JSONSchemaBench: Evaluating Constrained Decoding with LLMs on Efficiency, Coverage and Quality

Saibo Geng [1 *] **Hudson Cooper** [2] **Michał Moskal** [2] **Samuel Jenkins** [2] **Julian Berman** [3] **Nathan Ranchin** [1]
Robert West [1 2] **Eric Horvitz** [2] **Harsha Nori** [2]

Abstract

Reliably and efficiently generating structured outputs has become a critical capability for modern language model (LM) applications. *Constrained decoding* has emerged as the dominant technology across sectors for enforcing structured outputs during generation. Despite its growing adoption, little has been done with the systematic evaluation of the behaviors and performance of constrained decoding. Constrained decoding frameworks have standardized around JSON Schema as a structured data format, with most uses guaranteeing constraint compliance given a schema. However, there is poor understanding of the effectiveness of the methods in practice. We present an evaluation framework to assess constrained decoding approaches across three critical dimensions: **efficiency** in generating constraint-compliant outputs, **coverage** of diverse constraint types, and **quality** of the generated outputs. To facilitate this evaluation, we introduce **JSONSchemaBench**, a collection of 10K real-world JSON schemas that encompass a wide range of constraints with varying complexity. We pair the benchmark with the existing official JSON Schema Test Suite and evaluate six state-of-the-art constrained decoding frameworks, including Guidance, Outlines, Llamacpp, XGrammar, OpenAI, and Gemini. Our work provides actionable insights for improving constrained decoding frameworks and structured generation tasks, setting a new standard for evaluating constrained decoding and structured generation. Code is available at https://github.com/guidance-ai/jsonschemabench.

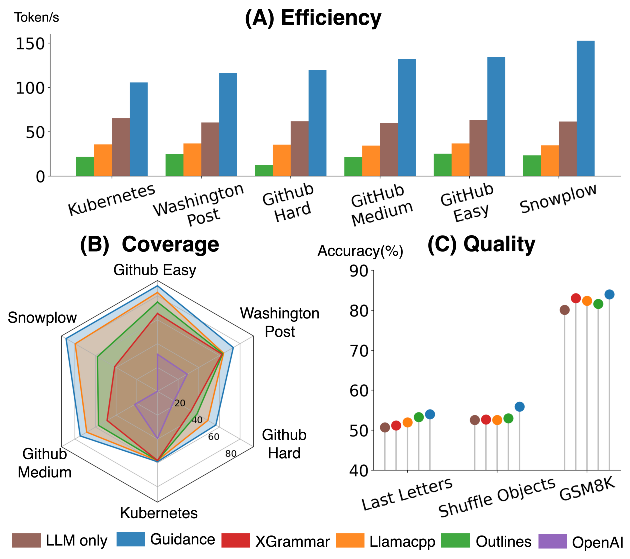


Figure 1. Comparison across various constrained-decoding frameworks by efficiency (speed of output generation), coverage (support for JSON Schema features), and quality (effects on underlying task accuracy). Guidance outperforms other frameworks on these dimensions.

1. Introduction

The rapid advancements in LMs in recent years have significantly broadened their applications, extending beyond natural language tasks to more complex challenges such as web navigation (Yao et al., 2023b), data extraction (Polak & Morgan, 2024), and tool use (Schick et al., 2023). Unlike traditional natural language processing (NLP) tasks where the output is aimed at review by humans, output in these applications is often consumed by machines such as controller and service APIs. The machine-oriented nature of these applications requires LMs to generate structured outputs that strictly adhere to predefined formats and constraints. However, the LM generation process is probabilistic and does not provide guarantees on the output's structure, making it challenging to deploy LMs in applications requiring structured inputs and high reliability.

The methodology of constrained decoding, a technique that

*Work done during internship at Microsoft [1]EPFL [2]Microsoft [3]JSON Schema. Correspondence to: Saibo Geng <saibo.geng@epfl.ch>.

Non-archival presentation at ES-FoMo III: Efficient Systems for Foundation Models Workshop, ICML 2025, Vancouver, Canada. 2025.

integrates constraints into the decoding process of LMs, has been developed to address the need to adapt LM generations to the challenge of providing structured output. Constrained decoding intervenes in the decoding process of LMs by masking out invalid tokens based on given constraints and prefix tokens. This intervention guides the LM to sample only from valid tokens, ensuring that the final output perfectly conforms to a predefined structure.

The strong demand for structured generation (Liu et al., 2024) has led to the development of various constrained-decoding frameworks[1], such as Guidance (Guidance AI, 2023), Outlines (Willard & Louf, 2023), XGrammar (Dong et al., 2024) and the grammar module of Llamacpp (Gerganov & al., 2023) These frameworks provide broad support for different types of constraints, minimal overhead, and compatibility with various LM ecosystems, facilitating the adoption of constrained decoding in real-world applications.

JSON Schema offers a high level, domain-specific way to define constraints for JSON data, a widely adopted data interchange format. As a result, JSON Schema has emerged as a key specification language for constrained decoding. Commercial LM providers, such as OpenAI, have embraced constrained decoding by incorporating support for JSON Schema directly into their APIs. These integrations highlight the emergence of JSON Schema as an industry-wide standard for specifying constraints on structured outputs, ensuring compatibility across diverse applications. Despite the growing adoption of constrained decoding for structured generation, several issues and questions persist:
Q1: Efficiency: Does constrained decoding slow down or speed up the generation process? Which framework is the most efficient?
Q2: Coverage: The JSON Schema specification has an evolving and expansive feature set. How well do existing constrained decoding frameworks support these features?
Q3: Quality: While constrained decoding guarantees that LM outputs conform to a desired structure, does it negatively affect the semantic quality of outputs?

To answer these questions, we need to study constrained-decoding methods with a large-scale, diverse, and real-world collection of user-defined structures. To evaluate the performance of constrained decoding frameworks, we introduce **JSONSchemaBench**, a collection of 10K real-world JSON schemas from various sources, Organized into 10 datasets of varying complexity and diversity, the benchmark spans domains such as function signatures, service APIs, and system configurations. We evaluate six state-of-the-art constrained decoding frameworks, including Guidance, Outlines, Llamacpp, XGrammar, OpenAI, and Gemini, on JSONSchemaBench. We pair this real-world schema dataset with the official JSON Schema Test Suite (JSON Schema Org, 2024) in order to extract detailed insights into coverage of JSON Schema functionality across these frameworks, and to further evaluate them with considerations of end-to-end task accuracy in the context of multiple real-world tasks. Altogether, our evaluation takes three aspects into consideration: efficiency, coverage, and quality. We define specific metrics to measure these three functional aspects and evaluate constrained decoding frameworks against them. Through extensive experiments, we converge on the following findings as illustrated in Figure 1. (1) Constrained decoding can speed up the generation process by 50% compared to unconstrained decoding. (2) Frameworks demonstrate significant differences in their actual support for real-world JSON schemas, with the best framework supporting twice as many schemas as the worst. (3) Constrained decoding consistently improves the performance of downstream tasks up to 4%, even for tasks with minimal structure like GSM8k.

Contributions Our contributions are three-fold:

- We assemble JSON schemas from various sources and organize them into a benchmark, JSONSchemaBench, to facilitate the evaluation of constrained decoding frameworks on JSON schema.
- We propose a fine-grained evaluation framework to assess the versatility of constrained decoding frameworks in handling diverse JSON schema features, including declared coverage, empirical coverage, and compliance rate.
- We evaluate six state-of-the-art constrained decoding frameworks on JSONSchemaBench, uncovering their strengths and limitations in generating schema-compliant JSON outputs and analyzing their impact on downstream tasks.

2. Background and Related Work

JSON Schema is a meta-language that describes the structure of JSON data. It is capable of expressing a wide variety of constraints, such as the types of JSON object properties, the length of JSON arrays or the pattern that a JSON string must match. The syntax and capabilities of JSON Schema are defined in the JSON Schema specification (Wright et al., 2022), which defines a large number of *keywords*, each of which may be used or combined with other keywords within a schema to enforce constraints like the ones mentioned. JSON Schema is widely used in the software ecosystem, and previous work has been done to collect extensive examples of JSON Schemas with a focus both on real-world use as well as on overall correctness.

[1]We use the terms *constrained decoding framework* and *grammar engine* interchangeably.

Baazizi et al. (2021) collected over 6,000 JSON schemas from publicly available GitHub repositories. Attouche et al. (2022) used it alongside additional collected JSON schemas in order to evaluate a witness generation algorithm for JSON Schema. Separately, the official JSON Schema Test Suite (JSON Schema Org, 2024) is a collection of manually created test cases, maintained by the JSON Schema core team, which exercises a large portion of the functionality defined in the JSON Schema specification. It was originally written to assist implementers of JSON Schema validation tools with testing their compliance against the specification, and therefore contains a wide variety of examples for each of JSON Schema's keywords, including in edge case scenarios. Notably, Bowtie (Bowtie, 2025) leverages the test suite as a foundation for comparing and understanding different implementations of the JSON Schema specification across programming languages. Taken together, these two datasets form a large number of examples both of JSON Schema's diverse feature set as well as its use in the wild.

Algorithm 1 Constrained Decoding

```
Require: Constraint $C$, LLM $f$, Prompt $x$
Ensure: Output $o$ adhering to $C$
 1: $o \leftarrow []$
 2: loop
 3:     $C$.update($o$)                ▷ advance state of $C$
 4:     $m \leftarrow C$.mask()        ▷ compute mask
 5:     $v \leftarrow f(x + o)$        ▷ compute logits
 6:     $v' \leftarrow m \odot v$
 7:     $t \leftarrow$ decode($\alpha'$)   ▷ sample
 8:     if $t$ = EOS then
 9:         break
10:     end if
11:     $o$.append($t$)
12: end loop
13: return $o$                         ▷ output
```

Constrained decoding (Deutsch et al., 2019; Shin et al., 2021; Scholak et al., 2021; Poesia et al., 2022; Wang et al.; Geng et al., 2023) refers to methods that guide the generation process of language models (LMs) by masking out tokens that do not adhere to predefined constraints at each step. Recently, highly optimized grammar-constrained decoding frameworks (Guidance AI, 2023; Beurer-Kellner et al., 2023; Willard & Louf, 2023; Kuchnik et al., 2023; Zheng et al., 2024; Dong et al., 2024) have been developed to improve the efficiency and usability of constrained decoding.

The evaluation of constrained decoding remains an underexplored topic, with no consensus on what defines the effectiveness of constrained decoding. While some research has pursued comparisons of constrained decoding with unconstrained LMs (Roy et al., 2024; Tang et al., 2024; Yao et al., 2023a), the studies to date fail to provide comparisons across different constrained decoding frameworks. The benchmarks employed have either narrowly focused on specific tasks or rely on formal-grammar–based artificial setups, that have unclear relevance to real-world use cases.

3. The JSONSchemaBench

Our goal is to design a benchmark that is (1) diverse enough to cover the most common constraint types encountered in real-world applications, (2) large enough to provide a reliable evaluation, and (3) equipped with fair and multidimensional metrics to ensure comprehensive assessments.

3.1. Data Collection

We start with the 6K JSON schemas collected by (Baazizi et al., 2021) from publicly available GitHub repositories, and with the set of schemas from the JSON Schema Test Suite (JSON Schema Org, 2024). We further collect JSON schemas from other sources, such as the JSON Schema Store (Schema Store Org, 2014), the GlaiveAI function calling dataset V2 (GlaiveAI, 2024), and from Kubernetes configuration files (Kubernetes, 2022). We filter out invalid schemas and standardize the schemas to ensure that they conform to the version of JSON Schema declared[2] in each schema The GitHub JSON schemas collection from (Baazizi et al., 2021) contains schemas of varying complexity and diversity, ranging from simple type constraints to complex constraints with nested objects and arrays. For more fine-grained evaluation, we split the data into five collections based on the schema size: trivial, small, medium, large, ultra. The suites finalized after all collection and processing are listed in Table 1. We excluded GitHub-Trivial and GitHub-Ultra from the experiments as they were too easy or too hard. However, we retained these datasets in the benchmark, with GitHub-Ultra serving as an aspirational target for future advancements. For more information on post-processing

Dataset	Category	Count
GlaiveAI-2K	Function Call	1707
Github-Trivial	Misc	444
Github-Easy	Misc	1943
Snowplow	Operational API	403
Github-Medium	Misc	1976
Kubernetes	Kubernetes API	1064
Washington Post	Resource Access API	125
Github-Hard	Misc	1240
JSONSchemaStore	Misc	492
Github-Ultra	Misc	164
Total		9558

Table 1. Schema collection metadata.

[2]The `$schema` keyword, defined in the JSON Schema specification, allows any schema to self-identify which version of JSON Schema it is written for.

and dataset splitting, we refer the reader to Appendix A.

4. Efficiency

Naïve implementations of constrained decoding add overhead to the standard LM inference process, including a per-step mask computation and an optional one-time grammar compilation. However, several optimizations can significantly reduce this overhead. For instance, mask computation can run in parallel with the LM's forward pass, and grammar compilation can be performed concurrently with pre-filling computations (Guidance AI, 2023; Dong et al., 2024). Other optimizations such as grammar caching and constraint-based speculative decoding (GuidanceAI, 2024b; Beurer-Kellner et al., 2023; Kurt, 2024a) can further reduce overhead.

Metrics We break down the efficiency evaluation into the following components:

- **Grammar Compilation Time (GCT):** The time spent on grammar compilation.
- **Time to First Token (TTFT):** Time from the start of generation to the production of the first token.
- **Time per Output Token (TPOT):** Average time to generate each token after the first.

4.1. Setup

The efficiency experiment depends on both the size of the model and the tokenizer's vocabulary size. We used **Llama-3.1-8B-Instruct** with the **Llamacpp** inference engine as backend for Outlines, Guidance, and Llamacpp. As XGrammar doesn't support Llamacpp as backend , we add an additional experiment with the **Hugging Face Transformers** inference engine for XGrammar. All experiments are conducted on a single **NVIDIA A100-SXM4-80GB GPU** with **AMD EPYC 7543 (12 cores)** CPU. The batch size is set to 1 for all experiments. Additional details about setup are provided in the Appendix E.

Addressing coverage bias. The efficiency metrics are meaningful only for instances that a grammar engine can process. Different engines exhibit varying levels of schema coverage, with some engines handling a wider range of schemas than others. Engines with lower coverage often process simpler, shorter schemas, which naturally compile and generate faster. As a result, averaging efficiency metrics across covered instances can introduce bias favoring engines with lower coverage. For a more detailed discussion on coverage, see Section 5. To ensure fairness, we calculate efficiency metrics on the intersection of covered instances across all engines.

Dataset	Framework	GCT (s)	TTFT (s)	TPOT (ms)
GlaiveAI	LM only	NA	**0.10**	15.40
	Guidance	**0.00**	0.24	**6.37**
	Llamacpp	0.05	0.20	29.98
	Outlines	3.48	3.65	30.33
GitHub Easy	LM only	NA	**0.10**	15.83
	Guidance	**0.00**	0.34	**7.44**
	Llamacpp	0.05	0.18	27.22
	Outlines	3.71	3.97	39.78
Snowplow	LM only	NA	**0.11**	16.23
	Guidance	**0.00**	0.28	**6.55**
	Llamacpp	0.05	0.20	28.90
	Outlines	3.91	4.14	42.66
GitHub Medium	LM only	NA	**0.20**	16.68
	Guidance	**0.01**	0.54	**7.57**
	Llamacpp	0.06	0.30	29.08
	Outlines	8.05	8.38	46.57
Kubernetes	LM only	NA	**0.16**	15.32
	Guidance	**0.01**	0.45	**9.47**
	Llamacpp	0.05	0.28	28.04
	Outlines	5.29	5.55	46.10

Table 2. **Efficiency metrics** for different engines with **LlamaCpp** as the inference engine. **GCT**: Grammar Compilation Time, **TTFT**: Time to First Token, **TPOT**: Time Per Output Token. Bold values indicate the smallest in each column for GCT, TTFT, and TPOT. All values are **median** of the samples. Results for the GitHub Hard and Washington Post datasets are provided in Appendix E.

Grammar compilation time. There are notable differences in grammar compilation times between the engines. Both Guidance and Llamacpp dynamically compute their constraints during token generation, leading to minimal grammar compilation time. In the middle, XGrammar does include a non-trivial compilation step, but they are able to largely mitigate its impact by running it concurrently with prompt pre-filling. Finally Outlines, which converts JSON schemas into regular-expression based constraints, has significantly higher compilation time.

Time per output token. While Outlines and Llamacpp demonstrate substantially lower throughput than the LM-only approach, Guidance achieves even higher efficiency, which it accomplishes by fast-forwarding [3] certain generation steps with its *guidance acceleration* (GuidanceAI, 2024b). Comparing Guidance and XGrammar with the HF Transformers backend shows that Guidance has a significantly better TPOT.

[3] See Tables12 and 13 for the number of tokens fast-forwarded.

Dataset	Framework	GCT (s)	TTFT (s)	TPOT (ms)
GlaiveAI	Guidance	**0.01**	0.36	**36.92**
	XGrammar	0.12	**0.30**	66.78
GitHub Easy	Guidance	**0.01**	0.37	**42.03**
	XGrammar	0.11	**0.33**	65.57
GitHub Medium	Guidance	**0.01**	0.55	**44.21**
	XGrammar	0.20	**0.48**	65.51
GitHub Hard	Guidance	**0.01**	0.73	**35.88**
	XGrammar	0.30	**0.65**	65.20

Table 3. As XGrammar doesn't support **llama.cpp**, we add an additional experiment with the **Hugging Face Transformers** inference engine for XGrammar and Guidance. All values are **median** of the result samples.

5. Coverage

Each constrained decoding framework has limitations when it comes to translating JSON schemas into a set of constraints that can reliably guarantee the validity of LM outputs. To systematically evaluate the effectiveness of these frameworks, we define three notions of coverage:

Definition 5.1 (Declared Coverage)**.** A schema is considered *declared covered* if the framework processes the schema without explicitly rejecting it or encountering runtime errors such as exceptions or crashes.

Definition 5.2 (Empirical Coverage)**.** A schema is considered *empirically covered* if our experiments show that the constraints generated by the framework result in LM outputs that are schema-compliant.

Definition 5.3 (True Coverage)**.** A schema is considered *truly covered* if the framework produces constraints that are precisely equivalent to the original JSON Schema definition, i.e., permitting all schema-compliant generations while rejecting all schema-noncompliant generations.

The most ideal coverage metric is the *true coverage*, denoted as $\mathcal{C}_{\text{True}}$. However, due to the infinite number of JSON instances that could be validated against a schema, it is difficult to measure in practice without a formal verification method that is capable of exhaustively comparing the schema's semantics against the framework's implementation. $\mathcal{C}_{\text{Empirical}}$ is an approximation of $\mathcal{C}_{\text{True}}$ as it only checks whether the finitely many outputs seen during our experiments conform to a given schema[4]. While $\mathcal{C}_{\text{Declared}}$ is not an estimate of $\mathcal{C}_{\text{True}}$ per se, it is an upper-bound of both $\mathcal{C}_{\text{Empirical}}$ and $\mathcal{C}_{\text{True}}$ and is useful in deriving an additional metric from the coverage evaluation: **Compliance Rate** $= \mathcal{C}_{\text{Empirical}}/\mathcal{C}_{\text{Declared}}$. The *Compliance Rate* estimates the reliability of the constrained decoding framework in guaranteeing compliance given it accepts a given schema.

[4]Additionally, we define *theoretical coverage* as the proportion of schemas whose features are fully supported by the grammar engine, with details provided in Appendix C.

5.1. Setup

To measure empirical coverage, we conduct all experiments using the Llama-3.2-1B-Instruct model as it is small enough to run efficiently while still producing high-quality outputs. The prompt consists of a simple instruction with two-shot examples (Figure 3), and validation is performed using the `jsonschema` Python library (Berman, 2025) (using JSON Schema Draft2020-12) with string-format checks enabled. We use greedy decoding with zero-temperature, performing a single generation run, and enforce a 40-second timeout for grammar compilation and an additional 40 seconds for generation. Exceeding these limits is treated as a schema processing failure. Additional details are provided in Appendix B.

5.2. Results

Empirical Coverage Guidance shows the highest empirical coverage on six out of the eight datasets, with Llamacpp taking the lead on the remaining two: the domain-specific Washington Post and notably hard JSON Schema Store. On the other hand, closed-source grammar engines consistently have the lowest coverage; they came in last on all but one dataset. We note that while empirical coverage is a reasonable indicator of a framework's real-world performance, it is influenced by factors such as the LM being used and the sampling methods employed.

Compliance Rate Among open-source engines, guidance consistently demonstrates the highest compliance rate across all datasets, making it the most reliable option for ensuring schema compliance. Outlines has a comparatively lower compliance rate, primarily due to timeouts during generation. Our analysis reveals that JSON Schema features like minItems, maxItems, enum, and Array, while supported, often take 40 seconds to 10 minutes for Outlines to process. While closed-source implementations have low empirical coverage, they have very high compliance rates, indicating that their providers have taken a more conservative strategy, implementing only a subset of JSON Schema features that they can reliably support.

5.3. JSON Schema Test Suite: Complementary Evaluation

Originally designed to test the correctness and compliance of JSON Schema validation implementations, the official JSON Schema Test Suite (JSON Schema Org, 2024) is a comprehensive collection of test cases spanning the many features of the JSON Schema specification. We believe that the test suite is an ideal tool for assessing the correctness of

Dataset	Framework	Declared	Empirical	Compliant Rate
GlaiveAI	Guidance	0.98	**0.96**	**0.98**
	Llamacpp	0.98	0.95	0.97
	Outlines	0.99	0.95	0.96
	XGrammar	1.00	0.93	0.93
	OpenAI	0.89	0.89	1.00
	Gemini	0.86	0.86	1.00
GitHub Easy	Guidance	0.90	**0.86**	**0.96**
	Llamacpp	0.85	0.75	0.88
	Outlines	0.86	0.59	0.83
	XGrammar	0.91	0.79	0.87
	OpenAI	0.30	0.29	0.97
	Gemini	0.08	0.07	0.88
Snowplow[*]	Guidance	0.87	**0.82**	**0.94**
	Llamacpp	0.92	0.74	0.81
	Outlines	0.95	0.36	0.61
	XGrammar	NA	NA	NA
	OpenAI	0.21	0.21	1.00
GitHub Medium[*]	Guidance	0.79	**0.69**	**0.87**
	Llamacpp	0.77	0.57	0.74
	Outlines	0.72	0.29	0.40
	XGrammar	0.79	0.52	0.66
	OpenAI	0.13	0.12	0.92
Kubernetes[*]	Guidance	0.98	**0.91**	**0.92**
	Llamacpp	0.98	0.76	0.78
	Outlines	0.98	0.57	0.58
	XGrammar	0.12	0.07	0.58
	OpenAI	0.21	0.21	1.00
Washington Post[*]	Guidance	0.86	0.86	**1.00**
	Llamacpp	0.97	**0.94**	0.97
	Outlines	0.97	0.22	0.23
	XGrammar	0.85	0.64	0.75
	OpenAI	0.13	0.13	1.00
GitHub Hard[*]	Guidance	0.60	**0.41**	**0.69**
	Llamacpp	0.61	0.39	0.63
	Outlines	0.47	0.03	0.06
	XGrammar	0.69	0.28	0.41
	OpenAI	0.09	0.09	1.00
JsonSchema Store[*]	Guidance	0.35	0.30	**0.88**
	Llamacpp	0.54	**0.38**	0.69
	Outlines	0.38	0.09	0.24
	XGrammar	0.76	0.33	0.43
	OpenAI	0.06	0.06	1.00

Table 4. **Coverage of all the frameworks** on JSONSchemaBench. Empirical coverage between Open Source engines and OpenAI/Gemini are not directly comparable due to differences in the underlying model (Llama 3.2-1B vs. proprietary models).
[*] Gemini results are ommitted for dataset suites with $< 1\%$ support.

grammar engines.

The test suite organizes its test cases into 45 categories, each of which corresponds to a feature of JSON Schema, typically a specific keyword such as `required` or group of tightly related keywords such as `if-then-else`. A small number of additional categories test broader behaviors, such as `infinite-loop-detection`. Each test case contains a single schema paired with a collection of JSON instances that are marked as either valid or invalid under that schema. For the purpose of evaluating coverage, we assert that an engine must successfully generate each valid instance and block generation of each invalid instance to "pass" a test case. In addition to compilation failures, we define two failure modes that a grammar engine can exhibit:

Framework	Compile-Error	Over-constrained	Under-constrained
Outlines	42	16	8
Llamacpp	37	18	7
XGrammar	**3**	**5**	38
Guidance	25	7	**1**

Table 5. Number of categories for which each failure type occurred at least once. Rows do not necessarily sum to the total number of categories, as some categories may have more than one failure type or no failures at all. Bold numbers indicate the framework with the fewest number of failures of a given type.

Definition 5.4 (Over-constrained)**.** A framework is *over-constrained* if it rejects JSON instances that are vsalid according to a given JSON Schema. This means the engine is too strict and excludes outputs that should be allowed.

Definition 5.5 (Under-constrained)**.** A framework is *under-constrained* if it allows JSON instances that are invalid according to a given JSON Schema. This means the engine is overly permissive and allows outputs that should be rejected.

An illustration is given in Figure 5 in Appendix D. *Over-constrained* grammar engines risk limiting the expressive power of LMs, potentially preventing the generation of valid responses and negatively impacting downstream task performance. Conversely, under-constrained engines cannot guarantee that all responses will be valid, often necessitating additional post-processing or retry logic.

5.3.1. Results

Coverage Analysis For each grammar engine and category in the test suite, we calculate *test coverage* as the proportion of passing test cases, reported in Figure 6 in Appendix D Additionally, Table 6 aggregates these metrics, counting categories with minimal coverage ($> 0\%$), partial coverage ($> 25\%$), moderate coverage ($> 50\%$), high coverage ($> 75\%$), and full coverage (100%). We indicate the number of categories for which each framework achieves the highest test coverage (either as the single highest or as the sole leader) as well as the number of categories for which each framework is the sole leader.

Coverage	Outlines	Llamacpp	XGrammar	Guidance
Minimal coverage (>0%)	20	21	28	**30**
Partial coverage (>25%)	11	11	16	**25**
Moderate coverage (>50%)	2	5	3	**21**
High coverage (>75%)	0	2	1	**17**
Full coverage (100%)	0	1	1	**13**
Tied for highest (>0%)	4	6	14	**25**
Single highest	1	0	10	**19**

Table 6. Number of categories with a given level of coverage. Each row represents a cumulative coverage threshold, with higher thresholds indicating stricter levels of success. Bold numbers indicate the framework with the highest value in that row.

- **Overall Performance**: Guidance outperforms other engines at all coverage levels, achieving full coverage on 13 categories and moderate coverage on 21. In comparison, Llamacpp and XGrammar have full coverage on only one category and moderate coverage on five and three categories, respectively, while Outlines has no full coverage on any category and moderate coverage on two categories.
- **Single Highest**: Guidance has the single highest coverage in 19 categories, followed by XGrammar with 10, and Outlines with one, and Llamacpp with none.

Failure Analysis Table 5 provides a breakdown of failure modes for each framework across the test suite, detailing the number of categories with compilation errors, failures to generate positive instances (over-constrained), and failures to block negative instances (under-constrained). Overall, Guidance demonstrates the fewest total failures, in particular minimizing under-constrained errors. Outlines, Llamacpp, and Guidance follow a consistent failure pattern, with most errors occurring during compilation and over-constrained failures being more frequent than under-constrained ones. In contrast, XGrammar minimizes compilation errors but shows the highest number of under-constrained failures, indicating a trade-off favoring permissiveness.

We acknowledge that there is no straightforward correspondence between test suite performance and empirical coverage. One reason for this is that not all features are equally represented in real-world schemas. As a result, strong or weak performance on specific features can have disproportionate impacts depending on their prevalence. Another reason is under-constraining effectively delegates responsibility to the LM, which may produce valid output despite a lack of strict constraints. We emphasize that while under-constraining can be a legitimate strategy, it requires careful implementation and transparency to ensure reliability.

6. Quality

In principle, constrained decoding should not affect the quality of the generated output as it only filters out the invalid tokens. However, things become more complicated due to ambiguity of tokenization (Vivien, 2024; GuidanceAI, 2024a; Geng et al., 2024) and the distributional shifts caused by the intervention (Geng et al., 2023; Tam et al., 2024). As a hypothetical toy example, an LM might answer 89,000 instead of the correct 89000 in a GSM8K question. Constrained decoding can block the invalid token , , enforcing structural compliance but potentially may cause the LM to go out of distribution and generate 890000 instead. (Kurt, 2024b) argued that the performance decline observed in previous studies (Tam et al., 2024) comes from inadequate prompting, insufficient contextual information, and poorly crafted schemas.

6.1. Setup

Kurt (2024b); Tam et al. (2024) have introduced a series of tasks to investigate potential quality concerns in constrained decoding, which we leverage and extend in this benchmark. Specifically, we adopt the three reasoning tasks from these studies to evaluate the impact of constrained decoding on task accuracy, as detailed in Table 14. The simple output structure of these tasks was designed to isolate the effects of constrained decoding on reasoning, as outlined by (Tam et al., 2024). For our experiments, we use the Llama-3.1-8B-Instruct model to measure task performance. We follow the original setup and prompt specifications from (Kurt, 2024b), with full details provided in Appendix F.

We implement the following constraints for the first three tasks: **(1) Last Letter** the output needs to be a concatenation of letters from a-z; **(2) Shuffle Objects** the output needs to be a single letter from A-E enclosed in parentheses; **(3) GSM8K** the output is an valid integer or float number. The outputs for all three tasks are structured as JSON objects with two fields: `"reasoning"` and `"answer"`, formatted as `{"reasoning": <reasoning about the answer>, "answer": <final answer>}`.

6.2. Results

The results in Table 7 show that the constrained decoding, regardless of the framework, achieves higher performance than the unconstrained setting. Among the frameworks evaluated, Guidance consistently delivers the best performance across all tasks, with approximately a 3% improvement over the LM-only approach in every task. We believe this may be attributed to its token-healing implementation (GuidanceAI, 2024a).

	Last Letters	Shuffle Objects	GSM8K
LM only	50.7%	52.6%	80.1%
XGrammar	51.2%	52.7%	83.7%
Llamacpp	52.0%	52.6%	82.4%
Outlines	53.3%	53.0%	81.6%
Guidance	**54.0%**	**55.9%**	**83.8**%

Table 7. Accuracy on the quality tasks.

7. Conclusion

We have proposed a comprehensive evaluation framework for constrained decoding frameworks with JSON schemas, focusing on efficiency, coverage, and output quality. We introduced **JSONSchemaBench**, a benchmark comprising 10K real-world JSON schemas, to enable robust assessment under realistic conditions. Our evaluation highlights both the advancements and limitations of current state-of-the-art constrained decoding frameworks. We hope our findings and benchmark guide future research in structured generation, helping the community identify effective tools and extend capabilities with constrained decoding.

References

Analytics, S. Iglu central. `https://github.com/snowplow/iglucentral`, 2022. Commit hash 726168e. Retrieved 19 September 2022.

Attouche, L., Baazizi, M.-A., Colazzo, D., Ghelli, G., Sartiani, C., and Scherzinger, S. Witness Generation for JSON Schema. *Proceedings of the VLDB Endowment*, 15(13):4002–4014, September 2022. ISSN 2150-8097. doi: 10.14778/3565838.3565852. URL `https://dl.acm.org/doi/10.14778/3565838.3565852`.

Baazizi, M. A., Colazzo, D., Ghelli, G., Sartiani, C., and Scherzinger, S. A json schema corpus, 2021. `https://github.com/sdbs-uni-p/json-schema-corpus`.

Berman, J. python-jsonschema. `https://github.com/python-jsonschema/jsonschema`, 2025. URL `https://github.com/python-jsonschema/jsonschema`. Accessed: 2025-01-05.

Beurer-Kellner, L., Fischer, M., and Vechev, M. Prompting Is Programming: A Query Language for Large Language Models. *Proceedings of the ACM on Programming Languages*, 7(PLDI):1946–1969, June 2023. ISSN 2475-1421. doi: 10.1145/3591300. URL `http://arxiv.org/abs/2212.06094`. arXiv:2212.06094 [cs].

Bowtie. Bowtie: A meta-validator of the json schema specification, 2025. URL `https://github.com/bowtie-json-schema/bowtie/`. DOI: 10.5281/zenodo.14646449.

Deutsch, D., Upadhyay, S., and Roth, D. A General-Purpose Algorithm for Constrained Sequential Inference. In *Proceedings of the 23rd Conference on Computational Natural Language Learning (CoNLL)*, pp. 482–492, Hong Kong, China, 2019. Association for Computational Linguistics. doi: 10.18653/v1/K19-1045. URL `https://www.aclweb.org/anthology/K19-1045`.

Dong, Y., Ruan, C. F., Cai, Y., Lai, R., Xu, Z., Zhao, Y., and Chen, T. XGrammar: Flexible and Efficient Structured Generation Engine for Large Language Models, November 2024. URL `http://arxiv.org/abs/2411.15100`. arXiv:2411.15100 [cs].

Geng, S., Josifoski, M., Peyrard, M., and West, R. Grammar-constrained decoding for structured NLP tasks without finetuning. In Bouamor, H., Pino, J., and Bali, K. (eds.), *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 10932–10952, Singapore, December 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.emnlp-main.674. URL `https://aclanthology.org/2023.emnlp-main.674`.

Geng, S., Gambhir, S., Wendler, C., and West, R. Byte bpe tokenization as an inverse string homomorphism, 2024. URL `https://arxiv.org/abs/2412.03160`.

Gerganov, G. and al. Llama.cpp: A port of facebook's llama model in c++. `https://github.com/ggerganov/llama.cpp`, 2023. Accessed: 2025-01-16.

GlaiveAI. Glaive function calling dataset. `https://huggingface.co/datasets/glaiveai/glaive-function-calling`, 2024. Accessed: 2024-12-21.

Guidance AI. Guidance: A language model programming framework, 2023. URL `https://github.com/guidance-ai/guidance`. Accessed: 2024-12-18.

GuidanceAI. Prompt boundaries and token healing. `https://github.com/guidance-ai/guidance/blob/main/notebooks/art_of_prompt_design/prompt_boundaries_and_token_healing.ipynb`, 2024a. Accessed: 2024-12-21.

GuidanceAI. Guidance acceleration tutorial. https://guidance.readthedocs.io/en/stable/example_notebooks/tutorials/guidance_acceleration.html, 2024b. Accessed: 2025-01-16.

JSON Schema Org. Json schema test suite. https://github.com/json-schema-org/JSON-Schema-Test-Suite, 2024. URL https://github.com/json-schema-org/JSON-Schema-Test-Suite. Accessed: 2024-12-19.

Kubernetes. Kubernetes json schemas. https://github.com/instrumenta/kubernetes-json-schema, 2022. Commit hash 133f848.

Kuchnik, M., Smith, V., and Amvrosiadis, G. Validating Large Language Models with ReLM, May 2023. URL http://arxiv.org/abs/2211.15458. arXiv:2211.15458 [cs].

Kurt, W. Coalescence: Making llm inference 5x faster. https://blog.dottxt.co/coalescence.html, 2024a. Accessed: 2024-12-21.

Kurt, W. Say what you mean: A response to 'let me speak freely', 2024b. URL https://.txt.co/blog/say-what-you-mean-a-response-to-let-me-speak-freely.

Liu, M. X., Liu, F., Fiannaca, A. J., Koo, T., Dixon, L., Terry, M., and Cai, C. J. "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*, pp. 1–9, May 2024. doi: 10.1145/3613905.3650756. URL http://arxiv.org/abs/2404.07362. arXiv:2404.07362 [cs].

Poesia, G., Polozov, O., Le, V., Tiwari, A., Soares, G., Meek, C., and Gulwani, S. Synchromesh: Reliable code generation from pre-trained language models, January 2022. URL http://arxiv.org/abs/2201.11227. arXiv:2201.11227 [cs].

Polak, M. P. and Morgan, D. Extracting accurate materials data from research papers with conversational language models and prompt engineering. *Nature Communications*, 15(1), February 2024. ISSN 2041-1723. doi: 10.1038/s41467-024-45914-8. URL http://dx.doi.org/10.1038/s41467-024-45914-8.

Post, T. W. ans-schema. https://github.com/washingtonpost/ans-schema, 2022. Commit hash abdd6c211. Retrieved 19 September 2022.

Roy, S., Thomson, S., Chen, T., Shin, R., Pauls, A., Eisner, J., and Durme, B. V. BenchCLAMP: A Benchmark for Evaluating Language Models on Syntactic and Semantic Parsing, January 2024. URL http://arxiv.org/abs/2206.10668. arXiv:2206.10668 [cs].

Schema Store Org. The largest collection of independent json schemas in the world. https://www.schemastore.org/json/, 2014. A universal JSON schema store where schemas for popular JSON documents can be found. Contributions are welcome; see CONTRIBUTING.md for more information.

Schick, T., Dwivedi-Yu, J., Dessi, R., Raileanu, R., Lomeli, M., Hambro, E., Zettlemoyer, L., Cancedda, N., and Scialom, T. Toolformer: Language Models Can Teach Themselves to Use Tools. In Oh, A., Naumann, T., Globerson, A., Saenko, K., Hardt, M., and Levine, S. (eds.), *Advances in Neural Information Processing Systems*, volume 36, pp. 68539–68551. Curran Associates, Inc., 2023.

Scholak, T., Schucher, N., and Bahdanau, D. PICARD: Parsing incrementally for constrained auto-regressive decoding from language models. In Moens, M.-F., Huang, X., Specia, L., and Yih, S. W.-t. (eds.), *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pp. 9895–9901, Online and Punta Cana, Dominican Republic, November 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.emnlp-main.779. URL https://aclanthology.org/2021.emnlp-main.779/.

Shin, R., Lin, C., Thomson, S., Chen, C., Roy, S., Platanios, E. A., Pauls, A., Klein, D., Eisner, J., and Van Durme, B. Constrained Language Models Yield Few-Shot Semantic Parsers. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pp. 7699–7715, Online and Punta Cana, Dominican Republic, November 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.emnlp-main.608. URL https://aclanthology.org/2021.emnlp-main.608.

Tam, Z. R., Wu, C.-K., Tsai, Y.-L., Lin, C.-Y., Lee, H.-y., and Chen, Y.-N. Let me speak freely? a study on the impact of format restrictions on large language model performance. In Dernoncourt, F., Preoţiuc-Pietro, D., and Shimorina, A. (eds.), *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing: Industry Track*, pp. 1218–1236, Miami, Florida, US, November 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.emnlp-industry.91. URL https://aclanthology.org/2024.emnlp-industry.91/.

Tang, X., Zong, Y., Phang, J., Zhao, Y., Zhou, W., Cohan, A., and Gerstein, M. Struc-Bench: Are Large Language Models Good at Generating Complex Structured Tabular Data? In Duh, K., Gomez, H., and Bethard, S. (eds.), *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 2: Short Papers)*, pp. 12–34, Mexico City, Mexico, June 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.naacl-short.2. URL https://aclanthology.org/2024.naacl-short.2.

Vivien. Llm decoding with regex constraints. https://vivien000.github.io/blog/journal/llm-decoding-with-regex-constraints.html, 2024. Accessed: 2024-12-21.

Wang, B., Wang, Z., Wang, X., Cao, Y., Saurous, R. A., and Kim, Y. Grammar Prompting for Domain-Specific Language Generation with Large Language Models.

Willard, B. T. and Louf, R. Efficient Guided Generation for Large Language Models, August 2023. URL http://arxiv.org/abs/2307.09702. arXiv:2307.09702 [cs].

Wright, A., Andrews, H., Hutton, B., and Dennis, G. Draft 2020-12: Json schema core specification. https://json-schema.org/draft/2020-12/json-schema-core.html, 2022. Published 16 June 2022. Metaschema available at https://json-schema.org/draft/2020-12/schema.

Yao, S., Chen, H., Hanjie, A. W., Yang, R., and Narasimhan, K. COLLIE: Systematic Construction of Constrained Text Generation Tasks, July 2023a. URL http://arxiv.org/abs/2307.08689. arXiv:2307.08689 [cs].

Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K., and Cao, Y. ReAct: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*, 2023b.

Zheng, L., Yin, L., Xie, Z., Sun, C., Huang, J., Yu, C. H., Cao, S., Kozyrakis, C., Stoica, I., Gonzalez, J. E., Barrett, C., and Sheng, Y. SGLang: Efficient Execution of Structured Language Model Programs, June 2024. URL http://arxiv.org/abs/2312.07104. arXiv:2312.07104 [cs].

A. JSON Schema Collections Details

JSONSchemaBench includes a diverse collection of schemas curated from multiple real-world applicationsAttouche et al. (2022), designed to represent a wide range of use cases:

Sources:

- **GitHub (Baazizi et al., 2021)**: Extracted from open-source repositories containing schema definitions, representing practical, widely-used applications. Schemas from GitHub are of various complexities, totaling 6,000 schemas. We split the collection into trivial (fewer than 10 fields), easy (10–30 fields), medium (30–100 fields), hard (100–500 fields), and ultra (more than 500 fields), based on the total number of fields in each JSON schema to reflect increasing complexity and scale.
- **Snowplow (Analytics, 2022)**: Sourced from event-based analytics frameworks, showcasing schemas tailored for event-driven data structures.
- **Kubernetes (Kubernetes, 2022)**: Schemas defining configurations for container orchestration systems, highlighting schemas with intricate hierarchical structures.
- **WashingtonPost (Post, 2022)**: Schemas for The Washington Post's ANS specification.
- **GlaiveAI2K (GlaiveAI, 2024)**: 2,000 schemas extracted from a function-calling dataset. Each schema represents a function signature.
- **JSON Schema Store (Schema Store Org, 2014)**: The largest collection of independent JSON schemas in the world.

A.1. Data Processing

To ensure the quality and reliability of JSONSchemaBench, we applied the following preprocessing steps:

1. Validation

- Verified schemas conform to the JSON Schema specification using the `jsonschema` library in Python, specifically targeting the `Draft2020-12` version. Drop invalid schemas.
- Identified additional invalid schemas using validators from Rust and JavaScript libraries.

2. Cleaning

- **Deduplicate:** Removed duplicate schemas to eliminate redundancy and maintain a diverse dataset. Key ordering within schemas was ignored when determining duplicates.

Dataset	Count	Size (KB) Med / Max	Field Count Med / Max	Max Fan-Out Med / Max	Schema Depth Med / Max
GlaiveAI-2K	1707	0.5 / 1.2	21 / 44	4 / 7	5 / 8
Github-Trivial	444	0.2 / 10.8	6 / 9	4 / 9	2 / 6
Github-Easy	1943	0.5 / 20.3	18 / 29	5 / 19	4 / 10
Snowplow	403	0.9 / 15.6	37 / 450	7 / 131	3 / 13
Github-Medium	1976	1.5 / 58.3	51 / 99	8 / 42	6 / 15
Kubernetes	1064	2.7 / 818.6	41 / 11720	5 / 600	5 / 7
Washington Post	125	1.7 / 81.1	44 / 2093	7 / 84	4 / 10
Github-Hard	1240	5.1 / 136.1	175 / 498	18 / 133	8 / 25
JSONSchemaStore	492	5.9 / 2934.8	155 / 108292	14 / 6543	6 / 22
Github-Ultra	164	25.8 / 359.6	694 / 6919	37 / 412	8 / 23

Table 8. Baisc statistics of the datasets used in the experiments.

- **Empty Schema:** Excluded schemas that were lacking meaningful constraints, effectively "empty."
- **Unresolved References:** Removed schemas containing unresolved `$ref` references to external URLs.
- **Schema Version Fixes:** Corrected mismatched or missing draft versions.
- **Extraneous Field Removal:** Eliminated unrelated fields such as `command`, `config`, `path`, and `controls`.
- **Regex Escaping:** Fixed escaping issues in regular expressions to ensure validity.
- **Schema Extraction:** Extracted schemas embedded within non-root levels of JSON files.

A.2. Draft versions

A.3. Feature Distribution

We count the appearance of each feature (keyword) in the 10K schemas and show the most frequent features in Figure 2a. We separately plot usage of the `format` keyword, which is used to specify format of string such as `date-time`, `email`, `uri`. This is worth highlighted because each of these formats can be quite complex to implement on its own. The distribution of formats used is shown in Figure 2b.

B. Coverage Experiment Details

The prompting template used for the coverage experiment is shown in Figure 3.

Decoding Method We use greedy decoding with no top P or top K sampling for all the experiments. We only get one output from the model, which we will use to validate the schema compliance. It's totally plausible to sample more outputs and validate them all, and it might detect more schema violations. The fact that we only sample the top 1 output may quantify our *empirical coverage* as *Top 1 Empirical Coverage*.

Validation We use the `jsonschema` library with the Draft-2020-12 version of the JSON Schema standard to validate the generated JSON object. We turn on the 'format' checks, which are not enabled by default in Python. Strictly speaking, the `jsonschema` library doesn't guarantee the validation of all the schema constraints, even with the 'format' checks enabled. It is possible, though very rare, for a schema-noncompliant output to be validated as compliant by the `jsonschema` library, leading to a slight overestimation of empirical coverage. However, such occurrences are corner cases and happen infrequently.

C. Theoretical Coverage Details

Definition C.1 (Theoretical Coverage)**.** A schema is considered *theoretically covered* if all of its features are supported by the grammar engine.

The *theoretical coverage*, noted as $\mathcal{C}_{\text{Theoretical}}$, measures the proportion of JSON schemas that a grammar engine supports based on its implementation. It doesn't involve any model inference or experiments and is solely based on the grammar engine's implementation. $\mathcal{C}_{\text{Theoretical}}$ is an *upper bound* of the *true coverage*, which cannot be empirically measured due to the infinite number of possible generations under the schema constraints.

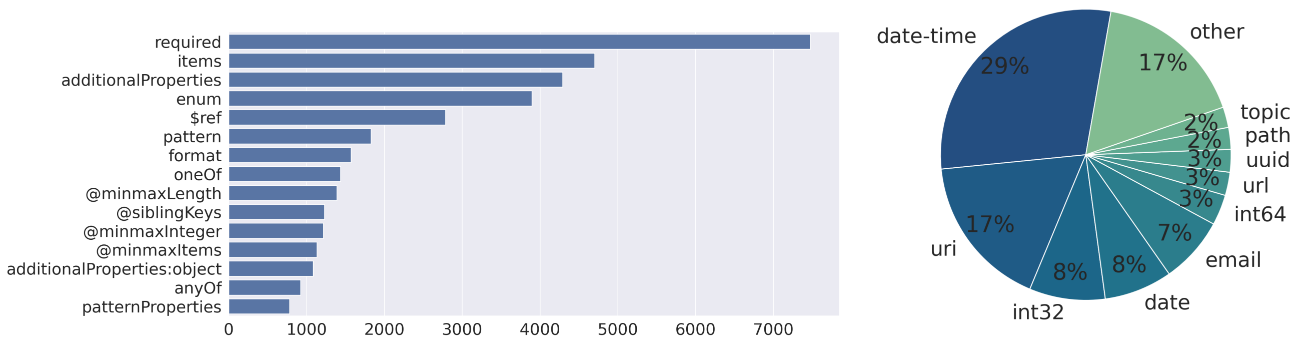
Overall, the theoretical coverage provides a good indication of the grammar engine's capability to support a wide range of schema constraints.

In our experiment, the theoretical coverage for each framework was determined based on the documentation and resources listed in Table 10.

The theoretical support for each feature in JSON Schema is summarized in Figure 4

	draft-04	draft-06	draft-07	2019-09	2020-12	unknown
Github-easy	1310	54	136	0	5	438
Github-hard	841	30	87	0	23	259
Github-medium	1221	80	140	0	7	528
JsonSchemaStore	199	5	268	5	11	4
Kubernetes	0	0	0	0	0	1087
Snowplow	0	0	0	0	0	408
WashingtonPost	125	0	0	0	0	0
Glaiveai2K	0	0	0	0	0	1707
total	4097	193	706	5	50	5155

Table 9. JSON Schema Draft Version Counts



(a) Feature Count in the 10K Schemas

(b) Format keyword distribution

Figure 2. Feature and Format constraint distribution.

The theoretical coverage of each grammar engine is summarized in Table 11.

D. JSON Schema Test Suite Experiment Details

We evaluated each constrained decoding framework's performance on the JSON Schema Test Suite using the following criteria: a framework is considered to pass a test case if it permits generating every valid instance in the test case while preventing the generation of every invalid instance. Some test cases consist exclusively of invalid instances, such as those involving *unsatisfiable* schemas, i.e., schemas for which no valid instances exist. In these cases, engines raising compile-time errors were allowed to pass.

Cleaning We removed the 'format' category of tests, as the current JSON Schema standard mandates that this keyword be ignored entirely by default. The test suite comes bundled with an 'optional' set of tests, including tests for each officially recognized value of the 'format' keyword. We hope to extend this work to include these optional tests in a follow-up.

Furthermore, some tests require external resources in the form of JSON schemas available at a remote URL. We dropped these tests from the analysis, as the constrained decoding libraries discussed in the current work do not fetch these resources by default. After filtering out these tests, we are left with 43 of the original 45 test categories.

Implementation To check whether a given framework accepts or blocks the generation of a particular JSON instance, we tokenize[5] JSON-serialized form of the instance and walk the framework's constraints forward one token at a time, essentially simulating the generation process of an LLM attempting to produce the given token sequence:

- XGrammar directly expose an interface for updating the token mask after inserting a token and checking validity.
- Outlines does not expose a public interface for interacting with the token mask, but `outlines-core`, which `outlines` is built on top of, is easily adapted

[5]The particular choice of tokenizer is not particularly important, but we use the Llama 3.1 tokenizer for consistency with our other experiments.

Prompt Template for JSON Generation

System Message:
You need to generate a JSON object that matches the schema below.

Demo Examples:
```
## Input Schema: [JSON schema]
## Expected Output: [JSON object matching the schema]
...
```

Figure 3. Prompt template used to generate JSON objects in the coverage experiment.

Frameworks	Lib Version	Release Date	JSON Schema Support Documentation
Guidance	0.2.0rc	2024.11.26	LLGuidance Documentation
Llamacpp	0.3.2	2024.11.16	llama.cpp JSON Schema to gbnf Conversion
XGrammar	0.1.6	2024.12.07	XGrammar JSON Schema to gbnf Conversion
Outlines	0.1.8	2024.12.06	Outlines JSON Schema to Regex Conversion
OpenAI	UNK	UNK	OpenAI Structured Output API
Gemini	0.8.3	2024.10.31	Gemini Structured Output Content Types

Table 10. Grammar Engine Documentation and Resources

for this purpose.

- Similarly, Guidance does not expose a public interface for interacting with the token mask, but `llguidance`, which `guidance` is built on top of, is easily adapted for this purpose.
- Llamacpp does not expose this interface, but it shares a common grammar-specification language with XGrammar. We use `llamacpp` to generate GGML BNF and check token-sequence validity using `xgrammar`'s interface.

E. Efficiency Experiment Details

For efficiency experiments, the results depend on both the size of the model and the tokenizer's vocabulary size. We used **Llama-3.1-8B-Instruct** (quantized to Q8bit) with a 128K token vocabulary to achieve a balance between computational efficiency and model capability.

Below, we outline specific considerations related to grammar and prefix caching:

- **Grammar Cache (Compilation):** Since each schema in the dataset is unique, caching grammar compilations does not offer any benefits.
- **Prefix Cache (LLM Inference):** We implement prefix caching during LLM inference for all cases to enhance efficiency by reusing computed results where applicable.

F. Quality Experiment Details

Prompt and JSON Schema For the task of **Shuffle Objects**, and **GSM8K**, we use the same prompt and JSON schema from the dottxt's "let me speak freely" rebuttal.

For the task of **Last Letter**, we make a slight modification because the original prompt used was a bad example as pointed out by (Kurt, 2024b). We also put it into a JSON format to better align with the other tasks.

Figure 8 reveals non-empty exclusive regions for each engine, indicating that no single engine outperforms the others across all instances.

Dataset	LM only	Guidance	Llamacpp	Outlines	XGrammar	OpenAI	Gemini
GlaiveAI	0.00	**0.96**	0.95	0.95	0.87	0.87	0.87
GitHub Easy	0.00	**0.87**	0.83	0.75	0.65	0.31	0.31
Snowplow	0.00	**0.80**	0.74	0.58	NA	0.29	NA
GitHub Medium	0.00	**0.73**	0.69	0.57	0.49	0.22	NA
Kubernetes	0.00	**0.58**	0.58	0.58	0.58	0.40	NA
Washington Post	0.00	**0.70**	0.64	0.63	0.62	0.29	NA
GitHub Hard	0.00	**0.54**	0.49	0.38	0.33	0.00	NA
JsonSchemaStore	0.00	**0.31**	0.24	0.20	0.13	0.00	NA

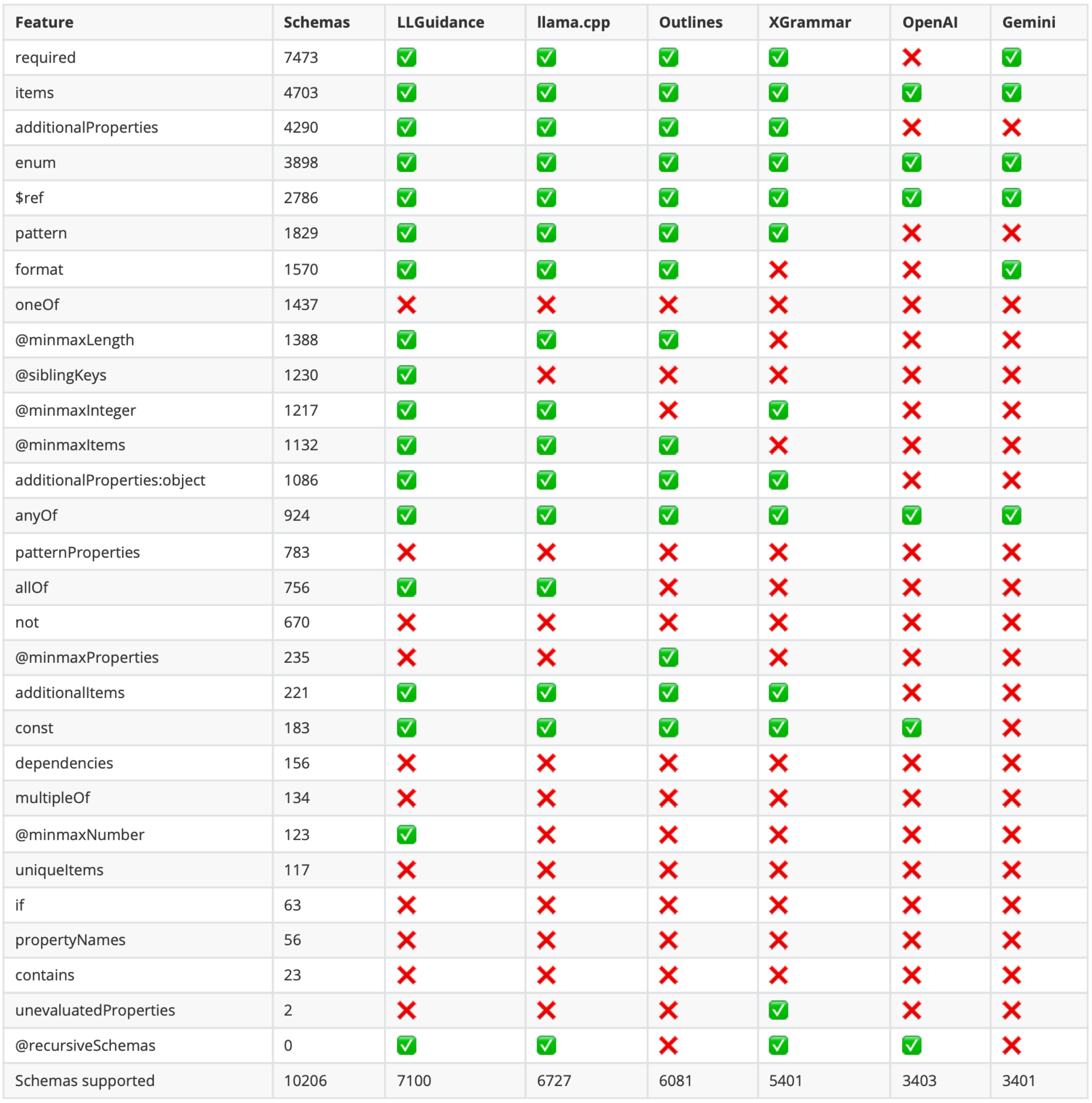
Table 11. Theoretical coverage across datasets.

Dataset	Framework	GCT (s)	TTFT (s)	TPOT (ms)	TGT (s)	Output Tokens (FF)
GlaiveAI	LLM only	NA	**0.10**	15.40	1.08	64.94 (00.00)
	Guidance	**0.00**	0.24	**6.37**	**0.50**	41.56 (15.70)
	Llamacpp	0.05	0.20	29.98	1.47	43.18 (00.00)
	Outlines	3.48	3.65	30.33	4.84	40.39 (00.00)
GitHub Easy	LLM only	NA	**0.10**	15.83	0.95	53.91 (00.00)
	Guidance	**0.00**	0.34	**7.44**	**0.60**	34.92 (10.02)
	Llamacpp	0.05	0.18	27.22	1.10	33.93 (00.00)
	Outlines	3.71	3.97	39.78	5.29	34.19 (00.00)
Snowplow	LLM only	NA	**0.11**	16.23	1.01	55.31 (00.00)
	Guidance	**0.00**	0.28	**6.55**	**0.51**	36.77 (14.50)
	Llamacpp	0.05	0.20	28.90	1.24	37.21 (00.00)
	Outlines	3.91	4.14	42.66	5.65	35.65 (00.00)
GitHub Medium	LLM only	NA	**0.20**	16.68	2.56	142.10 (00.00)
	Guidance	**0.01**	0.54	**7.57**	**1.29**	99.66 (31.42)
	Llamacpp	0.06	0.30	29.08	2.85	87.71 (00.00)
	Outlines	8.05	8.38	46.57	12.23	84.64 (00.00)
Kubernetes	LLM only	NA	**0.16**	15.32	0.84	44.38 (00.00)
	Guidance	**0.01**	0.45	**9.47**	**0.71**	28.75 (04.40)
	Llamacpp	0.05	0.28	28.04	1.06	28.09 (00.00)
	Outlines	5.29	5.55	46.10	6.56	22.26 (00.00)

Table 12. **Efficiency metrics** for different engines with **LlamaCpp** as the inference engine. **GCT**: Grammar Compilation Time, **TTFT**: Time to First Token, **TPOT**: Time Per Output Token, **TGT**: Total Generation Time, **FF**: Fast-Forwarded output tokens. Bold values indicate the smallest in each column for GCT, TTFT, TPOT, and TGT. All values are **median** of the samples.

Dataset	Framework	GCT (s)	TTFT (s)	TPOT (ms)	TGT (s)	Output Tokens (FF)
GlaiveAI	Guidance	**0.01**	0.36	**36.92**	**1.87**	41.45(16.76)
	XGrammar	0.12	**0.30**	66.78	2.87	39.47(00.00)
GitHub Easy	Guidance	**0.01**	0.37	**42.03**	**1.60**	27.67(06.75)
	XGrammar	0.11	**0.33**	65.57	4.07	59.45(00.00)
GitHub Medium	Guidance	**0.01**	0.55	**44.21**	**4.84**	96.31(26.93)
	XGrammar	0.20	**0.48**	65.51	6.53	92.93(00.00)
GitHub Hard	Guidance	**0.01**	0.73	**35.88**	**10.25**	211.40(101.40)
	XGrammar	0.30	**0.65**	65.20	14.99	221.40(00.00)

Table 13. **Efficiency metrics** for different engines with **Hugging Face Transformers** as the inference engine. All values are **median** of the samples.



Feature	Schemas	LLGuidance	llama.cpp	Outlines	XGrammar	OpenAI	Gemini
required	7473	✅	✅	✅	✅	❌	✅
items	4703	✅	✅	✅	✅	✅	✅
additionalProperties	4290	✅	✅	✅	✅	❌	❌
enum	3898	✅	✅	✅	✅	✅	✅
$ref	2786	✅	✅	✅	✅	✅	✅
pattern	1829	✅	✅	✅	✅	❌	❌
format	1570	✅	✅	✅	❌	❌	✅
oneOf	1437	❌	❌	❌	❌	❌	❌
@minmaxLength	1388	✅	✅	✅	❌	❌	❌
@siblingKeys	1230	✅	❌	❌	❌	❌	❌
@minmaxInteger	1217	✅	✅	❌	✅	❌	❌
@minmaxItems	1132	✅	✅	✅	❌	❌	❌
additionalProperties:object	1086	✅	✅	✅	✅	❌	❌
anyOf	924	✅	✅	✅	✅	✅	✅
patternProperties	783	❌	❌	❌	❌	❌	❌
allOf	756	✅	✅	❌	❌	❌	❌
not	670	❌	❌	❌	❌	❌	❌
@minmaxProperties	235	❌	❌	✅	❌	❌	❌
additionalItems	221	✅	✅	✅	✅	❌	❌
const	183	✅	✅	✅	✅	✅	❌
dependencies	156	❌	❌	❌	❌	❌	❌
multipleOf	134	❌	❌	❌	❌	❌	❌
@minmaxNumber	123	✅	❌	❌	❌	❌	❌
uniqueItems	117	❌	❌	❌	❌	❌	❌
if	63	❌	❌	❌	❌	❌	❌
propertyNames	56	❌	❌	❌	❌	❌	❌
contains	23	❌	❌	❌	❌	❌	❌
unevaluatedProperties	2	❌	❌	❌	✅	❌	❌
@recursiveSchemas	0	✅	✅	❌	✅	✅	❌
Schemas supported	10206	7100	6727	6081	5401	3403	3401

Figure 4. Feature checklist for different structured output engines

Task	Example	Structure	Metric
Last Letter	**Input:** Ian Peter Bernard Stephen **Output:** nrdn	CoT reasoning + answer in $a - z$	Case-sensitive exact match
Shuffle Objects	**Input:** Sequence of exchanges among individuals + choices **Output:** A-E	CoT reasoning + answer in $A - E$	Exact match
GSM8K	**Input:** Basic calculation problems **Output:** Number, e.g., 8	CoT reasoning + answer as integer	Number exact match

Table 14. Task Descriptions and Structures

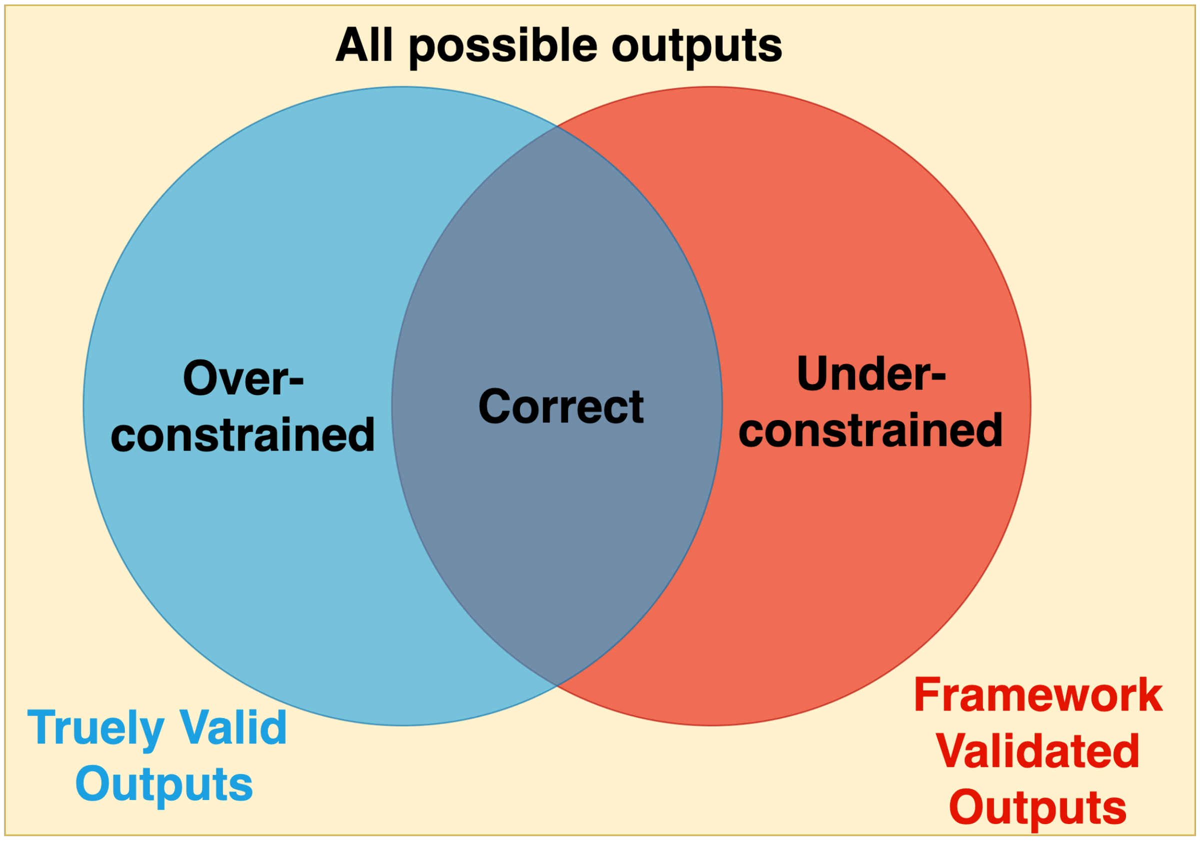
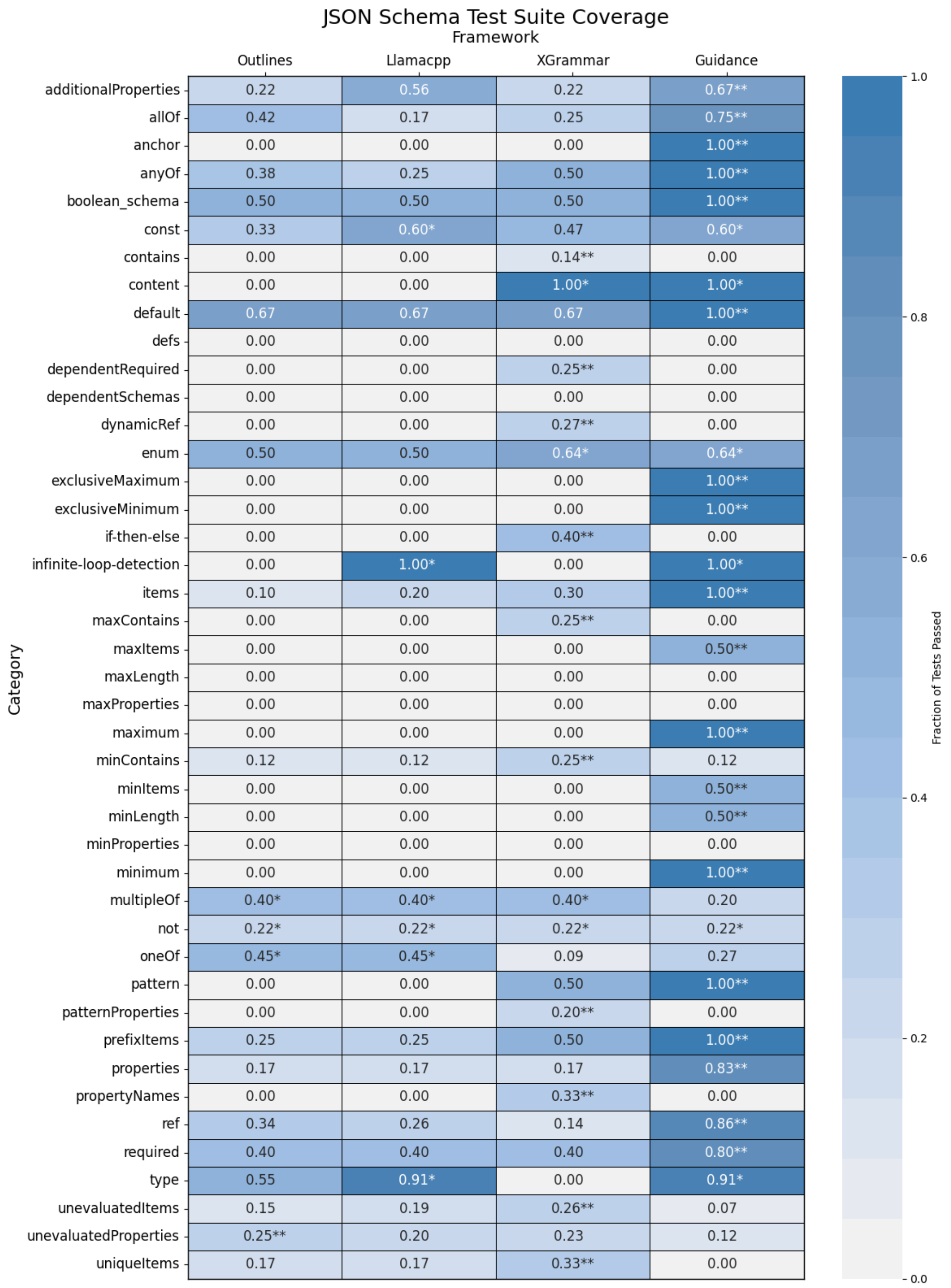


Figure 5. Illustration of over-constrained and under-constrained.



Category	Outlines	Llamacpp	XGrammar	Guidance
additionalProperties	0.22	0.56	0.22	0.67**
allOf	0.42	0.17	0.25	0.75**
anchor	0.00	0.00	0.00	1.00**
anyOf	0.38	0.25	0.50	1.00**
boolean_schema	0.50	0.50	0.50	1.00**
const	0.33	0.60*	0.47	0.60*
contains	0.00	0.00	0.14**	0.00
content	0.00	0.00	1.00*	1.00*
default	0.67	0.67	0.67	1.00**
defs	0.00	0.00	0.00	0.00
dependentRequired	0.00	0.00	0.25**	0.00
dependentSchemas	0.00	0.00	0.00	0.00
dynamicRef	0.00	0.00	0.27**	0.00
enum	0.50	0.50	0.64*	0.64*
exclusiveMaximum	0.00	0.00	0.00	1.00**
exclusiveMinimum	0.00	0.00	0.00	1.00**
if-then-else	0.00	0.00	0.40**	0.00
infinite-loop-detection	0.00	1.00*	0.00	1.00*
items	0.10	0.20	0.30	1.00**
maxContains	0.00	0.00	0.25**	0.00
maxItems	0.00	0.00	0.00	0.50**
maxLength	0.00	0.00	0.00	0.00
maxProperties	0.00	0.00	0.00	0.00
maximum	0.00	0.00	0.00	1.00**
minContains	0.12	0.12	0.25**	0.12
minItems	0.00	0.00	0.00	0.50**
minLength	0.00	0.00	0.00	0.50**
minProperties	0.00	0.00	0.00	0.00
minimum	0.00	0.00	0.00	1.00**
multipleOf	0.40*	0.40*	0.40*	0.20
not	0.22*	0.22*	0.22*	0.22*
oneOf	0.45*	0.45*	0.09	0.27
pattern	0.00	0.00	0.50	1.00**
patternProperties	0.00	0.00	0.20**	0.00
prefixItems	0.25	0.25	0.50	1.00**
properties	0.17	0.17	0.17	0.83**
propertyNames	0.00	0.00	0.33**	0.00
ref	0.34	0.26	0.14	0.86**
required	0.40	0.40	0.40	0.80**
type	0.55	0.91*	0.00	0.91*
unevaluatedItems	0.15	0.19	0.26**	0.07
unevaluatedProperties	0.25**	0.20	0.23	0.12
uniqueItems	0.17	0.17	0.33**	0.00

Figure 6. JSON Schema test suite coverage by category. Each cell represents the proportion of passing tests for a given category-framework pair, with darker shades indicating higher coverage. A single asterisk (*) marks frameworks tied for the highest (non-zero) coverage, while a double asterisk (**) marks the framework with the single highest coverage in the category.

Prompt Template for GSM8K

System Message:
You are an expert in solving grade school math tasks. You will be presented with a grade-school math word problem and be asked to solve it. Before answering, you should reason about the problem (using the ”reasoning” field in the JSON response format described below). Always respond with JSON in the format: {`"reasoning": <reasoning about the answer>, "answer": <final answer>`}. The ”reasoning” field contains your logical explanation, and the ”answer” field contains the final numeric result.

Demo Examples:

```
## Input: "[example question]"
## Output: "reasoning": "[example reasoning]", "answer": [example answer]
```

...

Figure 7. Prompt template for solving GSM8K with JSON responses.

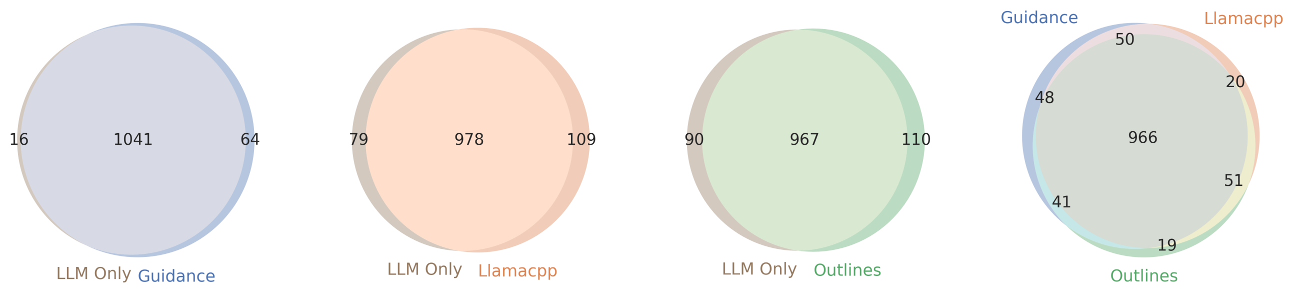


Figure 8. Overlap of Correct Instances Across Models on GSM8K