CONSTRAINED DECODING: HARMFUL OR BENEFICIAL? REVISITING ITS IMPACT ON LLM PERFORMANCE

Anonymous authors
Paper under double-blind review

ABSTRACT

Constrained decoding guarantees that Large Language Models (LLMs) produce outputs conforming to predefined formats, but is often assumed to inherently degrade reasoning. We examine this assumption through a taxonomy of three constraint types: *format constraints*, which enforce output structure; *validity constraints*, which mask invalid answer; and *reasoning constraints*, which restrict reasoning trajectories. We theoretically show that appropriate constraints can preserve or improve performance by eliminating invalid paths. Across four model families and eleven open-source models, causal ablations on diverse benchmarks using state-of-the-art constrained decoding strategies reveal that apparent performance degradation largely stems from non-equivalent prompting and output formats rather than from decoder-level masking. Meanwhile, validity constraints improve reliability and extractability by eliminating invalid answers, while reasoning constraints can improve reasoning and, consequently, accuracy when they enforce beneficial trajectories. These findings show that the impact of constrained decoding depends on the type of constraint and the output space it removes. Our code is available online[1].

1 INTRODUCTION

LLMs have shown strong performance in tasks such as text completion, question answering, code generation, and data extraction (Minaee et al., 2024). Their capabilities can be extended by incorporating external systems through function calling, tool use, and agentic frameworks using JSON schemas or other formal formats (Schick et al., 2023; Patil et al., 2024). However, LLMs do not always produce structurally valid outputs, creating a growing need for reliable format control in industrial applications (Liu et al., 2024). To address this gap, grammar-constrained decoding methods ((Willard & Louf, 2023; Koo et al., 2024; Dong et al., 2024; guidanceAI, 2025)) mask invalid tokens during generation, ensuring syntactically valid outputs.

Recent observations show that such strict constraints can negatively affect LLM reasoning, leading to a significant drop in accuracy (Tam et al., 2024; Banerjee et al., 2025; Nguyen et al., 2026; Reddy et al., 2026; Lee et al., 2026). Several studies have investigated the causes of this degradation. Most studies attribute the performance drop to constrained decoding and propose solutions such as adding a `reason` field to JSON schema, two-step generation-then-formatting approaches (Tam et al., 2024; Reddy et al., 2026), and free reasoning followed by constrained decoding for final formatting via trigger tokens (Nguyen et al., 2026). Other studies, however, suggest that the drop is mainly caused at prompt-level (Lee et al., 2026) or unfair comparisons between constrained and unconstrained settings (dottxt, 2024).

To investigate these different perspectives, we separate constraints into three types: *format constraints*, which enforce a specific output format, such as JSON; *validity constraints*, which mask invalid continuations based on requirements such as data types (e.g., integers or strings), length; and *reasoning constraints*, which restrict the model's reasoning process.

[1]The code is included in the Supplementary Materials and will be publicly released upon acceptance.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

Theoretically, we show that constrained decoding does not harm LLM generation when the prompt and expected output format[2] remain unchanged. We further show that *format constraints* (FC) guarantee the expected output format; *validity constraints* (VC) provide a near-guaranteed, near-zero-cost gain by eliminating invalid outputs based on the task structure; and *reasoning constraints* (RC) can improve performance when the enforced reasoning trajectory is beneficial.

Experiments on eleven open-source models ranging from 1.7B to 32B across four model families—Qwen (Qwen Team, 2026), Gemma (Team et al., 2024), Llama (Dubey et al., 2024), and SmolLM (Bakouch et al., 2025)—suggest that, while a noticeable performance gap exists across different output formats, as observed in prior work (Tam et al., 2024; Banerjee et al., 2025; Nguyen et al., 2026; Reddy et al., 2026; Lee et al., 2026), this gap is mainly caused by prompt-level differences rather than constrained decoding (Lee et al., 2026). Under an apples-to-apples comparison with the same prompt and expected output format, constrained decoding consistently matches or outperforms unconstrained decoding, with validity constraints providing the most consistent gains (up to 17.4%) and reasoning constraints providing gains only when the enforced trajectory is itself beneficial.

Our main contributions are: (1) a taxonomy of three types of constraints: *format constraints* (FC), *validity constraints* (VC), and *reasoning constraints* (RC), which characterizes how constrained decoding can benefit LLM generation beyond format guarantees; (2) a theoretical proof that, under the same prompt and expected output format, constrained decoding does not harm LLM generation; (3) a comprehensive evaluation across models from different families and sizes and benchmarks ranging from standard reasoning tasks to competition-level problems, demonstrating the benefits of constrained decoding; (4) an extended dataset that further evaluates these benefits and shows that they remain valuable even for models of substantial size; and (5) a comprehensive analysis of the sources of performance gains from constrained decoding.

This paper is structured as follows: Section 2 reviews related work, Section 3 presents a theoretical proof of why constrained decoding is beneficial under the same prompt and expected output format, and Sections 4 and 5 present the experimental setup, results and discussion, respectively.

2 Related Work

Constrained decoding (Willard & Louf, 2023; Koo et al., 2024; Dong et al., 2024; guidanceAI, 2025) guarantees the required format without the need for fine-tuning by controlling the generation process from start to end. Constrained decoding is often assumed to be limited to structured formats such as JSON, driven by its common practical use cases.

Tam et al. (2024) show the performance degradation of constrained decoding compared with unconstrained decoding, specifically when reasoning within a JSON schema rather than reasoning freely. Banerjee et al. (2025) prove that finite output grammars without reasoning reduce model expressivity, while Reddy et al. (2026) attribute the degradation to distribution shift caused by constrained generation that enforces the output format.

Several strategies have been proposed to allow LLMs to reason while still producing an extractable format, ranging from adding a `reason` field to JSON, to two-step free reasoning followed by formatting, **NL-to-Format** (Tam et al., 2024; Reddy et al., 2026), or single-pass free reasoning followed by formatting through a trigger token mode switch, **InWriting** (Nguyen et al., 2026)

Our theoretical analysis shows that, when the prompt and expected output format are aligned with the grammar, constrained decoding excludes only invalid generations and should therefore not impair model performance. Empirically, while reasoning within JSON can underperform free-form reasoning, we find that, under matched prompts and output formats, constrained decoding consistently outperforms unconstrained decoding across the state-of-the-art (SOTA) decoding strategies we evaluate.

[2]The term *expected output format* refers to the format explicitly required by the prompt and accepted by the evaluation; see Section A.2 for details.

3 METHODOLOGY

In this section, we first study when constrained decoding can help LLMs produce better answers. In particular, we show that constrained decoding can improve answers both syntactically and semantically by removing invalid outputs and shifting probability toward valid ones.

We then study whether constrained decoding can also improve reasoning. We show that this is possible, but requires a suitable prior condition on the reasoning traces allowed by the constraint.

3.1 CONSTRAINED DECODING CAN HELP LLMS PRODUCE BETTER ANSWERS

We first show that, under the same conditions[3], applying constrained decoding to the answer generally leads to better performance than unconstrained decoding.

Let x be the input prompt, and R be a fixed reasoning trace.

Let $\mathcal{Y}$ be the set of all possible candidate final-answer token sequences of length T, generated from vocabulary V.

Let $Y^* \in \mathcal{Y}$ be the ground-truth correct final answer

Let $\mathcal{Y}_C \subseteq \mathcal{Y}$ denote the subset of all candidate answers that satisfy the predefined constraint C.

Let Y be a token sequence $s = (s_1, \ldots, s_T)$ representing the final answer, generated given the input prompt x and reasoning trace R. Since the model is autoregressive,

$$P_\theta(Y \mid x, R) = \prod_{t=1}^{T} P_\theta(s_t \mid x, R, s_{<t}). \tag{1}$$

At each step, constrained decoding masks invalid next tokens given the current prefix and renormalizes the distribution locally:

$$P_{\mathcal{C}}(s_t \mid x, R, s_{<t}) = \frac{P_\theta(s_t \mid x, R, s_{<t})\mathbf{1}[s_t \in V_C(s_{<t})]}{Z_t(x, R, s_{<t})}, \tag{2}$$

where

$$Z_t(x, R, s_{<t}) = \sum_{s'_t \in V_C(s_{<t})} P_\theta(s'_t \mid x, R, s_{<t}) \tag{3}$$

and $V_C(s_{<t})$ denotes the set of tokens allowed by C at that position.

At each step, constrained decoding masks invalid tokens and renormalizes the remaining distribution. We assume that, for every t, $V_C(s_{<t})$ contains at least one token with positive probability. Otherwise, $Z_t(x, R, s_{<t}) = 0$, meaning that no permissible token can be sampled at step t. Thus,

$$0 < Z_t(x, R, s_{<t}) \leq 1 \qquad \text{for all } t, x, R, s_{<t}. \tag{4}$$

Therefore, the probability of the complete constrained answer Y can be written as

$$P_{\mathcal{C}}(Y \mid x, R) = \prod_{t=1}^{T} P_{\mathcal{C}}(s_t \mid x, R, s_{<t}) = \frac{P_\theta(Y \mid x, R)\mathbf{1}[Y \in \mathcal{Y}_C]}{\prod_{t=1}^{T} Z_t(x, R, s_{<t})}. \tag{5}$$

as the product of indicators is equivalent to a single sequence-level constraint:

$$\prod_{t=1}^{T} \mathbf{1}[s_t \in V_C(s_{<t})] = \mathbf{1}[Y \in \mathcal{Y}_C]\,. \tag{6}$$

[3]The same prompt, expected output format, and reasoning traces

Accuracy Metric:

Define the accuracy indicator function $g : \mathcal{Y} \to \{0, 1\}$ as:

$$g(Y) = \mathbb{I}(Y = Y^*) = \begin{cases} 1 & \text{if } Y = Y^* \\ 0 & \text{if } Y \neq Y^* \end{cases}$$

The expected accuracy without constrained decoding is:

$$\mathbb{E}_{Y \sim P}[g(Y) \mid R, x] = \sum_{Y \in \mathcal{Y}} P(Y \mid R, x)\, \mathbb{I}(Y = Y^*) = P(Y^* \mid R, x). \tag{7}$$

The expected accuracy with constrained is:

$$\mathbb{E}_{Y \sim P_C}[g(Y) \mid R, x] = \sum_{Y \in \mathcal{Y}_C} P_C(Y \mid R, x) \cdot \mathbb{I}(Y = Y^*) \tag{8}$$

3.1.1 When constrained decoding does not hurt LLMs Performance

Assumption 1 – Soundness of output constraints. The ground-truth answer Y^* satisfies[4] the output constraint C.

$$Y^* \in \mathcal{Y}_C. \tag{9}$$

From Eq. 4, we have

$$0 < \prod_{t=1}^{T} Z_t(x, R, s_{<t}) \leq 1. \qquad \text{for all } t, x, R, s_{<t}. \tag{10}$$

Therefore, under Assumption 1, Eq. 8 becomes

$$\mathbb{E}_{Y \sim P_C}[g(Y) \mid R, x] = \frac{P(Y^* \mid R, x)}{\prod_{t=1}^{T} Z_t(x, R, s^*_{<t})} \geq P(Y^* \mid R, x). \tag{11}$$

where $s^*_{<t} = (s^*_1, \ldots, s^*_{t-1})$ denotes the prefix of length $t-1$ belonging to the ground-truth sequence $Y^* = (s^*_1, \ldots, s^*_T)$.

Assumption 1—the condition that C does not hurt LLM performance—is usually satisfied in practice by defining C to enforce only the output format while allowing the answer field to take any value. It can fail when C semantically excludes the correct ground-truth answer.

3.1.2 When constrained decoding improve LLMs Performance

For the ground-truth answer $Y^* = (s^*_1, \ldots, s^*_T)$, define the *path factors*

$$Z^*_t := Z_t(x, R, s^*_{<t}), \qquad Z^* := \prod_{t=1}^{T} Z^*_t. \tag{12}$$

Thus, Z^*_t is the total unconstrained probability mass of the tokens allowed by C after the prefix $s^*_{<t}$, and Z^* is the product of these masses along the path of Y^*.

Assumption 2 – Non-trivial masking on the path of Y^*.

There exists a step t and a token $v \neq s^*_t$ such that

$$v \notin V_C(s^*_{<t}) \quad \text{and} \quad P_\theta(v \mid x, R, s^*_{<t}) > 0.$$

[4]Here, "satisfies" means that Y^* is both semantically correct and syntactically valid under C. Thus, a semantically correct answer that cannot be extracted according to C is considered incorrect, details in Sec. A.4.3.

That is, at some prefix of Y^*, the mask removes an invalid token to which the model assigns positive probability.

Under Assumption 2,

$$Z_t^* = 1 - \sum_{v \notin V_C(s^*_{<t})} P_\theta(v \mid x, R, s^*_{<t}) < 1, \qquad \text{for that step } t. \tag{13}$$

Since every other factor is at most 1, $Z^* < 1$. Therefore, under Assumptions 1 and 2,

$$\mathbb{E}_{Y \sim P_C}[g(Y) \mid R, x] = \frac{P_\theta(Y^* \mid x, R)}{Z^*} > P_\theta(Y^* \mid x, R). \tag{14}$$

Conversely, if the constraint masks no token with positive probability along the path of Y^*, then $Z^* = 1$ and the two expectations are equal.

Equation 13 clarifies when constrained decoding can provide a benefit:

1. **Format constraint.** If the model already follows C, the constraint has no effect. If the model assigns positive probability to syntactically invalid tokens, such as a missing "{" or "`"`" required by C, constrained decoding removes this probability mass and increases the relative probability of Y^*.
2. **Validity constraint.** If C additionally incorporates semantic information about the ground-truth answer, more invalid tokens can be ruled out along the path of Y^*, making Z^* smaller and the resulting gain larger.

3.2 Can Constrained Decoding helps LLMs become a Better Reasoner

We extend the analysis of constrained decoding to reasoning traces.

Let Y^* denote the ground-truth answer, and R^* denote the set of reasoning traces that lead to Y^*.

Let C denote the constraint set over reasoning–answer pairs (R, Y). We consider three cases:

Case 1 — Total exclusion (hard elimination). If C completely excludes the good reasoning traces and correct answer,

$$(R^*, Y^*) \cap C = \varnothing,$$

Then the constraint admits neither better reasoning traces R^* nor the correct answer Y^*. Hence, no correct reasoning–answer pair can be generated, yielding zero expected correctness. Thus, constrained decoding strictly hurts performance whenever the unconstrained model assigns positive probability to a correct pair.

Case 2 — Oracle constraint (perfect retention). If C retains only the good reasoning traces and correct answer,

$$C = \{(R^*, Y^*)\},$$

then constrained decoding always generates a correct reasoning–answer pair. Therefore, it achieves perfect correctness and strictly outperforms unconstrained decoding whenever the latter assigns positive probability to an incorrect pair.

Case 3 — Partial retention (practical case).

The constraint retains some good reasoning–answer pairs while excluding others, which may include both good and bad pairs. Formally,

$$(R^*, Y^*) \cap C \neq \varnothing, \quad (R^*, Y^*) \not\subseteq C,$$
$$C \setminus (R^*, Y^*) \neq \varnothing.$$

Thus, both favorable and unfavorable reasoning–answer pairs remain possible, and the effect on expected correctness is indeterminate in general.

As shown in Equation 13, constrained decoding is more effective when we have greater knowledge of the reasoning paths that lead to the correct answer. Its goal is therefore to guide the model toward better reasoning–answer pairs, although such improvement is not guaranteed because the quality of the resulting reasoning cannot generally be predicted a priori.

4 EXPERIMENTAL SETUP

4.1 MODELS, DATASETS, PROMPTS VARIATIONS AND OUTPUT FORMATS

Models

We evaluate eleven open-source models in total. Ten models: Qwen3-1.7B, Qwen3-4B, Qwen3-8B, Qwen3.5-2B, Qwen3.5-4B, Qwen3.5-9B, Qwen3.5-27B, Llama3-8B, Gemma2-9B, and SmolLM3-3B, are evaluated for task solving, while Qwen3-32B is used as the extractor model in the two-step NL-to-Format approach. All models were released between 2024 and 2026. Detailed configurations are provided in Sec. A.3.

Datasets

The datasets span two levels of reasoning difficulty. Standard benchmarks include **GSM8K** (Cobbe et al., 2021), **Last Letter Concatenation** (Wei et al., 2022), which cover multi-step mathematical and symbolic reasoning tasks for which prior work (Tam et al., 2024) has reported performance degradation under constrained decoding.

Reasoning-intensive benchmarks include **AIME 2026**, challenging competition-level mathematics problems (Dekoninck et al., 2026), and **LastLetter-Extended**, our extension of Last Letter Concatenation from four to eight names. Both were released or constructed after the release of all models in our experiments, reducing the potential for data leakage. Dataset construction details are provided in Section A.3.3.

4.2 EVALUATION

We report **accuracy** as the primary metric, along with four additional metrics: **parse rate**, **format gain**, **validity gain**, and **reasoning gain**.

Accuracy Metrics. Accuracy is computed by exact match between extracted and ground-truth answers. We report mean $\pm$ std across seeds and task/format-prompt combinations (see Sec. A.2)

Parse Metrics. Parse rate measures the proportion of generated outputs that can be successfully parsed according to the specified format schema.

Format Gain. Format gain measures the improvement obtained by enforcing the required output format through constrained decoding. We denote this setting as **+FC**, indicating that *format constraints* (**FC**) are applied.

Validity Gain. Validity gain measures the improvement obtained by incorporating additional information about the viable answer subspace into the predefined valid schema. We denote this setting as **+VC**, indicating that *validity constraints* (**VC**) are applied.

Reasoning Gain. Reasoning gain measures the improvement in accuracy obtained by enforcing beneficial trajectories through constrained decoding. We denote this setting as **+RC**, indicating that *reasoning constraints* (**RC**) are applied.

4.3 OUTPUT FORMAT AND DECODING STRATEGIES

We experiment with current SOTA decoding strategies that support reasoning with constrained decoding: (1) **JSON**, which adds a `reason` field to the JSON output; (2) **NL-to-Format**, which uses two-step free reasoning followed by formatting (Tam et al., 2024; Reddy et al., 2026); and (3) **InWriting**, which uses single-pass free reasoning followed by formatting through a trigger-token mode switch (Nguyen et al., 2026). Each strategy specifies an expected output format in the prompt and a corresponding schema for constrained decoding. Details are provided in Sec. A.2.

We apply constrained decoding using Litelines, following prior work (Nguyen et al., 2026). Litelines allows models to perform free-form reasoning before switching to constrained decoding through

Table 1: LastLetter Task: comparison of unconstrained and combined Format and Validity Constraints (FC+VC) for plain-text answers using the NL-to-Format decoding strategy with the same prompt and reasoning traces. **Bold** values indicate the highest accuracy.

Model	NL (Baseline)	JSON +FC+VC	NL-to-Format Unconstr. (plain-text)	+FC+VC
Qwen3-1.7B	33.6 ± 21.9	45.4 ± 19.1	60.7 ± 4.3	**66.1** ± 4.8
Qwen3-4B	32.1 ± 26.2	41.9 ± 21.6	57.5 ± 7.5	**60.4** ± 7.1
Qwen3-8B	63.8 ± 17.4	78.4 ± 7.9	69.4 ± 11.4	**72.4** ± 10.6
Qwen3.5-2B	22.4 ± 29.9	69.1 ± 2.2	62.9 ± 5.6	**68.7** ± 5.2
Qwen3.5-4B	81.6 ± 1.6	71.6 ± 28	84.1 ± 2.3	**86.9** ± 2.6
Qwen3.5-9B	85.9 ± 1.2	77 ± 27	86.8 ± 1.2	**89.9** ± 0.5

trigger tokens and is built on `outlines-core`, the same underlying library used by Outlines (Willard & Louf, 2023). The format schemas and Litelines configurations are detailed in Sec. A.3.

In reporting our experimental results, we use the decoding method name to denote the expected output format. For unconstrained decoding, we append **Unconstr.**. For constrained decoding, we append the types of constraints applied, using **+FC**, **+VC**, and **+RC** for format, validity, and reasoning constraints, respectively. Multiple constraints combine labels, such as **+FC+VC**.

5 Results and Discussion

In this section, we first show that (1) constrained decoding is not limited to the JSON format, (2) under the same prompt and expected output format, constrained decoding consistently outperforms unconstrained decoding, not only by enforcing the output format but also by validating answers, and (3) constrained decoding can improve LLM performance by enforcing beneficial reasoning traces.

We then provide additional analyses of (4) the parsability of different decoding strategies, including JSON, InWriting, and NL-to-Format, (5) the sources of performance gains from Format Constraints (FC), Validity Constraints (VC), and Reasoning Constraints (RC), and (6) token efficiency across different decoding strategies.

5.1 Constrained Decoding Beyond JSON

Constrained decoding is not limited to JSON or other strict formats; it is a general logit-masking mechanism that also applies to plain-text generation. To demonstrate this, we compare it with the two-step NL-to-Format approach of Tam et al. (2024), which uses a second LLM to extract answers from the first LLM's reasoning as plain text. We adopt the same setup but apply constrained decoding to the second-step plain-text generation. As shown in Table 1 and 5), constrained decoding consistently outperforms unconstrained decoding, even for plain-text generation.

Few-shot examples improve pattern adherence and string-based extraction (see **NL (Baseline)** in Table 5), but offer limited reasoning benefits, consistent with prior work (Tam et al., 2024; Nguyen et al., 2026). They are also impractical for reasoning-intensive tasks such as AIME2026. We therefore use zero-shot settings for all subsequent experiments.

5.2 JSON as a Practical Interface for Information Extraction

While constrained decoding is not specific to JSON, JSON provides a structured interface for extracting and processing multiple fields. We thus compare NL-to-Format with plain text (original) and JSON. As shown in Table 2, JSON performs comparably to or better than plain text across all models, while using the same free-reasoning step (Natural Language—NL). Since structured outputs become more useful as the number of fields increases, we use JSON as the target format for the remaining experiments.

Table 2: LastLetter Task: comparison of plain-text and JSON answer formats using the NL-to-Format decoding strategy with the same free-reasoning (NL) traces, under unconstrained decoding and combined Format and Validity Constraints (FC+VC). **Bold** values indicate the highest accuracy.

Model	NL-to-Format Unconstr. (plain-text)	+FC+VC	NL-to-Format Unconstr. (JSON)	+FC+VC
Qwen3-1.7B	60.7 ± 4.3	66.1 ± 4.8	67.3 ± 4.1	**68.9** ± 4.6
Qwen3-4B	57.5 ± 7.5	60.4 ± 7.1	60.3 ± 6.4	**61.4** ± 6.5
Qwen3-8B	69.4 ± 11.4	72.4 ± 10.6	72.1 ± 10.5	**73.5** ± 10.6
Qwen3.5-2B	62.9 ± 5.6	68.7 ± 5.2	68.6 ± 5.9	**71.3** ± 5.4
Qwen3.5-4B	84.1 ± 2.3	86.9 ± 2.6	87.7 ± 1.5	**88.8** ± 1.5
Qwen3.5-9B	86.8 ± 1.2	89.9 ± 0.5	89.6 ± 1.0	**91.6** ± 1.0

5.3 When Constrained Decoding Improve LLM Performance

The performance gap between JSON and free-form reasoning approaches: InWriting and NL-to-Format, shown in Tables 1, 5, 6, 7, and 8 is consistent with prior work (Tam et al., 2024; Banerjee et al., 2025; Nguyen et al., 2026; Reddy et al., 2026; Lee et al., 2026). However, when the prompt and expected output format are fixed, the combined use of **Format Constraint (FC)** and **Validity Constraint (VC)** (+FC+VC) consistently outperforms unconstrained (see Tables 1, 5, 6, 7, 8, 9, and 10). This supports Lee et al. (2026), who attribute differences across output styles primarily to prompt-level factors.

To further investigate the source of these gains, we consider two levels of constraint: (1) **Format Constraint (FC)**, which ensures that answers follow the required format and are extractable, and (2) the additional **Validity Constraint (VC)**, which rules out invalid answers and can turn incorrect answers into correct ones. Results in Fig 1, Tables 9, and 10 show that VC provides more consistent gains than FC (up to 17.4%). This is because VC provides semantic information about the ground-truth answer, while FC mainly enforces the output format and has less effect when the model already follows the format instruction well. Details of the semantic constraints used in our experiments are provided in Sec. A.2, while details on how Validity Constraints help LLMs turn incorrect answers into correct ones are discussed in Sec. A.4.1, and demonstrate its potential for broader applications through a preliminary experiment on prohibited content filtering in Table 4 and Sec. A.4.2.

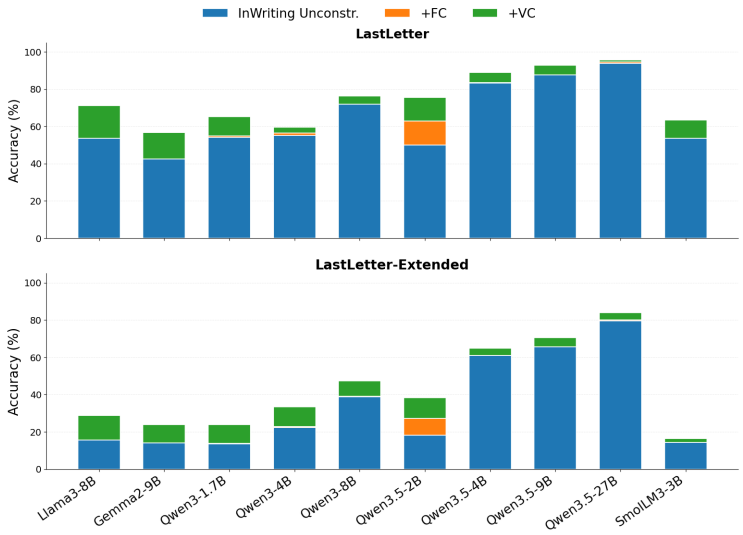


Figure 1: Comparison of the performance gains from Format Constraint (FC) and Validity Constraint (VC) on the LastLetter and LastLetter-Extended tasks.

5.4 Can Reasoning Constraints Benefit LLM Reasoning?

Beyond post-hoc validity constraints, reasoning constraints (**RC**) can potentially improve LLM reasoning and accuracy. As shown in Table 14 and 15, enforcing LLMs to begin with "Let's think step by step" generally improves performance on GSM8K and AIME2026, consistent with prior work (Kojima et al., 2022). However, the 6.6% degradation on AIME2026 with Qwen3.5-4B highlights that RC benefits are model- and task-dependent. This suggests that RC is not universally practical: (1) its benefits depend on the task and model; (2) specific constraints may exclude alternative valid reasoning paths; and (3) identifying beneficial traces may require unavailable oracle knowledge.

5.5 Parsability Analysis

We analyze the parsability of each decoding strategy in Tables 11 and 12, with a sample-level analysis provided in Sec. A.4.3. Across all strategies, constrained decoding consistently improves format completion compared with unconstrained decoding. This improvement remains consistent even for larger models, such as Qwen3-32B and Qwen3.5-27B, suggesting that constrained decoding provides reliable control over output formatting regardless of model size.

5.6 Token Efficiency Analysis

We report the generated token counts for each decoding method in Table 13.

Although additional tokens are modest compared with unconstrained decoding, token count does not fully capture computational cost. Constrained decoding requires determining the allowed tokens and applying token-level masking at each step; for JSON, this overhead begins at the start and persists throughout generation, growing with constraint complexity (Geng et al., 2025). NL-to-Format requires an additional LLM call for parsing, increasing computation and token usage. InWriting adds fewer than 15 tokens in our experiments and applies constrained decoding only to the final output, reducing constraint overhead while preserving free-form reasoning and providing a balance between reasoning flexibility and computational efficiency.

6 Conclusion

While the degradation between reasoning in a structured format (JSON) and free reasoning is well documented in prior work and observed in our experiments, we show theoretically and empirically that it arises from the prompt-level constraint rather than constrained decoding itself. In fact, constrained decoding can improve performance when the constraint is appropriately designed.

Beyond guaranteeing the output format, we show that *validity constraints* (VC) provide a near-guaranteed, near-zero-cost gain when the subspace containing valid answers can be characterized from the task structure. By eliminating invalid outputs, VC can improve both syntactic and semantic validity without requiring prior knowledge of the correct answer. This principle naturally extends to applications such as enforcing valid molecular structures in chemistry or filtering prohibited content.

Reasoning constraints (RC), however, show mixed results across models and tasks. They can improve performance when the enforced reasoning traces are beneficial, but degrade performance otherwise, highlighting the difficulty of identifying beneficial reasoning traces in practice.

Finally, constrained decoding introduces computational and token-efficiency costs that grow with structural complexity. Reasoning inside JSON requires constraints to be enforced throughout generation, while NL-to-Format requires a second parsing step and nearly doubles token usage. Mode switching through trigger tokens, such as InWriting, offers a lower-overhead alternative that preserves free-form reasoning while constraining only the desired output format.

This work has limitations that motivate future research across a broader range of domains, tasks, and constraint types, as discussed in Sec. A.1.

AI USE STATEMENT

In this work, we used generative AI tools, primarily the free versions of ChatGPT and Claude, to draft and edit parts of the manuscript for improved readability.

All research questions, ideas, concepts, and experiments were developed and conducted by the authors.

We take full responsibility for the final content of this work, including all text, claims, and artifacts produced with the assistance of generative AI.

REPRODUCIBILITY STATEMENT

Clarification of Assumptions 1 and 2 in Sec. 3 Assumption 1 guarantees non-degradation, while both Assumptions 1 and 2 are required for a performance gain. Both are practical to satisfy. For Assumption 1, we set all fields (e.g., in JSON) to `str` under the same prompt and expected output format, allowing all possible generations. Thus, if the LLM already follows the predefined structure, constrained decoding has no effect.

Assumption 2 is similarly practical. If the LLM generates an invalid syntax token, such as {" or "", the unconstrained output becomes unextractable, while constrained decoding prevents such tokens and preserves extractability. Importantly, the potential gain increases when more information about the correct answer space is known in advance through the given task, question, or other context.

Clarification on LastLetter-Extended Dataset Construction Details are provided in Sec A.3.3 and included in the Supplementary Materials.

REFERENCES

Elie Bakouch, Loubna Ben Allal, Anton Lozhkov, Nouamane Tazi, Lewis Tunstall, Carlos Miguel Patiño, Edward Beeching, Aymeric Roucher, Aksel Joonas Reedi, Quentin Gallouédec, Kashif Rasul, Nathan Habib, Clémentine Fourrier, Hynek Kydlicek, Guilherme Penedo, Hugo Larcher, Mathieu Morlon, Vaibhav Srivastav, Joshua Lochner, Xuan-Son Nguyen, Colin Raffel, Leandro von Werra, and Thomas Wolf. SmolLM3: smol, multilingual, long-context reasoner. `https://huggingface.co/blog/smollm3`, 2025.

Debangshu Banerjee, Tarun Suresh, Shubham Ugare, Sasa Misailovic, and Gagandeep Singh. Crane: Reasoning with constrained llm generation. *arXiv preprint arXiv:2502.09061*, 2025.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, et al. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.

Jasper Dekoninck, Nikola Jovanović, Tim Gehrunger, Kári Rögnvaldsson, Ivo Petrov, Chenhao Sun, and Martin Vechev. Beyond benchmarks: Matharena as an evaluation platform for mathematics with llms. 2026. URL `https://arxiv.org/abs/2605.00674`.

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. *arXiv preprint arXiv:2411.15100*, 2024.

dottxt. Say what you mean: A response to "let me speak freely", 2024. URL `https://blog.dottxt.ai/say-what-you-mean.html`.

Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. The llama 3 herd of models. *arXiv e-prints*, pp. arXiv–2407, 2024.

Saibo Geng, Hudson Cooper, Michał Moskal, Samuel Jenkins, Julian Berman, Nathan Ranchin, Robert West, Eric Horvitz, and Harsha Nori. Generating structured outputs from language models: Benchmark and studies. *arXiv preprint arXiv:2501.10868*, 2, 2025.

guidanceAI. guidance: A guidance language for controlling large language models. GitHub repository, 2025. URL https://github.com/guidance-ai/guidance.

Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. Large language models are zero-shot reasoners. *Advances in neural information processing systems*, 35:22199–22213, 2022.

Terry Koo, Frederick Liu, and Luheng He. Automata-based constraints for language model decoding. *arXiv preprint arXiv:2407.08103*, 2024.

Ivan Yee Lee, Loris D'Antoni, and Taylor Berg-Kirkpatrick. The format tax. *arXiv preprint arXiv:2604.03616*, 2026.

Michael Xieyang Liu, Frederick Liu, Alexander J Fiannaca, Terry Koo, Lucas Dixon, Michael Terry, and Carrie 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*, pp. 1–9, 2024.

Andrew L. Maas, Raymond E. Daly, Peter T. Pham, Dan Huang, Andrew Y. Ng, and Christopher Potts. Learning word vectors for sentiment analysis. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, pp. 142–150, Portland, Oregon, USA, June 2011. Association for Computational Linguistics. URL http://www.aclweb.org/anthology/P11-1015.

Shervin Minaee, Tomas Mikolov, Narjes Nikzad, Meysam Chenaghlu, Richard Socher, Xavier Amatriain, and Jianfeng Gao. Large language models: A survey. *arXiv preprint arXiv:2402.06196*, 2024.

Ngoc Trinh Hung Nguyen, Alonso Silva, Laith Zumot, Liubov Tupikina, Armen Aghasaryan, and Mehwish Alam. Thinking before constraining: A unified decoding framework for large language models. *arXiv preprint arXiv:2601.07525*, 2026.

Shishir G Patil, Tianjun Zhang, Xin Wang, and Joseph E Gonzalez. Gorilla: Large language model connected with massive apis. *Advances in Neural Information Processing Systems*, 37:126544–126565, 2024.

Qwen Team. Qwen3.5: Towards native multimodal agents, February 2026. URL https://qwen.ai/blog?id=qwen3.5.

Avinash Reddy, Thayne T Walker, James S Ide, and Amrit Singh Bedi. The hidden cost of structured generation in llms: Draft-conditioned constrained decoding. *arXiv preprint arXiv:2603.03305*, 2026.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. *Advances in neural information processing systems*, 36:68539–68551, 2023.

Zhi Rui Tam, Cheng-Kuang Wu, Yi-Lin Tsai, Chieh-Yen Lin, Hung-yi Lee, and Yun-Nung Chen. Let me speak freely? a study on the impact of format restrictions on large language model performance. In Franck Dernoncourt, Daniel Preoţiuc-Pietro, and Anastasia Shimorina (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/.

Gemma Team, Morgane Riviere, Shreya Pathak, Pier Giuseppe Sessa, Cassidy Hardin, Surya Bhupatiraju, Léonard Hussenot, Thomas Mesnard, Bobak Shahriari, Alexandre Ramé, et al. Gemma 2: Improving open language models at a practical size. *arXiv preprint arXiv:2408.00118*, 2024.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.

Brandon T Willard and Rémi Louf. Efficient guided generation for large language models. *arXiv preprint arXiv:2307.09702*, 2023.

A Appendix

A.1 Limitations and Future Work

Besides the well-known benefit of constrained decoding in guaranteeing the expected output format, our findings highlight two additional benefits: *validity constraints* (VC), which provide a near-guaranteed, near-zero-cost gain by eliminating invalid outputs, and *reasoning constraints* (RC), whose benefits depend on whether the enforced reasoning trajectory is beneficial.

For VC, our main limitation is the narrow experimental scope. While VC can naturally extend to domains such as chemistry, where constraints can enforce valid molecular structures, and content filtering, where constraints can prevent prohibited words or patterns, we evaluate VC primarily on LastLetter and LastLetter-Extended, with a preliminary content-filtering experiment on IMDb. Expanding VC to these domains remains an important direction for future work.

For RC, we show that enforcing beneficial reasoning paths can improve reasoning and answer quality. However, identifying such paths requires prior information, so our experiments demonstrate feasibility using patterns from prior work rather than providing practical guidance for finding them.

Lastly, our experiments use relatively simple JSON formats, consistent with prior work. More complex tasks that require both extensive reasoning and complex output formats remain an important direction for future work.

A.2 Prompt, Expected Format, and Constraint Variations

We use a prompt template consisting of a task description, format description, few-shot examples, and a question, following prior work (Tam et al., 2024; Nguyen et al., 2026).

Prompt Structure

```
{task_description}
{format_description}
{fewshots}
{question}
```

We use 3 task-description variations per task, except AIME2026, which uses 1. JSON uses 1 fixed format, while other settings use 3. Each AIME2026 prompt combination is evaluated with 3 seeds.

- **GSM8K, LastLetter, LastLetter-Extended – JSON.** $3 \times 1 = 3$ prompt combinations.
- **GSM8K, LastLetter, LastLetter-Extended – InWriting, NL-to-Format.** $3 \times 3 = 9$ prompt combinations.
- **AIME2026 – InWriting.** $1 \times 3 = 3$ prompt combinations $\times$ 3 seeds $= 9$ evaluation runs.

A.2.1 Prompt Variants

For **GSM8K**, **LastLetter**, and **LastLetter-Extended**, we use the same task-description variations across all three decoding strategies: **JSON**, **InWriting**, and **NL-to-Format/NL-to-JSON**, following prior work (Tam et al., 2024; Nguyen et al., 2026), as follows:

Task-description Variations for GSM8K

Variation 1
Follow the instruction to complete the task.
You are a math tutor who helps students of all levels understand and solve mathematical problems.
Read the last question carefully and think step by step before answering, the final answer must be only a number.

Variation 2
Follow the instruction to complete the task.
Read the last question carefully and think step by step before answering, the final answer must be only a number.
You are a math tutor who helps students of all levels understand and solve mathematical problems.

Variation 3
Follow the instruction to complete the task.
Mathematical problem-solving task:
Given: A mathematical question or problem
Required: A numerical answer only
Role: You are a math tutor assisting students of all levels
Process: Think step by step to solve the problem
Note: Read the question carefully before beginning your analysis.

Task-description Variations for LastLetter (Both original and Extended) – Used for all decoding strategies

Variation 1
Follow the instruction to complete the task.
You are given a string of words and you need to take the last letter of each word and concatenate them.
Read the last question carefully and think step by step before answering.

Variation 2
Follow the instruction to complete the task.
Read carefully for each of the last question and think step by step before answering.
You are given a string of words and you need to take the last letter of each word and concatenate them.

Variation 3
Follow the instruction to complete the task.
String manipulation task:
Given: a sequence of words.
Required: A new string made from the last letter of each word.
Process: Think step by step to solve this challenge.
Note: Ensure you've read the question thoroughly before beginning.

For **AIME2026**, we adopt the task description from Dekoninck et al. (2026), with one modification: we remove the instruction "Put your final answer within \boxed{{}}." This format is specific to answer extraction in the original work and is not compatible with all of our decoding strategies.

Task Description for AIME2026

The answer is an integer between 0 and 999, inclusive.

For Format description, we apply the same for all tasks based on the expected output format of each decoding strategies:

For **JSON** decoding strategy:

Format Specification for JSON Decoding

Provide your output in the following valid JSON format:

```
{
"reason": <think step by step>,
"answer": <answer>
}
```

For **NL-to-Format** decoding strategy:

Format Variations for NL-to-Format Decoding Strategies

Variation 1
Instruct: Provide your output in the following text format:
Answer: <reasoning first>. The final answer is <answer>

Variation 2
Instruct: Provide your output in the following text format:
Step by step reasoning: ...
Answer: The final answer is ...

Variation 3
Instruct: Provide your output in the following text format:
Answer: <think step by step>. The final answer is <answer>

For **InWriting** decoding strategy, we keep the same variations as **NL-to-Format**, with an additional phrase added to each variation to specify the expected output format for the **InWriting** decoding strategy.

Format Variations for InWriting Decoding Strategies

Variation 1
Instruct: Provide your output in the following text format:
Answer: <reasoning first>. The final answer is <answer>

```
Then, provide the final answer in valid JSON format:
{"answer": <answer>}
```

Variation 2
Instruct: Provide your output in the following text format:
Step by step reasoning: ...
Answer: The final answer is ...

```
Then, provide the final answer in valid JSON format:
{"answer": <answer>}
```

Variation 3
Instruct: Provide your output in the following text format:
Answer: <think step by step>. The final answer is <answer>

```
Then, provide the final answer in valid JSON format:
{"answer": <answer>}
```

A.2.2 Expected Format and Constraint Variations

Expected Format. The expected format differs across strategies. For **JSON**, the expected format is a JSON object with `reason` and `answer` fields. For **InWriting**, it consists of free-form reasoning followed by a JSON `answer`, while for **NL-to-Format**, it consists of free-form reasoning in the first step followed by a JSON `answer` in the second. We evaluate each strategy in two settings: *unconstrained*, where only the expected format is specified, and *constrained*, where the format is additionally enforced through constrained decoding.

Constraint Variations. For *format constraints* (**FC**), the goal is to enforce only the output format. We therefore set `answer` to `str`, allowing any possible answer to be generated. For *validity constraints* (**VC**), we additionally encode prior knowledge about the potential answer space, based on the task or question, directly into the output schema. For *reasoning constraints* (**RC**), we enforce beneficial reasoning traces by encoding the desired reasoning pattern into the constraint.

The VC configurations used in our experiments are:

- **GSM8K**: The answer is restricted to a non-negative integer.
- **AIME2026**: The answer is restricted to an integer between 0 and 999.
- **LastLetter-Original**: The answer is restricted to exactly four alphabetic characters.
- **LastLetter-Extended**: The answer is restricted to exactly eight alphabetic characters.

Output Schema Construction. We use Pydantic[5] to define and customize the output schemas. The **JSON** strategy uses two fields, `reason` and `answer`, while **InWriting** and **NL-to-Format** use only the `answer` field. For example, the Pydantic schema for the JSON strategy is:

```
class Schema(BaseModel):
reason: str
answer: int = Field(ge=0, le=9_999_999_999)
```

The `reason` field is fixed as a string, while `answer` is customized per task. For *validity constraints*, we additionally restrict the answer space by encoding the constraints directly in the field definition, e.g., `int = Field(ge=0, le=9_999_999_999)`. For *format constraints*, we simply set `answer` to `str`.

[5] https://docs.pydantic.dev/latest/

A.3 Experimental Setup

A.3.1 Code

Our code is provided in the Supplementary Material in accordance with the double-blind review policy. The code will be made publicly available upon acceptance.

A.3.2 Datasets

All datasets, except for LastLetter-Extended, are open-source and can be directly accessed through Hugging Face Datasets (https://huggingface.co/datasets).

A.3.3 LastLetter-Extended Dataset Construction

We construct LastLetter-Extended by randomly sampling names from the `names-dataset` library (https://pypi.org/project/names-dataset/), following prior work (Wei et al., 2022; Kojima et al., 2022). We make a minor modification to the sampling procedure to exclude compound names, such as "Juan Pablo," which are treated as a single name in the original dataset without an explicit indication that they consist of multiple names.

A.3.4 Constrained Decoding Configurations

We use Litelines[6] for implementing constrained decoding. It provides three key controls: `allow_preamble` for free generation before a trigger, `max_preamble_tokens` to limit unconstrained generation, and `trigger_token_ids` to specify trigger tokens. Following (Nguyen et al., 2026), we use EOS and PAD as triggers. These controls enable **InWriting** to switch from free to constrained generation after a trigger.

An example configuration for **InWriting** is:

```
allow_preamble=True,
max_preamble_tokens=600,
trigger_token_ids=[tokenizer.eos_token_id, tokenizer.pad_token_id]
```

For **JSON** and **NL-to-Format**, we set `allow_preamble=False` to enforce direct generation in the specified format.

A.3.5 Experimental Configurations

The configurations of the LLMs used across all experiments are detailed in Table 3.

Table 3: Experimental configurations for the two experimental settings.

Experiment Type	Parameter	Value	Note
Standard Setting	Temperature	0	
	Sampling	False	
	Max Reasoning Tokens	600	
	Thinking Mode	Off	
	Random Seed	1	
Extended-Reasoning Setting (AIME-2026 only)	Temperature	Default	
	Sampling	Default	
	Max Reasoning Tokens	12800	
	Thinking Mode	Off	
	Random Seed	42, 43, 44	

[6] `https://github.com/alonsosilvaallende/litelines/`

A.4 Full Experimental Results and Analysis

A.4.1 Sample-Level Gain Analysis from Validity Constraints Across Tasks

For numerical tasks such as GSM8K, validity constraints primarily improve performance by standardizing answers. For example, `28.00` is normalized to the required representation, `28`, allowing the correct answer to be matched successfully.

For string-manipulation tasks: LastLetter and LastLetter-Extended, validity constraints correct formatting artifacts and deviations between the answer and the reasoning trace. We observe several representative cases.

First, validity constraints can recover characters that are incorrectly omitted from the answer:

```
"Model output": "Answer: To solve this problem, I will iterate through each word in the given
     string, extract the last letter of each word, and concatenate them. \n\nThe given string
     is \"Camilo Becky Eliza Rebecca\". \n\nThe last letters of each word are: o, y, a, a.\n\
     nThe final answer is \"oya\".\n\n{\"answer\": \"oya\"}",
"Unconstrained parsed_answer": "oya",
"Constrained parsed_answer": "oyaa",
```

The reasoning trace identifies the last letters as `o`, `y`, `a`, and `a`, indicating `oyaa` as the intended answer. The unconstrained parser omits the final `a`, while the constrained parser recovers the answer.

A similar effect occurs when the model produces repeated characters in the answer:

```
"Model output": "Answer: To solve this problem, I will iterate through each word in the given
     string, extract the last letter of each word, and concatenate them. \n\nThe given string
     is \"Selina Tasha Jarrod Maddy\". \n\nThe last letters of each word are: a, a, d, y.\n\
     nThe final answer is \"aaday\".\n\n{\"answer\": \"aaday\"}",
"Unconstrained parsed_answer": "aaday",
"Constrained parsed_answer": "aady",
```

Here, the reasoning trace identifies the last letters as `a`, `a`, `d`, and `y`, yielding `aady`. The unconstrained parser retains an additional `a`, while the constrained parser removes it.

Second, validity constraints can remove formatting artifacts such as spaces between characters:

```
"Model output": "Answer: To solve this problem, I will iterate through each word in the given
     string, extract the last letter of each word, and concatenate them. \n\nThe given string
     is \"Yamileth Dane Aron Dee\". \n\nThe last letters of each word are: \n- Yamileth: H\n-
     Dane: E\n- Aron: N\n- Dee: E\n\nThe final answer is: H E N E\n\n{\"answer\": \"H E N E
     \"}",
"Unconstrained parsed_answer": "h e n e",
"Constrained parsed_answer": "hene"
```

In this example, the model correctly identifies `H`, `E`, `N`, and `E`, but inserts spaces between them. The unconstrained parser preserves these spaces, producing `h e n e`, while the constrained parser removes them, producing `hene`.

While these gains may appear to result from simple post-hoc processing, the token-level behavior reveals a more substantive role for VC. In these examples, spaces and additional characters are not necessarily independent tokens; rather, they are part of tokens such as `' e'`, `' n'`, and `'da'`. Thus, under VC, the model must select tokens that satisfy the validity constraint while remaining consistent with the reasoning trace. From this perspective, VC acts as a post-hoc consistency enforcement rather than merely a processing step, guiding the model toward a valid continuation based on both its reasoning trace and the validity constraints. This perspective extends constrained decoding beyond format enforcement to a broader range of applications, such as enforcing valid molecular structures in chemistry and filtering prohibited content.

A.4.2 Preliminary Experiment on Prohibited Content Filtering with IMDB

Motivated by findings regarding validity constraints as a post-hoc consistency enforcement, we conduct a preliminary experiment on the IMDb Large Movie Review Dataset (Maas et al., 2011). We investigate whether an LLM can perform sentiment classification when the standard labels, *Positive* and *Negative*, are specified in the prompt but prohibited from generation.

We evaluate three conditions: (1) an **Unconstrained Baseline** with no decoding restrictions; (2) **Label-Constrained Decoding**, where the output is restricted to *Positive* or *Negative*; and (3)

Alternative-Constrained Decoding, where the output is restricted to a predefined set of sentiment-related alternatives:

- **Negative:** Bad, Critical, Unfavorable, Disappointing, Unsatisfactory.
- **Positive:** Good, Favorable, Impressive, Satisfying, Enjoyable.

The third condition tests whether the model can retain the correct sentiment when constrained to alternative expressions rather than the original class labels.

Prompt for IMDb

> Classify the sentiment of the following text as either Positive or Negative.
> Return exactly one word: Positive or Negative.
> Do not include explanations, punctuation, or any other text.

Table 4: IMDb sentiment classification under different decoding constraints, comparing the Unconstrained Baseline, Label-Constrained Decoding, and Alternative-Constrained Decoding.

Model	Unconstrained Baseline	Label-Constrained	Alternative-Constrained
Qwen3-1.7B	81.6	81.6	76.0
Qwen3-4B	92.6	92.6	83.4
Qwen3.5-2B	89.6	89.6	80.8
Qwen3.5-4B	94.6	94.8	85.8

As shown in Table 4, although Alternative-Constrained Decoding does not match Label-Constrained Decoding, it maintains substantial performance across models. This suggests that LLMs can preserve the intended answer using alternative expressions when the canonical label is prohibited, which is promising for content filtering.

A.4.3 Sample-Level Analysis of Failed Parses

While InWriting and NL-to-Format achieve relatively high parse rates without constrained decoding, JSON-format generation suffers substantially in the absence of constraints (see Table 11 and 12). Beyond cases where the expected JSON structure is incorrect or missing, we identify two main output patterns in the JSON strategy that lead to parsing failures and need to be discussed further.

Case 1: Free-form Generation with JSON at the End


```
"Model output": "### Step-by-step Explanation:\n\n1. **Identify the input string**: \n The
    input string is `\"Homoh Aniba Grewar Manying Mordoch Bjarndis Leveron Rahymoff\"`.\n\n2.
     **Split the string into individual words**: \n The string is split into words using
    spaces as delimiters: \n `[\"Homoh\", \"Aniba\", \"Grewar\", \"Manging\", \"Mordoch\", \"
    Bjarndis\", \"Leveron\", \"Rahymoff\"]`.\n\n3. **Extract the last letter of each word**:
    \n For each word, take the last character:\n - \"Homoh\" \u2192 \"h\"\n - \"Aniba\" \
    u2192 \"a\"\n - \"Grewar\" \u2192 \"r\"\n - \"Manging\" \u2192 \"g\"\n - \"Mordoch\" \
    u2192 \"h\"\n - \"Bjarndis\" \u2192 \"s\"\n - \"Leveron\" \u2192 \"n\"\n - \"Rahymoff\" \
    u2192 \"f\"\n\n4. **Concatenate the last letters**: \n Combine the extracted letters into
     a single string: \n `\"hagr hsf n f\"`.\n\n### Final Answer:\n\n```json\n{\n \"reason\":
     \"The input string is split into individual words, and the last letter of each word is
    extracted and concatenated.\",\n \"answer\": \"hagr hsf n f\"\n}\n```"
```


Case 2: JSON wrapped in a Markdown code fence


```
"Model output": "```json\n{\n\"reason\": \"To solve this, I will identify each word in the
    provided string 'Lalesca Hasiic Elrayan Weixu Alzadgali Samphoas Stierlin Thwmas' and
    extract its last letter. The words and their last letters are: 'Lalesca' -> 'a', 'Hasiic'
     -> 'c', 'Elrayan' -> 'n', 'Weixu' -> 'u', 'Alzadgali' -> 'i', 'Samphoas' -> 's', '
    Stierlin' -> 'n', 'Thwmas' -> 's'. Finally, I will concatenate these letters in order: 'a
    ' + 'c' + 'n' + 'u' + 'i' + 's' + 'n' + 's', resulting in 'acnuisns'.\",\n\"answer\": \"
    acnuisns\"\n}\n```\n"
```


It may seem harsh at first to classify both cases as failures, as their outputs are still extractable, particularly Case 2, where JSON wrapped in a Markdown code fence is commonly accepted. How-

ever, neither case follows the output format specified in the prompt or the schema enforced by constrained decoding. If Case 2 is considered acceptable, the schema must explicitly allow this format; otherwise, constrained decoding would eliminate all paths beginning with ```` ```json ````, contradicting Assumption 1 that constrained decoding does not hurt LLM performance (see Sec. 3). These cases highlight the need to understand constrained decoding and carefully design schemas that allow all output formats that we consider correct.

A.4.4 Full Experimental Results

Table 5: LastLetter Task: comparison of unconstrained and combined Format and Validity Constraints (FC+VC) for plain-text answers using the NL-to-Format decoding strategy with the same prompt and reasoning traces. **Bold** values indicate the highest accuracy.

n_{shots}	Model	NL (Baseline)	JSON +FC+VC	NL-to-Format Unconstr. (plain-text)	+FC+VC
0	Qwen3-1.7B	33.6 ± 21.9	45.4 ± 19.1	60.7 ± 4.3	**66.1** ± 4.8
	Qwen3-4B	32.1 ± 26.2	41.9 ± 21.6	57.5 ± 7.5	**60.4** ± 7.1
	Qwen3-8B	63.8 ± 17.4	78.4 ± 7.9	69.4 ± 11.4	**72.4** ± 10.6
	Qwen3.5-2B	22.4 ± 29.9	69.1 ± 2.2	62.9 ± 5.6	**68.7** ± 5.2
	Qwen3.5-4B	81.6 ± 1.6	71.6 ± 28	84.1 ± 2.3	**86.9** ± 2.6
	Qwen3.5-9B	85.9 ± 1.2	77 ± 27	86.8 ± 1.2	**89.9** ± 0.5
1	Qwen3-1.7B	54.1 ± 4.6	35.8 ± 24.6	58 ± 3.1	**62.2** ± 2.5
	Qwen3-4B	60.1 ± 10.5	21.1 ± 26.6	67.6 ± 4.7	**69.9** ± 3.9
	Qwen3-8B	65.1 ± 8.4	64.4 ± 20.3	67.2 ± 6.7	**69.7** ± 6.6
	Qwen3.5-2B	17.4 ± 22.4	56.6 ± 1.7	39.6 ± 19	**42.6** ± 20.6
	Qwen3.5-4B	81.9 ± 2.3	63.2 ± 35.1	82.7 ± 1.9	**85.6** ± 1.9
	Qwen3.5-9B	85.5 ± 1.9	81.3 ± 27.3	86 ± 1.5	**88.3** ± 1.1
4	Qwen3-1.7B	55 ± 3.4	39 ± 19.7	56.7 ± 3.5	**61.3** ± 4
	Qwen3-4B	75.5 ± 2.9	27.6 ± 31	76.4 ± 3.4	**78.4** ± 2.4
	Qwen3-8B	72.7 ± 2.4	77.5 ± 3.4	73.9 ± 2.3	**75.9** ± 2.9
	Qwen3.5-2B	58.7 ± 2.7	63.5 ± 1.1	60.1 ± 3.1	**66.5** ± 2.1
	Qwen3.5-4B	82 ± 1.4	63.8 ± 34.4	83.4 ± 1.1	**85.8** ± 0.9
	Qwen3.5-9B	87.8 ± 0.9	82.1 ± 26.2	88.7 ± 1.6	**90.8** ± 1.4

Table 6: GSM8K: comparison of unconstrained (Unconstr.) and combined Format and Validity Constraints (FC+VC) across JSON, InWriting, and NL-to-Format decoding strategies. **Bold** values indicate the highest accuracy.

Model	JSON Unconstr.	+FC +VC	InWriting Unconstr.	+FC +VC	NL-to-Format Unconstr.	+FC +VC
Llama3-8B	42.1 ± 11.8	71.8 ± 1.6	74.7 ± 1.3	**77.8** ± 1.2	76.4 ± 1.1	77.1 ± 1.2
Gemma2-9B	0 ± 0	86.7 ± 0.3	82.9 ± 1.4	**86.7** ± 0.6	83.6 ± 1.6	84.4 ± 1.3
Qwen3-1.7B	17.6 ± 24.8	63.4 ± 0.5	68.8 ± 8.5	**75.8** ± 0.8	72.9 ± 1.1	74.8 ± 1.1
Qwen3-4B	85.3 ± 1.2	88.3 ± 0.2	85.8 ± 3.0	**89.7** ± 0.4	86.0 ± 1.1	87.4 ± 0.8
Qwen3-8B	56.7 ± 22.4	90.9 ± 0.4	89.4 ± 0.9	**91.4** ± 0.5	88.8 ± 1.6	90.1 ± 1.2
Qwen3.5-2B	61 ± 1.2	69.9 ± 0.9	47.1 ± 28.4	**74.9** ± 2.2	69.0 ± 4.6	72.1 ± 4.0
Qwen3.5-4B	88.8 ± 2.2	90.6 ± 0.4	88.2 ± 1.5	**91.8** ± 0.4	88.7 ± 1.7	90.6 ± 0.9
Qwen3.5-9B	91.2 ± 1.5	91.9 ± 0.4	90.4 ± 1.8	**93.9** ± 0.3	91.1 ± 1.6	92.6 ± 1.0
SmolLM3-3B	76.7 ± 1.6	77.1 ± 1.5	78.8 ± 2.5	**82.8** ± 0.6	81.8 ± 1.3	82.4 ± 1.3

Table 7: LastLetter Task: comparison of unconstrained (Unconstr.) and combined Format and Validity Constraints (FC+VC) across JSON, InWriting, and NL-to-Format decoding strategies. **Bold** values indicate the highest accuracy. *X* indicates that the corresponding experiment is unavailable.

Model	JSON Unconstr.	+FC+VC	InWriting Unconstr.	+FC+VC	NL-to-Format Unconstr.	+FC+VC
Llama3-8B	0 ± 0	2.0 ± 1.4	53.6 ± 5.4	71.0 ± 3.8	68.7 ± 4.4	**71.3** ± 4.0
Gemma2-9B	0 ± 0	43.8 ± 2.2	42.5 ± 4.1	**56.6** ± 4.4	51 ± 5.8	54.1 ± 5.0
Qwen3-1.7B	0 ± 0	9.8 ± 1.1	54.2 ± 4.8	65.2 ± 5.1	67.3 ± 4.1	**68.9** ± 4.6
Qwen3-4B	55.6 ± 5.7	56.7 ± 5.5	55.2 ± 8.0	59.6 ± 7.3	60.3 ± 6.4	**61.4** ± 6.5
Qwen3-8B	75.6 ± 2.8	79.6 ± 2.8	72.0 ± 4.9	76.4 ± 5.6	72.1 ± 10.5	73.5 ± 10.6
Qwen3.5-2B	60.9 ± 1.7	69.1 ± 2.8	50.1 ± 21.9	**75.5** ± 2.7	68.6 ± 5.9	71.3 ± 5.4
Qwen3.5-4B	83.8 ± 1.1	87.3 ± 0.9	83.3 ± 1.2	**88.8** ± 1.4	87.7 ± 1.5	**88.8** ± 1.5
Qwen3.5-9B	23.1 ± 16.4	88.9 ± 0.3	87.6 ± 0.9	**92.8** ± 0.9	89.6 ± 1.0	91.6 ± 1.0
Qwen3.5-27B	90.4 ± 2.2	93.3 ± 0.5	93.9 ± 0.8	**95.7** ± 0.3	X	X
SmolLM3-3B	0.2 ± 0.3	0.2 ± 0.3	53.6 ± 3.5	**63.4** ± 5.2	59.3 ± 9.7	62.4 ± 10.0

Table 8: LastLetter-Extended Task: comparison of unconstrained (Unconstr.) and combined Format and Validity Constraints (FC+VC) across JSON, InWriting, and NL-to-Format decoding strategies. **Bold** values indicate the highest accuracy. *X* indicates that the corresponding experiment is unavailable.

Model	JSON Unconstr.	+FC+VC	InWriting Unconstr.	+FC+VC	NL-to-Format Unconstr.	+FC+VC
Llama3-8B	0 ± 0	2.4 ± 3.5	15.5 ± 3.6	28.7 ± 5.8	28.8 ± 9.6	**31.9** ± 10.4
Gemma2-9B	0 ± 0	17.6 ± 0.3	14.0 ± 1.8	23.9 ± 4.1	24.0 ± 3.1	**25.6** ± 3.0
Qwen3-1.7B	0 ± 0	0 ± 0	13.5 ± 4.8	23.8 ± 6.9	27.0 ± 5.6	**28.4** ± 5.3
Qwen3-4B	15.3 ± 0.0	19.1 ± 0.6	22.4 ± 6.4	**33.4** ± 7.8	26.1 ± 7.7	27.5 ± 7.3
Qwen3-8B	34.4 ± 2.7	39.6 ± 3.5	38.7 ± 5.4	**47.3** ± 6.2	42.1 ± 7.2	43.5 ± 6.9
Qwen3.5-2B	15.1 ± 4.4	20.4 ± 5.4	18.1 ± 9.3	**38.4** ± 8.2	34.4 ± 11.3	36.5 ± 10.0
Qwen3.5-4B	61.1 ± 2.6	63.1 ± 3.1	61.0 ± 2.5	**64.8** ± 2.1	58.4 ± 1.9	59.6 ± 2.2
Qwen3.5-9B	41.3 ± 29.0	62.7 ± 5.8	65.7 ± 1.4	**70.5** ± 1.4	66.7 ± 2.8	68.2 ± 3.3
Qwen3.5-27B	69.3 ± 15.1	81.1 ± 1.4	79.5 ± 1.9	**83.9** ± 1.4	X	X
SmolLM3-3B	0 ± 0	0 ± 0	14.4 ± 5.9	16.5 ± 6.5	17.9 ± 6.9	**18.8** ± 7.3

Table 9: LastLetter Task: decomposing constrained decoding gains relative to unconstrained (Unconstr.) into Format Constraint (FC) and Validity Constraint (VC) across JSON and InWriting decoding strategies.

Model	JSON Unconstr.	+FC	+FC+VC	InWriting Unconstr.	+FC	+FC+VC
Llama3-8B	0 ± 0	1.8 ± 1.6	2.0 ± 1.4	53.6 ± 5.4	53.6 ± 5.4	71.0 ± 3.8
Gemma2-9B	0 ± 0	35.8 ± 2.7	43.8 ± 2.2	42.5 ± 4.1	42.6 ± 4.0	56.6 ± 4.4
Qwen3-1.7B	0 ± 0	8.4 ± 1.4	9.8 ± 1.1	54.2 ± 4.8	55.0 ± 5.4	65.2 ± 5.1
Qwen3-4B	55.6 ± 5.7	55.8 ± 5.6	56.7 ± 5.5	55.2 ± 8.0	56.4 ± 6.9	59.6 ± 7.3
Qwen3-8B	75.6 ± 2.8	75.6 ± 2.8	79.6 ± 2.8	72.0 ± 4.9	72.0 ± 4.9	76.4 ± 5.6
Qwen3.5-2B	60.9 ± 1.7	61.6 ± 2.7	69.1 ± 2.8	50.1 ± 21.9	63.0 ± 2.2	75.5 ± 2.7
Qwen3.5-4B	83.8 ± 1.1	83.8 ± 1.1	87.3 ± 0.9	83.3 ± 1.2	83.6 ± 1.4	88.8 ± 1.4
Qwen3.5-9B	23.1 ± 16.4	86.7 ± 0.5	88.9 ± 0.3	87.6 ± 0.9	87.6 ± 0.9	92.8 ± 0.9
Qwen3.5-27B	90.4 ± 2.2	92.7 ± 0.9	93.3 ± 0.5	93.9 ± 0.8	94.7 ± 0.4	95.7 ± 0.3
SmolLM3-3B	0.2 ± 0.3	0.2 ± 0.3	0.2 ± 0.3	53.6 ± 3.5	53.6 ± 3.5	63.4 ± 5.2

Table 10: LastLetter-Extended Task: decomposing constrained decoding gains relative to unconstrained (Unconstr.) into Format Constraint (FC) and Validity Constraint (VC) across JSON and InWriting decoding strategies.

Model	JSON Unconstr.	+FC	+FC+VC	InWriting Unconstr.	+FC	+FC+VC
Llama3-8B	0 ± 0	1.8 ± 2.5	2.4 ± 3.5	15.5 ± 3.6	15.5 ± 3.6	28.7 ± 5.8
Gemma2-9B	0 ± 0	15.6 ± 0.6	17.6 ± 0.3	14.0 ± 1.8	14.0 ± 1.8	23.9 ± 4.1
Qwen3-1.7B	0 ± 0	0 ± 0	0 ± 0	13.5 ± 4.8	13.8 ± 4.9	23.8 ± 6.9
Qwen3-4B	15.3 ± 0.0	15.8 ± 0.3	19.1 ± 0.6	22.4 ± 6.4	22.8 ± 6.5	33.4 ± 7.8
Qwen3-8B	34.4 ± 2.7	34.4 ± 2.7	39.6 ± 3.5	38.7 ± 5.4	39.0 ± 5.4	47.3 ± 6.2
Qwen3.5-2B	15.1 ± 4.4	15.1 ± 4.4	20.4 ± 5.4	18.1 ± 9.3	27.2 ± 6.6	38.4 ± 8.2
Qwen3.5-4B	61.1 ± 2.6	61.1 ± 2.6	63.1 ± 3.1	61.0 ± 2.5	61.0 ± 2.5	64.8 ± 2.1
Qwen3.5-9B	41.3 ± 29.0	60.9 ± 5.8	62.7 ± 5.8	65.7 ± 1.4	65.7 ± 1.4	70.5 ± 1.4
Qwen3.5-27B	69.3 ± 15.1	79.3 ± 1.1	81.1 ± 1.4	79.5 ± 1.9	80.0 ± 2.0	83.9 ± 1.4
SmolLM3-3B	0 ± 0	0 ± 0	0 ± 0	14.4 ± 5.9	14.4 ± 5.9	16.5 ± 6.5

Table 11: GSM8K: Parse sucess rate analysis across JSON, InWriting, and NL-to-Format decoding strategies, comparing unconstrained with Format Constraints (FC).

Model	JSON Unconstr.	+FC	InWriting Unconstr.	+FC	NL-to-Format Unconstr.	+FC
Llama3-8B	58.7	99.9	99.4	100	99.3	100
Gemma2-9B	0.2	100	99.8	100	99.7	100
Qwen3-1.7B	28.5	97.3	90.7	100	97.9	100
Qwen3-4B	96.7	99.8	97.3	100	98.2	100
Qwen3-8B	62.6	99.8	99.7	100	98.6	100
Qwen3.5-2B	82.2	88.4	63.1	99.6	95.9	100
Qwen3.5-4B	98.6	98.6	95.6	99.8	97.9	100
Qwen3.5-9B	98.2	98.2	96.3	99.9	98.2	100
SmolLM3-3B	98.4	99.5	99.2	100	99.4	100

Table 12: LastLetter-Extended: Parse sucess rate analysis across JSON, InWriting, and NL-to-Format decoding strategies, comparing unconstrained with Format Constraints (FC). *X* indicates that the corresponding experiment is unavailable.

Model	JSON Unconstr.	+FC	InWriting Unconstr.	+FC	NL-to-Format Unconstr.	+FC
Llama3-8B	0	100	99.8	100	99.6	100
Gemma2-9B	0	100	100	100	100	100
Qwen3-1.7B	0.7	96.2	97.4	100	99.6	100
Qwen3-4B	100	100	99.7	100	100	100
Qwen3-8B	100	100	99.5	100	99.6	100
Qwen3.5-2B	99.6	99.6	67.9	99.6	99.6	100
Qwen3.5-4B	93.8	94.0	92.1	100	99.9	100
Qwen3.5-9B	60.4	99.3	93.1	99.9	99.8	100
Qwen3.5-27B	82.4	93.8	88.7	99.9	X	X
SmolLM3-3B	97.6	99.8	99.5	99.9	99.7	100

Table 13: Comparison of generated output token counts for JSON, InWriting, and NL-to-Format decoding strategies, under unconstrained and combined Format and Validity Constraints (FC+VC).

Tasks	Model	JSON Unconstr.	+FC +VC	InWriting Unconstr.	+FC +VC	NL-to-Format Unconstr.	+FC +VC
GSM8K	Qwen3-1.7B	221.1	129.2	189.5	201.1	190.4	190.2
	Qwen3-4B	135.8	135.0	164.6	172.8	161.4	161.3
	Qwen3-8B	223.0	159.0	176.4	188.2	174.4	174.3
	Qwen3.5-2B	242.2	242.8	278.4	286.9	304.2	302.2
	Qwen3.5-4B	164.2	163.5	272.8	280.6	269.8	269.7
	Qwen3.5-9B	184.7	183.7	277.5	285.2	259.5	259.4
LastLetter	Qwen3-1.7B	192.0	73.4	145.5	157.1	151.7	151.8
	Qwen3-4B	101.4	101.5	112.6	119.8	108.0	108.1
	Qwen3-8B	109.4	109.4	127.4	137.7	129.1	129.2
	Qwen3.5-2B	152.8	152.5	181.3	187.8	155.0	154.5
	Qwen3.5-4B	152.8	144.7	189.4	195.7	192.0	192.0
	Qwen3.5-9B	135.1	132.0	178.3	184.1	167.0	167.0

Table 14: GSM8K performance under the InWriting decoding strategy: comparison of Format and Validity Constraints (FC+VC) with FC+VC combined with Reasoning Constraints (FC+VC+RC). **Bold** values indicate the highest accuracy in each row.

Model	+FC+VC	+FC+VC +RC
Qwen3-1.7B	75.8 ± 0.8	**77.7** ± 0.9
Qwen3-4B	**89.7** ± 0.4	89.6 ± 0.3
Qwen3-8B	91.4 ± 0.5	**91.9** ± 0.6
Qwen3.5-2B	74.9 ± 2.2	**77.4** ± 0.5
Qwen3.5-4B	**91.8** ± 0.4	91.5 ± 0.5
Qwen3.5-9B	**93.9** ± 0.3	**93.9** ± 0.3
SmolLM3-3B	82.8 ± 0.6	**83.7** ± 0.6

Table 15: AIME2026 performance under the InWriting decoding strategy: comparison of Format and Validity Constraints (FC+VC) with FC+VC combined with Reasoning Constraints (FC+VC+RC). **Bold** values indicate the highest accuracy in each row.

Model	+FC+VC	+FC+VC +RC
Qwen3-4B	14.8 ± 4.5	**16.7** ± 2.7
Qwen3-8B	12.2 ± 2.2	**13.3** ± 3.1
Qwen3.5-4B	**62.2** ± 3.1	55.6 ± 5.7
Qwen3.5-27B	81.9 ± 3.9	**84.3** ± 5.6