Language Models Need Some Space

On the Sensitivity of Constrained Decoding to Completeness

Jahrim Gabriele Cesario
University of St. Gallen
St. Gallen, Switzerland
jahrimgabriele.cesario@unisg.ch

ABSTRACT

The widespread adoption of language models for code generation makes the enforcement of formal guarantees on their outputs ever more important. Recent constrained decoding techniques are promising, but when applied to practical programming languages they typically face a trade-off between soundness and completeness, either permitting some invalid programs or rejecting some valid ones. We argue that invalid programs should be excluded entirely, yet find that overly restrictive acceptance criteria can significantly impair a model's ability to follow human intent and generate functionally correct code. We attribute this degradation to a mismatch between the model's internal representation of the language and the constrained language induced by the formal guarantees. To better align the two, we propose reducing variability through training-set preprocessing, language design, or neural architecture design.

1 BACKGROUND

We describe a formal model of decoding for language models, based on prior work [18]. Language models generate *tokens* from a token vocabulary $\mathcal{V}$, and each token corresponds to a *string*, which is a sequence of symbols drawn from a character vocabulary Σ. We assume that the token vocabulary $\mathcal{V}$ is *complete*, meaning that any string $\omega \in \Sigma^*$ can be represented as a finite sequence of tokens $t_1, \ldots, t_n \in \mathcal{V}^*$. More precisely, a *tokenizer* is a pair of functions

$$\mathcal{T}_\Sigma : \Sigma^* \to \mathcal{V}^* \qquad \mathcal{T}_\mathcal{V} : \mathcal{V}^* \to \Sigma^*$$

that map strings to token sequences ($\mathcal{T}_\Sigma$), and vice versa ($\mathcal{T}_\mathcal{V}$).

Given the previous definitions, a *language model* is a function

$$\mathcal{M} : \mathcal{V}^* \to \Delta(\mathcal{V})$$

that, given a prefix $t_1, \ldots, t_n \in \mathcal{V}^*$, produces a probability distribution over the continuation token $t_{n+1} \in \mathcal{V}$.

Decoding is the process of generating a sequence from a language model by iteratively sampling each token t_{n+1} from the model's distribution conditioned on the preceding context $t_1, \ldots, t_n$. We define the decoding step as the function

$$\mathcal{D} : \mathcal{V}^n \to \mathcal{V}^{n+1}$$
$$\mathcal{D}(p) = p + x \qquad x \sim \mathcal{M}(p)$$

where x is the chosen continuation for the prefix p, $+$ denotes concatenation, and $\sim$ denotes sampling from a probability distribution. Then, decoding is the function composition $\mathcal{D}^i(p)$, which appends the next i continuations to the prefix p. Normally, decoding is bounded by additional stopping criteria, such as generating an end-of-sequence token $EOS \in \mathcal{V}$.

Recently, *constrained decoding* has been proposed as an extension of decoding that enforces some property on the generated string, e.g., syntactic and type correctness with respect to a programming language. Such properties are checked by a *constrainer* predicate

$$C : \mathcal{V}^* \to \mathbb{B}$$

that is satisfied by only prefixes of desirable sequences. The insight of constrained decoding is that, instead of sampling the space denoted by $\mathcal{V}^*$ for a valid sequence using $\mathcal{D}^*$ repeatedly (i.e., *rejection sampling*), one decoding step can prune the search space as soon as one continuation precludes the satisfiability of the constraints in C. Formally, the constrained decoding step is the function

$$(\mathcal{D} \mid C)(p) = p + x, \quad x \sim \mathcal{M}(p) \mid C(p + x)$$

where x is the first continuation sampled without repetitions that satisfies the constraints of C. Classical decoding is recovered when C is the tautological predicate $C_\top(p) = \top$.

Listing 1 summarizes the decoding algorithm and highlights in yellow the extensions for constrained decoding. Given the existence of the tokenizer $\mathcal{T}$, we can reason in terms of strings for simplicity. We can also generalize the algorithm to an abstract generator `G` that, given a prefix `p`, produces an iterator over the possible continuations, in the order they should be sampled. A large language model is only a specific instance of such a generator, where the order of the continuations is determined by its output distribution, the sampling strategy, the stopping criteria, and other parameters.

Listing 1: Decoding algorithm for an abstract generator.

```
G: Generator, C: Constrainer
fun decode(p: String): String =
  continuations: Iterator[String] = G(p)
  for t in continuations do
    p': String = p + t
    if not C(p') then continue
    return if t == EOS then p' else decode(p')
  error "No valid continuation found"
```

2 LIMITATIONS OF CONSTRAINING

Compared to rejection sampling, constraining can improve efficiency, but it can also alter the model's distributions. In general, global conditioning is not necessarily equivalent to local conditioning, that is $(\mathcal{D}^n \mid C) \neq (\mathcal{D} \mid \hat{C})^n$ even when $C = \hat{C}$. More precisely, manipulating the probability distributions of language models introduces *biases*, which distort the knowledge of the models and can degrade their performance [2, 3, 12, 15, 22]. For example, the constrainer might prefer low-probability continuations, which are associated with text degeneration and repetitions [6, 11, 13].

Biases can also be introduced by specific kinds of constrainers. For example, syntactic constrainers can limit continuations to the next lexemes accepted by a given grammar [5, 8, 22, 23, 28, 29, 32]. However, forcing the vocabulary of the language model to match the lexemes of the grammar can introduce biases due to approximation

Table 1: Experimental results for the sensitivity to completeness. In bold, the best of each metric.

Decoding	Functional	Compilable	Max Tokens	Timeout	Confidence	Output Tokens	Time (s)
unconstrained	0.59	0.92	**0.00**	**0.00**	**0.99**	81.14	**12.04**
constrained	**0.63**	**1.00**	**0.00**	**0.00**	0.97	78.14	13.40
no number formats	**0.63**	0.99	**0.00**	**0.00**	0.97	78.22	14.20
no multiline strings	**0.63**	0.99	**0.00**	**0.00**	0.97	78.22	15.88
no dotted keys	**0.63**	**1.00**	**0.00**	**0.00**	0.97	78.33	15.62
no comment	**0.63**	0.99	**0.00**	**0.00**	0.97	78.39	14.87
no array tables	0.60	0.93	**0.00**	**0.00**	0.96	**75.45**	15.09
no inline tables	0.59	0.98	**0.00**	**0.00**	0.95	78.50	17.67
no space before equal	0.48	0.78	0.22	**0.00**	0.72	170.92	30.68
no optional spaces	0.39	0.90	0.09	**0.00**	0.46	90.23	16.83

(i.e., *token misalignment*) [24, 27, 28]. Another example is type constrainers, which are type-checkers on program prefixes [1, 17, 18]. Sufficiently complex type systems require choosing between *soundness* and *completeness* of the constrainer (i.e., $C \neq \hat{C}$), either rejecting valid continuations or permitting invalid ones [17, 18, 31].

We argue that the value in constrained decoding is to provide guarantees on the output of language models. In fact, if the goal was to only improve conformance to a set of constraints, then *fine-tuning* on a dataset of valid sequences would be sufficient. Therefore, we consider soundness a necessary property for constrainers, while completeness is a desirable property. Since prioritizing soundness will inevitably decrease completeness for certain constrainers, we study the latter as a primary source of bias in the following section.

3 SENSITIVITY TO COMPLETENESS

To assess the sensitivity of constrained decoding to completeness, we evaluate a sound and complete constrainer for code generation and compare its performance to that of several incomplete variants.

Constrainers. As a case study, we consider a syntactic constrainer that is complete with respect to the TOML language specification [26]. We choose TOML because it is widely adopted, has a simple syntax and semantics, and includes several derived forms. These properties make TOML a good fit for our study: the model is likely to understand it well, and we can easily derive incomplete variants by removing some syntactic features or derived forms of the language, without losing expressivity. In particular, we consider the following restrictions of the complete constrainer:

- *No Number Formats*: forbid binary, octal, and hex formats.
- *No Multiline Strings*: forbid multiline strings.
- *No Dotted Keys*: forbid dotted keys (e.g., `a.b.c=d`).
- *No Comments*: forbid comments (e.g., `#comment`).
- *No Array Tables*: forbid array tables (e.g., `[[array]]`).
- *No Inline Tables*: forbid inline tables (e.g., `a={b=c}`).
- *No Spaces Before Equal*: forbid spaces before the `=` symbol.
- *No Optional Spaces*: forbid all optional spaces.

All constrainers are implemented as GGML BNF grammars [9], which are used by llama-cpp to perform constrained decoding.

Benchmarks. To reduce the risk of benchmark contamination, we build a new benchmark, TomlModeEval. We start from the JsonModeEval dataset [19], used for evaluating SynCode [28]. JsonModeEval contains 100 JSON solutions, each paired with a specification consisting of a JSON schema and natural language instructions. The task is to generate the JSON solution from its specification. In TomlModeEval, we replace the JSON solutions with their equivalent TOML solutions and keep the same specifications, except for minimal adjustments (e.g., asking for TOML instead of JSON).

Results. Table 1 shows the results of the experiment using Qwen3-Coder-30B-A3B-Instruct, a 30B mixture-of-experts code model with 3.3B active parameters and 256K context, trained for coding tasks. For decoding, we use greedy sampling with a temperature of 0.1. We consider the following metrics for the generated programs:

- *Functional*: ratio of programs that are functionally correct, i.e., match the AST of the solution exactly, ignoring key order.
- *Compilable*: ratio of programs that are syntactically and type-correct. For TOML, type-correctness is uniqueness of keys. Notably, our constrainers only ensure syntactic correctness.
- *Max Tokens*: ratio of programs that are truncated by the token limit, i.e., the EOS token is not generated before 300 tokens.
- *Timeout*: ratio of programs that are truncated by timeout, i.e., after 5 minutes of decoding.
- *Confidence*: average per-token probability in the programs.
- *Output Tokens*: average token count in the programs.
- *Time*: average time for decoding a program.

We report averages over 10 runs for *unconstrained* and complete *constrained* decoding, and the aforementioned incomplete variants.

Observations. Compared to unconstrained decoding, we can see that the complete constrainer improves functional correctness and compilability of the generated programs by 4% and 8% points, respectively. We observe a slight 2% points decrease in confidence, which is expected as the constrainer forces the model to discard probable but invalid continuations. We expect low confidence to lead to text degeneration and repetitions, which are correlated with the number of incomplete programs and output tokens. However, in this case, the impact of the constrainer on confidence is not significant enough to harm the performance of the model.

Comparing the incomplete constrainers with the complete one, we can see that incompleteness leads to a significant decrease in functional correctness. The most extreme case is the constrainer with *no optional spaces*, showing a decrease of 24% points in functional correctness. Incompleteness also correlates with a decrease in confidence and an increase in truncated programs, which are signs of text degeneration and repetitions. The most extreme case

is the constrainer with *no spaces before equal*, showing an increase of 22% points in truncated programs. This is surprising, given that the incomplete constrainer only differs from the complete one by a single space, indicating that even minor errors in the constrainer can significantly degrade performance. Finally, we observe that it is hard to predict the impact of a feature on the performance of the model, as it depends on the internal representation of the language learned by the model, which itself depends on the training data and the architecture of the model. For example, the results for functional correctness show that optional spaces are crucial to the model's representation of TOML, while other features such as dotted keys are not as important. This suggests that the model's learned representation of the language is very different from the language specification used to design the constrainers.

Manual Inspection. A manual inspection of the generated programs reveals that the unconstrained model confuses the syntax of TOML with JSON, which is prevented by the constrainers. We also observe that constraining can affect generated programs long after a constraint is enforced. For example, in the following excerpt, enforcing no optional spaces causes the model to later generate a list of objects instead of the required list of strings.

```
# unconstrained (correct)
facilityId = "BLD-4021"
urgencyLevel = "high"
requestedServices = [
    "HVAC repair", ...
]
```

```
# no optional spaces
facilityId="BLD-4021"
urgencyLevel="high"
[[requestedServices]]
value="HVAC repair"
...
```

Finally, we observe cases of repetition in which constrained decoding appears to get stuck in a loop. In the excerpt below, forbidding spaces before the equal sign causes the model to repeatedly emit spaces after a key. This occurs because spaces can still follow a key if it is extended into a dotted key, but generating a dotted key would preclude functional correctness. Thus, a dotted key is likely a low-probability continuation for the model and generating spaces is coincidentally more likely with greedy decoding.

```
# unconstrained (correct)
SKU = "TOB-1928"
```

```
# no spaces before equal
SKU    \t\t\t\t\t\t\t...
```

Summary. Consistent with prior work [17, 18, 28], we find that constrained decoding can improve the functional correctness of generated code. However, small losses in completeness can significantly degrade performance, often in unpredictable ways.

Threats to Validity. An extensive evaluation should consider different model families and sizes, temperatures, and sampling strategies, as well as lesser-known languages and more expressive ones.

4 MITIGATING INCOMPLETENESS

When the completeness of a constrainer cannot be improved directly, the only alternative is to adapt the languages on which constrained decoding operates. For the same expressivity, reducing variability in the constrained language and the model's learned representation should reduce their difference, thereby increasing completeness. This intuition is consistent with prior work showing that code models are sensitive to semantics-preserving syntactic changes [20, 30]. We outline three promising research directions.

Training Set. The first direction is to reduce variability at the level of the training data. Rather than adopting a new language, one can preprocess the training set so that the model is exposed to a canonical version of the language that is closer to the constrainer. A simple example is formatting all programs according to a unique style, thereby removing redundant surface forms, such as alternative layouts. More generally, one could canonicalize programs before training so that semantically equivalent fragments are syntactically similar. This view is supported by recent work indicating that formatting variability contributes little to training quality, as well as by training pipelines that normalize code [21, 25].

Programming Languages. A second direction is to design languages with less variability, i.e., admitting fewer equivalent surface forms for the same semantics by construction. This should facilitate defining constrainers and training a model to learn a representation that is faithful to the intended semantics of the language. Recent work reports that DSLs can outperform general-purpose languages for code generation, supporting this perspective [16]. One can also explore languages that trade expressivity for learnability and completeness, to identify the most effective point in the design space for LLM-based constrained code generation.

Network Architectures. A third direction is to design neural architectures that are inherently more robust to language variability. Current language models operate primarily over token sequences, which makes them sensitive to syntactic details that are often irrelevant to program semantics. An alternative is to build models whose representations are organized around more structured views of code. Recent work suggests that structure-aware representations such as ASTs and grammars can improve code generation [10, 14], indicating that robustness to syntactic variability also depends on the architecture's representational bias.

5 SCOPE AND APPLICABILITY

The proposed directions are intended specifically for code generation rather than for code understanding tasks such as code completion. Accordingly, we do not expect them to improve existing developer–AI collaboration workflows. Our vision is a language designed for language models to synthesize code that is guaranteed to satisfy the requirements enforced by a user-defined constrainer.

Orthogonal directions include more efficient constrainers (e.g., GPU-accelerated [7]); adapting unconstrained frameworks to constrained decoding (e.g., self-debugging [4]); or studying the granularity of constraint enforcement (e.g., every k tokens).

6 CONCLUSIONS

We studied the role of soundness and completeness in constrained decoding for code generation. While soundness is needed to rule out invalid outputs, our results show that small losses in completeness can impose a large and unpredictable cost on generation quality. This suggests a discrepancy between the internal representation of the language learned by the model and the language induced by the constrainer. We propose reducing this gap by lowering variability in both languages and identify three research directions: canonicalizing training data, designing less variable languages, and developing models that are more robust to syntactic variation.

ACKNOWLEDGMENTS

This work has been co-funded by the Swiss National Science Foundation (SNSF, Grant No. 10001777), by ArmaSuisse Science and Technology, and by European Union's Horizon research and innovation program (CAPE Project, Grant No. 101189899).

REFERENCES

[1] Lakshya A Agrawal, Aditya Kanade, Navin Goyal, Shuvendu Lahiri, and Sriram Rajamani. 2023. Monitor-Guided Decoding of Code LMs with Static Analysis of Repository Context. In *Advances in Neural Information Processing Systems*, A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine (Eds.), Vol. 36. Curran Associates, Inc., 32270–32298. https://proceedings.neurips.cc/paper_files/paper/2023/file/662b1774ba8845fc1fa3d1fc0177ceeb-Paper-Conference.pdf

[2] Kareem Ahmed, Kai-Wei Chang, and Guy Van den Broeck. 2025. Controllable Generation via Locally Constrained Resampling. In *The Thirteenth International Conference on Learning Representations*. https://openreview.net/forum?id=8g4XgC8HPF

[3] Debangshu Banerjee, Tarun Suresh, Shubham Ugare, Sasa Misailovic, and Gagandeep Singh. 2025. CRANE: Reasoning with constrained LLM generation. In *Forty-second International Conference on Machine Learning*. https://openreview.net/forum?id=wKs9fHYxCV

[4] Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2024. Teaching Large Language Models to Self-Debug. In *The Twelfth International Conference on Learning Representations*. https://openreview.net/forum?id=KuPixIqPiq

[5] Yixin Dong, Charlie F. Ruan, Yaxing Cai, Ziyi Xu, Yilong Zhao, Ruihang Lai, and Tianqi Chen. 2025. XGrammar: Flexible and Efficient Structured Generation Engine for Large Language Models. In *Proceedings of Machine Learning and Systems*, M. Zaharia, G. Joshi, and Y. Lin (Eds.), Vol. 7. MLSys. https://proceedings.mlsys.org/paper_files/paper/2025/file/5c20ca4b0b20b0bd2f1d839dc605e70f-Paper-Conference.pdf

[6] Zihao Fu, Wai Lam, Anthony Man-Cho So, and Bei Shi. 2021. A Theoretical Analysis of the Repetition Problem in Text Generation. *Proceedings of the AAAI Conference on Artificial Intelligence* 35, 14 (May 2021), 12848–12856. doi:10.1609/aaai.v35i14.17520

[7] Ashkan Vedadi Gargary, Soroosh Safari Loaliyan, and Zhijia Zhao. 2025. cuJSON: A Highly Parallel JSON Parser for GPUs. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1* (USA) *(ASPLOS '26)*. Association for Computing Machinery, New York, NY, USA, 85–100. doi:10.1145/3760250.3762222

[8] Saibo Geng, Martin Josifoski, Maxime Peyrard, and Robert West. 2023. Grammar-Constrained Decoding for Structured NLP Tasks without Finetuning. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, Houda Bouamor, Juan Pino, and Kalika Bali (Eds.). Association for Computational Linguistics, Singapore, 10932–10952. doi:10.18653/v1/2023.emnlp-main.674

[9] ggml-org. 2026. GGML BNF Grammars. https://github.com/ggml-org/llama.cpp/blob/master/grammars/README.md. GitHub, Accessed 2026-03-12.

[10] Linyuan Gong, Mostafa Elhoushi, and Alvin Cheung. 2024. AST-T5: Structure-Aware Pretraining for Code Generation and Understanding. arXiv:2401.03003 [cs.SE] https://arxiv.org/abs/2401.03003

[11] Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. 2020. The Curious Case of Neural Text Degeneration. arXiv:1904.09751 [cs.CL] https://arxiv.org/abs/1904.09751

[12] Alexander K. Lew, Tan Zhi-Xuan, Gabriel Grand, and Vikash K. Mansinghka. 2023. Sequential Monte Carlo Steering of Large Language Models using Probabilistic Programs. arXiv:2306.03081 [cs.AI] https://arxiv.org/abs/2306.03081

[13] Huayang Li, Tian Lan, Zihao Fu, Deng Cai, Lemao Liu, Nigel Collier, Taro Watanabe, and Yixuan Su. 2023. Repetition In Repetition Out: Towards Understanding Neural Text Degeneration from the Data Perspective. In *Advances in Neural Information Processing Systems*, A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine (Eds.), Vol. 36. Curran Associates, Inc., 72888–72903. https://proceedings.neurips.cc/paper_files/paper/2023/file/e6c2e85db1f1039177c4495ccd399ac4-Paper-Conference.pdf

[14] Qingyuan Liang, Zhao Zhang, Zeyu Sun, Zheng Lin, Qi Luo, Yueyi Xiao, Yizhou Chen, Yuqun Zhang, Haotian Zhang, Lu Zhang, Bin Chen, and Yingfei Xiong. 2025. Grammar-Based Code Representation: Is It a Worthy Pursuit for LLMs?. In *Findings of the Association for Computational Linguistics: ACL 2025*, Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar (Eds.). Association for Computational Linguistics, Vienna, Austria, 15640–15653. doi:10.18653/v1/2025.findings-acl.807

[15] Benjamin Lipkin, Benjamin LeBrun, Jacob Hoover Vigly, João Loula, David R. MacIver, Li Du, Jason Eisner, Ryan Cotterell, Vikash Mansinghka, Timothy J. O'Donnell, Alexander K. Lew, and Tim Vieira. 2025. Fast Controlled Generation from Language Models with Adaptive Weighted Rejection Sampling. arXiv:2504.05410 [cs.CL] https://arxiv.org/abs/2504.05410

[16] Saif Khalfan Saif Al Mazrouei. 2025. Anka: A Domain-Specific Language for Reliable LLM Code Generation. arXiv:2512.23214 [cs.CL] https://arxiv.org/abs/2512.23214

[17] Niels Mündler, Jingxuan He, Hao Wang, Koushik Sen, Dawn Song, and Martin Vechev. 2025. Type-Constrained Code Generation with Language Models. *Proc. ACM Program. Lang.* 9, PLDI, Article 171 (June 2025), 26 pages. doi:10.1145/3729274

[18] Shaan Nagy, Timothy Zhou, Nadia Polikarpova, and Loris D'Antoni. 2026. ChopChop: A Programmable Framework for Semantically Constraining the Output of Language Models. *Proc. ACM Program. Lang.* 10, POPL, Article 66 (Jan. 2026), 28 pages. doi:10.1145/3776708

[19] NousResearch. 2025. NousResearch/json-mode-eval. Hugging Face dataset repository. https://huggingface.co/datasets/NousResearch/json-mode-eval Accessed 2026-03-12.

[20] Pedro Orvalho and Marta Kwiatkowska. 2025. Are Large Language Models Robust in Understanding Code Against Semantics-Preserving Mutations? arXiv:2505.10443 [cs.SE] https://arxiv.org/abs/2505.10443

[21] Dangfeng Pan, Zhensu Sun, Cenyuan Zhang, David Lo, and Xiaoning Du. 2025. The Hidden Cost of Readability: How Code Formatting Silently Consumes Your LLM Budget. arXiv:2508.13666 [cs.SE] https://arxiv.org/abs/2508.13666

[22] Kanghee Park, Jiayu Wang, Taylor Berg-Kirkpatrick, Nadia Polikarpova, and Loris D' Antoni. 2024. Grammar-Aligned Decoding. In *Advances in Neural Information Processing Systems*, A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang (Eds.), Vol. 37. Curran Associates, Inc., 24547–24568. doi:10.52202/079017-0774

[23] Kanghee Park, Timothy Zhou, and Loris D'Antoni. 2025. Flexible and Efficient Grammar-Constrained Decoding. arXiv:2502.05111 [cs.CL] https://arxiv.org/abs/2502.05111

[24] Gabriel Poesia, Alex Polozov, Vu Le, Ashish Tiwari, Gustavo Soares, Christopher Meek, and Sumit Gulwani. 2022. Synchromesh: Reliable Code Generation from Pre-trained Language Models. In *International Conference on Learning Representations*. https://openreview.net/forum?id=KmtVD97J43e

[25] Houxing Ren, Zimu Lu, Weikang Shi, Haotian Hou, Yunqiao Yang, Ke Wang, Aojun Zhou, Junting Pan, Mingjie Zhan, and Hongsheng Li. 2025. Alignment with Fill-In-the-Middle for Enhancing Code Generation. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, Christos Christodoulopoulos, Tanmoy Chakraborty, Carolyn Rose, and Violet Peng (Eds.). Association for Computational Linguistics, Suzhou, China, 8304–8320. doi:10.18653/v1/2025.emnlp-main.419

[26] TOML. 2026. TOML: Tom's Obvious Minimal Language, v1.1.0. https://toml.io/en/. Accessed 2026-03-13.

[27] Shubham Ugare. 2024. Grammar Mask vs Grammar Strict Decoding. https://github.com/structuredllm/syncode/issues/193#issuecomment-2819678327. Syncode Repository on GitHub, Accessed 2026-03-12.

[28] Shubham Ugare, Tarun Suresh, Hangoo Kang, Sasa Misailovic, and Gagandeep Singh. 2025. SynCode: LLM Generation with Grammar Augmentation. *Transactions on Machine Learning Research* (2025). https://openreview.net/forum?id=HiUZtgAPoH

[29] Bailin Wang, Zi Wang, Xuezhi Wang, Yuan Cao, Rif A. Saurous, and Yoon Kim. 2023. Grammar Prompting for Domain-Specific Language Generation with Large Language Models. In *Advances in Neural Information Processing Systems*, A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine (Eds.), Vol. 36. Curran Associates, Inc., 65030–65055. https://proceedings.neurips.cc/paper_files/paper/2023/file/cd40d0d65bfebb894ccc9ea822b47fa8-Paper-Conference.pdf

[30] Shiqi Wang, Zheng Li, Haifeng Qian, Chenghao Yang, Zijian Wang, Mingyue Shang, Varun Kumar, Samson Tan, Baishakhi Ray, Parminder Bhatia, Ramesh Nallapati, Murali Krishna Ramanathan, Dan Roth, and Bing Xiang. 2023. ReCode: Robustness Evaluation of Code Generation Models. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Anna Rogers, Jordan Boyd-Graber, and Naoaki Okazaki (Eds.). Association for Computational Linguistics, Toronto, Canada, 13818–13843. doi:10.18653/v1/2023.acl-long.773

[31] J.B. Wells. 1999. Typability and type checking in System F are equivalent and undecidable. *Annals of Pure and Applied Logic* 98, 1 (1999), 111–156. doi:10.1016/S0168-0072(98)00047-5

[32] Brandon T. Willard and Rémi Louf. 2023. Efficient Guided Generation for Large Language Models. arXiv:2307.09702 [cs.CL] https://arxiv.org/abs/2307.09702