Abstract

Language models (LMs) are powerful yet mostly for text generation tasks. Tools have substantially enhanced their performance for tasks that require complex skills. However, many works adopt the term “tool” in different ways, raising the question: What is a tool anyway? Subsequently, where and how do tools help LMs? In this survey, we provide a unified definition of tools as external programs used by LMs, and perform a systematic review of LM tooling scenarios and approaches. Grounded on this review, we empirically study the efficiency of various tooling methods by measuring their required compute and performance gains on various benchmarks, and highlight some challenges and potential future research in the field.

1 Introduction

Language Models (LMs) have become increasingly effective in solving text-generation tasks, by taking in natural language (NL) instructions from users and outputting NL responses, such as answering the “What is the capital of the US?” with “Washington D.C.”. However, LMs often struggle to perform tasks that require complex skills (e.g., math or complex reasoning), and are fundamentally unable to solve other tasks that require access to information not included in their training data (e.g., the current weather or date).

To solve this problem, researchers and practitioners are turning to LMs enhanced with tools, which help facilitate the task-solving process of LMs, or extend LMs with new abilities that the LM does not possess otherwise (Qin et al., 2023; Mialon et al., 2023). For example, a calculator tool may be used to facilitate mathematical calculations, or a get_time() tool could be used to obtain the current time, which is not available purely through the LM’s parameters. Inspired by the tools used by humans (Shumaker et al., 2011), some works introduce application-specific software as tools, such as using a search engine to obtain knowledge (Lazaridou et al., 2022; Komeili et al., 2022), using a translator to process unknown languages (Schick et al., 2023), or using a SQL engine to query databases (Hao et al., 2023). With the development of numerous application programming interfaces (APIs) on the web, many works collect APIs as tools to access world data in real-time (Balog et al., 2016; Xu et al., 2023; Qin et al., 2024) via multiple modalities (Yang et al., 2023), even performing professional activities such as financial analysis (Li et al., 2023) and digital marketing (Huang et al., 2024). Instead of using black-box APIs with unseen implementations, other works use locally-crafted functions to query over structured tables (Wang et al., 2024a; Cao et al., 2023) or images (Suris et al., 2023), where the function tools can be created by humans (Gupta & Kembhavi, 2022) or model experts (Wang et al., 2023a; Cai et al., 2023; Wang et al., 2024b).

However, despite this broad and burgeoning area of tool use in LMs, existing surveys only cover certain tool categories such as software (Mialon et al., 2023) or APIs (Qin et al., 2023).
In this paper, we (1) provide a unified view of tool use across a broad range of scenarios, (2) empirically analyze the cost efficiency of tooling methods, to give practical guidance on when and how one should use tools, and (3) offer concrete suggestions for evaluations.

We start with proposing a unified definition of tools and explain why tools help task-solving (§2). We first introduce the basic tool-use paradigm (§3) and study a variety of tool-using scenarios by enumerating which tools exist and to which tasks they apply (§4). Next, we study advanced approaches for complex tool usage and even make new tools if they are unavailable for the task (§5). We then summarize existing testbeds and evaluation metrics across LM tooling works, and highlight several missing aspects with concrete metric suggestions (§6). Lastly, grounding on our empirical analysis about when tools are effective, we identify the most efficient tooling approaches and the tasks that benefit most from tools (§7).

2 Background

2.1 What are tools?

Because LMs are products of the digital world, tools employed by LMs are often computer programs that are executable in corresponding environments, e.g., Python programs are executable in Python environments. Referring back to human-used tools, Shumaker et al. (2011) defines animal tool use as "the external employment of an unattached or manipulable attached environmental object to alter more efficiently the form, position, or condition of another object." Similar to this definition of physical tools, LM-used program tools should also be external to the employer (i.e., the LM) and are part of the environment. In the meantime, instead of arbitrary program snippets, a tool is a function (e.g., plus_one), meaning that it can be applied to other objects (e.g., data) and yield an output (e.g., plus_one(1) → 2).

Existing definitions of LM-used tools touch on some of these aspects. Qin et al. (2023) make an intuitive appeal to the similarity to human tool use, but do not define what entails a tool. Mialon et al. (2023) define a tool as "an external module that is typically called using a rule or a special token and whose output is included in the augmented LM’s context." We argue for a somewhat more broad definition than this, which encompasses a wide variety of more recent works on tool usage:

**Definition 1.** An LM-used tool is a function interface to a computer program that runs externally to the LM, where the LM generates the function calls and input arguments in order to use the tool.

2.2 Why are tools helpful?

Tools can help task-solving in different ways, depending on the functionality of individual tools. We summarize their functions into three major categories: perception, action, and computation. A tool may belong to one or more of these three categories.

**Perception** Perception tools provide or collect information from the environment. An example is using a get_time() API to obtain the current time, which is not included in the LM’s parametric knowledge learned from training.

**Action** Action tools can exert actions on the environment and change its state. For example, turn_left() can shift the direction of an embodied agent, or executing make_post(website, post) can change the content on a website.

**Computation** Computation tools do not necessarily perceive or modify the external environment, but use programs to tackle complex computational tasks. For example, a calculator is a computation tool for mathematical calculation. Note that the computation also includes more general acts of computing beyond numerical calculation. Therefore, a translator is also a computation tool that can be used to translate between languages.

Note that many tools can fall into multiple categories. For instance, a search engine is a tool that can perform both computation and perception. As computation, it measures

---

1 Under our definition, tool functions can be implemented by any means, including symbolic computation or neural networks — the functions only require a programmatic interface.
document similarity and selects relevant ones, but it also perceives the environment (i.e., the web) and fetches data (i.e., returned documents) from it. In a similar spirit, SQL queries can be used as computation tools (e.g., SELECT SQRT(16) / 10 AS result), perception tools for viewing data (e.g., SELECT name FROM data), action tools to modify data (e.g., INSERT INTO data VALUES name), or all of the above (e.g., INSERT INTO counts (grp_id, grp_cnt) SELECT grp_id, COUNT(*) FROM data GROUP BY grp_id).

2.3 Tools and “Agents”

There has recently been a burgeoning of work on LM-powered agents (Xi et al., 2023; Sumers et al., 2024; Russell & Norvig, 2010) define agents as “anything that can be viewed as perceiving its environment through sensors and acting upon that environment through actuators.” According to this definition, agents are programs that use perception tools to perceive the situated environment, or action tools to interact with the environment. Models that only use computation tools and do not interact with their environments through perception or action tools arguably do not fall under the category of “agents” according to this definition.

3 The basic tool use paradigm

First, in this section, we show an illustrative example of a basic tool-use paradigm introduced by Toolformer (Schick et al., 2023), which many tool-related works adopt (Figure 2). Assuming an LM communicates with users mainly in natural language, upon receiving a user query such as “How is the weather today?”, the LM then proceeds to generate either text or tool calls. In the example, starts with generating a few tokens of text “It is …”. When the LM needs to seek external tools to complete the task, e.g., get real-time weather information, it generates tokens of the tool name and corresponding input arguments enclosed with (parentheses) to construct a complete tool calling expression. This completed expression will trigger a shift from text-generation mode to tool-execution mode. The server hosting the tool will execute the expression and return the execution result to the LM.

Taking the example in Figure 2, the LM sends the check_weather() call to the weather server and receives the output “sunny”. The returned result replaces the tool call in the LM-generated tokens (e.g., from “It is check_weather()” to “It is sunny”), which is used for subsequent steps of generation. Accordingly, the LM shifts back to the text generation mode and continues to finish the response by generating new text tokens, e.g., adding ‘today’, and finally returning the response to the user.

In order for LMs to use this basic paradigm of using tools, current works mainly leverage inference-time prompting and training-time learning methods.

Inference-time prompting  Leveraging the ability of LMs to learn in-context (Brown et al., 2020), many works provide tool information through a prompt and expect LMs to acquire abilities to use these tools from input contexts. This is achieved by providing instructions about the task, example pairs of queries and solutions that use tools (Gupta & Kembhavi, 2022; Lu et al., 2023a; Paranjape et al., 2023; Shen et al., 2023a; Yang et al., 2023), and/or documentation of the tools’ functionality (Hsieh et al., 2023).

Learning by training  Beyond learning tools from test-time contexts, LMs can learn from examples that use these tools during training. LMs can simply be trained to generate tool-using solutions, where the examples can be manually annotated by humans (Li et al., 2023), synthesized by larger teacher LMs (Tang et al., 2023; Qin et al., 2024; Huang et al., 2024), or bootstrapped by the test-time LM itself (Schick et al., 2023).
4 Scenarios where tools are useful

While LMs may easily learn to do many tasks to high accuracy without tools, many other tasks greatly benefit from tool use. In this section, we study a broad range of scenarios where tools have been used to assist agents. We discuss tasks where human-created, application-specific tools can improve their performance or other positive aspects (§4.1), as well as scenarios where tools may not be as useful (§4.2).

4.1 Utilizing existing tools for specific applications

While it is difficult to exhaustively enumerate every scenario where tools could be useful, we summarize some major categories of tool use in Table 1 and below. Note that a tool may fall into one or more categories.

<table>
<thead>
<tr>
<th>Category</th>
<th>Example Tools</th>
</tr>
</thead>
<tbody>
<tr>
<td>Computation activities</td>
<td>calculator(formula: str) -&gt; value: int</td>
</tr>
<tr>
<td>Interaction w/ the world</td>
<td>get_weather(city_name: str) -&gt; weather: str get_location(ip: str) -&gt; location: str calendar.fetch_events(date: str) -&gt; events: list email.verify(address: str) -&gt; result: bool</td>
</tr>
<tr>
<td>Non-textual modalities</td>
<td>cat_image.delete(image_id: str) -&gt; None spotify.play_music(name: str) -&gt; None visual_qa(query: str, image: Image) -&gt; answer: str</td>
</tr>
<tr>
<td>Special-skilled LMs</td>
<td>QA(question: str) -&gt; answer: str translation(text: str, language: str) -&gt; text: str</td>
</tr>
</tbody>
</table>

Table 1: Exemplar tools for each category.

4 Computation activities

Complex computing activities such as math calculations are known to be challenging for neural LMs (Schick et al., 2023). While even a calculator can enhance LMs’ numeracy abilities (Parisi et al., 2022; Hao et al., 2023), more generic Python programs are also employed to aid reasoning tasks (Gao et al., 2023b; Chen et al., 2023; Wang et al., 2023b). For more complex professional jobs, business tools are also applied, such as using worksheet to manipulate Google Sheets (Xu et al., 2023), or even tools for financial, medical, education, or advertising domains (Tang et al., 2023; Huang et al., 2024).

Interaction with the world

LMs without tools are fundamentally unable to perceive and act in the world around them, necessitating tool use where such perception and action is necessary. For instance, LMs can access real-time information such as weather (Xu et al., 2023; Tang et al., 2023), or positional knowledge such as location (Qin et al., 2023). On the other hand, LMs can manipulate real-world information such as managing calendars (Schick et al., 2023) and emails (Qin et al., 2024). In addition to web-based activities, LMs can engage in physical activities in embodied environments, such as fishing with rods or mining with axes in the Minecraft world (Wang et al., 2023a); further propagate to the real-world
tasks to perform cooking (Singh et al., 2022; Shridhar et al., 2020), plotting (Liang et al., 2023), and even conducting chemical research (Boiko et al., 2023).

Non-textual modalities While many LMs only consume and generate texts, some works bring in access to visual (Gupta & Kembhavi, 2022; Suris et al., 2023), audio (Yang et al., 2023; Gao et al., 2023a), or other modalities. For example, LMs can access images with cat_image APIs (Xu et al., 2023; Tang et al., 2023) or songs (Huang et al., 2024) provided by spotify, even answer questions about them (Gupta & Kembhavi, 2022; Gao et al., 2023a).

Accessing specialized LMs Some works propose to use specialized LMs as tools, essentially using the main LM as a task planner to dispatch requests to other LMs. Schick et al. (2023) propose QA models to fill in factoid details in responses, Thoppilan et al. (2022); Schick et al. (2023); Paranjape et al. (2023) use machine translation models to assist multilingual tasks. Beyond specific tasks, some works adopt multiple neural models from Hugginface or similar platforms (Patil et al., 2023; Shen et al., 2023a), or further fine-tune them on various data (Viswanathan et al., 2023). Compared to the base LM, these tool models mainly vary in their specialized skills, and may or may not have substantial architectural differences from the base LMs.

4.2 Where are tools not useful?

Despite the fact that tools can be helpful under many scenarios discussed above, it is also important to note scenarios where tools are arguably not very helpful. Some examples of tasks where tools have not (yet) been used to great effect include machine translation, summarization, and sentiment analysis (among others). These are tasks that are not easy to perform using non-ML methods (c.f. solving math problems or accessing databases, which can be done using a calculator or SQL), and can be performed with high accuracy by a powerful LM alone. One intuitive reason is that the tools currently leveraged for these tasks are neural networks and have limited advantages over the base LM. Imagine if we leverage tools on these tasks, the tools would mostly generally be another neural LM with specialized skills, e.g., an LM specifically trained on many summarization datasets to perform this task. However, this special-skilled neural LM may not have significant architectural differences from the base tool-using LM, or be smaller in size or training tokens hence having inferior language modeling abilities in general. In comparison, the base LM capable of solution planning and tool management, usually are more powerful (e.g., GPT-4) and can achieve reasonable performance on a wide variety of tasks, perhaps even outperforming special-purpose LMs (Robinson et al., 2023).

5 Advanced tool-use methods

Given this understanding of the basic tooling paradigm and the scenarios in which tools are useful, we now discuss more advanced approaches for tools. Concretely, we study multi-tool selection and usage (§5.1), complex tooling under programmatic contexts (§5.2), and creation of tools when they are not available a-priori (§5.3).

5.1 Complex tool selection and usage

Depending on the number of tools available, the system may include an implicit or explicit tool selection module. If tools are already designated for the task (Lazaridou et al., 2022; Thoppilan et al., 2022), then no tool selection is needed. If a small number (e.g., 5–10) of tools are available, metadata and use cases of these tools can be provided as input contexts along with the user query (Schick et al., 2023; Paranjape et al., 2023), and LMs can directly select tools from contexts via a standard generation process. If the toolbox size further grows (e.g., to hundreds), fitting all tools into model inputs is not feasible. Thus an extra retrieval step is often incorporated: a retriever model short-lists the most relevant tools and feeds their metadata to the solution-generation LM. Specifically, Zhou et al. (2023); Qin et al. (2024) train retriever models that map NL intents to tool documentation. Yuan et al. (2023) ask LMs to write hypothetical descriptions and use the SimCSE retriever (Gao et al., 2021) to
find similar tools. More easily, one can directly use off-the-shelf embeddings (Meng et al., 2024; OpenAI) or training-free sparse retrievers (Robertson et al., 2009).

For complex queries that require multiple tools to solve, the common approach so far is to break down the task and tackle each step sequentially (Paranjape et al., 2023) by selecting and using tools with intermediate contexts. However, this sequential multi-turn paradigm may not be reflective of more complex or realistic usage of the involved tools. For example, a user may prefer nested function calls `check_weather(set_local_time('Pittsburgh'))` to allow information hiding or encapsulation (Rogers, 2001), parallel calls to reduce round trips with the API (Eleti et al., 2023), or iterative calls `buy_ticket(event)` in a loop until it returns True to indicate a successful transaction.

5.2 Tools in programmatic contexts

Unlike text-based tasks where tools are auxiliary modules to extend LM abilities, on programmatic tasks, where code LMs can solve the problem by generating programs, tools can be seen as compositions of basic functions. In this part, we discuss tools in programmatic tasks for domain-specific (§5.2.1) and general-purpose problems (§5.2.2).

Focus on varied tools Depending on the tasks of interest, existing works focus on different types of tools under programmatic contexts. With the increasing complexity of these tools and presumably a decreasing familiarity of LMs about them, there are works that adopt (i) built-in functions of a programming language (PL) to augment LMs in symbolic reasoning, (ii) external libraries in pre-designed packages to tackle complex open-domain coding queries (Wang et al., 2023c), and (iii) utility functions unseen at training time to solve specific tasks.

![CodeLM](image)

Figure 3: Relative to what is considered as the base LM or base actions, tools can refer to built-in functions, external libraries, or task-specific utility functions (from left to right).

5.2.1 Domain-specific semantic parsing

NL-to-code generation systems have been studied for many years on special-domain tasks such as querying databases (Zelle & Mooney, 1996; Zettlemoyer & Collins, 2012) or knowledge graphs (Berant et al., 2013). Code produced by these systems is often domain-specific logical forms (DSL) manually designed by experts, such as lambda expressions (Liang, 2013) or SQL queries (Yu et al., 2018), and more recently, the QDMR grammar (Wolfson et al., 2020) as an extension to SQL. In addition to knowledge-oriented tasks, many agentic tasks adopt DSL to operate in corresponding environments, such as click or type in web navigation (Liu et al., 2018; Yao et al., 2022; Zhou et al., 2024), `placeItem` and `killMob` in the embodied Minecraft world (Wang et al., 2023a), or `set_joint_target` for robot dogs (Yu et al., 2023). Because DSLs are often specific enough to the target problems, most works directly use these built-in actions. Yet still, for complex task queries, solution programs written in basic DSL actions alone can be hard to interpret or cumbersome to use, e.g., it is hard to tell that the lambda expression `(fold xs (λ (n x) (+ 1 n)) 0)` is to calculate the length of `xs`.

5.2.2 General-purpose code generation

Recent code generation systems have expanded from using DSL to more general-purpose PLs such as Python or Java (Yin & Neubig, 2017; Chen et al., 2021). These languages enable more programming flexibility and readily apply to versatile scenarios. As we have
introduced using built-in actions as tools in §4.1, we discuss more on two other common categories of tools for code LMs, namely external libraries and task-specific utility functions.

### External libraries

From the usage of PLs, built-in functions are internal to whichever environment, whereas third-party libraries lie externally and need to be imported to tackle specific contexts, such as Figure 3 (middle). Aligning with this conception, Zhang et al. (2023) use Python libraries such as matplotlib to plot figures and pandas to manage data.

### Utility functions

For more task-specific applications, expert-crafted utility functions, usually unseen at training time, are incorporated as tools. E.g., in Figure 3 (right), the highlighted `locate_objects` function is designed by human experts (Gupta & Kembhavi, 2022; Suris et al., 2023) to load neural models and perform post-processing to obtain the detected box region. In a similar spirit, Cheng et al. (2023) use GPT as a tool to query world facts external to the tabular contents, Cao et al. (2023) further design macro operation APIs to support advanced tabular operations. However, because human tool curation requires expertise and effort, some works explore using LMs to automatically create tools instead.

### 5.3 Tool creation and reuse

While one can readily use tools for tasks equipped with pre-designed tools, for tasks that do not have readily-applicable, human-created tools, some works explore using LMs to make tools and use them.

#### Domain-specific library abstraction

Works that use DSLs often compose frequently-used-together actions as shortcut tools. For example, Ellis et al. (2023) learn function abstractions such as `length` and `count` from lambda primitives (e.g., 0, +) for the list processing task. Wong et al. (2021); Bowers et al. (2023) similarly build functions bottom-up from a large corpus of DSL programs. More recently, Grand et al. (2023) use LLMs to abstract libraries with auto-documentation. Further for agentic tasks, Liu et al. (2018) learn common workflows to guide web navigation, such as composing the basic `{click, like}` actions to form a higher-level login action `click(like('login'))`.

#### General-purpose tool making

Nonetheless, on general-purpose PLs, running the DSL-oriented methods above may expand their search space and limit their scalability. Instead, recent works often leverage LMs’ procedural knowledge to alleviate the search issue. To start, Wang et al. (2023a) designs an automatic learning curriculum in Minecraft to make and use Java program tools. LATM (Cai et al., 2023) use LMs to build, verify, and use Python tools on BigBench tasks, where however, all examples require the same single tool hence have limited difficulty. CREATOR (Qian et al., 2023) extend tool-making to harder tasks such as math and table world problems, and improves task success by creating tools yet repetitively for individual examples, thus CRAFT (Yuan et al., 2023) add heuristic-based training to craft less repetitive tools. Towards more efficient pipelines, ReGAL (Stengel-Eskin et al., 2024) learns from refactoring a smaller number of programs, while TroVE (Wang et al., 2024b) purely relies on inference-time execution signal and induces reusable tools on-the-fly.

### 6 How to evaluate tool use?

In this section, we study existing LM-tooling benchmarks (§6.1) and their evaluation metrics (§6.2), lastly, we discuss the missing yet important evaluation aspects of tools (§6.3).

#### 6.1 Testbeds for evaluating tools

LM tool use can be evaluated on (i) repurposed existing datasets that can additionally benefit from tools (§6.1.1), and (ii) newly crafted benchmarks that necessitate tool use (§6.1.2).
6.1.1 Repurposed Existing Datasets

Many tasks are solvable by using LMs, yet often with great difficulty or inefficiency. Therefore, some works use tool-augmented LMs as an alternative approach to solve these tasks. Many of these datasets require reasoning. Starting from when questions are expressed in NL, such as complex reasoning with the Big-bench (bench authors, 2023) dataset, mathematical problems with the MATH (Hendrycks et al., 2021) dataset, and reasoning over world knowledge to answer questions in NaturalQuestions (Kwiatkowski et al., 2019) and TriviaQA (Joshi et al., 2017) datasets. Beyond free-form texts, datasets that require reasoning over structured data can also benefit from tools. These tasks include table-based QA with tabular math world problems in TabMWP (Lu et al., 2023b), Wikipedia tables in WTQ (Pasupat & Liang, 2015), and complex-structured tables in HiTab (Cheng et al., 2022). Beyond the text modality, datasets that require reasoning over other modalities also benefit from modality-extending tools, e.g., answering questions about an image with the GQA (Hudson & Manning, 2019) dataset, or image pairs with the NLVR2 dataset (Suhr et al., 2019).

Because tool use is proposed as an alternative method to solve these datasets, evaluations of these tool-augmented systems follow the standard evaluation process for individual datasets. Concretely, almost all tasks are measured by answer exact match, either in textual or numerical formats. Note that, to obtain the final answers for lexical matching evaluations, all tool-calling expressions need to be executed, and the execution outputs are incorporated into the final answers produced by the tool-augmented systems, as introduced in §3.

6.1.2 Aggregated API Benchmarks

Existing benchmarks can only benefit from a limited set of tools, yet there are far more tools we can utilize to perform versatile tasks in the real world, particularly the API tools created by human developers spread on the web. Therefore, many recent works aggregate API tools from various web sources and create benchmarks for using these APIs, as shown in Table 2.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Tool Source</th>
<th>Example Curation</th>
<th>Domain (§4.1)</th>
<th>Executable</th>
</tr>
</thead>
<tbody>
<tr>
<td>ToolBench</td>
<td>existing dataset adopted, human annotated</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>ToolBench2</td>
<td>RapidAPI</td>
<td>model synthesized</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>ToolQA</td>
<td>existing dataset model synthesized</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>ToolAlpaca</td>
<td>PublicAPIs</td>
<td>model synthesized</td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>API-Bank</td>
<td>PublicAPIs</td>
<td>human annotated</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>MetaTool</td>
<td>OpenAI Plugins model synthesized</td>
<td></td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Gorilla</td>
<td>HF, Torch, TF model synthesized</td>
<td></td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>HuggingGPT</td>
<td>HF</td>
<td>human annotated</td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>Task Bench</td>
<td>HF, PublicAPIs model synthesized</td>
<td></td>
<td>x</td>
<td></td>
</tr>
</tbody>
</table>

Table 2: Benchmarks of providing aggregated APIs to LMs as tools. HF is short for HuggingFace. ‘x’ means that: though tools employed by HuggingGPT are executable, it does not evaluate the execution output due to the cost of hosting and inferencing.

**Tool sources** Tools are mainly aggregated from existing datasets or public APIs. While Xu et al. (2023), Zhuang et al. (2023) adopt existing datasets and propose alternative methods via tool augmentation, these benchmarks are limited in domains. Several works scrape more APIs from online sources such as Public APIs (Tang et al., 2023), RESTful APIs (Tang et al., 2023), or the OpenAI plugin list (Huang et al., 2024). Beyond human-crafted APIs (Li et al., 2023), neural models from ML platforms can be similarly presented in an API format (Patil et al., 2023) Shen et al., 2023a,b. Nonetheless, as tools are collected from heterogeneous sources, it is challenging to select the best benchmark or unify all these varied benchmarks.

**Example curation** Examples can be adopted from existing datasets, annotated by humans, or synthesized by LMs. While most examples adopted from existing datasets are human annotated (Xu et al., 2023), only Li et al. (2023) do so for scraped APIs, by surveying 500 people and creating 314 dialogues manually. Most other works prompt GPT models to
synthesize examples (Qin et al., 2024; Tang et al., 2023; Shen et al., 2023b; Zhuang et al., 2023; Huang et al., 2024), however, leading to issues of naturalness and executability.

First, LMs are often asked to create examples, even tool outputs in Tang et al. (2023), given a heuristically selected set of tools. This approach leads to potential issues in two-fold: (i) the selected tools may not be used together in practice, and (ii) the synthesized examples may not reflect the natural use cases of these tools. Second, 5 out of 9 benchmarks in Table 2 do not support tool execution, to alleviate the cost of hosting multiple APIs, especially when they may fail or produce unstable outputs. For example, the weather returned by the check_weather API may change over time. This un-executability causes issues in evaluation. Instead of matching final execution results using lexical- (Li et al., 2023) or neural-based metrics (Tang et al., 2023; Qin et al., 2024), works with unexecutable tools resort to pseudo matching of API calling expressions with lexical (Tang et al., 2023; Shen et al., 2023a; Huang et al., 2024) and syntactical (Patil et al., 2023; Shen et al., 2023b) means.

6.2 What metrics are measured now?

**Task completion** Tools are used to assist task solving. Most works that allow tool execution evaluate the task completion score to quantify the effectiveness of utilizing tools.

**Tool selection** For datasets with execution issues (Huang et al., 2024; Shen et al., 2023b), another common metric is the accuracy of selecting the correct tools. This helps disentangle incorrect tool selection errors from inaccurate tool usage errors. Despite that tool selection mainly serves as a proxy for evaluating task completion when having unexecutable tools, it can be seen as a measure of LM planning abilities — the process of breaking down a task into multiple steps and selecting tools to complete individual steps.

**Tool reusability** While tool reusability is often deemed important in tool-making literature (Cai et al., 2023; Yuan et al., 2023), only Wang et al. (2024b) evaluates tool reusability by the size of induced toolboxes over a fixed number of examples. As its literal meaning, reusable tools can be (re)used to solve multiple examples hence having more generic functionalities. Adopting a reusable tool is more efficient than using multiple specific tools, and facilitates human verification in both speed and accuracy dimensions (Wang et al., 2024b).

6.3 What properties are missing?

**Efficiency of tool integration** As demonstrated by our empirical study (§7), the benefits brought by the tools come with the cost of additional computation, especially for teaching LMs to use tools via training or prompting. In addition to performance gain, reporting the computation overhead can enable fairer comparisons between different approaches.

**Quality of tools** While existing works mostly focus on how tools improve task accuracy, the performance of tools themselves is also important. Tool performance can cover multiple aspects such as completing the call quickly, requiring less computation, and not putting users at risk or failing unexpectedly. One way to measure these aspects is to conduct API testing (Yasar, 2022; Ehsan et al., 2022) on their runtime, memory usage, and success rate.

**Reliability of unstable tools** Particularly for tools that involve neural models or randomized components, their output quality may be unstable and unpredictable. For example, the VQA tool (Gupta & Kembhavi, 2022) may answer some questions correctly but others incorrectly. It is important to be aware of this uncertainty in contrast to stable, rule-based tools such as a calculator, further alleviating this instability and guarantee more predictable outputs.

**Reproducible testing** Many tools interact with the real world and may return different results at different times. For example, check_weather may return “sunny” today but “cloudy” tomorrow. This irreproducible behavior poses great challenges to creating static evaluation benchmarks with reference answers. While some works alleviate this by evaluating API calls without executing them, a more rigorous method could be parallel testing (Sharma et al., 2018) — executing the model-generated program and the reference program in parallel, and measuring if their final outputs match.
Most systems may only opt to use tools if they are trusted to be secure (Barbir et al., 2007). At the very least, users favor tools that can be easily understood and verified. Further, systems may need to enforce mutual authentication and ensure data integrity (Ehsan et al., 2022). Yet there are more security threats and methods beyond the discussion here. We encourage readers to peruse the referenced works above for thorough studies.

7 Trade-offs in tool usage

Leveraging tools often brings better performance, however, should we always use tools? More concretely, is the performance gain from using tools worthy of the computation cost spent for LMs to learn to use tools, or the inference cost at test time? Existing works mainly focus on task accuracy, but a more nuanced picture emerges when we take other factors into account. We empirically study the performance gain and learning cost of various methods on their experimented datasets in Table 3, using which we discover more efficient (i.e., achieve greater gains with less compute) methods and tasks that benefit more from tools.

<table>
<thead>
<tr>
<th>Type</th>
<th>Method</th>
<th>Task</th>
<th>∆ Perf.</th>
<th># Params (B)</th>
<th># Tokens (M)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>tool use</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ToolFormer</td>
<td>cloze</td>
<td></td>
<td>+ 14.7</td>
<td>6.7</td>
<td>642.1</td>
</tr>
<tr>
<td></td>
<td>math</td>
<td></td>
<td>+ 30.4</td>
<td>6.7</td>
<td>3864.2</td>
</tr>
<tr>
<td></td>
<td>QA</td>
<td></td>
<td>+ 5.8</td>
<td>6.7</td>
<td>1101.2</td>
</tr>
<tr>
<td></td>
<td>multilingual</td>
<td></td>
<td>- 0.2</td>
<td>6.7</td>
<td>606.0</td>
</tr>
<tr>
<td></td>
<td>temporal</td>
<td></td>
<td>+ 13.0</td>
<td>6.7</td>
<td>508.8</td>
</tr>
<tr>
<td>API-Bank</td>
<td>API</td>
<td></td>
<td>+ 24.4</td>
<td>7</td>
<td>190414.6</td>
</tr>
<tr>
<td>ToolAlpaca</td>
<td>API</td>
<td></td>
<td>+ 45.2</td>
<td>7</td>
<td>241889.3</td>
</tr>
<tr>
<td>Chameleon</td>
<td>science</td>
<td></td>
<td>+ 2.6</td>
<td>-</td>
<td>0.0</td>
</tr>
<tr>
<td></td>
<td>table</td>
<td></td>
<td>+ 1.9</td>
<td>-</td>
<td>0.0</td>
</tr>
<tr>
<td>LATM</td>
<td>BigBench</td>
<td></td>
<td>+ 29.1</td>
<td>-</td>
<td>28.5</td>
</tr>
<tr>
<td></td>
<td>math</td>
<td></td>
<td>+ 4.5</td>
<td>-</td>
<td>0.0</td>
</tr>
<tr>
<td></td>
<td>table</td>
<td></td>
<td>+ 0.0</td>
<td>-</td>
<td>0.0</td>
</tr>
<tr>
<td>CRAFT</td>
<td>math</td>
<td></td>
<td>+ 13.2</td>
<td>-</td>
<td>4126.6</td>
</tr>
<tr>
<td></td>
<td>table</td>
<td></td>
<td>+ 17.2</td>
<td>-</td>
<td>2750.6</td>
</tr>
<tr>
<td>TroVE</td>
<td>math</td>
<td></td>
<td>+ 21.0</td>
<td>-</td>
<td>0.0</td>
</tr>
<tr>
<td></td>
<td>table</td>
<td></td>
<td>+ 12.0</td>
<td>-</td>
<td>0.0</td>
</tr>
</tbody>
</table>

Table 3: Computation cost (number of tokens in M and parameters in B) of tooling methods and their performance gain on experimented datasets. To fairly compare costs on datasets with different sizes, we report the average number of tokens spent on a testing example.

For each work and each dataset they experimented with, we evaluate the performance gain after LM learned or made tools to solve tasks, compared to the baseline LM with no prior exposure to tool-related information. We also quantify the computation cost of their tooling approaches during the token-consuming training and inference processes. For works using models with known sizes, we report both (i) the number of tokens in input prompts and outputs, and (ii) the parameters in experimented models to achieve corresponding performance improvements. For methods using the size-unknown GPT-4 model, which are also comparable w.r.t. to model size since they use the same GPT-4 model, we only report the number of tokens processed. We elaborate more on computation details in §A.

**What tasks benefit the most from tools?** In general, tasks that cover multiple domains experience the highest increase, such as the ToolAlpaca benchmark in tool-using and the BigBench dataset in tool-making scenarios. Nonetheless, substantial gains may be expected.

---

2We did not measure some works due to insufficient resources.
on API benchmarks (i.e., API-Bank and ToolAlpaca), because all examples are synthesized use cases for designated tools (§4.1), no-tool baselines are deprived of necessary components (i.e., tools) to solve the task, therefore achieving much lower accuracy.

On existing benchmarks, the ToolFormer method is the most efficient on MATH problems, showing the highest 30.4 increase with little computation (0.17 MB). While other tasks improve less, multilingual tasks even degrade by ⋅0.2 points, despite using a similar amount of compute. This variance across tasks aligns with expectations: using a calculator tool greatly improves the arithmetic ability of probabilistic LMs, which are not naturally suitable for symbolic calculations; however, LMs are originally built to solve language tasks such as machine translation (MT), so assigning the MT task to another (usually smaller) LM may not bring substantial improvements.

What methods are efficient in tool-making? While it is hard to conduct fair comparisons for many works experimenting on different datasets, in tool-making scenarios (Figure 6), the three methods (Creator, CRAFT, TROVE) experiment on the same MATH and TabMWP datasets, thus enabling fair comparisons in both cost and performance dimensions. TROVE appears to be the most efficient method in general, costing only 1.2–1.4K tokens while improving the performance by 12.0–21.0 points in accuracy. In contrast, CREATOR and CRAFT are less efficient, costing 3.8–6.0 times of compute, yet achieve only minimal (0.0–4.5%) or comparable (4.1–5.0%) accuracy increases.

Training-time vs inference-time cost Training-time and inference-time costs may not be equally important to many practitioners, since inference may be run many times but training often only needs to be done once.3 If we only consider inference-time cost in the efficiency ranking of tooling methods changes. On one hand, tool-making method rankings roughly remain the same, except that CRAFT requires less compute than CREATOR on both tasks after getting rid of the training cost. On the other hand, however, the ranking among tool-using methods drastically changes: ToolFormer requires more compute than API-Bank and ToolAlpaca when considering only inference-time cost. We conjecture this is mainly due to differences in baseline setups: ToolFormer adds in-context examples than the CoT baseline, API-Bank and ToolAlpaca use the same prompt for baseline and fine-tuned LMs with varied abilities to utilize tools presented in the prompt. In general, if the user has sufficient budgets for training but higher demands on inference-time efficiency, the training approaches proposed by API-Bank and ToolAlpaca could be more suitable.

8 Final Remarks

Our survey provides definitions for LM-used tools and systematic summaries of existing approaches. While our empirical analysis guides when (on what tasks) and how (use what methods) should one use tools, we hope readers can more clearly understand the scenarios and techniques of LM tooling, from basic paradigm to advanced settings, and across LMs speaking natural and programming languages. We believe tools can greatly extend and facilitate LM abilities, and hope our work elicits more discussions and research developments in (i) proposing benchmarks with natural use cases and executable tools, (ii) utilizing comprehensive evaluation metrics proposed in §6, and (iii) exploring more challenging and realistic scenarios for tool-using and tool-making techniques.

3 Another measure of the inference process is latency, which also heavily depends on implementation or hardware choices. We do not report latency since these methods are implemented differently.
Acknowledgments

We thank Saujas Vaduguru, Sherry Tongshuang Wu, Jiawei Liu, Shihao Liang, Pengfei Liu for the helpful discussions. Zora Zhiruo Wang is supported by the Teng Family Presidential Fellowship. Hao Zhu is supported by NSF EAGER Award #2141751.

References


BIG bench authors. Beyond the imitation game: Quantifying and extrapolating the capabilities of language models. Transactions on Machine Learning Research, 2023. URL https://openreview.net/forum?id=uyTL5Bvosj


Mojtaba Komeili, Kurt Shuster, and Jason Weston. 


Angeliki Lazaridou, Elena Gribovskaya, Wojciech Stokowiec, and Nikolai Grigorev. 

Minghao Li, Yingxiu Zhao, Bowen Yu, Feifan Song, Hangyu Li, Haiyang Yu, Zhoujun Li, Fei Huang, and Yongbin Li. 

Jacky Liang, Wenlong Huang, Fei Xia, Peng Xu, Karol Hausman, Brian Ichter, Pete Florence, and Andy Zeng. 

Percy Liang. 

Evan Zheran Liu, Kelvin Guu, Panuponent Pasupat, and Percy Liang. 


Pan Lu, Liang Qiu, Kai-Wei Chang, Ying Nian Wu, Song-Chun Zhu, Tanmay Rajpurohit, Peter Clark, and Ashwin Kalyan. 

Rui Meng, Ye Liu, Shafiq Rayhan Joty, Caiming Xiong, Yingbo Zhou, and Semih Yavuz. 

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. 

OpenAI. 


A Detailed computation process for tooling trade-offs

For each method measured in §7, we describe the detailed processes in estimating their computation cost and performance improvement. For open-source models, we estimate cost $C = 6ND$, where $N$ is the number of tokens and $D$ is the parameter size (Figure 7, left). Because the parameter size $D$ of closed-source GPT is unknown, we only measure the number of extra tokens $N$ per example (Figure 7, right).

Figure 7: Computation cost of different approaches using open-source (left) and closed-source (right) models, and their performance gain on experimented datasets. We use different colors to represent tasks and different shapes to represent methods.

A.1 Methods using known-sized models

For methods using models whose parameter sizes are known, we estimate the computation cost by the FLOPs during any additional modules such as training and inference with additional context. In general, the computation cost is majorly affected by (1) the number of tokens processed, and (2) the parameter size of models.

API-Bank (Li et al., 2023) This work trains the Lynx model that uses tools to solve problems in the proposed API-Bank dataset. The Lynx model is initialized by Alpaca 7B parameters, and trained on the API-Bank training set with 3 epochs. Therefore, we adopt the Alpaca 7B as the baseline and Lynx as the tool-using model, where the 3-epoch training is the additional computation cost introduced to enable tool use. We calculate the total number of tokens involved in the training process, including the example i/o and additional instructions. Because the baseline and proposed method use the same prompt at inference time, no additional computation is required. Regarding task performance, we adopt the total correctness across all evaluation systems, as reported in Table 3. We report the difference between the fine-tuned Lynx-7B and the zero-shot Alpaca-7B.

ToolAlpaca (Tang et al., 2023) This work proposes the ToolAlpaca dataset and trains Vicuna models to use tools. The baseline models are Vicuna-7B and Vicuna-13B models. The trained tool-using models are called ToolAlpaca-7B and ToolAlpaca-13B models. All ToolAlpaca models are trained on the training split for 3 epochs, so we estimate the cost during this training process for 7B and 13B models, respectively. We adopt the ‘overall’ results reported in Table 3, on examples with both simulated tools and real-world APIs, and report their average results. We measure the performance gain by the difference between the ToolAlpaca-7/13B and Vicuna-7/13B.

Toolformer (Schick et al., 2023) This work integrates five tools — question answering system, calculator, Wikipedia search, machine translation system, and calendar — respectively for five tasks transformed from a subset of CCNet (Wenzek et al., 2020). Starting with GPT-J models (Wang & Komatsuzaki, 2021) as the no-tool baseline, they train on 25\textsuperscript{K} model-synthesized examples for each tool and obtain the Toolformer models, causing a total of 1M FLOPs for each task. At inference time, they add special instruction and in-context examples to prompt tool using, resulting in extra compute. Because each task contains multiple datasets, we report the average results to represent the general task performance.
A.2 Models with unknown size

While many of the works use GPT-3.5 or GPT-4 models that do not release their parameter size, we estimate the cost by using the number of tokens processed in extra modules.

**Chameleon** (Lu et al., 2023a) This work proposes to take a tool-augmented approach to improve on two existing datasets — ScienceQA and TabMWP. Because all experiments use ChatGPT and GPT-4 models, whose parameter sizes are unknown, we only examine results with (the better) GPT-4 model to fairly compare with other methods using GPT-4. Specifically for the ScienceQA dataset, we adopt the Chain-of-Thought (CoT) baseline reported in the paper, and report task accuracy as in the ALL column in Table 3. We calculate the difference in number of tokens between the proposed Chameleon methods against the CoT baseline. For the TabMWP dataset, we adopt the Program-of-Thought (PoT) baseline and similarly calculate the token number difference using the provided results.

**LATM** (Cai et al., 2023) This work proposes to use LMs to make tools for individual tasks in BigBench. Compared to the chain-of-thought (CoT) baseline, the proposed LATM method integrates training, validation, and inference stages to make tools and solve questions. We estimate the compute cost by the additional number of tokens used for LATM than for CoT. We measure each method by averaging its accuracy across all six selected tasks.

**CRAFT** (Yuan et al., 2023) This work uses LMs to make tools for math, table, and image reasoning tasks. We calculate the number of tokens used during training and inference, using its released code and data. CRAFT similarly implements CoT as the baseline, and proposes further training, verification, and finally testing in the CRAFT method. We report its task accuracy on the representative datasets from each task — MATH, TabMWP, and GQA — to enable fairer comparison with other works having overlapping datasets.

**CREATOR** (Qian et al., 2023) As a prior work for CRAFT, CREATOR similarly tests on MATH and table tasks, but designs its methods differently. In addition to CoT, this work implements a stronger program-oriented baseline called Program-of-Thought (PoT). We also adopt PoT as the main baseline without tool making or using. The CREATOR method operates at test time, with multiple steps through tool making, solution generation, verification, rectification, etc. We calculate the difference in number of tokens between the CREATOR approach and the baseline PoT setting. We adopt the task accuracy reported in Table 2 (MATH) and Table 3 (TabMWP) from the original paper.

**TroVE** (Wang et al., 2024b) TroVE also induces tools without training supervision. This work adopts the primitive baseline, a presumably stronger version of PoT yet without much textual explanation. The main implementation change in TroVE is the three-mode generation and multi-candidate sampling. We calculate the additional tokens used in TroVE compared to the primitive baseline. The dataset reports task accuracy, solution complexity, and toolbox size, we only adopt the task accuracy to fairly compare with other works.

---

4 https://github.com/lupantech/chameleon-llm
5 https://github.com/lifan-yuan/CRAFT