ToolReflection: Improving Large Language Models for Real-World API Calls with Self-Generated Data

Gregory Polyakov[1]*, Ilseyar Alimova[2]*, Dmitry Abulkhanov[3]*, Ivan Sedykh[4]*, Andrey Bout[5]*, Sergey Nikolenko[6], Irina Piontkovskaya[7]
[1]University of Tübingen, [2]Skoltech, [3]Independent, [4]MTS AI, [5]Yandex,
[6]St. Petersburg Department of the Steklov Institute of Mathematics,
[7]AI Foundation and Algorithm Lab

Abstract

While open-source large language models (LLMs) have advanced in leveraging third-party tools, significant challenges remain in real-world API usage, where behavior is unpredictable or poorly specified. Existing benchmarks often fail to capture this complexity. We propose *ToolReflection*, a novel method that improves LLMs' ability to self-correct API calls by utilizing real-time API feedback. We also introduce new datasets specifically designed to test model performance under realistic conditions. In *ToolReflection*, models undergo instruction tuning on a dataset augmented with self-generated errors and corrections. Our evaluation across ToolAlpaca, ToolBench benchmarks, and three newly developed datasets (GPT4Tools-OOD, GPT4Tools-OOD-Hard, and Multistep-100) demonstrates its effectiveness. *ToolReflection* boosts overall success rates by 25.4% on GPT4Tools-OOD, 56.2% on GPT4Tools-OOD-Hard, and 4% on Multistep-100, outperforming original models. On ToolAlpaca, we show a 14% improvement in the "Simulated" setting and 10.5% in the "Real-world" scenario. Our error analysis highlights *ToolReflection* significantly enhances recovery from incorrect tool calls, even with incomplete or erroneous API documentation. We have released the code, prompts, and data at https://github.com/polgrisha/ToolReflection.

1 Introduction

Modern LLMs excel at various tasks, including text generation, coding, question answering, and ranking (Zhao et al., 2023; Minaee et al., 2024). However, the knowledge LLMs gain during pretraining is often inadequate for tasks requiring precise algorithmic reasoning, such as arithmetic, or those dependent on real-time data, for example, travel planning or weather updates. To bridge these gaps,

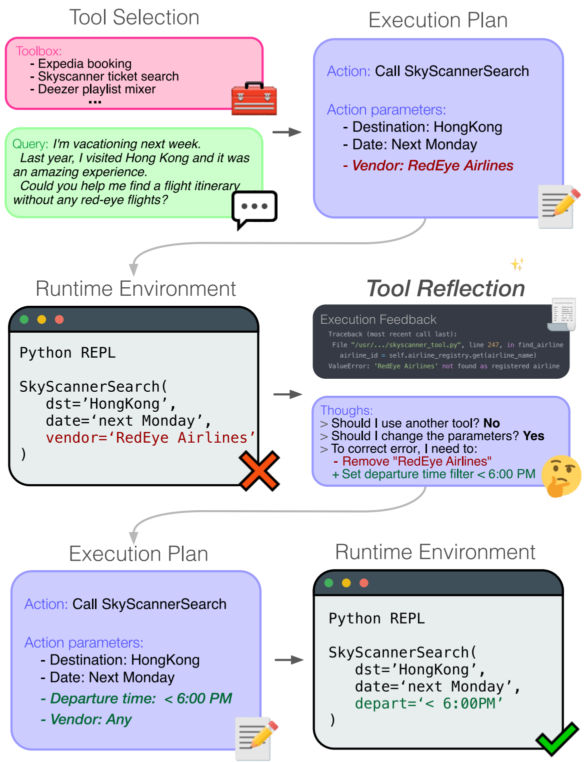

Figure 1: Overview of *ToolReflection*. Given a retrieved set of tools, the model fine-tuned with ToolReflection selects a tool and its parameters, generates an API call, and processes execution feedback. If an error occurs, it provides a rationale and adjusts the call accordingly.

LLMs have been extended with third-party tools, allowing them to execute complex actions using API documentation included in their prompts.

Significant research has focused on improving LLMs' tool-usage capabilities. Advances include instruction-tuned models for better tool calls (GPT4Tools (Yang et al., 2023), ToolAlpaca (Tang et al., 2023), Gorilla (Patil et al., 2024)) and sophisticated external frameworks for multi-step reasoning and planning (ToolChain* (Zhuang et al., 2024), ToolLLaMA with depth-first search-based decision tree (DFSDT) (Qin et al., 2023)). However, even with higher-level orchestration strategies,

*Work done while at AI Foundation and Algorithm Lab.

Proceedings of the 1st Workshop for Research on Agent Language Models (REALM 2025), pages 184–199
July 31, 2025 ©2025 Association for Computational Linguistics

the underlying ability of LLMs to robustly handle individual API invocations in real-world settings remains a considerable challenge. Models frequently hallucinate tool names or parameters and fail to recognize and correct API call errors. Existing solutions to these granular errors, like guiding models with a finite state machine (Zhang et al., 2024), simplifying tool documentation (Yuan et al., 2024), or incorporating external model feedback (Wang et al., 2024b), are often impractical due to computational demands or the need to rewrite tool documentation.

In this work, we explore tool-augmented LLMs' ability to self-correct using feedback directly from invoked tools. We focus on two key challenges: (1) the lack of diverse, high-quality benchmarks that reflect real-world scenarios and (2) models' frequent failure in real-world tasks. To address these challenges, we propose two solutions. First, we extend existing datasets and develop new, annotated evaluation sets that better mimic real-world conditions, specifically GPT4Tools-OOD and Multistep-100 based on GPT4Tools and ToolBench respectively. Second, we enhance the models' self-correction abilities by introducing a fine-tuning phase using self-generated examples of errors, tool responses, and corresponding corrections. Our approach, evaluated on GPT4Tools, ToolAlpaca, and ToolBench, shows consistent improvements across all settings. Fig. 2 provides examples of how our method successfully corrects tool usage errors. Notably, even curated API sets often contain incomplete or incorrect documentation, making self-correction after an error message the only viable solution.

Our main contributions are: **(1)** We analyse three existing benchmarks for tool use and propose improvements to make them better adapted to real-world tools, including the training dataset (GPT4Tools), evaluation datasets (GPT4Tools, ToolBench), data cleaning (ToolBench), and fixes in dataset format and the evaluation approach (GPT4Tools, ToolAlpaca). **(2)** We provide a method to improve self-correction abilities of LLMs after getting error messages from the external tool via additional fine-tuning on self-generated examples with error corrections. **(3)** We evaluate and prove the effectiveness of our approach with a comprehensive experimental study, which demonstrates that the ability to correct the output based on external error feedback is necessary for successful communication with third-party APIs.

The rest of the paper is structured as follows: Section 2 surveys related work, Section 3 discusses benchmark improvements, Section 4 introduces our *ToolReflection* method, Section 5 presents experimental results, Section 6 provides error analysis, and Section 7 concludes the paper.

2 Related work

Tool-augmented language models Significant research has been devoted in recent years to enhancing the tool invocation capabilities of LLMs. The *Toolformer* model (Schick et al., 2023) showed that additional model fine-tuning on API calls in a self-supervised way leads to improvements in zero-shot performance of LLMs on downstream tasks. Based on this idea, GPT4Tools (Yang et al., 2023) fine-tuned compact models to incorporate multi-modal tools and evaluated the tool usage accuracy on an automatically generated benchmark. The ToolAlpaca framework (Tang et al., 2023) has been designed to address the issue of tool calling abilities on previously unseen tools by massively pretraining on a highly diversified tool use corpus. The ToolLLaMA model (Qin et al., 2023) took a step towards applying real-world APIs from the *RapidAPI Hub*. ToolLLaMA demonstrates a remarkable ability to execute complex instructions and generalize to unseen APIs. To sum up, the studies of tool enhanced LLMs have mostly focused on creating tool use datasets for fine-tuning, while methods that construct chains of thought leading to the correct answer for tool calling remains problematic. However, recent studies address this challenge via external algorithms and structures. For instance, ToolLLaMA proposes to improve its reasoning strategy with DFSDT by allowing the backbone model to choose between different reasoning chains using a tree structure. *ToolChain** (Zhuang et al., 2024) leverages the A* search algorithm for the same purpose. Some works also explore ways to improve the quality of tool-based language models with external algorithms such as finite state machines (Zhang et al., 2024) or feedback from an external model (Wang et al., 2024b). In contrast, our *ToolReflection* approach focuses on a complementary aspect: directly enhancing the LLM's intrinsic ability to understand and self-correct individual API calls using real-time feedback. While these external frameworks could potentially benefit from a base model improved by *ToolReflection*, our primary focus is on enhancing the model's direct API engagement.

Figure 2: Examples of successfully resolved mistakes after fine-tuning on self-generated examples of error corrections. Model thoughts are shown in blue; model API calls after thoughts, in light; unsuccessful API responses, in red; successful API responses, in green.

Self-correction from feedback Attempts to incorporate self-correction to LLMs have led to improvements across a variety of tasks including question answering (Shinn et al., 2023), reasoning (An et al., 2024), code generation (Zhang et al., 2023), and summarization (Liu and Liu, 2021). Self-correcting models apply two types of feedback: *self-feedback* — feedback obtained from the LLM itself, and *external feedback* — feedback derived from external models, tools, or knowledge sources. Self-feedback can involve asking the model itself to evaluate the quality of generated outputs via prompting (Madaan et al., 2023) or additional fine-tuning of the model on automatically generated self-correction samples (Ye et al., 2023). The approach of learning from external feedback is widely used for code generation tasks since it is relatively easy to receive such feedback through the execution of generated code with the corresponding compilers or interpreters (Wang et al., 2024a; Chen et al., 2023). The TRICE framework (Qiao et al., 2024) recently proposed techniques for learning from execution feedback in tool-enhanced models, enabling a form of self-correction. While it's a notable step forward, TRICE mainly focuses on single-step mathematical and question-answering tasks using simple tools, where "execution feedback" mostly refers to the correctness of the final, single-answer output. Our *ToolReflection* approach builds on similar self-correction ideas but applies them to API calls closer to real-world usage. It focuses on direct, real-time error messages and structured responses from tool invocations.

3 Datasets analysis and extensions

In this work, we focus on zero-shot realistic tool support, where tools are provided as APIs or Python functions with natural language descriptions of their usage and parameters. At each step, the LLM is presented with several unseen tool descriptions and must select the appropriate tool and fill in its parameters. This task is typically solved via instruction tuning, training models on datasets of tool use examples formatted with dataset-specific inputs, chain-of-thought steps, and expected outputs. Each dataset includes its own tool execution and evaluation framework. Table 1 provides an overview of the main datasets we consider in this work. We identify key issues in these datasets and their evaluation procedures, proposing methods to better align them and the models trained on them with real-world usage scenarios.

However, existing benchmarks present several issues. First, they often rely exclusively on synthetic data and assess models in environments without real tool feedback. Second, some benchmarks, which already use real-world tools, lack annotated data; for instance metrics used in ToolBench, including pass and win rates, are calculated via LLM

Dataset	APIs	Train	Eval	Real Eval	Chain
GPT4Tools	23	71.4K	1170+652	✗	1.0
ToolAlpaca	426	3.9K	100+100	✔	1.7
ToolBench	16.5K	120K	200x5+100	✔	4.0

Table 1: Comparison of datasets based on the number of APIs, train size, eval size, presence of real tools in eval, and average tool chain length.

Model	Accuracy	Precision	Recall
Llama2-7B-chat (FS)	0.81	0.73	0.78
Llama2-7B-chat (ZS)	0.89	0.96	0.72

Table 2: Performance of LLaMA2-7B-chat on the error detection task in HTTP responses. ZS — zero-shot setting; FS — few-shot setting.

prompting, making results hard to reproduce and prone to variability. LLM-based evaluation may also miss issues arising in real tool communication. In this work, we aim to provide realistic evaluation with callable tools for every setup.

For each dataset, we use its original prompting with minor adjustments. For GPT4Tools and ToolBench, we create novel evaluation sets enabling actual tool invocation and exact output checks. For GPT4Tools, we also generate additional training data. Details on each dataset are discussed below.

3.1 GPT4Tools

GPT4Tools (Yang et al., 2023) is one of the earliest datasets featuring tool usage, containing 71.4K instruction-following examples, 35.7K of which involve tools. It includes a limited set of 23 tools primarily focused on visual tasks like face detection, image generation, and object removal. While effective for specialized visual tasks, these APIs do not represent the broader range of real-world tools. Moreover, tool parameters in GPT4Tools are limited to simple text strings, restricting the dataset's ability to model complex tool interactions in real-world applications.

Although GPT4Tools provides callable tools, its evaluation is limited to single-step invocation and neither extends to more nuanced API interactions nor reflects the real success of tool calls. For example, in image generation tasks, the framework does not check whether the content of the generated image aligns with the user's intent. To adapt GPT4Tools to more realistic tool interactions, we implemented several improvements: we refined the prompt format for better evaluation, created a training dataset containing a wider range of tools with more diverse tool signatures, and developed a test set with callable tools, ensuring that outputs can be verified against the intended results. Further details are provided below.

Prompt format correction Examples from the original dataset are organized so that each tool call begins with the tool name followed by a list of parameters. While suitable for simple textual parameters, this format may encounter parsing issues with complex parameters like lists or floating-point numbers. To address this, we convert the format into Python function calls and utilize the Python interpreter to parse tool calls. Our new instruction prompt can be found in Appendix G.

Since GPT4Tools includes only 23 tools related to image editing or generation, which typically require textual descriptions or paths to images as parameters, we propose to extend it with new synthetic but realistic tools to evaluate whether diverse synthetic data improves out-of-domain quality.

We start with GPT4Tools tool descriptions as a small seed set. In each iteration, we uniformly sample a batch of random descriptions and prompt ChatGPT (OpenAI, 2022) to generate more. To ensure diversity, we retain only those with a ROUGE-L similarity below 0.7 to any existing descriptions. This process resulted in 141 diverse tool descriptions. We then apply the same procedure to generate tool usage examples, reformat them as Python function calls, and clean non-parsable or incorrect cases. The final dataset contains 636 instances. Prompts for tool descriptions, queries, and usage examples are in Appendix D. Since the original GPT4Tools training set has 35.7K tool usage items, we sample each generated instance five times, producing 3180 additional samples. We refer to this dataset as GPT4FakeTools.

GPT4Tools-OOD and GPT4Tools-OOD-Hard To evaluate performance in a realistic setup, we selected five open-source tools callable via APIs: measurement conversion, time conversion, geolocation information, nutrition analysis, and flight schedules. We converted these APIs into executable Python functions, manually wrote descriptions, and created a set of queries that implicitly require their invocation. The resulting dataset contains 89 instances, with two versions: GPT4Tools-OOD and GPT4Tools-OOD-Hard. The latter omits tool usage examples in the prompt to test model behavior in the case of poorly annotated documentation. Examples are in Appendix F.

We evaluate on GPT4Tools-OOD and GPT4Tools-OOD-Hard as follows: for each query, the model has three attempts to call the tool correctly. We report success rates after the first ($\mathrm{SR}_{\mathrm{first}}$) and last attempt ($\mathrm{SR}_{\mathrm{last}}$). For some tools that accept multiple equivalent valid inputs, we consider a tool call successful if the returned answer is correct.

3.2 ToolAlpaca

ToolAlpaca (Tang et al., 2023) made a step forward by introducing a framework for simulating API responses. The dataset contains 3.9K tool usage instances from 400+ real-world APIs across 50 categories. It was created using a simulation environment with three agents: a user generating instructions, an assistant choosing tools, and a tool executor simulating feedback, all emulated by the language model. Emulating tool feedback with an LLM has led to one of ToolAlpaca's main limitations: the tool provides an output regardless of any errors occurring during the API call. The authors included a function to check format correctness, but it was disabled in the evaluation code.

Tasks in this dataset typically require 1–2 steps to solve. The training set includes some examples where the model receives an error from the tool simulator (e.g., "Response 404"), prompting it to make another tool call to obtain a correct response. In addition to the emulated test set, where tool calls are simulated and evaluated by the LLM agent, *ToolAlpaca* provides a small test set of curated, callable tools for a more realistic evaluation.

Changes in the Format Check Procedure The *ToolAlpaca* framework includes a rule-based tool format checker used during training data generation but disabled in evaluation. In training data generation, this checker runs before LLM-based evaluation, providing a structured response for API name or parameter errors.

We modify this procedure in two ways. First, we enable the format check in evaluation, which lowers scores on the simulated test set but does not affect the success rate in realistic evaluation. Second, we unify error messages in the training set by converting the format checker's responses into HTTP-style errors. Our experiments demonstrate that this translation is necessary to adapt the model to format-related responses from real tools, especially when the tool description is incomplete.

3.3 ToolBench

ToolBench (Qin et al., 2023) represents an effort to create a dataset grounded in real APIs, increasing both the diversity of APIs and the complexity of tasks compared to other benchmarks. The authors collected over 16,000 REST APIs from the RapidAPI Hub and used them to generate synthetic instructions by prompting ChatGPT, which also generated the corresponding solutions. However, this fully automatic dataset creation without human verification leads to several issues. First, errors returned by tools due to incorrect calls are not handled, so the training set contains examples where the model repeatedly encounters errors but fails to rectify them. Second, the wide diversity of APIs makes it difficult to consistently identify steps where tools return errors, as each API has its own format. Finally, relying on ChatGPT's judgment for evaluation lacks rigor and can lead to inaccurate assessments of model performance. To address these issues, we cleaned the dataset by filtering incorrect tool usage examples, implemented error-detection mechanisms, and created an evaluation set with more challenging tool use cases and a proper evaluation procedure.

Dataset Cleaning We use the latest dataset version, which contains nearly 120K tool invocation chains with reasoning traces. Since all solutions were generated by ChatGPT, annotation errors are possible. We filter out incomplete chains, those with "give up and restart" messages, and those containing tool errors. However, due to varying tool response formats, rule-based error detection remains challenging. Therefore, we apply LLM-based filtering and assess its effectiveness on a manually curated test set of 100 examples (36 with HTTP tool errors, 64 without). Table 2 shows that Llama-2-7B-chat (Touvron et al., 2023) achieves strong performance in zero-shot and few-shot settings, so we leverage it for error detection.

Multistep-100 The most challenging tasks require multiple steps of tool invocation, where each step depends on the previous result. Evaluating these tasks is difficult without actual tool outputs. To address this, we construct fully annotated test sets supporting multi-step tool calls. We select 16 APIs from ToolBench, absent from the training set, that can be chained to solve a single task. Then, we manually collect 10 query templates with miss-

Method	Seen SR_t	Seen SR_{act}	Seen SR_{args}	Seen SR	Unseen SR_t	Unseen SR_{act}	Unseen SR_{args}	Unseen SR	OOD SR_{first}/ SR_{last}	Hard OOD SR_{first}/ SR_{last}
GTP4Tools	98.7	97.6	91.4	94.1	98.2	97.0	92.2	90.6	45.9 / 47.5	6.7 / 6.7
+GPT4FakeTools	**99.8**	**98.9**	**93.6**	**98.0**	98.8	**98.0**	**96.1**	**95.7**	**72.9 / 72.9**	40.4 / 40.4
ToolReflection										
Post-finetune	99.4	97.7	91.9	96.3	**99.3**	97.5	93.4	93.2	69.4 / 70.5	33.6 / 57.3
Fine-tune	99.2	97.9	91.8	96.2	98.9	97.7	95.0	94.3	**72.9 / 72.9**	**44.9 / 62.9**

Table 3: Results on seen and unseen test sets from GPT4Tools and in out-of-domain settings.

ing parameter values, requiring multiple steps to complete; sample templates are in Appendix H. We ask LLaMA-2-7B to rephrase these queries, generating 100 examples. Next, we create a table of actual parameters (names, IDs, numerical values) and randomly fill in missing values using these parameters in the query templates. Query templates and ground-truth tool invocation chains can be found in Appendix H.

4 ToolReflection

In the frameworks we considered, tools can be implemented as Python functions (e.g., in GPT4Tools) or as external API calls via HTTP (e.g., in ToolBench and ToolAlpaca). If an error occurs, the response often contains valuable semantic information, such as the error type, description, or the function where it was triggered. Although response formats vary, we hypothesize that LLMs can leverage this feedback to improve their outputs. Our experiments in Sections 5 and 6 support this intuition, showing that (i) most errors generate feedback that the model can use for corrections, and (ii) many errors cannot be resolved without this feedback since tool documentation is often incomplete. This proves that understanding tool feedback is essential for accurate problem-solving.

To enhance this capability, we propose the *ToolReflection* method, applicable across frameworks. The core idea of *ToolReflection* is to leverage the model's own errors and subsequent corrections for additional fine-tuning. The key novelty of our approach lies not in the fine-tuning strategy itself, but in the *source* of these error-correction pairs: feedback obtained directly from the model's self-interactions with APIs or executable Python functions.

Generating a dataset with error examples involves three steps. First, we collect queries from existing datasets that require tool use. Second, we run a pretrained model on these queries and collect samples where it made tool invocation errors. We use feedback from the Python interpreter or HTTP responses to identify errors. Finally, we form a new example containing: (i) an incorrect tool call, (ii) feedback from the tool, and (iii) the correct tool call. We integrate this feedback into the model's reasoning process as a *self-reflection* step (see Fig. 1 and Fig. 2 for examples). After generating this dataset, we fine-tune the model on it and test it on out-of-domain examples (see below).

For synthetic datasets without executable tools, we create dummy functions that check parameter correctness, simulating feedback from tool calls.

We propose two setups for *ToolReflection*. In both, we start with a model fine-tuned on tool usage instructions and use it to generate error correction examples. Then, we either (i) further fine-tune on these examples with a smaller sample of original data (Post-finetune), or (ii) augment the original data with error corrections and fine-tune from scratch (Fine-tune).

5 Experiments

5.1 GPT4Tools

We conducted experiments using the original seen and unseen test sets from GPT4Tools, adjusting prompts and tool calls to our format (see Section 3.1 and Appendix G).

Following the GPT4Tools setup, we evaluate several metrics: SR_t (Success Rate of Thoughts, i.e. the accuracy of decisions whether to use tools or not), SR_{act} (Success Rate of Actions, accuracy of tool names), SR_{args} (Success Rate of Arguments, accuracy of tool arguments), and SR (overall Success Rate). Details regarding the training procedure, including the base model and hyperparameters, are provided in Appendix A.1.

Table 3 shows the results. First, we present the results on seen and unseen GPT4Tools evaluation sets ("Seen" and "Unseen" columns in Table 3). Adding GPT4FakeTools to the training data improves all metrics and, importantly, the overall SR by a large margin.

Next, the "OOD" columns present the results on our collected GPT4Tools-OOD evaluation dataset. The model fine-tuned only on GPT4Tools performs poorly on out-of-domain test sets, achieving just 6.7% SR in the hardest setting with tool invocation examples in the prompt ("Hard OOD"). Adding a single example ("OOD") improves SR to 47.5%. Fine-tuning on both GPT4FakeTools and GPT4Tools significantly boosts performance—by absolute 25.4% in OOD and 33.7% in Hard OOD. Models trained without error correction fail to improve after several iterations of calling and obtaining feedback from the same tool, as seen in $\mathrm{SR}_{\mathrm{first}}$ and $\mathrm{SR}_{\mathrm{last}}$ scores.

Finally, we evaluate the effect of our *ToolReflection* approach. We test two variations of *ToolReflection*: "Post-finetune" and "Fine-tune" (see Section 4). While *ToolReflection* outperforms the GPT4Tools baseline on the original Seen and Unseen evaluation sets, its results are slightly worse than fine-tuning only on GPT4Tools+GPT4FakeTools without error correction examples. However, since the GPT4Tools test set lacks real callable APIs, the effect of *ToolReflection* cannot be fully observed. This experiment primarily ensures *ToolReflection* does not degrade quality on the original test set; more realistic results appear in the last two out-of-domain columns.

Incorporating error correction examples into instruction tuning data significantly improved performance on GPT4Tools-OOD-Hard ("Hard OOD"): final SR increased by 16.9% after post-finetuning the model already trained on GPT4Tools and GPT4FakeTools on self-generated corrections ("Post-finetune") and by 22.5% when fine-tuned from scratch ("Fine-tune"). The difference between first and last tool call accuracy (23.7% in post-finetune, 18% in fine-tune) shows that *ToolReflection* primarily enhances the model's ability to recover from initial errors.

5.2 ToolAlpaca

We follow the setting of ToolAlpaca and evaluate our approaches on its original simulated and real-world datasets. We measure procedure accuracy ("Proc." in Table 4), which evaluates action and parameter selection, response accuracy ("Resp."), which measures whether the final response satisfies the user request, and overall accuracy ("Acc."), which requires both to be correct.

The ToolAlpaca corpus contains 3.9K instances.

Method	Simulated			Real-world		
	Proc.	Resp.	Acc.	Proc.	Resp.	Acc.
Original ToolAlpaca						
Train + Spec check	44.0	49.0	42.0	37.7	36.8	35.1
ToolReflection						
Post-finetune						
Synth on val	51.0	54.0	<u>49.0</u>	44.7	43.4	43.0
Synth on train+val	<u>52.0</u>	53.0	<u>49.0</u>	47.4	42.1	40.4
Fine-tune						
Synth on val	**57.0**	**66.0**	**56.0**	<u>50.8</u>	<u>47.4</u>	<u>43.9</u>
Synth on train+val	<u>52.0</u>	<u>61.0</u>	<u>49.0</u>	**52.6**	**48.2**	**45.6**

Table 4: Results on simulated and real-world test sets from ToolAlpaca.

We split its training set into training (2,261 examples) and validation (1,676 examples) subsets, selecting the first 268 distinct API functions for training and the remainder for validation.

We train the model on the training subset and generate examples with error corrections on the training and validation parts of the dataset. For *ToolReflection* experiments, we use errors detected by the internal format checker to generate self-correction examples, including parsing issues, incorrect API names, and incorrect API parameter names or types. In addition to Post-finetune and Fine-tune setups described in Section 4, we explore generating synthetic data from either the validation set only ("Synth on val" in Table 4) or both training and validation subsets ("Synth on train+val"). Further details regarding training hyperparameters are provided in Appendix A.2.

As a baseline, we reproduce the original pipeline with the training set reduced as above and include the parameter specification checker from ToolAlpaca code ("Train + Spec check"). This checker is necessary for simulated tools, as we cannot ensure that the tool simulator (ChatGPT) follows the tool description.

Table 4 shows that our *ToolReflection* approach consistently outperforms the original setup with the parameter checker, improving accuracy by 14 % on simulated and 10.5 % on real-world datasets.

Similar to GPT4Tool results, Fine-tune outperforms Post-finetune, suggesting that mixing standard data with error reflection chains is more effective than separate fine-tuning stages.

For this fine-tuning approach, using a larger correction dataset ("Synth on train+val") slightly improves real-world performance. On synthetic evaluation data ("Simulated"), separating error correction and standard datasets ("Synth on val") leads to a surprising quality boost. Because this increase does not generalize to the real setting ("Real-world"), we believe that the model has managed to learn a specific behaviour of the simulated evaluation in this case (see also Section 6).

Overall, *ToolReflection* training significantly improves real-world performance. The setup with mixed training on a larger set of errors (Fine-tune, train+val) slightly outperforms others.

5.3 ToolBench and Multistep-100

We follow the original ToolBench setup, measuring Pass Rate (Passes; percentage of succesfully completed user instructions) and Win Rate (Wins; comparison of model solution paths to ChatGPT results) across six evaluation sets (L1-Inst, L1-Tool, L1-Cat, L2-Inst, L2-Cat, L3-Inst). Details regarding training procedure and hyperparameters are provided in Appendix A.3.

To apply the *ToolReflection* pipeline to ToolBench, we need clean tool invocation chains without any errors. To obtain a clean subset, we took the already cleaned dataset and used the Llama-2-7B-chat model to eliminate chains with errors resulting from calling external tools (see also Section 3.3). We then collected all distinct tools, allocating 1,629 to the training set and 702 to validation. To prevent overlap, we split the training set into train and validation based on tools in these datasets.

We also evaluate on our Multistep-100 benchmark. Unlike ToolBench, tool invocation steps in this benchmark are interdependent, so it is sufficient to check the correctness of the final tool calls alone. Thus, we measure the success rate (SR) of the last tool call.

Figure 3 and Table 5 show the results on the ToolBench and Multistep-100 evaluation sets. On the original ToolBench test set, the *ToolReflection* pipeline improves the pass rate by 1.7% but does not affect win rate. However, on the Multistep-100 dataset the overall success rate increases by 4%.

6 Error Analysis

In this section, we analyze the types of errors made by the model, some of which were corrected using *ToolReflection*, while others were not. Tool calls can be generally categorized into three types: (1) a correct tool call; (2) an incorrect tool call that produces error feedback from the tool; (3) a formally correct tool call with no negative feedback, but with wrong choice of tool or parameters. Both the second and third types can be considered errors. Our method specifically addresses errors where the tool (API or code snippet) returns non-fatal feedback, focusing on correcting calls of the second type.

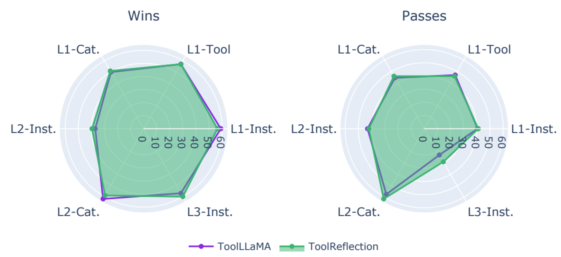

Figure 3: ToolBench results comparing ToolReflection against ToolLLaMA baseline. *Left:* Win Rate (model solutions vs. ChatGPT). *Right:* Pass Rate (successfully completed user instructions). Radial axes represent performance across six ToolBench evaluation sets (L1-Inst, L1-Tool, L1-Cat, L2-Inst, L2-Cat, L3-Inst).

Table 5: Average results on ToolBench and MultiStep-100 evaluation sets.

Model	ToolBench Passes	ToolBench Wins	Multistep SR
ToolLLaMA	42.6	53.7	0.13
+*ToolReflection*, Synth on val, Post-finetune	**44.3**	**53.8**	**0.17**

Table 6 shows that in the GPT4Tools + GPT4FakeTools setup, 9% of the calls fell into the third category, which is a relatively small fraction. In contrast, 51% of the calls belong to the second type, where feedback from the tool was provided. After applying *ToolReflection* fine-tuning, the number of calls of the second type decreased by an absolute 26%.

Fig. 2 (left panel) illustrates an important use case which is successfully handled by the *ToolReflection*-trained model. Here, the generated call is semantically correct, but API requires strict input format, which can be extracted from the API response: the `pintapi_convert_units` tool accepts `Pa`, but not `Pascals`.

For further details please refer to Appendix B.

7 Conclusion

In this work, we aim to improve tool invocation by LLMs, a core technology in next-generation virtual assistants and AI applications. Modern LLMs support a paradigm where the backbone model manages user communication and invokes external tools as needed. For this paradigm to succeed, the toolset must be extensible: an instruction-tuned backbone should support new tools based on a simple paragraph of API documentation, learned in context.

We demonstrated that existing solutions struggle with complex real-world APIs, which often return unpredictable or variable responses. This makes it difficult to accurately evaluate the success of a user request. Moreover, third-party APIs may be costly, unstable, or subject to failure, making training on their outputs impractical. These issues are pronounced when handling user queries requiring multi-step solutions, where each tool call depends on previous results.

Our analysis focused on the limitations of three existing solutions: GPT4Tools, ToolAlpaca, and ToolBench. We proposed methods to bridge the gap between academic datasets and real-world applications. We developed realistic evaluation protocols and showed that practical models can be built even from entirely synthetic data and descriptions, as shown in our GPT4Tools experiments. Additionally, our *ToolReflection* approach proved effective in recovering from errors using API feedback, and our experiments show this compensates for insufficient tool documentation, a frequent challenge with third-party APIs.

However, our error analysis highlights that *ToolReflection* still cannot address certain error classes. In future work, we plan to explore methods to reduce uncorrectable errors and further enhance LLM self-correction capabilities.

Limitations

While our work demonstrates significant improvements in LLM tool usage through the *ToolReflection* method and enhanced evaluation datasets, several limitations should be acknowledged.

First, the focus on single-step correction. Our implementation primarily addresses errors by correcting the most recent tool invocation. This may be insufficient for multi-step tasks where errors propagate from earlier steps. Developing mechanisms for multi-step error diagnosis, potentially involving backtracking, represents an important direction for future research.

Second, computational overhead associated with the method. *ToolReflection* requires generating an error-correction dataset, adding computational cost. Furthermore, adapting the method to diverse API response formats across different benchmarks (GPT4Tools, ToolAlpaca, ToolBench) and real-world tools often requires significant manual effort. Future work could investigate more efficient or even unsupervised methods for generating correction data.

Third, reliance on quality error feedback. The effectiveness of *ToolReflection* is tied to the quality and availability of error feedback from APIs. It performs best with informative error messages. Future work could explore training models to infer errors even from implicit or subtle negative signals.

Fourth, limited comparison of feedback learning strategies. Our current approach relies exclusively on supervised fine-tuning using the generated error-correction pairs. The nature of this feedback, particularly the comparison between corrected and uncorrected tool invocations, naturally lends itself to preference learning paradigms such as Direct Preference Optimization (DPO) (Rafailov et al., 2023) or other methods inspired by Reinforcement Learning from Human Feedback (RLHF) (Stiennon et al., 2020). We acknowledge that our evaluation does not currently include these alternatives, which presents an avenue for future investigation.

Fifth, benchmark realism. Our enhancements to the evaluation sets make them more realistic, but they still necessarily simplify the complexities and unpredictability of real-world tools. Future work should continue to focus on developing more realistic and comprehensive benchmarks that better capture these nuances.

References

Shengnan An, Zexiong Ma, Zeqi Lin, Nanning Zheng, Jian-Guang Lou, and Weizhu Chen. 2024. Learning from mistakes makes llm better reasoner. *Preprint*, arXiv:2310.20689.

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

Yixin Liu and Pengfei Liu. 2021. Simcls: A simple framework for contrastive learning of abstractive summarization. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 2: Short Papers)*, pages 1065–1072.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, and 1 others. 2023. Self-refine: Iterative refinement with self-feedback. *Advances in Neural Information Processing Systems*, 36:46534–46594.

Shervin Minaee, Tomas Mikolov, Narjes Nikzad, Meysam Chenaghlu, Richard Socher, Xavier Amatriain, and Jianfeng Gao. 2024. Large language models: A survey. *Preprint*, arXiv:2402.06196.

OpenAI. 2022. Chatgpt: Optimizing language models for dialogue. `https://openai.com/blog/chatgpt/`.

Shishir G Patil, Tianjun Zhang, Xin Wang, and Joseph E Gonzalez. 2024. Gorilla: Large language model connected with massive apis. *Advances in Neural Information Processing Systems*, 37:126544–126565.

Shuofei Qiao, Honghao Gui, Chengfei Lv, Qianghuai Jia, Huajun Chen, and Ningyu Zhang. 2024. Making language models better tool learners with execution feedback. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 3550–3568.

Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, Sihan Zhao, Runchu Tian, Ruobing Xie, Jie Zhou, Mark Gerstein, Dahai Li, Zhiyuan Liu, and Maosong Sun. 2023. ToolLLM: Facilitating large language models to master 16000+ real-world APIs. *Preprint*, arXiv:2307.16789.

Rafael Rafailov, Archit Sharma, Eric Mitchell, Christopher D Manning, Stefano Ermon, and Chelsea Finn. 2023. Direct preference optimization: Your language model is secretly a reward model. *Advances in Neural Information Processing Systems*, 36:53728–53741.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: Language models can teach themselves to use tools. *Advances in Neural Information Processing Systems*, 36:68539–68551.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik R Narasimhan, and Shunyu Yao. 2023. Reflexion: Language agents with verbal reinforcement learning. In *Thirty-seventh Conference on Neural Information Processing Systems*.

Nisan Stiennon, Long Ouyang, Jeffrey Wu, Daniel Ziegler, Ryan Lowe, Chelsea Voss, Alec Radford, Dario Amodei, and Paul F Christiano. 2020. Learning to summarize with human feedback. *Advances in neural information processing systems*, 33:3008–3021.

Qiaoyu Tang, Ziliang Deng, Hongyu Lin, Xianpei Han, Qiao Liang, and Le Sun. 2023. Toolalpaca: Generalized tool learning for language models with 3000 simulated cases. *Preprint*, arXiv:2306.05301.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, and 49 others. 2023. Llama 2: Open foundation and fine-tuned chat models. *Preprint*, arXiv:2307.09288.

Xingyao Wang, Hao Peng, Reyhaneh Jabbarvand, and Heng Ji. 2024a. Leti: Learning to generate from textual interactions. In *Findings of the Association for Computational Linguistics: NAACL 2024*, pages 223–239.

Xingyao Wang, Zihan Wang, Jiateng Liu, Yangyi Chen, Lifan Yuan, Hao Peng, and Heng Ji. 2024b. Mint: Evaluating llms in multi-turn interaction with tools and language feedback. *Preprint*, arXiv:2309.10691.

Rui Yang, Lin Song, Yanwei Li, Sijie Zhao, Yixiao Ge, Xiu Li, and Ying Shan. 2023. Gpt4tools: Teaching large language model to use tools via self-instruction. *Advances in Neural Information Processing Systems*, 36:71995–72007.

Seonghyeon Ye, Yongrae Jo, Doyoung Kim, Sungdong Kim, Hyeonbin Hwang, and Minjoon Seo. 2023. Selfee: Iterative self-revising llm empowered by self-feedback generation. Blog post.

Siyu Yuan, Kaitao Song, Jiangjie Chen, Xu Tan, Yongliang Shen, Ren Kan, Dongsheng Li, and Deqing Yang. 2024. Easytool: Enhancing llm-based agents with concise tool instruction. *Preprint*, arXiv:2401.06201.

Kechi Zhang, Zhuo Li, Jia Li, Ge Li, and Zhi Jin. 2023. Self-edit: Fault-aware code editor for code generation. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 769–787.

Kexun Zhang, Hongqiao Chen, Lei Li, and William Wang. 2024. Don't fine-tune, decode: Syntax error-free tool use via constrained decoding. *Preprint*, arXiv:2310.07075.

Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, Yifan Du, Chen Yang, Yushuo Chen, Zhipeng Chen, Jinhao Jiang, Ruiyang Ren, Yifan Li, Xinyu Tang, Zikang Liu, and 3 others. 2023. A survey of large language models. *Preprint*, arXiv:2303.18223.

Yuchen Zhuang, Xiang Chen, Tong Yu, Saayan Mitra, Victor Bursztyn, Ryan A. Rossi, Somdeb Sarkhel, and Chao Zhang. 2024. Toolchain*: Efficient action space navigation in large language models with a* search. In *The Twelfth International Conference on Learning Representations*.

A Hyperparameters

A.1 GPT4Tools

We use the same hyperparameters as the original work, with Vicuna-13B as the base model. We tune

LoRa weights for query, key, value, and output projection layers, using a LoRa rank of 16, batch size 64, and AdamW optimizer with a learning rate of 3×10^{-4}. We reserve 2048 training examples for validation, fine-tune for 5 epochs, and select the best checkpoint on validation.

A.2 ToolAlpaca

We reproduce ToolAlpaca's setup (Tang et al., 2023), fine-tuning Vicuna-7B-v1.1 for three epochs on 8 Nvidia V100 GPUs with a total batch size of 128 and a 2e-5 learning rate using the AdamW optimizer.

A.3 ToolBench and Multistep-100

We fine-tune LLaMA 7B on instruction-solution paths, using the original hyperparameters except for context length. The model is trained for 2 epochs with a 5e-5 learning rate using the AdamW optimizer on 8 Nvidia V100 GPUs (batch size 64). We limit the context length to 4096 and use positional interpolation to extend the context length of LLaMA 7B.

B Error Analysis

B.1 GPT4Tools

Table 7 provides a detailed breakdown of specific types of errors corrected by *ToolReflection* on the GPT4Tools-OOD-Hard evaluation set. After analyzing model outputs on the GPT4Tools-OOD-Hard dataset, we decided to divide type 2 errors (with feedback) into the following categories: (1) format errors (the model is unable to provide a parsable API call; it either provides incorrect Python code or hallucinates the tool name); (2) errors in parameter format (the model either provides nonexistent parameters, does not provide required parameters, or provides parameters of incorrect types, e.g., mixing strings and integers); (3) errors in parameter values (the model provides syntactically correct but semantically wrong parameters; in a real example, when asked to find the coordinates for Rue de l'Église in Paris, the model called `geocoderapi_geocode(location="Rue de l'Église")` and received the coordinates of Rue de l'Église in Boësses, a different city); (4) meaningless output errors (sometimes the LLM generates meaningless output such as cyclically repeating the same words etc.).

Table 7 shows that the least corrected errors are errors in parameter values and meaningless output. The correction rate for meaningless output is a flat 0%: once the LLM begins generating meaningless text, it cannot return to normal conversation. Errors in parameter values are often hard to correct, because in such cases (see Fig. 2) the error feedback is often useless. Parameter format errors should be easier to fix with feedback, so the result of 57% represents significant room for improvement in this category.

B.2 ToolAlpaca

In the *ToolAlpaca* dataset, unlike the GPT4Tools setup, it is harder to authomatically calculate the number of all errors and corrections due to multiple tool calls needed to give an answer and the nature of HTTP responses where the error format may be different from tool to tool. To understand typical errors for this benchmark, we analyse all cases where tools return any code not equal to 200 (e.g. 400, 404). Although the original evaluation of ToolAlpaca is entirely automated and depends solely on external LLM judgments, this manual analysis may provide additional insight.

Table 8 presents error analysis on simulated and real evaluation sets for each API call in the format of "First attempt / Last attempt". Here "Meaningless" means errors caused by nonsense LLM output, like hallucinations or cyclic generations; "HTTP feedback" denotes normal response from the API or simulator that contains a more or less informative message. The middle column shows cases where the response is incorrect: for real-world tools, these are cases of internal API errors, for simulated tools, cases of incorrect behaviour of the LLM checker. A typical case for this dataset is when the LLM incorrectly applies the requirements from API documentation. The most frequent case of such mistake is when there are several examples of possible values provided in the documentation, but LLM considers them as the *only possible* examples (e.g., assumes that "the color could be cyan, yellow, etc" means that acceptable values for the color are only [*cyan*, *yellow*]). It is clear from the table that even uninformative feedback from the tool, such as an error message, may help the model to find another solution and succeed. In general, the *ToolReflection*-trained model is often able to recover after erroneous API feedback, even in case of complicated APIs with several subfunctions with complex parameter sets.

To illustrate the importance of the tool feedback, consider the set of tasks in ToolAlpaca

Last Tool Call	GPT4Tools + GPT4FakeTools	ToolReflection
Correct call, no error received	40%	63%
Incorrect call, error received	51%	25%
Incorrect call, no error received	9%	12%

Table 6: GPT4Tools-OOD-Hard error analysis.

Error Type	Format	Param. Format	Param. Values	Meaningless Output
% corrected	93%	57%	46%	0%

Table 7: Analysis of corrected errors on GPT4Tools-OOD-Hard.

Setup	Examples	Meaningless	Check err. Internal err.	HTTP feedback
Sim	100	0/2	0/5	4/7
Real	114	0/3	2/2	6/10

Table 8: ToolAlpaca error analysis.

Real test set based on the `chucknorris.io` API, developed to provide Chuck Norris jokes for a given topic. The model is given a choice of 4 functions: "*jokes_random_get*, *jokes_random_category_get*, *jokes_categories_get*, *jokes_random_get*, *jokes_search_get*". Task 2 of this subset has the following query: *"I'm writing a blog post about Chuck Norris and his impact on pop culture. I need a joke related to 'music'. Can you find one for me?"*. The expected tool call provided as the ground truth in the dataset is `jokes_random_category_get{"category": "music"}`. But in practice, this call does not work. The model starts from the exactly this tool call, obtaining `{status:404, error:Not Found, path:/jokes/random/music}`. Getting this response, our *ToolReflection*-trained model replaces the call by `jokes_search_get{"query": "music"}`, which returns the joke *"Someone asked Chuck Norris what kind of music he listened to. He answered: I don't listen to music, music listens to me."*

B.3 ToolBench and Multistep-100

In this section, we examine errors made on the Multistep-100 evaluation set. Similar to the *ToolAlpaca* dataset, it is harder to examine errors automatically for the ToolBench format, so we do it manually. Due to the fact that the ToolBench evaluation set contains 1100 queries and tool invocation chains, and since only Multistep-100 has a set of ground truth answers, we focus on examining errors on Multistep-100 only. Note that in the ToolBench setup, we do not have a script-based parameter checker, and all error feedback comes from tool HTTP responses.

Table 9 indicates that 45% of cases failed due to errors that did not get any feedback (we refer to these calls as type 3 calls and do not aim to correct them with our method). These errors include, for instance, successfully calling a tool with incorrect parameters, skipping the calling of the first tool and calling the second one with hallucinated parameters, or not calling one of the tools at all and finishing the conversation. By analyzing errors further, we noticed that the model was able to correct errors in 7% of cases; 2% of them led to a tool call of the third type and 5% led to a successful correct answer.

Surprisingly, the model is not able to correct a significant amount of the 28% of errors with feedback. We noticed that in all cases, the model correctly identified an error, which could be noticed by an internal thought such as "Tool call returned an error. To correct it, I need to do the following," which has appeared between errors during the fine-tuning stage. In some cases, after receiving an error, the model simply terminated the generation and called the "Finish" tool. In most cases, however, the model tried to generate another tool call and made up the tool response by itself in the end, making the parser unable to extract the tool call. Such cases highlight that the fine-tuning procedure in the case of ToolBench should be explored in detail in the future.

C Changed GPT4Tools Prompts

Prompt for naive generation and fine-tuning with tool usage instructions

Case type	Correct	Fatal	Corrected	Not corrected	No feedback
% cases	12%	10%	5%	28%	45%

Table 9: Multistep-100 error analysis.

````
GPT4Tools can handle various tasks.
It generates human-like text and uses
    tools to follow user instructions.
To call API tool it writes python code
    according to the API tool's
    description.

TOOLS:
------

GPT4Tools has access to the following
    API tools:

{tools}

To use a tool, please use the following
    format:

```
Thought: Do I need to use a tool? Yes
Thought: Which tool should I use? the
    action to take should be one of the
    API tools
AI: python code according to the API
    tool's description, including python
     function with the exact same name
    as the action name and its
    parameters
Output: the result of the action
```

When you have a response to say to the
    human, or if you do not need to use
    a tool, you MUST use the format:

```
Thought: Do I need to use a tool? No
Output: [your response here]
```

Follow the API tool description rules.
Do not make up function names and
    parameters of those functions.

Previous conversation:

{previous_input}

Input: {input}
Begin! Let's think step by step.

{previous_conversation}
````

Prompt for generation and fine-tuning with *ToolReflection*

````
GPT4Tools can handle various tasks.
It generates human-like text and uses
    tools to follow user instructions.
To call API tool it writes python code
    according to the API tool's
    description.

TOOLS:
------

GPT4Tools has access to the following
    API tools:

{tools}

To use a tool, please use the following
    format:

```
Thought: Do I need to use a tool? Yes
Thought: Which tool should I use? the
    action to take, should be one of the
     API tools
AI: python code according to the API
    tool's description, including python
     function with the exact same name
    as the action name and it's
    parameters
Observation: the result of the action
```

If the tool returned an error and this
    error is the mistake of GPT4Tools,
    use the following format:

```
Thought: Is the python code correct? No
Thought: Do I need to rewrite the code?
    Yes
Thought: Do I need to use a tool? Yes
Thought: Which tool should I use? the
    action to take
AI: correct python code according to the
     API tool's description
```

When you have a response to say to the
    Human, or if you do not need to use
    a tool, you MUST use the format:

```
Thought: Do I need to use a tool? No
Output: [your response here]
```

Follow the API tool description rules.
Do not make up function names and
    parameters of those functions.

Previous conversation:

{previous_input}

Input: {input}
Begin! Let\'s think step by step.

{previous_conversation}
````

D GPT4FakeTools generation prompts

Example prompts for the tool description and queries with examples of tool usage generation.

````
Try to come up with new tools and their
    descriptions. Each tool description
    should
follow the format  Tool  Name: usage
    scenario. Parameter descriptions
1. Speech Recognition: useful when you
    want to recognize speech from a
    microphone
or audio file. The input to this tool
    should be an audio file path or a
    microphone
input.
2. Object Tracking in Image: useful when
     you want to track the position of
    an object
in an image across multiple frames. The
    input to this tool should be a
    string,
representing the path of the image file
    sequence.
3. Audio Speed Changer: useful when you
    want to change the speed of an audio
     file.
The input to this tool should be a
    string, representing the path of the
     audio file,
and another string, representing the new
     speed.
4. Video Editing: useful when you want
    to edit a video by trimming,
    cropping, adding
music, or enhancing the video quality.
    The input to this tool should be a
    string,
representing the path of the video file.
5. Background Removal: useful when you
    want to remove the background from
    an image
and create a transparent background. The
     input to this tool should be a
    string,
representing the path of the image file.
6.

  ― ― ―

Please generate instruction for each of
    the given tools.
Each tool is defined as "<Tool Name>: <
    usage scenario>"
1. Speech Recognition: useful when you
    want to recognize speech from a
    microphone
or audio file. The input to this tool
    should be an audio file path or a
    microphone
input.
2. Object Tracking in Image: useful when
     you want to track the position of
    an object
in an image across multiple frames. The
    input to this tool should be a
    string,
representing the path of the image file
    sequence.
3. Audio Speed Changer: useful when you
    want to change the speed of an audio
     file.
The input to this tool should be a
    string, representing the path of the
     audio file,
and another string, representing the new
     speed.
4. Video Editing: useful when you want
    to edit a video by trimming,
    cropping, adding
music, or enhancing the video quality.
    The input to this tool should be a
    string,
representing the path of the video file.
5. Background Removal: useful when you
    want to remove the background from
    an image
and create a transparent background. The
     input to this tool should be a
    string,
representing the path of the image file.

Here is an example for the tool "Video
    Splitter" -- "Divide the video
    located at
/path/to/video.mp4 into 10-second
    intervals.",
[Video Splitter, "/path/to/video.mp4",
    "10 seconds"]"
Try not repeating the words from tool
    description, where possible.
Provide diverse instructions.
````

E Dummy functions generation prompt

````
You are provided with the list of
    functions and the list of calls.
    Your task is to write python
    functions that will be executed by
    provided calls.
These functions should have the same
    signature as the calls. The
    functions should not do anything,
    but check if the parameters have
    correct types. If the parameters are
     wrong, throw an error.
Write code of the functions only.
Functions should be separated by
'######################################'

Do not wrap python code into ```
    brackets
Function names and descriptions: {
    descriptions}
Calls: {api_calls}
````

F GPT4Tools-OOD and GPT4Tools-OOD-Hard example tool descriptions

Examples of tool descriptions from GPT4Tools-OOD-Hard

````
API Name: pintapi_convert_units
API Parameter: The input to this tool
    should be the
````

```
from_value, from_unit, and to_unit.
API Description: Convert from one unit
    to another.

API Name: geocoderapi_reverse_geocode
API Parameter: The input to this tool
    should be two floats
representing the latitude and longitude
    coordinates.
API Description: Reverse geocode a given
     latitude and longitude
to obtain address information.
```

Examples of tool descriptions from GPT4Tools-OOD

````
API Name: pintapi_convert_units
API Parameter: The input to this tool
    should be the
from_value, from_unit, and to_unit.
API Description: Convert from one unit
    to another.
Usage Example: Hey, how many kilometers
    are there in 25 miles?
```python
pintapi_convert_units(from_value=25,
    from_unit=\"miles\", to_unit=\"
    kilometers\")```

API Name: geocoderapi_reverse_geocode
API Parameter: The input to this tool
    should be two floats representing
    the
latitude and longitude coordinates.
API Description: Reverse geocode a given
latitude and longitude to obtain address
     information.
Usage Example: What is the address for
    the location at coordinates 39.5501,
     -105.7821?
```python
geocoderapi_reverse_geocode(latitude
    =39.5501, longitude=-105.7821)
```
````

G Prompt for ToolBench API error detection

```
You will be given api calls from some
    model and api outputs from rapidapi
    apis.
Sometimes api return errors.
It is hard to parse them using rule-
    based methods.
An api may respond with an empty output
    (it could be empty list, or an empty
     dict),
contain a field that states it is an
    error or etc.
You will be given 5 examples. You should
     respond only "Yes" if the response
    does not
contain errors from an api and "No" if
    not.
Please make sure that api returned what
    the api call requested.
If not, return "No"
Examples:
API Call:
Thought: To fulfill the user's request
    for gathering user information and
    recent
media posts from Instagram for a
    specific username 'nike', I
    previously called the
'userinfo_for_instagram_cheapest'
    function with the argument
{"username": "nike"}. By doing so, I
    efficiently retrieved the desired
    user data,
including their biography, bio links,
    biographical information, and other
relevant details.  Now, to proceed with
    accessing the user's recent  media
    posts,
I will call the '
    user_medias_for_instagram_cheapest'
    function with the argument
{"user_id": "113294420064920"}. This
    will allow me to retrieve the most
    recent
media posts of the user 'nike' and
    provide the user with the requested
data effectively.
Action:
    user_medias_for_instagram_cheapest
Action Input: {
  "user_id": "113294420064920"
}
API Response:
{"error": "", "response": "{'items': [],
     'num_results': 0, 'status': 'ok'}"}
Result:
No
###
API Call:
Thought: The user requested the list of
    languages supported by the
    LanguageTool
API for grammar and writing assistance.
    To fulfill this request, I need to
    call
the "list_of_languages_for_languagetool"
     function without any arguments.
By doing so, I expect to receive the
    complete list of supported languages
    . This
information will be crucial for
    implementing multilingual support in
     the project,
ensuring accurate grammar and writing
    assistance for different languages.
Action:
    list_of_languages_for_languagetool
Action Input: {}
API Response:
{"error": "", "response": {'name': '
    Arabic', 'code': 'ar', 'longCode': '
    ar'},
{'name': 'Asturian', 'code': 'ast', '
    longCode': 'ast-ES'},
{'name': 'Belarusian', 'code': 'be', '
    longCode': 'be-BY'},
{'name': 'Breton', 'code': 'br', '
    longCode': 'br-FR'},
{'name': 'Catalan', 'code': 'ca', '
    longCode': 'ca-ES'},
{'name': 'Catalan (Valencian)', 'code':
    'ca', 'longCode': 'ca-ES-valencia'},
```
```

```
{'name': 'Catalan (Balearic)', 'code': '
    ca', 'longCode': 'ca-ES-balear...
Result:
Yes
###
API Call:
{api_call}
API Response:
{api_response}
Result:
```

H Multistep-100 example queries and ground-truth answers

Here are examples of query templates

```
"I'm looking for details on the [
    watch_brandName] [watch_family]
    model [watch_model].
Can you provide me with information on
    the brand, release date, features,
case material, dial color, movement type
    , and market price in euros?"

"Could you please provide me with
    information on the [watch_brandName]
     [watch_family]
model [watch_model]?
Specifically, I'm interested in knowing
    the brand, release date, features,
case material, dial color, movement type
    , and market price in euros."

"I'm planning travel from [city&1] state
     [state&1] to [city&2] city, which
    is located in
[state&2], USA, could you provide me the
     distance
between this two cities in miles and
    then convert it to [metric_length]?"

"I'm looking to travel from [city&1], [
    state&1] to [city&2], [state&2] and
    I need
to know the distance between them in
    miles.
Could you also convert it to [
    metric_length] for me?"
```

Here is an example of ground-truth multistep tool invocation chain with a query


```
{"user": "Can you provide me with the
    power reserve information for
the A. Lange & S\u00f6hne Lange 1
    101.039 watch?
I need to convert it from hours to
    seconds.",
"first_tool_name": "
    get_watches_by_brand_family_model_
    for_watch_database",
"first_tool_params": {
"watch_brandName": "A. Lange & S\
    u00f6hne", "watch_family": "Lange
    1",
"watch_model": "101.039"},
"first_tool_response": "[{
\"id\":309857,\
"title\":\"A. Lange \\u0026 S\\u00f6hne
    101.039\",
\"watchName\":\"Lange 1 White Gold \\/
    Silver\",
\"marketPriceEuro\":null,\"brand\":\"A.
    Lange \\u0026 S\\u00f6hne\",
\"family\":\"Lange 1\",
\"model\":\"101.039\"...
"second_tool_params": "{
\"value\": \"72\",
\"input_unit\": \"h\",
\"output_unit\": \"s\"
}"...
"second_tool_name": "
    convert_from_one_unit_of_measure
_to_another_for_measurement_units
_converter",
"second_tool_params": "{
\"error\":\"\",
\"response\":\"{'input': {'value': '72',
     'unit': 'h'},
'output': {'value': 259200, 'unit': 's
    '}}\"}"
```