



Thinking Forwards, Backwards, and in Code:

Improving Tool Acquisition in Large Language Models

Author:

Huey Sun¹

Supervisors:

Dr. Pontus Stenetorp

Yihong Chen

MSc Computational Statistics and Machine Learning

September 23, 2024

¹**Disclaimer:** This report is submitted as part requirement for the MSc in Computational Statistics and Machine Learning (CSML) at University College London. It is substantially the result of my own work except where explicitly indicated in the text. The report may be freely copied and distributed provided the source is explicitly acknowledged.

Abstract

Tool-augmented large language models are rising to prominence for their deft and dynamic ability to handle complex multi-step problems in real-world settings. While tools can be incredibly powerful and extend a model’s capability in non-parametric ways, the literature is divided on how to teach a model to acquire tools, and what a tool even is. All the while, recent studies show that the state of the art (SoTA) in open source is falling far behind proprietary tool-augmented LLMs. By doing a deep dive into the data, we uncover crucial flaws in public tool-acquisition corpuses that incite hallucination in tool-augmented models. We introduce “backwards thinking” and “code thinking”, two reasoning strategies that can be applied to small, diverse data sets to facilitate SoTA performance with a fraction of the training. Our best performing model achieves an average improvement of 17% against the top open-source model of the same size, and represents a 45% improvement from the base instruction-tuned model. Our experiments reveal compelling insights about the far-reaching implications of data augmentations such as name scrambling and its effects on robustness and hallucination reduction in tool calling.

Contents

1 Anatomy of a Tool	2
1.1 Contributions	3
1.2 Overview	4
2 Into The Literature	5
2.1 So What Exactly is a Tool?	5
2.1.1 Integrated Tools	5
2.1.2 Tool Frameworks	8
2.1.3 Tool Benchmarks	14
2.2 The Path Ahead	19
3 Identifying Failure Modes in Existing Tool Acquisition Methods	21
3.1 About Our Models	21
3.2 Reproducing the Benchmark	22
3.3 Categorizing Weaknesses	24
3.4 Experiments with Prompting	25
3.5 Experiments with Fine-tuning	26
4 A New Method for Tool Acquisition from Synthetic Datasets	28
4.1 Styles of Thinking	28
4.1.1 Thinking Forwards	30
4.1.2 Thinking Backwards	31
4.1.3 Thinking in Code	32
4.2 Building the Dataset	33
4.2.1 On Generalizability	33
4.2.2 On Planning	34
4.2.3 Data Ablations	35

4.3	Training	35
4.3.1	Tokenization	35
5	Results: A New SoTA for Open-Source Tool-Augmented LLMs	36
5.1	Base Results	36
5.1.1	Further Experimental Results	37
6	Limitations and Future Work	40
6.1	Limitations	40
6.2	Future Work	41
A	Additional Figures	42

Chapter 1

Anatomy of a Tool

From a stick in the hand of a chimpanzee [15], a rock in the paw of an otter [28], or a keyboard in the lap of a sleep-starved graduate student, using tools – whether to pry our way to a tasty snack or pen an introduction for a thesis – is a crucial way of extending our inherent capabilities and is viewed as a key indicator of emergent intelligence [11]. From the wheel to the web browser, tools have permeated every facet of work and society, allowing for the efficient and effective use of our resources. As tools evolve in sophistication, they often become indispensable for continued progress, and have led to the creation of technologies that are shaping the world in ways otherwise unimaginable.

As large language models (LLMs) grow increasingly adept at modeling natural language across a variety of tasks and concepts, augmenting models with external tools has become a new frontier of innovation. Models have been equipped to use calculators to tackle tough mathematical questions [3], check and cite Wikipedia [16], query knowledge stores [12], and interface between systems as autonomous agents [31]. These abilities give LLM’s non-parametric extensions to their model capability and have been developed in response to inherent limitations in fixed language models, such as the need for up-to-date information retrieval, advanced computation, and the ability to interact with the external world. Each of these examples of tool use has a different mechanism under the hood with a competing philosophy for how we should think of, implement, and train effective tool-augmented models.

While the extensibility of a single tool is limited to its application and properties, a model with the ability to access arbitrary or user-defined tools is instead constrained by its ability to recognize appropriate contexts for tool use, correctly execute tool calls, and properly integrate their responses. Models that are effective and flexible tool users can

adapt to a variety of tasks and domains, particularly in cases where updating a tool with new behaviour is simpler than individually fine-tuning the model. Beyond ability, tool use can also provide key insight for model explainability in applications where transparency is paramount, as each tool call provides an easily-accessible glimpse into the model’s internal reasoning.

However, consistency and adaptability are hard to train. In a recent study, frontier state of the art models were placed into a sandbox environment with all the required tools and prompted to perform tasks such as “set a reminder for 6PM next week” [14]. In their experiments, GPT-4o, Claude-3-Opus, and Gemini-1.5-Pro each had their struggles with different aspects of tool use such as hallucinating timestamps or failing to disambiguate arguments. Open-source models such as Mistral-7B-Instruct or Hermes lagged significantly behind, struggling to chain together tools, obtain local state, perform conversions, navigate errors, and otherwise execute on single and multi-step tool-use tasks. From the literature, leading tool-augmented models such as ToolLLM [22] or Gorilla [19] eluded evaluation as they either lacked the mechanism to specify input tools or had syntactic constraints that rendered them incapable of agentic behavior.

While at first glance, differences in syntax may seem superficial or interchangeable, they belie disparate philosophies for how we should define and evaluate tools. Despite the increasing number of proposed algorithms for tool acquisition, there is yet to be an authoritative benchmark or schema to unify and compare the body of work as a whole. And while richness in the current landscape of tool-augmented models is a strength, consolidation is required not only to measure progress but also to ensure that the advances and methods proposed are not subject to other critical but unexamined failings.

1.1 Contributions

In light of these challenges, our work makes the following contributions:

- First, we conduct an assessment of the major tool-augmented models in the literature. While previous surveys have categorized models based on implementation or algorithm, we focus on data and its effect on downstream performance. Our analysis reveals that many of the objectives used to train these models are highly specialized and do not generalize well for real-world application.

- Using ToolSandbox, we identify 5 critical failure scenarios in Mistral-7B-v0.3, Mistral-7B-Instruct-v0.3 and Hermes, and categorize the root cause of each shortcoming. We also identify key hallucinations in publicly available tool-calling datasets.
- We introduce two novel twists on chain-of-thought reasoning for tool use, “backward” thinking and “code” thinking, and compare models finetuned with the following approaches:
 - Forward thinking: Classic chain-of-thought reasoning, a goal-oriented approach where the model plans tool calls step by step from start to finish.
 - Backward thinking: An argument-oriented approach where the model reasons backwards from the desired outcome to the requisite parameters.
 - Code thinking: A code-oriented approach where the model plans in python.
- We outperform the current open-source SoTA at 7B by 17% with 3% of the publicly available tool data. We release our fine-tuning data¹ and provide example-driven insights, libraries for tool tuning, chat templates, and further analysis on developing robust fine-tuning data.

Through these contributions, we advocate for a foundation of robustness and quality in the available tool acquisition data. By exploring the limitations of current benchmarks and models, we hope to forge common ground and champion practical application as a key consideration in the development of tool-augmented models.

1.2 Overview

The remainder of the paper will be structured as follows. In Chapter 2, we will look at how tools have been defined in the literature and explore limitations in evaluating the current models from academia. In Chapter 3, we will explore specific scenarios that break open-source tool-calling models such as Mistral-7B-Instruct-v0.3 and shortcomings in publicly available tool-calling data. In Chapter 4, we will present our methodology and formally introduce forward thinking, backward thinking, and code thinking. We will present our results and analysis in Chapter 5 and discuss limitations and future work in Chapter 6.

¹All available here: <https://github.com/itshuey/tool-acquisition>

Chapter 2

Into The Literature

2.1 So What Exactly is a Tool?

The literature on tool-augmented LLMs can be loosely organized into three sections: integrated tools, tool frameworks, and tool benchmarks. While implementations vary, we will broadly consider any controllable, non-parametric extension of model capability to be a tool. This work focuses specifically on tool frameworks that accept user-defined tools in a functional schema. While these concepts are difficult to rigorously define due to exceptions in each rule, we find that they are intuitive and easy to understand through concrete examples. Thus, we will begin this section by providing an overarching background of integrated tools before discussing state of the art tool-augmented models and their limitations.

While LLM tool use is quite young as a field, its explosive growth has given rise to survey works that vary in focus from specific applications in autonomous agents [31] and biochemistry [24] to the field’s origins [21], the workflow of developing a tool-augmented LLM [23], and how tool augmentation can affect downstream tasks [9]. Rather than present a comprehensive account of all unique tool-augmented LLMs, our aim is to develop an opinionated overview of the most influential models and guide how we should think about tools and where we should go next.

2.1.1 Integrated Tools

Integrated tools like Retrieval-Augmented Generation (RAG) are the most established form of tool use in LLMs [12]. While large language models develop incredible ability to understand and generate language through their vast and varied training corpuses,

they remain fundamentally constrained by the information content in the data they are trained on. When asked about subjects which require niche or up-to-date knowledge not present in the training examples, these models will often hallucinate, producing nonfactual or nonsensical information based on the linguistic properties of the context [34]. While LLMs contain impressive ability to generalize within its parameters and can follow nuanced instructions, only further training can equip it with new knowledge.

To address this, RAG gives language models access to an external knowledge store, allowing them integrate outside information into the generation process instead of depending solely on their language modeling capability [12]. This provides LLMs a non-parametric mechanism to improve the factuality and relevance of its generations. Concretely, the external knowledge store is comprised of Wikipedia articles embedded as dense vectors. At inference-time, the top-k (in their implementation, 5 or 10) relevant documents z for a given prompt x are selected by a neural retriever with probability $p_\eta(z|x)$ and appended to the context for the generator $p_\theta(y|x, z)$. RAG then marginalizes these probabilities $p_\theta(y|x, z)$ over the latent documents z to form its response. As $p_\theta(y|x, z)$ is an autoregressive generator for the individual tokens y_i , we have:

$$p_{\text{RAG}}(y|x) \approx \sum_{z \in \text{top-k}} p_\eta(z|x) p_\theta(y|x, z) = \sum_{z \in \text{top-k}} p_\eta(z|x) \prod_i^N p_\theta(y_i|x, z, y_{1:i-1})$$

Alternatively, the top-k most relevant documents can be re-drawn for each token in the generation, allowing the provided information to shift with the model’s generation. In this paradigm, RAG marginalizes the per-token probabilities $p_\theta(y_i|x, z, y_{1:i-1})$ over the documents z :

$$p_{\text{RAG}}(y|x) \approx \prod_i^N \sum_{z \in \text{top-k}} p_\eta(z|x) p_\theta(y_i|x, z, y_{1:i-1})$$

During training, the documents drawn by the retriever are not supervised, and the document encoder used to convert the Wikipedia articles into vectors stays fixed. The generator and retriever (specifically, the query encoder) are trained with labelled examples (x, y) and the standard negative marginal log-likelihood. While the model doesn’t choose to retrieve documents discriminantly (the process happens automatically for every prompt), we can still classify the RAG-style mechanism as an example of an integrated tool as the tool is fully defined within the model, its usage requires training, and its functionality is fixed – while these LLMs may need to generalize to different kinds of documents, they do

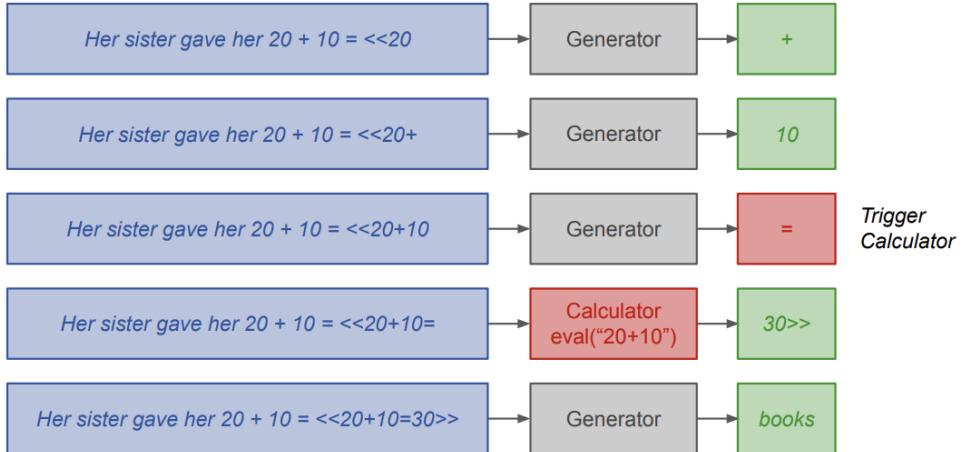


Figure 2.1: Per-token sampling procedure for the calculator tool, from [3]

not need to acquire new tools or mechanisms. While this work has inspired a family of related augmentations that introduce triggers, vary the retrieval mechanism, or incorporate documents in different ways [4], we can broadly categorize this technique as an integrated tool.

Integrated tools can also be operated using a model’s output tokens. On GSM8K, a dataset of grade school math word problems, both the 6B and the 175B models in the GPT-3 family [1] would make errors in routine calculations [3]. To address this, the models were equipped with a simple calculator tool and trained with annotated tool calls in the form `<<5+6=11>>` injected into the training set. During inference, when the model required a calculation, it would first generate the `<<` tag with the equation followed by the equals sign, which served as the trigger (see Figure 2.1). The tool would then check if the annotation was well-formed and evaluate the expression with python’s `eval` if so, returning the response and the end tag `>>`, after which the generator would continue on as usual. The `<<` and `>>` tags were all ordinary tokens, and if a tool call timed out it would be discarded and the model would default to ordinary sampling.

Similar efforts have given LLM’s access to python interpreters [18] and web browsers [16], which involves training with both annotated examples of human demonstrations and preference data. While these tools are incredibly dynamic, they still fall under the general category of integrated tools as their non-parametric capability comes from a fixed mechanism that they are explicitly trained to trigger and utilize.

2.1.2 Tool Frameworks

Tool frameworks, on the other hand, augment models with the ability to use an assortment of dynamic or user-provided tools, emphasizing generalisability to unseen tools. Tool frameworks can be tuning-free and rely solely on the few-shot capabilities of foundation models to elicit tool use. ReAct (Reason and Act) combines chain-of-thought (CoT) inference with specified action sets to give frozen pre-trained models a framework for interacting with different environments and tools [35]. For knowledge retrieval and reasoning tasks in the HotpotQA dataset [33], this action set includes tools like `search[entity]`, which returns the top 5 most relevant entities from Wikipedia, `lookup[string]`, which returns the next sentence in the page with `string` (the equivalent of Ctrl+F search), and `finish[answer]`, which outputs the final response.

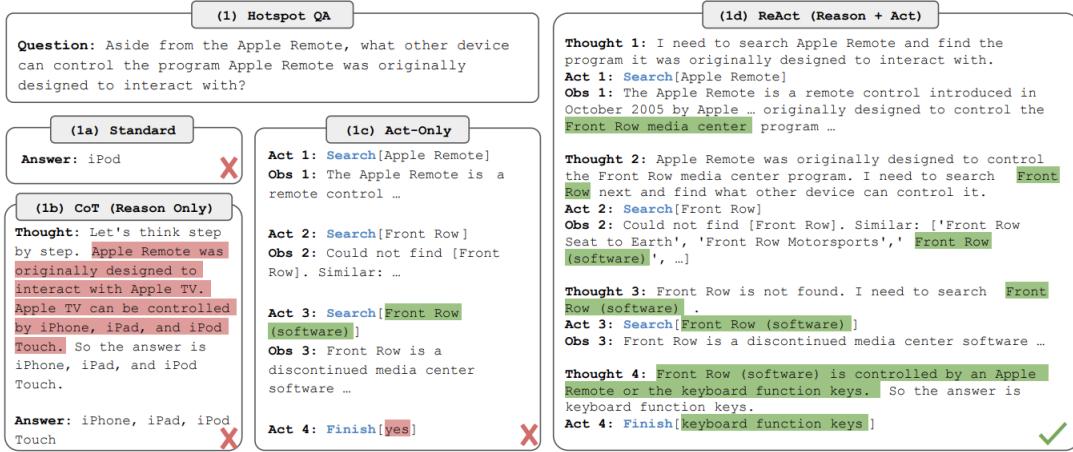


Figure 2.2: Comparing Standard, CoT, Act-Only, and ReACT responses on a HotpotQA prompt, from [35]. Only ReACT gets the correct answer “keyboard function keys”.

While both traditional chain-of-thought and ReACT prompt their models to perform intermediate reasoning with few-shot examples [32], ReACT interleaves these reasoning traces with actions, allowing the model a non-parametric mechanism in use in its planning (see Figure 2.2). While this method significantly outperformed naive and Act-Only prompting, it was edged out by CoT on HotPotQA: while ReACT reduced hallucination in its intermediate reasoning, a critical thorn in the side of CoT, it also became dependent on successful retrieval of useful knowledge and became less flexible in its reasoning. Interestingly, fine-tuning PaLM-62B on ReACT-annotated data yielded far greater returns on HotPotQA than fine-tuning on standard or CoT methods at the same model size, and even outperformed prompt-based ReACT with PaLM-540B. This seems to imply that few-shot

examples alone may not be sufficient for a model to dynamically integrate tool use into its planning and generation.

Furthermore, while prompt-level frameworks hypothetically allow models to abstract beyond tools in their training data, each tool supported by the framework requires examples of the context and syntax involved in their use, which can quickly blow up if the tool set is large or contains interactions that require complex planning. Beyond context-length restraints, it also places a burden on the prompter to develop reasoning traces for the required tools, which can be prohibitively intrusive for user-facing applications. While fine-tuning shows promise even at smaller model size, there aren’t any takeaways on downstream generalizability as their model was trained and evaluated on the same task with just the three aforementioned tools.

An example of cross-domain tool-augmentation is Toolformer, a model equipped with a question answering system, a machine translation system, and a calculator [26]. Tools are represented in the training data as $\langle \text{API} \rangle \text{tool}(\text{input}) \rightarrow \text{output} \langle / \text{API} \rangle$, where $\langle \text{API} \rangle$ is a designated token for tool calls and `tool` is the name of the specified tool. During inference, decoding occurs as normal until the model produces the \rightarrow token, after which the execution environment parses the tool name and input parameters, makes the tool call, and returns the output with the closing $\langle / \text{API} \rangle$ token. Inference then continues as normal with the added context from the specified tool.

Toolformer is agnostic to the implementation of the tools in the execution environment – they can be scripts, another neural network, or a hard-coded lookup table – but they must be implemented manually and individually connected to the model. To learn to use these tools, Toolformer constructs its training data in a self-supervised manner to circumvent the burden of collecting extensive human annotation. To do this, it takes ground-truth statements and generates tool annotations to insert into the text. After receiving tool responses from the environment, it keeps only the examples where the tool call increases the chance of predicting the correct tokens (see Figure 2.1 for concrete examples). Specifically, for a given model M , they define $p_M(x_j | z, x_{1:j-1})$ as the probability M assigns to token x_j given the previous sequence of tokens x_1, \dots, x_{j-1} and a prefix z . For weights w_i , the weighted cross entropy loss over the tokens x_i, \dots, x_j is defined as:

$$L_i(z) = - \sum_{j=i}^n w_{j-i} \cdot \log p_M(x_j | z, x_{1:j-1})$$

To determine the usefulness of a tool call, L_i^+ is defined as the loss when the tokens are

Example	$L_i^- - L_i^+$	Useful
The Flodden Window (a war memorial dedicated to The Middleton Archers), in the Grade I-listed Church of St Leonard in Middleton is said to be the oldest war memorial in the United Kingdom. <API> WikiSearch(War memorial Flodden) → Battle of Flodden > Commemoration > The stained-glass Flodden Window in Middleton Parish Church [...] was constructed by Sir Richard Assheton in memory of the Battle of Flodden and the archers from Middleton who fought in it. </API> Sir Richard Assheton of Middleton (who built St Leonard) was granted knighthood [...]	5.49	✓
Note: The WL will be open on Friday, <API> Calendar() → Today is Thursday, March 9, 2017. </API> March 10, and Sunday, March 19 for regular hours.	2.11	✓
The Nile has an approximate length of <API> QA(What is the approximate length of the Nile?) → 6,853 km </API> 6,853 kilometers, the White Nile being its main source.	2.08	✓
85 patients (23%) were hospitalised alive and admitted to a hospital ward. Of them, <API> Calculator(85 / 23) → 3.70 </API> 65% had a cardiac aetiology [...]	-0.02	✗
But hey, after the <API> Calendar() → Today is Saturday, June 25, 2011. </API> Disneyland fiasco with the fire drill, I think it's safe to say Chewey won't let anyone die in a fire.	-0.41	✗
The last time I was with <API> QA(Who was last time I was with?) → The Last Time </API> him I asked what he likes about me and he said he would tell me one day.	-1.23	✗

Table 2.1: Toolformer tool call examples ranked by usefulness, modified from [26]

prefixed with both the tool call c_i and response r_i , and L_i^- is the minimum of the losses when nothing (ε) is provided as a prefix and when c_i is provided without r_i . Keep in mind that this procedure is used to select generated annotations to use as training data, so M is unfamiliar with tool call notation and cannot handle it naturally in place, hence the prefixing. We can summarize this as:

$$L_i^+ = L_i(c_i, r_i)$$

$$L_i^- = \min(L_i(\varepsilon), L_i(c_i))$$

where only tool calls that provide a benefit beyond a threshold τ_f are kept:

$$L_i^- - L_i^+ \geq \tau_f$$

With Toolformer’s approach, a tool-augmented GPT-J with 6.7B parameters outperforms GPT-3 (175B) on a variety of question answering and mathematical reasoning benchmarks. This vastly reduces the effort for tool acquisition by removing the costly burden of human data annotation. However, it still requires re-training for each new tool, which is burdensome for users.

To address this, ToolkenGPT uses in-context learning to minimize the burden of additional fine-tuning [7]. In this framework, individual tools are represented as unique tokens.

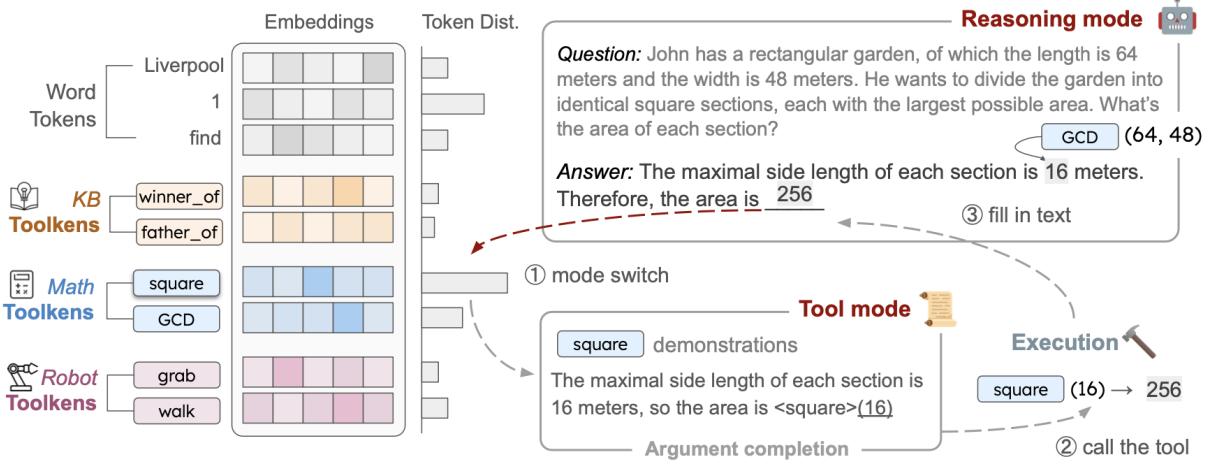


Figure 2.3: An example of tool calling from ToolkenGPT [7]. The model is set to reasoning mode until a `<toolken>` is predicted, after which it uses in-context learning to fill in the required parameters and complete the tool call.

For example, a tool that squares an input number would be represented by a corresponding token `<square>`, while a tool that gets a country’s capital would have a corresponding token `<capital>`. When one of these `<toolkens>` is predicted, the model switches modes to figure out the parameters of the specified tool from the current context. To do this, a separate prompt is created with the contents of the original prompt and several tool-specific examples for how the tool is meant to be used, with solely to extract relevant parameters for the chosen tool. Once these parameters are generated, the tool is executed and its response is returned to the original context to replace the `<toolken>`, where inference then proceeds as normal (see Figure 2.3).

With this approach, training ToolkenGPT to acquire new tools only requires the model to predict the specified tools in the correct context, which can be done by training the embedding layer with a frozen transformer body. Similar to previous work, it uses ground truth tool data from datasets that involve calculations or fact-based Question-Answer as well as synthesized annotations. When evaluated on GSM8K-XL, the previously mentioned grade school-level math word problem benchmark, and FuncQA, their synthetic benchmark requiring more challenging operations like exponentiation or lowest common denominator, their LLaMA-33B model [29] outperforms ReACT and 0-shot ChatGPT.

Compared to Toolformer, where the model must predict the correct tool *and parameters* in place during inference, ToolkenGPT’s use of in-context learning allows the model to skirt dedicated training for parameter extraction for each of its tools. As opposed

to fully in-context methods, which require demonstrations for each tool in conversation, ToolkenGPT’s compartmentalization of the parameter extraction process and use of a separate prompt reduces demands on context-window size and separates the recognition and completion aspects of tool use. However, while re-training the embedding layer is much less friction-less than re-training the model, this approach still may be prohibitively inflexible from a user perspective.

One way to broach the generalization problem without requiring additional fine-tuning is to train models with a large and diverse set of tools. To achieve this, ToolBench collects over 16,000 APIs across 49 categories from RapidAPI, an online platform for users to host or subscribe to real-world APIs, forming a comprehensive data set for consumer API use [6]. While RapidAPI typically requires a manual subscription process to access each API on its platform, ToolBench hosts a free server that interfaces between users and RapidAPI, allowing them to access and perform inference with their entire set of APIs. As such, ToolBench has gained significant influence in the tool-augmented LLM space and has led to the development of several tool frameworks, including their own tool framework ToolLLM.

Inspired by a key shortcoming in ReACT, where mistakes made by tool-augmented LLM’s in one step of reasoning propagates into further error, the ToolLLM framework [22] uses a depth-first search-based decision tree (DFSDT) to plan its tool use, allowing for multiple reasoning traces that explore the computation space. ToolLLM is comprised of ToolBench, a synthetic dataset of single and multi-tool API calls, ToolLLaMA, their trained model, and ToolEval, a ChatGPT-based evaluator. To generate ToolBench, random APIs are sampled and ChatGPT is prompted with the following excerpt, which has been edited for length:

“ [...] Your task involves creating 10 varied, innovative, and detailed user queries that employ API functions of multiple tools. For instance, given three tools ‘nba news’, ‘cat-facts’, and ‘hotels’: ‘nba news’ has API functions ‘Get individual NBA source news’, ‘cat-facts’ has API functions ‘Get a random fact about cats’, ‘hotels’ has API functions ‘properties/get-details (Deprecated)’ and ‘locations/v3/search’. Your query should articulate something akin to: ‘I want to name my newborn cat after Kobe and host a party to celebrate its birth. Get me some cat facts and NBA news to gather inspirations for the cat name. Also, find a proper hotel around my house in Houston Downtown for the party.’ [...] Additionally, you must incorporate the input parameters required for each API call. [...] Don’t refer to ‘my company’, invent a company name instead. [...] The final three queries should be complex and lengthy, describing a complicated scenario where all the provided API calls can be utilized to provide assistance within a single query [...]” (from [22])

The eclectic and diverse training examples result in a model that is incredibly capable at creatively connecting disparate topics to solve a multi-tool goals. During evaluation, like RAG, a neural retriever selects the most relevant tools from the collection of APIs to provide the model. However, unlike RAG (and Toolformer), the models are evaluated on ToolBench with ToolEval in lieu of a traditional NLP task such as question answering or world knowledge. ToolEval grades responses based on pass-rate, which measures whether the instruction was successfully completed or correctly deemed unsolvable, and win-rate, which judges the qualitative performance of a response against another based on information richness, factuality, exploration, cost, and other criteria. ToolLLaMA (derived from LLaMA-2 7B) narrowly edges out ChatGPT with DFSDT but loses out to GPT4 with DFDST, demonstrating its capability despite a significant parameter disadvantage.

ToolLLM represents a break in the mold against traditional tool-acquisition in LLMs. While previously, tools were primarily considered for their utility in improving performance on canonical tasks and benchmarks, ToolLLM evaluates a model’s ability to connect disjointed ideas, generalize to unseen APIs, and plan complex multi-step tool calls. While this dimension of tool-augmented models is certainly important to measure and improve, their approach also has critical flaws. Due to the data’s generation strategy, tasks in Toolbench can be artificially contrived and unrealistic. Prompts often contain all the requisite parameters for the tools and may even name an API explicitly, which a real user is unlikely to do. ToolLlaMA runs DFDST and makes tool calls regardless of whether it needs to, which adds around 15-20 seconds to each use. The tool responses from RapidAPI also vary drastically in quality, which, while realistic, results in training examples of subpar quality. ToolEval’s win rate is also difficult to interpret, as its ground truth is often subjective: the ToolLLM team found that human judges scored responses in very different ways, and even certain criteria such as information-richness can be debated in different contexts. However, fundamentally, tool calls in this paradigm are no longer non-parametric extensions in response to specific shortcomings in the model, such as up-to-date information retrieval or calculation, but rather the default form of information gathering that the model is then used to glue into a coherent response, of which the user has no control.

Nonetheless, ToolLLM has been prolific, sparking a host of development in algorithmic tool acquisition. α -UMi decomposes tool use in a multi-agent fashion with three separate objectives: planning tool calls, executing tool calls, and summarizing their outputs [27]. Starting from LLaMA-2-chat-7B and 13B, α -UMi’s finetunes the model backbone together on ToolBench before specializing into their respective tasks, yielding an overall framework

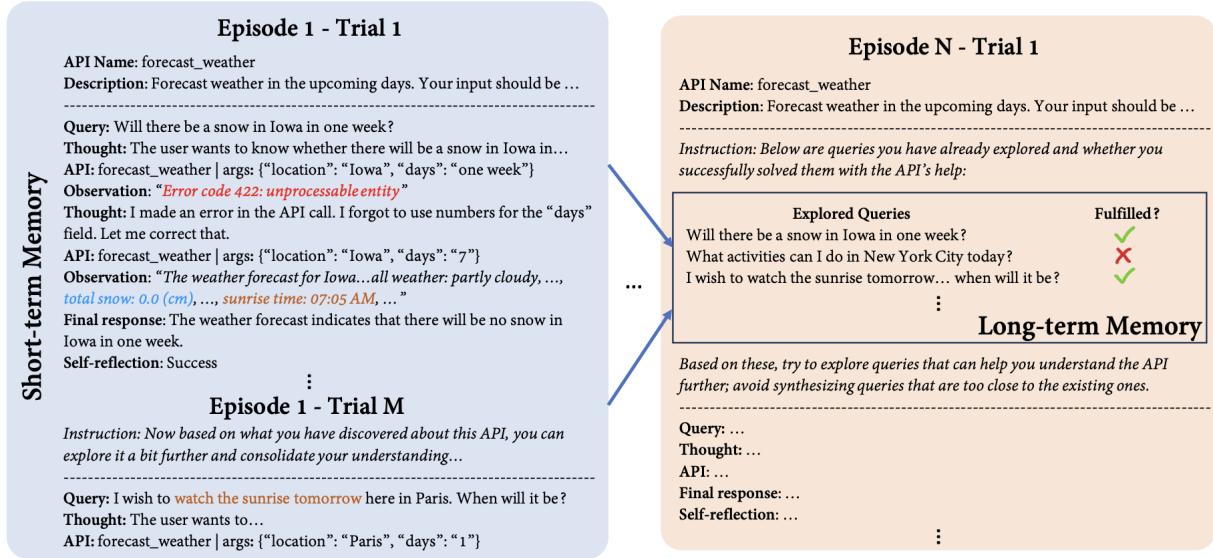


Figure 2.4: Exploration on a weather forecasting API with simulated trial and error showing the development of short and long term memory, from [30].

that outperforms ToolLLaMA. Other approaches augment models with memory. In simulated trial and error (STE) [30], a LLM first imagines potential scenarios for which an API from ToolBench can be applied, calls the API, and then reflects on the trial, storing recent trial and error trajectories in its short term memory and successful fulfillment of the scenario in its long-term memory (see Figure 2.4). Through this exploration, the models derived from Mistral-Instruct-7B are able to make well-formed and correct calls more frequently than ToolLLaMA and even GPT4.

There are also many tool frameworks that do not rely on ToolLLM. ToolDeer carefully considers whether tool calls are required before potential use [5], Toolink creates its own toolkit for reasoning tasks in code [20], and Gorilla converts natural language prompts into structured outputs, allowing for integration into workflows as a reasoning cog [19]. But for users with tools seeking to extend a language models abilities, potentially into the real world, academia doesn't yet have a good solution.

2.1.3 Tool Benchmarks

As we hope to have motivated, meaningful benchmarks in tool acquisition don't just play a critical role in comparing models – they also serve to define desirable traits and behavior in a domain with no dearth of competing ideology and implementations. Beyond

Query	Answer
I need to remember to attend a meeting at 10 a.m. Can you set a reminder for me?	No, it is not necessary to use external tools for this query. As an intelligent agent, I have the capability to set reminders for users and can do so without the need for external tools.
Create a fun math question for children.	Yes, it is necessary to use external tools. The query requires the creation of a fun math question for children, which involves generating a specific type of content that may not fall within the capabilities of the language model. External tools or resources may be needed to generate appropriate and engaging math questions that are suitable for children.

Table 2.2: Examples of model overconfidence (from ChatGPT) and lack of confidence (from Vicuna) surrounding tool use capacity, modified from [8].

task-level performance, benchmarks have been introduced to evaluate how well models understand the tools they are calling. One such study called Rotbench tests the robustness of tool-calling models by scrambling function names with varying levels of noise, showing significant room for improvement in both closed-source LLMs like GPT-4 and open-source models like ToolLLaMA alike to recognize functions by their descriptions and parameters [36]. Another benchmark, MetaTool, tests whether models understand the limits of their abilities and the contexts for when they should call tools [8]. By sampling ChatGPT plugins and various benchmarks to create positive and negative examples of tool use, MetaTool creates a diverse landscape of tasks and discovers that LLMs including ChatGPT and Llama-2-13B can exhibit overconfidence and lack of confidence alike, and have inconsistent perceptions of their own capability depending on the the type of mathematical or logical query at hand.

However, both benchmarks use prompting that deviate from the conditions under which the models were trained, which naturally leads to digression in performance. RotBench uses in-context ReACT formatting during inference to evaluate GPT, which is distinct from modern tool-calling syntax, while MetaTool simply asks in natural language whether certain tasks require tools wthout providing any tool information (see Figure 2.2). While these are both desirable metrics to evaluate the generalization and awareness of LLMs, they don't entirely capture or target a model's tool use capability.

In contrast, benchmarks like API-Bank directly assess and evaluate model performance on downstream tasks by implementing 73 unique APIs within its benchmarking framework [13]. Depending on the task, models are graded on their ability to call single-tool and multi-

tool tasks with retrieval and planning. However, evaluation is done with strict synthetic trajectories, where 1) predicted API calls are compared against annotated API calls to determine their consistency, and 2) responses are graded by ROGUE-L, which looks at the longest common subsequence between the model’s output and the expected output. While ToolEval suffered from lack of interpretability and subjectiveness, API-Bank suffers from strictness, penalizing models that may correctly execute the given task but deviate from the expected trajectory or response style.

In response, ToolSandbox was created as an interactive, stateful, robust, and human-authored benchmark for LLMs [14]. In ToolSandbox, user models (either ChatGPT or GPT4) are given task-specific context and placed in conversation with agent models to evaluate their tool-calling ability. While the initial prompt is defined for each test case, the remainder of the trajectory is flexible and robust to different solution paths, with the session concluding when the max conversation length is reached or when the user agent determines the task has been completed. At the beginning of each scenario, the model is provided with the designated functions, and at each turn, the framework evaluates whether or not the model responded with a tool call, which requires parsing a structured syntax that varies by model (for Mistral, its the token `[TOOL_CALLS]`). If a tool call is detected, ToolSandbox will extract the arguments and attempt to execute tool calls, providing the response from the execution environment before continuing the simulation. This aligns with the standard pattern of interaction for tool-calling in user-facing language models.

To evaluate, conversations include milestones and minefields that are designed specifically for the given scenario. Milestones are tool calls that need to occur for a model to successfully complete the task. For example, for the task “send a message to Fredrik Thordendal asking about the new album” and a simulation environment where the cellular option is initialized off, milestones include 1) searching contacts with the correct name to retrieve the phone number, 2) turning on the cellular service to enable messaging, 3) sending a message with the correct number and message content, which 4) results in a database entry in the simulated message database (see Figure 2.5). The evaluation framework is not dogmatic about the exact computation path – the agent can turn on cellular before searching contacts, it can explicitly check the cellular status and enable it or recover from an error message, and it can experiment with different tool calls and probe the user for more information in cases where insufficient information is provided.

Concretely, we can consider the set of milestones M_+ for a given task as a directed acyclic graph $G_{M_+}(V_{M_+}, E_{M_+})$, where the vertices correspond to the individual milestone

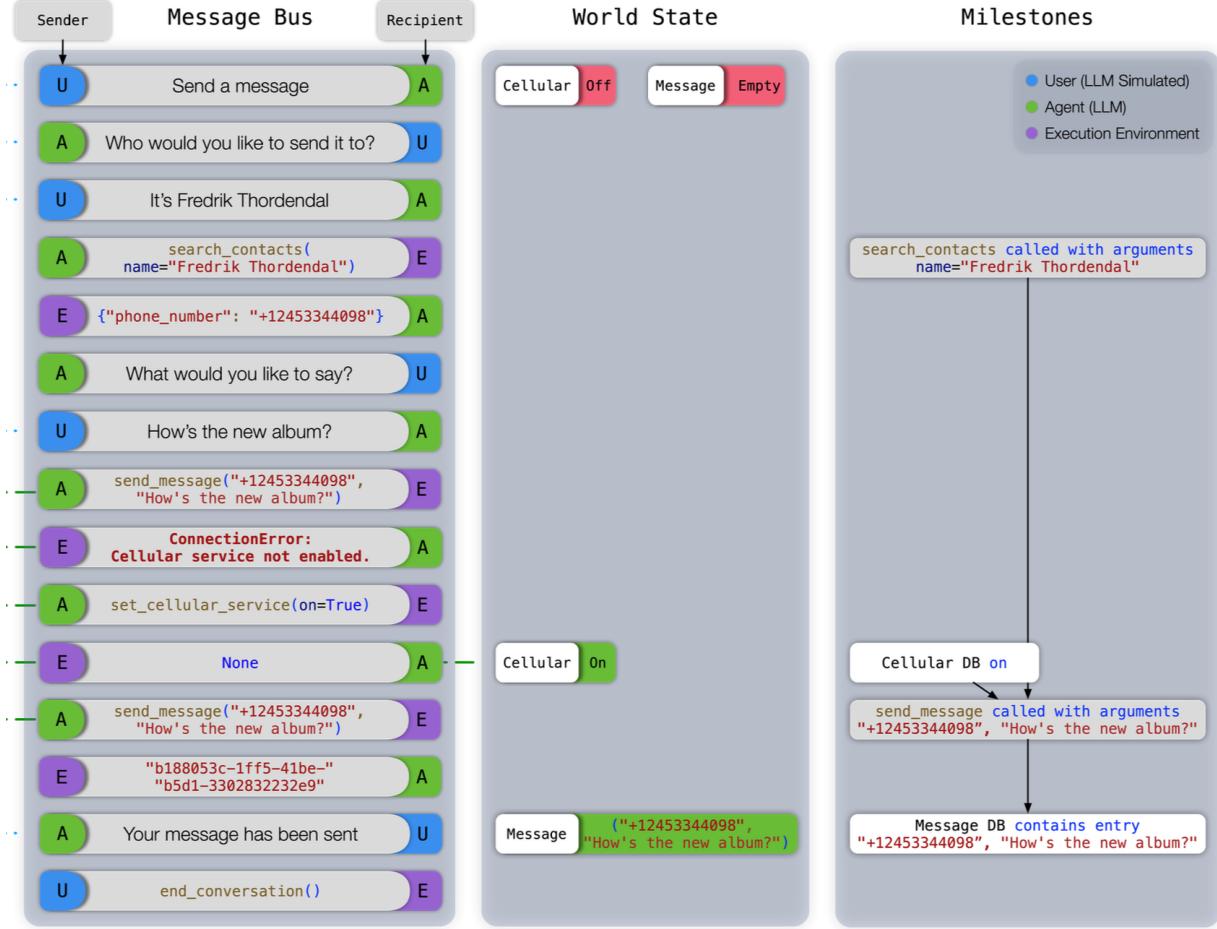


Figure 2.5: A conversation in ToolSandbox, the evolution of the world state, and the milestone DAG. Completion of the task requires additional information from the user, as well as multiple tool calls. From [14].

objectives and the edges correspond to their causal chains. Each milestone vertex v has an associated similarity measure $sim(v, s)$ with a snapshot s of the current database state, which encompasses internal state such as cellular status, calendar contents, and contact details. Depending on the nature of the milestone, $sim(v, s)$ involves different implementations: for tool calls like switching on cellular service, exact matching is used, while fuzzier operations like evaluating message content use ROUGE-L F instead. To evaluate tool traces, abstract syntax trees (ASTs) are used for fine-grained precision over the function's name, parameter types, arguments (see Appendix A.1 for an example flowchart). Each of these metrics are specialized for the specific type of milestone evaluated, and maps a milestone vertex and database snapshot to a score $[0, 1]$.

To allow for different solution paths in the overall trajectory score, the metric searches for a mapping $f^+ : S \rightarrow V_{M_+}$ that achieves the highest average similarity score for the milestones over a sequence of database snapshots $S_n = (s^1, s^2, \dots, s^n)$ at each conversation turn, provided that the mapped milestones form a topological sort of G_{M_+} . This ensures that the model is only graded on its performance towards concrete objectives, with ample flexibility to accommodate different style or planning. If there are less turns than milestones, or $|S_n| < |V_{M_+}|$, not all vertices will have a mapped snapshot, and we can consider $\text{sim}(v, \emptyset) = 0$. In this case, the mapped milestones will be considered valid if they form a subsequence of the topological sort of G_{M_+} . If there are more turns than milestones, or $|S_n| > |V_{M_+}|$, we only want to evaluate a subset $S_m \subset S_n$ of snapshots, so we will represent this by allowing f^+ to assign excess snapshots to a null milestone $v_{M_+}^0$, which are skipped when checking against the topological sort of the milestones.

Thus, as long as the causal chains in connected milestones are respected, they can appear at any turn in the conversation and receive the same score. For m milestones and a topological sort $\text{top}(G_{M_+})$, we can formalize the milestone score as:

$$\begin{aligned} \text{avg sim}_+ &= \frac{1}{m} \sum_{i=1}^m \left\{ \text{sim}(v_{M_+}^i, s^j) \mid f(s^j) = v_{M_+}^i \right\} \\ f^+ &= \arg \max_{f(S_n) \sqsubseteq \text{top}(G_{M_+})} \text{avg sim}_+ \\ \text{score}_{M_+} &= \max \text{avg sim}_+. \end{aligned}$$

Minefields M_- are the same concept as milestones, but for scenarios to avoid. Minefields are used to prevent tool hallucination and are typically assigned to scenarios where a model should not be able to complete a task with the specified tools. Just like milestones, they form a DAG $G_{M_-}(V_{M_-}, E_{M_-})$ and have an associated score_{M_-} . Unlike a milestone, hitting any minefield is immediately considered as failure and are thus placed sparingly. We can summarize the final score for a given scenario and conversation trajectory as:

$$\text{score} = \text{score}_{M_+} \times \mathbb{I}(\text{score}_{M_-} = 0)$$

While this evaluation method does not factor in conversational efficiency, ToolSandbox tracks the number of turns for the conversation along with the average similarity score as a rough proxy for the fluidity of task evolution. ToolSandbox contains scenarios for single and multi-step tool calls, scenarios with insufficient information, and scenarios where details

in the tools such as their names or parameters are scrambled. Its findings showcase a broad spectrum of ability in tool calling, with GPT4o taking the lead alongside Claude-3-Opus with an average score of around 70, while open source models such as Mistral-7B-Instruct-v0.3 and NousResearch’s Hermes-2-Pro-Mistral-7B¹ languished with an average score around 30 (see Table 2.3). While in-depth investigation into milestones and minefields are required to understand these numbers in context, they provide a rough signal of the general tool-calling ability and awareness model’s exhibit in natural conversation.

Avg Score ↑	Scenario Categories										Tool Augmentations					
	STC	MTC	SUT	MUT	SD	C	II	0 DT	3 DT	10 DT	AT	TNS	TDS	ADS	ATS	
GPT-4o-2024-05-13	73.0	87.8	80.1	84.2	74.7	84.0	76.6	42.0	75.1	75.0	74.6	72.6	72.4	69.3	73.0	71.9
Claude-3-Opus-20240229	69.2	83.5	70.0	74.5	67.2	74.5	71.1	57.3	68.3	68.6	70.0	67.5	70.8	71.5	65.8	71.1
GPT-3.5-Turbo-0125	65.6	93.4	73.9	81.8	66.6	82.6	70.4	22.3	67.3	63.2	67.0	65.4	63.9	64.3	66.7	66.9
GPT-4-0125-Preview	64.3	89.1	69.0	74.4	68.6	69.2	65.2	33.6	66.8	62.5	64.0	65.1	69.7	64.4	58.1	63.5
Claude-3-Sonnet-20240229	63.8	82.1	66.2	69.1	69.7	84.5	65.5	44.2	67.2	64.5	63.2	58.8	63.7	61.9	62.5	68.7
Gemini-1.5-Pro-001	60.4	82.6	49.8	63.1	37.3	70.5	51.6	76.2	63.3	63.1	60.8	59.8	62.2	60.5	58.7	54.4
Claude-3-Haiku-20240307	54.9	80.9	54.2	64.3	46.0	69.5	54.4	39.4	56.0	56.9	54.1	52.2	56.6	54.1	54.5	55.1
Gemini-1.0-Pro	38.1	68.7	21.6	36.5	14.6	39.3	18.2	65.5	38.2	39.5	41.9	37.7	40.1	35.3	36.7	34.9
Hermes-2-Pro-Mistral-7B	31.4	63.3	18.3	29.9	18.6	27.1	19.9	48.3	33.1	31.9	30.6	28.3	31.8	31.0	32.6	32.2
Mistral-7B-Instruct-v0.3	29.8	48.1	9.5	20.1	7.9	19.5	6.1	76.8	30.5	30.2	24.7	27.1	32.0	30.7	32.8	30.1
C4AI-Command-R-v01	26.2	52.6	12.7	23.0	12.7	3.1	18.0	47.8	24.8	27.9	25.6	23.3	25.0	25.6	28.7	28.3
Gorilla-Openfunctions-v2	25.6	36.2	8.2	15.1	9.3	0.0	8.9	69.2	25.5	27.5	26.1	18.6	24.5	27.1	26.8	28.6
C4AI-Command R+	24.7	57.2	13.6	24.3	15.2	4.0	19.4	35.3	23.4	27.2	24.9	23.5	21.7	27.6	24.8	24.8

Table 2.3: ToolSandbox model performance statistics from [14]. The first column represents the average similarity score across all categories. The next columns represent: Single Tool Call, Multiple Tool Call, Single User Turn, Multiple User Turn, State Dependency, Canonicalization, Insufficient Information, 0, 3, and 10 Distraction Tools, All Tools, Tool Name Scrambled, Tool Description Scrambled, Argument Description Scrambled and Argument Type Scrambled, respectively. These are explained in Section 3.2.

ToolSandbox is not without its shortcomings, specifically with the cost of inference, occasional unreliability of the user agents, and limited test cases, but it represents an exciting step forward, and its results reveal the immense advantages proprietary LLMs currently have over their open-source counterparts. Our work is part of a burgeoning movement to develop methods, data, and models to close this gap.

2.2 The Path Ahead

In this chapter, we’ve introduced many tool-augmented models with different implementations and focuses. From ReACT, we learn that intermediate reasoning can improve tool calling, and that fine-tuning is important to crystallize reasoning in smaller models. From ToolkenGPT, we see the importance in explicit focus on the parameters of tool calls. From

¹<https://huggingface.co/NousResearch/Hermes-2-Pro-Mistral-7B>

TooLLM, we see how diverse training data can allow for robust downstream generalization. From Toolformer, we glean the necessity of aligning our models with actionable objectives. And from ToolSandbox, we see the importance of clarifying ambiguity and recovering from errors in stateful simulated environments. To combine these insights, we will implement a fine-tuning plan that 1) takes explicit care of parameters 2) involves reasoning traces 3) is broad and generalizable 4) is robust to tool response errors, scrambled details, and lack of information. Before we start training, let's first explore the problem in more depth.

Chapter 3

Identifying Failure Modes in Existing Tool Acquisition Methods

To meaningfully improve on the current state of the art in open-source tool-augmented models, we must first understand where the performance gap lies. While we initially sought out to include academic models in evaluation, early experiments with ToolLLaMA showed that each instance of inference induced latency of more than 15 seconds. Furthermore, the model would often fail to respond at all to natural language instructions that did not require tools from its retrieved API set. As noted in the original ToolSandbox paper, some open-source models like Gorilla are powerful but unable to consume tool responses, making them unsuitable for a conversational sandbox [14].

Fundamentally, however, we felt that the current performance gap was a data distribution problem, not an algorithmic limitation. Hence, our experiments center around the open-source models evaluated in ToolSandbox, Mistral-7B-Instruct-v0.3 and Hermes-2-Pro-Mistral-7B, along with public datasets commonly used for tool acquisition. As neither model has released their tool-use data for inspection, our goal is to infer its deficiencies by examining scenarios under which model fails to correctly make tool calls.

3.1 About Our Models

Both Mistral-7B-Instruct-v0.3 and Hermes-2-Pro-Mistral-7B are derived from Mistral-7B, and are instruction-tuned with a variety of task and tool-calling data. While the architectural details of Mistral-7B are readily available in its release paper [10], information on data is far more illusory. Here’s what we know:

- Mistral-7B-Instruct-v0.3 extends the base Mistral-7B-v0.2 vocabulary size to 32768 and features a 32k context window. In its release blog¹, the training data is said to be “instruction datasets publicly available on HuggingFace” without further specification on mixing proportions, data content, or tool acquisition.
- NousResearch’s Hermes-2-Pro-Mistral-7B is fine-tuned from Mistral-7B-v0.2 and has a vocabulary size of 32000. While its massive instruction tuning dataset OpenHermes² is accessible with over a million examples, this corpus does not contain tool-calling data, which is separately mentioned in its HuggingFace model card but never specified.

As we are unable to inspect the data distribution for potential biases directly, we will instead use ToolSandbox to infer properties of the data via the models’ success and failure modes. For readability, all subsequent mentions of Mistral or Hermes will refer to these two specific models unless explicitly specified otherwise.

3.2 Reproducing the Benchmark

To begin, we replicated ToolSandbox’s experiments with Mistral and Hermes (see Table 3.1). While all scores were fairly similar, the models performed noticeably worse across the board in our evaluation. We hypothesize that this is due to differences in the user agent: while we use ChatGPT (GPT_3_5_0125) due to limitations with throughput and cost, the original ToolSandbox benchmarking used GPT4o (GPT_4_o_2024_05_13) instead³. This becomes significant in extreme cases with Mistral in particular as repeated responses in code can cause the user model to inadvertently try calling tools itself, raising an exception in the simulation environment and relinquishing any accumulated partial similarity scores for the scenario. An example of a user failure and error trace can be found in Appendix A.3.

Among the scenario categories, STC and MTC stand for single-tool call and multi-tool call, respectively. These categories provide the clearest measure of a model’s ability to use tools and perform multi-step planning. SUT and MUT stand for single user turn and multi-user turn, which refer to the amount of information the user provides to make the tool call. While SUT scenarios deliver all the details at once, MUT details require the

¹Quote from <https://mistral.ai/news/announcing-mistral-7b/>

²<https://huggingface.co/datasets/teknium/OpenHermes-2.5/>

³From <https://github.com/apple/ToolSandbox/issues/1>

Model	Avg Score	Scenario Categories							Tool Augmentations							
		STC	MTC	SUT	MUT	SD	C	II	0 DT	3 DT	10 DT	AT	TNS	TDS	ADS	ATS
Hermes-2-Pro-Mistral-7B	31.4	63.3	18.3	29.9	18.6	27.1	19.9	48.3	33.1	31.9	30.6	28.3	31.8	31.0	32.6	32.2
Hermes-2-Pro-Mistral-7B (us)	30.5	63.4	16.2	27.9	17.8	23.9	16.4	49.3	36.3	31.9	30.1	28.9	25.3	30.8	30.5	29.0
Mistral-7B-Instruct-v0.3	29.8	48.1	9.5	20.1	7.9	19.5	6.1	76.8	30.5	30.2	24.7	27.1	32.0	30.7	32.8	30.1
Mistral-7B-Instruct-v0.3 (us)	24.7	45.1	6.0	16.4	5.4	7.7	5.7	73.0	33.1	23.4	22.8	24.4	24.0	28.3	27.9	26.8

Table 3.1: Comparing ToolSandbox benchmark statistics (in grey) with our evaluations.

agent to clarify or request further information to complete the task (see Figure 2.5 for an example). SD, C, and II stand for State Dependency, Canonicalization, and Insufficient Information. In State Dependency, resolution of the task involves modifying the world state, such as turning the cellular status on. Canonicalization involves turning a natural language parameter such as "5PM" into the correct form for a function's consumption. Insufficient Information scenarios do not contain the necessary information for tool calls, and evaluate whether models can recognize to convey their tool-calling limitations.

Among the Tool Augmentations, 0 DT, 3 DT, and 10 DT stand for 0, 3, and 10 distraction tools, respectively, where the distraction tools are made up of valid and well-formed tools not needed for the task at hand. In practice, while models like Mistral leave the tool call for the user to parse and execute, agentic models could very well handle these executions themselves, with a variety of tools always available to the model regardless of the prompt. Thus, this metric represents a realistic scenario and required exploration of model reobustness in tool calling. AT stands for all tools available, and TNS, TDS, ADS, and ATS stand for tool names scrambled, tool descriptions scrambled, argument descriptions scrambled, and argument type scrambled. When names are scrambled, they no longer provide useful signal to the model, and force it to attend to other information in the tool description. While the noise may be a bit extreme for real-world scenarios, we agree with the emphasis on robustness as a proxy for in-depth tool acquisition.

As a single scenario may fall under multiple categories, these do not (and cannot, due to a degree of necessary overlap) represent ablations of isolated tool-use capacity. However, they are still informative in providing a general overview of the areas where models struggle and excel. In our results, we see dismal multi-step planning in both Mistral and Hermes with significant room for improvement in most categories, particularly with canonicalization of tool parameters. This implies that the tool acquisition training data for these models may not contain sufficient multi-step tool data and format conversions, which would definitely make sense if the data were highly synthetic.

3.3 Categorizing Weaknesses

When we examined the results on a per-scenario basis, we identified five critical failure modes for these models.

1. **Let Me Code That:** Rather than making tool calls, Mistral often responds by generating functions in a natural language response

Update the phone number of the last person I sent a message to to +10293847563

To update the phone number of the last person you sent a message to, you can use the `modify_contact` function. However, you first need to find the person's unique identifier (`person_id`). To do that, you'll need to search for the last message you sent and extract the sender's `person_id`. Here's the code to do that:

```
let last_message = search_messages({}); let
sender_person_id = last_message[0].sender_person_id;
modify_contact(person_id: sender_person_id,
phone_number: "+10293847563");
```

This code searches for the last message you sent (`search_messages({})`), extracts the sender's `person_id` (`last_message[0].sender_person_id`), and then updates the phone number of the contact with that `person_id` (`modify_contact({person_id: sender_person_id,
phone_number: "+10293847563"})`).

Figure 3.1: Mistral producing code in response to a user prompt instead of calling tools. In this format, users have a gray bubble while models have the green.

2. **I've got the time, I swear!** Models are prone to hallucinating model arguments, particularly with time and timestamps. In Hermes, the hallucinated arguments disproportionately favor feature the year 2022. See Figure 3.2.
3. **Was that a typo?** Models struggle to accommodate scrambled function names and argument types, failing to call tools they ordinarily have no problem with.

Add a to do to buy chocolate milk 3/22/2024 5PM

```
addReminder('content': 'Buy chocolate milk',  
'reminder_timestamp': 1476646400)
```

Figure 3.2: Mistral hallucinating a timestamp for `add_reminder` instead of using `convert_datetime_to_timestamp`. Tool calls are distinguished from normal response with the blue bubble. The provided UNIX timestamp is for October 16, 2016.

4. **Wait, where am I?** Models have fuzzy grasp on state like place or time, and have a tough time understanding how to reason about the future. See Appendix A.4.
5. **Drills, drills, drills!** Models will occasionally output tool calls with improper JSON formatting, causing exceptions that end the current round

3.4 Experiments with Prompting

To investigate model limitations, we first explored various ReACT-inspired prompting techniques to enhance model performance. As Mistral defaulted to outputting code for complex multi-step tool calls, we used Hermes for the majority of our experiments. By default, ToolSandbox supplied the agent models with the following system prompt before user input: “Don’t make assumptions about what values to plug into functions. Ask for clarification if a user request is ambiguous.”

To start, we experimented by appending “Think step-by-step” to the system prompt, which had no impact on Hermes’ responses. This is because the Hermes would exclusively respond either in natural language or a `<tool_call>`, leaving no room for natural language reasoning if tool use was triggered. We then tried using structured prompting methods such as NousResearch’s suggested `<scratch_pad>` framework, which instructed the model to reflect on the goal, required actions, and current progress before generating a tool call. Full details and an end-to-end example of the raw model responses can be found in Appendix A.5, A.6, A.7, and A.8.

Occasionally, with a high temperature and the scratch pad, stars would align and Hermes would output the contours of a plan. Unfortunately, even then, it would still fail to follow its own scratch pad and call the correct functions. We initially thought the struggles may be contained to one or two discrepancies and could be fixed by isolating the source

of error propagation. However, even when providing the correct first step to the message history, Hermes still struggled to produce the correlated intermediate function call (see Appendix A.8). While experiments contained flashes of surprisingly advanced reasoning capability, advanced prompting techniques still failed to allow Hermes to concretely handle tool calling in multi-step problem solving. As this intermediate tool reasoning seems to be out of distribution for Hermes-2 models, this finding is unfortunate but no real surprise.

3.5 Experiments with Fine-tuning

Next, we tried fine-tuning the Mistral-7B-v0.3 base model with naive tool calling data to get a sense for tool acquisition process. The mistral-finetune library⁴ suggests using Glaive AI’s open source tool-calling data⁵, a dataset of 113,000 examples of primarily single-call tool use in a chat format. However, early experiments demonstrated a surprising tendency for glaive-tuned models to hallucinate arguments like dates, mirroring Hermes. After digging deeper into the data, we found numerous examples of hallucinated functions (see Appendix A.2) and parameters, especially with respect to time-related data. Here is a training example where the year is never specified but set to 2022 by default:

```
USER: I need to find a hotel in New York from 1st May to 5th May.  
ASSISTANT: <functioncall> "name": "find_nearby_hotels", "arguments": ''  
"location": "New York", "check_in_date": "2022-05-01", "check_out_date":  
"2022-05-05" , <|endoftext|> FUNCTION RESPONSE: "hotels": [{"name": "Hotel  
A", "rating": 4.5, "price": "$200 per night", "name": "Hotel B", "rating":  
4.7, "price": "$250 per night", "name": "Hotel C", "rating": 4.3, "price":  
"$180 per night"}] ASSISTANT: Here are some hotels in New York from 1st  
May to 5th May: 1. Hotel A with a rating of 4.5 and it costs $200 per  
night. 2. Hotel B with a rating of 4.7 and it costs $250 per night. 3.  
Hotel C with a rating of 4.3 and it costs $180 per night. <|endoftext|>
```

Even when hallucinations do not occur, datetime data is rarely reformatted, which severely impacts models on other tasks as it mixes natural language into fields that are typically highly formatted for tool calling. If the training data for Hermes and Mistral contains similar errors, it’s no wonder they struggled with canonicalization. Here’s an example with two such instances:

⁴<https://github.com/mistralai/mistral-finetune>

⁵<https://huggingface.co/datasets/glaiveai/glaive-function-calling-v2>

USER: I need to set a reminder for my doctor's appointment. ASSISTANT: Sure, I can help with that. Could you please provide me with the date and time of your appointment? <|endoftext|> USER: It's on the 15th of next month at 10 AM. ASSISTANT: <functioncall> "name": "createReminder", "arguments": {"title": "Doctor's Appointment", "date": "15th of next month", "time": "10 AM"} <|endoftext|> FUNCTION RESPONSE: "status": "success", "message": "Reminder for 'Doctor's Appointment' on 15th of next month at 10 AM has been successfully created." ASSISTANT: Your reminder for the doctor's appointment on the 15th of next month at 10 AM has been successfully created. <|endoftext|>

Furthermore, while the data includes many different function names, the same scenarios for tool calls were often repeated, leading to redundancy in its arguments. For example, the tools `get_nearby_restaurants`, `find_nearest_restaurant`, `find_nearest_restaurants` (plural), `search_for_restaurants`, and many more all used the same exact conversation about looking for Italian Food in New York City. This suggests that the synthetic generation strategy was primarily focused on producing variants of tool names but not content, which led to catastrophic hallucinations of functions, arguments, and time in our initial finetuned model.

Between these findings and other syntax-related bugs in the message formatting, we developed conviction that a small, high-quality dataset could align Mistral for tool calling.

Chapter 4

A New Method for Tool Acquisition from Synthetic Datasets

As datasets of just a thousand hand-curated examples have been shown to elicit effective alignment, even for complex queries and specific response formats [37], we decided to use principles from our findings to develop a small, diverse training set for tool-acquisition. As existing synthetic datasets include valuable signal but lots of noise, we introduce heuristics to 1) get rid of undesirable examples in large tool-corpuses and 2) construct high-quality, varied scenarios for real-world robustness.

4.1 Styles of Thinking

First, let's try an exercise. Given these functions and this request, how would *you* solve this task?

```
{  
    'type': 'function',  
    'function': {  
        'name': 'get_user_location',  
        'description': 'Retrieves the current latitude and  
                        longitude of the user.',  
        'parameters': {  
            'type': 'object',  
            'properties': {}},
```

```

        'required': []
    },
}
}

{
  'type': 'function',
  'function': {
    'name': 'convert_latlng_to_address',
    'description': 'Converts geographical coordinates (latitude and
                    longitude) into a human-readable address',
    'parameters': {
      'type': 'object',
      'properties': {
        'latitude': {
          'type': 'number',
          'description': 'The latitude to ...'},
        'longitude': {
          'type': 'number',
          'description': 'The longitude to...'}
      },
      'required': ['latitude', 'longitude']
    }
  }
}

{
  'type': 'function',
  'function': {
    'name': 'get_travel_time_to_destination',
    'description': 'Calculates the estimated travel time...',
    'parameters': {
      'type': 'object',
      'properties': {

```

```

        'origin': {
            'type': 'string',
            'description': 'The origin address...'},
        'destination': {
            'type': 'string',
            'description': 'The destination address...'},
        'mode_of_transport': {
            'type': 'string',
            'description': 'The mode of transport...'},
        'avoid_tolls': {
            'type': 'boolean',
            'description': 'Whether to avoid potential tolls.'}
    },
    'required': ['origin', 'destination', 'mode_of_transport']
}
}
}

```

User prompt: How much time will it take to walk to Buckingham Palace?

4.1.1 Thinking Forwards

Forward thinking is the most canonical form of reasoning and is the typical style of thought used in multi-step planning. For this example, the first forward thought trace is:

[THOUGHT] To calculate how long it will take the user to walk to Buckingham Palace, we can first use the `get_user_location` function to retrieve the user's current latitude and longitude. Once we have the coordinates, we can then use the `convert_latlng_to_address` function with the acquired latitude and longitude to translate the coordinates into a human-readable address. Finally, we can use the `get_travel_time_to_destination` function with the user's address as the origin, Buckingham Palace as the destination, and set the `mode_of_transport` to "walking". [/THOUGHT]

Specifically, forward thinking connects the required functions in the order they are used, with particular emphasis on the parameters. In the aforementioned example, if asked to verbalize the solution path explicitly, you might have said something very similar to the

thought above.

4.1.2 Thinking Backwards

Backwards thinking, however, starts with the desired goal and seeks the requisite parameters. Take our original example and imagine ten functions instead of the three provided. Now imagine some function names are scrambled, the arguments are non-descriptive, and you received them out of order. Would that change your strategy for addressing the user prompt?

With tool calls, knowing how to start often requires knowledge of the entire reasoning chain, particularly if the format of the parameters is changed during the course of the task. While in our example, it may seem evident that getting the travel time requires the user's location, imagine some implementation where `get_user_location` requires an additional `user_id` from a `get_user_id` function. If this example was out of distribution, it seems far more demanding for the model to properly attend to the arguments and start its forward reasoning chain correctly with `get_user_id` than it would be adding an extra reasoning step in reverse.

Furthermore, imagine an example without the intermediate function `convert_latlng_to_address`. With backwards thought, we can immediately see from the arguments of `get_travel_time_to_destination` that this scenario has insufficient information to make a tool call. With forwards thought, this becomes harder to ascertain, and implicitly requires a backwards pass to verify. Here's the same example in backwards thought:

```
[THOUGHT] To calculate how long it will take the user to walk to Buckingham Palace, we can use the provided function get_travel_time_to_destination function with the parameters destination set to "Buckingham Palace" and mode_of_transport set to "walking", but we are missing the required parameter origin for the user's origin location. To get this location, we can use the get_user_location function to retrieve the user's current latitude and longitude and convert_latlng_to_address function to translate the coordinates into a human-readable address. [/THOUGHT]
```

Notice that in this example of backwards thought, we package together the `get_user_location` and `convert_latlng_to_address` functions. In examples where going backwards felt too unnatural (getting coordinates before an address), we collapse a conversion into the previous tool call. In the majority of cases, this is unnecessary.

4.1.3 Thinking in Code

The final variant is code thought, inspired by Mistral’s initial code generations in lieu of tool calls. Reasoning in code is not new: both Program of Thought [2] and Scratchpad [17] show that outputting intermediate reasoning traces in code improves numerical reasoning and algorithmic execution.

Nonetheless, tool calling presents its subtle differences. Rather than generating code to execute a task, or using code to reason numerically, we merely require the model to construct a plan with pre-provided functions to abide by. Specifically, we want to curtail creative generation that may lead to hallucination and instead learn to manipulate and reason with the provided tool components.

Concretely, code thought involves generating the tool calls in a functional style, helping align the model’s pre-trained coding abilities to execute complex tasks. Like forwards thought, code thought traces through scenarios from beginning to end:

```
[THOUGHT] To calculate how long it will take the user to walk to Buckingham Palace, we can chain together the provided functions:  
'''  
location = get_user_location["function"]["name"]()  
address = convert_latlng_to_address["function"]["name"](latitude=  
location["latitude"], longitude=location["longitude"])  
get_travel_time_to_destination["function"]["name"](origin=address,  
destination="Buckingham Palace", mode_of_transport="walking")  
''' [/THOUGHT]
```

While code thought is straightforward when the functions can be directly chained, it’s not clear what should happen when format conversions are required for arguments such as dates. In our implementation, we alternated between specifying the change of parameters with comments in the code blocks and breaking them up into natural language. An example of a disjointed code thought is:

```
[THOUGHT] To track a $50 expense on groceries, we can first get the current date: 'datestring = get_datestring()'. Then, we can reformat the "YYYY-MM-DD" 'datestring' into the "MM-DD-YYYY" format. Finally, we can call 'track_expenses(amount=50, category="groceries", date=formatted_datestring)'.  
[/THOUGHT]
```

4.2 Building the Dataset

To create our fine-tuning dataset¹, we are driven by two key objectives: inducing the capacity for generalization to out-of-distribution functions and activating multi-step planning. In doing so, we make the upmost effort to avoid any hallucinated functions, parameters, or thoughts, as we fear even a few noisy examples would interfere with successful tool acquisition.

4.2.1 On Generalizability

From Glaive’s function calling dataset of 113k rows, we undergo a selective filtering process to extract 516 high-quality single-tool trajectories for fine-tuning, representing 3096 training messages. We believe that de-duplicating the synthesized examples is the best way to ensure quality and diversity in our training set.

First, we got rid of examples where tools were not provided or the wrong tools were provided for the specified user request, leaving 73k and then 59k examples, respectively. While failure cases for tool calling are important to include, Glaive would often suggest completely unrelated tools in its follow-up message, which we viewed as undesirable behavior. This operation can be done by filtering for successful tool calls in the message history, particularly in the first assistant response, and checking whether tools are provided in the prompt.

From the remaining examples, we built a `set` out of the unique function names, of which there were 591. While we initially developed sampling-based techniques to form a training set from these function names, we realized that the examples we sampled for a given tool were almost entirely the same, with minor deviations in phrasing (see Appendix A.9). Hence, we decided to simply select one or two representative examples from each tool category for our training set. As we previously discovered hallucinations in examples with time data or multiple tool calls, we removed any tool that required time or date parameters from the pool, and sliced the conversations in the remaining examples to include only the context surrounding the first tool call. We also removed functions we deemed too similar to others or had poor descriptions (a full list of culled tools will be available on our GitHub), leaving a final dataset of 516 entries. Finally, we prompt GPT4o-mini with a representative hand-written example to generate the forwards, backwards, and code thought reasoning traces for our data annotations.

¹Our data can be found here: <https://github.com/itshuey/tool-acquisition>

4.2.2 On Planning

The second component of the data is a collection of multi-step tool trajectories that involve scenarios such as checking album release times, setting the thermostat in one’s home, and checking the weather in Tokyo.

To ideate the scenarios, we prompted GPT4o for potential applications of autonomous agents and example functions, requesting additional optional parameters when responses were too terse. This phase was largely informal and involved various versions of rewrites. After manually drafting the thoughts and responses for one scenario, we prompted GPT4o to rewrite the content in a different setting for additional parameter diversity.

Concretely, for a given multi-tool scenario, we include the following variants:

- All the requisite functions are supplied, allowing the agent to successfully complete the multi-tool scenario.
- A crucial intermediate tool is missing, forcing the agent to ask the user for clarification. We include examples where the user is helpful and when they are not, representing success and failure cases.
- No relevant tools are provided. The predicted response is taken directly from the model’s own response to the user prompt without any tool syntax, inspired by [25].
- If relevant, an example where information must be canonicalized from one version to another, whether it be time data, abbreviations, or shorthand.
- An example where function names are scrambled (but not parameter names or descriptions)

In addition, the scenarios encompass examples that involve error recovery, user time and place, and missing parameters. Care was taken that no scenario in ToolSandbox was leaked, though some individual tools such as `search_stock` and `get_time` are similar to examples in the sandbox. As early iterations without time or location-related tools exhibited no stateful reasoning, we settled on keeping the overlapping tools in our data but avoiding similar prompts, function names, and arguments.

With around 40 unique functions in 20 scenarios, we generate 877 high-quality assistant messages for training. In a final step, we use GPT4o to ensure that all tool calls were well-formed and had valid parameters.

4.2.3 Data Ablations

Along the development process, we ablated the two components separately, and confirmed that the model couldn't generalize to new tools very well without being fine-tuned on the cleaned Glaive data and couldn't reason very well without the planning data. Interestingly, we also found that training with an extra message for the assistant to acknowledge the tool response was a simple but important way to prevent spiraling. Since longer conversations fall into out-of-distribution territory if they progress past the training script, the model needs to have examples of how to end conversations, lest it spew tool calls endlessly even after completing the user request.

We also found that at this data scale, extremely small bugs could lead to detrimental downstream performance. In one of the experiments, one of my examples featured a single tool call with a boolean parameter set to `false` rather than `False`. After fine-tuning Mistral on the corrupted data, the model picked up the behavior, causing a `JsonDecodingError` whenever it predicted a `false` value for a tool call, despite having been previously instruction-tuned on python code generation. In total, our training set of $3096 + 877$ examples represents 3946 total examples, or 3.4% of the glaive dataset.

4.3 Training

All fine-tuning was done in 3 epochs on 4 H200 GPUs with a learning rate of 5e-6 and the AdamW optimizer. To facilitate tool acquisition, we implemented support for tool-calling dataset templates, chat formats, and recipes in a fork of the `torchtune` library².

4.3.1 Tokenization

To add support for thought traces, we added special tokens `[THOUGHT]` and `[/THOUGHT]` to the Mistral-7B-Instruct-v0.3 tokenizer in the style of the existing special tokens³ for `[TOOL_CALLS]`, `[AVAILABLE_TOOLS]`, and `[TOOL_RESULTS]`. Since Mistral currently does not support both natural language and tool calls in a single assistant message, we modified the chat template for easy conversion into the thoughtful tool-use format, and can be accessed on our github: <https://github.com/itshuey/tool-acquisition>.

²<https://github.com/itshuey/torchtune>

³https://huggingface.co/mistralai/Mistral-7B-Instruct-v0.3/blob/main/tokenizer_config.json

Chapter 5

Results: A New SoTA for Open-Source Tool-Augmented LLMs

5.1 Base Results

After fine-tuning Mistral-7B-Instruct-v0.3 on the forward, backward, and code thoughts, and well as a version with thoughts ablated, we evaluated the variants in ToolSandbox.

Model	Avg Score	Scenario Categories						
		STC	MTC	SUT	MUT	SD	C	II
Mistral	24.7	45.1	6.0	16.4	5.4	7.7	5.7	73.0
Code	27.9 (+13%)	51.8	23.0	30.9	21.8	31.3	21.3	21.9
Forward	29.6 (+20%)	56.9	24.8	32.6	26.2	21.2	28.2	22.5
Backwards	33.5 (+36%)	62.6	27.5	37.7	24.6	29.3	30.7	27.9
No Thought	20.3 (-18%)	13.6	2.5	5.6	2.1	3.7	1.8	90.0

Table 5.1: Average and scenario similarity scores for the fine-tuned variants and the base Mistral-7B-Instruct-v0.3 model. Percentage change from Mistral-7B-Instruct-v0.3 shown.

Among the fine-tuned models, backwards thought comes away as the clear winner in the majority of the scenarios, and represents a significant overall improvement over Mistral (+36%). While each of the Code, Forward, and Backward fine-tuned variants showed hefty improvement over the base Mistral model in both the single and multi-tool categories, backwards thought seems to excel in guiding complex planning.

Interestingly, No Thought, the model fine-tuned on the training data with thought traces removed, became the overwhelming favorite on Insufficient Information at the ex-

pense of every other category. Upon inspection, we discovered that the score was a mirage: the model frequently failed to produce well-formed tool calls, and thus avoided minefields on a technicality. This also accounts for Mistral’s inflated II score, as its code-generation responses similarly avoids minefields [14]. In many scenarios, No Thought gets stuck in a loop endlessly repeating parameters (see Figure A.10), which also plagued the base Mistral-7B-v0.3 model (not Instruct), indicating serious regression.

Model	Tool Augmentations							
	0 DT	3 DT	10 DT	AT	TNS	TDS	ADS	ATS
Mistral	33.1	23.4	22.8	24.4	24.0	28.3	27.9	26.8
Code	28.3	27.8	24.6	27.9	19.2	28.2	32.5	27.4
Forward	35.0	41.1	25.1	27.7	7.9	28.6	33.5	33.1
Backwards	32.2	31.4	33.3	38.6	28.3	32.8	32.3	33.0
No Thought	24.4	24.0	23.6	22.7	22.5	22.2	22.8	23.1

Table 5.2: Tool augmentation similarity scores for the fine-tuned variants and the base Mistral-7B-Instruct-v0.3 model

While forwards and backwards thought also topped the benchmark in augmentation categories, forwards thought had significant difficulties with scrambled tool names (TNS). We hypothesize that the nature of forward thought in tool calling makes it far more dependant on specific overlap between the training distribution and a test case than backwards thought, as it is much more difficult to learn to predict the first tool required in a sequence than the final tool before the output.

5.1.1 Further Experimental Results

Disappointed by the code thought results, we dug into the scenarios to investigate the root cause of the poor performance. While looking into the time-based scenarios, we noticed the Code model hallucinating tools it had seen during training. To address this, we introduced additional scrambling for the time and location functions without changing any of the base scenarios, and fine-tuned Code-v2 from Mistral-7B-Instruct-v0.3 model and the revised data. Additionally, we wanted to quantify the impact of intermediate instruction tuning on tool acquisition. Thus, we fine-tuned the base Mistral-7B-v0.3 with the Backwards data set and evaluated it in ToolSandbox.

This time, Code-v2 topped the charts, showing serious improvement from the original Code model and running away with most categories, representing a 17% improvement on

Model	Avg Score	Scenario Categories						
		STC	MTC	SUT	MUT	SD	C	II
Code-v2	35.8 (+45%)	66.6	31.1	39.9	32.2	45.0	29.2	20.5
Backwards	33.5 (+36%)	62.6	27.5	37.7	24.6	29.3	30.7	27.9
Hermes-2-Pro-Mistral-7B	30.5 (+23%)	63.4	16.2	27.9	17.8	23.9	16.4	49.3
Forward	29.6 (+20%)	56.9	24.8	32.6	26.2	21.2	28.2	22.5
Base-Backwards	28.7 (+16%)	64.1	21.1	34.4	15.8	33.2	21.1	21.3
Code	27.9 (+13%)	51.8	23.0	30.9	21.8	31.3	21.3	21.9
Mistral-7B-Instruct-v0.3	24.7 (+0%)	45.1	6.0	16.4	5.4	7.7	5.7	73.0
No Thought	20.3 (-18%)	13.6	2.5	5.6	2.1	3.7	1.8	90.0

Table 5.3: Scenario similarity scores for all evaluated models ranked in descending order. Percentage increase shown is in comparison to Mistral-7B-Instruct-v0.3 (although Base-Backwards and Hermes are trained from base Mistral).

the previous SoTA model Hermes and a 45% improvement on Mistral-7B-Instruct-v0.3. Notably, however, it struggled with canonicalization, which makes sense given that format changes are typically expressed explicitly and with functions in code data. Code-v2 also lagged in insufficient information, suggesting that code thought may inspire overconfidence. As code generation is typically unrestrained, it could be that out-of-distribution tool calling scenarios are activating parts of the model used in pure code generation, and more diverse examples are required to distinguish code thought from code generation. Additionally, while Base-Backwards edged out Mistral-7B-Instruct-v0.3, it could not catch up to the instruction-and-fine-tuned Backwards model, indicating that instruction tuning presents a significant advantage for tool acquisition.

Model	Tool Augmentations							
	0 DT	3 DT	10 DT	AT	TNS	TDS	ADS	ATS
Code-v2	41.8	37.9	39.6	30.8	10.1	36.3	38.8	36.9
Backwards	32.2	31.4	33.3	38.6	28.3	32.8	32.3	33.0
Hermes-2-Pro-Mistral-7B	36.3	31.9	30.1	28.9	25.3	30.8	30.5	29.0
Forward	35.0	41.1	25.1	27.7	7.9	28.6	33.5	33.1
Base-Backwards	32.5	29.6	28.3	29.0	14.6	27.7	30.9	27.5
Code	27.9	51.8	23.0	30.9	21.8	31.3	21.3	21.9
Mistral-7B-Instruct-v0.3	33.1	23.4	22.8	24.4	24.0	28.3	27.9	26.8
No Thought	24.4	24.0	23.6	22.7	22.5	22.2	22.8	23.1

Table 5.4: Tool augmentation similarity scores for all evaluated models.

Tool robustness showed much more diversity, with no clear trends in the relationship

between different categories. Surprisingly, the Code-v2 model did worse on scrambled tool names than Code, a strange finding given that the only difference between the model finetuning data was increased scrambling in the training set. In this context, scrambling does not involve rearranging the letters in the name of the function but rather replacing it with something non-descriptive, such as `function1` in our case or `rapid_api_search-tools_1` in ToolSandbox. Upon looking into the logs, the results held: while the Code-v2 model reduced its hallucination in normal settings, it had trouble thinking with functions that did not have descriptive naming. While code thinking, like forward thinking, can struggle with chaining out of distribution functions in its thoughts, that does not account for the regression between v1 and v2 models.

Overall, undergoing fine-tuning with thought traces significantly improved the tool-calling ability of the base models, while fine-tuning without thought traces degraded the base model. Amongst the thought traces, backwards and code thought seem to outperform forward thought, though the comparison is not entirely on even ground due to the extra scrambling advantage that Code-v2 received. Nonetheless, the experimental results largely verify our initial hypotheses about the different reasoning styles, and highlight the importance of data augmentations such as scrambling or obfuscation in promoting generalization and parameter consistency in tool calling.

Chapter 6

Limitations and Future Work

In this work, we have taken a comprehensive journey through the literature to understand best practices in tool acquisition, synthesizing them into a high-quality, diverse training set that elicits SoTA open-source tool-calling performance at 7B. Through our experiments, we find that 1) small training sets can effectively align models for tool acquisition, 2) any hallucinations in the training data are fatal for downstream performance, 3) backwards thinking and code thinking outperform forward thinking in single and multi-tool scenarios on average, and 4) scrambling function names can have significant implications on hallucination reduction even if it doesn't translate to improvements in robustness. We hope that these insights can improve current best practices in tool acquisition, and can be combined in tandem with algorithmic approaches to build effective and intelligent open-source tool-augmented models.

6.1 Limitations

While we are excited about the initial validation for our approach, there is still a lot of ground to cover. As we developed our current methodology through experimentation, our data generation process consists of a series of scripts in interactive sessions and a lot of human supervision, which can be difficult to reproduce beyond the current dataset size. We believe that there is potential to formalize our stated principles into a synthetic generation process to create high-quality data at scale based off hand-written seeds, minimizing manual intervention.

Furthermore, many more experiments are required to provide rigorous insight into the data acquisition process. While we evaluated impact on a few scenarios before and af-

ter introducing new examples or features (such as error recovery or user non-helpfulness), interpretation is limited by the lack of formal ablation data from ToolSandbox. Unfortunately, due to compute and inference limitations (with OpenAI and RapidAPI), we were restricted in the number of runs we could perform. We would have also liked to how the increased scrambling procedure from our top Code-v2 model would affect the other thought paradigms.

Looking beyond the fine-tuning data, we also found support that intermediate instruction-tuning meaningfully improved tool acquisition. However, as Mistral-7B-Instruct-v0.3’s instruction tuning process includes tool-use data, it’s difficult to draw concrete conclusions about training dynamics and best practices. To investigate this, we would need to start with Mistral-7B-v0.3 and perform our own instruction tuning with different mixing proportions. As tool calling is rising to prominence in user-facing applications, a well-balanced instruction-tuning dataset for tool acquisition would be a very valuable contribution. Evaluation on instruction-tuning benchmarks would also be a critical way to ensure that catastrophic forgetting or poor alignment were not occurring as a result of the tool acquisition process.

Finally, we faced limitations in inconsistent user simulation from GPT3.5 in ToolSandbox, which contributed to our benchmarking instability. As the user role is fairly limited, it would be interesting to see whether we could fine-tune a small model to substitute GPT3.5, or even slightly align GPT3.5 or GPT4o-mini (through the OpenAI API) to end conversations when needed and avoid tool calling even if prompted, allowing the agent model a fair evaluation in every scenario.

6.2 Future Work

This work raises questions about thought dynamics and tool acquisition which require thorough investigation from an interpretability perspective. To better understand the differences between forward, backward, and code thought, we could explore the exact tokens that are being attended to during the course of a toy tool call scenario. In addition, we could try to quantify the impact of tool name scrambling and other such permutations during fine-tuning by comparing the activations of the ablated models in self-contained scenarios. We find these questions fascinating – if only August went on forever, we could have given it a shot ourselves!

Appendix A

Additional Figures

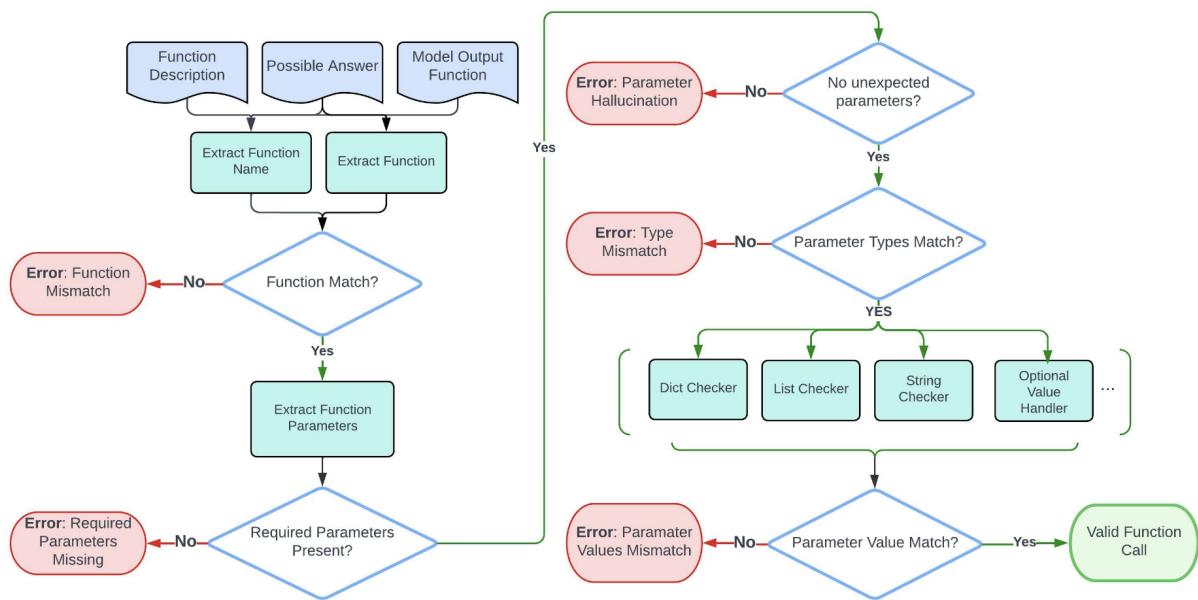


Figure A.1: An example of using abstract syntax trees to evaluate a function call. Taken from https://gorilla.cs.berkeley.edu/blogs/8_berkeley_function_calling_leaderboard.html [19]

```
SYSTEM: You are a helpful assistant with access to the following functions. Use them if required -  
{ "name": "search_jobs", "description": "Search for jobs based on title, location, or salary",  
"parameters": { "type": "object", "properties": { "title": { "type": "string", "description": "The  
title of the job" }, "location": { "type": "string", "description": "The location of the job" },  
"salary": { "type": "number", "description": "The minimum salary of the job" } }, "required": [  
"title", "location", "salary" ] } }
```

```
USER: Hi, I am looking for a job as a software engineer in New York with a minimum salary of  
$100,000. Can you help me? ASSISTANT: <functioncall> {"name": "search_jobs", "arguments":  
'{"title": "software engineer", "location": "New York", "salary": 100000}' } <|endoftext|> FUNCTION  
RESPONSE: {"jobs": [{"title": "Software Engineer", "location": "New York", "company": "Tech Corp",  
"salary": 105000}, {"title": "Software Engineer", "location": "New York", "company": "Innovative  
Solutions", "salary": 110000}]} ASSISTANT: I found 2 jobs for you. The first one is at Tech Corp  
with a salary of $105,000. The second one is at Innovative Solutions with a salary of $110,000.  
Would you like more information on these positions? <|endoftext|> USER: Yes, please tell me more  
about the job at Innovative Solutions. ASSISTANT: <functioncall> {"name": "get_job_details"  
"arguments": {"job_id": 2} } <|endoftext|> FUNCTION RESPONSE: {"job_details": {"title": "Software  
Engineer", "location": "New York", "company": "Innovative Solutions", "salary": 110000,  
"description": "You will be responsible for developing and maintaining our core software  
products.", "requirements": "Bachelor's degree in Computer Science or related field, 5+ years of  
experience in software development"} } ASSISTANT: The job at Innovative Solutions involves  
developing and maintaining their core software products. They require a Bachelor's degree in  
Computer Science or a related field and at least 5 years of experience in software development.  
<|endoftext|>
```

Figure A.2: Example of function hallucination in the Glaive function calling dataset.

USER	AGENT	How many km to Golden Gate Bridge?
AGENT	USER	<p> To find the distance to the Golden Gate Bridge, I would need the current location (latitude and longitude) as well as the latitude and longitude of the Golden Gate Bridge. Once I have those values, I can use the `calculate_lat_lon_distance` function to calculate the distance in kilometers.</p> <p> Here's an example of how you can call the function:</p> <pre> ````python current_latitude = 37.7749 current_longitude = -122.4194 golden_gate_bridge_latitude = 37.8212 golden_gate_bridge_longitude = -122.4542 distance = calculate_lat_lon_distance(latitude_0=current_latitude, longitude_0=current_longitude, latitude_1=golden_gate_bridge_latitude, longitude_1=golden_gate_bridge_longitude) print(distance) ````</pre> <p> Please provide the latitude and longitude of the Golden Gate Bridge or I can help you find it using the `search_location_around_lat_lon` function.</p>
USER	AGENT	<p> Let's try to find the location of the Golden Gate Bridge using the `search_location_around_lat_lon` function. Let me know if you need any assistance with that.</p>
AGENT	USER	<p> To find the location of the Golden Gate Bridge, I will use the `search_location_around_lat_lon` function with the location "Golden Gate Bridge" as the search query. Here's the code:<0x0A><0x0A>````python<0x0A>location = "Golden Gate Bridge"<0x0A>result = search_location_around_lat_lon(location=location)<0x0A>print(result)<0x0A>````<0x0A><0x0A>This will return a list of locations that match the search query. You can then access the latitude and longitude from the result.</p>
USER	AGENT	<p> Looks like I need to try something else. Would you like me to try a different approach to find the distance to the Golden Gate Bridge?</p>

Figure A.3: A failed conversation trajectory between ChatGPT3.5 (User) and Mistral (Agent). After some back and forth, conversation halts as the user attempts to call an agent function resulting in an error in the logs: `KeyError: "Agent tool call agent_facing_tool_name='get_current_location' is not a known allowed tool. Options are available_tool_names={'end_conversation'}"`

How many days is it till Christmas Day.

```
search_holiday(holiday_name='Christmas Day')
```

```
{"tool_name": "search_holiday", "arguments": {"holiday_name": "Christmas Day", "year": null}, "result": 1735084800.0}"
```

It is currently <tool_response>"content"</tool_response> seconds until Christmas Day. Let me convert that to days and seconds for you.

Great! Let me know once you have the converted value

```
timestamp_diff(timestamp_0=1735084800.0, timestamp_1=1683564800.0)
```

```
{"tool_name": "timestamp_diff", "arguments": {"timestamp_0": 1735084800.0, "timestamp_1": 1683564800.0}, "result": {"days": -597, "seconds": 60800}}}
```

There are <tool_response>"content"</tool_response> days and <tool_response>"content"</tool_response> seconds until Christmas Day.

Figure A.4: Hermes failing to understand state such as time. Tool responses are in dark gray and are provided by the execution environment. Beyond the <tool_response> syntax errors and the hallucination, the core issue is Hermes not realizing that it needs to get the current timestamp. While this also concerns time, this case is distinct from the case in Figure 3.2 as it requires stateful understanding. The hallucinated UNIX timestamp is for May 8, 2023.

```

<|im_start|>system
    You are a function calling AI model. You are provided with function signatures within <tools> </tools> XML tags. You may call one or more functions to assist with the user query. If available tools are not relevant in assisting with user query, just respond in natural conversational language. Don't make assumptions about what values to plug into functions. After calling & executing the functions, you will be provided with function results within <tool_response> </tool_response> XML tags.

<tools>
[{"type": "function", "function": {"name": "add_reminder", "description": "Add a reminder with optional location data.", "parameters": {"type": "object", "properties": {"content": {"type": "string", "description": "Content of the reminder"}, "reminder_timestamp": {"type": "number", "description": "POSIX timestamp indicating when the reminder should trigger"}, "latitude": {"type": "number", "description": "Optional latitude of the location associated with the reminder"}, "longitude": {"type": "number", "description": "Optional longitude of the location associated with the reminder"}, "required": ["content", "reminder_timestamp"]}, "returns": {"type": "string", "description": "Unique identifier for the reminder, used in other functions requiring this reminder"}}, {"type": "function", "function": {"name": "get_current_timestamp", "description": "Get the current POSIX timestamp.", "parameters": {"type": "object", "properties": {}}, "returns": {"type": "number", "description": "Current POSIX timestamp as a float value."}}, {"type": "function", "function": {"name": "timestamp_to_datetime_info", "description": "Convert a POSIX timestamp to a dictionary of date and time information.", "parameters": {"type": "object", "properties": {"timestamp": {"type": "number", "description": "Float representing POSIX timestamp"}}, "required": ["timestamp"]}, "returns": {"type": "object", "description": "A dictionary containing date and time information including year, month, day, hour, minute, second, and isoweekday.", "properties": {"year": {"type": "integer"}, "month": {"type": "integer"}, "day": {"type": "integer"}, "hour": {"type": "integer"}, "minute": {"type": "integer"}, "second": {"type": "integer"}, "isoweekday": {"type": "integer", "description": "Day of the week, where Monday is 1 and Sunday is 7"}}, {"type": "function", "function": {"name": "datetime_info_to_timestamp", "description": "Convert date and time information to a POSIX timestamp.", "parameters": {"type": "object", "properties": {"year": {"type": "integer", "description": "Year of the timestamp"}, "month": {"type": "integer", "description": "Month of the timestamp"}, "day": {"type": "integer", "description": "Day of the timestamp"}, "hour": {"type": "integer", "description": "Hour of the timestamp in 24-hour format"}, "minute": {"type": "integer", "description": "Minute of the timestamp"}, "second": {"type": "integer", "description": "Second of the timestamp"}}, "required": ["year", "month", "day", "hour", "minute", "second"]}, "returns": {"type": "number", "description": "POSIX timestamp as a float value."}}]
</tools>

For each function call return a JSON object, with the following pydantic model json schema:
{
  "title": "FunctionCall",
  "type": "object",
  "properties": {
    "name": {"title": "Name", "type": "string"},
    "arguments": {"title": "Arguments", "type": "object"}, "required": ["arguments", "name"]
  }
}
Each function call should be enclosed within <tool_call> </tool_call> XML tags. You must use <scratch_pad> </scratch_pad> XML tags to record your reasoning and planning before you call the functions as follows.

Example:
<scratch_pad>
Goal: <state task assigned by user>
Actions:
<if tool calls need to be generated>
- {result_var_name1} = functions.{function_name1}({param1}={value1}, ...)
- {result_var_name2, result_var_name3} = ...
<if no tool call needs to be generated> None
Observation: <set observation 'None' with tool calls; plan final tools results summary when provided>
Reflection: <evaluate query-tool relevance and required parameters when tools called; analyze overall task status when observations made>
</scratch_pad>
<tool_call>
{'name': <function-name>, 'arguments': <args-dict>}
</tool_call>

```

Figure A.5: The raw tool-calling system prompt for Hermes. We denote the boundaries of a chat message with <|im_start|> and <|im_end|>. After the system prompt (just the default from the documentation), we provide the available tools in <tools> tags, then define the <scratch_pad>. You can find more information about this here: <https://github.com/NousResearch/Hermes-Function-Calling>

```

<|im_start|>user
Can you set a reminder for me to pick up chocolate milk tomorrow at 5pm?<|im_end|>
<|im_start|>assistant
<scratch_pad>
Goal: Set a reminder for the user to pick up chocolate milk.
Actions:
- Get current timestamp to determine today's date.
- Calculate the timestamp for tomorrow at 5pm.
- Add a reminder with the given content and triggered timestamp.
Observation: The timestamp calculation and addition of the reminder are necessary.
Reflection: This task involves time manipulation and reminder setting functionality.
</scratch_pad>
<tool_call>
{'name': 'get_current_timestamp', 'arguments': {}}
</tool_call>

```

Figure A.6: To perform inference, we attach the user message to the system prompt in Appendix A.5 and pass it into the model. All the text in the figure past `<|im_start|>assistant` is the model’s response. On first glance, the plan seems solid. After the model generates a `<scratch_pad>` thought and `<tool_call>`, it is on the user to fulfill the tool call.

```

<|im_start|>tool
1723746829<|im_end|>
<|im_start|>assistant
<scratch_pad>
Goal: Set a reminder for the user to pick up chocolate milk.
Actions:
- Get current timestamp (<tool_response>1723746829</tool_response>)
- Calculate the timestamp for tomorrow at 5pm using the obtained current timestamp.
- Add a reminder with the given content and triggered timestamp.
Observation: The current timestamp is obtained successfully.
Reflection: Next step is to calculate the timestamp for tomorrow at 5pm.
</scratch_pad>
<tool_call>
{'name': 'timestamp_to_datetime_info', 'arguments': {'timestamp': 1723746829}}
</tool_call>
<|im_start|>tool
{year:2024, month:8, day:15, hour:19, minute:33, second: 49}<|im_end|>

```

Figure A.7: Continuing from Appendix A.6, we call the `get_current_timestamp` function and append the response to the prompt to feed back into the model for another round of inference. It correctly acknowledges the result and makes the correct next tool call, which we also fulfill. In reality, the model does not end generation after its message, and generates its own tool responses and user requests. During experimentation, we manually slice and append only the correct part of the response to the growing prompt.

```

<|im_start|>assistant
<scratch_pad>
Goal: Set a reminder for the user to pick up chocolate milk.
Actions:
- Get current timestamp (<tool_response>1723746829</tool_response>)
- Calculate the timestamp for tomorrow at 5pm using the obtained current timestamp: {year:2024, month: 8, day:15, hour:19, minute:33, second:49}.
- Add a reminder with the given content and triggered timestamp.
Observation: The datetime info for the current timestamp is obtained successfully.
Reflection: Next step is to add the reminder with the calculated timestamp.
</scratch_pad>
<tool_call>
{'name': 'addReminder', 'arguments': {'content': 'Pick up chocolate milk', 'reminder_timestamp': 1724912829, 'latitude': None, 'longitude': None}}
</tool_call>

```

Figure A.8: Continuing from Appendix A.7, the assistant attempts to directly increment the UNIX timestamp instead of calling `datetime_info_to_timestamp` with the state obtained from the `timestamp_to_datetime_info` tool response, incremented where appropriate. The UNIX timestamp in the reminder corresponds to Aug 29, 2024 at 06:27:09, which is not the correct datetime. Despite coming up with good intermediate thoughts, the model is unable to complete the task.

```

>>> for d in d4:
...     if d['tools'][0]['function']['name'] == 'check_movie_showtimes':
...         print(d['messages'][1]['content'])
...
Hi, can you check the showtimes for the movie "Avengers: Endgame" on 2022-03-20 in New York, NY?
Hi, I want to watch the movie "Inception". Can you check the showtimes in New York for me?
Hi, I want to watch the movie "Inception". Can you check the showtimes for it in New York today?
Hi, I want to watch the movie "Inception" tomorrow in New York. Can you check the showtimes for me?
Hi, I want to watch the movie "Inception". Can you check the showtimes for it in New York today?
Can you check the showtimes for the movie "Inception" in New York?
Hi, I want to watch the movie "Inception". Can you check the showtimes in New York for me?

```

Figure A.9: Examining the examples in the Glaive function calling dataset with the tool `check_movie_showtimes`.

Bibliography

- [1] Tom Brown et al. “Language Models are Few-Shot Learners”. In: *Advances in Neural Information Processing Systems*. Ed. by H. Larochelle et al. Vol. 33. Curran Associates, Inc., 2020, pp. 1877–1901. URL: https://proceedings.neurips.cc/paper_files/paper/2020/file/1457c0d6bfc4967418bfb8ac142f64a-Paper.pdf.
- [2] Wenhui Chen et al. “Program of Thoughts Prompting: Disentangling Computation from Reasoning for Numerical Reasoning Tasks”. In: *Transactions on Machine Learning Research* (2023). ISSN: 2835-8856.
- [3] Karl Cobbe et al. “Training verifiers to solve math word problems”. In: *arXiv preprint arXiv:2110.14168* (2021).
- [4] Yunfan Gao et al. *Retrieval-Augmented Generation for Large Language Models: A Survey*. 2024. arXiv: 2312 . 10997 [cs.CL]. URL: <https://arxiv.org/abs/2312.10997>.
- [5] Anchun Gui et al. *Look Before You Leap: Towards Decision-Aware and Generalizable Tool-Usage for Large Language Models*. 2024. arXiv: 2402 . 16696 [cs.CL]. URL: <https://arxiv.org/abs/2402.16696>.
- [6] Zhicheng Guo et al. *StableToolBench: Towards Stable Large-Scale Benchmarking on Tool Learning of Large Language Models*. 2024. arXiv: 2403 . 07714 [cs.CL]. URL: <https://arxiv.org/abs/2403.07714>.
- [7] Shibo Hao et al. “Toolkengpt: Augmenting frozen language models with massive tools via tool embeddings”. In: *Advances in neural information processing systems* 36 (2024).
- [8] Yue Huang et al. *MetaTool Benchmark for Large Language Models: Deciding Whether to Use Tools and Which to Use*. 2024. arXiv: 2310 . 03128 [cs.SE]. URL: <https://arxiv.org/abs/2310.03128>.

- [9] Alon Jacovi et al. “A Comprehensive Evaluation of Tool-Assisted Generation Strategies”. In: Jan. 2023, pp. 13856–13878. doi: 10.18653/v1/2023.findings-emnlp.926.
- [10] Albert Q. Jiang et al. *Mistral 7B*. 2023. arXiv: 2310.06825 [cs.CL]. URL: <https://arxiv.org/abs/2310.06825>.
- [11] Bethany A. Kahrs, Wendy P. Jung, and Jeffrey J. Lockman. “When does tool use become distinctively human? Hammering in young children”. In: *Child Development* 85.3 (May 2014). Epub 2013 Oct 15, pp. 1050–1061. doi: 10.1111/cdev.12179.
- [12] Patrick Lewis et al. “Retrieval-augmented generation for knowledge-intensive nlp tasks”. In: *Advances in Neural Information Processing Systems* 33 (2020), pp. 9459–9474.
- [13] Minghao Li et al. “API-Bank: A Comprehensive Benchmark for Tool-Augmented LLMs”. In: *The 2023 Conference on Empirical Methods in Natural Language Processing*. 2023. URL: <https://arxiv.org/abs/2304.08244>.
- [14] Jiarui Lu et al. *ToolSandbox: A Stateful, Conversational, Interactive Evaluation Benchmark for LLM Tool Use Capabilities*. 2024. arXiv: 2408.04682 [cs.CL]. URL: <https://arxiv.org/abs/2408.04682>.
- [15] W.C. McGrew. “Tool use by wild chimpanzees in feeding upon driver ants”. In: *Journal of Human Evolution* 3.6 (1974), pp. 501–508. ISSN: 0047-2484. doi: [https://doi.org/10.1016/0047-2484\(74\)90010-4](https://doi.org/10.1016/0047-2484(74)90010-4). URL: <https://www.sciencedirect.com/science/article/pii/0047248474900104>.
- [16] Reiichiro Nakano et al. “WebGPT: Browser-assisted question-answering with human feedback”. In: *CoRR* abs/2112.09332 (2021). arXiv: 2112.09332. URL: <https://arxiv.org/abs/2112.09332>.
- [17] Maxwell I. Nye et al. “Show Your Work: Scratchpads for Intermediate Computation with Language Models”. In: *CoRR* abs/2112.00114 (2021). arXiv: 2112.00114. URL: <https://arxiv.org/abs/2112.00114>.
- [18] Liangming Pan et al. “Fact-Checking Complex Claims with Program-Guided Reasoning”. In: *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Ed. by Anna Rogers, Jordan Boyd-Graber, and Naoaki Okazaki. Toronto, Canada: Association for Computational Linguistics,

July 2023, pp. 6981–7004. DOI: 10.18653/v1/2023.acl-long.386. URL: <https://aclanthology.org/2023.acl-long.386>.

- [19] Shishir G. Patil et al. *Gorilla: Large Language Model Connected with Massive APIs*. 2023. arXiv: 2305.15334 [cs.CL]. URL: <https://arxiv.org/abs/2305.15334>.
- [20] Cheng Qian et al. “Toolink: Linking Toolkit Creation and Using through Chain-of-Solving on Open-Source Model”. In: *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*. Ed. by Kevin Duh, Helena Gomez, and Steven Bethard. Mexico City, Mexico: Association for Computational Linguistics, June 2024, pp. 831–854. DOI: 10.18653/v1/2024.naacl-long.48. URL: <https://aclanthology.org/2024.naacl-long.48>.
- [21] Yujia Qin et al. *Tool Learning with Foundation Models*. 2024. arXiv: 2304.08354 [cs.CL]. URL: <https://arxiv.org/abs/2304.08354>.
- [22] Yujia Qin et al. “ToolLLM: Facilitating Large Language Models to Master 16000+ Real-world APIs”. In: *The Twelfth International Conference on Learning Representations*. 2024. URL: <https://arxiv.org/abs/2307.16789>.
- [23] Changle Qu et al. *Tool Learning with Large Language Models: A Survey*. 2024. arXiv: 2405.17935 [cs.CL]. URL: <https://arxiv.org/abs/2405.17935>.
- [24] Mayk Caldas Ramos, Christopher J. Collison, and Andrew D. White. *A Review of Large Language Models and Autonomous Agents in Chemistry*. 2024. arXiv: 2407.01603 [cs.LG]. URL: <https://arxiv.org/abs/2407.01603>.
- [25] Xuan Ren, Biao Wu, and Lingqiao Liu. *I Learn Better If You Speak My Language: Understanding the Superior Performance of Fine-Tuning Large Language Models with LLM-Generated Responses*. 2024. arXiv: 2402.11192 [cs.CL]. URL: <https://arxiv.org/abs/2402.11192>.
- [26] Timo Schick et al. “Toolformer: Language models can teach themselves to use tools”. In: *Advances in Neural Information Processing Systems* 36 (2024).
- [27] Weizhou Shen et al. *Small LLMs Are Weak Tool Learners: A Multi-LLM Agent*. 2024. arXiv: 2401.07324 [cs.AI]. URL: <https://arxiv.org/abs/2401.07324>.
- [28] Robert W. Shumaker, Kristina R. Walkup, and Benjamin B. Beck. *Animal tool behavior: the use and manufacture of tools by animals*. Baltimore, MD: Johns Hopkins University Press, 2011.

- [29] Hugo Touvron et al. *LLaMA: Open and Efficient Foundation Language Models*. 2023. arXiv: 2302.13971 [cs.CL]. URL: <https://arxiv.org/abs/2302.13971>.
- [30] Boshi Wang et al. “LLMs in the Imaginarium: Tool Learning through Simulated Trial and Error”. In: *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Ed. by Lun-Wei Ku, Andre Martins, and Vivek Srikumar. Bangkok, Thailand: Association for Computational Linguistics, Aug. 2024, pp. 10583–10604. URL: <https://aclanthology.org/2024.acl-long.570>.
- [31] Lei Wang et al. “A survey on large language model based autonomous agents”. In: *Frontiers of Computer Science* 18.6 (2024). ISSN: 2095-2236. DOI: 10.1007/s11704-024-40231-1. URL: <http://dx.doi.org/10.1007/s11704-024-40231-1>.
- [32] Jason Wei et al. “Chain-of-thought prompting elicits reasoning in large language models”. In: *Advances in neural information processing systems* 35 (2022), pp. 24824–24837.
- [33] Zhilin Yang et al. “HotpotQA: A Dataset for Diverse, Explainable Multi-hop Question Answering”. In: *CoRR* abs/1809.09600 (2018). arXiv: 1809.09600. URL: <http://arxiv.org/abs/1809.09600>.
- [34] Jia-Yu Yao et al. *LLM Lies: Hallucinations are not Bugs, but Features as Adversarial Examples*. 2024. arXiv: 2310.01469 [cs.CL]. URL: <https://arxiv.org/abs/2310.01469>.
- [35] Shunyu Yao et al. “ReAct: Synergizing Reasoning and Acting in Language Models”. In: *International Conference on Learning Representations (ICLR)*. 2023.
- [36] Junjie Ye et al. *RoTBench: A Multi-Level Benchmark for Evaluating the Robustness of Large Language Models in Tool Learning*. 2024. arXiv: 2401.08326 [cs.CL]. URL: <https://arxiv.org/abs/2401.08326>.
- [37] Chunting Zhou et al. “Lima: Less is more for alignment”. In: *Advances in Neural Information Processing Systems* 36 (2024).