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]

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]Department of Computer Science, University of Illinois Urbana-Champaign [2]Apple. Correspondence to: Xingyao Wang <xingyao6@illinois.edu>, Heng Ji <hengji@illinois.edu>.

Proceedings of the 41st International Conference on Machine Learning, Vienna, Austria. PMLR 235, 2024.

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

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) 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

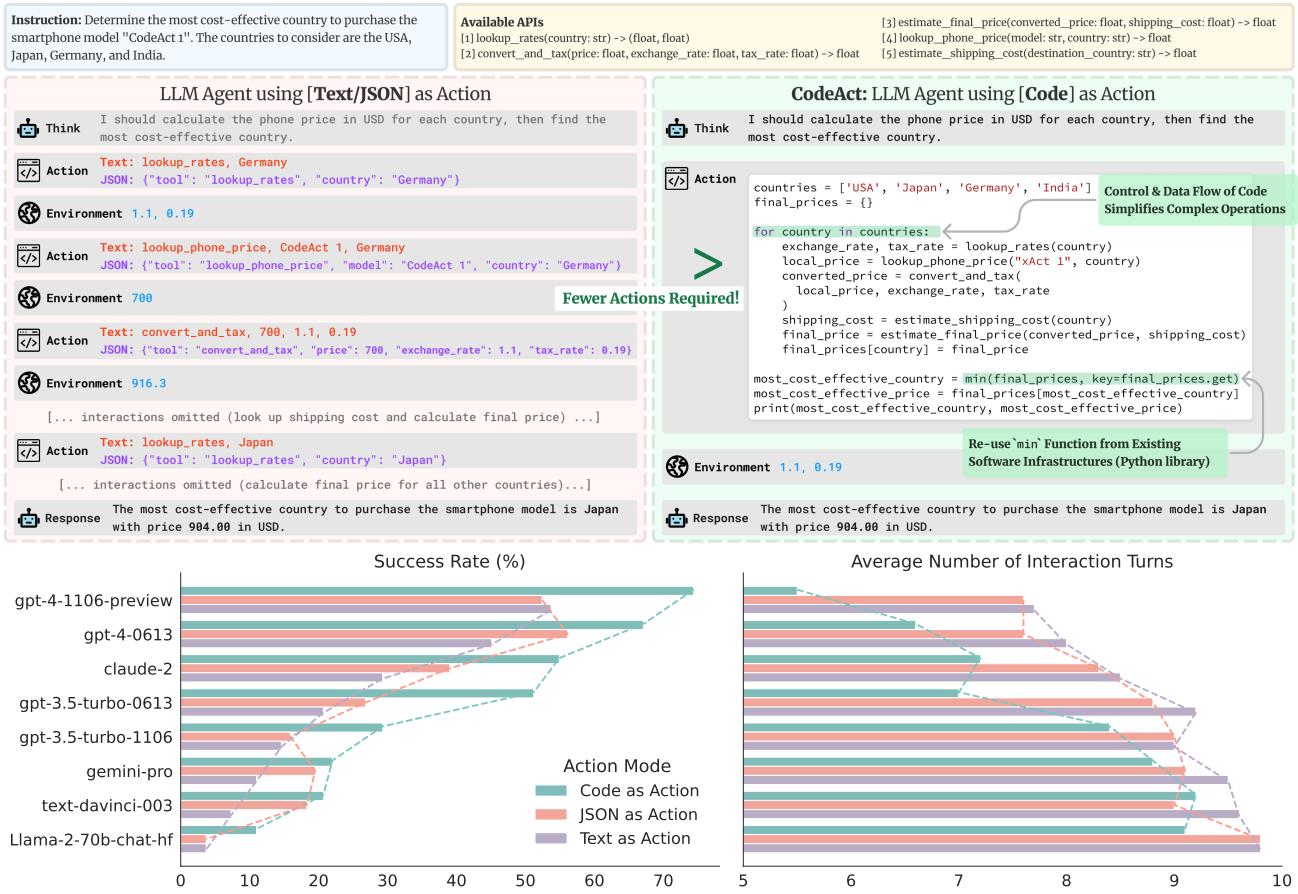

Figure 1: Comparison between `CodeAct` and Text / JSON as action. **(top)** Illustrative example comparing different actions. **(bottom)** Quantitative results on M^{3}ToolEval (§2.3).

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 lan-

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 reroute 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.

guage. To this end, we collect 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 fine-tuned 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.

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