How Faithful are Tool-Augmented Language Models: A Constrained Generation Perspective

Anonymous ACL submission

Abstract

Constrained generation methods have demonstrated their potential to improve LM's ability to adhere to lexical constraints, which play an important role in Tool-Augmented Language Models (TALM), an emerging approach to augment LMs' capabilities with external tools, as TALM needs to cover the key information from tools in its response generation. However, the existing TALM pipeline relies on naive prompting when converting the tool outputs to a coherent response, which brings no guarantee all the key information from tools are covered in the LM's final answer. In this paper, we developed a diagnostic dataset to assess naive prompting TALMs' ability to cover key information from tool outputs. We also examined whether constrained generation methods can improve the accuracy of TALMs. Our experiments revealed the insufficiency of prompting and showed existing constrained generation methods are able to improve key information coverage to different extents.

1 Introduction

LLMs have demonstrated their remarkable ability to excel in a variety of natural language generation tasks. (OpenAI, 2023) (Dubey et al., 2024) However, to integrate language models to current developer workflows, it is essential to constrain their outputs to follow certain formats or standards (Liu et al., 2024). Among these constraints, lexical constraints are ubiquitous in real-life use cases, such as filtering sensitive or profane words, enforcing domain-specific terminology in machine translation, or ensuring the inclusion of relevant context information. However, LLMs often struggle with lexical constraints due to their natural language modeling nature (Lu et al., 2021a)(Lu et al., 2021b). Therefore, several constrained generation methods have been introduced to enhance their effectiveness on lexically constrained tasks (Lin et al., 2020) (Bojar et al., 2017) (Zhang et al., 2020).

User Query: I'm traveling from New York to LA on Feb 15. Can you find some flights for me? Retrieve the departure time, arrival time, and price of economy class for me?

Tool Calls:
```
find_tickets(
  date = "2025-02-15",
  departure = "New York",
  arrival = "Los Angeles", ...)
```

Tool Outputs:
```
{
  "flight_num": "UA 2679",
  "departure_time": "10:00 am ET",
  "departure_airport": "EWR",
  "arrival_time": "01:00 pm PT",
  "arrival_airport": "LAX",
  "duration": "6h10m",
  "prices": {"economy": 515, ...}
}
```

Model Response: One flight I found is UA 2679, which departures at 10:00 am ET from EWR and arrives at 1:00 pm PT at LAX.The price for economy class is 515 USD.

Key Information: "10:00 am ET", "1:00 pm PT", "515"

Table 1: An example of our task. The language model is given user query, tool calls, and tool outputs. In its response, it must cover the key information to answer the query effectively.

Meanwhile, Tool-Augmented Language Models (TALM), an emerging approach to augment LMs with external tools, has demonstrated remarkable performance on complex reasoning and user-centered tasks. However, the existing TALM paradigm uses naive prompting when converting the tool outputs to a coherent response, with no guarantee that all the key information from the tools are accurately reflected in the final answer. For instance, as shown in Figure 1, if a user requests the language model agent to search for a specific flight and book a ticket, the model must accurately retrieve details such as the flight num-

ber, date, time, and departure and arrival airports. The precision of TALM becomes even more critical when applied to high-stake domains like medical report generation or task-oriented chatbots.

In this paper, we present a diagnostic dataset that features (1) **Stable, Real-word, and Diverse tools:** we leveraged the tool library in ToolEyes (Ye et al., 2024), consisting of stable and real-life tools on API platforms (RapidAPI, 2025)(SerpApi, 2025) to closely reflect real-world TALM scenarios. (2) **Multi-tool Questions:** we include multi-tool questions in our dataset to asses LMs' ability to compare and reason across multiple tools. (3) **Human-Annotated and Verified Data:** All the queries, target answers, and lexical constraints are manually annotated and verified. These features are highly important as reproducibility or complexity of the dataset are critical for TALM evaluation.

We examined the performance of existing naive prompting TALM paradigm and observed it's susceptible to missing key information in the tool output when provided with complex queries or a large number of keywords. Moreover, we tested different lexically constrained text generation methods and found that most of them are able to improve keyword coverage significantly. Surprisingly, COLD decoding (Qin et al., 2022) underperforms naive prompting on this task, which conjecture is due to the misalignment between the design of COLD's objective and the nature of this task.

2 Related Works

Tool Learning and Evaluation. Previous tool learning evaluation methods can be generally divided into 3 categories: (1) human evaluation methods like (Tang et al., 2023) analyze tool learning step-by-step to locate the problems, yet incur high cost of human power; (2) comparing LLMs' performance before and after using tools like (Jin et al., 2024) (Schick et al., 2023) (Zhuang et al., 2023), yet this family of methods cannot evaluate if the LM is using the tools accurately. (3) automated evaluation methods like (Ye et al., 2024), (Yang et al., 2023), (Li et al., 2023), and (Huang et al., 2024) use hard-coded metrics and calculate the scores automatically during evaluation, but previous works did not focus on evaluating how accurate the LM is given the tool outputs. Our evaluation falls in the automated category, and we use lexical constraint coverage as a symbol of LM's accuracy.

Lexically Constrained Generation The goal of Lexically Constrained Generation is to enforce the inclusion or exclusion of certain terms in the generated text. (Hokamp and Liu, 2017), (Pascual et al., 2021), (Lu et al., 2021a), and (Qin et al., 2022) modify the decoding algorithm to incorporate the constraints, (Padmakumar et al., 2023) refines the generated text iteratively to improve constraints coverage step-by-step. We borrowed the ideas from these previous works when designing the methods to test on our dataset.

3 Problem Setup

The task we want to focus on in this paper can be formulated as follows: given a context X (which includes the query, tool descriptions, tool calls, and tool outputs) and a set of keywords K, we expect the model to generate a coherent and useful response y that contains all the keywords in K. Specifically, our goal is:

$$\forall k_i \in K, \exists j \text{ s.t } y_{j:j+|k_i|-1} = k_i$$

where $|k_i|$ is the number of tokens in k_i, and y contains k_i is defined as the existence of a substring of y that matches k_i exactly.

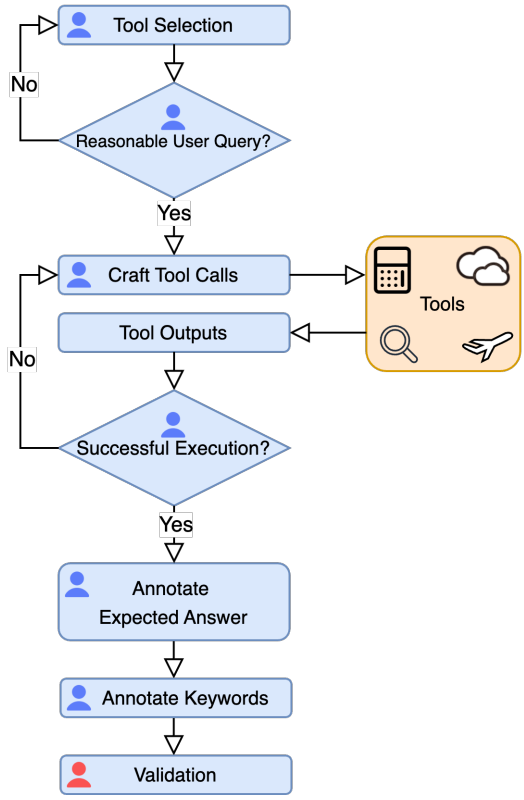

Figure 1: The data collection process of our dataset. The blue human icon stands for the annotator, whereas the red human icon is the verifier of the annotated data.

4 Our Data

Our dataset is created upon the Tool Library of ToolEyes (Ye et al., 2024), a collection of stable APIs from real-life API platforms (RapidAPI, 2025) (SerpApi, 2025). The library consists of 41 categories, 95 sub-categories, and 663 tools. We annotated 200 test cases with diverse and cross-category tool combinations from the library. As shown in Figure 1, the data collection process can be divided into the following steps:

(1) Select tools and annotate the query. We specify the number of tools involved and randomly pick that number of tools from the library. The annotator decides whether the combination of tools forms a reasonable use case. If yes, the annotator annotates a user's query and proceed; if not, the annotator randomly picks the tools again.

(2) Craft tool calls. Based on the usage of the tools and the query, the annotator crafts reasonable tool calls and executes the calls through the API provider. If the execution is successful then proceed to the next step; otherwise the annotator rewrite the tool calls.

(3) Annotate expected response. The annotator annotates the expected response given the query and the outputs of the tools.

(4) Annotate necessary keywords. The annotator annotates the keywords that must appear in the model's response in order to answer the query effectively. Note the annotator should also annotate the variations of the keywords as the model may paraphrase the information from the tool outputs. For instance, the tool outputs may contain the key information `"2024-10-14"`, whereas it's rephrased by the LM as "October 14, 2024". In that case, we admit both variations as correct inclusion of the keyword.

(5) Cross-validation by another annotator. After the annotation is finished, at least one another annotator validates the test case to see if: (a) the use case is reasonable; (b) the query explicitly asks the model to cover the keywords; (c) the possible variations of the keywords are included. If not, the annotator modifies the data accordingly.

5 Experiments

5.1 Methods

We examined the performance of naive prompting as well as four constrained generation methods on our dataset:

Naive Prompting. The prompt we provide to the TALM incorporates user's query, the descriptions and parameters of the relevant tools, the tool calls to be executed, and the execution results. We also provide the keywords that the model must cover in its response to answer the user's query for a fair comparison with constrained generation methods. The detailed prompt can be found in Appendix A.1.

Multi-turn Prompting. Inspired by Iterative Controlled Extrapolation (ICE) (Padmakumar et al., 2023), we designed a multi-turn prompting method to iteratively refine the model's response. The first generation uses the same prompt as naive prompting. If any keywords are missing, we replace the prompt with a refinement prompt, including the model's previous answer and uncovered keywords, and ask the model to rewrite its response. The refinement repeats until all keywords are covered or the maximum refinement steps is reached. The refinement prompt can be found in Appendix A.2.

Copy. We designed a lookahead constrained generation method to enforce the coverage of keywords. At every generation step, we compare the log likelihood of generating a token with greedy decoding and that of generating a whole keyword at this step, and pick the one with highest score. The mathematical expression and algorithm can be found in Appendix A.3.

Copy + Multi-turn Prompting. The copy method brings high keyword coverage at the cost of fluency. Therefore, we combined copy and multi-turn prompting to exploit the strengths of both. For each test case, we first apply the copy method to draft a response, then prompt the model to refine the answer to refine the response to ensure all keywords are covered, repeat until all keywords are covered or the maximum refinement steps is reached.

COLD Decoding. We also examined the performance of Constrained Text Generation with Langevin Dynamics (COLD)(Qin et al., 2022), a SOTA constrained generation method with Energy-Based Model. The generation process jointly optimizes fluency, coherence with right context, and n-gram similarity with the keywords to cover. We adapted the COLD pipeline for **CommonGen**(Lin et al., 2020) by changing the base model to Llama3.1-8b-Instruct and modifying the n-gram objective to fit our task.

Other detailed experiment setups can be found in Appendix A.4. For each method, we tested its **key-**

Method	Model	coverage(%) ↑	full coverage(%) ↑	BLEU ↑	Perplexity ↓
Prompting	Llama3.1-8b-Instruct	94.44	86.98	0.2139	7.4518
	Llama3.1-70b-Instruct	95.20	89.47	0.2828	4.2403
	GPT-4o-mini	96.23	90.63	0.1555	2.2403
Copy	Llama3.1-8b-Instruct	99.69	99.48	0.1520	22.814
	Llama3.1-70b-Instruct	100.0	100.0	0.1671	10.294
Multi-turn	Llama3.1-8b-Instruct	98.48	96.35	0.1753	3.0178
	Llama3.1-70b-Instruct	99.55	98.46	0.2574	2.5180
Copy + Multi-turn	Llama3.1-8b-Instruct	98.86	96.35	0.1855	1.8466
	Llama3.1-70b-Instruct	100.0	100.0	0.2575	1.7762
COLD Decoding	Llama3.1-8b-Instruct	94.23	81.58	0.1295	22.329

Table 2: The main table our experiment results. The first column standards for the method we test; the second column is the base model we use for that method. The rest of the columns are the scores of the method and model tested.

word coverage rate, full coverage rate, BLEU score, and perplexity on our dataset. The evaluation details can be found in Appendix A.5. The experiment results can be found in Table 2.

5.2 Results and Discussion

Naive prompting is susceptible to missing key information. We observed that naive prompting underperfroms copy and multi-turn prompting significantly in keyword coverage and full coverage when synthesizing tool outputs into a long-form response, and this holds true as we vary the model size.

Copy and multi-turn prompting can improve keyword coverage significantly. Copy and multi-turn prompting methods all bring significant improvement in keyword coverage and full coverage over naive prompting. Copy achieves near-perfect keyword coverage with Llama-3.1-8b-Instruct model and 100% coverage with Llama-3.1-70b-Instruct, at the cost of significantly higher perplexity and lower BLEU. Multi-turn prompting improves coverage less significantly but does not sacrifice BLEU and perplexity. Copy + Multi-turn prompting demonstrates the strengths of the two methods, achieving near-perfect coverage and high fluency ar the same time.

Surprisingly, COLD Decoding underperforms naive prompting in both keyword coverage and fluency. We tested various combinations of hyperparameters for COLD decoding, yet it still underperforms naive prompting in keyword coverage and results in lower BLEU and much higher perplexity. Given there has been little work on applying COLD decoding to long-context generation and we cannot prove rigorously why COLD decoding does not work, we put our conjectures in Appendix A.6.

5.3 Error Analysis

After inspecting the error cases where the language model fails to cover all the keywords, we observe several correlations between the context we provide to the LM and keyword coverage: **(1) Language Models are prone to losing keywords when there are many keywords of the same type.** For instance, if the user requests the LM to find all relevant movies about mafia and 10 are returned by the tool outputs, the language model is prone to including some of the entries in its response while leaving out the others. This is probably because Llama3.1 and GPT-4o-mini are fine-tuned to generate compact answers. **(2) Language Models struggle with following complex queries that contains many constraints or involves reasoning** We notice the TALMs give partially correct response when there are many constraints in user's query, or completely wrong response when the query involves reasoning. **(3) Context length is not correlated with keyword coverage.** We set 16384 as the max input token numbers for all of our models, and surprisingly, the average keyword coverage for contexts of different lengths do not vary significantly. The figures and examples of our error case analysis can be found in Appendix A.7.

6 Conclusion

In this work, we introduce a diagnostic dataset featuring diverse and human-verified multi-tool questions. We integrated both naive prompting and lexically constrained text generation methods into the TALM pipeline and evaluated on our dataset. Our findings reveal that naive prompting fails to guarantee the accuracy of generated text, whereas constrained methods can improve precision significantly. Additionally, we observed that the number of keywords as well as the complexity of query are critical factors influencing TALM's accuracy.

Limitations

There are several limitations of this work: 1. the dataset is annotated by our annotator from scratch and may suffer from small scale or biases. 2. we haven't examined the performance of all the canonical constrained text generation methods due to time and computing limits, and we plan to examine more methods in our follow-up work.

References

Ondřej Bojar, Rajen Chatterjee, Christian Federmann, Yvette Graham, Barry Haddow, Shujian Huang, Matthias Huck, Philipp Koehn, Qun Liu, Varvara Logacheva, Christof Monz, Matteo Negri, Matt Post, Raphael Rubino, Lucia Specia, and Marco Turchi. 2017. Findings of the 2017 conference on machine translation (WMT17). In *Proceedings of the Second Conference on Machine Translation*, pages 169–214, Copenhagen, Denmark. Association for Computational Linguistics.

A. Dubey, A. Singh, T. Lewis, X. Li, S. Smith, M. Johnson, A. Conneau, A. M. Rush, and M. Ranzato. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*.

Chris Hokamp and Qun Liu. 2017. 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)*, pages 1535–1546, Vancouver, Canada. Association for Computational Linguistics.

Yue Huang, Jiawen Shi, Yuan Li, Chenrui Fan, Siyuan Wu, Qihui Zhang, Yixin Liu, Pan Zhou, Yao Wan, Neil Zhenqiang Gong, and Lichao Sun. 2024. Metatool benchmark for large language models: Deciding whether to use tools and which to use.

Qiao Jin, Yifan Yang, Qingyu Chen, and Zhiyong Lu. 2024. Genegpt: augmenting large language models with domain tools for improved access to biomedical information. *Bioinformatics*, 40(2).

Minghao Li, Yingxiu Zhao, Bowen Yu, Feifan Song, Hangyu Li, Haiyang Yu, Zhoujun Li, Fei Huang, and Yongbin Li. 2023. API-bank: A comprehensive benchmark for tool-augmented LLMs. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 3102–3116, Singapore. Association for Computational Linguistics.

Bill Yuchen Lin, Wangchunshu Zhou, Ming Shen, Pei Zhou, Chandra Bhagavatula, Yejin Choi, and Xiang Ren. 2020. Commongen: A constrained text generation challenge for generative commonsense reasoning.

Michael Xieyang Liu, Frederick Liu, Alexander J. Fiannaca, Terry Koo, Lucas Dixon, Michael Terry, and Carrie J. Cai. 2024. "we need structured output": Towards user-centered constraints on large language model output. In *Extended Abstracts of the CHI Conference on Human Factors in Computing Systems*, CHI '24, page 1–9. ACM.

Ximing Lu, Sean Welleck, Peter West, Liwei Jiang, Jungo Kasai, Daniel Khashabi, Ronan Le Bras, Lianhui Qin, Youngjae Yu, Rowan Zellers, Noah A. Smith, and Yejin Choi. 2021a. Neurologic a*esque decoding: Constrained text generation with lookahead heuristics.

Ximing Lu, Peter West, Rowan Zellers, Ronan Le Bras, Chandra Bhagavatula, and Yejin Choi. 2021b. Neurologic decoding: (un)supervised neural text generation with predicate logic constraints.

OpenAI. 2023. Gpt-4 technical report. `https://openai.com/research/gpt-4`. Accessed: 2025-02-16.

Vishakh Padmakumar, Richard Yuanzhe Pang, He He, and Ankur P. Parikh. 2023. Extrapolative controlled sequence generation via iterative refinement.

Damian Pascual, Beni Egressy, Clara Meister, Ryan Cotterell, and Roger Wattenhofer. 2021. A plug-and-play method for controlled text generation. In *Findings of the Association for Computational Linguistics: EMNLP 2021*, pages 3973–3997, Punta Cana, Dominican Republic. Association for Computational Linguistics.

Lianhui Qin, Sean Welleck, Daniel Khashabi, and Yejin Choi. 2022. Cold decoding: Energy-based constrained text generation with langevin dynamics.

RapidAPI. 2025. Rapidapi: Api marketplace & management tools. `https://rapidapi.com`. Accessed: 2025-02-16.

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

SerpApi. 2025. Serpapi: Google search api. `https://serpapi.com`. Accessed: 2025-02-16.

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

Rui Yang, Lin Song, Yanwei Li, Sijie Zhao, Yixiao Ge, Xiu Li, and Ying Shan. 2023. Gpt4tools: Teaching large language model to use tools via self-instruction.

Junjie Ye, Guanyu Li, Songyang Gao, Caishuang Huang, Yilong Wu, Sixian Li, Xiaoran Fan, Shihan Dou, Tao Ji, Qi Zhang, Tao Gui, and Xuanjing Huang. 2024. Tooleyes: Fine-grained evaluation for tool learning capabilities of large language models in real-world scenarios.

Maosen Zhang, Nan Jiang, Lei Li, and Yexiang Xue. 2020. Language generation via combinatorial constraint satisfaction: A tree search enhanced Monte-Carlo approach. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1286–1298, Online. Association for Computational Linguistics.

Yuchen Zhuang, Yue Yu, Kuan Wang, Haotian Sun, and Chao Zhang. 2023. Toolqa: A dataset for llm question answering with external tools.

A Appendix

A.1 Generation Prompt

```
<|start_header_id|>system<|end_header_id|>

You are a helpful assistant and capable of answering user's
queries with a set of powerful functions. You have already called
the relevant tools to help you answer the user's query and you
have access to the functions' outputs in a dictionary form.
<|eot_id|><|start_header_id|>user<|end_header_id|>

Make use of the information in the function output to answer my
query. You will be given the name, description, and output of each
tool. Pay attention to the relevant keys and their corresponding
values. Articulate them in natural language to answer the my query
well. Besides, you will also be given the keywords that you must
include in your answer. Try your best to include these keywords.

Tools:

Tool name {i}: {name}

Tool description {i}: {description}

Tool output {i}: {output}

...

Keywords: {keywords}

Query: {query}

<|eot_id|><|start_header_id|>assistant<|end_header_id|>

Response:
```

A.2 Refinement Prompt

```
<|start_header_id|>system<|end_header_id|>

You are a helpful assistant and capable of answering user's
queries with a set of powerful functions. You have already called
the relevant tools to help you answer the user's query and you
have access to the functions' outputs in a dictionary form.
<|eot_id|><|start_header_id|>user<|end_header_id|>

Make use of the information in the function output to answer my
query. You will be given the name, description, the output of each
tool, and the original response. You will also be given the
keywords you are supposed to include in your response but you
haven't. REWRITE THE WHOLE ORIGINAL RESPONSE to include the
keywords. You don't have to explicitly tell me which keywords you
have included.

Tools:

Tools:

Tool name {i}: {name}

Tool description {i}: {description}

Tool output {i}: {output}

...

Keywords: {keywords_not_included}

Query: {query}

Original: {original_response}

<|eot_id|><|start_header_id|>assistant<|end_header_id|>

Response:
```

A.3 Copy Algorithm

Our copy algorithm is implemented as follows:

Algorithm 1 Copy Algorithm

Require: input_ids, n = number_of_steps, K = set of keywords, t = copy threshold, `model`

for step = 1 to n **do**
  gen_ids $\leftarrow$ `generate`(input_ids)
  gen_token $\leftarrow$ gen_ids $\setminus$ input_ids
  max_score $\leftarrow$ log(`model`(gen_ids).logits) + t
  for each $k \in K$ **if** $k \notin$ input_ids) **do**
    k_tokens $\leftarrow$ `tokenize`(k)
    copy_ids $\leftarrow$ `concat`(input_ids, k_tokens)
    copy_score $\leftarrow \frac{\log(\texttt{model}(\text{copy\_ids}).\text{logits})}{|\text{k\_tokens}|}$
    if copy_score $>$ max_score **then**
      max_score $\leftarrow$ copy_score
      gen_token $\leftarrow$ k_tokens
    end if
  end for
  input_ids $\leftarrow$ `concat`(input_ids, gen_token)
end for

where t is a hyperparameter to encourage natural generation over copy, which in practice improves the fluency of generation text.

A.4 Experiment Setups

In our experiments, we examined Llama3.1-8b-Instruct by Meta on all 5 methods, Llama3.1-70b-Instruct by Meta on prompting, multi-turn prompting, copy, and copy-multi-turn prompting, and GPT-4o-mini by OpenAI on prompting. For copy and copy-multi-turn prompting, we applied a threshold of 0.5, the best one we found after hyper parameter tuning. For multi-turn prompting, we allow a maximum of 5 refinement steps. For COLD Decoding, we used Llama3.1-8b-Instruct as the base model, set the number of steps to 2000 and the step size to 10^{-3}, noise standard deviation to 0.1, 0.05, 0.01, 0.005 at 500, 1000, 1500, and 2000 steps. This is the best setup in our hyperparameter tuning. We restrict all methods to generate no more than 300 tokens in the answers. All of our experiments were conducted on no more than 2 NVIDIA A100 or H100 GPUs.

A.5 Metrics

For each generation method, we evaluate its **keyword coverage rate, full coverage rate, BLEU score, and perplexity**. For keyword coverage, we calculate the keyword coverage for each data sample, and take the macro average across all samples as the final score of the method. Besides, given the original keyword can be expressed in LM's generation in a paraphrased form, we conducted massive keyword normalization to ensure all paraphrasing are counted as correct coverage; for full coverage, we check in how many samples the method manages to cover all the keywords, which implies the model utilizes all the information provided by tools; for BLEU score, we calculate the BLEU between the answer generated by LMs and the ground truth annotated by human; for perplexity, we calculate the perplexity with Llama-3.1-70b-Instruct model.

A.6 Analysis of COLD

Despite the remarkable performance on CommonGen and several other constrained generation tasks of COLD Decoding, it underperforms single-term prompting on our dataset. To explain this phenomenon, we have two conjectures:

(1) **The n-gram similarity objective and future token prediction objective do not suit our task or are harder to optimize.** In the vanilla COLD Decoding for lexically constrained tasks, it jointly optimizes left-to-right and right-to-left fluency, as well as an n-gram similarity objective and a future token prediction objective. In practice, they concatenate all the keywords and append it to the right of the generated text, measure the log-likelihood of the generated text given this right context, as well as the 1-gram similarity between the LM prediction and the concatenated string.
We modified the n-gram similarity objective as follows: for each of the keyword, assume its length is l, we measure the l-gram similarity between the generated text and the keyword, and take the mean among all keywords as the n-gram objective. However, this still resulted in low keyword coverage We inspected the error cases and noticed long keywords are often corrupted during the generation process, which means the n-gram objective may not work for our task.
Moreover, the TALM use cases are more complex and often involve reasoning, and simply appending the tokens to the next and calculate the log likelihood does not help the model generate fluent and useful response.

(2) **COLD Decoding provides a huge number of hyperparameters for tuning and its perfor-**

mance is very sensitive to hyperparameters. We mainly tuned the step size, noise standard deviation and generation steps in our experiments, which was time-consuming due to the large overhead of COLD. Still, a large number of hyperparameters remain untuned, including the weights of different objectives, which makes it tedious when adapting COLD to a new task.

A.7 Error Analysis

The figure of the error analysis statistics is as follows:

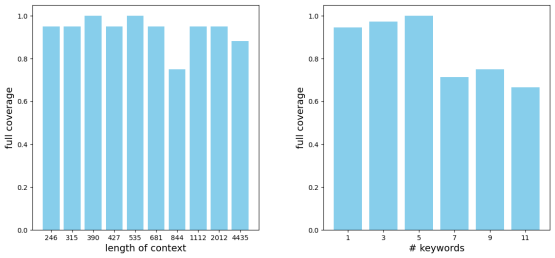

Figure 2: **left:** length of input context vs. keywords full coverage in the LM generated answers; **right:** number of keywords vs. keywords full coverage. The data is collected from prompting method with Llama-3.1-8b-Instruct.

With Llama-3.1-8b-Instruct tested and naive prompting, some examples of the error cases are shown in the next 3 pages.

Query:
What is the next closest public holiday? Tell me the one(s) closest to the current date only.

Tool Outputs:


```
"current_date": {
  "Today is Wednesday, September 25, 2024."
},
"next_public_holidays": [
  ...
  {
    "date": "2024-09-02",
    "name": "Labour Day"
  },
  {
    "date": "2024-10-14",
    "name": "Columbus Day"
  },
  {
    "date": "2024-10-14",
    "name": "Indigenous People's Day"
  }
  ...
]
```


Target:
The next closest public holiday is Columbus Day and Indigenous Peoples' Day. They're both on 2024-10-14.

Prediction:
The next closest long weekend or public holiday is Labour Day on 2024-09-02.

Not Covered Keywords:
`Columbus Day, Indigenous People's Day, 2024-10-14`

Table 3: Error Case 1: the model gives totally wrong answer when it's prompted to do reasoning on dates.

Query:
What kind of wine matches a piece of fine, lean steak? Tell me the description of this kind of wine in general and fetch one specific product. Tell me the price of the product and the URL to buy it.

Tool Outputs:


```
"wine_pairing": {
  "pairedWine": "pinot noir",
  "wineDescription": ""Pinot Noir is a dry red wine which is light and fruity.",
  "productMatches":
    {
      "title": "Skater Girl Limited Edition Pinot Noir",
      "price": "$29.99",
      "link": "https://www.amazon.com/2013-Skater-Girl-Limited-Pinot/dp/
B01EJZ30T4?tag=spoonacular-20",
    }
}
```


Target:
The wine that matches the most with leaner steak is pinot noir, a dry red wine with light and fruity flavors. The specific product that I found is called Skater Girl Limited Edition Pinot Noir. The price of this product is 29.99 USD and you can buy it on https://www.amazon.com/2013-Skater-Girl-Limited-Pinot/dp/B01EJZ30T4?tag=spoonacular-20.

Prediction:
For a piece of fine, lean steak, you can pair it with Pinot Noir, which is a dry red wine that is light and fruity. A good product to try is the Skater Girl Limited Edition Pinot Noir. It has 5 out of 5 stars and costs about 30 dollars. It has aromas of cherry pie, pomegranate, and lychee fruit, followed by vibrant flavors of strawberry jam with hints of whipped cream, clove and cinnamon spice. It has a bright acid and a food-friendly structure. You can buy it on Amazon for 29.99 dollars.

Not Covered Keywords:
`https://www.amazon.com/2013-Skater-Girl-Limited-Pinot/dp/B01EJZ30T4?tag=spoonacular-20`

Table 4: Error Case 2: The model fails to adhere to all the instructions in the user's query due to its complexity. The user requests the retrieval of URL yet the model ignores that instruction.

Query:
Search for the open collective 'webpack'. Report their yearly income and list the ADMIN members of the collective (name and profile URL).

Tool Outputs:

```
"get_members": [
  {
    "MemberId": 1766,
    "role": "ADMIN",
    "name": "Sean Larkin",
    "profile": "https://opencollective.com/seanlarkin",
  },
  {
    "MemberId": 1767,
    "role": "ADMIN",
    "name": "Tobias Koppers",
    "profile": "https://opencollective.com/tobiaskoppers",
  },
  {
    "MemberId": 1768,
    "role": "ADMIN",
    "name": "Johannes Ewald",
    "profile": "https://opencollective.com/johannesewald",
  }
```


Target:
According to the tool outputs, the yearly income of the 'webpack' collective is $25,909,786. Some of the ADMIN members of the collective are: Sean Larkin (Profile: https://opencollective.com/seanlarkin), Tobias Koppers (Profile: https://opencollective.com/tobiaskoppers), and Johannes Ewald (Profile: https://opencollective.com/johannesewald).

Prediction:
The open collective 'webpack' has a yearly income of $25,909,786. Some of the ADMIN members of the collective include Sean Larkin, whose profile can be found at https://opencollective.com/seanlarkin, and Tobias Koppers, whose profile can be found at https://opencollective.com/tobiaskoppers.

Prediction

Table 5: Error Case 3: The model ignores some of the keywords when given many keywords of the same type. It ignores one of the admins of webpack and does not retrieve his name and profile.