Good-Enough Structured Generation: A Case Study on JSON Schema

Ivan Lee **Loris D'Antoni** **Taylor Berg-Kirkpatrick**
UC San Diego
iylee@ucsd.edu

Abstract

Grammar-constrained decoding—which masks invalid tokens during generation to guarantee outputs stay within a specified formal language—promises to eliminate structural errors in language model outputs. Yet when tested on JSON Schema (the most common application of grammar-constrained decoding), popular implementations achieve as low as 50% coverage on real-world schemas. Through experiments on 9,558 real-world JSON schemas, we find that treating validation as an external tool—using validation failures as feedback for runtime alignment—outperforms sophisticated constrained decoding methods, achieving 95% coverage in exchange for higher latency—typically an additional 1-4 seconds per schema. This gap stems from multiple issues: grammar-constrained decoding is theoretically limited to context-free grammars, real-world schemas often require context-sensitive validation, and even within context-free constraints, implementations struggle with token-boundary misalignment and state explosion. While our analysis focuses specifically on JSON Schema—where language models may excel due to extensive training exposure—it raises questions about whether increasingly complex decoding algorithms are the right approach. As language models improve, treating validation as a separate feedback tool in an agentic loop may prove more practical than embedding constraints into the decoding process itself.

1 Introduction

The ability of large language models (LLMs) to produce structured data is fundamental to their integration into modern software systems. To ensure reliability, a principled approach known as Grammar-Constrained Decoding (GCD) [1] has emerged as a leading technique. By masking invalid tokens during the generation process, GCD offers the theoretical elegance of a formal guarantee: every output will be syntactically correct according to a specified grammar. This promise of an infallible generator has motivated extensive research, positioning GCD as a key technique for building robust, predictable LLM-powered applications.

While GCD is a general technique for enforcing structured outputs, its primary application in both industry and research is the enforcement of JSON Schema [2]. JSON has become the de facto lingua franca for modern systems, mediating everything from web APIs and configuration files to, most critically, the tool-calling protocols that underpin the emerging agentic paradigm. The ability to reliably generate JSON that conforms to a specific schema is therefore not an academic exercise, but a core requirement for enabling LLMs to act as reliable components and autonomous agents. This paper challenges the prevailing assumption that GCD is the optimal solution for this critical task. Through a rigorous case study on real-world JSON Schemas, we demonstrate that popular GCD implementations are surprisingly brittle, often failing on the complex schemas that characterize modern software systems. We contrast this with simple iterative approaches that treat validation as an external tool: naive rejection sampling that retries from scratch, and a verifier loop that incorporates diagnostic

39th Conference on Neural Information Processing Systems (NeurIPS 2025) Workshop: Deep Learning For Code in the Agentic Era.

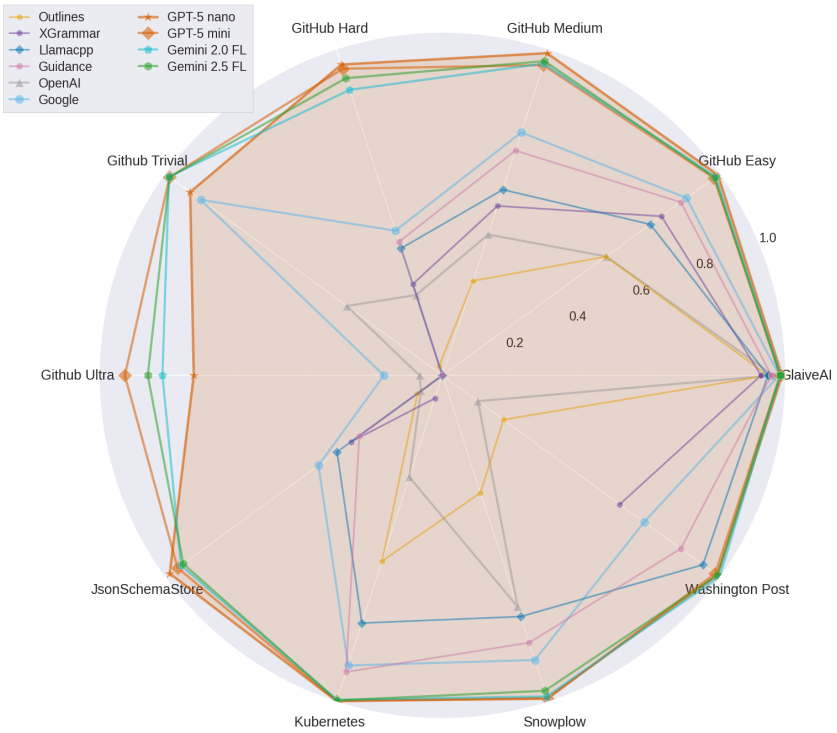

Figure 1: Coverage comparison across the JSONSchemaBench datasets. Each axis represents a dataset of increasing complexity; 100% coverage on all axes indicates a robust method. The verify-retry approaches (right side of legend) consistently achieve high coverage across all complexities. In contrast, GCD-based methods (left side of legend) exhibit brittle, dataset-specific failures, with coverage collapsing on more complex schemas. Results for Outlines, XGrammar, Llama.cpp, and Guidance from [2].

feedback. Both achieve near-perfect coverage, with the verifier loop providing additional gains on complex schemas when paired with reasoning tokens. We argue that these simpler, tool-centric approaches are not only more robust but also better aligned with the trajectory of both foundation models and the agentic systems they enable.

We are careful to scope our claims. This paper presents a case study focused on JSON Schema—a high-resource format that is ubiquitous in LLM training data. We acknowledge that our findings may not generalize to novel, out-of-distribution Domain-Specific Languages (DSLs), where the formal guarantees of GCD could be more critical. Our goal is not to dismiss GCD universally, but to initiate a conversation about its cost-benefit trade-off for the majority of common applications. We aim to question whether the pursuit of decode-time perfection remains the right goal as foundation models grow more capable and the simple, scalable power of feedback loops becomes the defining feature of the agentic era.

Our contributions are threefold: (1) We provide an empirical comparison showing a simple verify-retry loop surpasses specialized GCD frameworks in robustness with an increase in latency. (2) We perform a root-cause analysis, explaining how GCD's failures stem from a combination of a theoretical mismatch with the JSON Schema specification and documented practical implementation challenges. (3) We contextualize these findings within two competing paradigms for structured generation, prompting a re-evaluation of their respective trade-offs in the agentic era.

2 Empirical Reality: Coverage vs. Efficiency

We ground our analysis in a large-scale empirical study designed to test the real-world robustness of structured generation. Our evaluation uses the diverse JSONSchemaBench benchmark [2], which

Table 1: Cost vs. Coverage analysis. The verifier loop and rejection sampling methods achieve near-perfect final coverage at the cost of higher latency compared to the brittle structured output (GCD) approach. The Coverage (%) column for these iterative methods shows the range from the first attempt to the final result after up to five attempts. The Reasoning column specifies the budget of "thinking" tokens the model was instructed to use before generating its final output.

Model	Reasoning	Decoding Strategy	Coverage (%)	Latency (s)	Tokens
gpt-5-nano	minimal	Structured Output	50.2	1.7	93
		Rejection Sampling	86.3→93.6	4.1	360
		Verifier Loop	86.0→94.2	3.6	363
	medium	Structured Output	49.5	8.1	1022
		Rejection Sampling	94.5→98.4	11.2	2008
		Verifier Loop	94.8→98.9	11.4	1932
gpt-5-mini	minimal	Structured Output	50.2	2.8	106
		Rejection Sampling	94.3→97.1	6.3	313
		Verifier Loop	93.9→98.1	6.3	309
gemini-2.0-flash-lite	0	Structured Output	77.8	2.0	196
		Rejection Sampling	93.2→95.8	2.3	363
		Verifier Loop	92.9→96.2	2.3	383
gemini-2.5-flash-lite	0	Structured Output	72.2	1.1	136
		Rejection Sampling	91.3→96.2	2.1	363
		Verifier Loop	91.5→96.6	1.9	389
	512	Structured Output	77.9	1.9	445
		Rejection Sampling	91.0→96.5	3.2	812
		Verifier Loop	91.0→97.6	3.1	772

contains 9,558 schemas from sources like GitHub, Kubernetes, and JsonSchemaStore. On this benchmark, we tested leading models from OpenAI (GPT-5 family) and Google (Gemini family) using three distinct decoding strategies. As a baseline, we evaluate the proprietary **structured output** (GCD) endpoints. We compare this to two iterative methods: a naive **rejection sampling** that retries from scratch, and our proposed **verifier loop**, which uses diagnostic error messages as feedback for runtime alignment. We measure performance primarily by coverage (the percentage of schemas successfully passed), with secondary metrics of end-to-end latency and generated tokens.

2.1 Verify-Retry Achieves Superior Coverage

Across our benchmark of real-world JSON Schemas, a pragmatic verify-retry loop consistently achieves near-perfect compliance, whereas all tested GCD methods exhibit brittle performance and low coverage. The results, visualized in Figure 1, are stark. Our verify-retry approach using GPT-5 nano (orange lines) rapidly converges to near-perfect compliance on all datasets within five attempts. Notably, even a single unconstrained attempt is competitive with or superior to many constrained methods on complex datasets like JsonSchemaStore and Kubernetes. The result is the consistent, near-perfect polygon described in Figure 1, visually representing a method that achieves robust coverage across all tested schema complexities.

In sharp contrast, every GCD implementation produces an irregular, spiky shape, indicating an inability to handle the full benchmark. While proprietary APIs (OpenAI, Gemini) and advanced open-source frameworks (XGrammar [3], Guidance [4]) perform reasonably on simpler datasets like GitHub Easy, their coverage collapses on more complex, real-world schemas. Popular libraries such as Outlines [5] fail almost entirely. This demonstrates that while GCD methods can handle simple structures, their coverage of the features and complexities found in modern software development is severely limited—a weakness that the simple, iterative nature of verify-retry completely overcomes.

2.2 The Practical Cost of Full Coverage

The core appeal of GCD lies in its promise of single-shot efficiency. By guaranteeing a valid output in one pass, it offers predictable latency and minimal token usage—a theoretically elegant alternative

to iterative correction. This is a powerful and compelling advantage, especially in latency-sensitive or resource-constrained environments.

However, our empirical results reveal a stark trade-off: this theoretical efficiency is purchased at the cost of practical coverage. Table 1 quantifies this choice. Achieving near-100% coverage with our verifier loop required, on average, fewer than two attempts per schema. For example, with gpt-5-nano, this translates to an end-to-end latency of 3.6 seconds per schema, an increase of 1.9 seconds over a single, unconstrained generation pass. Given that GCD methods often fail entirely on these tasks, this additional latency provides a near-guarantee of success.

2.3 Is Rejection Sampling Sufficient?

Table 1 reveals a potentially surprising result: rejection sampling achieves aggregate coverage comparable to the verifier loop (e.g., 98.4% vs 98.9% for GPT-5 Nano with medium reasoning). This raises a natural question: if independent retries perform nearly as well as cumulative feedback, is the verifier loop's added complexity justified?

To answer this, we analyze performance across JSONSchemaBench subsets, averaging results separately for models with and without reasoning tokens. Table 2 shows coverage across the three decoding strategies.

Table 2: Coverage (%) across decoding strategies for models with and without reasoning tokens. Δ_{verifier} shows marginal gain from rejection sampling to verifier loop.

Subset	Structured	Rejection	Verifier	Δ_{verifier}
With reasoning tokens				
Github_trivial	60.9	98.2	99.1	+0.9
Github_easy	73.9	98.9	99.1	+0.3
Github_medium	59.2	97.8	98.3	+0.6
Github_hard	33.4	91.7	95.5	**+3.9**
Github_ultra	9.5	86.6	92.1	**+5.5**
Glaiveai2K	97.0	98.8	98.8	-0.0
JsonSchemaStore	25.5	97.0	96.9	-0.0
Kubernetes	59.9	99.9	99.9	-0.0
Snowplow	76.9	97.9	99.5	+1.6
WashingtonPost	43.2	99.6	100.0	+0.4
Without reasoning tokens				
Github_trivial	60.7	96.5	97.0	+0.6
Github_easy	73.3	97.8	98.4	+0.6
Github_medium	56.5	94.6	95.4	+0.7
Github_hard	32.7	88.8	90.5	+1.7
Github_ultra	10.7	86.0	84.8	-1.2
Glaiveai2K	97.4	98.4	98.7	+0.2
JsonSchemaStore	25.1	92.8	93.2	+0.4
Kubernetes	56.8	99.5	99.5	-0.1
Snowplow	77.1	96.8	97.8	+1.0
WashingtonPost	40.2	96.2	97.0	+0.8

The data reveals a clear pattern: verifier loop effectiveness depends on whether reasoning tokens are leveraged. With reasoning enabled, verifier loop provides clear gains over rejection sampling on difficult subsets: +5.5 percentage points on Github_ultra and +3.9pp on Github_hard. On easier subsets already near saturation (e.g., Glaiveai2K at 98.8%), the gains are minimal. Without reasoning tokens, verifier loop shows little benefit and can even degrade performance (-1.2pp on Github_ultra). These results demonstrate that while rejection sampling is competitive in aggregate, verifier loop shows larger gains on hard schemas when paired with reasoning tokens.

Having established that verify-retry methods—particularly verifier loop with reasoning—achieve better coverage than GCD, a critical question remains: is GCD's failure an implementation flaw, or a sign of a more fundamental problem? In the next section, we perform a root-cause analysis to answer this question.

3 A Root Cause Analysis of GCD's Brittleness

The stark performance gap documented in Section 2 is not a simple implementation flaw but stems from a deep, two-part mismatch: a theoretical ceiling imposed by the limits of context-free grammars, and the immense practical hurdles of implementing even a subset of the specification.

First, GCD is fundamentally limited to context-free grammars (CFGs), but the JSON Schema specification is not purely context-free. Many of its most powerful and commonly used features require validation that goes beyond syntactic structure. This includes not only explicitly context-sensitive keywords like `if/then/else` and `dependentSchemas`, but also fundamental value-based constraints such as `minimum`, `maximum`, `minLength`, and `multipleOf`. These constraints require semantic interpretation of a parsed value—a capability that pure CFGs do not possess. For this entire class of common schemas, any CFG-based decoder is guaranteed to be insufficient, not because of a bug, but by design, making it theoretically incapable of supporting the full specification.

Second, even for the theoretically context-free subset of JSON Schema, practical implementations face severe implementation challenges. Prior work has documented immense engineering challenges, including token-boundary misalignment where an LLM's vocabulary does not cleanly map to a grammar's terminals, and developers making pragmatic trade-offs to offer only incomplete feature support [6, 7]. Furthermore, complex features like regex patterns can lead to state explosion, creating unmanageably large automata during parsing. Thus, the 'solvable' portion of the problem remains a significant practical hurdle, explaining the low coverage rates even on simpler schemas.

4 Discussion

Our findings are not merely a benchmark result, but a symptom of a larger shift in AI systems. They suggest a fundamental re-evaluation of constrained decoding in the agentic era, where the ability to use tools and learn from feedback is paramount. The case for GCD is challenged by the "Bitter Lesson" of machine learning [8]: simple methods that leverage scale often outperform complex, specialized ones. As models become fluent with common formats like JSON, the core problem shifts from enforcing syntax to correcting occasional errors. This is analogous to code generation, where developers rely on an ecosystem of external tools like compilers and linters, not an infallible generator.

This does not render GCD obsolete, but it demands a clear-eyed cost-benefit analysis. While its single-shot guarantee remains valuable for latency-critical applications or novel DSLs, our work shows that for the common case of JSON Schema, this efficiency is purchased at the cost of profound brittleness. The simpler, more robust paradigm is the agentic loop. This approach accepts generators as competent but fallible, and focuses on perfecting the feedback loop between the model and its verification tools. We argue the future of reliable AI lies not in building ever-more-complex, perfect decoders, but in embracing this pragmatic, powerful, and scalable tool-centric paradigm.

References

[1] Saibo Geng, Martin Josifoski, Maxime Peyrard, and Robert West. Grammar-constrained decoding for structured NLP tasks without finetuning. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 10932–10952. Association for Computational Linguistics, 2023.

[2] Saibo Geng, Hudson Cooper, Michał Moskal, Samuel Jenkins, Julian Berman, Nathan Ranchin, Robert West, Eric Horvitz, and Harsha Nori. Jsonschemabench: A rigorous benchmark of structured outputs for language models, 2025.

[3] Yixin Dong, Charlie F. Ruan, Yaxing Cai, Ruihang Lai, Ziyi Xu, Yilong Zhao, and Tianqi Chen. Xgrammar: Flexible and efficient structured generation engine for large language models, 2024.

[4] Scott Lundberg and Marco Tulio Ribeiro. Guidance: A guidance framework for constrained generation, 2023. Microsoft Research. Software framework available at `https://github.com/guidance-ai/guidance`.

[5] Brandon T. Willard and Rémi Louf. Efficient guided generation for large language models, 2023.

[6] Luca Beurer-Kellner, Marc Fischer, and Martin Vechev. Guiding llms the right way: Fast, non-invasive constrained generation, 2024.

[7] Shubham Ugare, Tarun Suresh, Hangoo Kang, Sasa Misailovic, and Gagandeep Singh. Syncode: Llm generation with grammar augmentation, 2024.

[8] Richard S Sutton. The bitter lesson. *Incomplete Ideas (blog)*, March 2019. `http://www.incompleteideas.net/IncIdeas/BitterLesson.html`.