PLAY2PROMPT: Zero-shot Tool Instruction Optimization for LLM Agents via Tool Play

Wei Fang[†], **Yang Zhang**[‡], **Kaizhi Qian**[‡], **James Glass**[†], **Yada Zhu**[‡]
[†]Massachusetts Institute of Technology, Cambridge MA, USA
[‡]MIT-IBM Watson AI Lab, Cambridge MA, USA
{weifang,glass}@mit.edu, {yang.zhang2,kqian}@ibm.com, yzhu@us.ibm.com

Abstract

Large language models (LLMs) are increasingly integrated with specialized external tools, yet many tasks demand zero-shot tool usage with minimal or noisy documentation. Existing solutions rely on manual rewriting or labeled data for validation, making them inapplicable in true zero-shot settings. To address these challenges, we propose PLAY2PROMPT, an automated framework that systematically "plays" with each tool to explore its input-output behaviors. Through this iterative trial-and-error process, PLAY2PROMPT refines tool documentation and generates usage examples without any labeled data. These examples not only guide LLM inference but also serve as validation to further enhance tool utilization. Extensive experiments on real-world tasks demonstrate that PLAY2PROMPT significantly improves zero-shot tool performance across both open and closed models, offering a scalable and effective solution for domain-specific tool integration[1].

1 Introduction

Recently, there has been growing research interest in agentic large language model (LLM) frameworks, where, rather than having LLMs answer requests and queries from their own knowledge, LLMs can call a set of external tools with specialized capabilities. This allows LLMs to address more complex tasks and produce responses more accurately (Mialon et al., 2023; Qin et al., 2024a). One key challenge in developing agentic LLM frameworks is how to *dynamically learn to use new, user-defined tools*, which is crucial because having a fixed, general-purpose tool set is often insufficient for real-world scenarios requiring domain-specific functionalities.

The existing mainstream paradigm for dynamically incorporating new tools is by supplementing user-defined tools at inference time in a zero- or few-shot manner via prompting (Lu et al., 2023; Shen et al., 2023), leveraging zero-shot tool-calling capabilities of current LLMs that have been tuned with tool-use instructions. The success of this paradigm depends on providing sufficient information about the new tool in the prompt. Specifically, existing approaches generally rely on two types of information: ❶ *Comprehensive tool documentation* detailing the tool's functionalities and input/output formats, and ❷ *In-context demonstrations* that include example queries and corresponding tool calls (Hsieh et al., 2023; Patil et al., 2023). Inadequate documentation can lead to failures in tool usage, such as syntax errors in both zero-shot and fine-tuned models (Zhang et al., 2023a), hallucinations due to incomplete or incorrect tool documentation (Hsieh et al., 2023), and diminished performance resulting from inadequate demonstrations (Xu et al., 2023).

However, in many practical scenarios, it is often not realistic to rely on users, who are often non-experts in AI, to provide adequate documentation for their tools, nor to craft tool-use examples. When users do provide documentation, it may lack crucial details needed for LLMs to call the tools correctly. While automatic prompt optimization techniques (Wang et al., 2024) could enhance tool documentation, they still require sufficient tool-use examples, which are unavailable in true zero-shot settings. In short, without tool-use examples, neither polished documentation nor in-context demonstrations can be supplied, leading to significant performance degradation in integrating new tools.

To address these challenges in zero-shot tool utilization, we introduce PLAY2PROMPT, an automated framework that generates both high-quality tool documentation and tool-use demonstrations, as illustrated in figure 1. Unlike prior works, PLAY2PROMPT does not rely on any external labeled examples. Instead, it systematically interacts

[1] Source code is available at https://github.com/wfangtw/play2prompt.

Findings of the Association for Computational Linguistics: ACL 2025, pages 26274–26290
July 27 - August 1, 2025

with the new tools—mimicking human trial-and-error—and observes both successful and failed attempts to gather evidence about each tool's correct usage. Using insights from this "tool-play", PLAY2PROMPT creates example demonstrations and refines the tool documentation that better guide LLMs in subsequent inference.

PLAY2PROMPT consists of two steps. In Step 1, a set of tool-use examples are generated via a trial-and-error process, where an LLM agent iteratively call the target tool with different invocation parameters until correct invocation is found. Then, for each correct tool invocation instance, a query is generated such that it can be answered by the tool invocation, forming a question-answer pair as a tool-use example. In step 2, the tool documentation are refined, using the generated tool-use examples as a validation set. In both steps, We employ self-reflection (Madaan et al., 2023; Pryzant et al., 2023; Shinn et al., 2023) to generate error feedback, thereby directing the search algorithm towards progressively improved outputs. Because PLAY2PROMPT operates entirely in a zero-shot manner and is inherently task-agnostic, it offers a practical and scalable solution for enhancing LLM tool utilization without additional labeled data or human intervention.

We evaluate PLAY2PROMPT with benchmark on real-world scenarios. On the Berkeley Function-Calling Leaderboard (Yan et al., 2024) and the StableToolBench benchmark (Guo et al., 2024), our approach consistently surpasses baseline methods for both open (Dubey et al., 2024; Liu et al., 2025; Lin et al., 2025) and closed models (Achiam et al., 2023). Extensive experiments and analyses further underscore the robustness of our approach.

Our contributions can be summarized as follows:

- We propose PLAY2PROMPT, a novel automated framework that iteratively refines tool documentation and generates usage examples, enabling more effective zero-shot tool utilization without any labeled data.
- PLAY2PROMPT integrates a search-based trial-and-error process augmented with self-reflection, allowing LLMs to explore and "play" with tools to refine both tool documentation and demonstrations for enhanced performance.
- PLAY2PROMPT is entirely zero-shot, scalable, and task-agnostic, making it broadly applicable across diverse tools and domains, and practical for large-scale enhancement of LLM tool use without additional manual effort.

2 Methodology

2.1 Problem Formulation and Notation.

A typical agentic LLM framework contains two components: ❶ A task LLM, denoted as $\mathcal{M}_T$, and ❷ a set of tools, $\mathcal{F} = \{F_{1:K}\}$, where K represents the number of tools, and $1 : K$ represents a set of indices running from 1 to K. Given an input query x, rather than directly answering it based on its own knowledge, the task LLM $\mathcal{M}_T$ first selects a sequence of N tools, $(F_{k_{1:N}})$, generates the appropriate input parameters, I_{k_n}, to call each tool, *i.e.*, $F_{k_n}(I_{k_n})$, and finally produces the answer y based on the tool outputs.

We consider the setting where the tool list $\mathcal{F}$ is ad-hoc and dynamic. In order for the task LLM to learn to use the tools in $\mathcal{F}$ at inference time, we follow the conventional prompting-based pipeline (Lu et al., 2023; Hsieh et al., 2023), where the prompt, in addition to an instruction, contains the following tool-specific information:

- **Tool Documentation**, denoted as $\mathcal{D} = \{D_{1:K}\}$;
- **In-context examples of tool-use**, denoted as $\mathcal{E} = \{E_{1:M}\}$, where each example E_m contains an input query, x, the tool invocation details, $(F_{k_{1:N}}, I_{k_{1:N}})$, and the answer, y.

In many real-world scenarios where users supply their own tool list, it is unrealistic to require them to supply high-quality documentation or tool-use examples. To model these scenarios, we adopt a challenging zero-shot setting with the following constraints: ❶ The initial documentation $\mathcal{D}_0$ is sub-optimal and may lack important details, and ❷ No tool-use examples $\mathcal{E}$ are available. Given these constraints, our objective is to generate tool-use examples and refine the tool documentation to enhance the task LLM's tool-use performance.

2.2 PLAY2PROMPT Overview

The primary goal of PLAY2PROMPT is to utilize knowledge gained from tool interactions to optimize tool documentation and example demonstrations. PLAY2PROMPT consists of the following two steps. *First*, a tool-use example set, $\mathcal{E}$, is generated. *Second*, using $\mathcal{E}$ as the validation set, a refined tool documentation $\mathcal{D}$ is generated based on the initial one $\mathcal{D}_0$. After the two steps are accomplished, $\mathcal{D}$ and a subset of $\mathcal{E}$ will be fed as the prompt to the task LLM during inference.

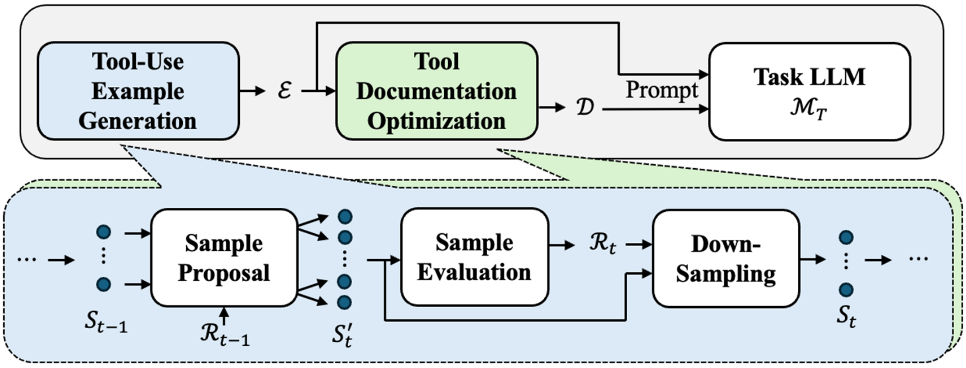

Figure 1: The PLAY2PROMPT framework: Beam search iteratively searches tool-use examples, incorporating tool play into the proposal process. After examples are generated, beam search is once again applied to optimize documentation by incorporating tool-use outputs and errors. Finally, optimized tool-use examples and documentation are used as prompts for $\mathcal{M}_T$ at inference.

In both steps, we adopt a tree search framework similar to Wang et al. (2024) to generate tool-use examples or tool documentation. Therefore, we will first briefly introduce the beam search framework (see figure 1) in Section 2.3, and then elaborate on the two steps in Sections 2.4 and 2.5, respectively.

2.3 Search Framework

The beam search framework is an iterative algorithm to generate high-quality samples, denoted $\mathcal{S} = \{S^{(i)}\}$, which correspond to $\mathcal{E}$ in step 1, and $\mathcal{D}$ in step 2. At the start of iteration t, the algorithm has access to $\mathcal{S}_{t-1} = \{S^{(i)}_{t-1}\}$, which are samples generated in the previous iteration, as well as their corresponding reward $\{R^{(i)}_{t-1}\}$, which depicts the quality of each sample. Then iteration t involves the following procedure to generate a set of improved samples:

- **Sample Proposal.** For each old sample, $S^{(i)}_{t-1}$, generate L new samples from the proposal distribution $p(S^{(i)}_t | S^{(i)}_{t-1}, R^{(i)}_{t-1})$, based on the reward of the old sample. This is accomplished by prompting a generator LLM, denoted as $\mathcal{G}$, to perform self-reflection on why the old sample is imperfect, how to improve the sample quality, and finally generate the improved samples. All the new samples form a new sample set, denoted as $\mathcal{S}'_t$, whose size is L times the size of $\mathcal{S}_{t-1}$.
- **Sample Evaluation.** For each new sample $S^{(i)}_t$ in $\mathcal{S}'_t$, compute its reward $R^{(i)}_t$.
- **Subsampling.** Trim $\mathcal{S}'_t$ down to the size of $\mathcal{S}_{t-1}$ by keeping the samples with the highest reward. Denote the trimmed sample set as $\mathcal{S}_t$.

The iterations terminate when the pre-set maximum number of iterations is reached. With the beam search framework, the algorithm design boils down to designing ❶ the sample proposal distribution, and ❷ the reward and feedback of each sample. In the following sections, we detail how these design choices are set in tool-use example generation and tool documentation optimization.

2.4 Tool-Use Example Generation

In this work, we only consider generating examples where only a single tool is used. We will show that (Section 3.2) the task LLM can still learn to solve queries that require multiple tools with the examples of single tool use. In this way, examples for different tools can be generated separately. Specifically, examples of using tool F_k take the form of $E = (x, F_k, I_k, y)$, which can be generated via the beam search framework in Section 2.3, with the design choices detailed below.

Sample Proposal. The sample proposal is performed by prompting an example generator LLM, denoted as $\mathcal{G}_E$. Since the tool is fixed to F_k, it only needs to propose new samples for the query x, the invocation parameters I_k, and the final answer y. However, the challenge lies in the limited information available about F_k—only an (incomplete) initial documentation $\mathcal{D}_0$ and no user-supplied examples—making it likely that the generated samples fail to invoke the tool correctly.

To address this, we generate samples in **reverse order**: first, we explore valid invocations I_k, observe the tool's output, and then construct a corresponding query x and answer y. This approach effectively *"plays with" the tool* to understand its behavior before defining its use cases.

Legitimate samples of I_k are generated in a rejection-sampling process, where $\mathcal{G}_E$ first generates a tentative sample invocation given $\mathcal{D}_0$, observes the tool outputs and error messages, performs self-reflection, and generates the next one. Note that this inner loop is nested in the outer loop of the beam search framework, so the rejection sampling is also conditional on the tool-use examples generated in the previous outer iteration $t-1$, along with their reward (see Section 2.3), except for in outer iteration 0. This inner loop terminates at a fixed number of steps, after which L legitimate invocations are selected.[2]

For each sampled legitimate invocation, we feed the invocation and the tool output to $\mathcal{G}_E$, which is then prompted to generate a query and an answer by performing N_E steps of self-reflecting refinement, which concludes the sample proposal procedure.

Reward Design. For each generated sample $E^{(i)} = (x^{(i)}, F_k, I_k^{(i)}, y^{(i)})$, the corresponding reward consists of two terms,

$$R^{(i)} = R_q^{(i)} + \lambda R_e^{(i)}. \tag{1}$$

$R_q^{(i)}$ evaluates the quality of the generated example, including clarity and coherence between the input query and tool use. This is evaluated by prompting $\mathcal{G}_E$ to output a score of 1-3 given the grading criteria. $R_e^{(i)}$ evaluates the performance of the task LLM $\mathcal{M}_T$ in answering the query in this example:

$$R_e^{(i)} = -\mathcal{P}\{\mathcal{M}_T(x^{(i)}; \mathcal{D}_0, \varnothing); y^{(i)}, F_k, I_k^{(i)}\}, \tag{2}$$

where $\mathcal{P}\{\hat{M}; y, F_k, I_k, \}$ denotes the task performance metric (the higher the better, *e.g.*, accuracy) of the model output $\hat{M}$ against the ground-truth answer y and tool invocation F_k, I_k; $\mathcal{M}_T(x^{(i)}; \mathcal{D}_0, \varnothing)$ denotes the output of the task LLM in answering the query $x^{(i)}$ given the initial documentation $\mathcal{D}_0$ and *no in-context examples* (because we don't have any yet at this stage).

The negative sign in Eq. 2 indicates that we encourage *difficult examples* – examples that the task LLM cannot get right, because difficult examples bring more surprise to the task LLM and thus are more effective in shaping the LLM's behavior.

2.5 Tool Documentation Optimization

The goal of tool documentation optimization is to generate improved tool documentation $\mathcal{D}$ based on the initial one $\mathcal{D}_0$, which is again achieved by the beam search framework, where the documentation sample with the highest reward will be chosen as the final tool documentation.

[2] If the number of legitimate invocations is smaller than L, we will limit the number of proposed samples accordingly.

Sample Proposal. The sample proposal process primarily follows the approach described in Section 2.3. We provide the generator LLM $\mathcal{G}_D$, which differs from $\mathcal{G}_E$ used in the previous step, with the current documentation $\mathcal{D}^{(i)}$ along with tool use errors from $\mathcal{M}_T$. By conditioning the proposal distribution on these errors, we inform $\mathcal{G}_D$ of the documentation's deficiencies or ambiguities, prompting more effective revisions.

Reward Design. The reward is computed on the tool-use example set $\mathcal{E}$ generated in the previous step, essentially treating $\mathcal{E}$ as the validation set for documentation optimization. For each tool documentation sample, $\mathcal{D}^{(i)}$, the corresponding reward is the tool use performance given $\mathcal{D}^{(i)}$ on a small batch in the validation set:

$$R^{(i)} = \mathbb{E}_{(x,F,I,y)\in\mathcal{E}}\left[\mathcal{P}\{\mathcal{M}_T(x; \mathcal{D}^{(i)}, \varnothing); y, F, I\}\right]. \tag{3}$$

By comparing Eqs. 2 and 3, it can be observed that tool-use example generation and tool documentation optimization have adversarial objectives, the former seeking to reduce the tool use performance (while maintaining quality and alignment), and the latter to improve it. This resembles the active learning strategy of choosing high-loss examples (Yoo and Kweon, 2019), which reveals that maximizing the performance on the most difficult examples leads to high learning efficiency.

3 Experiments

3.1 Experimental Setup

Dataset: Berkeley Function-Calling Leaderboard (BFCL). We evaluate on the Berkeley Function-Calling Leaderboard (Yan et al., 2024), a benchmark of real-world data that assesses LLMs' tool-use abilities. Its data includes a non-executable subset evaluated by abstract syntax trees and an executable subset assessed by running the functions. We use the executable subset (referred to as Executable), because actual *tool-play* is central to our approach, reflecting realistic scenarios in which most real-world APIs are callable. This subset has four categories of Python functions (single-tool, multi-tool, parallel tool-calling, and multi-tool parallel tool-calling) and plus one category of REST functions, yielding 310 test queries.

Dataset: StableToolBench. We also evaluate on StableToolBench (Guo et al., 2024), an updated version of ToolBench (Qin et al., 2024b), one of the most widely-used tool-use dataset, which addressed RapidAPIs' instability through a fallback system with caching and API simulation. Our experiments use all six of its testing subsets, including single-tool (I1), multi-tool (I2-same category and I3-different category) queries. Individual APIs, grouped under "tools", correspond to F_k, so I1 test queries usually require calls to multiple APIs within a tool and are not single-tool under our definition. The original dataset's categorization based on tool overlap with training data are thus less relevant for our strictly zero-shot setting. In total, these six subsets cover 790 queries, spanning 2479 unique APIs with an average of 5.3 APIs per query.

Inference and Evaluation. We adhere to official inference settings of each benchmark, where a set of tools is provided for each test query. For task LLM $\mathcal{M}_T$, we tested the 8B and 70B LLaMA models (Dubey et al., 2024), and GPT-3.5 and GPT-4o were used for GPT (Achiam et al., 2023). We also tested on BFCL two state-of-the-art LLMs trained specifically for tool-calling, namely ToolACE (Liu et al., 2025) and Hammer (Lin et al., 2025), with both models achieving high rankings on the BFCL leaderboard as of this publication. For BFCL, single-turn prompting with official prompts is used as the baseline inference method for LLaMA models, while direct function-calling mode is used for the GPT models, ToolACE, and Hammer. We do not provide tool-use examples or additional documentation in these baseline runs, complying with a zero-shot setting. For PLAY2PROMPT, optimized in-context examples and documentation are supplied as prompts and runs the baseline inference methods. It supports multi-tool queries by independently optimizing examples and documentation for each tool and then performing inference. We use the official evaluation metric of accuracy, with exact or structural matches depending on categories to determine correctness. We report the average across five categories, and, following the official setup, the weighted average ("Simple" categories weighted by 0.5).

StableToolBench employs a more complex chain-of-thought inference method, ReAct (Yao et al., 2023). We compare against EasyTool (Yuan et al., 2024), which uses direct prompting to optimize tool documentation and generate usage scenarios, but requires documentation and labeled in-context examples from additional tools and is thus not entirely zero-shot[3]. Additionally, we compare against DRAFT (Qu et al., 2025), a concurrent work, which optimizes tool documentation only in an iterative manner [4]. An evaluation LLM is used to judge whether a response adequately answers a user query, due the dataset's free-form output design. We follow the official pipeline, using official ReAct prompts and report solvable pass rate, which measures the percentage of queries deemed solvable by the evaluation LLM. Further details are provided in appendix B.

Optimization Details. PLAY2PROMPT first uses beam search to optimize tool-use examples. We set $\lambda = 1$, $N_E = 3$, limit depth to 3, and explore $L = 3$ beams per node. We use beam width $W = 10$ for BFCL and $W = 3$ for StableToolBench to generate and select the top W examples for each tool, which then are passed to the documentation optimization procedure. Beam search is again applied to select the best tool documentation. We employ `llama-3.1-8b-instruct` as both $\mathcal{G}_E$ and $\mathcal{G}_D$.

3.2 Results and Analyses

Results on BFCL and StableToolbench. In table 1, we summarize the results on BFCL with LLaMA and GPT task models, both with and without tool-use examples and tool documentation produced by PLAY2PROMPT. With PLAY2PROMPT, absolute gains of 4-7% are observed for the open models and GPT-3.5, while GPT-4o achieves 3% increase in average accuracy despite its already high baseline. Notably, PLAY2PROMPT addresses challenging categories such as REST for LLaMA-8B and Hammer-7B, and Multiple-Parallel for all models, yielding improvements of 10-17%. These gains suggest that optimized in-context examples and documentation can correct specific shortcomings in tool usage. Moreover, for the more difficult multi-tool queries in Multiple and Multiple-Parallel, we see larger gains compared to single-tool, even when we optimize each tool independently.

In table 2, we show the solvable pass rates on StableToolBench for LLaMA and GPT models with and without PLAY2PROMPT. Our approach sur-

[3]We use the optimized tool documentation and usage scenarios provided at `https://github.com/microsoft/JARVIS/tree/main/easytool`. The inference setting of Yuan et al. (2024) differs from ours as we follow the official inference prompts provided with StableToolBench.

[4]We use the optimized documentation very recently open-sourced at `https://github.com/quchangle1/DRAFT`. Note that Qu et al. (2025) tested only on the I2-Cat and I3-Inst subsets.

Base Model	Method	Simple-Python	Simple-REST	Multiple	Parallel	Multiple-Parallel	Weighted Avg	Avg
LLaMA-8B	Prompting	96.0	70.0	96.0	90.0	77.5	86.6	85.9
	+PLAY2PROMPT	97.0	87.1	96.0	92.0	92.5	**93.1**	**92.9**
LLaMA-70B	Prompting	100.0	91.4	96.0	84.0	82.5	89.6	90.8
	+PLAY2PROMPT	100.0	91.4	98.0	88.0	95.0	**94.2**	**94.5**
GPT-3.5	Function-calling	97.0	94.3	90.0	86.0	67.5	84.8	87.0
	+PLAY2PROMPT	98.0	95.7	94.0	90.0	85.0	**91.5**	**92.5**
GPT-4o	Function-calling	98.0	98.6	94.0	94.0	77.5	91.0	92.4
	+PLAY2PROMPT	99.0	95.7	98.0	92.0	90.0	**94.3**	**94.9**
ToolACE-8B	Function-calling	96.0	92.9	92.0	86.0	70.0	85.6	87.4
	+PLAY2PROMPT	95.0	90.0	94.0	90.0	82.5	**89.8**	**90.3**
Hammer-7B	Function-calling	96.0	72.9	88.0	86.0	70.0	82.1	82.6
	+PLAY2PROMPT	95.0	82.8	92.0	86.0	80.0	**86.7**	**87.2**

Table 1: Results on BFCL Executable. Accuracy scores are shown.

Base Model	Method	I1-Inst	I1-Cat	I1-Tool	I2-Inst	I2-Cat	I3-Inst	Avg
LLaMA-8B	ReAct	50.6	59.3	53.2	58.0	61.3	52.3	55.8
	ReAct+EasyTool	54.7	58.9	57.9	56.1	64.2	48.1	56.7
	ReAct+DRAFT	-	-	-	-	62.6	57.9	-
	ReAct+PLAY2PROMPT	56.6	65.7	60.9	62.5	63.5	63.8	**62.2**
LLaMA-70B	ReAct	58.4	68.3	61.1	63.1	63.2	64.3	63.1
	ReAct+EasyTool	59.0	70.3	63.3	67.6	70.6	63.0	65.6
	ReAct+DRAFT	-	-	-	-	68.5	62.3	-
	ReAct+PLAY2PROMPT	67.5	73.6	64.1	66.5	72.8	70.7	**69.2**
GPT-3.5	ReAct	56.0	64.6	67.4	56.0	63.4	57.4	60.7
	ReAct+EasyTool	57.3	61.4	69.5	61.5	68.3	60.7	63.1
	ReAct+DRAFT	-	-	-	-	68.3	56.1	-
	ReAct+PLAY2PROMPT	61.1	66.6	66.2	67.0	68.3	66.3	**65.9**
GPT-4o	ReAct	54.0	69.7	66.1	59.8	65.3	61.7	62.8
	ReAct+EasyTool	49.9	69.0	65.2	61.3	65.3	52.5	60.5
	ReAct+DRAFT	-	-	-	-	62.1	63.9	-
	ReAct+PLAY2PROMPT	60.7	68.7	71.9	58.8	63.7	65.6	**64.9**

Table 2: Results on StableToolBench. Solvable pass rates are shown.

passes all baselines with average gains of 5-7% across LLaMA models and GPT-3.5, and outperforms the specifically designed EasyTool, which is not fully zero-shot as it requires in-context demonstrations during optimization. PLAY2PROMPT is also consistent across subsets, avoiding large performance drops in multiple subsets when using EasyTool, highlighting the benefits of real-time "tool-play" and evaluation of candidate examples and documentation during optimization. Moreover, as GPT-4o exhibits more sensitivity to documentation and in-context demonstration changes on this dataset, EasyTool degrades GPT-4o on all subsets, whereas PLAY2PROMPT achieves a 2% improvement on average. Overall, PLAY2PROMPT essentially boosts performance of models up to the baseline performance of the larger models. Compared to DRAFT, which optimizes tool documentation only, PLAY2PROMPT outperforms for all four models on both I2-Cat and I3-Inst. This suggests that optimizing not only documentation, as DRAFT does, but also generating tool-use examples, can synergize and achieve even larger improvements. Additionally, our optimization model of LLaMA-8B is much smaller than the GPT-4o model used in DRAFT, further confirming the effectiveness of PLAY2PROMPT. It is noteworthy that most test samples for all 6 subsets in StableToolBench, as well as the Multiple and Multiple-Parallel subsets in BFCL, are multi-tool queries (see Section 3.1), underscoring the effectiveness of PLAY2PROMPT, which operates on single tools independently during optimization.

Robustness of PLAY2PROMPT. To evaluate the robustness of PLAY2PROMPT under incomplete documentation, we simulate noisy tool descriptions on the BFCL Executable dataset by randomly drop-

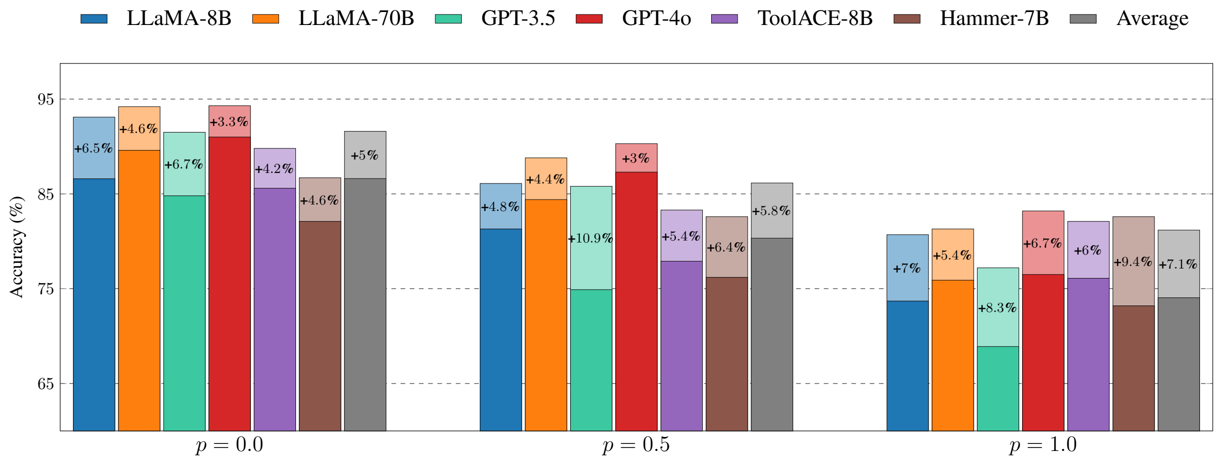

Figure 2: Average (weighted) accuracy improvements with PLAY2PROMPT on BFCL, for different parameter description dropout p.

Base Model	Method	BFCL	STB
LLaMA-8B	Baseline	85.9	55.8
	+P2P-Desc	89.9	57.9
	+P2P-Demo	90.8	59.5
	+P2P	**92.9**	**62.2**
LLaMA-70B	Baseline	90.8	63.1
	+P2P-Desc	93.6	64.4
	+P2P-Demo	92.7	67.8
	+P2P	**94.5**	**69.2**
GPT-3.5	Baseline	87.0	60.7
	+P2P-Desc	89.0	63.5
	+P2P-Demo	91.9	65.3
	+P2P	**92.5**	**65.9**

Table 3: Ablation on PLAY2PROMPT (P2P) using generated example demonstrations only (P2P-Demo) and generated descriptions only (P2P-Desc). Scores indicate average accuracy for BFCL and average solvable pass rate for StableToolBench (STB).

Base Model	Method	STB
LLaMA-8B	ReAct	55.8
	+PLAY2PROMPT-Easy	60.7
	+PLAY2PROMPT	**62.2**
LLaMA-70B	ReAct	63.1
	+PLAY2PROMPT-Easy	68.2
	+PLAY2PROMPT	**69.2**

Table 4: Ablation on demonstration difficulty for PLAY2PROMPT. We report average solvable pass rates.

ping parameter descriptions with increasing probabilities $p \in \{0.0, 0.5, 1.0\}$, while retaining overall tool descriptions and parameter names. This setup reflects varying degrees of real-world documentation sparsity. As expected, the degradation of documentation leads to reduced baseline accuracy across all models.

Despite these challenges, PLAY2PROMPT consistently improves tool-use performance across all models and noise levels. As shown in figure 2, PLAY2PROMPT achieves steady accuracy gains, with average improvements growing from 5.0% at $p = 0.0$ and reaching 7.1% at $p = 1.0$. Notably, the benefit of PLAY2PROMPT becomes even more pronounced as documentation quality deteriorates, demonstrating its ability to compensate for missing parameter details and its strong robustness in low-resource settings. Detailed per-model results can be found in appendix C.

Comparing Demonstrations and Documentation. We investigate how much the optimized examples contribute to PLAY2PROMPT's performance gains compared documentation. Using the same optimization procedure, we test two inference settings: one employs new in-context demonstrations alongside original documentation, and the other uses updated documentation without demonstrations. As shown in table 3, employing only optimized demonstrations yields larger improvements than solely relying on optimized documentation, and combining both consistently achieves the best results. These findings suggest that demonstrations and documentation complement each other in guiding LLMs' tool use, confirming the effectiveness of our two-stage approach.

Ablation on Example Difficulty. We further examine how challenging examples influence performance by removing the R_e term during example generation and only using R_q. Without R_e, the model no longer targets difficult examples, yielding what we denote as P2P-Easy. Under the same inference configuration, we substitute these exam-

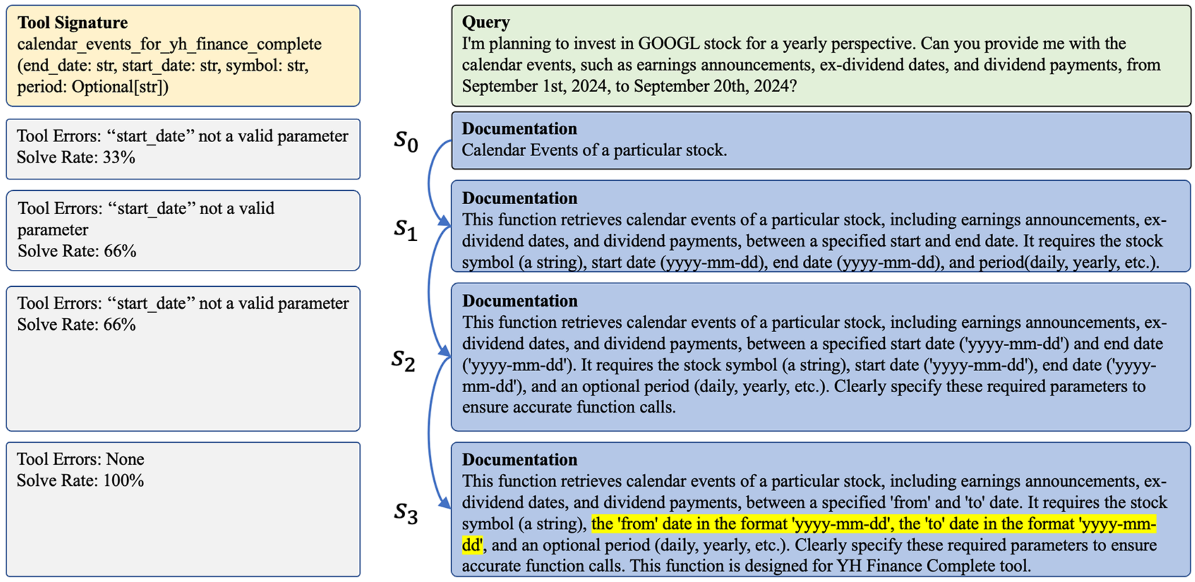

Figure 3: An example of PLAY2PROMPT facing incorrect documentation. We show the beam search trajectory with the highest solve rate on the validation set. At each step, a new documentation is explored based on error feedback.

Search strategy	**Pass Rate**	**# Proposals Explored**
ReAct	64.3	-
+P2P-MC (depth= 1)	65.2	3
+P2P-MC (depth= 5)	66.7	15
+P2P-MC (depth= 5) & N_E= 3	68.9	45
+P2P-Beam (depth= 5) & N_E= 3	**70.7**	135

Table 5: Ablation on search strategies, on the I3-Inst subset of StableToolBench. LLaMA-70B is used for task model $\mathcal{M}_T$. MC denotes Monte Carlo search.

ples in place of the original demonstrations and report results for LLaMA on StableToolBench in table 4. Ignoring difficult examples reduces final performance by an average of 1-2% across all subsets, confirming that generating in-context examples with higher difficulty enhance tool usage.

Ablation on Search Strategy. We investigate alternative strategies for exploring the example-documentation space by comparing beam search in PLAY2PROMPT to Monte Carlo search (MC) at various depths, sharing the same sample proposal scheme but with different number of proposals and sub-sampling approach. We also study the role of N_E self-reflection steps during example generation. Results are shown in table 5, focusing on I3-Inst, which combines multiple tools from different categories and is thus more challenging. The results show that MC underperforms beam search due to its limited exploration and is more subject to randomness, while deepening the search and adding self-reflection actions significantly improves performance, highlighting the importance of systematic search in optimizing examples and documentation. Given the tradeoff between performance and efficiency, a suitable search method can be chosen when using PLAY2PROMPT based on the user's computational budget.

Qualitative Analysis. We give an example in figure 3 to show how PLAY2PROMPT leverages tool play errors to refine tool documentation. The documentation is outdated, specifying `start_date` and `end_date` instead of `from` and `to`, and wrongly labeling them as required. Initially, PLAY2PROMPT is confused by contradictory information but gradually incorporates more precise details, solves queries without needing those parameters, and eventually identifies the correct names, leading to superior performance. An additional example of typical improvements appears in appendix D.

Human Evaluation of Generation Quality. Since our primary objective is to improve LLMs' tool-use capabilities, our main evaluation focuses on tool-use accuracy, which directly measures the effectiveness of our approach. While the quality of documentation and examples may not directly reflect tool-use improvements, they can provide complementary insights—particularly in enhancing clarity for human users and serving as secondary evidence of optimization quality. To assess these aspects, we conducted human evaluations on 50 randomly selected documentation outputs and 30 tool-use examples using six annotators. For documentation, we follow Qu et al. (2025) and assess

Dataset	Completeness			Conciseness			Accuracy		
STB	P2P	Raw	Equal	P2P	Raw	Equal	P2P	Raw	Equal
	85%	2%	13%	35%	**47%**	18%	**43%**	19%	37%
BFCL	P2P	Raw	Equal	P2P	Raw	Equal	P2P	Raw	Equal
	79%	5%	17%	15%	**70%**	15%	18%	19%	**63%**

Table 6: Human evaluation on tool documentation quality, comparing completeness, conciseness, and accuracy of PLAY2PROMPT versus raw (initial) documentation.

Examples	Naturalness	Alignment
BFCL Test Set	4.39	4.68
PLAY2PROMPT	4.17	4.52

Table 7: Human evaluation on tool usage example (question-function call) quality, rated on a scale of 1 (low) to 5 (high).

completeness, conciseness, and accuracy relative to the original dataset-provided descriptions. For tool-use examples, we adopt the evaluation criteria from Shen et al. (2024), scoring the naturalness of the query and its alignment with the function call[5], each on a scale of 1 to 5. As shown in table 6, our optimized documentation achieves significantly higher completeness and comparable or better accuracy, though with reduced conciseness due to increased verbosity. As shown in table 7, the zero-shot-generated examples are rated similarly to manually curated ones in terms of both naturalness and alignment, demonstrating their practical utility.

4 Related Work

LLMs for Tool Use. Recent years have seen notable advances in employing large language models (LLMs) as agents to master tool use for solving complex tasks (Mialon et al., 2023; Qin et al., 2024a), thereby enhancing LLMs' capabilities in multi-modal understanding (Gupta and Kembhavi, 2023; Surís et al., 2023; Wu et al., 2023), programming tools (Gao et al., 2023; Paranjape et al., 2023; Team et al., 2023; Zhang et al., 2023b; Cai et al., 2024), and other domain-specific functionalities. The conventional approach involves training base models with tool-use data (Thoppilan et al., 2022; Dubey et al., 2024) or fine-tuning LLMs (Patil et al., 2023; Schick et al., 2023; Yang et al., 2023; Parisi et al., 2022; Liu et al., 2025; Lin et al., 2025), but may require continual learning as new tools are added. Hao et al. (2023) addressed scalability by training tool embeddings plug-and-play usage, though still requiring labeled data. Alternatively, LLMs can be augmented with meta-prompts or tool-use instructions at inference time (Lu et al., 2023; Shen et al., 2023; Song et al., 2023; Qin et al., 2024b; Zhuang et al., 2024). As the range of applications and tools expands, enhancing LLMs' capacity to handle new tools remain pivotal, which we address through PLAY2PROMPT.

Tool-Use Instructions and Optimization. Tool documentation and example demonstrations are key to prompting LLMs for effective tool use. Hsieh et al. (2023) highlighted the risk of hallucinations when documentation is lacking, and Xu et al. (2023) showed performance declines without in-context examples. To automate generating tool-use instances, Shen et al. (2024) leveraged a graph of tool relations for back-instructing queries, relying on the availability of external tool graphs. Yuan et al. (2024) proposed direct prompting to rewrite tool documentation, which relies on labeled documentation examples and lacks systematic optimization and the ability to measure optimization quality. Qu et al. (2025) explored LLMs interacting with tools, focusing on tool documentation only with single-thread iterative refinement. Automatic prompt tuning (Pryzant et al., 2023; Wang et al., 2024) adapts prompts to domain-specific tasks but require held-out test sets, rendering it unsuitable for zero-shot tool instruction rewriting (Wu et al., 2024). These constraints underscore the need for approaches that optimize tool instructions and demonstrations without labeled data or manual effort, which PLAY2PROMPT achieves by interacting directly with the tool itself.

5 Conclusion

We present PLAY2PROMPT, an automated framework that iteratively refines tool documentation and creates usage demonstrations, enhancing LLMs' tool use in zero-shot settings. Through a search-based trial-and-error process with self-reflection, PLAY2PROMPT allows models to explore and improve tool use without labeled data or extensive manual effort. This approach addresses the limitations of methods that rely on handcrafted prompts or labeled data, providing a scalable and task-agnostic solution for real-world applications.

[5]We omit the complexity metric used in Shen et al. (2024), as PLAY2PROMPT generates single-tool usage examples.

Acknowledgments

This research is supported by the MIT-IBM Watson AI Lab and the Centre for Perceptual and Interactive Intelligence (CPII) Ltd. under the Innovation and Technology Commission's InnoHK Scheme. The views and conclusions are those of the authors and should not be interpreted as representing those of IBM.

Limitations

In this work, PLAY2PROMPT optimizes a single tool, but it's still applicable to queries that require multiple tools by optimizing each tool independently, and then using the optimized examples and documentation from multiple tools together at inference. Although results (on Multiple and Multiple-Parallel in BFCL and the entirety of StableToolBench) on multi-tool queries show sizeable performance gains, scaling the optimization process itself from single-tool scenarios to multiple tools can likely enhance LLM's tool use effectiveness even more. Additionally, for example demonstrations, we use rejection sampling to generate tool invocations first, which do not work for functions whose parameter space is too large, for instance parameters that take long ID string inputs or authentication tokens that require calls to other tools beforehand. Exploring multi-tool dependencies could potentially resolve this issue and improve tool play. In our work we focus on tool documentation and demonstrations only, relegating other information as meta-information, which could be potential next steps to explore.

Ethics Statement

We used publicly available models and datasets for training and evaluation, and did not collect data or any personal information in this work. The trained models could however potentially be misused and pose ethical risks typical of large language models when deployed in real-world applications, if not thoroughly audited.

References

Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*.

Tianle Cai, Xuezhi Wang, Tengyu Ma, Xinyun Chen, and Denny Zhou. 2024. Large language models as tool makers. In *The Twelfth International Conference on Learning Representations*.

Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*.

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

Zhicheng Guo, Sijie Cheng, Hao Wang, Shihao Liang, Yujia Qin, Peng Li, Zhiyuan Liu, Maosong Sun, and Yang Liu. 2024. StableToolBench: Towards stable large-scale benchmarking on tool learning of large language models. In *Findings of the Association for Computational Linguistics ACL 2024*, pages 11143–11156, Bangkok, Thailand and virtual meeting. Association for Computational Linguistics.

Tanmay Gupta and Aniruddha Kembhavi. 2023. Visual programming: Compositional visual reasoning without training. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 14953–14962.

Shibo Hao, Tianyang Liu, Zhen Wang, and Zhiting Hu. 2023. Toolkengpt: Augmenting frozen language models with massive tools via tool embeddings. *Advances in neural information processing systems*, 36.

Cheng-Yu Hsieh, Si-An Chen, Chun-Liang Li, Yasuhisa Fujii, Alexander Ratner, Chen-Yu Lee, Ranjay Krishna, and Tomas Pfister. 2023. Tool documentation enables zero-shot tool-usage with large language models. *arXiv preprint arXiv:2308.00675*.

Qiqiang Lin, Muning Wen, Qiuying Peng, Guanyu Nie, Junwei Liao, Jun Wang, Xiaoyun Mo, Jiamu Zhou, Cheng Cheng, Yin Zhao, Jun Wang, and Weinan Zhang. 2025. Robust function-calling for on-device language model via function masking. In *The Thirteenth International Conference on Learning Representations*.

Weiwen Liu, Xu Huang, Xingshan Zeng, xinlong hao, Shuai Yu, Dexun Li, Shuai Wang, Weinan Gan, Zhengying Liu, Yuanqing Yu, Zezhong WANG, Yuxian Wang, Wu Ning, Yutai Hou, Bin Wang, Chuhan Wu, Wang Xinzhi, Yong Liu, Yasheng Wang, Duyu Tang, Dandan Tu, Lifeng Shang, Xin Jiang, Ruiming Tang, Defu Lian, Qun Liu, and Enhong Chen. 2025. ToolACE: Winning the points of LLM function calling. In *The Thirteenth International Conference on Learning Representations*.

Pan Lu, Baolin Peng, Hao Cheng, Michel Galley, Kai-Wei Chang, Ying Nian Wu, Song-Chun Zhu, and Jianfeng Gao. 2023. Chameleon: Plug-and-play compositional reasoning with large language models. *arXiv preprint arXiv:2304.09842*.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. 2023. Self-refine: iterative refinement with self-feedback. In *Proceedings of the 37th International Conference on Neural Information Processing Systems*, pages 46534–46594.

Grégoire Mialon, Roberto Dessi, Maria Lomeli, Christoforos Nalmpantis, Ramakanth Pasunuru, Roberta Raileanu, Baptiste Roziere, Timo Schick, Jane Dwivedi-Yu, Asli Celikyilmaz, Edouard Grave, Yann LeCun, and Thomas Scialom. 2023. Augmented language models: a survey. *Transactions on Machine Learning Research*. Survey Certification.

Bhargavi Paranjape, Scott Lundberg, Sameer Singh, Hannaneh Hajishirzi, Luke Zettlemoyer, and Marco Tulio Ribeiro. 2023. Art: Automatic multi-step reasoning and tool-use for large language models. *arXiv preprint arXiv:2303.09014*.

Aaron Parisi, Yao Zhao, and Noah Fiedel. 2022. Talm: Tool augmented language models. *arXiv preprint arXiv:2205.12255*.

Shishir G Patil, Tianjun Zhang, Xin Wang, and Joseph E Gonzalez. 2023. Gorilla: Large language model connected with massive apis. *arXiv preprint arXiv:2305.15334*.

Reid Pryzant, Dan Iter, Jerry Li, Yin Lee, Chenguang Zhu, and Michael Zeng. 2023. Automatic prompt optimization with "gradient descent" and beam search. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 7957–7968, Singapore. Association for Computational Linguistics.

Yujia Qin, Shengding Hu, Yankai Lin, Weize Chen, Ning Ding, Ganqu Cui, Zheni Zeng, Yufei Huang, Chaojun Xiao, Chi Han, Yi Ren Fung, Yusheng Su, Huadong Wang, Cheng Qian, Runchu Tian, Kunlun Zhu, Shihao Liang, Xingyu Shen, Bokai Xu, Zhen Zhang, Yining Ye, Bowen Li, Ziwei Tang, Jing Yi, Yuzhang Zhu, Zhenning Dai, Lan Yan, Xin Cong, Yaxi Lu, Weilin Zhao, Yuxiang Huang, Junxi Yan, Xu Han, Xian Sun, Dahai Li, Jason Phang, Cheng Yang, Tongshuang Wu, Heng Ji, Zhiyuan Liu, and Maosong Sun. 2024a. Tool learning with foundation models. *Preprint*, arXiv:2304.08354.

Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, Sihan Zhao, Lauren Hong, Runchu Tian, Ruobing Xie, Jie Zhou, Mark Gerstein, dahai li, Zhiyuan Liu, and Maosong Sun. 2024b. ToolLLM: Facilitating large language models to master 16000+ real-world APIs. In *The Twelfth International Conference on Learning Representations*.

Changle Qu, Sunhao Dai, Xiaochi Wei, Hengyi Cai, Shuaiqiang Wang, Dawei Yin, Jun Xu, and Ji-Rong Wen. 2025. From exploration to mastery: Enabling llms to master tools via self-driven interactions. In *The Thirteenth International Conference on Learning Representations*.

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

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

Yongliang Shen, Kaitao Song, Xu Tan, Wenqi Zhang, Kan Ren, Siyu Yuan, Weiming Lu, Dongsheng Li, and Yueting Zhuang. 2024. Taskbench: Benchmarking large language models for task automation. In *ICLR 2024 Workshop on Large Language Model (LLM) Agents*.

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*.

Yifan Song, Weimin Xiong, Dawei Zhu, Cheng Li, Ke Wang, Ye Tian, and Sujian Li. 2023. Restgpt: Connecting large language models with real-world applications via restful apis. *arXiv preprint arXiv:2306.06624*.

Dídac Surís, Sachit Menon, and Carl Vondrick. 2023. Vipergpt: Visual inference via python execution for reasoning. *arXiv preprint arXiv:2303.08128*.

Gemini Team, Rohan Anil, Sebastian Borgeaud, Yonghui Wu, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M Dai, Anja Hauth, et al. 2023. Gemini: a family of highly capable multimodal models. *arXiv preprint arXiv:2312.11805*.

Romal Thoppilan, Daniel De Freitas, Jamie Hall, Noam Shazeer, Apoorv Kulshreshtha, Heng-Tze Cheng, Alicia Jin, Taylor Bos, Leslie Baker, Yu Du, et al. 2022. Lamda: Language models for dialog applications. *arXiv preprint arXiv:2201.08239*.

Xinyuan Wang, Chenxi Li, Zhen Wang, Fan Bai, Haotian Luo, Jiayou Zhang, Nebojsa Jojic, Eric Xing, and Zhiting Hu. 2024. Promptagent: Strategic planning with language models enables expert-level prompt optimization. In *The Twelfth International Conference on Learning Representations*.

Chenfei Wu, Shengming Yin, Weizhen Qi, Xiaodong Wang, Zecheng Tang, and Nan Duan. 2023. Visual chatgpt: Talking, drawing and editing with visual foundation models. *arXiv preprint arXiv:2303.04671*.

Shirley Wu, Shiyu Zhao, Qian Huang, Kexin Huang, Michihiro Yasunaga, Kaidi Cao, Vassilis N Ioannidis, Karthik Subbian, Jure Leskovec, and James

Zou. 2024. Avatar: Optimizing llm agents for tool-assisted knowledge retrieval. *arXiv preprint arXiv:2406.11200*.

Qiantong Xu, Fenglu Hong, Bo Li, Changran Hu, Zhengyu Chen, and Jian Zhang. 2023. On the tool manipulation capability of open-source large language models. *arXiv preprint arXiv:2305.16504*.

Fanjia Yan, Huanzhi Mao, Charlie Cheng-Jie Ji, Tianjun Zhang, Shishir G. Patil, Ion Stoica, and Joseph E. Gonzalez. 2024. Berkeley function calling leaderboard.

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.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. 2023. React: Synergizing reasoning and acting in language models. In *The Eleventh International Conference on Learning Representations*.

Donggeun Yoo and In So Kweon. 2019. Learning loss for active learning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*.

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. *arXiv preprint arXiv:2401.06201*.

Kexun Zhang, Hongqiao Chen, Lei Li, and William Wang. 2023a. Syntax error-free and generalizable tool use for llms via finite-state decoding. *arXiv preprint arXiv:2310.07075*.

Tianhua Zhang, Jiaxin Ge, Hongyin Luo, Yung-Sung Chuang, Mingye Gao, Yuan Gong, Xixin Wu, Yoon Kim, Helen Meng, and James Glass. 2023b. Natural language embedded programs for hybrid language symbolic reasoning. *Preprint*, arXiv:2309.10814.

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 Dataset License

BFCL and StableToolBench are both licensed under Apache License 2.0. We adhere to intended uses stated in the license.

B Additional Implementation Details

For task LLM $\mathcal{M}_T$, `llama-3.1-8b-instruct` is used for LLaMA-8B, `llama-3.3-70b-instruct` for LLaMA-70B, `gpt-3.5-turbo-0125` for GPT-3.5, `gpt-4o-2024-11-20` for GPT-4o, `toolace-2-llama-3.1-8b` for ToolACE-8B, and `hammer2.1-7b` for Hammer-7B. On StableToolBench, we use `llama-3.3-70b-instruct` as the evaluation LLM, which produces stable assessments.

For GPT models, we keep the exact same inference setting as used in the original benchmark, except for requiring it to return a single action at each step from the provided toolset by supplying a flag to the OpenAI API. For LLaMA models, on BFCL we follow the exact official inference prompts, and on StableToolBench we adapt the ReAct prompts into LLaMA-3's prompt format but keep everything else fixed as much as possible. For ToolACE and Hammer on BFCL, we also follow the exact official inference prompts.

During inference, for PLAY2PROMPT we set the number of tool-use example demonstrations to 5 per tool for BFCL and 1 per tool for StableToolBench, as StableToolBench averages more tools per query. The sampling temperature is set to 0.001. We report scores averaged over 3 independent runs.

As both our optimization and inference stages perform only LLM inference, we call hosted inference APIs and do not report total computation in GPU hours. An estimated 1M API calls were made in total for this work.

C Details on Robustness Experiments

To assess the robustness of PLAY2PROMPT, we artificially introduce noise to tool documentation in BFCL Executable by dropping each parameter description with a probability p, leaving general tool descriptions and signatures intact. As shown in table 8, with $p = 0.5$, this degradation reduces baseline performance by about 5% for the LLaMA models and GPT-4o, and by 10% for GPT-3.5, especially for the most challenging Multiple-Parallel category (c.f. table 1). With PLAY2PROMPT, performance consistently exceeds baselines across all models, improving by 3-4% for the larger LLaMA-70B and GPT-4o models, 5% for LLaMA-8B, and most significantly, recovering the full 10% loss for GPT-3.5.

In table 9, we further include experiments on BFCL with an even higher parameter description

Base Model	Method	Simple-Python	Simple-REST	Multiple	Parallel	Multiple Parallel	Weighted Avg	Avg
LLaMA-8B	Prompting	91.0	67.1	92.0	84.0	70.0	81.3	80.8
	+PLAY2PROMPT	94.0	81.4	92.0	82.0	82.5	**86.1**	**86.4**
LLaMA-70B	Prompting	96.0	90.0	92.0	80.0	72.5	84.4	86.1
	+PLAY2PROMPT	95.0	88.6	92.0	84.0	87.5	**88.8**	**89.4**
GPT-3.5	Function-calling	92.0	87.1	84.0	76.0	50.0	74.9	77.8
	+PLAY2PROMPT	98.0	88.6	86.0	84.0	80.0	**85.8**	**87.3**
GPT-4o	Function-calling	93.0	95.7	92.0	88.0	75.0	87.3	88.7
	+PLAY2PROMPT	98.0	94.3	94.0	88.0	82.5	**90.3**	**91.6**
ToolACE-8B	Function-calling	92.0	84.3	86.0	80.0	57.5	77.9	80.0
	+PLAY2PROMPT	93.0	84.3	90.0	82.0	72.5	**83.3**	**84.4**
Hammer-7B	Function-calling	93.0	71.4	78.0	82.0	62.5	76.2	77.4
	+PLAY2PROMPT	92.0	82.9	86.0	82.0	75.0	**82.6**	**83.6**

Table 8: Results on BFCL Executable, with parameter description dropout $p = 0.5$. Accuracy scores are shown.

Base Model	Method	Simple-Python	Simple-REST	Multiple	Parallel	Multiple Parallel	Weighted Avg	Avg
LLaMA-8B	Prompting	85.0	62.9	84.0	82.0	55.0	73.7	73.8
	+PLAY2PROMPT	88.0	78.6	90.0	82.0	67.5	**80.7**	**81.2**
LLaMA-70B	Prompting	86.0	87.1	86.0	76.0	55.0	75.9	78.0
	+PLAY2PROMPT	89.0	87.1	92.0	80.0	65.0	**81.3**	**82.6**
GPT-3.5	Function-calling	84.0	87.1	78.0	72.0	40.0	68.9	72.2
	+PLAY2PROMPT	92.0	84.3	82.0	86.0	52.5	**77.2**	**79.4**
GPT-4o	Function-calling	85.0	95.7	84.0	84.0	47.5	76.5	79.2
	+PLAY2PROMPT	91.0	94.3	92.0	88.0	60.0	**83.2**	**85.0**
ToolACE-8B	Function-calling	88.0	80.0	86.0	82.0	52.5	76.1	77.7
	+PLAY2PROMPT	89.0	85.7	90.0	86.0	65.0	**82.1**	**83.1**
Hammer-7B	Function-calling	85.0	75.7	76.0	84.0	52.5	73.2	74.6
	+PLAY2PROMPT	90.0	82.9	88.0	86.0	70.0	**82.6**	**83.4**

Table 9: Results on BFCL Executable, with parameter description dropout $p = 1.0$. Accuracy scores are shown.

dropout with $p = 1.0$, that is, dropping out all parameter descriptions. The documentation still contains information from the overall tool description and parameter name in this scenario. These results illustrate PLAY2PROMPT's robustness to incomplete documentation, and especially shines when initial documentation is poor.

D Additional Qualitative Example

We show another example to illustrate how PLAY2PROMPT commonly aids LLMs' tool usage in figure 4.

E Detailed Optimization Procedures

Below, we present the full procedures for both example demonstration optimization and documentation optimization as pseudo-code, shown in Algorithm 1 and Algorithm 2, respectively. Please refer to Section 2.1 for notations. In the algorithms, m_i represents meta-prompts.

F Meta-Prompts

The meta-prompts are listed below in tables 10 through 17.

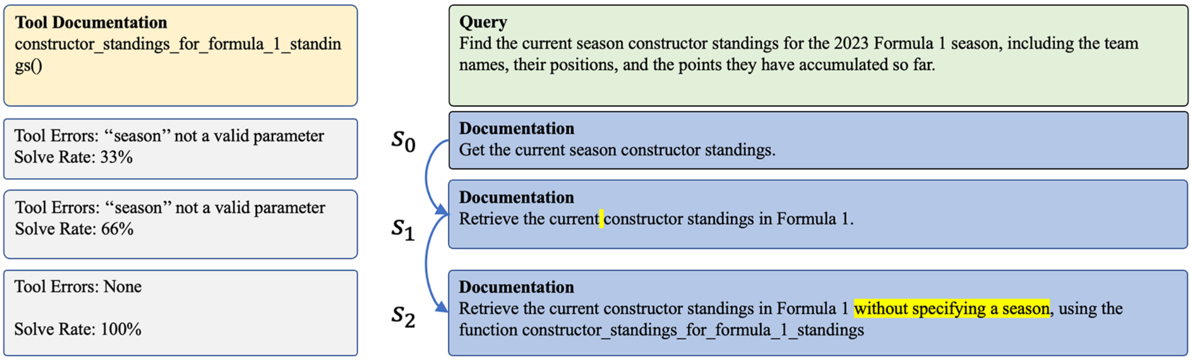

Figure 4: A typical example of PLAY2PROMPT assisting LLMs in correcting errors in generating parameter values. The task model $\mathcal{M}_T$ (LLaMA-8B) gets confused by the query specifying the year, which PLAY2PROMPT first attempts to remove "year" from the description, and further explicitly prompts $\mathcal{M}_T$ to not use the parameter.

Algorithm 1 EXAMPLEOPTIMIZATIONSTEP

Input: $\mathcal{S}_t = E_t = (x_t, F, I_t, y_t)$: tool-use example, $\mathcal{D}_0$: documentation, a_t: reflection

Output: $\mathcal{S}_{t+1} = E_{t+1} = (x_{t+1}, F, I_{t+1}, y_{t+1})$: updated example, r_{t+1}: score, a_{t+1}: reflection

$c \leftarrow$ false
while $\neg c$ **do** ▷ Rejection Sampling of tool invocation i
  $I_{t+1} \sim p_{\mathcal{G}_E}(I|I_t, c, \mathcal{D}_0, a_t, m_1)$ ▷ Sample candidate tool invocation, incorporating reflection a_t
  $o_{t+1} \leftarrow F(I_{t+1})$ ▷ Execute tool function
  $c \sim p_{\mathcal{G}_E}(c|I_{t+1}, o_{t+1}, \mathcal{D}_0, m_2)$ ▷ Verify validity of tool call
end while
for $n \leftarrow 1$ to N_E **do** ▷ Rollout for with self-reflection policy
  $x_{t+1} \sim p_{\mathcal{G}_E}(x|I_{t+1}, o_{t+1}, \mathcal{D}_0, m_3)$ ▷ Sample user query x
  $y_{t+1} \sim p_{\mathcal{G}_E}(y|I_{t+1}, o_{t+1}, x_{t+1}, \mathcal{D}_0, m_4)$ ▷ Sample corresponding answer y
  $\mathcal{R}_q \sim p_{\mathcal{G}_E}(r|y_{t+1}, I_{t+1}, o_{t+1}, x_{t+1}, \mathcal{D}_0, m_5)$ ▷ Evaluate example quality
  $\mathcal{R}_e \leftarrow -\mathcal{P}\{\mathcal{M}_T(x_{t+1}; \mathcal{D}_0, \varnothing); y_{t+1}, F, I_{t+1}\}$ ▷ Evaluate example difficulty
  $r_{t+1} \leftarrow \mathcal{R}_q + \mathcal{R}_e$
  $a_{t+1} \sim p_{\mathcal{G}_E}(a|r_{t+1}, y_{t+1}, i_{t+1}, o_{t+1}, x_{t+1}, \mathcal{D}_0, m_6)$ ▷ Generate self-reflection action
end for

Algorithm 2 DOCUMENTATIONOPTIMIZATIONSTEP

Input: $\mathcal{S}_t = D_t$: documentation, $\mathcal{E} = \{(x_j, F, I_j, y_j)\}_{j=1}^{W}$: validation set, a_t: reflection

Output: : $\mathcal{S}_{t+1} = \mathcal{D}_{t+1}$: updated documentation, r_{t+1}: score, a_{t+1}: reflection

$\mathcal{D}_{t+1} \sim p_{\mathcal{G}_D}(\mathcal{D}|\mathcal{D}_t, a_t, m_7)$ ▷ Sample documentation $\mathcal{D}_{t+1}$ from $\mathcal{G}_D$, applying reflection a_t
$\hat{M}_j, e_j \leftarrow \mathcal{M}_T(x_j; \mathcal{D}_{t+1}, \varnothing)\ \forall j$ ▷ Gather I/O & errors e_j from running $\mathcal{M}_T$ on validation set $\mathcal{E}$ with $\mathcal{D}_{t+1}$
$r_{t+1} \leftarrow \mathbb{E}_j[\mathcal{P}\{\hat{M}_j; y_j, F, I_j\}]$ ▷ Evaluation score on $\mathcal{E}$
$a_{t+1} \sim p_{\mathcal{G}_D}(a|\mathcal{D}_{t+1}, r_{t+1}, \{x_j, I_j, y_j, \hat{M}_j, e_j\}, m_8)$ ▷ Self-reflection action

You are given an API tool with the following documentation: `{Documentation}` Your task is to write 1 example API call for the given API tool given its purpose and parameters list. The API call you produced will be executed as function call later and return result if correct, or error if you provide incorrect syntax, format, or parameters. Given the documentation and description, think of possible example API calls and produce those that are likely to be correctly executed. Think of parameter values that are reasonable, make sense, and are likely API calls that people use in the real world. The generated API call must be executable and real. Parameter values must be filled in and not placeholding text. You must include the required parameters, and optionally give parameters that are labeled as "optional parameters". Do not hallucinate and produce parameters that are not under "required" or "optional". Produce diverse parameter values, but be factual and do not use fake parameters. You can only use the given function `{function_name}`. Create an API call that include the function name, and the parameters to be input to the API. Include all the required and optional parameters in a single dictionary without separating them. Do not include the URL or other irrelevant information. The output should be in the following JSON format that represents a function call: {"name": "function_name","parameters": {"properties": {"parameter_1": value 1}}} You must strictly follow the output format, including "name", "parameters", "properties", and parameters. Previously you generated the following API calls for this function, which where then executed and critiqued: fn_call="`{fn_call}`" fn_output="`{fn_output}`" status=`{status}` reflection="`{reflection}`"

Table 10: Meta-prompt m_1

You are given an API tool with the following documentation: `{Documentation}` Previously you were asked to write an example API call for the function `{function_name}` given its purpose and parameters list, and you generated the following function call: `{fn_call}`. The function call you produced was later executed and returned the following result: "`{fn_output}`". Your task is to analyze the response and check if there are any errors. 1. If there are no errors and everything looks reasonable, give an err_code of 0, and don't provide analysis. 2. If there is an error, give an err_code of -1. Then in your analysis, describe and analyze in detail why the error occurred based on the error message. Then, based on your analysis, give detailed suggestions to improve the function call so that no errors will be produced. You must give detailed analysis and suggestions, do not simply repeat the error message. The analysis and suggestions should be in the "analysis" field in the output. Note that even if the "error" field in the result is empty, the "response" field may contain an error when using the function call. If this is the case you must treat this as an error and analyze the failure. The response field may also be in HTML format. Your output should be in the following JSON format: {"analysis": your analysis and suggestions, "err_code": error code (-1 for error, 0 for correct)}

Table 11: Meta-prompt m_2

You are given an API tool with the following documentation: `{Documentation}` For the function `{function_name}`, you are given the following function call: `{fn_call}`, and executing the function call returned the following result: `{fn_output}`. Your task is to generate a user instruction in natural language that requires the given function call to be completed. Here are some guidelines to follow: 1. The instruction must be a scenario or problem that cannot be solved without calling the given function `{function_name}`. This is your main objective. 2. You should not directly or explicitly ask for the function to be called; the problem itself must inherently be solved by the function. 3. Based on the function, function call, its parameters, parameter values, and function execution responses, you should produce a real and reasonable instruction. 4. You must use information from the parameter values of the function call to create the response. You must include the value of every parameter from the given function call in the user instruction you generated, including each list/dict element of the parameter values. Do not ignore any parameters/values from the function call. 5. You must not include specific function calls in your response. You should not explicitly show the function names. You should also never explicitly name the parameter names in your response. You should not show any variable names. 6. Your response has to be in natural language. Do not show any variables, function calls, or code. 7. You should respond in the user's first-person perspective. 8. You are a human user. You are asking a question or giving an instruction. Do not answer in the perspective of an AI assistant. Remember, the user does not know about the API function and thus cannot ask to call the function. 9. Remember, you are asking a question, so do not answer your own question in the response. Your goal is to give a querying instruction or question, not producing answers or function calls. Your output should be in the following JSON format: {"instruction": generated instruction} Previously you generated the following instructions for this function call, which were rated and analyzed: instruction="`{instruction}`" score=`{score}` Based on these ratings, you are given the following analysis: `{reflection}`. You should improve your instructions based on these suggestions.

Table 12: Meta-prompt m_3

You are given an API tool with the following documentation: `{Documentation}` You are given the following instruction: "`{instruction}`" To produce a response to the instruction, you made an API call to the given tool, which returned the following results: `{fn_output}` Given the instruction and the results of API call, produce an effective and short answer to the user in natural language. Your answer must be based on the results of the API call, do not hallucinate or answer anything not in the API results. You must not include code, comments, JSON data structures, notes, or other irrelevant information in your answer. If there is an error or failure using the tool, you must report the error in your answer and do not make things up, especially when you receive an input about invalid parameters.

Table 13: Meta-prompt m_4

You are given an instruction "`{instruction}`", function call "`{fn_call}`" and an answer "`{answer}`", your task is to give a 'score' based on the following rules: 1. You must return 1 if any of the following conditions is met (for instruction only): (1) instruction is empty, nonsense, or not in natural language; or (2) instruction is explicitly including function calls or asking for function calls or contains function names; or (3) instruction includes exact function parameter names; or (4) instruction includes code or variable assignment; or (5) instruction is longer than 3 sentences or 300 letters; or (6) instruction does not include a question, query, request, or problem to be solved; or (7) instruction is not in first-person perspective, or is in the perspective of an AI assistant instead of a user; or (8) any parameter value in the function call is not present in the instruction An instruction that satisfies any of these conditions is a bad instruction and should be scored a 1. 2. If the answer is a error message or mentions any errors (API error, invalid parameter error, ..., etc.), mentions cannot use API or cannot respond, return 1. 3. If the answer is a positive response for the given instruction, you have to further check. 3.1 If the answer is not sufficient to determine whether they solve the instruction or not, return 2. 3.2 If you are confident that the answer is sufficient to determine whether the solve the instruction or not, return 3 if solvable or 1 if unsolvable. Finally, organize your output in the following JSON format:{"analysis": your reasoning, "score": score}

Table 14: Meta-prompt m_5

You are given an API tool with the following documentation: {Documentation}

Previously, given the function call {fn_call}, you were asked to generate example instructions that require the use of the function {function_name} to complete. The example instructions generated by you were then scored by an expert on whether the instructions can be fulfilled using the given API function. Scores are in a scale between 1 (lowest) and 3 (highest).

Below are the generated instructions, scores, and analyses:
instruction="{instruction}" score={score} analysis="{analysis}"

Task:
1. Firstly, identify and contrast the patterns of instructions and function calls that have achieved good scores with those that have not. If there are no bad scores, only summarize the patterns of the good ones.
2. Next, specify the suggestions that can lead to improved performance for the generated instructions and function calls with bad scores. You should focus on capturing the high-level pattern of the examples relevant to the API documentation. Note that both the function and the function call cannot be changed, and focus your suggestions on how to improve the example instructions, including deciding what information to use from parameters of the function call.

Table 15: Meta-prompt m_6

You are given an API tool with the following documentation: {Documentation}

Previously, the given tool was used in solving instructions by a tool assistant with the following descriptions:
Iteration #{iteration}, description="{description}", score={score}%, stdev={stdev}.

Furthermore, an analysis was performed on the descriptions for the previous iterations: "{analysis}"

Your task is to further enhance the description for the function {function_name} based on these results for the next iteration, with the objective of maximizing the score, minimizing the stdev, and help the assistant correctly use the function without errors. The descriptions for each parameter might be unclear, underspecified, or incorrect, so you should include clear parameter descriptions and usage for every single required and optional parameter, including its type, usage, and possible values. Be as clear, descriptive, and comprehensive as possible. Be factual and do not consider parameters that are not listed. Incorporate the analysis and generate the enhanced descriptions.

Table 16: Meta-prompt m_7

You are given an API tool with the following documentation: {Documentation}

Previously, the given tool was used in solving instructions by a tool assistant with the following descriptions:
Iteration #{iteration}, description="{description}"

Here are the instructions the assistant tried to solve with this tool description, with their corresponding answers and errors produced by the assistant:
instruction="{instruction}", answer="{answer}", errors: function_call={function_name}, arguments={arguments}, error={error_message}, ground truth should be {fn_call}

Overall the performance of this description is: score={score}

Now your task is to critique the descriptions based on these results. A good description maximizes the score, minimizing the stdev, and helps the assistant correctly use the function without errors. In your analysis:
(1) Identify how the descriptions affect the function call errors of the assistant. Be specific on which errors the assistant tends to make, and find patterns in the description that causes the assistant to make such errors.
(2) Identify and contrast the patterns of descriptions that have achieved good scores (> 60%) with those that have not. Analyze how the description can be improved.

Table 17: Meta-prompt m_8