TOOLDEC: Syntax Error-Free and Generalizable Tool Use for LLMs via Finite-State Decoding

Hongqiao Chen* Northwood High School

Kexun Zhang* UC Santa Barbara

Lei Li Carnegie Mellon University

William Yang Wang UC Santa Barbara

Abstract

Large Language Models (LLMs) have shown promising capabilities in using external tools. However, existing approaches rely on fine-tuning or in-context learning to use tools, which make syntactic mistakes and are difficult to generalize. In this paper, we propose TOOLDEC, a finite-state machine-guided decoding algorithm for tool-augmented LLMs. TOOLDEC eliminates tool-related errors by ensuring valid tool names and type-conforming arguments. Furthermore, TOOLDEC enables LLM to effectively select tools using only the information contained in their names, with no need for tool-specific fine-tuning. Our experiments on multiple word problem datasets show that TOOLDEC reduces syntactic errors to zero, consequently achieving significantly better performance and as much as a 2x speedup. We also show that TOOLDEC achieves superior generalization performance on unseen tools, performing up to 8x better than the baseline.

1 Introduction

Augmenting Large Language Models (LLMs) with external tools (Mialon et al., 2023) enables them to solve complex problems. The performance of a tool-augmented LLM depends on its ability to make three key decisions—when to use a tool, which tool to use, and how to invoke a tool. Existing approaches learn to make these decisions through fine-tuning or in-context learning. However, these approaches still make syntactic mistakes in tool calls, calling non-existent tools and passing invalid arguments. Furthermore, prior approaches do not generalize to unseen tools well. They either need in-context documentation or extra training data to adopt new tools.

To address these issues, we propose TOOLDEC, a decoding algorithm guided by a finite-state machine (FSM) to ensure LLMs invoke tools properly. TOOLDEC transitions from state to state as decoding progresses. At each decoding step, TOOLDEC only samples from a subset of tokens allowed by the current state. The FSM that gives guidance to TOOLDEC is constructed from tool documentation and API signature so that the machine precisely represents the grammar of tool calls. In this way, TOOLDEC is able to always generate syntactically correct tool calls. Figure 1 illustrates that an LLM enhanced by TOOLDEC is able to generate the right function call `multiply` with precise arguments.

Furthermore, TOOLDEC generalizes to new tools much more efficiently. TOOLDEC automatically constructs a finite-state machine from a new tool's API signature and adds it to the existing FSM. TOOLDEC is then able to call new tools without fine-tuning on extra data or in-context tool documentation. In Figure 1, both `product` and `multiply` sound plausible for the scenario, but only `multiply` is a given tool. Since TOOLDEC only calls existent tools, it won't hallucinate a plausible yet non-existent tool and can rely on the tool names to find the right tool.

*Equal contribution. Correspondence to `kexun@ucsb.edu`.

37th Conference on Neural Information Processing Systems (NeurIPS 2023) Workshop on MATH-AI.

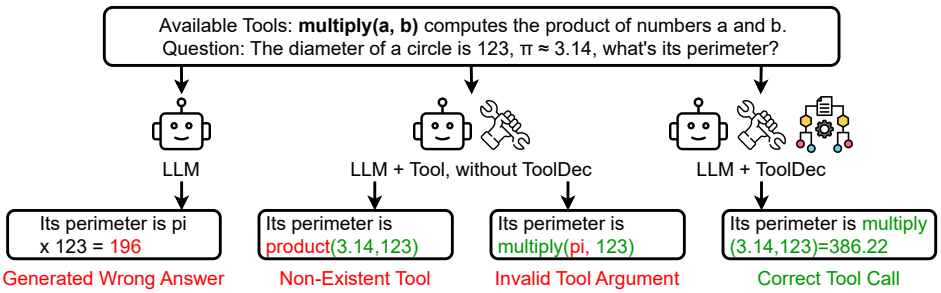

Figure 1: LLMs using external tools. LLMs without tools cannot multiply, so they just generate a probable next token. Tool-augmented LLMs can access external tools like `multiply`, but they may call a non-existent tool like `product` and pass invalid arguments like the string "pi". Our proposed ToolDec always generates tool calls without syntax errors.

Our extensive experiments show ToolDec eliminates all syntax errors and hallucinated tool names, resulting in better accuracy and as much as 50% less inference time. Our results also indicate that ToolDec generalizes efficiently without extra training data, achieving 8x better than baselines on mathematical reasoning with 9 out of 13 tools unseen.

2 ToolDec: LLM Tool Use via Finite-State Decoding

Motivated by the fact that a finite state machine can verify the syntax of a tool call, we propose ToolDec, a constrained decoding algorithm guided by an FSM. During each decoding step, the model samples from a subset of the vocabulary that only contains syntactically correct tokens. The FSM that specifies the token subsets can be constructed from the tool documentation. For example, in Figure 2, an FSM is constructed for functions `add`, `exp`, `square` and `sqrt`. Table 1 shows how ToolDec answers the question "the side of a square is 5, what's its area?" using the FSM. With the guidance from the FSM, ToolDec achieves the following goals:

- **Switching Modes.** Switch between "text mode" when the model is free to generate any text and "tool mode" when the model can only generate valid tool calls.
- **Generating Tool Names.** At the beginning of a tool call, only generate *correct existent* tool names from a pre-defined list of tools.
- **Passing Valid Arguments.** Only pass type-conforming arguments to the tool.

2.1 Finite-State Decoding

ToolDec is guided by a finite-state machine (FSM). An FSM is a 5-tuple (S, V, g, s_0, R), consisting of a finite state set S, an alphabet V, a transition function $g : S \times V \to S$, an initial state s_0 and a set of accepting states R. In our case, S and g are constructed from the tool signature. V is the token vocabulary of the language model. R corresponds to pre-defined tokens that can determine the LM has completed the task, like '<EOS>'.

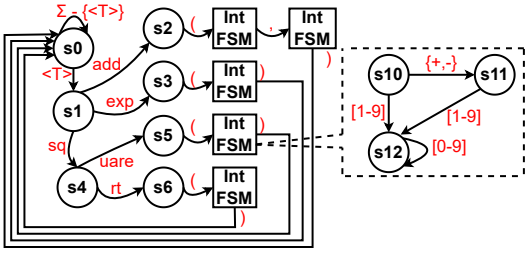

Figure 2: A finite-state machine for ToolDec constructed for math functions `add`, `exp`, `square`, `sqrt` that take integers as arguments. The names of the tools are represented with a trie structure. "IntFSM" is a submodule that parses integers.

At each decoding step t, ToolDec maintains a current state s. It can only generate the tokens permitted by the FSM, i.e. the tokens for which $g(s, \cdot)$ is defined. These permitted tokens are a subset of V and we denote them as V_s. After generating one token a, ToolDec transits to another state $g(s, a)$ specified by the FSM transition function. At each step t, we do not directly sample from the next token distribution $P(x_t|x_{1..t-1})$ calculated by the LLM. Instead, we zero out the probabilities

Table 1: How TOOLDEC answers the question "the side of a square is 5, what's its area?"

Step	Generated Text	Current State g	Valid Next Tokens	Next Token	Next State
t	Its	s_0	whole vocab.	'area'	s_0
$t+1$	Its area	s_0	whole vocab.	'is'	s_0
$t+2$	Its area is	s_0	whole vocab.	'<T>'	s_1
$t+3$	Its area is <T>	s_1	'add', 'exp', 'sq'	'sq'	s_4
$t+4$	Its area is <T>sq	s_4	'uare', 'rt'	'uare'	s_5
$t+5$	Its area is <T>square	s_5	'('	'('	s_{10}
$t+6$	Its area is <T>square(	s_{10}	'+','-','1'-'9'	'5'	s_{12}
$t+7$	Its area is <T>square(5	s_{12}	'0'-'9', ')'	')'	s_{10}
$t+8$	Its area is <T>square(5)	s_{10}	whole vocab.	-	-

of invalid tokens for which the transition function is undefined, and normalize the probabilities,

$$\tilde{P}(x_t = a|x_{1..t-1}, s) = \begin{cases} \frac{P(x_t=a|x_{1..t-1})}{\sum_{a' \in V_s} P(x_t=a'|x_{1..t-1})}, & g(s,a) \text{ is defined,} \\ 0, & \text{otherwise} \end{cases}.$$

The next token a is then sampled from the modified distribution $\tilde{P}(x_t|x_{1..t-1}, s)$. With the next token, we move on to the next decoding step and transition the current state s to the next state $g(s, a)$. The pseudo-code of this algorithm is listed in Appendix A.2.

2.2 Constructing FSMs that Guarantee Syntactically Correct Tool Calls

Switching Modes. We use two states in the FSM to represent whether the language model is in text mode or tool mode. The text mode is denoted by the initial state s_0, during which the model is free to generate any token in its vocabulary, i.e. $V_{s_0} = V$. The tool mode is denoted by state s_1. The model needs to output the special token `<T>` to switch from text mode to tool mode. Starting at s_1 is a smaller FSM that describes the grammar of tool calls.

Generating Tool Names. Once the model is at state s_1, the next step would be to generate a new tool call. To generate a correct tool call, the model needs to output the correct name for the tool and a correct list of arguments. If every tool had a single-token name, we could simply define V_{s_1} as the set of tool names. However, that is not the case for most tools. Many tools have long names that need multiple tokens to represent. For example, the tool `square` in Figure 2 consists of two tokens— 'sq' and 'uare'. Therefore, we need to construct an FSM for multi-token tool names.

We use a trie (Fredkin, 1960) to construct the FSM for tool names. In Figure 2, the trie for tool names consists of states $\{s_1, s_2, s_3, s_4, s_5, s_6\}$. A trie is a rooted tree in which each edge represents a token. A node in the tree represents a string that's the concatenation of the path from the root to this node. In our example, s_4 represents the string "sq" and s_5 represents the string "square". All nodes in a trie represent a set of strings. We construct a trie for the valid tool names and make s_1 the root of this trie.

To construct a trie, we insert all the strings into it one by one. Inserting a string into a trie means going from the root down the path made by the string and creating new nodes when the next step in the path does not exist. For example, we show how two more tools names, `exp10` and `expand` can be added to the trie in Figure 3.

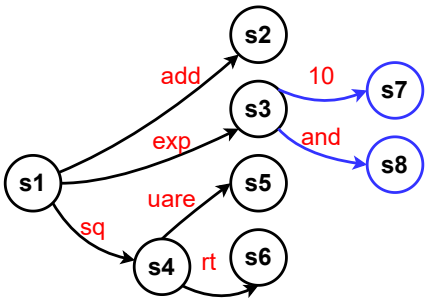

Figure 3: Inserting two more tool names `exp10` and `expand` into the trie. They are represented by the two blue states s_7 and s_8.

Note that the construction of trie depends on one assumption: no two tools have the same name. While this is a reasonable assumption to make, there could be exceptions in real applications. In that case, we could rewrite the tool names to include more details to disambiguate them. Rewriting abstract and hard-to-understand tool names can also make it easier for the language model to select them by name.

Generating Syntactically Valid Tool Arguments. Tool arguments have specified types. Like arguments in a program, they need to follow certain grammar rules. These rules can be specified by finite-state machines. For example, the "IntFSM" in Figure 2 depicts a finite-state machine that only accepts integer literals. For all arguments of a tool, we chain their corresponding FSMs together and use the last state corresponding to the tool name as the initial state of this FSM chain. Note that in practice, it's not necessary to explicitly construct this FSM. Any grammar checker that tells the set of valid next tokens suffice.

3 Experiments

Since the fine-tuning approach is very effective in math reasoning (Hao et al., 2023) , our main experiments focus on demonstrating how TOOLDEC can eliminate all syntax errors and hallucinated tool names while enabling the fine-tuned LLM to generalizes efficiently without extra training data.

3.1 Baseline and Benchmark

ToolkenGPT (Hao et al., 2023) represents each tool as a special token and optimizes only the embedding of the tool tokens for tool use. During inference, ToolkenGPT invokes a tool once the corresponding special token is predicted. During a tool call, it passes arguments by learning from in-context demonstrations. ToolkenGPT uses LLaMA-33B (Touvron et al., 2023) as its base model.

We evaluated TOOLDEC's performance on **FuncQA$_{\text{multi}}$**(Hao et al., 2023), which tests LLMs' ability in numerical reasoning tasks with 68 math problems. LLMs are required to produce a numerical answer using a few of the 13 arithmetic operations as tools (e.g. `multiply`, `power`, `lcm`). Following Hao et al. (2023), we report results of other baselines, including ChatGPT without tools and LLaMA with ReAct and tools.

3.2 Integrating TOOLDEC with the Base Model

Since ToolkenGPT uses special tokens to call tools, in the first setting we apply TOOLDEC only to guarantee the syntax of arguments. Our FSM guarantees that every argument is a valid number, and arguments are separated by commas. It also guarantees that the actual number of arguments passed to a function is exactly the number needed by it. We compared TOOLDEC to two variants of the baseline in Hao et al. (2023), one with backtrace and one without. Backtrace tries to avoid failed tool calls by allowing the LLM to go back and try the next probable token, in place of the failed tool call. To evaluate TOOLDEC, we report the average inference time per problem and tool error rates in addition to accuracy.

Next, to study how TOOLDEC can enable generalizable tool selection, we mimic ToolkenGPT's planning method and prompt LLM to generate a tool name. We fine-tune the embedding of a single special token <T> to represent all tools, reducing the size of extra vocabulary to 1. Once <T> is generated, a tool call begins. We prompt LLM to generate a tool name. The generation of this tool name is guided by an FSM constructed from a list of all available tools. This tool name is then plugged back into the context to start the generation of arguments. We show an example in Figure 6.

We tune the baseline and TOOLDEC on a small set (4) of seen tools and demonstrate that TOOLDEC doesn't need additional data and further fine-tuning to adopt unseen (9) tools.

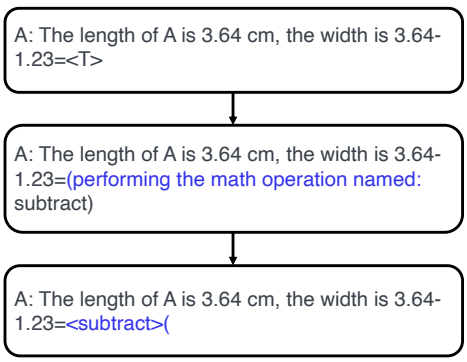

Figure 4: Once a tool call begins, TOOLDEC injects a special prompt (blue text) into context to generate the tool name.

3.3 Experimental Results

TOOLDEC Eliminates Syntax Errors. Table 2 shows the results on FuncQA$_{\text{multi}}$. Although ToolkenGPT eliminates the possibility of calling non-existent tool names by fine-tuning a special

Table 2: Results on $\text{FuncQA}_{\text{multi}}$. ToolkenGPT with TOOLDEC eliminated all tool errors. Compared to ToolkenGPT without backtrace, TOOLDEC was better in both accuracy and inference time. Compared to ToolkenGPT with backtrace, TOOLDEC achieved a comparable accuracy with only half the time.

	Accuracy ↑	Time ↓	Tool Error ↓
0-shot ChatGPT w/o tools	9%	-	-
LLaMA w/ tools + ReAct	6%	-	-
ToolkenGPT	10.3%	7.76s	27.9%
ToolkenGPT + Backtrace	**14.7%**	10.39s	0.0%
ToolkenGPT + TOOLDEC **(ours)**	13.2%	**5.95s**	**0.0%**

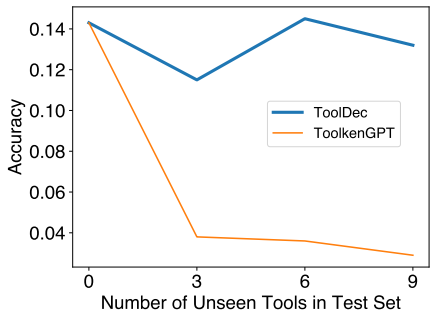

Figure 5: As the number of unseen tools increased, TOOLDEC kept a similar high performance.

token embedding, it can still suffer from other syntactic errors, which is demonstrated by the 27.9% tool error rate. As a drop-in replacement, TOOLDEC increased ToolkenGPT's accuracy while being much faster in inference. We observed that the short inference time was mainly due to TOOLDEC's role in preventing the generation of invalid tokens in the argument list, which could put the model in a confused state. We show examples in Appendix A.3. Although ToolkenGPT + backtrace achieved slightly better accuracy than TOOLDEC, it used 2x more time to try different tools. Note that since TOOLDEC eliminated all tool errors, there were no failed tool calls for backtrace to retry.

TOOLDEC enables generalizable tool selection. In Figure 5, we present the results on FuncQA. While ToolkenGPT and TOOLDEC achieved similar accuracies on tasks that involved only seen tools, ToolkenGPT failed to generalize to unseen tools, resulting in a significant performance drop. On the other hand, TOOLDEC reduces the token space to only prefixes of valid functions, allowing LLMs to exploit the semantics of function names. Consequently, TOOLDEC maintains a comparable accuracy even on unseen tools and achieve 8x better accuracy on multi-hop problems.

4 Conclusion

This paper presents TOOLDEC, a novel decoding algorithm designed to enhance Large Language Models (LLMs) by integrating external tools and ensuring their invocation is syntax-error-free. TOOLDEC, guided by a finite-state machine constructed from tool documentation, accurately represents the grammar of tool calls, addressing prevalent issues like erroneous tool calls and poor generalization to unseen tools in existing models. It also exhibits the ability to generalize to unseen tools without additional fine-tuning data.

The advancements by TOOLDEC open avenues for research in developing more sophisticated models adaptable to a wider range of tools and applications without additional training data, leading to more versatile and robust LLMs capable of solving a broader spectrum of complex problems. The success of TOOLDEC in eliminating syntax errors can inspire research focusing on semantic accuracy and contextual relevance of tool calls. This can lead to models that invoke, understand, and leverage tools more effectively, enhancing LLMs' overall problem-solving capabilities.

References

Peter Anderson, Basura Fernando, Mark Johnson, and Stephen Gould. Guided open vocabulary image captioning with constrained beam search. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pp. 936–945, Copenhagen, Denmark, September 2017. Association for Computational Linguistics. doi: 10.18653/v1/D17-1098. URL https://aclanthology.org/D17-1098.

Sebastian Borgeaud, Arthur Mensch, Jordan Hoffmann, Trevor Cai, Eliza Rutherford, Katie Millican, George Bm Van Den Driessche, Jean-Baptiste Lespiau, Bogdan Damoc, Aidan Clark, et al. Improving language models by retrieving from trillions of tokens. In *International conference on machine learning*, pp. 2206–2240. PMLR, 2022.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

Wenhu Chen, Xueguang Ma, Xinyi Wang, and William W Cohen. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks. *arXiv e-prints*, pp. arXiv–2211, 2022.

Jason Eisner. Parameter estimation for probabilistic finite-state transducers. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, pp. 1–8, 2002.

Edward Fredkin. Trie memory. *Communications of the ACM*, 3(9):490–499, 1960.

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

Kelvin Guu, Kenton Lee, Zora Tung, Panupong Pasupat, and Mingwei Chang. Retrieval augmented language model pre-training. In *International conference on machine learning*, pp. 3929–3938. PMLR, 2020.

Shibo Hao, Tianyang Liu, Zhen Wang, and Zhiting Hu. Toolkengpt: Augmenting frozen language models with massive tools via tool embeddings. *arXiv preprint arXiv:2305.11554*, 2023.

Chris Hokamp and Qun Liu. Lexically constrained decoding for sequence generation using grid beam search. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 1535–1546, Vancouver, Canada, July 2017. Association for Computational Linguistics. doi: 10.18653/v1/P17-1141. URL https://aclanthology.org/P17-1141.

Ximing Lu, Peter West, Rowan Zellers, Ronan Le Bras, Chandra Bhagavatula, and Yejin Choi. Neurologic decoding:(un) supervised neural text generation with predicate logic constraints. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pp. 4288–4299, 2021.

Ximing Lu, Sean Welleck, Peter West, Liwei Jiang, Jungo Kasai, Daniel Khashabi, Ronan Le Bras, Lianhui Qin, Youngjae Yu, Rowan Zellers, et al. Neurologic a* esque decoding: Constrained text generation with lookahead heuristics. In *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pp. 780–799, 2022.

Grégoire Mialon, Roberto Dessì, Maria Lomeli, Christoforos Nalmpantis, Ram Pasunuru, Roberta Raileanu, Baptiste Rozière, Timo Schick, Jane Dwivedi-Yu, Asli Celikyilmaz, et al. Augmented language models: a survey. *arXiv preprint arXiv:2302.07842*, 2023.

Ning Miao, Hao Zhou, Lili Mou, Rui Yan, and Lei Li. Cgmh: Constrained sentence generation by metropolis-hastings sampling. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pp. 6834–6842, 2019.

Reiichiro Nakano, Jacob Hilton, Suchir Balaji, Jeff Wu, Long Ouyang, Christina Kim, Christopher Hesse, Shantanu Jain, Vineet Kosaraju, William Saunders, et al. Webgpt: Browser-assisted question-answering with human feedback. *arXiv preprint arXiv:2112.09332*, 2021.

Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. Training language models to follow instructions with human feedback. *Advances in Neural Information Processing Systems*, 35: 27730–27744, 2022.

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

Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, et al. Toolllm: Facilitating large language models to master 16000+ real-world apis. *arXiv preprint arXiv:2307.16789*, 2023.

Pushpendre Rastogi, Ryan Cotterell, and Jason Eisner. Weighting finite-state transductions with neural context. In *Proceedings of the 2016 conference of the North American chapter of the Association for Computational Linguistics: human language technologies*, pp. 623–633, 2016.

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

Yongliang Shen, Kaitao Song, Xu Tan, Dongsheng Li, Weiming Lu, and Yueting Zhuang. Hugginggpt: Solving ai tasks with chatgpt and its friends in hugging face, 2023.

Yifan Song, Weimin Xiong, Dawei Zhu, Wenhao Wu, Han Qian, Mingbo Song, Hailiang Huang, Cheng Li, Ke Wang, Rong Yao, Ye Tian, and Sujian Li. Restgpt: Connecting large language models with real-world restful apis, 2023.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. Llama: Open and efficient foundation language models, 2023.

A Appendix

A.1 Related Works

Fine-tuning language models to use tools. Language models can be fine-tuned to use tools with data that contain interleaving text and tool use. Earlier studies make language models use a single tool like a retrieval module (Borgeaud et al., 2022; Guu et al., 2020) or a search engine (Nakano et al., 2021) by fine-tuning. Recent advances in tool-augmented language models that use multiple tools (Schick et al., 2023; Parisi et al., 2022) also fine-tune language models to use tools including QA models, translation models, calculators, and search engines. ToolkenGPT (Hao et al., 2023) proposes to use several special tokens to represent tools and only tunes the embeddings of the tokens so that new tool adoption can be more efficient. However, fine-tuning approaches for tool use still need new data and extra fine-tuning to adapt a model to new tools. We list the differences between finite-state decoding and the previous two paradigms in **??**.

In-context learning for tool use. Language models can learn from in-context examples (Brown et al., 2020) and follow instructions (Ouyang et al., 2022). This makes it possible to simply put the descriptions of tools in the prompt and ask language models to use them. Recent works have used this possibility to use neural models (Shen et al., 2023), RESTful APIs (Qin et al., 2023; Song et al., 2023), program interpreters (Chen et al., 2022; Gao et al., 2023) and many other tools to solve problems. In-context learning does not need extra model tuning to use new tools. However, the description and documentation of new tools still need to be in the prompt, which increases computation cost and limits the context budget for the model to actually reason about the task.

Constrained decoding and finite-state machines. Previous constrained decoding methods mainly focus on *lexical* constraints (Anderson et al., 2017). They reduce the large search space of lexically constrained decoding with finite-state machines (Anderson et al., 2017), grouping together similar candidates (Hokamp & Liu, 2017), and better search algorithms (Miao et al., 2019; Lu et al., 2021, 2022). However, lexical constraints are not expressive enough to regulate tool calls. While finite-state machines have to be weighted and probabilistic to deal with the soft constraints in natural language (Eisner, 2002; Rastogi et al., 2016), the constraints for syntactic tool calls are hard constraints that are much easier for FSMs. Therefore, we propose TOOLDEC to meet the *syntactic* constraints of a valid tool call.

A.2 Pseudo-Code of the Decoding Algorithm

Algorithm 1 Finite-State Machine Guided Decoding for Language Models

Input:
A DFSM defined by (S, V, g, s_0, R);
A language model M that produces the distribution of the next token given a prefix string;
An initial string of tokens $x_{1..k}$, which represents the prompt from the user.
Output: A string of tokens.

$s \leftarrow s_0$
while $s \notin F$ **do**
  $V_s \leftarrow \{a | a \in V \wedge g(s, a) \text{ is defined}\}$
  $P(x_{k+1}|x_{1..k}) \leftarrow M(x_{1..k})$
  $\tilde{P}(x = a|x_{1..k}, s) \leftarrow \begin{cases} \frac{P(x=a|x_{1..k})}{\sum_{a' \in V_s} P(x=a'|x_{1..k})}, & a \in V_s \\ 0, & \text{otherwise} \end{cases}$
  $x \sim \tilde{P}(x|x_{1..k}, s)$
  $x_{k+1} \leftarrow x$
  $k \leftarrow k + 1$
  $s \leftarrow g(s, x)$
end while
return $x_{1..k}$

A.3 Examples of TOOLDEC Eliminating Tool-related Errors

In this section, we show examples of TOOLDEC preventing tool-related errors on ToolkenGPT. ToolkenGPT responses are displayed in the left column and TOOLDEC is showed on the right.

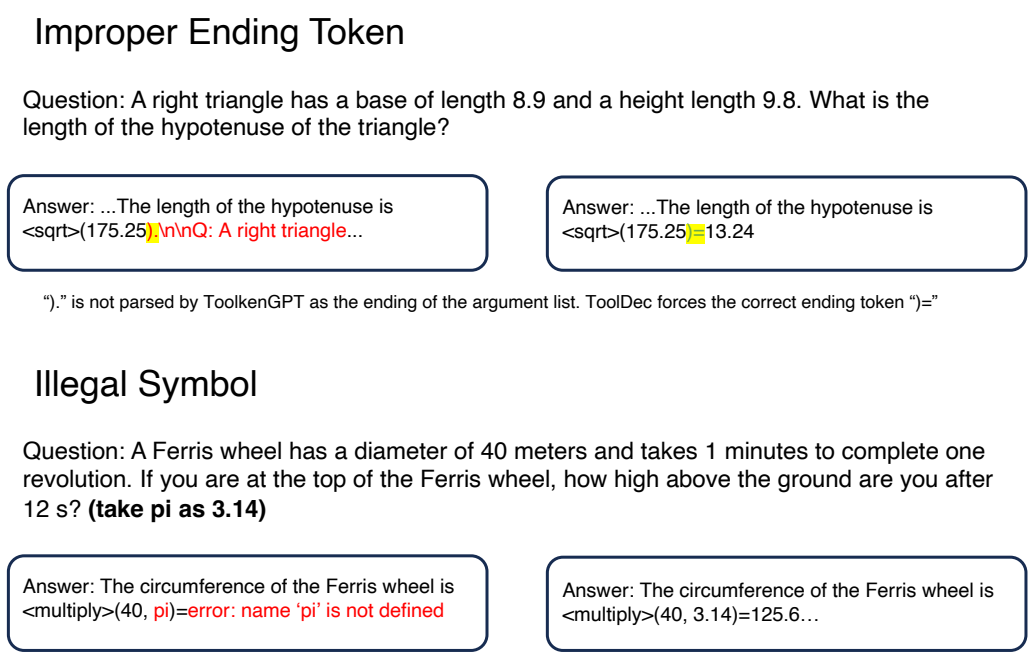

Figure 6: TOOLDEC can eliminate the common tool-related errors for fine-tuned models.