

Accelerating Constrained Decoding with Token Space Compression

Michael Sullivan and Alexander Koller

Department of Language Science and Technology
Saarland Informatics Campus
Saarland University, Saarbrücken, Germany
{msullivan, koller}@coli.uni-saarland.de

Abstract

To guarantee that an LLM’s outputs conform to a specified structure, context-free grammar (CFG) decoding engines force the selection of next tokens to produce strings that conform to a given CFG. Current CFG-constrained decoding engines are highly optimized, but still suffer from the inherent costs arising from their massive per-step search space—i.e. the entire token vocabulary. This results in intractably high overhead for more complex CFGs, which is precisely the situation where CFG engines are most useful. In this paper, we introduce CFGZIP, an offline technique for compressing the token search space, which massively reduces CFG engine overhead. In experiments, we report latency reduction of up to 75x during batched inference, cutting overhead down to $\sim 1.2\text{-}2\times$ on the hardest grammars: with CFGZIP, constrained decoding is now possible at scale for complex CFGs.

1 Introduction

Many modern LLM applications such as coding (Huynh and Lin, 2025; Jiang et al., 2026), tool calling (Qin et al., 2024; Sullivan et al., 2025; Patil et al., 2025), etc. necessitate structured generation: the generation of text that conforms to a specific format. Although LLMs excel at generating syntactically correct text in formats frequently found in their training data (Bogin et al., 2024; Mündler et al., 2025b), they struggle to conform to the specifications of novel formats such as domain-specific languages and unique programming languages (see Section 4). In such situations, it is therefore desirable to constrain the model’s outputs to guarantee syntactic correctness.

Structural constraints for LLM decoding are often defined by context-free grammars (CFGs; Chomsky, 1956; Angelov, 2009; Dong et al., 2025, etc.), which are strictly more expressive than regular expressions, and encompass use cases such

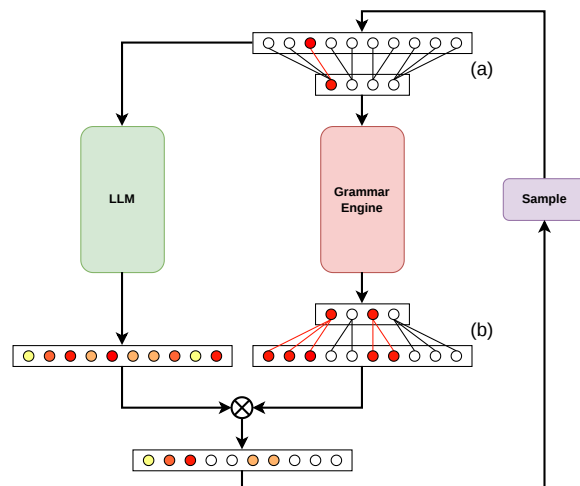


Figure 1: CFGZIP first pre-computes equivalence classes over the token vocabulary. At inference time, the grammar engine operates over the smaller vocabulary of equivalence class representatives: (a) a sampled token is mapped to the representative of its equivalence class; and (b) the grammar engine produces a mask over the equivalence class vocabulary, which is then mapped to a mask over the token vocabulary.

as the JSON file format, XML, many programming languages, chemical structure notation (e.g. Weininger, 1988), etc.

To guarantee that LLM outputs conform to a given CFG, constrained decoding engines are employed to modify the LLM’s next-token distribution, masking the probability of all tokens which would lead to a generated sequence that violates the format specified by the CFG.

Given the large overhead already associated with parsing a CFG— $\mathcal{O}(n^3)$ in the worst case (Earley, 1970)—the runtime cost associated with naively checking the validity of each token at each step can be prohibitively high. For this reason, SoTA grammar engines incorporate substantial online and offline optimization techniques. Despite these improvements, even the fastest grammar engines still

```

#include<stdio.h>
#include<vector>
#include<string>
using namespace std;
#include<algorithm>
#include<math.h>
#include<stdlib.h>
vector<string> filter_by_substring(vector<string> strings, string substring){
    vector<string> filtered_strings;
    for(auto str : strings){
        if(str.find(substring) != string::npos){
            filtered_strings.push_back(str);
        }
    }
    return filtered_strings;
}

#include<E.h>
#include<E>
#include<E>
using namespace Y;
#include<E>
#include<uy.h>
#include<E.h>
uy<E> Yuyuyuy(q<E> Y, Y Y){
    Y<E> Yuy;
    for(q Y : Y){
        if(q.G(quy) != Y::uy){
            Yuy.Guy(q);
        }
    }
    return Yuy;
}

```

Figure 2: C++ code generated by Llama-3.2-3B-Instruct using CFGZIP. The LLM operates over the actual tokens (left), while the grammar engine operates over the smaller vocabulary of equivalence class representatives (right): in this figure, each unique equivalence class is highlighted with a unique color for reference.

introduce massive latency—a 2-9x slowdown during unbatched inference—when used with more complex CFGs (e.g. programming languages; see Section 4).

To ameliorate this online overhead, we introduce CFGZIP (Section 3), a compression technique that vastly reduces the per-step runtime of the grammar engine. The key idea behind our approach is that many tokens are interchangeable with respect to the grammar, in that we need only check the validity of one to determine the validity of all tokens within a given equivalence class.

Our method first pre-computes these equivalence classes using a novel algorithm that constructs a decidable refinement of the syntactic congruence relation. CFGZIP then wraps the grammar engine at inference time, so that the engine operates over representatives of the equivalence classes, rather than the token vocabulary itself (see Figures 1-2).

In this way, CFGZIP synergizes with the grammar engine’s built-in optimizations, resulting in a 9-45x reduction in overhead over the current SoTA grammar engine. This yields **an up to 7.5x reduction in unbatched constrained generation time**—i.e. the combined runtime of the LLM and grammar engine—for more complex CFGs.

Furthermore, the relative latency reduction of our approach *increases* with batch size, making constrained generation feasible at scale for complex CFGs: **CFGZIP delivers speedups of up to ~75x during batched inference** with constrained decoding, **reducing overhead to merely ~1.2-2x on tasks where a SoTA grammar engine alone times out**.

CFGZIP is:

1. **Offline:** compression is performed once per tokenizer/CFG pair, and the compressed vocabulary is cached to disk. At inference time, CFGZIP itself incurs virtually no overhead.
2. **Lossless:** CFGZIP compression is provably lossless, meaning that the outputs of a grammar engine equipped with CFGZIP are guaranteed to be byte-identical to those of the original engine.

We release CFGZIP as a pip-installable Python package¹, and make all code used in the evaluation experiments in this paper publicly available on GitHub².

2 Background and Preliminaries

In the following, we adopt the Kleene star (Kleene, 1956) notation: given a set X , we define X^* as the set of all sequences of elements of X (i.e. strings over X), including the empty sequence/string ϵ . Given strings w, z , we denote their concatenation through juxtaposition: wz .

2.1 Context-Free Grammars (CFGs)

A CFG $G = (V, \Sigma, R, S)$ is defined by a set V of *non-terminal symbols*, an *alphabet* (set of characters) Σ , a *start symbol* $S \in V$, and a set R of productions (rewrite rules) of the form $A \rightarrow \alpha$,

¹<https://pypi.org/project/cfgzip>

²<https://github.com/coli-saar/cfgzip-experiments>

where $A \in V$ is a non-terminal and $\alpha \in (V \cup \Sigma)^*$ is a sequence of terminals and non-terminals.

Each CFG G defines a *language* (set of strings) $L(G) \subseteq \Sigma^*$, where $w \in L(G)$ if and only if $S \Rightarrow^* w$: i.e. there is a finite sequence of rules $S \rightarrow \alpha_0, A_1 \rightarrow \alpha_1, \dots, A_n \rightarrow \alpha_n \in R$ that derives w from S (a *parse* of w). We additionally define the set $Pr(L(G))$ of *prefixes* of $L(G)$: strings $w \in \Sigma^*$ such that there is some (possibly empty) string $z \in \Sigma^*$ such that the concatenation wz is in $L(G)$.

Greibach Normal Form (GNF; Greibach, 1965).

A CFG G is in GNF if all productions in G are of the form $S \rightarrow \epsilon$ (where S is the start symbol) or $A \rightarrow a \beta$, where $a \in \Sigma$ is a (non- ϵ) terminal symbol and $\beta \in (V - \{S\})^*$ is a (possibly empty) sequence of non-terminal, non-start symbols.

It is a well-known result that for every CFG G , there is an equivalent CFG G' in GNF—i.e. such that $L(G) = L(G')$. Each GNF CFG G defines a simple pushdown automaton (PDA³) that recognizes (parses) $L(G)$: this PDA consists of a stack σ of non-terminal symbols of G , and a transition function⁴ δ_G mapping pairs of terminals and non-terminals to sequences of non-terminals. For each terminal $a \in \Sigma$, non-terminal $A \in V$, and sequence of non-terminals $\beta \in V^*$, $\beta \in \delta_G(a, A)$ if and only if $A \rightarrow a \beta$ is a production in G .

Given a string w in Σ^* , the PDA parses w non-deterministically, with its stack initialized to $\sigma = (S)$, where S is the start symbol of G . The PDA scans w character-by-character: at the i^{th} character of w , the PDA non-deterministically (i) pops the top symbol A from σ ; and (ii) pushes β to σ for $\beta \in \delta_G(w_i, A)$. The PDA accepts w ($w \in L(G)$) if σ is empty when the end of w is reached—otherwise, w is not a member of $L(G)$.

2.2 XGrammar2

A CFG decoding engine ensures that an LLM generates a sequence that conforms to a given CFG G . Precisely, for each LLM output token y_i , the decoding engine ensures that the sequence $y_{:i+1} = y_i y_i$ is a valid prefix of $L(G)$: i.e. $y_{:i+1} \in Pr(L(G))$. This is achieved by logit masking: for all tokens $t^{(k)}$ in the token vocabulary T such that $y_{:i} t^{(k)} \notin Pr(L(G))$, the corresponding logit h_k is set to $-\infty$, so that the next-token probability of $t^{(k)}$ is 0.

³Every CFG defines a PDA: we defer an in-depth discussion of PDAs to prior work, and provide a semi-formal description here that is sufficient for the discussion at hand.

⁴We ignore states in the definition of the transition function, as PDAs derived from GNF CFGs have only one state.

We restrict the present discussion to an overview of XGrammar2 (Li et al., 2026)—a SoTA grammar engine—because this is the engine that we pair with CFGZIP in our experiments in Section 4: although CFGZIP is theoretically engine-agnostic, we were unable to find any other grammar engine that is capable of processing the more sophisticated CFGs used in our experiments (see the discussion in the Limitations section).

Underlying XGrammar2 is an Earley parser (Earley, 1970), which can run in time linear in the input string length n for a specific class of CFGs⁵, but has $\mathcal{O}(n^3)$ runtime in the worst-case scenario for arbitrary CFGs. Here, the Earley parser maintains a running parse of the generated sequence, and is used to check the compatibility of each token in the vocabulary to construct the next-token logit mask.

If done naively, constructing the logit mask requires verifying each token in the vocabulary at each decoding step, resulting in a worst-case runtime of $\mathcal{O}(|T| \cdot n^3)$. To mitigate this cost, XGrammar2 introduces several optimizations that reduce the number of tokens that need to be checked at each step: for example, pre-computing tokens that are guaranteed to be rejected in a given parser state.

The heavily optimized XGrammar2 can quickly compute token masks for simple CFGs such as JSON schema grammars, resulting in negligible overhead. However, as we demonstrate empirically in Section 4, more complex CFGs still present a challenge to XGrammar2: we record a $\sim 2\text{-}9\times$ slowdown over the base LLM during unbatched inference with CFGs such as C++ and a modified variant of Python.

3 CFGZIP

For a given CFG G and token vocabulary T , CFGZIP computes equivalence classes of tokens that are interchangeable with respect to G : at each step, the grammar engine needs only verify one token per equivalence class (see Figure 2).

In Section 3.1, we provide an overview of the CFGZIP pipeline using syntactic congruence (Myhill, 1957) as the underlying equivalence relation, for the sake of exposition; it follows as a consequence of Theorem 1 that equivalence-class compression using syntactic congruence is lossless and engine-agnostic.

However, computing the syntactic congruence relation is undecidable for arbitrary context-free

⁵Non-right recursive $LR(k)$ grammars.

languages (Bar-Hillel et al., 1961). In Section 3.2, we define a computable and tractable approximation of syntactic congruence. This relation is furthermore a refinement of syntactic congruence (Theorem 2), which guarantees that it maintains the lossless and engine-agnostic properties of the latter.

3.1 CFGZIP Pipeline

Syntactic Congruence. Intuitively, strings $t, u \in \Sigma^*$ are *syntactically congruent* ($t \equiv_{L(G)} u$) when they are interchangeable relative to $L(G)$: each instance of t in a string in $L(G)$ can be swapped with u to yield a string that is also in $L(G)$.

Definition 1 (Syntactic Congruence; Myhill, 1957). For $t, u \in \Sigma^*$, $t \equiv_{L(G)} u$ if and only if for all $w, z \in \Sigma^*$: $wtz \in L(G) \leftrightarrow wuz \in L(G)$

Given two length- n sequences of tokens $\tau = t_0 \dots t_{n-1}$, $\tau' = t'_0 \dots t'_{n-1}$ that are element-wise congruent—i.e. $t_i \equiv_{L(G)} t'_i$ for all $0 \leq i < n$ —it is straightforward to show that τ and τ' are themselves congruent: $\tau \equiv_{L(G)} \tau'$ (proof in Lemma 1 of Appendix B). It then follows from Definition 1 that τ is a valid prefix of $L(G)$ if and only if τ' is as well.

Theorem 1. For any $n \in \mathbb{N}$, alphabet Σ , language L over Σ , and length- n sequences of tokens $\tau = t_0 \dots t_{n-1}$, $\tau' = t'_0 \dots t'_{n-1}$ over Σ with $t_i \equiv_L t'_i$ (for all $0 \leq i < n$): $\tau \in \text{Pr}(L) \leftrightarrow \tau' \in \text{Pr}(L)$

Proof. Appendix B. $\square$

A critical corollary of Theorem 1 is that we can losslessly map between the token vocabulary T that is used by the LLM, and the set of equivalence class representatives $E \subseteq T$ used by the grammar engine under CFGZIP: replacing each token t_i in a sequence with another token that is congruent to t_i does not affect the validity of the sequence with respect to G (see Figure 2).

Pre-Computation. CFGZIP first groups the token vocabulary T into equivalence classes, using (an approximation of) syntactic congruence as the equivalence relation. We then assemble the set E of equivalence class representatives: for each class e , we choose as its representative the byte-wise shortest token in e , as the grammar engine’s underlying parser still operates on the character/byte level.

We then compute and cache a vector $c \in \mathbb{N}^{|T|}$ that maps token IDs to class IDs: for the i^{th} token

$t^{(i)} \in T$, c_i is the ID of that token’s equivalence class. We additionally compute and cache a vector $r \in \mathbb{N}^{|E|}$ that maps class IDs to representatives, such that the token r_k is the representative of the k^{th} equivalence class. For a token vocabulary size of 100,000 to 200,000 tokens, these two vectors occupy less than a megabyte in memory.

Inference. CFGZIP initializes the grammar engine with a vocabulary consisting of the set E of equivalence class representatives, in place of the entire token vocabulary T (see Figure 2).

At inference time, a token $t^{(i)} \in T$ is sampled from the next-token distribution, and appended to the current generation sequence. Rather than $t^{(i)}$, we pass to the grammar engine the class representative r_{c_i} to update its internal state (see Figure 1): by Theorem 1, this is equivalent to passing the original token $t^{(i)}$.

The LLM then produces next-token log probabilities $h \in \mathbb{R}^{|T|}$ over the token vocabulary T , and the grammar engine produces a binary mask $m \in \{-\infty, 0\}^{|E|}$ over the equivalence class vocabulary E . Rather than directly pointwise adding h to m (which is impossible), we instead employ a gather operation so that we add to h_i the mask entry whose index corresponds to the equivalence class of $t^{(i)}$: $h_i \leftarrow h_i + m_{c_i}$. This gather-then-mask operation can be readily parallelized on a GPU, and so incurs minimal runtime overhead.

3.2 Approximating Congruence

As syntactic congruence $t \equiv_{L(G)} u$ is undecidable for context-free languages, we use in practice a computable refinement $t \sim_{L(G)} u$ for equivalence-class-based compression in the CFGZIP pipeline of Section 3.1.

To that end, we first convert G to Greibach Normal Form (GNF), to obtain the simplified PDA discussed in Section 2.1. Conversion to GNF results in a worst-case $\mathcal{O}(n^4)$ increase in the number of grammar productions. However, as the relation $\equiv_{L(G)}$ is defined relative to the *language* $L(G)$ of G —and the approximation $\sim_{L(G)}$ is a refinement of $\equiv_{L(G)}$ —we need only convert G to GNF for the purpose of equivalence class computation: CFGZIP does not necessitate any modification to the grammar passed to the grammar engine itself.

3.2.1 Displacement Equivalence

For each token $t \in T$, we define the *displacement* $\Delta_G(t)$ as the set of all pairs $(\sigma^{(i)}, \sigma^{(o)}) \in V^* \times V^*$

Algorithm 1 Computation of $\Delta_G(t)$

Require: token $t \in \Sigma^*$, CFG $G = (V, \Sigma, R, S)$
stack adjacency relation $\mathcal{A}_S \subseteq V \times V$

- 1: **procedure** $\text{CD}(t, \sigma^{(i)}, \sigma^{(o)}, A_{\text{prev}})$
- 2: $Y \leftarrow \emptyset$
- 3: **if** $t = \epsilon$ **then**
- 4: $Y \leftarrow \{(\sigma^{(i)}, \sigma^{(o)})\}$
- 5: **else if** $\sigma^{(o)} = \epsilon$ **then** $\triangleright$ backtrack
- 6: **for** $A \rightarrow t_0 \beta \in R$ **do**
- 7: **if** $(A_{\text{prev}}, A) \in \mathcal{A}_S$ **or** $A_{\text{prev}} = \epsilon$ **then**
- 8: $Y \leftarrow Y \cup \text{CD}(t_1, \sigma^{(i)} A, \beta, A)$
- 9: **else**
- 10: **for** $\beta \in \delta_G(t_0, \sigma_0^{(o)})$ **do**
- 11: $Y \leftarrow Y \cup \text{CD}(t_1, \sigma^{(i)}, \beta \sigma_1^{(o)}, \sigma_0^{(o)})$
- 12: **return** Y
- 13:
- 14: $\Delta_G(t) \leftarrow \text{CD}(t, \epsilon, \epsilon, \epsilon)$

such that $\sigma^{(o)}$ is a stack that can be obtained by consuming the stack $\sigma^{(i)}$ to process t , using the PDA transition function δ_G (see Section 2.1).

For any tokens $t, u \in T$, $\Delta_G(t) = \Delta_G(u)$ implies $t \equiv_{L(G)} u$: if the sets of input stacks that accept t and u under δ_G are identical, the sets of strings to which t and u can be appended to form valid prefix strings are identical. Similarly, if t and u each map the same input stack $\sigma^{(i)}$ to the same output stacks $\sigma^{(o)}$ under δ_G , the sets of strings that can follow t and u in a given context are identical.

Theorem 2. For any alphabet Σ , CFG G over Σ , and $t, u \in \Sigma^*$: $\Delta_G(t) = \Delta_G(u) \rightarrow t \equiv_{L(G)} u$

Proof. Appendix C. $\square$

The conclusion of Theorem 2 states that the relation $\sim_{L(G)}$ defined by equivalence of displacements refines $\equiv_{L(G)}$: it can be used for lossless compression in the CFGZIP pipeline (Section 3.1).

3.2.2 Displacement Computation

For each token $t \in T$, we compute $\Delta_G(t)$ as in Algorithm 1 (see the example given in Figure 3).

Given an input token t , we initialize an input queue $\sigma^{(i)}$ and an output stack $\sigma^{(o)}$. We first non-deterministically select a production $A \rightarrow t_0 \beta$ that produces the first character (byte) t_0 of t , and push A and β to $\sigma^{(i)}$ and $\sigma^{(o)}$, respectively (Figure 3i). We then proceed one character at a time along t , non-deterministically popping the top symbol A from $\sigma^{(o)}$, and then pushing $\beta \in \delta_G(c, A)$ back to $\sigma^{(o)}$, where c denotes the current character of t (Algorithm 1, lines 9-11; Figures 3ii-3iii)—recall that $\beta \in \delta_G(c, A)$ corresponds to a production $A \rightarrow c \beta$ in G (see the discussion in Section 2.1).

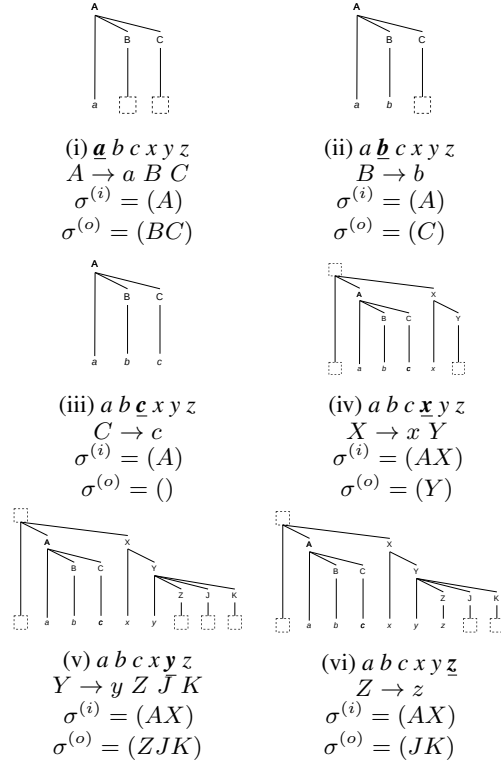


Figure 3: Illustration of Algorithm 1 applied to the token $t = abcxyz$. Each sub-figure (3i-3vi) illustrates the corresponding parse tree at that step: the caption indicates the current position/character t_0 (underlined/bolded), the rule $A \rightarrow t_0 \beta$ in G that is applied (i.e. transition $\beta \in \delta_G(t_0, A)$), and the resulting input/output stacks $\sigma^{(i)}/\sigma^{(o)}$ after applying the transition.

If the end of t is reached, we return $(\sigma^{(i)}, \sigma^{(o)})$, which indicates that t transitions the stack $\sigma^{(i)}$ to $\sigma^{(o)}$ under δ_G (Algorithm 1, lines 3-4). However, if $\sigma^{(o)}$ is emptied before the end of t , we perform a *stack backtrack*: non-deterministically select a production $A \rightarrow c \beta$, enqueue A to $\sigma^{(i)}$, and push β to $\sigma^{(o)}$ (Algorithm 1, lines 5-8; Figures 3iii-3iv).

As Algorithm 1 is non-deterministic, we found in practice that the naive stack backtrack was intractable: the blowup in possible input sequences caused by the greater number of non-terminals in larger CFGs resulted in out-of-memory errors.

We therefore restrict the stack backtrack operation using the pre-computed stack-adjacency relation $\mathcal{A}_S \subseteq V \times V$: given non-terminals $A, B \in V$, $(A, B) \in \mathcal{A}_S$ precisely when there is some string $w \in L(G)$ such that B is popped off of the stack immediately after A during some valid parse of w with δ_G (see Appendix D for a description of the stack-adjacency computation algorithm).

When computing $\Delta_G(t)$ in Algorithm 1, we track the stack symbol A_{prev} : this is either the sym-

Model	Vocab Size	JSON		XML		C++		Bython	
		Time	Classes	Time	Classes	Time	Classes	Time	Classes
Llama-3B	128256	0.28	255.4	0.54	1591	1.29	3095	0.63	2011
Qwen-4B	151936	0.33	257.8	0.64	1572	1.56	3057	0.71	1999
GPT-20B	201088	0.49	253.4	0.96	1517	1.80	2815	1.08	1783

Table 1: Wall-clock equivalence-class pre-computation time (seconds) and number of constructed equivalence classes per task for each model (vocabulary sizes given in number of tokens). JSON required a unique CFG for each of the 100 schemas: for this task, we report the mean across schema grammars of the pre-computation times and numbers of equivalence classes.

bol most recently popped from the output stack $\sigma^{(o)}$, or—if the last operation was a stack backtrack—the symbol most recently enqueued to $\sigma^{(i)}$. When a stack backtrack occurs, we may safely restrict the search space to those symbols A such that A is stack-adjacent to A_{prev} —i.e. $(A_{prev}, A) \in \mathcal{A}_S$ (Algorithm 1, line 7).

4 Experiments

To assess the effectiveness of CFGZIP, we evaluated Llama-3.2-3B-Instruct⁶ (Llama-3B; Grattafiori et al., 2024), Qwen3-4B-Instruct⁷ (Qwen-4B; Yang et al., 2025), and gpt-oss-20b⁸ (GPT-20B; Agarwal et al., 2025) on a series of structured-generation tasks (see Section 4.1).

As a constrained baseline, we adopted a SoTA grammar engine, XGrammar2, using task-specific CFGs (see Section 4.2); to evaluate our approach, we coupled CFGZIP with XGrammar2. We compared both methods to an unconstrained baseline (i.e. the vanilla model).

4.1 Tasks

For evaluation, we selected four structured-generation tasks. In addition to compute time, we evaluated the models on each benchmark across two binary dimensions: (i) syntactic correctness, i.e. syntactic well-formedness of the model’s output; and (ii) functional correctness, i.e. whether the output behaves as specified by the query.

JSON Schema: The json-mode-eval⁹ dataset consists of 100 queries along with schemas speci-

fying the required key strings and value data types of the target JSON objects. We measured syntactic correctness by executing the generated output and checking for syntax errors, and functional correctness by recursively checking key-value equivalence between the output and reference objects.

XML: We restricted the StructEval structured-generation benchmark (Yang et al., 2026) to the 200 instances requiring XML generation. We measured syntactic correctness by parsing the output using the Python XML module¹⁰, and functional correctness by verifying that all dot-path rules specified by the instance were satisfied by the generated output code.

C++: We restricted the HumanEval-X (Zheng et al., 2023) coding benchmark to the 164 C++ instances. We measured syntactic correctness by compiling the output, and calculated functional correctness by checking the output against all corresponding tests for each respective instance.

Bython: As the name suggests, Bython¹¹ is simply Python, but uses braces and semicolons to indicate scope and delimit statements (respectively), in place of meaningful whitespace. For our experiments, we translated all code in the 164 instances of the HumanEval-X Python split to Bython, and added a description of Bython to the system prompt. To evaluate syntactic correctness, we compiled the generated Bython code to Python. Functional correctness was measured as in the C++ task: namely, by checking if the output passed all tests for that instance.

While CFG engines provide guarantees of syntactically correct generation, we do not expect them

⁶<https://huggingface.co/meta-llama/Llama-3.2-3B-Instruct>

⁷<https://huggingface.co/Qwen/Qwen3-4B-Instruct-2507>

⁸<https://huggingface.co/openai/gpt-oss-20b>

⁹<https://huggingface.co/datasets/NousResearch/json-mode-eval>

¹⁰<https://docs.python.org/3/library/xml.html>

¹¹<https://github.com/mathialo/bython>

Model	Constraint	JSON		XML		C++		Bython	
		Syn.	Fun.	Syn.	Fun.	Syn.	Fun.	Syn.	Fun.
Llama-3B	Unconstrained	83.9	13.9	79.9	7.1	73.5	30.5	6.7	2.3
	Constrained	99.8	73.7	85.1	16.3	75.0	32.7	52.1	9.2
Qwen-4B	Unconstrained	100.0	76.5	83.9	61.1	88.9	70.6	2.2	1.0
	Constrained	96.9	74.7	83.2	61.4	90.4	72.7	43.3	18.9
GPT-20B	Unconstrained	100.0	80.2	82.1	60.1	48.4	41.2	13.0	10.8
	Constrained	100.0	81.2	81.3	59.0	49.6	40.2	72.0	46.9

Table 2: Constrained/unconstrained syntactic and functional correctness for each model and task. The best results for each model, task, and correctness type are indicated in bold.

Model	Constraint	BS=1				BS=8			BS=16		
		JSON	XML	C++	By.	XML	C++	By.	XML	C++	By.
Llama-3B	XGrammar2	3.60	4.18	2.14	12.55	1.51	1.27	44.51	0.77	0.85	—
	+CFGZIP	3.52	3.71	1.47	1.76	0.78	0.59	0.59	0.43	0.35	0.50
	Unconstrained	3.42	3.67	1.42	1.39	0.68	0.46	0.25	0.40	0.30	0.24
Qwen-4B	XGrammar2	1.75	6.22	5.14	19.50	1.29	2.62	97.07	0.97	4.21	—
	+CFGZIP	1.53	5.80	2.55	2.72	0.84	0.63	1.40	0.66	0.54	0.82
	Unconstrained	1.52	5.76	2.48	2.34	0.78	0.53	0.45	0.62	0.47	0.45
GPT-20B	XGrammar2	1.99	6.87	4.59	19.52	2.50	6.69	—	2.08	10.49	—
	+CFGZIP	1.68	6.23	2.41	3.52	1.76	1.29	1.89	1.60	1.56	2.11
	Unconstrained	1.67	6.20	2.34	2.67	1.67	0.97	1.50	1.55	1.05	1.82

Table 3: Mean generation time per instance (in seconds)—including the LLM overhead—for each batch size, model, constraint method, and task. A dash (“—”) indicates that the evaluation for that cell timed out after 24 hours.

to yield large improvements for tasks requiring common, “high-resource” context-free languages (CFLs) such as JSON, XML, and C++: they are ubiquitous in LLM training data, and so it is likely that SoTA models are already highly capable of generating strings that conform syntactically to these CFLs. On the other hand, a rare and/or domain-specific CFL such as Bython is far more likely to confound an LLM, and so it is precisely for these “low-resource” CFLs that we expect CFG-constrained generation to be most useful.

4.2 Setup

Hyperparameters and Evaluation. All models/methods were evaluated on all tasks across five seeds with a temperature of 0.2 and a maximum of 500 generated tokens. For the XML, C++, and Bython tasks, we evaluated at batch sizes of 1, 8, and 16. As the JSON task requires a separate grammar for each request, it is not conducive to batched inference with the current iteration of CFGZIP: we therefore only evaluated on this task using a batch

size of 1.

Inference was performed on a single NVIDIA H100 GPU, and the pre-computation and grammar engine operations were run 16-threaded on an Intel Xeon Gold 6430 CPU. Additional details of our evaluation setup are located in Appendix E.

Grammars. For the JSON task, we used XGrammar2’s built-in function for constructing CFGs from JSON schema specifications. We adapted an XML 1.1 specification¹² to the GBNF required by XGrammar2 for the XML task. For the C++ task, we used the C++ grammar from Mündler et al. (2025a). We constructed our Bython CFG from scratch: to the best of our knowledge, there is no existing Bython grammar specification.

Pre-Computation. Wall-clock times for the CFGZIP pre-computation procedure (see Section 3.1) and the number of equivalence classes for each tokenizer/CFG are reported in Table 1: CFGZIP

¹²https://www.liquid-technologies.com/Reference/Glossary/XML_EBNF1.1.html

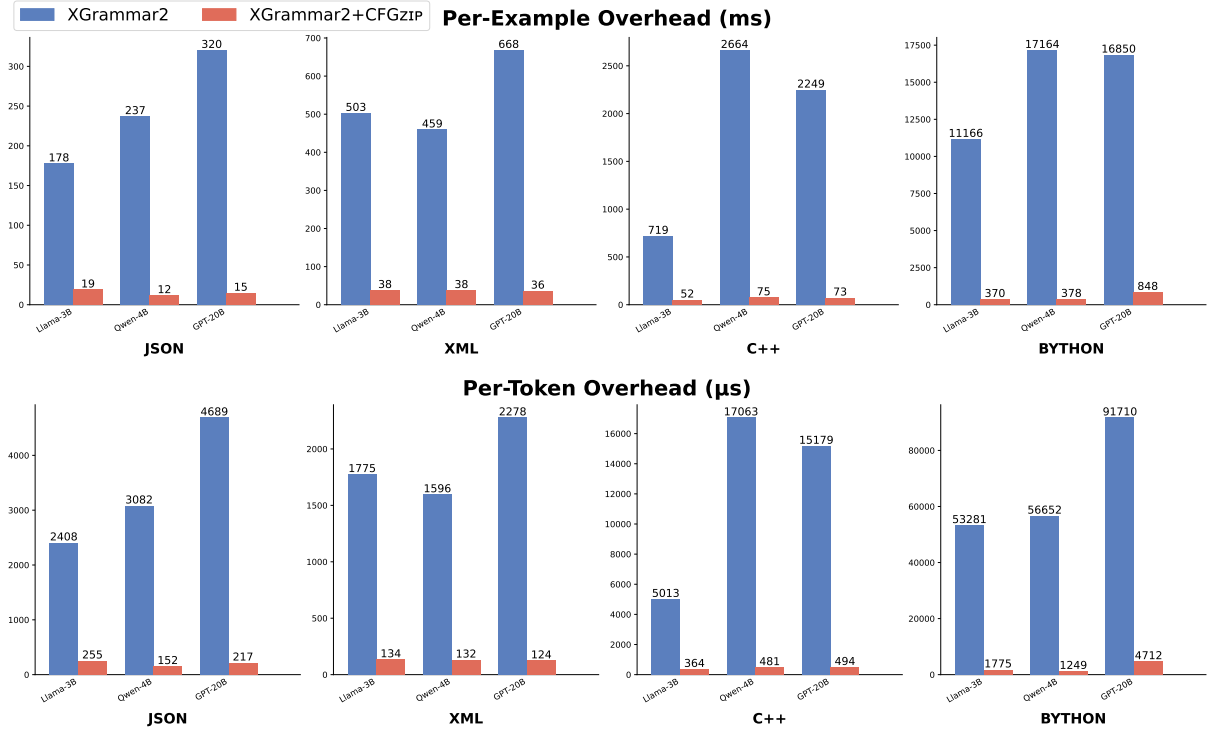


Figure 4: Comparison of the mean per-example (top; milliseconds) and per-token (bottom; microseconds) unbatched overhead of vanilla XGrammar2 (blue) compared to XGrammar2 equipped with CFGZIP (red).

achieves compression ratios ranging from $\sim 40:1$ (Llama-3B/C++) to $\sim 800:1$ (GPT-20B/JSON).

5 Results

Syntactic/functional correctness for each constrained/unconstrained model on each task are given in Table 2. We do not report separate scores for CFGZIP, as it produces byte-identical outputs to those of the unmodified XGrammar2.

Although constrained decoding improves syntactic correctness for most models on most tasks, in most cases it does not yield a 100% syntactic correctness score. This behavior is to be expected, as the grammar engine only ensures that the generated sequence is a valid *prefix* (see the discussion in Section 2.2): if the model’s generation is cut off by the 500-token limit, the resulting partial sequence typically does not pass the syntactic correctness checks.

Constrained Decoding Improves Performance with Smaller Models and Unfamiliar CFLs. The across-the-board improvement in correctness for Llama-3B demonstrates the utility of constrained decoding for smaller—and therefore cheaper—LLMs: rather than using a larger model to improve performance, simply constraining the

smaller LLM presents a viable, lower-cost alternative. On the other hand, the massive improvement in both syntactic and functional correctness for all three models on the Bython task shows that CFG-constrained generation is critical for using complex and unfamiliar context-free languages—even with 20-billion-parameter models such as GPT-20B.

SoTA Grammar Engines Have Unsustainable Overhead with Complex CFGs. XGrammar2 increases unbatched (i.e. batch size of 1) inference wall-clock time over the base (i.e. unconstrained) model by factors of $\sim 2\times$ and $\sim 9\times$ for C++ and Bython, respectively (see Table 3). This overhead is exacerbated at scale: during batched inference, the grammar engine must wait for the slowest parser state in the batch to compute its token mask before proceeding to the next generation step.

As a result, **XGrammar2 inference time increases to $\sim 3\text{--}7\times$ (C++) and $\sim 100\text{--}200\times$ (Bython)** that of the base model with batch size of 8. At a batch size of 16, **all three XGrammar2-constrained models timed out after 24 hours on the Bython task**. In production environments necessitating large batch sizes, constrained decoding with complex CFGs is infeasible—even when using SoTA grammar engines such as XGrammar2.

CFGZIP Makes Constrained Decoding with Complex CFGs Viable at Scale. By directly compressing the token space, CFGZIP nearly halves the total unbatched inference time of XGrammar2 on C++, and delivers batched constrained decoding speedups for XML and C++ by factors of $\sim 1.3\text{-}2\times$ and $\sim 2\text{-}8\times$, respectively. On the Bython task, our method cuts the unbatched overhead of XGrammar2 by up to 45x (see Figure 4): **unbatched constrained generation time decreases from $\sim 7\text{-}9\times$ that of the base model with vanilla XGrammar2 to $\sim 1.3\times$ with CFGZIP.**

At a batch size of 8, constrained generation time with CFGZIP is only $\sim 1.2\text{-}3\times$ that of the base model on Bython—including for GPT-20B, where XGrammar2 alone times out. For Llama-3B and Qwen-4B, this represents a **$\sim 75\times$ speedup over vanilla XGrammar2** at a batch size of 8. As batch size increases to 16 on Bython, **constrained generation time with CFGZIP remains at $\sim 1.2\text{-}2\times$ that of the base model, while vanilla XGrammar2 times out after 24 hours.** With CFGZIP, constrained decoding workloads that were previously impossible at scale can be performed with near-negligible overhead.

6 Related Work

There are several existing constrained decoding engines for structured generation. XGrammar2 achieves faster CFG compilation and mask generation than its predecessor, XGrammar (Dong et al., 2025), by swapping the PDA/stack-based parser of the latter for an Earley parser, and incorporating additional optimizations such as just-in-time grammar compilation. The llguidance engine¹³ similarly adopts an Earley parser that is highly optimized for use with JSON schema grammars.

The transformers-cfg engine (Geng et al., 2023) employs an incremental parser (Angelov, 2009), which is similar to the PDA parser used by the built-in CFG engine of the llama.cpp¹⁴ inference accelerator.

GreatGrammar (Park et al., 2025) also employs a PDA parser, and reduces overhead by pre-computing possible next tokens for certain stack prefixes. However, this engine is restricted to deterministic CFGs and does not ensure syntactic correctness: in some contexts, it fails to mask invalid tokens. SynCode (Ugare et al., 2025) pre-computes

possible next tokens for terminal finite-state machine (FSM) states, but is restricted to LR-parsable CFGs and can also fail to mask invalid tokens.

Outlines (Willard and Louf, 2023) and lm-format-enforcer¹⁵ use FSM- and tree-based parsers (respectively) for constrained decoding parameterized by regular expressions and JSON schemas, but are not compatible with CFGs in general.

CFGZIP is orthogonal and complementary to these methods: it is an algorithm for token space compression that can in theory be combined with any token-masking approach for CFG-constrained decoding. To the best of our knowledge, CFGZIP represents the first tool for compressing the token search space of a grammar engine through CFG-based equivalence classing.

7 Conclusion

We introduced CFGZIP, an LLM token vocabulary compression technique for inference-time speedup during CFG-constrained decoding. Our method achieves a $\sim 9\text{-}45\times$ reduction in unbatched overhead across model/token vocabulary sizes and CFGs. The speedup provided by CFGZIP scales with batch size, yielding up to $\sim 75\times$ faster batched constrained generation. As a result, tasks on which XGrammar2 alone times out can be accomplished using CFGZIP with only $\sim 1.2\text{-}2\times$ latency relative to the base model.

Our method unlocks previously infeasible performance from—and, therefore, applications of—CFG engines: although constrained decoding can more than quadruple functional correctness on a complex and unfamiliar context-free language such as Bython, even a SoTA grammar engine cannot handle the complexity of that language without introducing intractable overhead. CFGZIP shrinks this overhead to near-negligible levels—including at scale—allowing the practical use of CFG-constrained decoding in the settings where it is most effective.

Acknowledgments

We gratefully acknowledge the stimulating research environment of the GRK 2853/1 “Neuroexplicit Models of Language, Vision, and Action”, funded by the Deutsche Forschungsgemeinschaft (DFG; German Research Foundation) under project number 471607914.

¹³<https://github.com/guidance-ai/llguidance>

¹⁴<https://github.com/ggml-org/llama.cpp>

¹⁵<https://github.com/noamgat/lm-format-enforcer>

Limitations

For smaller, simpler CFGs (e.g. JSON schema grammars), the non-negligible offline pre-computation time for CFGZIP is longer than the reduction in overhead it provides (see Table 1 and Figure 4): this precludes the utility of CFGZIP for dynamic, one-off use cases with lightweight CFGs. Our method is intended instead for contexts requiring repeated use of the same grammar and/or larger, more complex CFG, where the inference-time speedup provided by CFGZIP ameliorates its pre-computation overhead.

Although CFGZIP is theoretically engine-agnostic, we only evaluated our method with XGrammar2 in this paper, due to the below-listed incompatibilities of the other grammar engines discussed in Section 6.

Outlines¹⁶ and lm-format-enforcer are only compatible with regular expressions and JSON schemas, and so cannot be used with the XML, C++, and Bython grammars of Section 4. Similarly, GreatGrammar and SynCode are not compatible with all CFGs—and do not guarantee syntactic correctness—so we did not evaluate our method with those engines.

The transformers-cfg package does not currently support the gpt-oss tokenizer, and so could not be used to evaluate that model. More importantly, transformers-cfg cannot parse CFGs with left-recursive productions, making it incompatible with our C++ and Bython grammars.

We found that llguidance incorrectly rejected grammatical C++ and Bython strings that were accepted by XGrammar2: we suspect this is due to the engine’s dynamic lexer prematurely rejecting valid paths under certain conditions. As it did not function correctly for half of our tasks, we did not evaluate CFGZIP with this engine.

The llama.cpp engine was incompatible with CFGZIP for practical reasons: CFGZIP is implemented in Python/Rust, while llama.cpp is implemented in C++ and does not expose Python bindings.

References

Sandhini Agarwal, Lama Ahmad, Jason Ai, Sam Altman, Andy Applebaum, Edwin Arbus, Rahul K. Arora, Yu Bai, Bowen Baker, Haiming Bao, Boaz

Barak, Ally Bennett, Tyler Bertao, Nivedita Brett, Eugene Brevdo, Greg Brockman, Sebastien Bubeck, Che Chang, Kai Chen, Mark Chen, Enoch Cheung, Aidan Clark, Dan Cook, Marat Dukhan, Casey Dvorak, Kevin Fives, Vlad Fomenko, Timur Garipov, Kristian Georgiev, Mia Glaese, Tarun Gogineni, Adam Goucher, Lukas Gross, Katia Gil Guzman, John Hallman, Jackie Hehir, Johannes Heidecke, Alec Helyar, Haitang Hu, Romain Huet, Jacob Huh, Saachi Jain, Zach Johnson, Chris Koch, Irina Kofman, Dominik Kundel, Jason Kwon, Volodymyr Kyrlyov, Elaine Ya Le, Guillaume Leclerc, James Park Lennon, Scott Lessans, Mario Lezcano-Casado, Yuanzhi Li, Zhuohan Li, Ji Lin, Jordan Liss, Lily (Xiaoxuan) Liu, Jiancheng Liu, Kevin Lu, Chris Lu, Zoran Martinovic, Lindsay McCallum, Josh McGrath, Scott McKinney, Aidan McLaughlin, Song Mei, Steve Mostovoy, Tong Mu, Gideon Myles, Alexander Neitz, Alex Nichol, Jakub Pachocki, Alex Paino, Dana Palmie, Ashley Pantuliano, Giambattista Parascandolo, Jongsoo Park, Leher Pathak, Carolina Paz, Ludovic Peran, Dmitry Pimenov, Michelle Pokrass, Elizabeth Proehl, Huida Qiu, Gaby Raila, Filippo Raso, Hongyu Ren, Kimmy Richardson, David Robinson, Bob Rotsted, Hadi Salman, Suvansh Sanjeev, Max Schwarzer, D. Sculley, Harshit Sikchi, Kendal Simon, Karan Singhal, Yang Song, Dane Stuckey, Zhiqing Sun, Philippe Tillet, Sam Toizer, Foivos Tsimpourlas, Nikhil Vyas, Eric Wallace, Xin Wang, Miles Wang, Olivia Watkins, Kevin Weil, Amy Wendling, Kevin Whinnery, Cedric Whitney, Hannah Wong, Lin Yang, Yu Yang, Michihiro Yasunaga, Kristen Ying, Wojciech Zaremba, Wenting Zhan, Cyril Zhang, Brian Zhang, Eddie Zhang, and Shengjia Zhao. 2025. gpt-oss-120b & gpt-oss-20b model card. *arXiv preprint arXiv:2508.10925*.

Krasimir Angelov. 2009. [Incremental parsing with parallel multiple context-free grammars](#). In *Proceedings of the 12th Conference of the European Chapter of the ACL (EACL 2009)*, pages 69–76, Athens, Greece. Association for Computational Linguistics.

Yehoshua Bar-Hillel, Micha Perles, and Eli Shamir. 1961. On formal properties of simple phrase structure grammars. *Sprachtypologie und Universalienforschung*, 14:143–172.

Ben Bogin, Shivanshu Gupta, Peter Clark, and Ashish Sabharwal. 2024. [Leveraging code to improve in-context learning for semantic parsing](#). In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 4971–5012, Mexico City, Mexico. Association for Computational Linguistics.

Noam Chomsky. 1956. Three models for the description of language. *IRE Transactions on Information Theory*, 2(3):113–124.

Yixin Dong, Charlie F Ruan, Yaxing Cai, Ziyi Xu, Yilong Zhao, Ruihang Lai, and Tianqi Chen. 2025. XGrammar: Flexible and efficient structured genera-

¹⁶Outlines implements a method to compile arbitrary CFGs, but calling this method currently raises a `NotImplementedError`.

- tion engine for large language models. *Proceedings of Machine Learning and Systems*, 7.
- Jay Earley. 1970. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102.
- 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*, pages 10932–10952, Singapore. Association for Computational Linguistics.
- Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. 2024. The Llama 3 herd of models. *arXiv preprint arXiv:2407.21783*.
- Sheila A Greibach. 1965. A new normal-form theorem for context-free phrase structure grammars. *Journal of the ACM (JACM)*, 12(1):42–52.
- Nam Huynh and Beiyu Lin. 2025. Large language models for code generation: A comprehensive survey of challenges, techniques, evaluation, and applications. *arXiv preprint arXiv:2503.01245*.
- Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. 2026. A survey on large language models for code generation. *ACM Transactions on Software Engineering and Methodology*, 35(2):1–72.
- Stephen Cole Kleene. 1956. *Representation of Events in Nerve Nets and Finite Automata*, volume 34. Princeton University Press.
- Linzhang Li, Yixin Dong, Guanjie Wang, Ziyi Xu, Alexander Jiang, and Tianqi Chen. 2026. XGrammar 2: Dynamic and efficient structured generation engine for agentic LLMs. *arXiv preprint arXiv:2601.04426*.
- Niels Mündler, Jasper Dekoninck, and Martin Vechev. 2025a. Constrained decoding of diffusion LLMs with context-free grammars. *arXiv preprint arXiv:2508.10111*.
- Niels Mündler, Jingxuan He, Hao Wang, Koushik Sen, Dawn Song, and Martin Vechev. 2025b. Type-constrained code generation with language models. *Proceedings of the ACM on Programming Languages*, 9(PLDI):601–626.
- John Myhill. 1957. Finite automata and the representation of events. *WADD Technical Report*, 57:112–137.
- Kanghee Park, Timothy Zhou, and Loris D’Antoni. 2025. Flexible and efficient grammar-constrained decoding. In *Forty-second International Conference on Machine Learning*.
- Shishir G Patil, Huanzhi Mao, Fanjia Yan, Charlie Cheng-Jie Ji, Vishnu Suresh, Ion Stoica, and Joseph E Gonzalez. 2025. The berkeley function calling leaderboard (BFCL): From tool use to agentic evaluation of large language models. In *Forty-second International Conference on Machine Learning*.
- Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, Sihan Zhao, Lauren Hong, Runchu Tian, Ruobing Xie, Jie Zhou, Mark Gerstein, Dahai Li, Zhiyuan Liu, and Maosong Sun. 2024. ToolLLM: Facilitating large language models to master 16000+ real-world APIs. In *The Twelfth International Conference on Learning Representations*.
- Michael Sullivan, Mareike Hartmann, and Alexander Koller. 2025. Procedural environment generation for tool-use agents. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 18555–18573.
- Shubham Ugare, Tarun Suresh, Hangoo Kang, Sasa Misailovic, and Gagandeep Singh. 2025. SynCode: LLM generation with grammar augmentation. *Transactions on Machine Learning Research*.
- David Weininger. 1988. SMILES, a chemical language and information system. *Journal of Chemical Information and Computer Sciences*, 28(1):31–36.
- Brandon T Willard and Rémi Louf. 2023. Efficient guided generation for large language models. *arXiv preprint arXiv:2307.09702*.
- An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chuji Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jing Zhou, Jingren Zhou, Junyang Lin, Kai Dang, Keqin Bao, Kexin Yang, Le Yu, Lianghao Deng, Mei Li, Mingfeng Xue, Mingze Li, Pei Zhang, Peng Wang, Qin Zhu, Rui Men, Ruize Gao, Shixuan Liu, Shuang Luo, Tianhao Li, Tianyi Tang, Wenbiao Yin, Xingzhang Ren, Xinyu Wang, Xinyu Zhang, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yinger Zhang, Yu Wan, Yuqiong Liu, Zekun Wang, Zeyu Cui, Zhenru Zhang, Zhipeng Zhou, and Zihan Qiu. 2025. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*.
- Jialin Yang, Dongfu Jiang, Tony He, Sherman Siu, Yuxuan Zhang, Disen Liao, Zhuofeng Li, Huaye Zeng, Yiming Jia, Haozhe Wang, Benjamin Schneider, Chi Ruan, Wentao Ma, Zhiheng Lyu, Yifei Wang, Yi Lu, Quy Duc Do, Ziyang Jiang, Ping Nie, and Wenhui Chen. 2026. StructEval: Benchmarking LLMs’ capabilities to generate structural outputs. *Transactions on Machine Learning Research*.
- Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Lei Shen, Zihan Wang, Andi Wang, Yang Li, Teng Su, Zhilin Yang, and Jie Tang. 2023. CodeGeeX: A pre-trained model for code generation with multilingual benchmarking on HumanEval-X. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 5673–5684.

A AI Use

The initial CFGZIP code was implemented in Python by hand. We used AI coding assistants to help port the research repository to a pip-installable module, including converting the implementation of the displacement-construction algorithm (see Section 3.2.2) component from pure Python to a Rust backend to decrease pre-computation overhead.

B Proof of Theorem 1

First, given a string $w \in \Sigma^*$, define the left quotient $w \backslash L$ of L by w as the set of all $z \in \Sigma^*$ such that $wz \in L$: $w \backslash L = \{z \in \Sigma^* \mid wz \in L\}$. It is clear that the definition of syntactic congruence $t \equiv_L u$ in Definition 1 is equivalent to the condition that for all $w \in \Sigma^*$: $wt \backslash L = wu \backslash L$.

$$t \equiv_L u \leftrightarrow \forall w \in \Sigma^*: wt \backslash L = wu \backslash L \quad (1)$$

We now prove that the quotient map $\Sigma^* \rightarrow \Sigma^* / \equiv_L$ is a monoid homomorphism.

Lemma 1. *For any $w, w', z, z' \in \Sigma^*$, $w \equiv_L w'$ and $z \equiv_L z'$ implies that $wz \equiv_L w'z'$.*

Proof. By assumption and Definition 1, we have the following equivalences for all $x, y \in \Sigma^*$:

$$(i) \quad xwy \in L \leftrightarrow xw'y \in L$$

$$(ii) \quad xzy \in L \leftrightarrow xz'y \in L$$

By (i), we have $xwzy \in L \leftrightarrow xw'zy \in L$, i.e. $wz \equiv_L w'z$. By (ii), we have $xw'zy \in L \leftrightarrow xw'z'y \in L$, i.e. $w'z \equiv_L w'z'$. Chaining equivalences, we have: $wz \equiv_L w'z \equiv_L w'z'$. $\square$

By Lemma 1, $\tau \equiv_L \tau'$, and so by Equation 1, we have $\tau \backslash L = \tau' \backslash L$.

For any $a \in \Sigma^*$, the condition $a \in Pr(L)$ is equivalent to the condition that $a \backslash L \neq \emptyset$. Therefore, $\tau \backslash L = \tau' \backslash L$ implies that $\tau \in Pr(L) \leftrightarrow \tau' \in Pr(L)$.

This completes the proof of Theorem 1.

C Proof of Theorem 2

Define $\pi: \Sigma^* \times V^* \rightarrow \mathcal{P}(V^*)$ to be the function that takes a sequence of terminals t (token) and a sequence of non-terminals α (stack) as input, and returns the set of stacks obtained by applying the transition function δ_G to t given α (Equation 2).

$$\pi(t, \alpha) = \begin{cases} \{\alpha\} & \text{if } t = \epsilon \\ \emptyset & \text{if } \alpha = \epsilon \\ \bigcup_{\beta \in \delta_G(t_1, \alpha_1)} \pi(t_1, \beta) & \text{otw.} \end{cases} \quad (2)$$

Now define $\pi_S: \Sigma^* \rightarrow \mathcal{P}(V^*)$ as the restriction of π to the start symbol S : $\pi_S(t) = \pi(t, S)$.

We now prove in Lemma 2 that equivalence of π_S -sets implies equivalence of left quotients (see Equation 1).

Lemma 2. *For all $w, z \in \Sigma^*$: $\pi_S(w) = \pi_S(z)$ implies that $w \backslash L(G) = z \backslash L(G)$.*

Proof. For all $x \in \Sigma^*$, $\pi_S(x)$ is by definition the result of non-deterministically applying δ_G to x , starting from S . By definition, $x \in L(G)$ if and only if δ_G halts on the empty stack. Therefore, we have $x \in L(G) \leftrightarrow \epsilon \in \pi_S(x)$.

Applying the definition of the left quotient and the above equivalence, we have $x \backslash L(G) = \{y \mid xy \in L(G)\} = \{y \mid \epsilon \in \pi_S(xy)\}$. As $\pi_S(xy) = \bigcup_{\beta \in \pi_S(x)} \pi(y, \beta)$ by definition (Equation 2), we have $x \backslash L(G) = \{y \mid \epsilon \in \pi^*(x, y)\}$, where $\pi^*(x, y) = \bigcup_{\beta \in \pi_S(x)} \pi(y, \beta)$.

By assumption, $\pi_S(w) = \pi_S(z)$, which implies that $\pi^*(w, -) = \pi^*(z, -)$ pointwise. By the above equivalence, this in turn implies that $w \backslash L(G) = z \backslash L(G)$. $\square$

We now define the *pre-displacement* $D_G(t)$ of a token t : $D_G(t) = \{(\alpha, \beta) \mid \alpha \in V^* \wedge \beta \in \pi(t, \alpha)\}$. The relation defined by equivalence of pre-displacements refines syntactic congruence (Lemma 3).

Lemma 3. *For all $t, u \in \Sigma^*$: $D_G(t) = D_G(u)$ implies that $t \equiv_{L(G)} u$.*

Proof. The assumption that $D_G(t) = D_G(u)$ implies by definition of $D_G(-)$ that $\pi(t, \gamma) = \pi(u, \gamma)$ for each $w \in \Sigma^*$ and each $\gamma \in \pi_S(w)$. This in turn implies that $\pi_S(wt) = \pi_S(wu)$, which by Lemma 2 implies that $wt \backslash L(G) = wu \backslash L(G)$.

We therefore have $wt \backslash L(G) = wu \backslash L(G)$ for all $w \in \Sigma^*$: this is equivalent to the definition of $t \equiv_{L(G)} u$ (see Equation 1). $\square$

Given a token t and a pair $(\alpha, \beta) \in D_G(t)$, at most $\text{len}(t)$ symbols are consumed from α when parsing t . This is to say that there is some $1 \leq n \leq \text{len}(t)$ such that only the first n symbols of α are

consumed, and the remaining subsequence γ of α is shared with β : $\alpha = \hat{\alpha}\gamma$ where $\hat{\alpha} = A_1 \dots A_n \in V^*$, and $\beta = \hat{\beta}\gamma$ where $\hat{\beta} \in V^*$.

$D_G(t)$ is thus *precisely* the set of pairs $(\hat{\alpha}\gamma, \hat{\beta}\gamma)$ such that $\hat{\alpha}$ is entirely consumed when parsing t , $\hat{\beta} \in \pi(t, \hat{\alpha})$, and $\gamma \in V^*$.

Now, define $V^{(0)} = V \times \{0\}$, $V^{(1)} = V \times \{1\}$, $f_i: V^* \rightarrow (V^{(i)})^*$ as $f_i(A_0 \dots A_n) = (A_0, i) \dots (A_n, i)$, and $g: (V^{(0)} \cup V^{(1)})^* \rightarrow V^*$ as $g((A_0, i_0) \dots (A_n, i_n)) = A_0 \dots A_n$. Define $\pi': \Sigma^* \times (V^{(0)} \cup V^{(1)})^* \rightarrow \mathcal{P}((V^{(0)} \cup V^{(1)})^*)$ analogously to π in Equation 2, with the exception that π' takes sequences of symbols of the form (A, i) (for $i \in \{0, 1\}$), and pushes to the stack sequences of symbols of the form $(B, 1)$ —the 0/1 are used to track whether the symbol was present in the initial input stack.

$$\pi'(t, \alpha) = \begin{cases} \{\alpha\} & \text{if } t = \epsilon \\ \emptyset & \text{if } \alpha = \epsilon \\ \bigcup_{\beta \in \delta_G(t_0, g(\alpha_0))} \pi'(t_1, f_1(\beta)\alpha_{1:}) & \text{otw.} \end{cases} \quad (3)$$

Define $D'_G(t) = \{(g(\alpha), g(\beta)) \mid \alpha \in (V^{(0)})^* \wedge \beta \in \pi'(t, \alpha) \cap (V^{(1)})^*\}$. This is exactly $D_G(t)$ restricted to those pairs (α, β) such that α is entirely consumed when parsing t to derive β .

By the reverse of the above argument, we have that $D_G(t) = \{(\alpha\gamma, \beta\gamma) \mid (\alpha, \beta) \in D'_G(t) \wedge \gamma \in V^*\}$. Therefore, we have $D'_G(t) = D'_G(u) \rightarrow D_G(t) = D_G(u)$ for all $t, u \in \Sigma^*$. Chained with Lemma 3, we now have $D'_G(t) = D'_G(u) \rightarrow t \equiv_{L(G)} u$.

As defined in Algorithm 1, the displacement of a token t is precisely $\Delta_G(t) = \{(\alpha, \beta) \in D'_G(t) \mid \forall 0 < i < \text{len}(\alpha) : (\alpha_{i-1}, \alpha_i) \in \mathcal{A}_S\}$, where $\mathcal{A}_S$ is the stack adjacency relation (see Appendix D).

This only eliminates from $D'_G(t)$ those pairs (α, β) such that α contains a pair (A_i, A_{i+1}) of non-terminals such that A_{i+1} cannot be popped off the stack immediately after A_i during any parse starting from S . It therefore still holds that $\Delta_G(t) = \Delta_G(u) \rightarrow \pi_S(wt) = \pi_S(wu)$ for all $w \in \Sigma^*$, which by the argument in Lemma 3 implies that $t \equiv_{L(G)} u$.

This completes the proof of Theorem 2.

D Stack Adjacency

Note that we require only a *subset* of the stack adjacency relation $\mathcal{A}_S \subseteq V \times V$: in practice, we

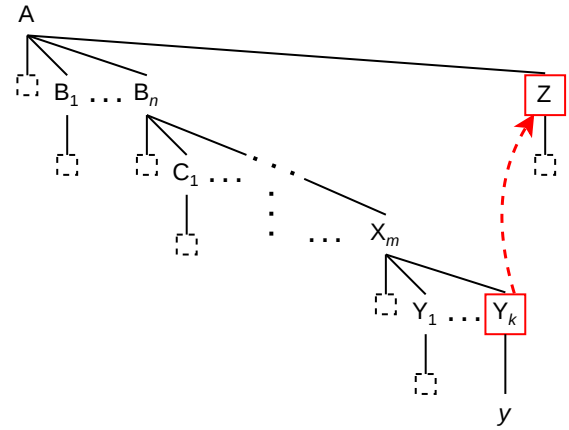


Figure 5: Illustration of the stack adjacency sub-relation $\mathcal{A}_S$. Here, Y_k is stack-adjacent to Z —i.e. $(Y_k, Z) \in \mathcal{A}_S$: there is some possible parse in which Y_k is popped off the stack immediately before Z .

may limit $\mathcal{A}_S$ to those pairs (X, Y) such that a stack backtrack can occur in Algorithm 1 after X , as we are computing this relation solely for the purpose of restricting the stack backtrack operation. By definition, a stack backtrack only occurs when the stack $\sigma^{(o)}$ is empty: i.e. X is the only symbol on the stack when it is popped, and no symbols are pushed back onto the stack—this corresponds to the application of a rule of the form $X \rightarrow c$ in G .

Pre-computation. We pre-compute $\mathcal{A}_S$ only once before constructing $\Delta_G(t)$ for each token t . We first compute the directed graph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V} = V$ is the set of non-terminals of the CFG G , and there is an edge $A \rightarrow B \in \mathcal{E}$ for each production $A \rightarrow a \beta B$ in G ($\beta \in V^*$)—i.e. where B is the right-most non-terminal of a rule headed by A .

As illustrated in Figure 5, we calculate $\mathcal{A}_S$ from $\mathcal{H}$ as follows. For each non-terminal Z and each production $A \rightarrow a B_1 \dots B_n Z$ $\beta \in R$ ($\beta \in V^*$), we perform a breadth-first search on $\mathcal{H}$ starting from B_n to find all Y_k such that there is a path from B_n to Y_k in $\mathcal{H}$ and there is a unary production $Y_k \rightarrow y$ in G : each such Y_k is popped from the stack immediately before Z during some valid parse—and can result in a stack backtrack in Algorithm 1—and so we add (Y_k, Z) to $\mathcal{A}_S$.

E Experimental Setup: Additional Details

For the Bython task of Section 4, several instances took multiple hours to generate with XGrammar2: for this reason, we placed a 15-minute generation

	Llama-3B	Qwen-4B	GPT-20B
Unconstrained	0.0	0.0	0.0
XGrammar2	5.1	5.2	4.9
+CFGZIP	0.0	0.0	0.2

Table 4: Percentage of Bython task instances across all five seeds that timed out after 15 minutes, by model and constraint type.

timeout window on all models for this task. This did not impact the correctness scores in Table 2: instances for which the XGrammar2-constrained model timed out were invariably pathological examples in which the model was caught in a loop of generating incoherent text.

Roughly 5% of all examples timed out with XGrammar2 across all five seeds (see Table 4), versus only 0.2% of the examples for the CFGZIP-equipped GPT model—and 0.0% for all other CFGZIP-equipped models—corresponding to two examples out of 164 across five seeds (i.e. 2/820).

We treated these timed-out examples as outliers and did *not* include them when calculating the overhead times reported in Figure 4 or the unbatched inference times in Table 3, and we entirely removed those instances from the dataset for batched evaluation: the speedup afforded by CFGZIP is not attributable to those pathological examples.