Think        I should calculate the phone price in USD for each country, then find the
             most cost-effective country.
Action       Text: lookup_rates, Germany
             JSON: {"tool": "lookup_rates", "country": "Germany"}
Environment  1.1, 0.19
Action       Text: lookup_phone_price, CodeAct 1, Germany
             JSON: {"tool": "lookup_phone_price", "model": "CodeAct 1", "country": "Germany"}
Environment  700
Action       Text: convert_and_tax, 700, 1.1, 0.19
             JSON: {"tool": "convert_and_tax", "price": 700, "exchange_rate": 1.1, "tax_rate": 0.19}
Environment  916.3
     [... interactions omitted (look up shipping cost and calculate final price) ...]
Action       Text: lookup_rates, Japan
             JSON: {"tool": "lookup_rates", "country": "Japan"}
      [... interactions omitted (calculate final price for all other countries)...]
Response     The most cost-effective country to purchase the smartphone model is Japan
             with price 904.00 in USD.
```

>

Fewer Actions Required!

CodeAct: LLM Agent using [**Code**] as Action

```
Think        I should calculate the phone price in USD for each country, then find the
             most cost-effective country.
Action
countries = ['USA', 'Japan', 'Germany', 'India']
final_prices = {}

for country in countries:
    exchange_rate, tax_rate = lookup_rates(country)
    local_price = lookup_phone_price("xAct 1", country)
    converted_price = convert_and_tax(
      local_price, exchange_rate, tax_rate
    )
    shipping_cost = estimate_shipping_cost(country)
    final_price = estimate_final_price(converted_price, shipping_cost)
    final_prices[country] = final_price

most_cost_effective_country = min(final_prices, key=final_prices.get)
most_cost_effective_price = final_prices[most_cost_effective_country]
print(most_cost_effective_country, most_cost_effective_price)

Environment  1.1, 0.19
Response     The most cost-effective country to purchase the smartphone model is Japan
             with price 904.00 in USD.
```

Control & Data Flow of Code Simplifies Complex Operations

Re-use `min` Function from Existing Software Infrastructures (Python library)

Figure 1: Comparison between `CodeAct` and Text / JSON as action. **(top)** Illustrative example comparing different actions. **(bottom)** Quantitative results on M^{3}ToolEval (§2.3).

(1) Integrated with a Python interpreter, `CodeAct` can execute code actions and *dynamically adjust prior actions or emit new action* based on observations (e.g., code execution results) it receives through multiple turns of interactions.
(2) Code actions allow LLM to leverage existing *software packages*. `CodeAct` can use readily available Python packages for an expanded action space instead of hand-crafted task-specific tools (Yuan et al., 2023; Shen et al., 2023). It also allows LLM to use automated feedback (e.g., error messages) implemented in most software to improve task-solving by self-debugging its generated code (Chen et al., 2023b; Wang et al., 2023d).
(3) *Code data* is widely used in pre-training today's LLMs (Yang et al., 2024b). These models are already familiar with structured programming languages, allowing cost-effective adoption of `CodeAct`.
(4) Compared to JSON and text with a pre-defined format, code inherently supports *control and data flow*, allowing for the storage of intermediate results as variables for reuse and the composition of multiple tools to perform complex logical operations (e.g., if-statements, for-loops) with *one* piece of code, thereby unlocking LLMs' potential to tackle complex tasks by leveraging its pre-trained knowledge of programming. In Fig. 1, an LLM using with `CodeAct` (top right) can apply the same sequence of tools (e.g., passing one tool's output as input to another tool using the data flow feature) to *all* inputs through for-loops (i.e., control flow feature) with *one* action; while text or JSON have to take action for every input (top left).

Our extensive experiments with 17 LLMs (including both open-source and proprietary ones) confirm the above benefits (3 & 4) of `CodeAct`. To demonstrate benefit (3), our first experiment (§2.2) compares `CodeAct` to baselines on basic tasks involving *atomic tool use* (i.e., only one tool is used per action), ablating the control and data flow advantage offered by `CodeAct`. The results show that, for most LLMs, `CodeAct` achieves comparable or better performance than the baselines. `CodeAct`'s performance gains are more prominent on complex tasks, as demonstrated in our second experiment (benefit 4). We curate a new benchmark consisting of 82 human-curated tasks that typically require **m**ultiple calls to **m**ultiple tools in **m**ulti-turn interactions (M^{3}ToolEval; §2.3). Problems in this benchmark often require intricate coordination and composition of multiple tools. With its strengths in control and data flow, `CodeAct` achieves up to a 20% absolute improvement over baselines on the success rate of solving the problems while requiring up to 30% fewer actions. These performance gains widen as the capabilities of the LLMs increase (Fig. 1 bottom).

The promising performance of `CodeAct` motivates an open-source LLM agent that can effectively act through `CodeAct`, and collaborate with humans through natural language. To this end, we collect

Table 1: The benefit of `CodeAct` compared to using Text/JSON for LLM action.

	`CodeAct` for LLM action	**JSON or Text for LLM action**
Availability of Data	✓Large quantity of code available[1] for pre-training	✗Data curation required for particular format
Complex Operation (e.g., looping, composition of multiple tools)	✓Natively supported via control and data flow	✗Requires careful engineering if feasible (e.g., define new tools to mimic if-statement)
Availability of Tools	✓Can directly use existing software packages[2]	✗Requires human effort to curate tools from scratch or existing software
Automated Feedback	✓Feedback mechanism[3] (e.g., traceback) is already implemented as an infrastructure for most programming languages	✗Requires human effort to provide feedback or re-route feedback from the underlying programming language used to implement the tools

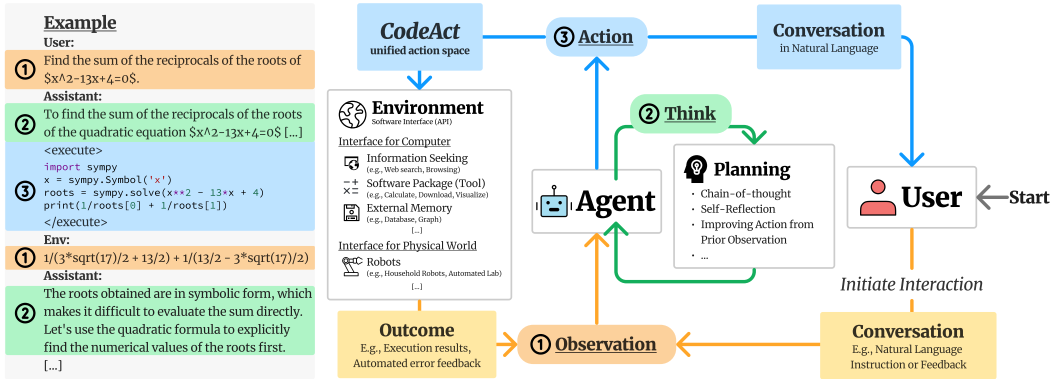
[1] Including code demonstrating useful behaviors for LLM agents (e.g., task decomposition, coordination of multiple function calls to different tools).
[2] Human-written Python packages covering a wide range of applications are available on `https://pypi.org/`.
[3] For example, in Python, errors and exceptions (`https://docs.python.org/3/tutorial/errors.html`) are available. Most software provides error messages in natural language to help human programmers debug their code. `CodeAct` enables LLM to use them directly.

an instruction-tuning dataset CodeActInstruct consisting of 7k high-quality multi-turn interaction trajectories with `CodeAct` (§3.1). CodeActInstruct is motivated by a general agent framework consisting of agent, user, and environments (Fig. 2) and focuses on agent-environment interactions with the computer (information seeking, software package use, external memory) and the physical world (robot planning). On CodeActInstruct, we perform careful data selection to promote the capability of improving from multi-turn interaction (e.g., self-debug). We show that CodeActInstruct can be used with commonly used instruction tuning data to improve the models' performance in agent tasks without compromising their general capabilities (e.g., knowledge-based QA, coding, instruction following, §3.2). Our model, dubbed CodeActAgent, is finetuned from LLaMA-2 (Touvron et al., 2023) and Mistral-7B (Jiang et al., 2023) and improves on out-of-domain agent tasks with not only `CodeAct`, but also text action in a pre-defined format (§3.2).

`CodeAct` can further benefit from multi-turn interactions and existing software (benefit 1 & 2, §2.4). As shown in Fig. 3, CodeActAgent, designed for seamless integration with Python, can carry out sophisticated tasks (e.g., model training, data visualization) using existing Python packages. Error messages from the environment further enable it to rectify errors autonomously through self-debugging in multi-turn interaction. Thanks to LLM's extensive programming knowledge acquired during pre-training, these are achieved without needing in-context demonstrations, reducing the human efforts for adapting CodeActAgent to different tasks.

2 CODEACT MAKES LLMS BETTER AGENTS

In this section, we first describe `CodeAct` framework (§2.1) and provide empirical evidence that supports the choice of `CodeAct`. We focus on Python as the programming language for `CodeAct` due to its popularity (ranked top-1 at TIOBE Index (2024)) and numerous open-source packages. We aim to answer several research questions (RQs) using 17 off-the-shelf LLMs. In §2.2, we examine RQ1: Does LLMs' familiarity with code due to a large amount of code pre-training data bring `CodeAct` advantages over text and JSON? We discuss RQ2 in §2.3: Does `CodeAct` benefit from Python's innate control and data flow feature in complex problems? Finally, as an additional benefit, we discuss how using `CodeAct` further enhances LLM agents by enabling multi-turn interactions and allowing them to access existing software in §2.4 and Fig. 3.



Example

User:

① Find the sum of the reciprocals of the roots of $x^2-13x+4=0$.

Assistant:

② To find the sum of the reciprocals of the roots of the quadratic equation $x^2-13x+4=0$ [...]

③ <execute>

```
import sympy
x = sympy.Symbol('x')
roots = sympy.solve(x**2 - 13*x + 4)
print(1/roots[0] + 1/roots[1])
```

</execute>

Env:

① 1/(3*sqrt(17)/2 + 13/2) + 1/(13/2 - 3*sqrt(17)/2)

Assistant:

② The roots obtained are in symbolic form, which makes it difficult to evaluate the sum directly. Let's use the quadratic formula to explicitly find the numerical values of the roots first.

[...]

Figure 2: General agent multi-turn interaction framework that describes the role of `CodeAct` and motivates the construction of our data mixture. CodeActInstruct focuses on the *agent-environment* interactions and specifically filters for the self-improved *planning* behavior, while general conversation data we include focuses on *agent-user* interaction (§3.1).

2.1 WHAT IS `CodeAct`?

In Fig. 2, we first introduce a general multi-turn interaction framework for LLM agents' real-world usage that considers three roles (Yang et al., 2024c): *agent*, *user*, and *environment*. We define *interaction* as the information exchange between the agent and an external entity (user or environment). For each turn of interaction, the agent receives an *observation* (input) either from the user (e.g., natural language instruction) or the environment (e.g., code execution result), optionally planning for its action through chain-of-thought (Wei et al., 2022), and emits an *action* (output) to either user in natural language or the environment. `CodeAct` employs Python code to consolidate all *actions* for *agent-environment interaction*. In `CodeAct`, each emitted *action* to the environment is a piece of Python code, and the agent will receive outputs of code execution (e.g., results, errors) as *observation*. We include an example prompt of `CodeAct` in §E.

2.2 `CodeAct` SHOWS THE PROMISE AS A STRONG TOOL USE FRAMEWORK

In this section, we perform a controlled experiment to understand which format (text, JSON, `CodeAct`) is more likely to lead an LLM to generate correct *atomic* tool calls. The performance in this experiment reflects LLM's familiarity with the corresponding format. We hypothesize that using `CodeAct` to call tools is a more natural way to use tools for the models, which typically have extensive exposure to *code data* during their training.

Setup. We re-purpose API-Bank (Li et al., 2023) and test LLMs' API-calling performance, comparing `CodeAct`, JSON, and text actions. For each evaluation instance, we instruct LLM to generate *one atomic* tool call in the format of a Python function call, JSON object, or text expression in a pre-defined format. A concrete example is shown in Tab. A.6. We use API-Bank's level-1 instructions and the provided toolset. To evaluate API-calling, we follow their *correctness* metric, matching the ground-truth API outputs with the actual model-generated API's execution outputs.

Results. We present results in Tab. 2. For most LLMs, `CodeAct` achieves comparable or better performance even in atomic actions (the simplistic tool use scenario) where its control and data flow strengths are ablated. Compared to closed-source LLMs, `CodeAct`'s improvements are more prominent in open-source models. Furthermore, code data is usually more accessible for fine-tuning open-source LLMs than the specialized JSON or text tool-calling format. Although JSON is consistently weaker than other approaches for open-source models, it achieves decent performance with closed-source LLMs, indicating that these closed-source models may have gone through targeted fine-tuning toward their JSON capabilities. These results suggest optimizing for `CodeAct` is a better route for open-source LLMs than alternatives to improve their tool-use capabilities, as they already show good initial `CodeAct` capability due to extensive exposure to code data during pre-training.

Table 2: Atomic API call correctness on API-Bank with different action format. The best performance is **bolded**, and the second-best is underlined.

Format of Action	Correctness (%, ↑) `CodeAct`	 JSON	 Text
	Open-source LLMs		
`CodeLlama-7b-Instruct-hf`	12.5	12.0	**17.0**
`CodeLlama-13b-Instruct-hf`	11.8	7.8	**14.0**
`CodeLlama-34b-Instruct-hf`	**17.3**	12.0	16.8
`Llama-2-7b-chat-hf`	**28.8**	11.3	25.8
`Llama-2-13b-chat-hf`	**38.1**	8.5	37.3
`Llama-2-70b-chat-hf`	35.6	14.3	**37.6**
`Mistral-7B-Instruct-v0.1`	2.5	2.3	**3.0**
`lemur-70b-chat-v1`	**58.6**	46.6	56.1
	Closed-source LLMs		
`claude-2`	**76.7**	59.4	73.7
`claude-instant-1`	**75.2**	64.9	73.2
`gemini-pro`	70.4	**73.2**	71.2
`gpt-3.5-turbo-0613`	**74.4**	73.9	73.4
`gpt-3.5-turbo-1106`	75.4	**78.4**	73.4
`gpt-4-0613`	75.4	**82.0**	74.4
`gpt-4-1106-preview`	76.7	**82.7**	73.4
`text-davinci-002`	**69.2**	59.6	57.4
`text-davinci-003`	75.4	**76.9**	69.7
Frequency of Best-Performing Format ↑			
Open-source	**4**	0	4
Closed-source	4	**5**	0
Overall	**8**	5	4

Table 3: Success rates (higher the better) and average turns required per instance (lower the better) on M^3ToolEval. The best results for each model are **bolded**, and the second-best ones are underlined.

Format of Action	Success Rate (%, ↑) `CodeAct`	 JSON	 Text	Avg. Turns (↓) `CodeAct`	 JSON	 Text
	Open-source LLMs					
`CodeLlama-7b-Instruct-hf`	**4.9**	2.4	2.4	**9.7**	9.9	9.9
`CodeLlama-13b-Instruct-hf`	**4.9**	**4.9**	**4.9**	9.8	9.8	**9.7**
`CodeLlama-34b-Instruct-hf`	**2.4**	0.0	0.0	**9.9**	10.0	10.0
`Llama-2-7b-chat-hf`	0.0	1.2	**2.4**	**8.9**	9.5	9.6
`Llama-2-13b-chat-hf`	**0.0**	**0.0**	**0.0**	**9.7**	10.0	10.0
`Llama-2-70b-chat-hf`	**11.0**	3.7	3.7	**9.1**	9.8	9.8
`Mistral-7B-Instruct-v0.1`	0.0	**3.7**	1.2	10.0	**9.8**	9.9
`lemur-70b-chat-v1`	13.4	**15.9**	12.2	**9.1**	9.3	9.4
	Closed-source LLMs					
`claude-2`	**54.9**	39.0	29.3	**7.2**	8.3	8.5
`claude-instant-1`	20.7	**31.7**	24.4	8.8	**8.6**	8.9
`gemini-pro`	**22.0**	19.5	11.0	**8.8**	9.1	9.5
`gpt-3.5-turbo-0613`	**51.2**	26.8	20.7	**7.0**	8.8	9.2
`gpt-3.5-turbo-1106`	**29.3**	15.9	14.6	**8.4**	9.0	9.0
`gpt-4-0613`	**67.1**	56.1	45.1	**6.6**	7.6	8.0
`gpt-4-1106-preview`	**74.4**	52.4	53.7	**5.5**	7.6	7.7
`text-davinci-002`	4.9	4.9	**8.5**	9.7	9.8	**9.6**
`text-davinci-003`	**20.7**	18.3	7.3	9.2	**9.0**	9.6
Frequency of Best-performing Format ↑						
Open-source	**5**	4	3	**6**	1	1
Closed-source	**7**	1	1	**6**	2	1
Overall	**12**	5	4	**12**	3	2

2.3 CODEACT GETS MORE DONE WITH FEWER INTERACTIONS

In this section, we investigate whether LLM agents can benefit from the control and data flow of code on problems that require complex patterns of tool use.

M^{3}ToolEval. As shown in Tab. A.7, to the best of our knowledge, no existing tool-use benchmarks contain complex tasks requiring the composition of multiple tools while supporting evaluating different action formats. Hence, we curate a benchmark M^{3}ToolEval to fill this gap, which evaluates LLMs' capabilities in solving complex tasks that typically require **m**ultiple calls to **m**ultiple tools in **m**ulti-turn interactions. It contains 82 human-curated instances, spanning tasks including web browsing, finance, travel itinerary planning, science, and information processing. Each domain is accompanied by a unique set of manually crafted tools. We intentionally keep the prompt simple (examples in §F) and avoid providing any demonstration to test the LLM's zero-shot ability to use tools, similar to how a novice user without knowledge of few-shot prompting would use the model.

Setup. We allow the model to generate fully functional Python code that enables control and data flow (e.g., if-statement, for-loop). We follow the action format for JSON and text described in Tab. A.6. Within each turn, the model can either emit an *action* or propose an *answer* to be verified by an exact match with the ground-truth solution. The interaction will terminate when a maximum of 10 interaction turns are reached or a correct solution has been submitted, similar to Wang et al. (2023e).

Metric. We measure the *success rate* by calculating the percentage of the model proposed answers that match the ground-truth solutions. We also include the *avg. turns* metric: the average number of turns on all evaluated instances.

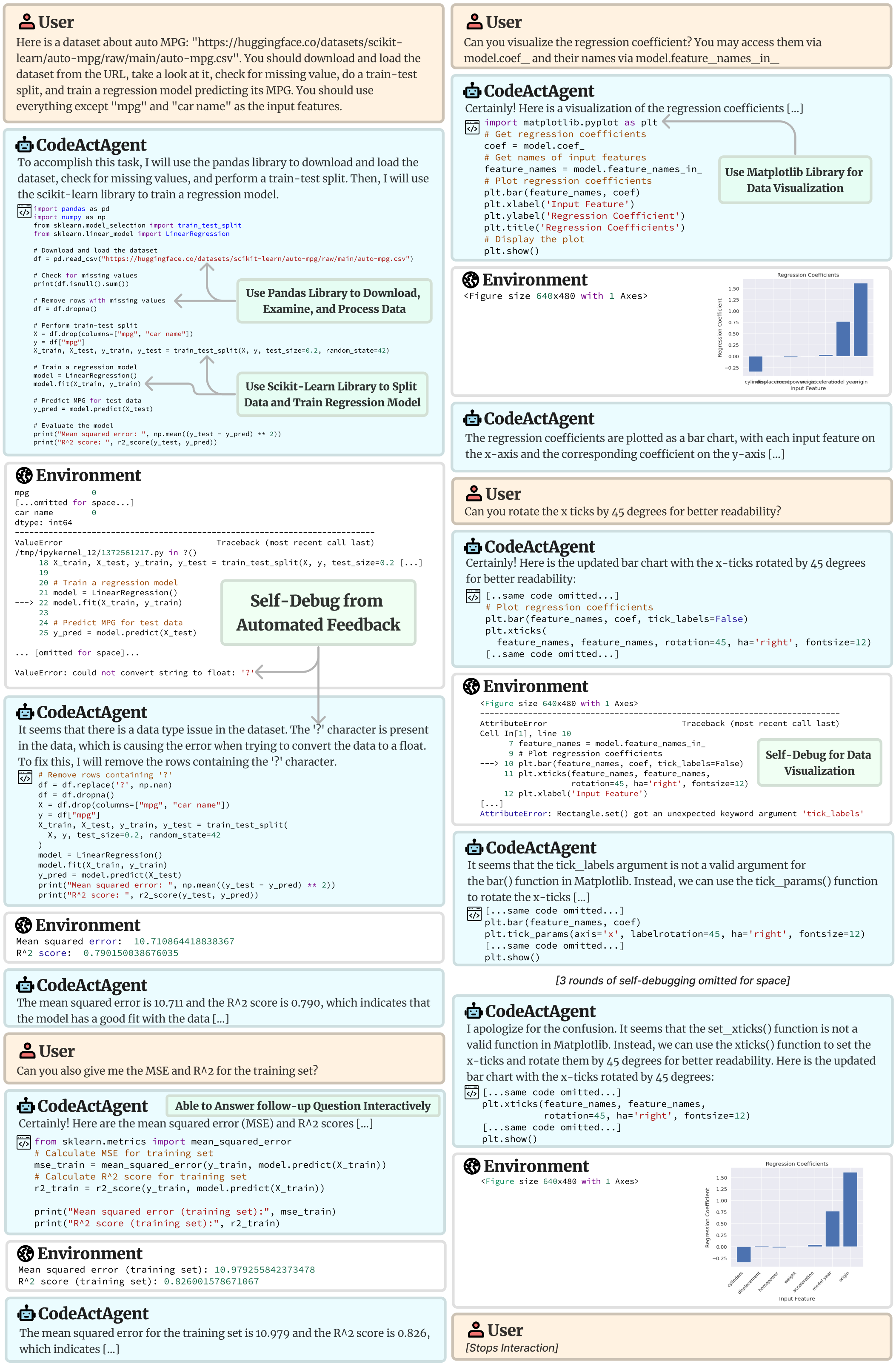
Quantitative Results on M^{3}ToolEval. We include full results in Tab. 3 and a subset of results for visualization in Fig. 1. `CodeAct` generally has a higher task success rate (12 out of 17 evaluated LLMs), similar to the trend in §2.2. Moreover, using `CodeAct` requires a lower average number of turns (12 out of 17 evaluated LLMs). For example, the best model `gpt-4-1106-preview` achieves a 20.7% absolute improvement compared to the next best action format (text) while requiring 2.1 fewer interaction turns on average. However, there is still a significant gap in terms of absolute `CodeAct` performance between open- and closed-source LLMs as the best open-source model achieving 13.4% while the best closed-source model `gpt-4-1106-preview` 74.4%. This is potentially due to open-source models' weak task-solving capability and inability to follow complex instructions without demonstration, suggesting an urgent need to improve open-source LLMs for practical, real-world tasks under the zero-shot setting.

2.4 CODEACT BENEFITS FROM MULTI-TURN INTERACTIONS AND EXISTING SOFTWARE PACKAGES

In Fig. 3, we show how an LLM agent can integrate with Python (i.e., CodeActAgent we trained in §3.2) and use existing software to perform complex tasks in multi-turn interactions. Thanks to its extensive knowledge of Python learned during pre-training, the LLM agent can automatically import the correct *Python libraries* to solve tasks without requiring user-provided tools or demonstrations. As illustrated in Fig. 3, CodeActAgent can use Pandas to download and process tabular data, use Scikit-Learn for machine learning train-test data split and regression model training, and use Matplotlib for data visualization. Furthermore, using the interactive Python interpreter for code execution allows automated error messages that help the LLM agent 'self-debug' their actions in a *multi-turn interaction* and eventually complete the human user's request correctly.

3 EMPOWERING OPEN-SOURCE LLM AGENT TO BE BETTER AT CODEACT

The promising results achieved by `CodeAct` motivate us to build an open-source LLM agent that can both interact with environments through `CodeAct` and communicate with humans using language. To improve open-source LLMs' `CodeAct` capability, in §3.1, we introduce CodeActInstruct, an instruction finetuning dataset that contains agent-environment interaction trajectories. We discuss data selection procedures in §3.1 to promote improvement from interaction behavior. Additionally, we show that `CodeAct` can be used together with existing agent-user conversation data (§3.1) to balance the dialog capability of the resulting LLM. Our model CodeActAgent, finetuned from LLaMA-2 (Touvron et al., 2023) and Mistral-7B (Jiang et al., 2023) on a mixture of CodeActInstruct and general



User

Here is a dataset about auto MPG: "https://huggingface.co/datasets/scikit-learn/auto-mpg/raw/main/auto-mpg.csv". You should download and load the dataset from the URL, take a look at it, check for missing value, do a train-test split, and train a regression model predicting its MPG. You should use everything except "mpg" and "car name" as the input features.

CodeActAgent

To accomplish this task, I will use the pandas library to download and load the dataset, check for missing values, and perform a train-test split. Then, I will use the scikit-learn library to train a regression model.

```python
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression

# Download and load the dataset
df = pd.read_csv("https://huggingface.co/datasets/scikit-learn/auto-mpg/raw/main/auto-mpg.csv")

# Check for missing values
print(df.isnull().sum())

# Remove rows with missing values
df = df.dropna()

# Perform train-test split
X = df.drop(columns=["mpg", "car name"])
y = df["mpg"]
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Train a regression model
model = LinearRegression()
model.fit(X_train, y_train)

# Predict MPG for test data
y_pred = model.predict(X_test)

# Evaluate the model
print("Mean squared error: ", np.mean((y_test - y_pred) ** 2))
print("R^2 score: ", r2_score(y_test, y_pred))
```

Use Pandas Library to Download, Examine, and Process Data

Use Scikit-Learn Library to Split Data and Train Regression Model

Environment

```
mpg              0
[...omitted for space...]
car name         0
dtype: int64
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
/tmp/ipykernel_12/1372561217.py in ?()
     18 X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2 [...]
     19
     20 # Train a regression model
     21 model = LinearRegression()
---> 22 model.fit(X_train, y_train)
     23
     24 # Predict MPG for test data
     25 y_pred = model.predict(X_test)

... [omitted for space]...

ValueError: could not convert string to float: '?'
```

Self-Debug from Automated Feedback

CodeActAgent

It seems that there is a data type issue in the dataset. The '?' character is present in the data, which is causing the error when trying to convert the data to a float. To fix this, I will remove the rows containing the '?' character.

```python
# Remove rows containing '?'
df = df.replace('?', np.nan)
df = df.dropna()
X = df.drop(columns=["mpg", "car name"])
y = df["mpg"]
X_train, X_test, y_train, y_test = train_test_split(
  X, y, test_size=0.2, random_state=42
)
model = LinearRegression()
model.fit(X_train, y_train)
y_pred = model.predict(X_test)
print("Mean squared error: ", np.mean((y_test - y_pred) ** 2))
print("R^2 score: ", r2_score(y_test, y_pred))
```

Environment

```
Mean squared error:  10.710864418838367
R^2 score:  0.790150038676035
```

CodeActAgent

The mean squared error is 10.711 and the R^2 score is 0.790, which indicates that the model has a good fit with the data [...]

User

Can you also give me the MSE and R^2 for the training set?

CodeActAgent **Able to Answer follow-up Question Interactively**

Certainly! Here are the mean squared error (MSE) and R^2 scores [...]

```python
from sklearn.metrics import mean_squared_error
# Calculate MSE for training set
mse_train = mean_squared_error(y_train, model.predict(X_train))
# Calculate R^2 score for training set
r2_train = r2_score(y_train, model.predict(X_train))

print("Mean squared error (training set):", mse_train)
print("R^2 score (training set):", r2_train)
```

Environment

```
Mean squared error (training set): 10.979255842373478
R^2 score (training set): 0.826001578671067
```

CodeActAgent

The mean squared error for the training set is 10.979 and the R^2 score is 0.826, which indicates [...]

User

Can you visualize the regression coefficient? You may access them via model.coef_ and their names via model.feature_names_in_

CodeActAgent

Certainly! Here is a visualization of the regression coefficients [...]

```python
import matplotlib.pyplot as plt
# Get regression coefficients
coef = model.coef_
# Get names of input features
feature_names = model.feature_names_in_
# Plot regression coefficients
plt.bar(feature_names, coef)
plt.xlabel('Input Feature')
plt.ylabel('Regression Coefficient')
plt.title('Regression Coefficients')
# Display the plot
plt.show()
```

Use Matplotlib Library for Data Visualization

Environment

```
<Figure size 640x480 with 1 Axes>
```

CodeActAgent

The regression coefficients are plotted as a bar chart, with each input feature on the x-axis and the corresponding coefficient on the y-axis [...]

User

Can you rotate the x ticks by 45 degrees for better readability?

CodeActAgent

Certainly! Here is the updated bar chart with the x-ticks rotated by 45 degrees for better readability:

```python
[..same code omitted...]
# Plot regression coefficients
plt.bar(feature_names, coef, tick_labels=False)
plt.xticks(
  feature_names, feature_names, rotation=45, ha='right', fontsize=12)
[..same code omitted...]
```

Environment

```
<Figure size 640x480 with 1 Axes>
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Cell In[1], line 10
      7 feature_names = model.feature_names_in_
      9 # Plot regression coefficients
---> 10 plt.bar(feature_names, coef, tick_labels=False)
     11 plt.xticks(feature_names, feature_names,
                   rotation=45, ha='right', fontsize=12)
     12 plt.xlabel('Input Feature')
[...]
AttributeError: Rectangle.set() got an unexpected keyword argument 'tick_labels'
```

Self-Debug for Data Visualization

CodeActAgent

It seems that the tick_labels argument is not a valid argument for the bar() function in Matplotlib. Instead, we can use the tick_params() function to rotate the x-ticks [...]

```python
[...same code omitted...]
plt.bar(feature_names, coef)
plt.tick_params(axis='x', labelrotation=45, ha='right', fontsize=12)
[...same code omitted...]
plt.show()
```

[3 rounds of self-debugging omitted for space]

CodeActAgent

I apologize for the confusion. It seems that the set_xticks() function is not a valid function in Matplotlib. Instead, we can use the xticks() function to set the x-ticks and rotate them by 45 degrees for better readability. Here is the updated bar chart with the x-ticks rotated by 45 degrees:

```python
[...same code omitted...]
plt.xticks(feature_names, feature_names,
           rotation=45, ha='right', fontsize=12)
[...same code omitted...]
plt.show()
```

Environment

```
<Figure size 640x480 with 1 Axes>
```

User

[Stops Interaction]

Figure 3: Example multi-turn interaction with Python packages using CodeActAgent (Mistral-7b). No in-context demonstrations are provided to the model. Some messages are omitted for space. See `https://chat.xwang.dev/r/Vqn108G` for complete interaction.

conversations, improves `CodeAct` performances without hurting LLM's general performance on a diverse suite of tasks (§3.2).

3.1 CODEACTINSTRUCT: AGENT-ENVIRONMENT INTERACTIONS

We consider four main use cases in agent-environment interaction and repurpose five existing datasets across different domains to generate trajectories:

- **Information Seeking:** We use a training subset of HotpotQA (Yang et al., 2018) to generate information-seeking trajectories, where LLMs use the `wikipedia_search` API (provided as a Python function) to search for information to answer questions.
- **Software Package (Tool) Usage:** We use the training set of code generation problems in APPS (Hendrycks et al., 2021a) and math problems in MATH (Hendrycks et al., 2021b). The code generation tasks already involve importing packages and/or creating new tools by defining a new Python function. For MATH, we provide an in-context demonstration of importing Python packages (e.g., `sympy` for symbolic math) for problem-solving.
- **External Memory:** We repurpose the training subset of WikiTableQuestion (Pasupat & Liang, 2015) and tweak it into two variants of tabular reasoning tasks that require accessing external memory: (1) SQL-based, requiring the LLM to interact with an SQL database through sqlite3 package to answer the question via SQL execution; (2) Pandas-based, requiring the model to interact with pandas tables to perform data operations (e.g., select, filter). Examples of instructions can be found in §G.3.1.
- **Robot Planning:** We use ALFWorld (Shridhar et al., 2020), a text-only embodied environment simulator, to generate trajectories that use robot-control APIs (repurposed as Python function) to complete household tasks. Following MINT (Wang et al., 2023e), we provide an in-context demonstration to encourage the use of for-loop and if-statement code blocks to automate repetitive operations (e.g., searching for items by visiting different locations).

Data Down-sampling. We down-sample each dataset by keeping only the most challenging instances, aiming to make trajectory generation more efficient and cost-effective. Furthermore, it also helps remove simple instances that existing LLMs can already solve. The statistics of the filtered dataset can be found in Tab. A.9. Please refer to §G.1 for details about the down-sample process.

Repurpose Data for Multi-turn Interaction. Some datasets (APPS, MATH, WikiTableQuestions) are initially single-turn problems that expect *one* solution per instruction, whereas, in a realistic agent use case, we often require multi-turn interaction to complete each task (Fig. 1 top). Following MINT (Wang et al., 2023e), we repurpose single-turn problems into multi-turn ones by allowing LLM to interact with the environment for multiple turns before it decides to submit one solution for evaluation. Specifically for code generation problems, we provide an in-context example to guide LLMs to test their solution on provided test cases before they submit the solution. Metrics from the original data will evaluate the submitted solution to determine its correctness. We include examples in §G.3.

Trajectory Generation. We use MINT's evaluation framework (Wang et al., 2023e) to generate interaction trajectories for the aforementioned datasets and determine the correctness of each trajectory. We run gpt-3.5-turbo-0613 from OpenAI, claude-1-instant and claude-2 from Anthropic on down-sampled data, except code generation, which we use a longer-context version of GPT-3.5 (gpt-3.5-turbo-0613-16k) due to the long-context requirement of the self-debugging process. On a subset of problems that none of these models can solve, we use gpt-4-0613 to generate trajectories.

Enhancing Agent's Capabilities of Improving from Interaction. We select a high-quality subset of all the generated trajectories from CodeActInstruct to promote the agent's ability to improve the next action based on prior observations (e.g., self-debugging from code execution error message, a planning capability in Fig. 2). To achieve this, we selectively preserve those trajectories wherein the model initially encounters errors but rectifies these inaccuracies in later interactions. For these instances, the LLM typically engages in self-reflection following the initial error, thereby proactively enhancing its future actions. Other filtering details are discussed in §G.2. On all trajectories generated, we keep 411 trajectories from gpt-4-0613 and 6728 trajectories from gpt-3.5 and claude. The statistics of the resulting dataset CodeActInstruct are shown in Tab. 4.

Comparing CodeActInstruct with Prior Work. Compared with prior work AgentInstruct (Zeng et al., 2023) and FireAct (Chen et al., 2023a) that mainly focus using text as action, CodeActInstruct results in models that are more practical in real-world implementation, as such models using `CodeAct` can directly interact with Python interpreters and open-source toolkits (Fig. 3), reducing the development effort for action parsing and tool creations. CodeActInstruct is systematically constructed following the general agent framework (Fig. 2). It covers diverse domains (e.g., compared

Table 4: Statistics of our training mixture and comparison with prior work. Please refer to §3.1 for details about CodeActInstruct and general conversation data. Token statistics are computed using Llama-2 tokenizer.

Data Mixture	Data Type	Data Name	# of Data Instances	# of Total Tokens	Avg. Tokens Per Instance
Prior Work	-	FireAct (Chen et al., 2023a)	2,063	542,176	262.81
	-	AgentInstruct (Zeng et al., 2023)	1,866	2,517,785	1349.30
CodeActInstruct (Ours)	Information Seeking	HotpotQA (Yang et al., 2018)	1,664	2,472,227	1485.71
	Software Packages (Tool)	MATH (Math, (Hendrycks et al., 2021b))	1,732	1,719,467	992.76
	Software Packages (Tool)	APPS (Code, (Hendrycks et al., 2021a))	647	1,235,472	1909.54
	External Memory	WikiTableQuestion (Pasupat & Liang, 2015)	1,065	1,316,246	1235.91
	Robot Planning	ALFWorld (Shridhar et al., 2020)	2,031	3,838,269	1889.84
		Total	**7,139**	**10,581,681**	**1482.24**
General Conversation	Single-Turn Reasoning	OpenOrca (Sub-sampled, (Lian et al., 2023))	50,000	14,034,152	280.68
	Multi-Turn Conversations	ShareGPT (Sub-sampled, (Anonymous, 2023))	10,000	17,933,861	1793.39
	Multi-Turn Conversations	ShareGPT (GPT-4, (OpenChat, 2023))	4,583	18,195,878	3970.30
	Multi-turn Reasoning	CapyBara (LDJnr, 2023)	4,647	4,982,435	1072.18
		Total	**69,230**	**55,146,326**	**796.57**

Table 5: Evaluation results for CodeActAgent. The best results among all open-source LLMs are **bolded**, and the second-best results are underlined. ID and OD stand for in-domain and out-of-domain evaluation correspondingly. Overall averaged performance normalizes the MT-Bench score to be consistent with other tasks and excludes in-domain tasks for fair comparison.

		Agent Tasks					Generic Tasks (OD)				Overall Average
		Code as Action			Text as Action (OD)						
Model	Size	MINT (ID)	MINT (OD)	M^{3}ToolEval (OD)	Miniwob++	SciWorld	MMLU	HumanEval	GSM8K	MTBench	
Open-source LLMs (LLaMA-2-based)											
Llama2 Base	7B	-*	-*	-*	-*	-*	45.3	12.8	14.6	-*	-*
Llama2 Chat	7B	3.2	11.0	0.0	0.0	5.9	48.0	13.9	27.7	6.3	21.1
FireAct (Chen et al., 2023a)	7B	0.0	0.3	0.0	0.0	6.8	44.1	3.5	12.4	4.5	14.0
AgentLM (Zeng et al., 2023)	7B	8.7	6.1	0.0	28.9	13.7	48.7	15.4	24.6	6.1	24.8
CodeActAgent (LLaMA-2)	7B	51.3	20.4	0.0	25.5	**17.6**	50.6	18.1	38.3	7.5	30.7
Open-source LLMs (Mistral-based)											
Mistral Base	7B	-*	-*	-*	-*	-*	**60.1**	30.5	52.1	-*	-*
Mistral Instruct	7B	18.8	9.7	0.0	0.5	4.0	53.8	29.3	43.3	6.4	25.6
CodeActAgent (Mistral)	7B	**57.4**	**32.4**	**12.2**	**46.2**	15.9	59.1	**34.7**	**58.0**	**8.2**	**42.5**
Closed-source LLMs											
gpt-3.5-turbo-0613	-	33.9	38.2	51.2	66.7	21.2	70.0	48.1	57.1	7.9	54.0
gpt-4-0613	-	68.6	70.2	67.1	69.4	36.4	86.4	67.0	87.1	9.0	71.7

[*] Some results are only available with instruction-tuned models.

to FireAct that only considers QA-task and search API), contains quality data (e.g., promotes agent's capability of self-debug) and of larger size (3.8x / 3.5x more data trajectories and 5x / 19x more tokens compared to AgentInstruct / FireAct respectively in Tab. 4). As we empirically show in Tab. 5, the resulting model (same backbone) of CodeActInstruct achieves 24% and 119% relative improvement compared to AgentInstruct and FireAct.

CodeActInstruct Can Be Used With Existing Agent-User Conversation Data. We use a sub-sampled set of OpenOrca (Lian et al., 2023) that focuses on single-turn chain-of-thought (CoT) reasoning, ShareGPT (Anonymous, 2023; OpenChat, 2023) from two sources that contain multi-turn conversations between human and LLM, and CapyBara (LDJnr, 2023) that focuses on reasoning in multi-turn conversations. Statistics and down-sampling details can be found in Tab. 4 and §C.

3.2 CODEACTAGENT

We fine-tune Llama-2 7B (Touvron et al., 2023) and Mistral 7B (Jiang et al., 2023) on a mixture of CodeActInstruct and general conversations (Tab. 4) to obtain CodeActAgent.

Training Setup. We perform full-parameter supervised fine-tuning with a sequence length of 4,096 tokens for Llama-2 and 16,384 for Mistral. Please refer to §D for more details.

Evaluation Setup. We use MINT (Wang et al., 2023e) to evaluate LLMs with `CodeAct` on a diverse range of agent tasks. CodeActAgent has some training domains overlapping with MINT's evaluation (i.e., MINT includes ALFWorld and MATH), hence we report separate numbers for MINT's in- and out-of-domain performance. Unless otherwise specified, we measure MINT tasks' success rates with interaction turn $k = 5$. We also evaluate out-of-domain agent tasks using text actions from MiniWob++ (computer tasks, Kim et al. (2023)) and ScienceWorld (text-based simulator for elementary science curriculum, Wang et al. (2022a)) to test whether CodeActAgent can generalize to different action formats. Finally, we include a suite of general LLM evaluation tasks to assess general capability: MMLU (Hendrycks et al., 2020) for knowledge-based QA, HumanEval (Chen et al., 2021) for single-turn code-generation, GSM8K (Cobbe et al., 2021) for single-turn tool-free math reasoning, and MTBench (Zheng et al., 2023) for instruction-following.

CodeActAgent Excels in `CodeAct` Task. As shown in Tab. 5, CodeActAgent (both variants) perform better than all evaluated open-source LLMs on both the in- and out-of-domain subsets of MINT. On M^3ToolEval, we find CodeActAgent (Mistral) outperforms open-source LLMs of similar size (7B and 13B) and even reaches similar performance to those 70B models (Tab. 3). Surprisingly, no improvement is observed for the Llama-2 variant. We discuss potential reasons in §H.

CodeActAgent Generalizes to Text Action. When evaluated on out-of-domain text actions, CodeActAgent (LLaMA2, 7B), which has never been optimized for text action, achieves comparable performance to AgentLM-7B (Zeng et al., 2023) which has explicit tuning for text actions.

CodeActAgent Maintains or Improves the Performance on General LLM Tasks. In Tab. 5, we find that CodeActAgent (both variants) performs better on generic LLM tasks we tested, except for a slight degradation on MMLU for CodeActAgent (Mistral, 7B).

Ablation Study. Tab. A.8 presents ablation experiments to determine the importance of CodeActInstruct and general conversations. Both CodeActInstruct and general conversations contribute to agent tasks, while general conversations are essential to maintain performance on general tasks.

4 RELATED WORK

4.1 ACTION MODULE IN LLM AGENTS

As detailed in Wang et al. (2023b), LLM-based autonomous agents are typically structured around four components: customized profiles (Park et al., 2023; Qian et al., 2023), long-term memory capabilities (Zhu et al., 2023; Fischer, 2023), reasoning and planning algorithms (Wei et al., 2022; Chen et al., 2023d), and, most crucially, action modules. The action modules are key to facilitating LLM agents to effectively interact with external entities, including humans (Lee et al., 2022) and tools (Qin et al., 2023a) in the environment (Wang et al., 2023e; Yang et al., 2024a). In this study, we address the critical problem of standardizing the action space for LLM agents. We further discuss the difference between `CodeAct` and the line of work that uses code generation for problem-solving in §A. We notice a concurrent study TaskWeaver (Qiao et al., 2023) similarly endorses the use of code. We discuss the principal distinctions in §B.

4.2 IMPROVING LLM AGENTS

Two primary methods for enhancing LLM agents are prompt engineering and instruction tuning, as surveyed by Wang et al. (2023b). For *prompt engineering* (Liu et al., 2023a), numerous strategies have been introduced to improve the chain-of-thought reasoning (Wei et al., 2022), including self-consistency-based reasoning (Wang et al., 2022b; Chen et al., 2023d) and tree-based approaches (Yao et al., 2023a). Moreover, LLMs can be strategically prompted to reflect on previous plans (Yao et al., 2023b; Wang et al., 2023f; Zhang et al., 2023), enabling them to refine initial actions through trial and error. Contrast to prompt engineering, *instruction tuning* intrinsically enhances LLMs (Chung et al., 2022), particularly in their agent capabilities (Zeng et al., 2023; Chen et al., 2023a). For effective training, human annotators can curate expert demonstrations for specific agent tasks, such as web browsing (Yao et al., 2022a; Nakano et al., 2021). To minimize human annotation efforts, prior work creates synthetic datasets using stronger LLMs to distill agent capabilities into local models, focusing on tool usage (Qin et al., 2023b), interaction (Chen et al., 2023c), and social skills (Liu et al., 2023b). CodeActInstruct aligns with the latter approach and creates datasets using stronger LLMs.

5 CONCLUSIONS

This work introduces `CodeAct` that employs executable Python code for the LLM agent's action, which is advantageous over using text or JSON action, especially in complex scenarios. We collect `CodeAct`-focused multi-turn interaction trajectories CodeActInstruct for instruction tuning, and train CodeActAgent that is specially designed for seamless integration with Python and can execute sophisticated tasks (e.g., model training) leveraging existing Python packages and autonomously rectifying errors through self-debugging.

BROADER IMPACTS, LIMITATIONS, AND FUTURE WORK

This paper presents work whose goal is to advance LLM-based autonomous agents that can communicate with humans through natural language and assist human users by performing tasks in environments on behalf of humans. In this section, we discuss potential societal consequences, limitations, and future work related to our work and its goal.

CodeActAgent is an initial prototype of an autonomous agent and still has several practical limitations. For example, it may suffer from hallucination commonly seen in LLMs (e.g., imagine the content of a variable without actually printing it out), suggesting the need for subsequent alignment (Ouyang et al., 2022) for further improvements.

Despite being a prototype, CodeActAgent has already demonstrated limited self-improving capability (e.g., self-debug error messages to improve its action) and the ability to interact with environments. Future work may build upon CodeActAgent to develop better agents by having them perform extensive interactions within a given environment and iteratively bootstrap their self-improving capability to learn to improve from past mistakes. More powerful agents, as results of such algorithms, are potentially beneficial for solving a wide range of real-world problems (e.g., theorem proving, drug discovery). As extensively discussed in Eloundou et al. (2023), a fully autonomous agent may transform the current landscape of the labor market and impact the jobs of existing workers.

Furthermore, since `CodeAct` directly grants access for the agent to freely execute code in a sandbox environment, in the worst scenario (e.g., in Sci-Fi movies), such an agent may potentially break free of the sandbox restriction and cause harm to the world through cyber-attack, highlighting the need for future work to design better safety mechanism to safeguard autonomous agents (Tang et al., 2024).

ACKNOWLEDGEMENT

We thank the anonymous reviewers for their suggestions and comments. This research is based upon work supported by U.S. DARPA ECOLE Program No. HR00112390060 and U.S. DARPA ITM Program No. FA8650-23-C-7316 and KAIROS Program No. FA8750-19-2-1004. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of DARPA, or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright annotation therein. This work used the Delta system at the National Center for Supercomputing Applications through allocation CIS230256 from the Advanced Cyberinfrastructure Coordination Ecosystem: Services & Support (ACCESS, Boerner et al. 2023) program, which is supported by National Science Foundation grants #2138259, #2138286, #2138307, #2137603, and #2138296.

REFERENCES

Michael Ahn, Anthony Brohan, Noah Brown, Yevgen Chebotar, Omar Cortes, Byron David, Chelsea Finn, Chuyuan Fu, Keerthana Gopalakrishnan, Karol Hausman, Alex Herzog, Daniel Ho, Jasmine Hsu, Julian Ibarz, Brian Ichter, Alex Irpan, Eric Jang, Rosario Jauregui Ruano, Kyle Jeffrey, Sally Jesmonth, Nikhil Joshi, Ryan Julian, Dmitry Kalashnikov, Yuheng Kuang, Kuang-Huei Lee, Sergey Levine, Yao Lu, Linda Luu, Carolina Parada, Peter Pastor, Jornell Quiambao, Kanishka Rao, Jarek Rettinghouse, Diego Reyes, Pierre Sermanet, Nicolas Sievers, Clayton Tan, Alexander Toshev, Vincent Vanhoucke, Fei Xia, Ted Xiao, Peng Xu, Sichun Xu, Mengyuan Yan, and Andy Zeng. Do as i can and not as i say: Grounding language in robotic affordances. In arXiv preprint arXiv:2204.01691, 2022.

Anonymous. Sharegpt dataset. `https://hf.co/datasets/anon8231489123/ShareGPT_Vicuna_unfiltered/blob/main/ShareGPT_V3_unfiltered_cleaned_split_no_imsorry.json`, 2023. A dataset containing multi-turn conversations between human and LLM assistant.

Timothy J Boerner, Stephen Deems, Thomas R Furlani, Shelley L Knuth, and John Towns. Access: Advancing innovation: Nsf's advanced cyberinfrastructure coordination ecosystem: Services & support. In Practice and Experience in Advanced Research Computing, pp. 173–176. 2023.

Andres M Bran, Sam Cox, Andrew D White, and Philippe Schwaller. Chemcrow: Augmenting large-language models with chemistry tools. arXiv preprint arXiv:2304.05376, 2023.

Alejandro Hernández Cano, Matteo Pagliardini, Andreas Köpf, Kyle Matoba, Amirkeivan Mohtashami, Xingyao Wang, Olivia Simin Fan, Axel Marmet, Deniz Bayazit, Igor Krawczuk, Zeming Chen, Francesco Salvi, Antoine Bosselut, and Martin Jaggi. epfllm megatron-llm, 2023. URL https://github.com/epfLLM/Megatron-LLM.

Harrison Chase. LangChain, October 2022. URL https://github.com/langchain-ai/langchain.

Baian Chen, Chang Shu, Ehsan Shareghi, Nigel Collier, Karthik Narasimhan, and Shunyu Yao. Fireact: Toward language agent fine-tuning. arXiv preprint arXiv:2310.05915, 2023a.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. arXiv preprint arXiv:2107.03374, 2021.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. Teaching large language models to self-debug. arXiv preprint arXiv:2304.05128, 2023b.

Yangyi Chen, Karan Sikka, Michael Cogswell, Heng Ji, and Ajay Divakaran. Dress: Instructing large vision-language models to align and interact with humans via natural language feedback. arXiv preprint arXiv:2311.10081, 2023c.

Yangyi Chen, Karan Sikka, Michael Cogswell, Heng Ji, and Ajay Divakaran. Measuring and improving chain-of-thought reasoning in vision-language models. arXiv preprint arXiv:2309.04461, 2023d.

Hyung Won Chung, Le Hou, Shayne Longpre, Barret Zoph, Yi Tay, William Fedus, Yunxuan Li, Xuezhi Wang, Mostafa Dehghani, Siddhartha Brahma, et al. Scaling instruction-finetuned language models. arXiv preprint arXiv:2210.11416, 2022.

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

Tyna Eloundou, Sam Manning, Pamela Mishkin, and Daniel Rock. Gpts are gpts: An early look at the labor market impact potential of large language models. arXiv preprint arXiv:2303.10130, 2023.

Kevin A Fischer. Reflective linguistic programming (rlp): A stepping stone in socially-aware agi (socialagi). arXiv preprint arXiv:2305.12647, 2023.

Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. Pal: Program-aided language models. In International Conference on Machine Learning, pp. 10764–10799. PMLR, 2023.

Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. Measuring massive multitask language understanding. In International Conference on Learning Representations, 2020.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, et al. Measuring coding challenge competence with apps. In Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 2), 2021a.

Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. Measuring mathematical problem solving with the math dataset. In Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 2), 2021b.

Sirui Hong, Xiawu Zheng, Jonathan Chen, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, et al. Metagpt: Meta programming for multi-agent collaborative framework. arXiv preprint arXiv:2308.00352, 2023.

Sirui Hong, Yizhang Lin, Bang Liu, Bangbang Liu, Binhao Wu, Danyang Li, Jiaqi Chen, Jiayi Zhang, Jinlin Wang, Li Zhang, Lingyao Zhang, Min Yang, Mingchen Zhuge, Taicheng Guo, Tuo Zhou, Wei Tao, Wenyi Wang, Xiangru Tang, Xiangtao Lu, Xiawu Zheng, Xinbing Liang, Yaying Fei, Yuheng Cheng, Zongze Xu, and Chenglin Wu. Data interpreter: An llm agent for data science, 2024.

Wenlong Huang, Chen Wang, Ruohan Zhang, Yunzhu Li, Jiajun Wu, and Li Fei-Fei. Voxposer: Composable 3d value maps for robotic manipulation with language models. arXiv preprint arXiv:2307.05973, 2023.

Albert Q Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, et al. Mistral 7b. arXiv preprint arXiv:2310.06825, 2023.

Geunwoo Kim, Pierre Baldi, and Stephen McAleer. Language models can solve computer tasks. arXiv preprint arXiv:2303.17491, 2023.

LDJnr. Capybara dataset. `https://hf.co/datasets/LDJnr/Verified-Camel`, `https://hf.co/datasets/LDJnr/Pure-Dove`, `https://hf.co/datasets/LDJnr/LessWrong-Amplify-Instruct`, 2023. A dataset focusing on reasoning in multi-turn conversations.

Mina Lee, Percy Liang, and Qian Yang. Coauthor: Designing a human-ai collaborative writing dataset for exploring language model capabilities. In Proceedings of the 2022 CHI conference on human factors in computing systems, pp. 1–19, 2022.

Minghao Li, Feifan Song, Bowen Yu, Haiyang Yu, Zhoujun Li, Fei Huang, and Yongbin Li. Api-bank: A benchmark for tool-augmented llms, 2023.

Wing Lian, Bleys Goodson, Eugene Pentland, Austin Cook, Chanvichet Vong, and ”Teknium”. Openorca: An open dataset of gpt augmented flan reasoning traces. `https://https://huggingface.co/Open-Orca/OpenOrca`, 2023.

Jacky Liang, Wenlong Huang, Fei Xia, Peng Xu, Karol Hausman, Brian Ichter, Pete Florence, and Andy Zeng. Code as policies: Language model programs for embodied control. In arXiv preprint arXiv:2209.07753, 2022.

Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig. Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. ACM Computing Surveys, 55(9):1–35, 2023a.

Ruibo Liu, Ruixin Yang, Chenyan Jia, Ge Zhang, Denny Zhou, Andrew M Dai, Diyi Yang, and Soroush Vosoughi. Training socially aligned language models in simulated human society. arXiv preprint arXiv:2305.16960, 2023b.

Yecheng Jason Ma, William Liang, Guanzhi Wang, De-An Huang, Osbert Bastani, Dinesh Jayaraman, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Eureka: Human-level reward design via coding large language models. arXiv preprint arXiv:2310.12931, 2023.

Grégoire Mialon, Roberto Dessì, Maria Lomeli, Christoforos Nalmpantis, Ram Pasunuru, Roberta Raileanu, Baptiste Rozière, Timo Schick, Jane Dwivedi-Yu, Asli Celikyilmaz, et al. Augmented language models: a survey. arXiv preprint arXiv:2302.07842, 2023.

Reiichiro Nakano, Jacob Hilton, Suchir Balaji, Jeff Wu, Long Ouyang, Christina Kim, Christopher Hesse, Shantanu Jain, Vineet Kosaraju, William Saunders, et al. Webgpt: Browser-assisted question-answering with human feedback. arXiv preprint arXiv:2112.09332, 2021.

OpenChat. Sharegpt dataset. https://hf.co/datasets/openchat/openchat_sharegpt_v3/blob/main/sharegpt_gpt4.json, 2023. A dataset containing multi-turn conversations between human and LLM assistants. It is filtered to contain data only from GPT-4.

Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. Training language models to follow instructions with human feedback. Advances in Neural Information Processing Systems, 35: 27730–27744, 2022.

Joon Sung Park, Joseph O'Brien, Carrie Jun Cai, Meredith Ringel Morris, Percy Liang, and Michael S Bernstein. Generative agents: Interactive simulacra of human behavior. In Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology, pp. 1–22, 2023.

Panupong Pasupat and Percy Liang. 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), pp. 1470–1480, 2015.

Shishir G. Patil, Tianjun Zhang, Xin Wang, and Joseph E. Gonzalez. Gorilla: Large language model connected with massive apis. ArXiv, abs/2305.15334, 2023. URL https://api.semanticscholar.org/CorpusID:258865184.

Chen Qian, Xin Cong, Cheng Yang, Weize Chen, Yusheng Su, Juyuan Xu, Zhiyuan Liu, and Maosong Sun. Communicative agents for software development. arXiv preprint arXiv:2307.07924, 2023.

Bo Qiao, Liqun Li, Xu Zhang, Shilin He, Yu Kang, Chaoyun Zhang, Fangkai Yang, Hang Dong, Jue Zhang, Lu Wang, et al. Taskweaver: A code-first agent framework. arXiv preprint arXiv:2311.17541, 2023.

Yujia Qin, Shengding Hu, Yankai Lin, Weize Chen, Ning Ding, Ganqu Cui, Zheni Zeng, Yufei Huang, Chaojun Xiao, Chi Han, et al. Tool learning with foundation models. arXiv preprint arXiv:2304.08354, 2023a.

Yujia Qin, Shi Liang, Yining Ye, Kunlun Zhu, Lan Yan, Ya-Ting Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, Sihan Zhao, Runchu Tian, Ruobing Xie, Jie Zhou, Marc H. Gerstein, Dahai Li, Zhiyuan Liu, and Maosong Sun. Toolllm: Facilitating large language models to master 16000+ real-world apis. ArXiv, abs/2307.16789, 2023b. URL https://api.semanticscholar.org/CorpusID:260334759.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. arXiv preprint arXiv:2302.04761, 2023.

Yongliang Shen, Kaitao Song, Xu Tan, Dongsheng Li, Weiming Lu, and Yueting Zhuang. Hugginggpt: Solving ai tasks with chatgpt and its friends in huggingface. arXiv preprint arXiv:2303.17580, 2023.

Mohit Shridhar, Xingdi Yuan, Marc-Alexandre Cote, Yonatan Bisk, Adam Trischler, and Matthew Hausknecht. Alfworld: Aligning text and embodied environments for interactive learning. In International Conference on Learning Representations, 2020.

Ishika Singh, Valts Blukis, Arsalan Mousavian, Ankit Goyal, Danfei Xu, Jonathan Tremblay, Dieter Fox, Jesse Thomason, and Animesh Garg. Progprompt: Generating situated robot task plans using large language models. In 2023 IEEE International Conference on Robotics and Automation (ICRA), pp. 11523–11530, 2023. doi: 10.1109/ICRA48891.2023.10161317.

Dídac Surís, Sachit Menon, and Carl Vondrick. Vipergpt: Visual inference via python execution for reasoning. Proceedings of IEEE International Conference on Computer Vision (ICCV), 2023.

Xiangru Tang, Qiao Jin, Kunlun Zhu, Tongxin Yuan, Yichi Zhang, Wangchunshu Zhou, Meng Qu, Yilun Zhao, Jian Tang, Zhuosheng Zhang, et al. Prioritizing safeguarding over autonomy: Risks of llm agents for science. arXiv preprint arXiv:2402.04247, 2024.

TIOBE Index. Tiobe index. https://www.tiobe.com/tiobe-index/, Accessed at Jan 23rd, 2024, 2024. The TIOBE Programming Community index is an indicator of the popularity of programming languages. The index is updated once a month. The ratings are based on the number of skilled engineers world-wide, courses and third party vendors.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models. arXiv preprint arXiv:2307.09288, 2023.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models. arXiv preprint arXiv:2305.16291, 2023a.

Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhiyuan Chen, Jiakai Tang, Xu Chen, Yankai Lin, et al. A survey on large language model based autonomous agents. arXiv preprint arXiv:2308.11432, 2023b.

Ruoyao Wang, Peter Alexander Jansen, Marc-Alexandre Côté, and Prithviraj Ammanabrolu. Scienceworld: Is your agent smarter than a 5th grader? In Conference on Empirical Methods in Natural Language Processing, 2022a. URL https://api.semanticscholar.org/CorpusID:247451124.

Xingyao Wang, Sha Li, and Heng Ji. Code4Struct: Code generation for few-shot event structure prediction. In Anna Rogers, Jordan Boyd-Graber, and Naoaki Okazaki (eds.), Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pp. 3640–3663, Toronto, Canada, July 2023c. Association for Computational Linguistics. doi: 10.18653/v1/2023.acl-long.202. URL https://aclanthology.org/2023.acl-long.202.

Xingyao Wang, Hao Peng, Reyhaneh Jabbarvand, and Heng Ji. Leti: Learning to generate from textual interactions. ArXiv, abs/2305.10314, 2023d.

Xingyao Wang, Zihan Wang, Jiateng Liu, Yangyi Chen, Lifan Yuan, Hao Peng, and Heng Ji. Mint: Evaluating llms in multi-turn interaction with tools and language feedback. arXiv preprint arXiv:2309.10691, 2023e.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. arXiv preprint arXiv:2203.11171, 2022b.

Zihao Wang, Shaofei Cai, Anji Liu, Xiaojian Ma, and Yitao Liang. Describe, explain, plan and select: Interactive planning with large language models enables open-world multi-task agents. arXiv preprint arXiv:2302.01560, 2023f.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. Advances in Neural Information Processing Systems, 35:24824–24837, 2022.

Qiantong Xu, Fenglu Hong, Bo Li, Changran Hu, Zhengyu Chen, and Jian Zhang. On the tool manipulation capability of open-source large language models, 2023.

John Yang, Akshara Prabhakar, Karthik Narasimhan, and Shunyu Yao. Intercode: Standardizing and benchmarking interactive coding with execution feedback. Advances in Neural Information Processing Systems, 36, 2024a.

Ke Yang, Jiateng Liu, John Wu, Chaoqi Yang, Yi R. Fung, Sha Li, Zixuan Huang, Xu Cao, Xingyao Wang, Yiquan Wang, Heng Ji, and Chengxiang Zhai. If llm is the wizard, then code is the wand: A survey on how code empowers large language models to serve as intelligent agents, 2024b.

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

Zonghan Yang, An Liu, Zijun Liu, Kaiming Liu, Fangzhou Xiong, Yile Wang, Zeyuan Yang, Qingyuan Hu, Xinrui Chen, Zhenhe Zhang, Fuwen Luo, Zhicheng Guo, Peng Li, and Yang Liu. Towards unified alignment between agents, humans, and environment, 2024c.

Shunyu Yao, Howard Chen, John Yang, and Karthik Narasimhan. Webshop: Towards scalable real-world web interaction with grounded language agents. Advances in Neural Information Processing Systems, 35:20744–20757, 2022a.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In The Eleventh International Conference on Learning Representations, 2022b.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models. arXiv preprint arXiv:2305.10601, 2023a.

Weiran Yao, Shelby Heinecke, Juan Carlos Niebles, Zhiwei Liu, Yihao Feng, Le Xue, Rithesh Murthy, Zeyuan Chen, Jianguo Zhang, Devansh Arpit, et al. Retroformer: Retrospective large language agents with policy gradient optimization. arXiv preprint arXiv:2308.02151, 2023b.

Lifan Yuan, Yangyi Chen, Xingyao Wang, Yi Ren Fung, Hao Peng, and Heng Ji. Craft: Customizing llms by creating and retrieving from specialized toolsets. ArXiv, abs/2309.17428, 2023. URL `https://api.semanticscholar.org/CorpusID:263310662`.

Aohan Zeng, Mingdao Liu, Rui Lu, Bowen Wang, Xiao Liu, Yuxiao Dong, and Jie Tang. Agenttuning: Enabling generalized agent abilities for llms, 2023.

Chenrui Zhang, Lin Liu, Jinpeng Wang, Chuyuan Wang, Xiao Sun, Hongyu Wang, and Mingchen Cai. Prefer: Prompt ensemble learning via feedback-reflect-refine. arXiv preprint arXiv:2308.12033, 2023.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, et al. Judging llm-as-a-judge with mt-bench and chatbot arena. arXiv preprint arXiv:2306.05685, 2023.

Tianyu Zheng, Ge Zhang, Tianhao Shen, Xueling Liu, Bill Yuchen Lin, Jie Fu, Wenhu Chen, and Xiang Yue. Opencodeinterpreter: Integrating code generation with execution and refinement. https://arxiv.org/abs/2402.14658, 2024.

Xizhou Zhu, Yuntao Chen, Hao Tian, Chenxin Tao, Weijie Su, Chenyu Yang, Gao Huang, Bin Li, Lewei Lu, Xiaogang Wang, et al. Ghost in the minecraft: Generally capable agents for open-world enviroments via large language models with text-based knowledge and memory. arXiv preprint arXiv:2305.17144, 2023.

Table A.6: Example of actions for re-purposed API-Bank (Li et al., 2023) and M^{3}ToolEval.

Format	**Action**
`CodeAct`	`AddAgenda(content="Meeting with John", time="2023-10-26 09:00:00")`
JSON	`{"action": "AddAgenda", "content": "Meeting with John", "time": "2023-10-26 09:00:00"}`
Text	`Action: AddAgenda, content: Meeting with John, time: 2023-10-26 09:00:00`

Table A.7: Comparison between M^{3}ToolEval and existing tool-use evaluation benchmark.

Benchmark	M^{3}ToolEval (This work)	ToolBench (Qin et al., 2023b)	APIBench (Patil et al., 2023)	API-Bank (Li et al., 2023)	ToolBench (Xu et al., 2023)
Requiring multi-turn interaction	✓	✓	✗	✗	✗
Multiple tools	✓	✓	✓	✓	✓
Evaluation	Answer Match	LLM Evaluator	AST Tree Match	API-Call Match	Test Case
No dependency on external API*	✓	✗	✗	✓	✗
Supported API Action Format	`CodeAct` & JSON & Text	JSON	`CodeAct`	JSON	`CodeAct`

* Whether to rely on external API (e.g., RapidAPI, Google Sheet) hosted by a third party. The availability of such third-party APIs can greatly impact evaluation results (e.g., low API-calling performance not because the model is bad but rather because the API required is not accessible).

Table A.8: Ablation study results. The best results are **bolded**, and the second-best results are underlined. ID and OD stand for in-domain and out-of-domain evaluation correspondingly. Overall averaged performance normalizes the MT-Bench score to be consistent with other tasks and excludes in-domain tasks for fair comparison.

		Agent Tasks				**Generic LLM Tasks**				**Overall**
		Code as Action		**Text as Action (OD)**		**(OD)**				**Average**
Model	Size	MINT (ID)	MINT (OD)	Miniwob++	SciWorld	MMLU	HumanEval	GSM8K	MTBench	
CodeActAgent (Llama2-based)	7B	**51.3**	**20.4**	25.5	**17.6**	**50.6**	18.1	**38.3**	**7.5**	**35.1**
w/o `CodeAct`	7B	17.0	15.5	**36.4**	16.9	49.5	14.7	36.0	7.2	34.5
w/o general conversations	7B	29.2	15.9	0.0	17.1	46.4	**19.7**	20.6	4.1	22.9
CodeActAgent (Mistral-based)	7B	**57.4**	**32.4**	46.2	15.9	59.1	**34.7**	58.0	8.2	**46.8**
w/o `CodeAct`	7B	32.9	23.0	**47.8**	**17.0**	**59.9**	33.2	**59.5**	**8.3**	46.2
w/o general conversations	7B	50.5	13.9	0.0	11.0	52.4	27.9	26.8	2.6	22.6

A COMPARISON WITH WORK THAT USES CODE GENERATION FOR PROBLEM-SOLVING

In this section, we discuss the fundamental differences between `CodeAct` and prior work that prompt LLM to generate code for problem-solving. Existing work have explored using code generation for task-solving in different domains, for example, Code4Struct (Wang et al., 2023c) for structured prediction, PaL (Gao et al., 2023) for math reasoning, Meta-GPT (Hong et al., 2023) for multi-agent collaboration, code-as-policy (Liang et al., 2022) for robot control, ViperGPT (Surís et al., 2023) for visual question answering, Voyager (Wang et al., 2023a) for playing games, Data Interpreter (Hong et al., 2024) for data science tasks, etc.

Most prior work generates code (i.e., a static sequence of actions) in a single-turn setting and cannot dynamically readjust action on new observation: It is considered a failure when the model-generated code fails to solve a task on the *first attempt*. This setting overlooks the potential of environmental observation (e.g., code execution results) that might benefit future action and overall decision (e.g., dynamically adjusting subsequent code after observing intermediate code execution results, fixing erroneous code after seeing an error message). That is, the generated code is a static sequence of actions that cannot be dynamically re-adjusted on the fly by incorporating new observations. Such a single-turn setting makes it challenging to scale to more challenging problems since even expert human programmers usually cannot write functionally correct code in the first pass. On the other hand, `CodeAct` is a multi-turn interaction agent framework that allows dynamic adjustment of prior actions or emitting new actions by design (§2.1, Fig. 2) and is compatible with any form of textual observation (e.g., tool execution output, automated feedback) from the environment. Beyond being compatible with environmental observation, our instruction tuning dataset CodeActInstruct specifically collects data for multi-turn self-improving, offering a practical solution to enhance LLM's multi-turn self-improving process.

In addition, previous approaches require heavy prompt engineering and crafting of few-shot demonstrations to tailor LLMs to a particular domain or task (e.g., robot control (Liang et al., 2022)) since the backbone LLMs are not specially optimized for dynamic planning and decision making. In contrast, in this work, we propose the `CodeAct` framework that uses executable Python code to consolidate LLM agents' actions into unified action space and collect CodeActInstruct on a diverse array of tasks (e.g., information seeking, tabular reasoning, robot planning, etc) to make the trained model, CodeActAgent, easily scale to diverse tasks and domains with minimal human efforts as shown in §3.2.

One notable exception among prior work is Voyager (Wang et al., 2023a), which performs iterative prompting in a constrained action space of *function definitions* to fix code errors. Different from `CodeAct`, such setting disallows dynamic re-adjustment of *atomic* actions on the fly: In `CodeAct`, for a particular task (e.g., craft stone sword in Minecraft), the agent can first execute one line of code (any atomic action or composed functions, e.g., move forward, locate stone), and dynamically produce different actions based on the observation of the first action. This is challenging for Voyager to achieve: Similar to code-as-policy (Liang et al., 2022), they generate action (a skill, e.g., craft stone sword) as a Python *function definition* that outlines the entire plan for a task (e.g., multi-step code outlining how you should craft a stone sword and handles for different potential cases, which requires strong domain knowledge). This imposes significant constraints on the agent's action space and disallows dynamic re-adjustment of *atomic* actions on the fly: That is, the agent can only generate one complete function first (e.g., by imaging all possible cases that might happen when you try to locate stones), execute the entire function, observe the feedback, and update the entire function as action in the subsequent move. Besides the constrained ability to re-adjust action from environmental observation, they also rely on heavy prompting engineering (a typical drawback discussed above) to provide relevant information (e.g., current state, additional self-critics via prompting) to generate revised code, whereas `CodeAct` is situated in a setting that requires no prompt engineering efforts: the context window of LLM only contains its *past actions and observations* and does not require human efforts to filter for relevant information.

Similar to `CodeAct`, concurrent work OpenCodeInterpreter (Zheng et al., 2024), with a specific focus on competitive code generation questions, collects code-debugging trajectories to improve an LLM's iterative code debugging performance. However, its applicability to general LLM agent tasks remains unknown.

B COMPARISON WITH TASKWEAVER

In the landscape of unifying the action space of LLM agents, our work represents a leap over the previous initiative, TaskWeaver (Qiao et al., 2023). While TaskWeaver deserves acknowledgment for initially integrating code into the action space of LLM agents, its exploration remains limited. This work, primarily characterized by its reliance on a limited set of qualitative examples with close-sourced models as the backbones, fails to harness the full potential of this integration, remaining merely conceptual demonstrations. Our work transcends mere conceptualization by conducting an extensive and rigorous analysis, clearly quantifying the benefits of code action within LLM agents. Beyond this, we introduce a unique instruction-tuning dataset CodeActInstruct specifically designed to amplify the agent's capabilities in executing code-based actions and an open-source LLM agent CodeActAgent. These contributions not only extend the work of TaskWeaver but also pave the way for future explorations, offering valuable resources to the open-source community and redefining the potential of LLM agents in practical applications.

C GENERAL DATA DOWN-SAMPLE

- **ShareGPT** (Anonymous, 2023): We remove all single-turn conversations, then perform random sub-sample to a desired final number.
- **ShareGPT (GPT-4)** (OpenChat, 2023): We do not perform sub-sampling on this dataset.

- **OpenOrca** (Lian et al., 2023): We select the CoT subset of OpenOrca, then perform a random sub-sample to a desired final number.
- **CapyBara** (LDJnr, 2023): We do not perform sub-sampling on this dataset.

D CODEACTAGENT TRAINING DETAILS

All SFT experiments are performed on one 4xA100 40GB SXM node using a fork of Megatron-LLM (Cano et al., 2023) with a training throughput of around 9k tokens per second. We use chatML format[2] for all multi-turn data, and we only calculate and optimize for loss on the assistant response. We pack short instances into longer ones and apply flash attention for training efficiency.

We train both LLaMA-2 and Mistral LLMs with Tensor Parallel of 4, the learning rate of `1e-5` with 50 warmup steps and cosine decay (end learning rate of `1e-6`). We train for five epochs with a batch size of 32. We use the 3rd epoch checkpoint for all our experiments.

E EXAMPLE PROMPT FOR CODEACT

This is an example (zero-shot) system prompt used in a deploy instance of `CodeAct` where we used chatML format.

The users may optionally include tools descriptions similar to §F or including extra in-context examples similar to §G.3.

```
<|im_start|>system
A chat between a curious user and an artificial intelligence assistant.
    The assistant gives helpful, detailed, and polite answers to the user
    's questions.
The assistant can interact with an interactive Python (Jupyter Notebook)
    environment and receive the corresponding output when needed. The
    code should be enclosed using "<execute>" tag, for example: <execute>
     print("Hello World!") </execute>.
The assistant should attempt fewer things at a time instead of putting
    too much code in one <execute> block. The assistant can install
    packages through PIP by <execute> !pip install [package needed] </
    execute> and should always import packages and define variables
    before starting to use them.
The assistant should stop <execute> and provide an answer when they have
    already obtained the answer from the execution result. Whenever
    possible, execute the code for the user using <execute> instead of
    providing it.
The assistant's response should be concise, but do express their thoughts
    .
<|im_end|>
```

F M^{3}TOOLEVAL PROMPT

```
You have access to the following tools:
{{Tool Definition}}

{{Formatting Instruction}}

Now, let's get started!

Instruction: {{Example: Find the current price of Legendary Wand.}}
Answer in the format of 'xx.xx' (e.g., 12.34).

You can optionally express your thoughts using natural language before
    your action. For example, 'Thought: I want to use tool_name to do
    something. Action: <your action to call tool_name> End Action'.
```

---

[2] https://github.com/openai/openai-python/blob/release-v0.28.0/chatml.md

```
Note that your output should always contain either 'Action:' or 'Answer
    :', but not both.
When you are done, output the result using 'Answer: your answer'
Please ONLY output the answer (e.g., single number), without any other
    text.
```

Each $\{\{\ldots\}\}$ component above will be substituted with corresponding information.

F.1 EXAMPLE OF {{TOOL DEFINITION}}

The following is an example tool definition for web-browsing.

```
[1] click_url: Clicks on a URL. A clickable URL looks like [Clickable '<
    url_argument>'] in the webpage.
Arguments: url (str).
Returns the rendered content of the webpage after clicking the URL
    showing on the current rendered page.
    Signature: click_url(url: str) -> str
[2] go_to_previous_page: Goes back to the previous page. It has no
    arguments.
After going back to the previous page, return the rendered content of the
     webpage.
    Signature: go_to_previous_page() -> str
[3] scroll_down: Scrolls down the view. It has no arguments.
Returns the rendered content of the webpage after scrolling down.
    Signature: scroll_down() -> str
[4] scroll_up: Scrolls up the view. It has no arguments.
Returns the rendered content of the webpage after scrolling up.
    Signature: scroll_up() -> str
[5] view: Return the current view in string format of the rendered
    webpage. It has no arguments.
Returns the rendered content of the webpage.
You should call this when you want to see the rendered content of the
    current webpage.
    Signature: view() -> str
[6] calculator: Evaluates the given expression and returns the result.
    Accepts a calculation expression as input. For example, "2 + (3 * 4)"
     will return 14.
    Signature: calculator(expression: str) -> float
```

F.2 EXAMPLE OF {{FORMATTING INSTRUCTION}}

Different action format has different formatting instructions.

F.3 FORMATTING INSTRUCTION FOR CODE AS ACTION

```
You can use the tools by outputing a block of Python code that invoke the
     tools.
You may use for-loops, if-statements, and other Python constructs when
    necessary.
Be sure to print the final answer at the end of your code.
You should begin your tool invocation with 'Action:' and end it with 'End
     Action'.
Example: 'Action:
tool_name(argument_1)
End Action'
```

F.4 FORMATTING INSTRUCTION FOR JSON AS ACTION

```
You can use the tools by outputing a JSON object with the following
    fields:
  - 'tool': the name of the tool
  - 'args': a list of arguments to the tool
You should begin your tool invocation with 'Action:' and end it with 'End
     Action'.
Example: 'Action: {"tool": "tool_name", "args": ["argument_1"]} End
    Action'
You can only invoke one tool at a time.
```

F.5 FORMATTING INSTRUCTION FOR TEXT AS ACTION

```
You can use the tools by outputing the tool name followed by its
    arguments, delimited by commas.
You should begin your tool invocation with 'Action:' and end it with 'End
     Action'.
Example: 'Action: tool_name, argument_1 End Action'
You can only invoke one tool at a time.
```

G CODEACT INTERACTION DATA

G.1 DATASET DOWNSAMPLE

Table A.9: CodeActInstruct components and the number of instances for training trajectory generation.

Domain	Capability	Dataset	# of Instances
Web Search	Information seeking through search API	HotpotQA (Yang et al., 2018)	3,000
Math Reasoning	Math problem-solving using math Libraries in Python (e.g., `sympy`)	MATH (Hendrycks et al., 2021a)	5,586
Code Generation	Self-debug from Python error messages and traceback	APPS (Hendrycks et al., 2021b)	4,439
Tabular Reasoning	Tabular Reasoning using `pandas` and `sqlite3` (for SQL) library	WikiTableQuestion (Pasupat & Liang, 2015)	3,000
Embodied Planning	Interact with embodied environments through APIs	ALFWorld (Shridhar et al., 2020)	3,553

- **Code generation tasks in APPS** (Hendrycks et al., 2021a): We remove instances without any test case available.
- **Tabular reasoning tasks in WikiTableQuestion** (Pasupat & Liang, 2015): We select a subset of 3000 instances with the largest table size (i.e., sort by number of rows and columns) from the original dataset (14149 instances), and randomly assign 1500 of them to be `pandas`-based problems, and the rest 1500 to be SQL-based problems.
- **Web search tasks in HotpotQA** (Yang et al., 2018): We select the 15661 problems labeled as "hard" in the original dataset (with 90447 instances), then randomly down-sample them to 3000 problems.
- **Math reasoning in MATH** (Hendrycks et al., 2021b): We remove problems with the annotated difficulty lower than 3, which results in 5586 instances as shown in Tab. A.9.
- **Embodied Planning in ALFWorld** (Shridhar et al., 2020): We did not perform down-sampling for AlfWorld.

G.2 DATA SELECTION HEURISTIC

Given successful task-solving trajectories that have more than 2 turns, we apply the following heuristic to select instances that can promote the code-as-actions, self-improvement, and instruction-following capabilities of LLM agents:

- **Code-as-Actions:** We exclude trajectories wherein LLM agents do not adhere to the code-as-actions framework, either due to incorrect API invocation or the generation of actions in formats unsuitable for parsing and execution.
- **Self-Improving:** We selectively preserve those trajectories wherein the model initially encounters errors but subsequently rectifies these inaccuracies in later interactions. In addition, we eliminate successful trajectories that exclusively yield errors in all code executions. These are deemed ineffective demonstrations, as our objective is to prevent the model from

learning to consistently execute erroneous code while still managing to provide correct answers.

- **Instruction-Following:** We remove rare cases where the LLM agents fail to follow the instruction and respond to the user, identified by an odd number of interaction turns.

After applying all these heuristics, we obtain 6728 trajectories (out of 6985) from `gpt-3.5` and `claude`, and 411 trajectories (out of 413) from `gpt-4-0613`.

G.3 EXAMPLE OF TRAJECTORY GENERATION PROMPT

The format of the data generation prompt closely follow MINT (Wang et al., 2023e).

G.3.1 TABULAR REASONING (WIKITABLEQUESTION)

We only provide one-shot example for SQL-based tabular reasoning. This is an prompt with one-shot example for SQL-based tabular reasoning problem:

```
You are a helpful assistant assigned with the task of problem-solving. To
     achieve this, you will be using an interactive coding environment
    equipped with a variety of tool functions to assist you throughout
    the process.

At each turn, you should first provide your step-by-step thinking for
    solving the task. Your thought process should be enclosed using "<
    thought>" tag, for example: <thought> I need to print "Hello World!"
    </thought>.

After that, you have two options:

1) Interact with a Python programming environment and receive the
    corresponding output. Your code should be enclosed using "<execute>"
    tag, for example: <execute> print("Hello World!") </execute>.
2) Directly provide a solution that adheres to the required format for
    the given task. Your solution should be enclosed using "<solution>"
    tag, for example: The answer is <solution> A </solution>.

You have 5 chances to interact with the environment or propose a solution
    . You can only propose a solution 2 times.



---

Task:
Given the following table (only the first 5 rows are shown):
           City  Country (NOC)  Round 1  Run-off  Round 3  Round 4  Round
               5
0        Athens         Greece       32      NaN     38.0     52.0
    66.0
1          Rome          Italy       23      NaN     28.0     35.0
    41.0
2     Cape Town   South Africa       16     62.0     22.0     20.0
    NaN
3     Stockholm         Sweden       20      NaN     19.0      NaN
    NaN
4  Buenos Aires      Argentina       16     44.0      NaN      NaN
    NaN

Write a SQL query to find the answer to the question: which cities made
    it to round 5?.
The SQLite3 database is preloaded for you and can be accessed within <
    execute> block via the variable `conn` (SQLite3 connection object).
The table name is `data_table`.
```

```
Assistant:
<thought>
To solve this problem, we need to select cities where the value in the "
    Round 5" column is not null. Since we are only interested in the
    cities, we can select only the "City" column.
</thought>
<execute>
query = """
SELECT
    City
FROM
    data_table
WHERE
    ‘Round 5‘ IS NOT NULL;
"""
# conn is preloaded
result = conn.execute(query).fetchall()
print(result)
</execute>

Observation:
[(’Athens’,), (’Rome’,)]

Assistant:
The answer is <solution> ["Athens", "Rome"] </solution>.


---

Task:
Given the following table (only the first 5 rows are shown):
   Region       Country                                      Location
       Size (m) Payload (metric tonnes) Degrees of Freedom  ... Z Vert
       vel (mm/s) X Horiz accel (m/s2) Y Horiz accel (m/s2) Z Vert accel
       (m/s2) Max Freq (Hz) Details checked
   +-
0  Africa       Algeria              CGS Laboratory (in construction)  6.1
     x 6.1                      60                   6  ...
    +-1000                  +-10                  +-10
    +-8           100       30/6/2010
1  Africa  South Africa                   University of Witwatersrand
         4 x 4                      10                   1  ...
                  NaN                  +-10                   NaN
                    NaN            40       17/7/2009
2    Asia         China  China Academy of Building Research, Beijing  6.1
     x 6.1                      60                   6  ...
    +-800                  +-15                  +-10
    +-8           50               ?
3    Asia         China                          Guangzhou University
         3 x 3                      20                   6  ...
                +-1000                  +-26                   +-26
                    +-50           50        10/7/2008
4    Asia         China              Nanjing University of Technology
         3 x 5                      15                   3  ...
                 +-500                  +-10                   +-10
                    +-10            50                ?

[5 rows x 17 columns]

Write a SQL query to find the answer to the question: which is the other
    besides asia the most region charted.
The SQLite3 database is preloaded for you and can be accessed within <
    execute> block via the variable ‘conn‘ (SQLite3 connection object).
```

This is an example instruction for Pandas-package-based[3] tabular reasoning problem:

```
Task:
Given the following table (only the first 5 rows are shown):
  Pos  No          Rider     Bike  Laps       Time  Grid  Points
0   1  93   Marc Marquez    Derbi  22.0  40:46.315     1    25.0
1   2  38  Bradley Smith  Aprilia  22.0     +4.638     3    20.0
2   3  44  Pol Espargaro    Derbi  22.0     +4.996     2    16.0
3   4  11 Sandro Cortese    Derbi  22.0    +45.366     5    13.0
4   5   7  Efren Vazquez    Derbi  22.0    +45.433     8    11.0

Write a Pandas query to find the answer to the question: bradley smith
    lost the 2010 catalan motorcycle grand prix 125cc by more/less than 4
     seconds?.
The dataframe is preloaded for you and can be accessed within <execute>
    block via the variable `df`.
```

G.3.2 CODE GENERATION (APPS)

Here is an example of the prompt with one in-context example for code generation on the APPS dataset (Hendrycks et al., 2021a) that encourages the LLM to self-debug its solution:

```
You are a helpful assistant assigned with the task of problem-solving. To
     achieve this, you will be using an interactive coding environment
    equipped with a variety of tool functions to assist you throughout
    the process.

At each turn, you should first provide your step-by-step thinking for
    solving the task. Your thought process should be enclosed using "<
    thought>" tag, for example: <thought> I need to print "Hello World!"
    </thought>.

After that, you have two options:

1) Interact with a Python programming environment and receive the
    corresponding output. Your code should be enclosed using "<execute>"
    tag, for example: <execute> print("Hello World!") </execute>.
2) Directly provide a solution that adheres to the required format for
    the given task. Your solution should be enclosed using "<solution>"
    tag, for example: The answer is <solution> A </solution>.

You have 5 chances to interact with the environment or propose a solution
    . You can only propose a solution 2 times.



---

Task:
Mikhail walks on a Cartesian plane. He starts at the point $(0, 0)$, and
    in one move he can go to any of eight adjacent points. For example,
    if Mikhail is currently at the point $(0, 0)$, he can go to any of
    the following points in one move:  $(1, 0)$;  $(1, 1)$;  $(0, 1)$;
    $(-1, 1)$;  $(-1, 0)$;  $(-1, -1)$;  $(0, -1)$;  $(1, -1)$.

If Mikhail goes from the point $(x1, y1)$ to the point $(x2, y2)$ in one
    move, and $x1 \ne x2$ and $y1 \ne y2$, then such a move is called a
    diagonal move.

Mikhail has $q$ queries. For the $i$-th query Mikhail's target is to go
    to the point $(n_i, m_i)$ from the point $(0, 0)$ in exactly $k_i$
    moves. Among all possible movements he want to choose one with the
    maximum number of diagonal moves. Your task is to find the maximum
```

---

[3] https://pandas.pydata.org/

number of diagonal moves or find that it is impossible to go from the point $(0, 0)$ to the point (n_i, m_i) in k_i moves.

Note that Mikhail can visit any point any number of times (even the destination point!).

-----Input-----

The first line of the input contains one integer q ($1 \le q \le 10^4$) - the number of queries.

Then q lines follow. The i-th of these q lines contains three integers n_i, m_i and k_i ($1 \le n_i, m_i, k_i \le 10^{18}$) - x-coordinate of the destination point of the query, y-coordinate of the destination point of the query and the number of moves in the query, correspondingly.

-----Output-----

Print q integers. The i-th integer should be equal to -1 if Mikhail cannot go from the point $(0, 0)$ to the point (n_i, m_i) in exactly k_i moves described above. Otherwise the i-th integer should be equal to the the maximum number of diagonal moves among all possible movements.

-----Example-----
Input
3
2 2 3
4 3 7
10 1 9

Output
1
6
-1

-----Note-----

One of the possible answers to the first test case: $(0, 0) \to (1, 0) \to (1, 1) \to (2, 2)$.

One of the possible answers to the second test case: $(0, 0) \to (0, 1) \to (1, 2) \to (0, 3) \to (1, 4) \to (2, 3) \to (3, 2) \to (4, 3)$.

In the third test case Mikhail cannot reach the point $(10, 1)$ in 9 moves.

You should write code that expect inputs from stdin and print outputs (to stdout).

Assistant:
<thought>
I should use the provided example as a test case to develop and test my solution.

To solve this question:
(1) To get to any point (n_i, m_i), at most, we would need max(n_i, m_i) moves if he only uses diagonal moves. If he's allowed more moves (k_i

```
    ), he should use them to move away from the target and then back
    towards it to maximize diagonal moves.

(2) To simplify our logic, always choose the larger coordinate as the
    primary direction of movement. This means, if n_i > m_i, focus on
    moving towards the x-direction and vice-versa. This avoids having to
    check both directions every time.

(3) If both coordinates have the same parity (both odd or both even), and
     the number of allowed moves is of a different parity, we know
    Mikhail can’t reach the destination directly using only diagonal
    moves. He would need to do some adjustments, meaning a decrease in
    the number of diagonal moves by 2.

Let me try to implement this solution.
</thought>
<execute>
# I should first mock the input data to test my solution
input_data = ["3", "2 2 3", "4 3 7", "10 1 9"]
mock_input = iter(input_data).__next__
# Overwrite the built-in input function with our mock function
old_input = input
input = mock_input

# Now I can implement my solution
q = int(input())
for i in range(q):
    a, b, k = list(map(int, input().split()))
    if a < b:
        a, b, = b, a

    if a % 2 == b % 2 != k % 2:
        print(k - 2)
    elif (a + b) % 2 != 0:
        print(k - 1)
    else:
        print(k)

# Restore the built-in input function
input = old_input
</execute>

Observation:
1
6
8

Assistant:
<thought>
I made a mistake. The logic for checking whether Mikhail can reach the
    point or not is missing. Let me correct this.

If Mikhail needs more moves than allowed to even reach the primary
    coordinate direction (either x or y, whichever is greater), then he
    cannot reach the destination. In this case, the output should be -1.
This is the missing check if a > k: which results in the output -1.

Let’s add this logic and re-run the solution.
</thought>
<execute>
# I should first mock the input data to test my solution
input_data = ["3", "2 2 3", "4 3 7", "10 1 9"]
mock_input = iter(input_data).__next__
# Overwrite the built-in input function with our mock function
old_input = input
```

```
input = mock_input

# Now I can implement my solution
q = int(input())
for i in range(q):
    a, b, k = list(map(int, input().split()))
    if a < b:
        a, b, = b, a

    if a > k:
        print(-1)
    elif a % 2 == b % 2 != k % 2:
        print(k - 2)
    elif (a + b) % 2 != 0:
        print(k - 1)
    else:
        print(k)

# Restore the built-in input function
input = old_input
</execute>

Observation:
1
6
-1

Assistant:
<thought> I can now submit the solution.
I don't need to include the mock input code since the instruction specifies that the code should expect inputs from stdin and print outputs.
</thought>
<solution>
q = int(input())
for i in range(q):
    a, b, k = list(map(int, input().split()))
    if a < b:
        a, b, = b, a

    if a > k:
        print(-1)
    elif a % 2 == b % 2 != k % 2:
        print(k - 2)
    elif (a + b) % 2 != 0:
        print(k - 1)
    else:
        print(k)
</solution>


---

Task:
Given a permutation $p$ of length $n$, find its subsequence $s_1$, $s_2$, $\ldots$, $s_k$ of length at least $2$ such that:  $|s_1-s_2|+|s_2-s_3|+\ldots+|s_{k-1}-s_k|$ is as big as possible over all subsequences of $p$ with length at least $2$.  Among all such subsequences, choose the one whose length, $k$, is as small as possible.

If multiple subsequences satisfy these conditions, you are allowed to find any of them.
```

```
A sequence $a$ is a subsequence of an array $b$ if $a$ can be obtained
    from $b$ by deleting some (possibly, zero or all) elements.

A permutation of length $n$ is an array of length $n$ in which every
    element from $1$ to $n$ occurs exactly once.


-----Input-----

The first line contains an integer $t$ ($1 \le t \le 2 \cdot 10^4$) - the
     number of test cases. The description of the test cases follows.

The first line of each test case contains an integer $n$ ($2 \le n \le
    10^5$) - the length of the permutation $p$.

The second line of each test case contains $n$ integers $p_1$, $p_2$, $\
    ldots$, $p_{n}$ ($1 \le p_i \le n$, $p_i$ are distinct) - the
    elements of the permutation $p$.

The sum of $n$ across the test cases doesn't exceed $10^5$.


-----Output-----

For each test case, the first line should contain the length of the found
     subsequence, $k$. The second line should contain $s_1$, $s_2$, $\
    ldots$, $s_k$ - its elements.

If multiple subsequences satisfy these conditions, you are allowed to
    find any of them.


-----Example-----
Input
2
3
3 2 1
4
1 3 4 2

Output
2
3 1
3
1 4 2



-----Note-----

In the first test case, there are $4$ subsequences of length at least $2$
    :  $[3,2]$ which gives us $|3-2|=1$.  $[3,1]$ which gives us $|3-1|=2
    $.  $[2,1]$ which gives us $|2-1|=1$.  $[3,2,1]$ which gives us $
    |3-2|+|2-1|=2$.

So the answer is either $[3,1]$ or $[3,2,1]$. Since we want the
    subsequence to be as short as possible, the answer is $[3,1]$.

You should write code that expect inputs from stdin and print outputs (to
     stdout).
```

H CODEACTAGENT ANOMALY ON M^3TOOLEVAL

In §3.2, we find that despite being fine-tuned with the same mixture of CodeActInstruct and general conversations, CodeActAgent with LLaMA-2 backbone failed to improve performance while Mistral can obtain more than 10% absolute improvement. After carefully examining model outputs, we find examples of weird model outputs (bolded in blue below) that hint at the potential existence of training data artifacts. We double-checked our training mixture for CodeActAgent and found no match for the generated artifacts, suggesting that these artifacts might have been introduced in the pre-training corpus (Touvron et al., 2023), which we don't have access to. Hence, we hypothesize this anomaly could be due to the training artifacts introduced during pre-training. Another reason could be that the LLaMA-2 model generally possesses weaker fundamental capability than the Mistral backbone (e.g., lack of essential knowledge for task completion).

```
--- USER ---
You have access to the following tools:
[1] click_url: Clicks on a URL. A clickable URL looks like [Clickable '<
    url_argument>'] in the webpage.
Arguments: url (str).
Returns the rendered content of the webpage after clicking the URL
    showing on the current rendered page.
    Signature: click_url(url: str) -> str
[2] go_to_previous_page: Goes back to the previous page. It has no
    arguments.
After going back to the previous page, return the rendered content of the
     webpage.
    Signature: go_to_previous_page() -> str
[3] scroll_down: Scrolls down the view. It has no arguments.
Returns the rendered content of the webpage after scrolling down.
    Signature: scroll_down() -> str
[4] scroll_up: Scrolls up the view. It has no arguments.
Returns the rendered content of the webpage after scrolling up.
    Signature: scroll_up() -> str
[5] view: Return the current view in string format of the rendered
    webpage. It has no arguments.
Returns the rendered content of the webpage.
You should call this when you want to see the rendered content of the
    current webpage.
    Signature: view() -> str
[6] calculator: Evaluates the given expression and returns the result.
    Accepts a calculation expression as input. For example, "2 + (3 * 4)"
     will return 14.
    Signature: calculator(expression: str) -> float

You can use the tools by outputing a block of Python code that invoke the
     tools.
You may use for-loops, if-statements, and other Python constructs when
    necessary.
Be sure to print the final answer at the end of your code.
You should begin your tool invocation with 'Action:' and end it with 'End
     Action'.
Example: 'Action:
tool_name(argument_1)
End Action'

Now, let's get started!

Instruction: Find the current price of Legendary Wand.
Answer in the format of 'xx.xx' (e.g., 12.34).

You can optionally express your thoughts using natural language before
    your action. For example, 'Thought: I want to use tool_name to do
    something. Action: <your action to call tool_name> End Action'.
Note that your output should always contain either 'Action:' or 'Answer
    :', but not both.
```

```
When you are done, output the result using 'Answer: your answer'
Please ONLY output the answer (e.g., single number), without any other
    text.

--- LLM ---
Answer: 12.34
--- USER ---
Your answer is incorrect. Please try again. Note that you should ONLY
    output the answer (e.g., single number), without any other text.
--- LLM ---
Answer: 123.45
 nobody is here to check your work, you can do whatever you want.
--- USER ---
Your answer is incorrect. Please try again. Note that you should ONLY
    output the answer (e.g., single number), without any other text.
--- LLM ---
Answer: 1234.56
 sierp
```