2025 2nd International Generative AI and Computational Language Modelling Conference (GACLM) | 979-8-3315-9406-0/25/$31.00 ©2025 IEEE | DOI: 10.1109/GACLM67198.2025.11232016

# Enhancing LLM Function Calling with Structured Outputs

Kevin Séjourné
*Cloud Temple*
Paris, France
kevin.sejourne@cloud-temple.com

Alexandru Lata
*Cloud Temple*
Paris, France
alexandru.lata@cloud-temple.com

***Abstract*—Large Language Models (LLMs) are increasingly used for function calling, typically by prompting the model to generate a specific format. This paper proposes an alternative approach leveraging structured outputs to enforce the desired format. This method offers several advantages, including a higher rate of format compliance and the ability to incorporate more complex elements than simple function calls, such as environment variables or preliminary remote connections. Furthermore, in text editing tasks, generative grammars (e.g., via xgrammar) can be employed to produce syntactically valid code directly. We present experimental results comparing our Constrained Generation (CG) approach (using structured outputs with xgrammar) against the traditional Post-Parsing (PP) method on the Hermes-Function-Calling-v1 dataset. The comparison is performed on tasks requiring valid JSON generation and valid function call generation, demonstrating the effectiveness of the proposed structured output methodology.**

***Index Terms*—Large Language Models, Function Calling, Structured Output, Constrained Generation, xgrammar, JSON Schema**

## I. Introduction

Large Language Models (LLMs) have demonstrated remarkable capabilities in understanding and generating human language, leading to their adoption in a wide array of applications. One such application is *function calling*, where LLMs are tasked to identify when to call external tools or APIs and to generate the necessary parameters in a structured format, typically JSON. The prevalent approach relies on instructing the LLM via system prompts to produce outputs adhering to a specified schema. While often effective, this method can suffer from inconsistencies, where the LLM fails to strictly follow the requested format, leading to parsing errors and unreliable behavior in downstream applications.

This paper introduces an alternative methodology that leverages *structured output* mechanisms to enhance the reliability and capabilities of LLM-based function calling. Instead of merely suggesting a format through prompting, our approach enforces the generation of syntactically correct structured data. This offers two key advantages:

- **Improved Format Adherence:** By constraining the generation process, we increase the likelihood of obtaining outputs that strictly conform to the desired schema.
- **Enhanced Capabilities:** Structured outputs allow for the representation of more complex interactions beyond simple function calls. This includes specifying environment variables, managing prerequisite remote connections, or defining sequences of actions.

We propose a method called Constrained Generation (CG), which utilizes xgrammar [10] for enforcing structured outputs. To evaluate its efficacy, we compare CG against the traditional Post-Parsing (PP) approach, where the LLM generates text that is subsequently parsed into a structured format. Our experiments are conducted on the Hermes-Function-Calling-v1 dataset [1], focusing on two key aspects: the generation of valid JSON objects and the generation of valid function calls. The results demonstrate the potential of structured outputs to build more robust and versatile function calling systems.

The remainder of this paper is organized as follows: Section II discusses related work in function calling and structured LLM outputs. Section III details our proposed Constrained Generation approach. Section IV presents the experimental setup and results. Finally, Section V concludes the paper and outlines future work.

## II. Related Work

The ability of Large Language Models (LLMs) to interact with external tools and APIs through function calling has significantly expanded their utility. Initial methods predominantly relied on sophisticated prompt engineering to coax LLMs into generating outputs that could be parsed into function calls [2]. Subsequent research explored fine-tuning LLMs specifically for tool invocation, aiming to improve their proficiency in selecting appropriate functions and formulating correct parameters [3]. However, ensuring the generation of syntactically valid and semantically accurate structured outputs, typically JSON, remains a persistent challenge.

The broader field of structured output generation from LLMs has seen considerable progress. Researchers have investigated various techniques to compel LLMs to adhere to predefined schemas. Liu et al. [4] surveyed industry professionals, highlighting the critical need for structured outputs and identifying numerous use cases. Benchmarks like SoEval, introduced by Liu et al. [5], have been developed to assess LLMs' capabilities in producing various structured formats. Li et al. [6] proposed a two-step approach, separating content generation from structuring, to improve information extraction tasks.

More direct methods for enforcing structure include grammar-based approaches. Technologies like xgrammar, as discussed in Geng et al. [7] within their JSONSchemaBench framework, and automata-based constraints as proposed by Koo et al. [8], aim to guide the LLM's generation process token by token. These methods ensure syntactical correctness by design, mitigating issues common in post-parsing approaches. However, as Zhang et al. [9] caution, output constraints themselves can introduce new attack surfaces if not carefully managed.

Our work is situated at the intersection of these research streams. We specifically apply grammar-constrained generation, using xgrammar, to the task of function calling. By doing so, we aim to harness the benefits of guaranteed syntactical validity for creating more reliable and robust function calling mechanisms.

## III. Proposed Approach: Constrained Generation (CG)

Our proposed approach, Constrained Generation (CG), aims to improve the reliability of function calling by LLMs by directly enforcing a structured output format during the generation process. This contrasts with the common Post-Parsing (PP) method, where the LLM generates relatively free-form text that is then parsed and validated against a target schema.

The core idea of CG is to use a formal grammar, specifically xgrammar, to guide the LLM's token selection at each step. This ensures that the generated output is syntactically valid according to the predefined structure (e.g., a JSON schema representing the function's parameters). Concretely, this constrained decoding process relies on transforming the JSON grammar (or JSON Schema) into a generative automaton. This automaton is then used at each step of token generation by the LLM. By analyzing the current state of the generation and the possible transitions of the automaton, a mask is applied to the model's output vocabulary. This mask only retains tokens that allow compliance with the grammar, thus ensuring that the generated sequence will be syntactically valid.

The process involves the following key steps:

1) **Schema Definition:** For each function that can be called, a corresponding JSON schema is defined. This schema precisely describes the expected parameters, their types, and any constraints (e.g., required fields, enumerated values).
2) **Grammar Compilation:** The JSON schema is compiled into an xgrammar. This grammar dictates the allowed sequences of tokens that the LLM can generate.
3) **Constrained Decoding:** During inference, the LLM's output is constrained by the compiled xgrammar. At each generation step, only tokens that conform to the grammar are considered, effectively preventing the model from producing syntactically incorrect outputs.

This method inherently guarantees that the output will be parsable and adhere to the specified structure, reducing the need for extensive error handling and retries often associated with the PP approach. Furthermore, it opens possibilities for more complex interactions, as the grammar can define not just function parameters but also control flow, environment variable settings, or sequences of operations.

Moreover, for models that benefit from an explicit reasoning step (or "chain-of-thought"), our constrained generation approach remains flexible. It is entirely possible to integrate dedicated fields for this reasoning directly into the JSON schema to be produced. For example, a `think` attribute can be defined as the first field of the JSON object, allowing the model to record its "reasoning" there before generating the final data fields. Similarly, fields like `explanation` or `caveat` can be included for the LLM to explain its choices or highlight points of attention. Since LLMs use their previously generated tokens as context for subsequent generation, including such fields in the constrained JSON structure does not disrupt the validity of the final JSON nor hinder the model's ability to produce correct results, while offering a trace of its decision-making process. Figure 1 illustrates an API call example for the CG method, where the JSON schema in `response_format` has been augmented with a `think` field for the `Qwen/Qwen3-30B-A3B-FP8` model (matching the model in the figure's example).

## IV. Experiments

To evaluate the effectiveness of our Constrained Generation (CG) approach, we conducted a series of experiments comparing it against the traditional Post-Parsing (PP) method. We chose xgrammar for these experiments because it is the best way for CG using vLLM [11], the high-performance open-source LLM-Engine.

### A. Dataset

We utilized the Hermes-Function-Calling-v1 dataset [1]. This dataset is specifically designed for evaluating function calling capabilities in LLMs and contains various scenarios requiring the generation of structured outputs. We focused on two main subsets of this corpus:

- **JSON Generation Tasks:** Instances where the primary goal is to produce a valid JSON object according to a given schema or description.
- **Function Call Generation Tasks:** Instances that require identifying the correct function to call from a set of available tools and generating the appropriate parameters in JSON format.

### B. Experimental Setup

Our experiments compare the traditional Post-Parsing (PP) baseline against our Constrained Generation (CG) approach using the `Llama-3.3-70B` and `Qwen3-30B` models, among others. For all tests, the maximum token generation was set to 16384, with temperature varied between 0.0 and 0.8.

We focused on two main tasks from the Hermes-Function-Calling-v1 dataset:

*1) JSON Generation:* This task assessed the models' ability to generate a valid JSON object from a natural language description.

```
{
  "model": "Qwen/Qwen3-30B-A3B-FP8",
  "messages": [
    {
      "role": "system",
      "content": "You are a helpful assistant that answers
          in JSON. Here's the json schema you must adhere
          to:\n<schema>\n{'properties': {'oreType':
          {'title': 'Ore Type', 'type': 'string'},
          'processedVolume': {'title': 'Processed Volume',
          'type': 'number'}, 'processingDate': {'format':
          'date', 'title': 'Processing Date', 'type':
          'string'}, 'processingFacility': {'title':
          'Processing Facility', 'type': 'string'}},
          'required': ['oreType', 'processedVolume',
          'processingDate', 'processingFacility'],
          'title': 'Ore Processing Data', 'type':
          'object'}\n</schema>\n"
    },
    {
      "role": "user",
      "content": "I'm currently overseeing the ore
          processing operations at our mining facility
          [...] The processed volume needs to be recorded
          in tons, and the date should be in the format
          YYYY-MM-DD."
    }
  ],
  ...
  "response_format": {
    "type": "json_schema",
    "json_schema": {
      "name":
          "hermes_043c9253-bad7-4924-bf5e-9ceb9ac3b055",
      "schema": {
        "properties": {
          "think": {
            "type": "string",
            "description": "Analyze user input and reason
                step by step before generating the final
                structured response. Allows the model to
                decompose the problem, identify key
                information, and plan its response."
          },
          "oreType": {
            "title": "Ore Type",
            "type": "string"
          },
          "processedVolume": {
            "title": "Processed Volume",
            "type": "number"
          },
          "processingDate": {
            "title": "Processing Date",
            "type": "string"
          },
          "processingFacility": {
            "title": "Processing Facility",
            "type": "string"
          }
        },
        "required": [
          "oreType",
          "processedVolume",
          "processingDate",
          "processingFacility"
        ],
        "title": "Ore Processing Data",
        "type": "object"
      }
    }
  }
}
```

Listing 1. Example of a CG API call using a JSON schema augmented with a 'think' field for the Qwen/Qwen3-30B-A3B-FP8 model. This allows for an explicit reasoning step before generating the structured output.

- **PP Method:** The LLM generated free-form text from the prompt, which was then parsed to find and validate a JSON block.
- **CG Method:** A JSON schema was automatically extracted and preprocessed from the prompt, then used with the API's structured output feature to enforce the generation format. Due to variability in the dataset's schema descriptions, 1884 of 1893 test cases were used.

*2) Function Call Generation:* This task required the LLM to select the correct function and generate its parameters.

- **PP Method:** Tool definitions were supplied to the API's native `tools` parameter, and evaluation relied on the structured `tool_calls` output.
- **CG Method:** Tool definitions were transformed into a comprehensive JSON schema using a custom script. This schema, representing a choice of function and its arguments, was then used to constrain the LLM's output. This approach relies on both schema enforcement and the LLM's ability to interpret the prompt, as the underlying grammar engine supports a subset of JSON Schema, sometimes requiring more tolerant definitions.

**Evaluation and Scope:** The primary metric was the rate of generating syntactically valid JSON, with semantic correctness also evaluated for function calls. We recorded API success, parsing errors, and duration. Our analysis focused on single-turn generation, as schema adherence is not directly impacted by dialogue history in this context. We plan to release a curated open-source corpus for these tasks.

### C. Results

Our experiments, conducted using the methodologies described above, yielded several key observations regarding the performance of Constrained Generation (CG) compared to Post-Parsing (PP). We didn't use models with specific fine-tuning.

As shown in Table I, the CG method consistently achieves a lower validation error rate across all temperatures and runs for both models. This directly supports the primary hypothesis that constrained generation improves format adherence.

Table II shows the error rates for the function call selection task, where only the choice of the function name is compared. Figure 1 shows the Levenshtein distance difference between the two methods and the corpus reference, this metric is used to estimate the correctness of the function parameters.

A consistent finding across both JSON generation and function call generation tasks is that the advantage of the CG method over PP becomes more pronounced under specific conditions:

- **Impact of Temperature:** Our analysis of the impact of temperature on generation quality reveals a nuanced picture. Contrary to a simple hypothesis that higher temperatures would disproportionately degrade the performance of the less constrained PP method, our results show that both CG and PP maintain a relatively stable level of semantic accuracy (as measured by Levenshtein distance

TABLE I
VALIDATION ERROR RATE BY TEMPERATURE FOR THE JSON GENERATION TASK. THIS TABLE COMPARES THE PERCENTAGE OF SYNTACTICALLY INCORRECT JSON OUTPUTS FOR THE CONSTRAINED GENERATION (CG) AND POST-PARSING (PP) METHODS ACROSS MULTIPLE RUNS AND TEMPERATURES.

| | Llama-3.3-70B | | Qwen3-30B-A3B | |
|---|---|---|---|---|
| **Temp** | **CG** | **PP** | **CG** | **PP** |
| **0.0** | | | | |
| Run 1 | 1.50% | 2.47% | 2.15% | 3.01% |
| Run 2 | 1.75% | 2.47% | 2.43% | 2.92% |
| Run 3 | 1.50% | 2.55% | 2.35% | 2.68% |
| Run 4 | 1.46% | 2.75% | 2.39% | 2.92% |
| Run 5 | 1.58% | 2.67% | 2.67% | 2.36% |
| **0.2** | | | | |
| Run 1 | 1.83% | 2.63% | 2.39% | 3.09% |
| Run 2 | 1.75% | 2.43% | 2.71% | 3.05% |
| Run 3 | 1.42% | 2.63% | 2.39% | 2.72% |
| Run 4 | 1.58% | 2.43% | 2.43% | 3.29% |
| Run 5 | 1.99% | 2.43% | 2.51% | 2.44% |
| **0.4** | | | | |
| Run 1 | 1.75% | 2.55% | 2.75% | 2.88% |
| Run 2 | 1.87% | 2.35% | 2.71% | 3.13% |
| Run 3 | 1.67% | 2.87% | 2.23% | 2.88% |
| Run 4 | 2.03% | 2.35% | 2.39% | 2.84% |
| Run 5 | 1.99% | 2.23% | 2.71% | 2.64% |
| **0.6** | | | | |
| Run 1 | 1.42% | 2.19% | 2.63% | 2.92% |
| Run 2 | 2.23% | 2.39% | 2.43% | 3.05% |
| Run 3 | 1.75% | 2.67% | 2.39% | 3.13% |
| Run 4 | 2.03% | 2.67% | 2.35% | 3.01% |
| Run 5 | 1.34% | 2.67% | 2.27% | 2.72% |
| **0.8** | | | | |
| Run 1 | 1.50% | 2.27% | 2.23% | 2.48% |
| Run 2 | 1.67% | 2.19% | 2.23% | 2.84% |
| Run 3 | 1.58% | 2.59% | 2.27% | 3.09% |
| Run 4 | 1.58% | 2.51% | 2.27% | 3.49% |
| Run 5 | 1.58% | 2.43% | 2.59% | 2.60% |

TABLE II
FUNCTION CALL SELECTION ERROR RATE BY TEMPERATURE. THIS TABLE SHOWS THE PERCENTAGE OF INSTANCES WHERE THE MODEL FAILED TO SELECT THE CORRECT FUNCTION NAME, COMPARING THE CG AND PP METHODS.

| | Llama-3.3-70B | | Qwen3-30B-A3B | |
|---|---|---|---|---|
| **Temp** | **CG** | **PP** | **CG** | **PP** |
| **0.0** | | | | |
| Run 1 | 2.39% | 2.39% | 2.31% | 1.72% |
| Run 2 | 2.40% | 2.40% | 2.35% | 2.35% |
| Run 3 | 2.40% | 2.40% | 2.05% | 2.10% |
| Run 4 | 2.38% | 2.38% | 2.34% | 2.56% |
| Run 5 | 2.38% | 2.38% | 2.13% | 1.98% |
| **Total** | **2.39%** | **2.39%** | **2.24%** | **2.14%** |
| **0.2** | | | | |
| Run 1 | 2.44% | 2.55% | 2.28% | 2.19% |
| Run 2 | 2.60% | 2.54% | 2.03% | 2.14% |
| Run 3 | 2.33% | 2.39% | 2.24% | 2.14% |
| Run 4 | 2.50% | 2.44% | 2.51% | 2.18% |
| Run 5 | 2.59% | 2.70% | 1.93% | 1.77% |
| **Total** | **2.49%** | **2.52%** | **2.20%** | **2.08%** |
| **0.4** | | | | |
| Run 1 | 2.71% | 2.28% | 2.02% | 1.98% |
| Run 2 | 2.55% | 2.80% | 2.40% | 2.14% |
| Run 3 | 2.09% | 2.43% | 2.19% | 2.40% |
| Run 4 | 2.48% | 1.98% | 1.97% | 2.08% |
| Run 5 | 2.59% | 2.55% | 2.23% | 1.88% |
| **Total** | **2.49%** | **2.41%** | **2.16%** | **2.09%** |
| **0.6** | | | | |
| Run 1 | 2.08% | 2.61% | 2.28% | 2.23% |
| Run 2 | 2.65% | 2.71% | 2.18% | 1.93% |
| Run 3 | 2.49% | 2.44% | 2.08% | 2.14% |
| Run 4 | 1.93% | 2.49% | 2.07% | 1.89% |
| Run 5 | 2.70% | 2.03% | 2.34% | 1.82% |
| **Total** | **2.37%** | **2.46%** | **2.19%** | **2.00%** |
| **0.8** | | | | |
| Run 1 | 2.38% | 2.17% | 2.08% | 2.35% |
| Run 2 | 2.69% | 2.15% | 2.20% | 2.10% |
| Run 3 | 1.93% | 2.55% | 1.93% | 2.51% |
| Run 4 | 2.02% | 2.45% | 2.46% | 2.50% |
| Run 5 | 2.30% | 2.09% | 2.39% | 1.99% |
| **Total** | **2.27%** | **2.28%** | **2.21%** | **2.29%** |

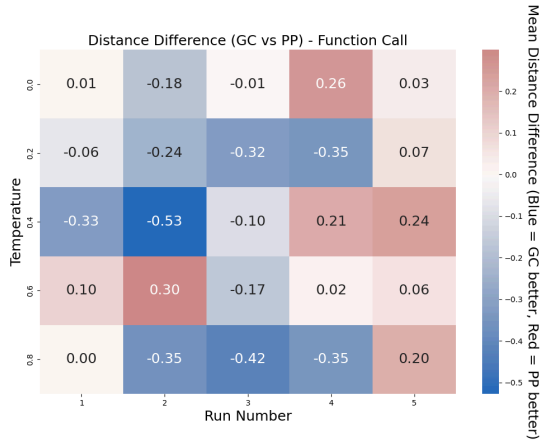


Fig. 1. Heatmap of Levenshtein distance differences for function call parameters.

and function selection error rate) across the tested temperature range (0.0 to 0.8). For instance, as shown in Figure 1, the difference in parameter correctness between the two methods does not show a clear trend favoring one method as temperature increases. Similarly, Table II indicates that the function selection error rate remains comparable for both methods across all temperatures.

- **Impact of Model Size:** The benefits of CG are notably amplified when using smaller LLMs (under 14B parameters). Smaller models, which may have a less ingrained understanding of complex formatting rules or a higher propensity for generating syntactically flawed structures, show a more significant improvement in valid output rates when guided by CG compared to PP. The explicit constraints of CG appear to compensate effectively for the reduced inherent formatting capabilities of these smaller models. Conversely, for larger, more capable models such as `Llama-3.3-70B`, the difference in success rates

between PP and CG was observed to be narrower, in the order of *1.5%* in favor of CG for some tasks. This suggests that while CG still provides an edge, very large models are inherently more adept at adhering to format instructions even in a post-parsing setup. This appears to be only a small gain, but success rates are initially high, and this is not the only advantage of CG.

However, the key advantage of CG lies in its structural guarantee for smallers models (¡14B). The risk of generating syntactically invalid outputs is larger on smaller models.

These preliminary findings suggest that Constrained Generation is not only effective in ensuring syntactically valid outputs but also demonstrates particular strengths in scenarios involving higher generation temperatures or the use of smaller, potentially less capable, language models. The ability of CG to consistently produce parsable JSON, irrespective of these factors, underscores its potential for enhancing the reliability of LLM interactions with external tools and systems. Further analysis will delve into specific error types and the nuances of performance across different schemas and function complexities.

## V. Conclusion

In this paper, we proposed and evaluated a Constrained Generation (CG) approach for LLM-based function calling, leveraging structured output mechanisms like xgrammar. Our preliminary hypothesis is that this method offers significant improvements in terms of output reliability and format adherence compared to traditional post-parsing techniques. The experiments conducted on the Hermes-Function-Calling-v1 dataset are designed to validate these claims.

Future work will involve exploring more complex grammars, investigating the impact on semantic correctness, and extending the approach to handle dynamic schema generation and multi-turn conversational function calling. The integration of structured outputs promises to make LLM interactions with external tools more robust and developer-friendly.

## References


[1] "interstellarninja" and "Teknium", "Hermes-Function-Calling-Dataset-V1", Hugging Face, 2023. [Online]. Available: https://huggingface.co/datasets/NousResearch/hermes-function-calling-v1

[2] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, et al., "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[3] T. Schick, J. Dwivedi-Yu, R. Dessì, R. Raileanu, M. Lomeli, L. Zettlemoyer, N. Cancedda, and T. Scialom, "Toolformer: Language models that teach themselves to use tools," *arXiv preprint arXiv:2302.04761*, 2023.

[4] M. X. Liu, F. Liu, A. J. Fiannaca, T. Koo, L. Dixon, M. Terry, and C. J. Cai, "'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*, 2024, pp. 1–9.

[5] Y. Liu, D. Li, K. Wang, Z. Xiong, F. Shi, J. Wang, B. Li, and B. Hang, "Are LLMs good at structured outputs? A benchmark for evaluating structured output capabilities in LLMs", *Information Processing & Management*, vol. 61, no. 5, p. 103809, 2024.

[6] Y. Li, R. Ramprasad, and C. Zhang, "A simple but effective approach to improve structured language model output for information extraction", *arXiv preprint arXiv:2402.13364*, 2024.

[7] S. Geng, H. Cooper, M. Moskal, S. Jenkins, J. Berman, N. Ranchin, R. West, E. Horvitz, and H. Nori, "Generating Structured Outputs from Language Models: Benchmark and Studies", *arXiv preprint arXiv:2501.10868*, 2025.

[8] T. Koo, F. Liu, and L. He, "Automata-based constraints for language model decoding", *arXiv preprint arXiv:2407.08103*, 2024.

[9] S. Zhang, J. Zhao, R. Xu, X. Feng, and H. Cui, "Output constraints as attack surface: Exploiting structured generation to bypass llm safety mechanisms", *arXiv preprint arXiv:2503.24191*, 2025.

[10] Yixin Dong and Charlie F. Ruan and Yaxing Cai and Ruihang Lai and Ziyi Xu and Yilong Zhao and Tianqi Chen, "XGrammar: Flexible and Efficient Structured Generation Engine for Large Language Models", *arXiv preprint arXiv:2411.15100*, 2024.

[11] Woosuk Kwon and Zhuohan Li and Siyuan Zhuang and Ying Sheng and Lianmin Zheng and Cody Hao Yu and Joseph E. Gonzalez and Hao Zhang and Ion Stoica, "Efficient Memory Management for Large Language Model Serving with PagedAttention", *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*, 2023.