# INSTAGRAMMAR: COMPILING STACK SIGNATURES FOR GRAMMAR-CONSTRAINED DECODING

**Anonymous authors**
Paper under double-blind review

## ABSTRACT

Grammar-constrained decoding (GCD) enforces syntactic constraints by masking invalid tokens at each generation step. Context-free grammars complicate this operation because token validity can depend on an unbounded parser stack. We present INSTAGRAMMAR, which compiles stack-dependent compatibility for a fixed grammar and vocabulary into finite response signatures. A first-pop decomposition makes these signatures compositional: a push follows a compiled transition and a pop restores the signature saved in its frame, so online masking selects and combines precomputed rows instead of simulating candidate tokens against the stack. On MaskBench, INSTAGRAMMAR's mean per-token matcher cost is 7.4 $\mu$s, falling to 3.6 $\mu$s when a grammar is reused, with a P99 of 10 $\mu$s under reuse against 118 $\mu$s for llguidance and 604 $\mu$s for XGrammar. It is faster than llguidance on 99.7% of schemas under reuse.These gains cost more preprocessing, a median of 96 ms per schema, recovered at mean per-token cost after about 1,800 (XGrammar) and 7,500 (llguidance) tokens of grammar reuse. In vLLM serving on schemas where existing engines incur heavy online cost, INSTAGRAMMAR delivers 5.4–5.9 times XGrammar's throughput and 1.6–1.9 times llguidance's.

## 1 INTRODUCTION

Large language models are increasingly deployed as components in larger systems, where their outputs must conform to formal interfaces: JSON schemas for tool calls, SQL for database queries (Poesia et al., 2022), and formulas for verification pipelines. *Grammar-constrained decoding* (GCD) (Willard & Louf, 2023; Geng et al., 2023) enforces syntactic constraints at the decoding step, computing a token-validity mask from the current grammar state and applying it to the model's logits before sampling. GCD is available in serving engines such as vLLM (Kwon et al., 2023). When a grammar makes many tokens depend on the parser stack, mask computation can outlast the model's decode step and throttle serving.

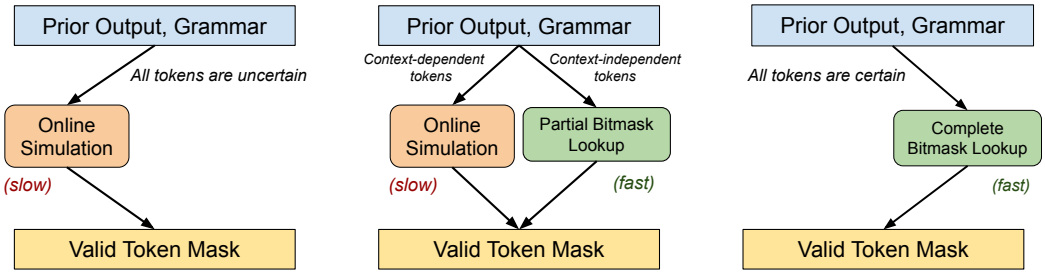
The difficulty arises from nested or recursive structure. A pushdown automaton (PDA) maintains an unbounded stack alongside its finite control state (Hopcroft & Ullman, 1979). A token's validity can depend on symbols below the current stack top, and one vocabulary token can close several nested structures. For regular languages, a finite automaton state summarizes the relevant prefix, permitting a precomputed state-to-mask index (Willard & Louf, 2023).

A direct approach simulates each vocabulary token against the live PDA (Figure 1a). XGrammar reduces this work by splitting the vocabulary (Dong et al., 2025): a *context-independent* token, whose validity is determined by the current parser state, is tabulated offline, while a *context-dependent* token, whose validity depends on stack contents below the current position, is resolved at decoding time by simulating it against the live configuration (Figure 1b). The online part grows with the number of context-dependent tokens, and therefore with grammar and tokenizer complexity. Other systems use lexical mask stores or incremental parsing (Ugare et al., 2025; guidance-ai, 2026). The infinite stack space does not itself preclude finite precomputation. Classical pushdown reachability provides regular representations of stack sets (Bouajjani et al., 1997), and PSC compiles a finite parser-stack classifier (Li et al., 2026b). The question is how to compute and maintain the information needed by a fixed vocabulary without repeatedly traversing the complete stack.

INSTAGRAMMAR answers this question by grouping stacks according to how they respond to the vocabulary. Two stacks that accept the same token suffixes from every parser state produce the



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.




(a) Direct simulation (b) Partial precomputation (c) Signature-based compilation

Figure 1: Mask-resolution strategies. *(a)*: every vocabulary token is simulated against the live PDA configuration. *(b)*: a partial mask is precomputed for context-independent tokens; context-dependent tokens are resolved online. *(c)*: masks are compiled for every (state, stack-signature) pair, so the mask is a table lookup. All three still advance the parser on the selected token.

same mask, so they can share one precomputed mask row; we call these equivalence classes *stack signatures*. A first-pop decomposition shows that a pushed stack's signature depends only on the pushed symbol and the signature of the stack below it. A push therefore advances the signature through a compiled transition, and a pop restores the signature saved in the removed frame. During decoding, the current state and signature select the mask row directly, without scanning the stack or simulating tokens against it. Our contributions are:

- **Stack signatures.** We define stack signatures and prove a first-pop decomposition that lets a push update the signature in one step.
- **Compiler and runtime.** We give a three-phase compiler (base responses, first-pop influences, signature discovery) and a runtime in which a pop restores the signature saved in its frame.
- **Evaluation.** We compare compilation and online mask costs across six engines on JSONSchemaBench-derived workloads, report coverage and latency tails, measure vLLM serving throughput on schemas with heavy online cost, and analyze the reuse needed to amortize preprocessing.

## 2 Background and Related Work

**Formal grammars and pushdown automata.** A grammar $G$ over a byte alphabet $\Sigma$ specifies a language $\mathcal{L}(G) \subseteq \Sigma^*$. Context-free grammars describe the syntactic fragments considered here, including nested JSON and expressions.

A PDA $\mathcal{P} = (Q, \Sigma, \Gamma, \delta, q_0, \gamma_0, F)$ combines finite control with a stack (Hopcroft & Ullman, 1979). A configuration $(q, \sigma)$ pairs the control state and stack; $\gamma\sigma$ has top symbol $\gamma$. We normalize transitions to consume a byte or epsilon and perform at most one push or one matching pop. A dedicated accepting state is entered only once the document is complete. For the running example, the bottom marker must be removed with no pending parenthesis frames. Thus an accepting state always means a complete document: reaching $q_1$ alone does not allow EOS.

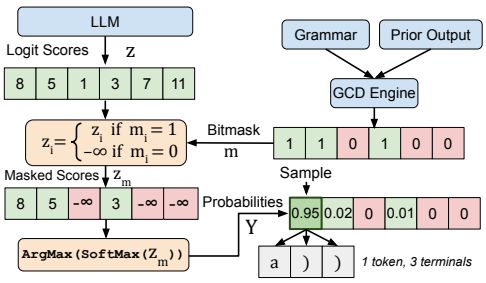
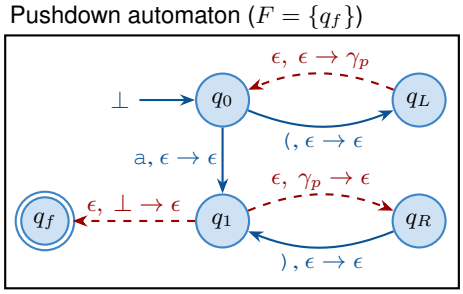



Figure 2: Grammar-constrained decoding builds a mask $\mathbf{m}$ over the vocabulary and sets masked logits to $-\infty$.

**Grammar-constrained decoding.** Let $L(q, \sigma)$ contain the byte strings accepted from $(q, \sigma)$, allowing epsilon transitions. For ordinary token byte strings $\mathcal{T} = \{w_1, \ldots, w_V\} \subseteq \Sigma^+$, define

$$\text{Valid}((q, \sigma), w_i) = \mathbf{1}[\exists z \in \Sigma^* : w_i z \in L(q, \sigma)], \quad \text{EOS}(q, \sigma) = \mathbf{1}[\epsilon \in L(q, \sigma)]. \tag{1}$$




| Prior input | Input | Transitions | Stack $\sigma$ |
|---|---|---|---|
| | | $q_0$ | $\perp$ |
| | ( | $q_0 \to q_L \to q_0$ | $\gamma_p\perp$ |
| ( | ( | $q_0 \to q_L \to q_0$ | $\gamma_p\gamma_p\perp$ |
| (( | a | $q_0 \to q_1$ | $\gamma_p\gamma_p\perp$ |
| ((a | ) | $q_1 \to q_R \to q_1$ | $\gamma_p\perp$ |
| ((a) | ) | $q_1 \to q_R \to q_1$ | $\perp$ |
| ((a)) | EOS | $q_1 \to q_f$ | $\epsilon$ |

Figure 3: PDA for $E \to a \mid (E)$ and its configuration sequence on `((a))`. Solid edges read input, dashed edges are epsilon moves; only $q_f$ accepts, so `(a` is a valid prefix but cannot terminate (Appendix G).

For the language $\{ab\}$, token $a$ is initially valid although EOS is not. We use byte continuations, assuming the vocabulary can spell required continuations, for example through byte fallback. Otherwise the continuation language must be restricted to realizable token sequences.

With active configurations $\mathcal{A}$, token validity is existential over $\mathcal{A}$. We set invalid logits to $-\infty$ before sampling (Figure 2); an empty mask, including EOS, reports constraint failure. A vocabulary token can cross several grammar terminals and trigger several pushes or pops. Its compatibility can therefore depend on more than the top stack symbol. A token is context-independent when it only pushes frames of its own or leaves the stack unchanged, so its validity is determined by the parser state; it is context-dependent when it must pop frames that were present before it began, so its validity depends on what lies below. In the running example of Figure 3, `(` only pushes, and `a` and `(a` never close a pending parenthesis, so these three tokens are context-independent. The tokens `a)` and `))` close one and two enclosing parentheses respectively; each is valid only if enough frames exist below, so both are context-dependent.

**Related work.** Outlines constructs an automaton index for regular constraints (Willard & Louf, 2023); Koo et al. (2024) develop automata-based constraints for language-model decoding. SynCode combines DFA mask stores with incremental parsing (Ugare et al., 2025). GreatGramma precomputes lexical token-spanning information (Park et al., 2025). Formatron supports KBNF/EBNF-style grammars and describes Earley-based dynamic pruning (Sun et al., 2025); llguidance uses an Earley parser with a regex lexer (guidance-ai, 2026). XGrammar and XGrammar-2 combine compiled grammar structure with online parsing and caching (Dong et al., 2025; Li et al., 2026a). DOMINO aligns constraints with subword tokens (Beurer-Kellner et al., 2024); Pre$^3$ precomputes prefix-conditioned edges of a deterministic PDA for LR(1) grammars (Chen et al., 2025); and Alpay & Senturk (2026) show that equivalent grammars share masks but can differ in online cost. Because these systems precompute different objects, their compile times are only comparable on matched grammars. PSC also precomputes stack-dependent compatibility. Its mask-selection algorithm traverses a lexer-state/stack representation before selecting a row (Li et al., 2026b). InstaGrammar maintains its signature during pushes and saves predecessor signatures for pops, avoiding a fresh complete-stack traversal for row selection. Appendix H summarizes the systems and their assumptions.

## 3 Stack Signatures

In this section, we introduce stack signatures, which allow stack-dependent token masks to be tabulated offline and maintained during parsing.

### 3.1 Motivation

Grammar-constrained decoding must determine which vocabulary tokens preserve a possible completion from the current configuration $(q, \sigma)$. Although stacks can grow without bound, the vocabulary $\mathcal{T}$ is fixed. The mask therefore depends on a finite family of Boolean queries. Two stacks giving the same answers produce the same mask at every control state. The remaining challenge is to make this

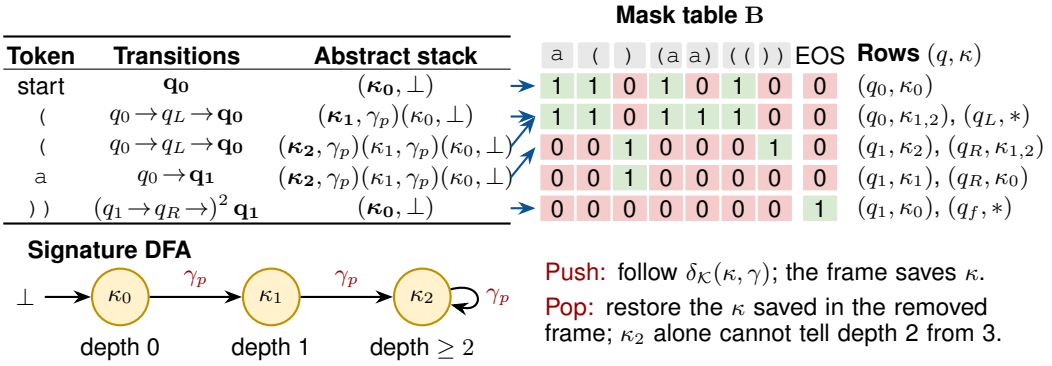



Figure 4: Signature-based masking for the parentheses example. Top left: trace of `((a))`; each frame stores its symbol and the signature up to it (bold: current state and signature). Right: distinct rows of $\mathbf{B}$, with EOS, and the $(q, \kappa)$ pairs sharing each row; $\kappa_0, \kappa_1, \kappa_2$ are the classes of $\gamma_p^d \bot$ for $d = 0, 1, \geq 2$. Bottom: the signature DFA.

equivalence *compositional*: a push should update the signature without traversing the stack, and a pop should recover its predecessor.

Consider again `expr ::= "a" | "(" expr ")"` and a byte-fallback vocabulary augmented with `(a`, `a)`, `((`, and `))`. Complete acceptance requires that every pending parenthesis has been closed. Figure 4 shows the mask rows for seven displayed tokens and EOS. Among stacks $\gamma_p^d \bot$, depths zero, one, and at least two give three distinct response classes. A push maps these classes as $0 \mapsto 1 \mapsto \geq 2 \mapsto \geq 2$. Depths two and three share a class, but popping produces depths one and two. Saving the predecessor signature in each frame resolves this difference.

Figure 4 traces the input `((a))` with this vocabulary, one token per row. Decoding starts at $(q_0, \kappa_0)$, whose mask row appears in the table $B$ on the right. Each `(` pushes $\gamma_p$ and advances the signature through the push transition, from $\kappa_0$ to $\kappa_1$ and then to $\kappa_2$; each pushed frame records the signature of the stack beneath it. The fourth token, `))`, spans two terminals and pops twice within one advancement. Each pop restores the signature stored in the removed frame, returning to $\kappa_0$ without traversing the stack. At every step, the current pair $(q, \kappa)$ selects the row of $B$ that gives the mask, including the context-dependent tokens `a)` and `))`.

### 3.2 Theoretical Justification

**Stack languages.** We use a normalized PDA with internal, single-symbol push, and matching single-symbol pop transitions, each labeled by a byte or epsilon. Acceptance is by a dedicated final control state. A bottom-marker transition enforces complete-document acceptance on runs from the initial configuration; generalized stack replacements can be normalized as described in Appendix B. Let $L(q, \sigma)$ be the language of finite runs from $(q, \sigma)$ to a final state, allowing epsilon transitions.

Let $S = \mathrm{suf}(\mathcal{T}) \cup \{\epsilon\}$ and $J = S \dot{\cup} \{\mathsf{eos}\}$. For $s \in S$, define

$$\mathcal{L}_{q,s} = \{\sigma : \exists z \in \Sigma^*,\ sz \in L(q, \sigma)\}, \qquad \mathcal{L}_{q,\mathsf{eos}} = \{\sigma : \epsilon \in L(q, \sigma)\}.$$

Thus an empty-suffix query asks whether some completion exists, whereas EOS asks whether the document can end now. These stack languages are effectively regular by pushdown reachability (Bouajjani et al., 1997); Appendix B.1 supplies the reduction. We use byte continuations throughout. Equivalence to realizable token continuations requires adequate vocabulary coverage, such as byte fallback.

**Definition 1** (Response Function and Stack Signatures)**.** The response of stack $\sigma$ is the Boolean vector

$$\mathcal{R}_\sigma : Q \times J \to \{0, 1\}, \qquad \mathcal{R}_\sigma(q, j) = \mathbf{1}[\sigma \in \mathcal{L}_{q,j}]. \tag{2}$$

Stacks are signature-equivalent, $\sigma_1 \sim_{\mathcal{R}} \sigma_2$, when their complete response vectors agree. Their equivalence classes $\mathcal{K}$ are the stack signatures.

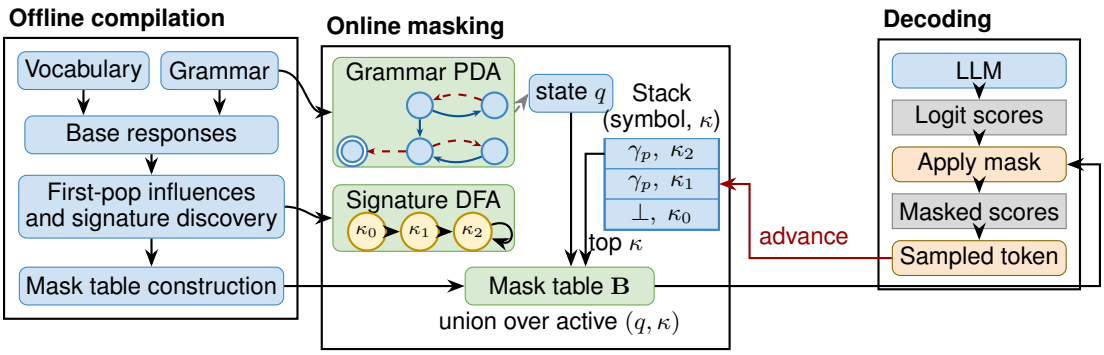



Figure 5: Overview of INSTAGRAMMAR. Offline, base responses, first-pop influences, and signature discovery produce the signature DFA and mask table $\mathbf{B}$. Online, each frame stores its saved signature; rows selected by (state, top signature) are combined over active configurations, and the sampled token advances the PDA, with pops restoring saved signatures.

Since each response vector has $|Q||J|$ Boolean entries, there are at most $2^{|Q||J|}$ of them. Finiteness alone does not give push congruence; the decomposition below does.

**Response decomposition.** Consider a run witnessing a query from $(q, \gamma\sigma)$. Until its first pop of the distinguished frame $\gamma$, it cannot inspect $\sigma$. Let $L(q, j, \gamma)$ indicate a successful run that never pops this frame. Let $\mathsf{Pop}(q, j, \gamma)$ collect the control state and residual query immediately after its first pop. For an ordinary query $s$, a pop before all of $s$ has been read leaves the corresponding suffix; a pop during the accepting continuation leaves the empty-suffix query. For EOS, only epsilon transitions are permitted and the residual query remains EOS.

**Theorem 1** (Response Decomposition)**.** *For the normalized PDA and query domain above,*

$$\mathcal{R}_{\gamma\sigma}(q, j) = L(q, j, \gamma) \vee \bigvee_{(q', j') \in \mathsf{Pop}(q, j, \gamma)} \mathcal{R}_{\sigma}(q', j'). \tag{3}$$

*Both summaries are independent of the stack $\sigma$ below $\gamma$.*

The proof splits a successful run at its first external pop and composes the resulting prefix with a successful parent continuation. It includes pops *after* the queried suffix has been consumed; merely surviving the token does not establish viable-prefix validity. Full details and effective summary computation appear in Appendix B.

**Corollary 2** (Signature DFA)**.** *The map $\delta_{\mathcal{K}}(\kappa(\sigma), \gamma) = \kappa(\gamma\sigma)$ is well-defined. Starting from $\kappa_0 = \kappa(\epsilon)$, it defines a finite automaton reading stack symbols from bottom to top. A push updates its state in one table access; a pop restores the predecessor signature saved in its frame.*

**Proposition 3** (Mask Table)**.** *For token $w_i$, set $B[q, \kappa][i] = \mathcal{R}_{\kappa}(q, w_i)$, and set the separate EOS bit to $\mathcal{R}_{\kappa}(q, \mathsf{eos})$. The row indexed by $(q, \kappa(\sigma))$ is the valid-token mask for $(q, \sigma)$.*

## 4 INSTAGRAMMAR

INSTAGRAMMAR compiles a grammar and vocabulary into response signatures, push transitions, and masks indexed by parser state and signature. Figure 5 summarizes the pipeline. Appendices D and E give the algorithms.

### 4.1 OFFLINE COMPILATION

**Phase 1: Base response.** For every state and query, compute $\mathcal{R}_{\epsilon}(q, j)$ with an initially empty stack. Record attempted pops of an absent symbol as uncertainty sets $U_{\gamma}$. Queries that share a prefix share work, and uncertainty found on a shared prefix applies to every query that extends it. The query monitor must also allow an accepting continuation after the suffix, while EOS permits epsilon moves only. A successful empty-stack run remains successful with an additional unseen frame; only

unsuccessful queries with a corresponding blocked pop need frame-specific recomputation. In the running example, the query ) ) from $q_1$ must pop $\gamma_p$ before reading its first ) ; with an empty stack that pop targets an absent frame, so $(q_1, ) ) )$ enters $U_{\gamma_p}$. These uncertain queries are exactly those whose answers depend on the stack below, and Phase 2 resolves them.

**Phase 2: Signature discovery.** First compute $L(q, j, \gamma)$ and $\mathsf{Pop}(q, j, \gamma)$ for the uncertain coordinates. Then initialize a worklist with $\kappa_0$ and evaluate Equation (3) for every push symbol from each discovered signature. Children with equal response vectors are the same signature, and each new vector is expanded once. This explores signatures obtainable by arbitrary stack-symbol sequences, which can include combinations never reached by a grammar execution. In the running example, consider the stacks $\gamma_p^d \perp$ of Figure 4. Pushing $\gamma_p$ onto $\perp$ resolves the single-closer query $(q_1, ) )$: its first pop removes $\gamma_p$, and completion then removes $\perp$. It does not resolve $(q_1, ) ) )$, whose second pop would need another $\gamma_p$. The child response is new and becomes $\kappa_1$. A second push also resolves $(q_1, ) ) )$, giving $\kappa_2$, and a third push reproduces the response of $\kappa_2$. The compiler additionally evaluates pushes from the empty stack and of the other stack symbols; Appendix G lists the resulting masks. Termination follows from finite summary computation and the finite vector count (Theorem 8).

**Phase 3: Mask projection.** Select whole-token response coordinates, $B[q, \kappa][i] = \mathcal{R}_\kappa(q, w_i)$, and the EOS coordinate. Identical masks can share storage. With $m = |Q||J|$, $k = |\mathcal{K}|$, $g = |\Gamma|$, and $e$ the total number of influence entries, dense child evaluation costs $O(k(gm + e))$ beyond summary construction; dense projection stores $|Q|k(|\mathcal{T}| + 1)$ bits. Summary construction, intermediate responses, and runtime stacks have separate costs (Appendix B).

**Lexer/parser factorization.** We group vocabulary tokens by the sequences of grammar terminals they traverse, so the parser works over terminals rather than bytes. A query can end at a terminal boundary, inside a partially read terminal, or at EOS, and the same first-pop construction applies to these finite parser queries. A boundary row is indexed by $(q, \kappa)$; a mid-terminal row additionally records the terminal, its lexer state, and the parser state reached after the terminal begins. These queries match byte-level viability when every reachable parser continuation is productive and the lexical classification is complete. Appendix E states the predicates precisely.

### 4.2 Online Mask Retrieval

**Push and pop.** Each frame retains its concrete stack symbol and the signature of the stack below it. A push saves the current signature and applies $\delta_{\mathcal{K}}$; a pop removes the frame and restores the saved signature. The concrete stack is kept alongside the signatures. Signature maintenance requires constant time per stack operation; the current signature alone does not determine a pop's result.

**Nondeterminism and mask construction.** For an active set $\mathcal{A}$, the mask is

$$mask = \bigvee_{(q,\hat{\sigma}) \in \mathcal{A}} B[q, \textsc{SigOf}(\hat{\sigma})].$$

Equal row keys can be deduplicated for this union, while distinct concrete stacks are retained for future updates. With $a$ active entries, $d$ distinct rows, vocabulary size $V$, and word width $W$, enumerating keys costs at least $O(a)$ and dense materialization costs $O((d + 1)\lceil (V + 1)/W \rceil)$ word operations. Reading one stored row takes constant time. The selected token still undergoes byte-level parsing and epsilon closure, and logit-mask application is separate. Proposition 7 makes these costs explicit.

## 5 Experimental Evaluation

We evaluate InstaGrammar against five baselines on MaskBench (Geng et al., 2025; guidance-ai, 2025), measuring preprocessing and per-token matcher cost, and serving throughput on schemas with heavy online cost.

### 5.1 Setup

**Baselines and workloads.** We compare llama.cpp (Gerganov et al., 2023), Outlines (Willard & Louf, 2023), Formatron (Sun et al., 2025), llguidance (guidance-ai, 2026), and XGrammar (Dong et al.,

Table 1: Six-engine comparison on a 7,029-schema MaskBench subset (Appendix A.1). Geometric means and medians are across dataset-family means.

| | Compile/load (ms) | | TBM ($\mu$s) | |
|---|---|---|---|---|
| Engine | GeoMean | Median | GeoMean | Median |
| llama.cpp | 0.5 | 0.4 | 33,026.7 | 32,586.0 |
| Outlines | 4,629.0 | 7,747.0 | 602.9 | 570.3 |
| Formatron | 3,842.2 | 1,638.1 | 1,792.6 | 1,939.0 |
| llguidance | 0.7 | 0.5 | 20.0 | 19.3 |
| XGrammar | 32.0 | 13.3 | 38.6 | 23.6 |
| INSTAGRAMMAR | 187.1 | 98.0 | 11.4 | 13.8 |

Table 2: Matcher cost (TBM, $\mu$s) on the 6,740 schemas passing under all three engines, for the cold and warm passes of Section 5.1. "INSTAGRAMMAR faster" is the share of schemas where INSTAGRAMMAR's per-schema mean TBM is lower. Compile errors are over all 9,795 schemas with test instances; validation outcomes are in Appendix A.3.

| Engine | Pass | Mean | P50 | P90 | P99 | P99.9 | INSTAGRAMMAR faster | Compile errors |
|---|---|---|---|---|---|---|---|---|
| INSTAGRAMMAR | cold | **7.4** | 4 | **14** | **52** | **148** | – | **3** |
| | warm | **3.6** | **3** | **5** | **10** | **19** | – | |
| llguidance 1.7.6 | cold | 17.9 | 9 | 31 | 152 | 690 | 90.8% | 886 |
| | warm | 15.1 | 8 | 26 | 118 | 683 | 99.7% | |
| XGrammar 0.2.4 | cold | 51.1 | **2** | 31 | 601 | 4,460 | 37.5% | 70 |
| | warm | 51.4 | **2** | 31 | 604 | 4,273 | 71.4% | |

2025). MaskBench supplies JSON schemas with labeled valid and invalid instances. Table 1 uses a 7,029-schema subset of MaskBench and Table 2 the full 11,306-input suite. All engines use the Llama-3.1 tokenizer (Grattafiori et al., 2024) and eight single-thread workers. Appendix A.1 gives the populations, versions, hardware, and aggregation.

**Metrics.** *Compile/load time* is the time to prepare an engine for a schema, including compilation and matcher setup. *Time between masks (TBM)* is the cost of computing a mask and advancing on the selected token; it excludes model inference. The runner replays each schema's labeled instances rather than generating with an LLM. The *cold* pass replays them on a freshly compiled grammar; the *warm* pass replays them again on the same grammar, modeling a schema reused across requests. Engines can reject an invalid instance at different tokens, so traces are not matched.

### 5.2 MATCHER LATENCY

**Six-engine comparison.** INSTAGRAMMAR has the lowest family-aggregated TBM in Table 1, 11.4 $\mu$s against 20.0 $\mu$s for llguidance and 38.6 $\mu$s for XGrammar, with a higher compile/load cost than llama.cpp, llguidance, and XGrammar. Figure 9 shows the per-schema distributions; Appendix A breaks them down by dataset family.

**Cold and warm matcher cost.** Table 2 compares INSTAGRAMMAR, llguidance, and XGrammar 0.2.4 on the 6,740 schemas that pass under all three. Against llguidance, INSTAGRAMMAR has a lower per-schema mean TBM on 90.8% of schemas cold and 99.7% warm, and a lower mean, median, and tail on both passes. Against XGrammar, XGrammar has the lower median (2 versus 4 $\mu$s) and is faster on 62.5% of schemas cold, while INSTAGRAMMAR's mean is 7 times lower and its P99 and P99.9 are 12 and 30 times lower. On the warm pass INSTAGRAMMAR is faster on 71.4% of schemas, with a P99 of 10 $\mu$s against 604 $\mu$s: its first-visit mask assembly is cached, whereas XGrammar's per-token cost does not change with reuse.

**Coverage.** INSTAGRAMMAR compiles all but 3 of 9,795 tested schemas, against 886 compilation errors for llguidance and 70 for XGrammar, at a higher cost: a median of 96 ms against 0.6 and 1.3 ms.

**Per-token behavior.** Figure 6 shows the first nine tokens of a valid instance of the WashingtonPost `wp_58` schema. INSTAGRAMMAR's cost stays between 2.4 and 2.6 $\mu$s per token. XGrammar is

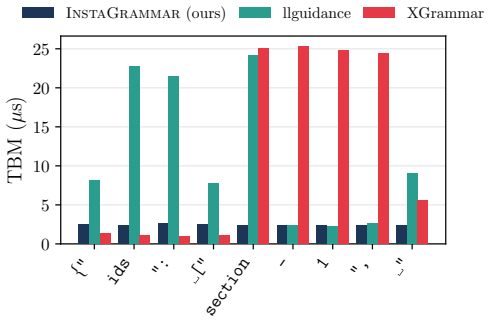



Figure 6: Per-token TBM on a valid `wp_58` instance (median of 30 replays).

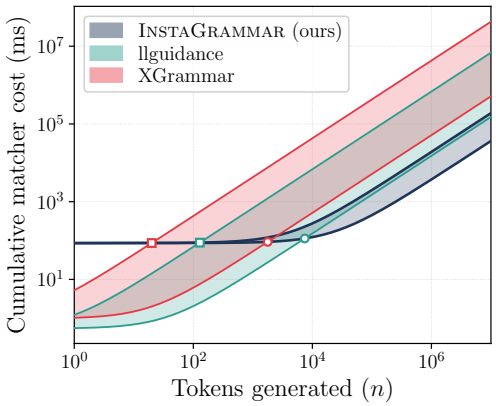



Figure 7: Modeled cumulative matcher cost $C + nt$ with grammar reuse. Shaded areas lie between the P99.9 and mean warm-pass TBM.

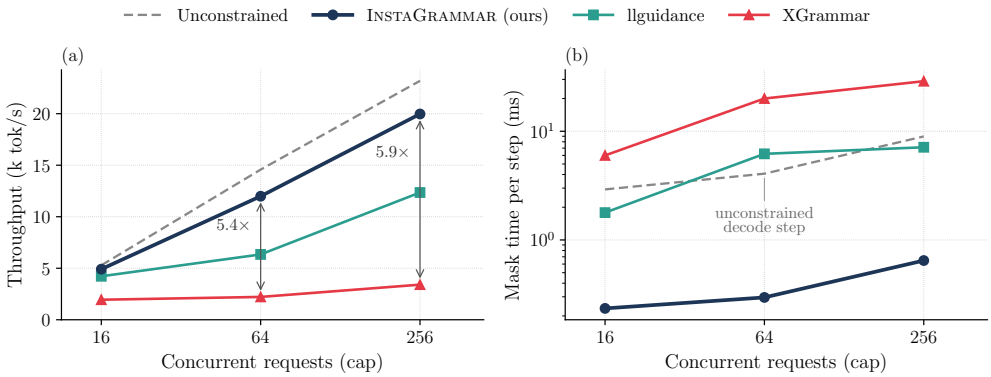



Figure 8: vLLM serving on the 155 tail schemas of Section 5.4. (a) Output throughput; arrows give InstaGrammar's speedup over XGrammar. (b) Mask time per decode step, against the unconstrained decode step.

fastest on the opening tokens but rises to about 25 $\mu$s on the `section-1` run, and llguidance varies between 2 and 24 $\mu$s.

### 5.3 Preprocessing and Amortized Cost

For a fixed grammar, let $C$ be compile/load time and $t$ the per-token matcher cost. Reusing its compiled artifact for $n$ tokens gives cumulative cost $C + nt$. Compared with a baseline, a positive crossover exists when $C_I > C_B$ and $t_B > t_I$, at

$$n^* = \frac{C_I - C_B}{t_B - t_I}. \tag{4}$$

Figure 7 applies this model to the engines of Table 2 when a grammar is reused: $C$ is the median compile/load time over the 6,740 common schemas and $t$ the warm-pass TBM, either its mean (the lower edge of each band) or its P99.9 (the upper edge). With mean costs, InstaGrammar's cumulative cost falls below XGrammar's after about 1,800 tokens and below llguidance's after about 7,500 (circles); with P99.9 costs, after about 20 and 130 tokens (squares). Both edges use aggregate costs, so they are illustrative, not per-grammar break-even points. A single short use of a new schema does not amortize preprocessing. This model counts matcher cost only; Section 5.4 measures serving throughput directly, and Appendix A.4 breaks compile/load time down by dataset family.

### 5.4 SERVING THROUGHPUT

The tails of Table 2 come from schemas on which an engine resolves many tokens online. To measure their cost in serving, we run INSTAGRAMMAR, llguidance, and XGrammar as structured-output backends of vLLM 0.27.1 on every schema that all three handle correctly and on which llguidance or XGrammar averages at least 200 $\mu$s per token in the warm pass (155 schemas). Llama-3.2-1B-Instruct generates four responses per schema on one RTX Pro 6000, with at most 16, 64, or 256 concurrent requests; grammars are compiled before timing (Appendix A.5).

At 64 and 256 concurrent requests, INSTAGRAMMAR delivers 5.4 and 5.9 times XGrammar's output throughput and 1.9 and 1.6 times llguidance's, retaining 82–86% of unconstrained throughput (Figure 8). XGrammar spends 20–29 ms per decode step computing masks, several times the 4–9 ms unconstrained step, so the grammar rather than the model paces decoding; INSTAGRAMMAR spends 0.3–0.7 ms. On 155 randomly sampled schemas the three engines are within 11%, with INSTAGRAMMAR fastest at 64 and 256 requests: the gain is specific to schemas with heavy online cost.

### 5.5 LIMITATIONS

The signature space can grow exponentially, and computing local summaries by enumerating configurations can fail to terminate on epsilon-push cycles. Exact masks for the converted grammar do not imply that the converter enforces every schema keyword: Appendix F.1 lists the JSON Schema keywords the converter enforces. Apart from the serving study, which uses selected tail schemas and a 1B-parameter model, our measurements replay recorded instances; they do not establish serving throughput on typical workloads or larger models, or complete EOS correctness. New tokenizers, programming-language grammars, and short-lived dynamic schemas require their own coverage and resource measurements, as does peak compilation memory.

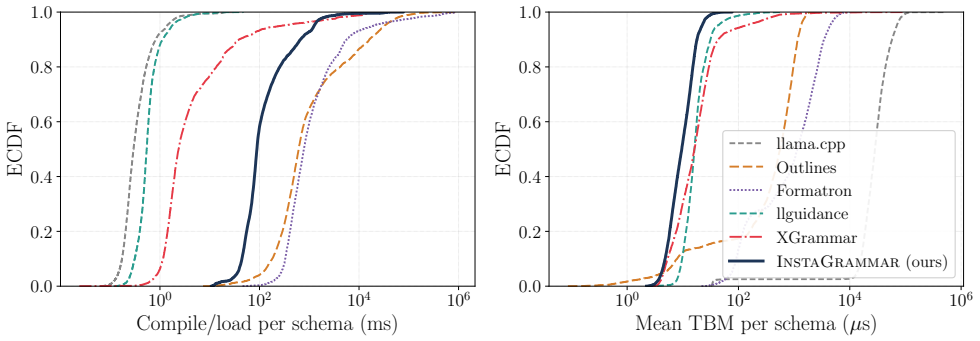



Figure 9: Per-schema compile/load time (left) and mean TBM (right) for the engines of Table 1.

## 6 CONCLUSION

We presented INSTAGRAMMAR, which shows that the stack dependence of grammar-constrained decoding need not be resolved while decoding: for a fixed grammar and vocabulary, stacks fall into finitely many signatures that each push or pop updates in constant time, so every mask is selected from precomputed rows. Across six engines, INSTAGRAMMAR has the lowest per-token matcher cost at a higher compile cost (Figure 9). Its advantage is largest in the tail: under grammar reuse its P99 falls to 10 $\mu$s, and on schemas with heavy online cost it delivers up to 5.9 times XGrammar's vLLM serving throughput. Because a serving step waits for the masks of every request in the batch, throughput on such schemas follows the per-token tail rather than the median. Its preprocessing, a median of 96 ms per schema, suits schemas reused across requests; reducing that cost and extending signatures to SQL and programming-language grammars are the main directions for future work.

## Reproducibility Statement

Sections 3 and 4 define the semantics and construction; Appendices B, D, and E give proofs and pseudocode. Section 5 and Appendix A state the evaluated populations, recorded configurations, aggregation rules, and timing boundaries. Appendix F lists every input of the evaluated suite. The reported comparisons retain failures and untested inputs and state which measurements cannot establish matched-trace or end-to-end performance.

## AI Use Statement

We used agentic tools to assist with manuscript revision, literature and reference checking, checking of mathematical arguments, experimental-protocol review, analysis of recorded results, and preparation of analysis code and figures. AI assistance also supported implementation and development. The paper's authors are responsible for the final text, claims, experimental interpretation, and artifacts.

## References

Faruk Alpay and Bilge Senturk. Attention meets reachability: Structural equivalence and efficiency in grammar-constrained LLM decoding. *arXiv preprint arXiv:2603.05540*, 2026. URL https://arxiv.org/abs/2603.05540.

Luca Beurer-Kellner, Marc Fischer, and Martin Vechev. Guiding LLMs the right way: Fast, non-invasive constrained generation. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 3658–3673. PMLR, 2024. URL https://proceedings.mlr.press/v235/beurer-kellner24a.html.

Ahmed Bouajjani, Javier Esparza, and Oded Maler. Reachability analysis of pushdown automata: Application to model-checking. In *Proceedings of the 8th International Conference on Concurrency Theory*, CONCUR '97, pp. 135–150, Berlin, Heidelberg, 1997. Springer-Verlag. ISBN 3540631410. doi: 10.1007/3-540-63141-0_10.

Junyi Chen, Shihao Bai, Zaijun Wang, Siyu Wu, Chuheng Du, Hailong Yang, Ruihao Gong, Shengzhong Liu, Fan Wu, and Guihai Chen. Pre$^3$: Enabling deterministic pushdown automata for faster structured LLM generation. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 11253–11267, Vienna, Austria, 2025. Association for Computational Linguistics. doi: 10.18653/v1/2025.acl-long.551.

Yixin Dong, Charlie F. Ruan, Yaxing Cai, Ziyi Xu, Yilong Zhao, Ruihang Lai, and Tianqi Chen. XGrammar: Flexible and efficient structured generation engine for large language models. In *Proceedings of Machine Learning and Systems*, volume 7, 2025. URL https://proceedings.mlsys.org/paper_files/paper/2025/hash/5c20ca4b0b20b0bd2f1d839dc605e70f-Abstract-Conference.html.

Saibo Geng, Martin Josifoski, Maxime Peyrard, and Robert West. Grammar-constrained decoding for structured NLP tasks without finetuning. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2023. URL https://aclanthology.org/2023.emnlp-main.674/.

Saibo Geng, Hudson Cooper, Michał Moskal, Samuel Jenkins, Julian Berman, Nathan Ranchin, Robert West, Eric Horvitz, and Harsha Nori. JSONSchemaBench: A rigorous benchmark of structured outputs for language models, 2025. URL https://arxiv.org/abs/2501.10868.

Georgi Gerganov et al. llama.cpp: LLM inference in C/C++. https://github.com/ggml-org/llama.cpp, 2023. Accessed September 24, 2026.

Aaron Grattafiori et al. The Llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024. URL https://arxiv.org/abs/2407.21783.

guidance-ai. MaskBench. https://github.com/guidance-ai/jsonschemabench/tree/main/maskbench, 2025.

guidance-ai. llguidance: Super-fast structured outputs. GitHub repository and design documentation, 2026. URL https://github.com/guidance-ai/llguidance. Accessed September 24, 2026.

John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.

Terry Koo, Frederick Liu, and Luheng He. Automata-based constraints for language model decoding. In *First Conference on Language Modeling (COLM)*, 2024. URL https://openreview.net/forum?id=BDBdblmyzY.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP)*, pp. 611–626. ACM, 2023. doi: 10.1145/3600006.3613165.

Linzhang Li, Yixin Dong, Guanjie Wang, Ziyi Xu, Alexander Jiang, and Tianqi Chen. XGrammar-2: Dynamic and efficient structured generation engine for agentic LLMs. In *Proceedings of the ACM Conference on AI and Agentic Systems (CAIS)*. ACM, 2026a. doi: 10.1145/3786335.3813124.

Yongmin Li, Yihong Dong, Jia Li, and Ge Li. Efficient grammar-constrained decoding via parser stack classification, 2026b. URL https://arxiv.org/abs/2608.03065v1.

NousResearch. json-mode-eval. https://huggingface.co/datasets/NousResearch/json-mode-eval, 2024.

Kanghee Park, Timothy Zhou, and Loris D'Antoni. Flexible and efficient grammar-constrained decoding. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 48262–48275. PMLR, 2025. URL https://proceedings.mlr.press/v267/park25l.html.

Shishir G. Patil, Huanzhi Mao, Fanjia Yan, Charlie Cheng-Jie Ji, Vishnu Suresh, Ion Stoica, and Joseph E. Gonzalez. The Berkeley function calling leaderboard (BFCL): From tool use to agentic evaluation of large language models. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 48371–48392. PMLR, 2025. URL https://proceedings.mlr.press/v267/patil25a.html.

Gabriel Poesia, Oleksandr Polozov, Vu Le, Ashish Tiwari, Gustavo Soares, Christopher Meek, and Sumit Gulwani. Synchromesh: Reliable code generation from pre-trained language models. In *International Conference on Learning Representations (ICLR)*, 2022. URL https://openreview.net/forum?id=KmtVD97J43e.

Xintong Sun, Chi Wei, Minghao Tian, and Shiwen Ni. Earley-driven dynamic pruning for efficient structured decoding. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 57773–57789. PMLR, 2025. URL https://openreview.net/forum?id=6hDNXCdTsE.

Shubham Ugare, Tarun Suresh, Hangoo Kang, Sasa Misailovic, and Gagandeep Singh. Syncode: LLM generation with grammar augmentation. *Transactions on Machine Learning Research*, 2025. ISSN 2835-8856. URL https://openreview.net/forum?id=HiUZtgAPoH.

Brandon T. Willard and Rémi Louf. Efficient guided generation for large language models. *arXiv preprint arXiv:2307.09702*, 2023. URL https://arxiv.org/abs/2307.09702.

# A SUPPLEMENTAL RESULTS

This appendix expands Section 5 with family-level plots, complete outcome denominators, and pooled tail statistics. The six-engine archive and the broader four-engine study are separate experiments; their aggregate values should not be combined.

## A.1 EVALUATION POPULATIONS AND SETTINGS

**Populations.** Table 1 uses 7,029 MaskBench schemas: those with at least one valid instance that use none of the keywords `format`, `minimum`, `maximum`, `exclusiveMinimum`, `exclusiveMaximum`, `multipleOf`, `not`, `if`/`then`/`else`, `dependencies`, or `uniqueItems`, i.e. constraints on formats, numeric ranges, negation, conditionals, dependencies, and uniqueness that the grammar frontends do not enforce. A canonical-family filter retains 6,976 of these before compilation failures; successful timing cohorts range from 3,839 to 6,947 schemas (Table 3). Table 2 and the coverage study below use all 11,306 inputs of the suite (Appendix F), including handwritten and synthesized cases.

**Measurement.** Compile/load time is the runner's `compile_grammar` call, including adapter initialization and any prewarming inside it. For INSTAGRAMMAR this comprises compilation, matcher setup, and a bounded prewarm of the initial state and its immediate successors; masks for states first visited later are assembled during replay and counted in TBM. It is not an isolated compiler-phase measurement or a direct measurement of the first returned mask. TBM excludes GPU logit-mask application. Both valid and invalid instances contribute timings until processing ends, so engines can record different token counts for the same schema. Compilation failures are reported separately rather than assigned timings.

**Aggregation.** For Table 1, token times are averaged within each schema, schema means within each dataset family, and the geometric mean and median are taken across family means, so each family receives equal weight. These statistics differ from a geometric mean across schemas and from pooled per-token quantiles; no schemas are trimmed. Engine-specific successful cohorts differ, so ratios of these aggregates are not paired speedups. Table 2 pools token times over the schemas that pass under all three engines, and compares per-schema means pairwise.

**Hardware and versions.** All runs use the Llama-3.1 tokenizer (an immutable tokenizer revision is not available) and eight single-thread workers. The llama.cpp, Outlines, Formatron, and XGrammar rows of Table 1 are archived cluster runs without complete version and hardware pins. The llguidance and INSTAGRAMMAR rows of Table 1, and all of Table 2, were measured on one workstation (Intel i9-14900KF) on September 25, 2026 with a 300 s per-schema time limit and a 14 GiB address-space limit, using llguidance 1.7.6 (NumPy 2.2.6) and XGrammar 0.2.4 (NumPy 2.4.6, CPU PyTorch 2.6.0). The XGrammar row of Table 4 comes from a cluster run on September 11, 2026 with a 900 s time limit and a 28 GiB address-space limit. Exact CPU affinity and repeated-run uncertainty are not recorded.

**llguidance adapter.** An earlier llguidance adapter raised an integer-overflow error in its mask-bit check whenever $t \bmod 32 = 31$; the runner recorded these as validation errors and truncated replay, affecting 3,021 archived inputs. All llguidance results reported here use the corrected adapter.

## A.2 SIX-ENGINE COHORT AND FAMILY BREAKDOWNS

Each engine has 7,029 recorded inputs. The canonical-family filter excludes 53 records per engine, leaving 6,976 candidates. Table 3 gives the resulting counts. Compilation errors contribute to coverage but have no successful latency measurement. The 17 represented families comprise the five GitHub complexity splits, Glaiveai2K, JsonSchemaStore, Kubernetes, MCPspec, WashingtonPost, and seven BFCL splits; Snowplow has no included timing family in this archive. Formatron has successful observations in only 15 of those families. The family selection is therefore narrower than the full benchmark inventory.

Figures 10 and 11 report *dataset-family means*, rather than one value for each grammar. A family's TBM first averages token times within each schema and then averages those schema means. Missing bars indicate absent successful observations. Compilation and per-token costs vary across families; a finite signature space does not imply uniformly small compilation cost or constant total matcher time.

Table 3: Six-engine archive denominators after the common 53-record family exclusion. "Timed" plus "Compile errors" equals 6,976 for each row. Successful records can still have validation errors. †Measured on the workstation (Appendix A.1); other rows are archived cluster runs.

| Engine | Timed schemas | Compile errors | Families | Token samples |
|---|---|---|---|---|
| llama.cpp | 6,510 | 466 | 17 | 1,170,745 |
| Outlines | 6,561 | 415 | 17 | 1,383,357 |
| Formatron | 3,839 | 3,137 | 15 | 352,887 |
| llguidance† | 6,800 | 176 | 17 | 1,771,307 |
| XGrammar | 6,947 | 29 | 17 | 1,729,762 |
| INSTAGRAMMAR† | 6,974 | 2 | 17 | 1,883,680 |

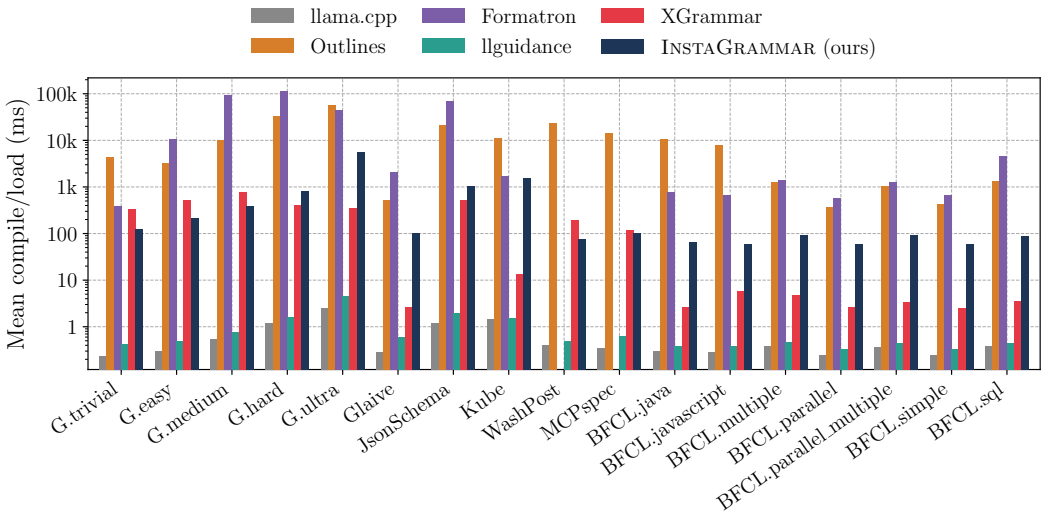



Figure 10: Arithmetic mean compile/load time within each represented dataset family. Successful engine-specific cohorts are used; the vertical axis is logarithmic.

### A.3 BROADER COVERAGE

This study attempts all 11,306 inputs; the settings are given in Appendix A.1.

Table 4: Outcomes over all 11,306 attempted schemas in the broader study. Categories are disjoint runner outcomes; "No error" is not a full-mask or EOS correctness certificate. Pending and unreadable results remain in the denominator. †Workstation runs; the XGrammar row is a cluster run.

| Engine | No error | Validation | Compile | No tests | Pending | Unreadable |
|---|---|---|---|---|---|---|
| INSTAGRAMMAR† | 7,718 | 2,073 | 3 | 1,511 | 1 | 0 |
| XGrammar 0.2.4 | 8,061 | 1,664 | 70 | 1,511 | 0 | 0 |
| llguidance 1.7.6† | 8,887 | 22 | 886 | 1,511 | 0 | 0 |

Compilation has a long tail: over all 9,791 successfully compiled INSTAGRAMMAR schemas, the median is 0.097 s, P95 is 1.630 s, P99 is 5.881 s, and the maximum is 149.759 s. These exclude failed and timed-out compilation and therefore are not timeout-penalized scores.

The runner checks supplied instance labels but not EOS, and stores only the last error per schema, so these counts neither establish full-mask equivalence nor certify all JSON Schema features. Converter support and matcher correctness are distinct: a converter can omit a schema constraint even if the matcher implements its resulting grammar exactly.

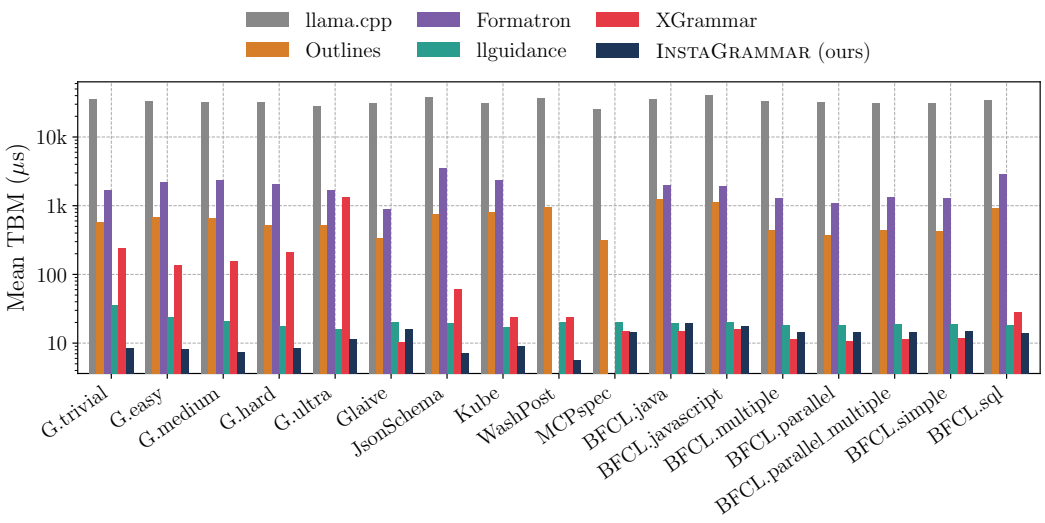



Figure 11: Arithmetic mean of per-schema TBM means within each represented family. TBM includes selected-token advancement. Family and successful-schema coverage differ by engine.

Table 5: Compiled grammar statistics per dataset family.

| | | $\|\mathcal{K}\|$ | | Tables | Median compile/load (ms) | | |
|---|---|---|---|---|---|---|---|
| Family | Schemas | Mean | Max | (MB) | InstaGrammar | llguidance | XGrammar |
| GitHub trivial | 365 | 48 | 732 | 16.0 | 43 | 0.4 | 0.5 |
| GitHub easy | 1,789 | 75 | 464 | 22.2 | 70 | 0.5 | 0.9 |
| GitHub medium | 1,823 | 193 | 2,454 | 37.4 | 180 | 0.7 | 3.3 |
| GitHub hard | 887 | 614 | 5,314 | 68.9 | 459 | 1.4 | 9.6 |
| GitHub ultra | 95 | 2,961 | 16,541 | 205.5 | 2,498 | 6.5 | 45 |
| Glaive | 1,634 | 108 | 238 | 19.3 | 86 | 0.6 | 0.8 |
| JSON Schema Store | 381 | 927 | 31,429 | 108.6 | 391 | 1.0 | 11 |
| Kubernetes | 1,014 | 985 | 11,741 | 63.5 | 144 | 0.7 | 1.6 |
| Snowplow | 394 | 136 | 2,886 | 44.0 | 150 | 0.6 | 345 |
| Washington Post | 99 | 137 | 3,667 | 21.1 | 66 | 0.5 | 1.8 |
| MCP spec | 44 | 225 | 1,448 | 24.6 | 88 | 0.6 | 30 |
| JSON-mode-eval | 100 | 85 | 180 | 18.9 | 96 | 0.5 | 0.8 |
| Handwritten | 13 | 323 | 3,756 | 35.6 | 17 | – | 0.7 |
| Synthesized | 109 | 3 | 3 | 5.7 | 27 | – | 0.7 |
| BFCL (7 splits) | 1,043 | 23 | 103 | 12.4 | 56 | 0.4 | 0.8 |
| All | 9,790 | 304 | 31,429 | 37.6 | 95 | 0.6 | 1.3 |

For InstaGrammar, the completed successfully compiled cohort matches 13,218 of 13,958 supplied valid labels and 19,604 of 23,035 invalid labels. Object keys are accepted only in declared schema order (Appendix F.1), so valid instances listing properties in another order count as mismatches. These are instance-level counters on that cohort, not proportions of the full input suite. Missing EOS checks and absent test instances limit their interpretation; they do not support a claim of universal JSON Schema conformance. Failures may arise in conversion, compilation, or matching, and cannot collectively be explained as non-context-free constraints.

### A.4 Compile Cost by Dataset Family

Table 5 breaks preprocessing down by dataset family. InstaGrammar's compile/load time grows with its number of trace signatures, from a median of about 50 ms on BFCL function schemas to 2.5 s on GitHub ultra, while llguidance and XGrammar compile most schemas in about 1 ms. Its compiled serving tables average 38 MB per schema.

### A.5 Serving Setup and Control Workload

**Backends.** vLLM selects its structured-output backend from a fixed list, so we add InstaGrammar by patching a private copy of vLLM 0.27.1; every configuration, including the stock XGrammar 0.2.3 and llguidance 1.7.6 backends and the unconstrained control, runs this same copy with the V1 model runner. The same wall-clock probe around vLLM's bitmask computation gives the per-step mask time for every backend.

**Workloads.** Both workloads draw from the 6,700 schemas that all three engines handle correctly in the warm pass of Table 2 and that InstaGrammar compiles in under 10 s. The *tail* workload is every such schema on which llguidance's or XGrammar's warm mean TBM is at least 200 $\mu$s (155 schemas; means of 2.9, 78, and 510 $\mu$s for InstaGrammar, llguidance, and XGrammar). The *control* workload is a uniform random sample of 155 of the 6,700 (means of 3.0, 13.9, and 18.3 $\mu$s).

**Requests.** Each prompt asks for a JSON instance of the schema (schema text truncated to 6,000 characters; the grammar always enforces the full schema), sampled at temperature 0.7 with at most 512 new tokens and a per-request seed shared by all backends. vLLM compiles XGrammar schemas in strict mode, which forbids undeclared object properties, while llguidance and InstaGrammar follow the JSON Schema default of allowing them; we therefore add `additionalProperties: false` to every object schema that declares properties without setting it, for all engines (vLLM's own `disable_additional_properties` transformation). All requests are submitted to the offline engine at once, so the cap sets the number in flight.

**Compilation.** One untimed request per schema compiles its grammar before timing. For the 155 tail schemas this takes 64 s for InstaGrammar, 11 s for XGrammar, and 2 s for llguidance. XGrammar rejects 3 tail and 4 control schemas and llguidance 1 control schema at this step; those schemas are excluded from that engine's timed run. Mean output lengths on the tail are 145–148 tokens for InstaGrammar, 163–164 for llguidance, 132–134 for XGrammar, and about 272 unconstrained.

Table 6: Output throughput (tokens/s) and mask time per decode step (ms) by workload and maximum number of concurrent requests.

| | | Throughput (tok/s) | | | Mask time per step (ms) | | |
|---|---|---|---|---|---|---|---|
| Workload | Engine | 16 | 64 | 256 | 16 | 64 | 256 |
| Tail | Unconstrained | 5,287 | 14,573 | 23,193 | – | – | – |
| | InstaGrammar | **4,909** | **11,989** | **19,973** | **0.23** | **0.30** | **0.65** |
| | llguidance | 4,221 | 6,341 | 12,351 | 1.78 | 6.20 | 7.13 |
| | XGrammar | 1,943 | 2,216 | 3,407 | 5.99 | 20.01 | 28.88 |
| Control | Unconstrained | 5,258 | 14,201 | 22,992 | – | – | – |
| | InstaGrammar | 4,870 | **11,813** | **19,393** | **0.19** | **0.22** | **0.54** |
| | llguidance | **4,873** | 11,792 | 18,231 | 0.29 | 0.89 | 1.35 |
| | XGrammar | 4,844 | 11,260 | 17,412 | 0.38 | 1.24 | 1.99 |

## B Theoretical Results

### B.1 Regularity of Stack Languages

The finite response count follows directly from the finite query domain. Regularity supplies effective computation of the query languages and influence summaries.

**Proposition 4** (Regularity of stack languages)**.** *For each $q \in Q$ and $j \in J$, $\mathcal{L}_{q,j}$ is an effectively regular language over $\Gamma$.*

*Proof.* For $j = s$, construct a finite monitor that first requires the bytes of $s$ in order and then permits arbitrary bytes. Epsilon transitions leave its position unchanged. Product the monitor with the PDA control. The target consists of completed-monitor states paired with final PDA states and arbitrary stacks, a regular configuration set. Erasing input labels turns existence of a matching run into pushdown reachability. Predecessor saturation (Bouajjani et al., 1997) computes a finite automaton for all configurations reaching this target. Its cross-section at the initial monitor position and state $q$ is

$\mathcal{L}_{q,s}$. For EOS, permit only epsilon transitions from the outset and use the final-state target. Saturation terminates on a finite automaton representation even when concrete configuration enumeration would not. □

**Corollary 5** (Finite index of $\sim_{\mathcal{R}}$)**.** *The equivalence of Definition 1 has at most* $2^{|Q||J|}$ *classes.*

*Proof.* Each of its classes is determined by a Boolean vector with $|Q||J|$ coordinates. This argument does not identify membership agreement with a Myhill–Nerode congruence; push congruence follows from Theorem 1. □

**Normalization and complete acceptance.** A transition replacing top $\gamma$ by $\alpha_1 \cdots \alpha_r$ can be replaced by a transition performing its matching pop and original input action, followed by epsilon pushes in reverse order through private intermediate states. The increase is $O(r+1)$ states/transitions. Intermediate states are not final. An empty-stack or bottom-marker completion convention is converted to final-state acceptance using a protected marker and a dedicated completion transition. This preserves the language of runs from the initial configuration. Our offline vectors are defined on all state/stack combinations of the normalized machine, including unreachable ones; they need not match another acceptance convention on those unreachable combinations.

### B.2 Proof of Theorem 1

*Proof.* Suppose $\mathcal{R}_{\gamma\sigma}(q, j) = 1$. For an ordinary query $s$, choose a finite accepting run whose input begins with $s$. If it never pops the distinguished initial frame $\gamma$, it witnesses $L(q, s, \gamma) = 1$. Otherwise split it immediately after that frame's first pop. If the run has consumed only a prefix of $s$, let $s'$ be the remaining suffix. If it has consumed all of $s$ and possibly part of its continuation, let $s' = \epsilon$. The split witnesses $(q', s') \in \mathsf{Pop}(q, s, \gamma)$, and the remaining run witnesses $\mathcal{R}_{\sigma}(q', s') = 1$. For EOS the same argument uses only epsilon transitions and retains the EOS query.

Conversely, a successful run that never pops $\gamma$ can be executed above every $\sigma$: all its stack tests concern locally pushed symbols or $\gamma$, and acceptance depends on the final control state. For an influence entry with true parent response, concatenate the run prefix exposing the parent stack with a witnessing parent continuation. The concatenation respects the required suffix, or the epsilon-only restriction for EOS, and reaches a final state. This proves both implications. □

### B.3 Local Independence

**Lemma 6** (Local Independence)**.** *The summaries* $L(q, j, \gamma)$ *and* $\mathsf{Pop}(q, j, \gamma)$ *depend only on the normalized transition system and* $(q, j, \gamma)$*, and are effectively computable.*

*Proof.* Tag the distinguished frame separately from symbols pushed above it. Pair the parser with the query monitor from Proposition 4. Until the first tagged-frame pop, all enabled transitions are determined locally. For $L$, prohibit that pop and ask whether a final control state with completed monitor is reachable. For Pop, turn the first tagged-frame pop into a terminal reporting transition; its finite report identifies the destination state and remaining monitor query. Each reporting target is regular. Finite pushdown saturation determines local success and which reports are reachable, independently of the unseen parent stack. In particular, saturation continues after the required bytes have been read, because an accepting continuation can itself expose the parent. □

### B.4 Online Mask Construction

**Proposition 7** (Row access and mask construction)**.** *Let* $a$ *be the number of active configurations and* $d \le \min(a, |Q||\mathcal{K}|)$ *the number of distinct byte-level row keys. Access to one materialized densely indexed row costs* $O(1)$*. Enumerating keys requires* $O(a)$ *work before deduplication; comparison-sort deduplication takes* $O(a \log a)$ *in the usual sorting model. Materializing a dense mask from* $d$ *rows takes* $O((d+1)\lceil (V+1)/W \rceil)$ *word operations. These bounds exclude selected-token advancement and logit-mask application.*

*Proof.* Read the stored signature of each active stack and construct its row key. Sorting and adjacent equality tests identify the distinct keys. Each row has $\lceil (V+1)/W \rceil$ words; initialize the output and

OR the selected rows. Idempotence reduces the number of rows merged but does not eliminate the initial scan of active configurations. □

For the factored implementation, count mid-terminal entries in $a$ and include their terminal/DFA state and destination state in the key space. A whole-mask cache hit still constructs its key and copies the output. A cold row additionally inspects response bits and unions precomputed group masks; sparse response layouts may require a search rather than dense indexing. Concrete parser histories remain distinct even when their current masks agree. Updating them depends on sampled-token length, branching, and epsilon closure. Fixed runtime capacities are resource limits, not complexity bounds for arbitrary grammars.

### B.5 Convergence and Signature Count

**Theorem 8** (Fixed-Point Convergence)**.** *With exact finite summaries and exact response equality, Algorithm 1 discovers every signature induced by a finite stack string. It expands at most $|\mathcal{K}| \leq 2^{|Q||J|}$ signatures and evaluates at most $|\Gamma||\mathcal{K}|$ children.*

*Proof.* The initial class represents the empty stack. Inductively, every push from a discovered class gives the correct child by Theorem 1; hence every finite stack's class is eventually discovered. A new vector is enqueued exactly once, so the finite vector count bounds the number of expansions. Each expansion evaluates one child per stack symbol. This proof concerns signature reachability under arbitrary pushes, which is a superset of configurations reached by input execution. □

**Compilation time and storage.** Write $m = |Q||J|$, $k = |\mathcal{K}|$, $g = |\Gamma|$, and $e = \sum_{\gamma,q,j} |\mathsf{Pop}(q, j, \gamma)|$. Let $T_{\mathrm{sum}}$ and $M_{\mathrm{sum}}$ be the time and peak memory of the monitor/saturation or equivalent exact local-summary computation; these depend on PDA size and total queried suffix length. They are not bounded by $k$ alone. A direct implementation uses $O(gm + e)$ summary storage and $O(k(gm + e))$ work for dense child evaluation, including expected $O(m)$ hashing/equality per child. Dense responses occupy $O(km)$ bits, push transitions $O(gk \log k)$ bits, and projected masks $|Q|k(V + 1)$ bits. Projection requires $O(|Q|kV)$ bit assignments. Deduplication can reduce physical mask storage while leaving intermediate responses and peak RSS substantial.

Concrete local simulation is an alternative only when its reachable configuration set is finite; epsilon-push cycles can defeat a visited-set enumeration. Finite saturation is the terminating specification above. Implementations that enumerate configurations must enforce suitable grammar restrictions or return an explicit resource failure. The finite outer worklist alone does not establish inner-loop termination.

## C Implementation Optimizations

The compilation and online algorithms in Sections 4.1 and 4.2 admit optimizations in table storage, repeated row assembly, and parser-state maintenance. These optimizations reduce work or storage for a fixed representation; they do not change the acceptance contract.

**Compile-time deduplication.** Content-addressed mask pools share one backing allocation between identical bitmasks. Hashing locates candidate matches; equality of the complete bit pattern is required before sharing. During row assembly, the set of contributing terminals and the resulting serialized mask can be memoized so repeated combinations require only one wide-bitvector OR. Such physical sharing must be distinguished from reducing the logical number of table entries. For vocabulary size $|\mathcal{T}|$, a dense table occupies $|Q|\,|\mathcal{K}|\,|\mathcal{T}|$ bits before alignment and metadata; logical table size, resident deduplicated masks, intermediate compiler storage, peak RSS, and runtime stack memory are different quantities.

**Compressed responses.** A response vector can be represented by a shared base and the positions that differ from it, provided reconstruction gives every response bit used by lookup or a future push transition. Signature interning compares these complete represented vectors, not equality on a provisional sample of query positions. Any elimination of query coordinates additionally requires closure under the local-pop dependency relation: omitted bits must be constant or reconstructible for

every transition. Without this invariant, equal compressed records need not identify interchangeable signatures.

**Compilation bookkeeping.** Base responses are computed along a suffix trie of the queries, so queries with a common prefix reuse the work for that prefix; uncertainty found before a trie leaf is propagated to every query below it, even if no concrete local configuration survives. Signature discovery interns child responses by exact vector equality, using hashing only to locate candidates. In the lexer/parser factorization, boundary and mid-terminal queries are collected before suffix closure, and final mask rows are assembled from precompiled token-group masks and cached on first use.

**Token bytes.** Token byte strings preserve partial UTF-8 fragments, which decoding a single token to text need not do. EOS and other special tokens follow explicit policies rather than being inferred from empty decoded strings.

**Online fast paths.** The native matcher sorts row keys to deduplicate them; cache lookup and cold row assembly add their own work. A preassembled row can be copied directly when one compiled lookup determines the output. With several active configurations, the output buffer is first cleared and each contributing row is ORed into it; identical row indices can be deduplicated for that mask without discarding distinct underlying stacks. Sparse representations store token indices for rows whose set bits make indexed writes cheaper than scanning the full bitmap. A sparse scatter sets the selected bits of the destination, whereas a full-row copy overwrites it; these operations are interchangeable only with the appropriate initialization and union semantics. Mask width and the number of participating rows remain relevant to the cost.

Backward reachability from each terminal DFA's accepting states identifies lexer states with no terminal completion. This permits constant-time rejection of dead lexer candidates, but lexer liveness alone does not prove that the surrounding parser configuration has an accepting continuation. Token bytes that cross terminal boundaries require the combined parser and lexer state, including active positions inside terminals.

**Engineering details.** Contiguous transition, DFA, and mask arrays improve locality. Word-aligned bitvector OR with a byte tail handles widths not divisible by the machine word size. C implementations replace selected Python loops, and cached foreign-function pointers avoid repeated argument marshalling. Reset can restore the initial epsilon closure from a snapshot. Persistent stack frames retain predecessor signatures so that a pop restores the appropriate earlier class even when the current class alone does not determine it.

# D ALGORITHMS

## D.1 OFFLINE COMPILATION

The algorithms below specify the byte-query construction with exact summaries. Shared tries and the terminal-trace factorization reuse its work without changing the required response semantics. The initial runtime stack may contain a bottom marker; its signature is obtained by applying the push DFA to that initial stack, rather than assuming it is the empty-stack class.

**Algorithm 1** INSTAGRAMMAR Offline Compilation

**Require:** Normalized PDA $\mathcal{P}$, vocabulary byte strings $w_1, \ldots, w_V$
**Ensure:** Mask table $B$, EOS table $E$, push transitions $\delta_{\mathcal{K}}$

$J \leftarrow \mathrm{suf}(\mathcal{T}) \cup \{\epsilon, \mathsf{eos}\}$
Initialize $\mathcal{R}_\epsilon[q, j] \leftarrow 0$ and $U_\gamma \leftarrow \emptyset$ for all indices
**for** $q \in Q$ **do**
    $(r, U^{(q)}) \leftarrow$ SUFFIXTRIENFA$(q, J, \mathcal{P})$
    $\mathcal{R}_\epsilon[q, \cdot] \leftarrow r$
    **for** $\gamma \in \Gamma$ **do**
        $U_\gamma \leftarrow U_\gamma \cup U_\gamma^{(q)}$
    **end for**
**end for**
**for** $\gamma \in \Gamma$ and $(q, j) \in U_\gamma$ **do**
    $(L[q, j, \gamma], \mathsf{Pop}[q, j, \gamma]) \leftarrow$ COMPUTEINFLUENCE$(q, j, \gamma, \mathcal{P})$
**end for**
$\mathcal{K} \leftarrow \{\kappa_0\}$ with response $\mathcal{R}_\epsilon$; work $\leftarrow [\kappa_0]$
**while** work $\neq \emptyset$ **do**
    Remove the first $\kappa$ from work
    **for** $\gamma \in \Gamma$ **do**
        $r' \leftarrow$ EVALCHILD$(\kappa, \gamma, \mathcal{R}_\epsilon, L, \mathsf{Pop}, U)$
        $(\kappa', \mathrm{new}) \leftarrow$ INTERNEXACT$(r', \mathcal{K})$
        $\delta_{\mathcal{K}}(\kappa, \gamma) \leftarrow \kappa'$
        **if** new **then**
            Append $\kappa'$ to work
        **end if**
    **end for**
**end while**
**for** $\kappa \in \mathcal{K}$ and $q \in Q$ **do**
    **for** $i = 1, \ldots, V$ **do**
        $B[q, \kappa][i] \leftarrow \mathcal{R}_\kappa(q, w_i)$
    **end for**
    $E[q, \kappa] \leftarrow \mathcal{R}_\kappa(q, \mathsf{eos})$
**end for**
**return** $B, E, \delta_{\mathcal{K}}$

### D.2 Online Mask Retrieval

**Algorithm 2** InstaGrammar Online Mask Retrieval

**Require:** Tables $B, E, \delta_{\mathcal{K}}$, active set $\mathcal{A}$, logits $\ell$
$K \leftarrow \textsc{Deduplicate}\{(q, \textsc{SigOf}(\hat{\sigma})) : (q, \hat{\sigma}) \in \mathcal{A}\}$
Initialize ordinary-token mask $mask \leftarrow 0$ and $e \leftarrow 0$
**for** $(q, \kappa) \in K$ **do**
    $mask \leftarrow mask \vee B[q, \kappa]$; $e \leftarrow e \vee E[q, \kappa]$
**end for**
Set the explicitly designated EOS token bits to $e$
**if** every allowed-token bit is zero **then**
    **return** constraint failure
**end if**
Set $\ell_i \leftarrow -\infty$ at every disallowed token ID
Sample $t$ from $\mathrm{softmax}(\ell)$
**if** $t$ is EOS **then**
    **return** completed output
**end if**
$\mathcal{A} \leftarrow \textsc{UpdateActiveSet}(\mathcal{P}, \mathcal{A}, w_t, \delta_{\mathcal{K}})$
**return** $t, \mathcal{A}$
**procedure** $\textsc{SigOf}(\hat{\sigma})$
    **if** $\hat{\sigma}$ is empty **then**
        **return** $\kappa_0$
    **else**
        $(\gamma, \kappa_{\mathrm{below}}) \leftarrow top(\hat{\sigma})$
        **return** $\delta_{\mathcal{K}}(\kappa_{\mathrm{below}}, \gamma)$
    **end if**
**end procedure**

## E Subroutines

**Query monitors and finite summaries.** For an ordinary query $s$, the input monitor reads exactly $s$ and then allows arbitrary byte transitions; epsilon moves never advance it. A query is successful only when the required prefix has been consumed and a final PDA control state is reached. For EOS, the monitor permits epsilon moves only. The local-summary routines use finite pushdown saturation on the product with this monitor, as in Lemma 6. Concrete enumeration is interchangeable only when its local configuration space is finite.

In a local stack, symbols pushed during the computation are distinct from the distinguished boundary. An absent-frame pop is recorded with its required symbol and monitor position; it is not followed. A first pop of the distinguished frame is instead a terminal report to a parent query. The monitor advances on a consuming pop before its residual query is recorded. Thus a pop during the unrestricted continuation produces the empty-suffix viability query, not EOS.

**Algorithm 3** SUFFIXTRIENFA: Base Response and Uncertainty for State $q$

**Require:** PDA $\mathcal{P}$, state $q$, query set $J$ with shared suffix trie
**Ensure:** $r[j] = \mathcal{R}_\epsilon(q, j)$ and uncertainty sets $U_\gamma^{(q)}$

```
Initialize r[j] ← 0 and U_γ^(q) ← ∅ for all indices
for j ∈ J do
    Construct the query monitor with an initially empty local stack
    Saturate the local product, recording blocked external-pop targets
    r[j] ← 1 iff a successful target is reachable without an external pop
    if r[j] = 0 then
        for each symbol γ with a reachable blocked-pop target do
            U_γ^(q) ← U_γ^(q) ∪ {(q, j)}
        end for
    end if
end for
return r, U^(q)
```

The per-query presentation makes propagation explicit. A shared-trie implementation must retain unresolved reports from every prefix for its descendant queries. An empty concrete configuration set does not justify pruning a subtree that has such reports. Uncertainty sets from different control states are combined by union in Algorithm 1. Empty-stack successes need no frame-specific resimulation because their witnessing runs never access the additional frame.

**Algorithm 4** COMPUTEINFLUENCE: First-Pop Summary

**Require:** PDA $\mathcal{P}$, control state $q$, query $j$, frame symbol $\gamma$
**Ensure:** Local flag $L(q, j, \gamma)$ and influence set $\mathsf{Pop}(q, j, \gamma)$

```
Tag the distinguished initial frame γ; initialize local stack above it to empty
Construct the finite query monitor for j
Replace the first tagged-frame pop by a terminal report (q', j')
Saturate the resulting local pushdown system from its initial configuration
L ← 1 iff a successful target is reachable without a tagged-frame pop
Pop ← {(q', j') : its reporting target is reachable}
return L, Pop
```

**Algorithm 5** EVALCHILD: Response under a Push

**Require:** Parent response $\mathcal{R}_\kappa$, symbol $\gamma$, base $\mathcal{R}_\epsilon$, summaries $L, \mathsf{Pop}, U$

```
r' ← a copy of R_ε
for (q, j) ∈ U_γ do
    r'[q, j] ← L[q, j, γ] ∨ ⋁_{(q', j') ∈ Pop[q, j, γ]} R_κ(q', j')
end for
return r'
```

INTERNEXACT inserts the response if and only if no equal vector exists, returning both its class and a new-entry flag. Hash collisions are resolved by full equality. A compressed representation must recover all response coordinates used by masks and influence sets; equality only on the current whole-token mask is insufficient for later pushes.

**Algorithm 6** INSTAGRAMMAR Online Active Set Update

**Require:** PDA $\mathcal{P}$, active set $\mathcal{A}$, sampled ordinary-token bytes $b_1 \cdots b_n$, push DFA $\delta_{\mathcal{K}}$

```
C ← EPSCLOSURE(A)
for i = 1, ..., n do
    D ← ∅
    for (q, σ̂) ∈ C and each enabled transition labeled b_i do
        Apply its stack operation, obtaining (q', σ̂')
        D ← D ∪ {(q', σ̂')}
    end for
    C ← EPSCLOSURE(D)
end for
return C
```

Both consuming transitions and epsilon closure apply the same stack rules: internal moves preserve the stack; a push of $\gamma$ appends $(\gamma, \textsc{SigOf}(\hat{\sigma}))$; an enabled matching pop removes the top frame. Closure retains all distinct full configurations, deduplicating exact state/stack equality, not merely row keys. Concrete closure requires a finite reachable closure or an explicit resource failure. Unrestricted epsilon-push cycles require a symbolic representation and cannot be handled by an unbounded concrete worklist.

**Terminal-trace implementation.** For a completed-terminal sequence $u$, a boundary query $(u, B)$ asks whether the parser can consume $u$; a mid query $(u, M_t)$ additionally asks whether it can start terminal $t$; an EOS query asks for an epsilon path to an accepting control state. The lexer has already checked that the final partial-terminal DFA state is live. The query universe includes all boundary and mid-terminal token classifications and their terminal-sequence suffixes. On first popping a frame, the residual terminal query is recorded just as above. A pop that completes a boundary query, or starts its final partial terminal, is already a stack-independent success and contributes to the local constant. These response predicates support the same compositional fixed point. Their use for complete-document masking requires that surviving reachable parser continuations are productive; DFA liveness alone establishes only terminal completion. The implementation's normalized frontend and resource limits are separate from the finite saturation specification.

# F JSONSCHEMABENCH

Table 7 lists the 11,306-input suite used by Table 2 and Appendix A.3: the 10,646 JSONSchemaBench schemas across twelve datasets, the 100 JSON-mode-eval schemas (NousResearch, 2024), and 560 handwritten and synthesized cases. The five GitHub complexity splits are separate datasets in this count. BFCL (Patil et al., 2025) is one dataset here but seven language/task families in the latency aggregation.

The six-engine timing archive uses 7,029 of these inputs, selected by the keyword filter described in Appendix A.1, of which 6,976 belong to the canonical timing-family selection. Exact outcome denominators for each experiment appear in Tables 3 and 4.

Schemas without instances cannot establish empirical acceptance or rejection accuracy. The 1,511 inputs without instances remain part of the suite but contribute no instance-validation result; 438 of them are handwritten or synthesized cases. BFCL and JSON-mode-eval supply only positive instances (1,043 and 100), so success on them cannot measure false acceptance. These inputs are reported as untested, rather than silently counted as correct.

## F.1 JSON SCHEMA CONVERSION SCOPE

Table 8 describes the schema-native converter used in our experiments. It accepts a mixed subset of JSON Schema keywords, including `definitions` and `$defs`, without selecting validation semantics from `$schema`. We therefore make no complete draft-07 or 2020-12 conformance claim. The statuses describe source-level conversion policies, not independently validated behavior for every keyword interaction. Correct masks for the converted grammar do not establish conformance to constraints omitted or approximated by that conversion.

Table 7: Inventory of the 11,306-input suite, grouped into fifteen datasets. “Valid” and “Invalid” are aggregate labeled instance counts within a dataset, not counts per schema. “No tests” counts inputs without instances.

| Dataset | Inputs | Valid inst. | Invalid inst. | No tests |
|---|---|---|---|---|
| Github_trivial | 444 | 460 | 771 | 79 |
| Github_easy | 1,943 | 2,641 | 4,611 | 154 |
| Github_medium | 1,976 | 3,091 | 6,119 | 153 |
| Github_hard | 1,240 | 1,493 | 3,405 | 352 |
| Github_ultra | 164 | 160 | 302 | 69 |
| Glaiveai2K | 1,707 | 1,634 | 1,104 | 73 |
| Kubernetes | 1,064 | 1,680 | 2,908 | 50 |
| Snowplow | 403 | 670 | 1,730 | 9 |
| WashingtonPost | 125 | 146 | 330 | 25 |
| MCPspec | 45 | 44 | 44 | 1 |
| JsonSchemaStore | 492 | 679 | 1,405 | 108 |
| BFCL | 1,043 | 1,043 | 0 | 0 |
| JSON-mode-eval | 100 | 100 | 0 | 0 |
| Handwritten | 147 | 14 | 39 | 134 |
| Synthesized | 413 | 109 | 279 | 304 |
| **Total** | **11,306** | **13,964** | **23,047** | **1,511** |

The converter also has serialization-level restrictions: equivalent JSON spellings need not yield equivalent accepted byte strings, unknown object keys may repeat, and the documented string terminals do not validate UTF-8 well-formedness or surrogate pairing. These are frontend limitations, distinct from the expressiveness of context-free grammars.

Table 8: Selected JSON Schema conversion policies. I: a dedicated implementation exists under the stated conditions; P: partial or approximated semantics; U: the constraint is not enforced. These categories are not conformance-test results.

| Keyword or feature | Status | Implemented policy and limitation |
|---|---|---|
| `type` | P | Dispatches the seven JSON types and type unions. Integers use integer-form literals; absent type with `properties`/`required` is inferred as object. |
| `properties`, `required` | P | Recursive value constraints; declared keys appear at most once and in declared order, as in llguidance and XGrammar. Optional keys may be skipped and required keys may not, for objects of any size. Required keys absent from `properties` are enforced as additional known keys. Valid instances with keys in another order are rejected. |
| `additionalProperties` | P | Handles false, true, and dictionary-schema values. Alternate escaped spellings of declared keys are not normalized and can evade their value constraints. |
| `patternProperties` | P | Uses the alternative value schemas for unknown keys; key regexes and overlap intersections are not enforced. |
| `items` | I | Applies a single dictionary schema to every element. Tuple-form `items` and `prefixItems` are unsupported. |
| `minItems`, `maxItems` | I | Builds count constraints for nonnegative integer bounds up to a configurable limit (default 10,000); larger or inconsistent bounds fail compilation. |
| `enum`, `const` | P | Admits compact serialized literals with fixed object-key order; sibling constraints are ignored. |
| `$ref`, `$defs`, `definitions` | P | Root definition references are resolved. Recursive and unresolved references fall back to any JSON value; sibling handling is a field merge. |
| `anyOf`, `oneOf`, `allOf` | P | Alternatives use union with merged siblings; `oneOf` exclusivity is lost. `allOf` uses a field merge rather than general intersection. |
| `pattern` | P | A byte-regex subset is applied to whole string content. Some repetitions are approximated; unsupported patterns are dropped. |
| `minLength`, `maxLength` | P | Byte-oriented counting with escapes counted as one unit. A maximum above 256 is dropped; a minimum-only constraint imposes a maximum of 256. |
| Numeric bounds, `multipleOf`, `uniqueItems`, `contains` | U | These constraints are ignored. |
| `not`, conditionals, dependencies, `unevaluated*`, `format` | U | No enforcement of these keyword constraints. Boolean-schema false is also treated as unconstrained. |

## G Detailed PDA Example

The grammar is `expr ::= "a" | "(" expr ")"` and the input is `((a))`. Figure 3 uses four progress states and a dedicated accepting state:

$$Q = \{q_0, q_L, q_1, q_R, q_f\}, \qquad \Gamma = \{\bot, \gamma_p\}, \qquad F = \{q_f\}.$$

State $q_0$ dispatches a new expression; $q_L$ pushes its pending closer; $q_1$ follows a completed subexpression; $q_R$ consumes a closer after its frame is popped. State $q_1$ alone does not authorize EOS. The transitions are

$$q_0 \xrightarrow{(} q_L, \qquad q_L \xrightarrow{\epsilon,\ \text{push } \gamma_p} q_0, \qquad q_0 \xrightarrow{\mathtt{a}} q_1,$$
$$q_1 \xrightarrow{\epsilon,\ \text{pop } \gamma_p} q_R, \qquad q_R \xrightarrow{)} q_1, \qquad q_1 \xrightarrow{\epsilon,\ \text{pop } \bot} q_f.$$

Unannotated edges leave the stack unchanged; pops require the indicated top symbol. The initial configuration is $(q_0, \bot)$. The protected bottom marker is never pushed by an input transition, and its pop is the unique route to $q_f$. Thus any reachable accepting configuration has finished all pending closers.

**Step-by-step trace.** The stack is written top first. Epsilon moves are shown separately so that a token spanning several bytes can include multiple stack operations.

| Step | Action | State | Stack |
|---|---|---|---|
| 0 | start | $q_0$ | $\bot$ |
| 1 | consume `(` | $q_L$ | $\bot$ |
| 2 | push $\gamma_p$ | $q_0$ | $\gamma_p\bot$ |
| 3 | consume `(` | $q_L$ | $\gamma_p\bot$ |
| 4 | push $\gamma_p$ | $q_0$ | $\gamma_p^2\bot$ |
| 5 | consume `a` | $q_1$ | $\gamma_p^2\bot$ |
| 6 | pop $\gamma_p$ | $q_R$ | $\gamma_p\bot$ |
| 7 | consume `)` | $q_1$ | $\gamma_p\bot$ |
| 8 | pop $\gamma_p$ | $q_R$ | $\bot$ |
| 9 | consume `)` | $q_1$ | $\bot$ |
| 10 | pop $\bot$ | $q_f$ | $\epsilon$ |

After step 5, EOS is invalid because two closers remain. After step 9, epsilon closure can reach $q_f$, so EOS is valid. The token `))` can perform steps 6–9 in one token advancement.

**Masks and signatures.** Use the byte-fallback vocabulary of Section 3, with the additional tokens `(a`, `a)`, `((`, and `))`. In the displayed six-coordinate order `a`, `(`, `(a`, `a)`, `((`, `))`, the masks are:

| State | Depth $d$ in $\gamma_p^d\bot$ | Ordinary-token coordinates | EOS |
|---|---|---|---|
| $q_0$ | 0 | 111010 | 0 |
| $q_0$ | $\geq 1$ | 111110 | 0 |
| $q_L$ | any | 111110 | 0 |
| $q_1$ | 0 | 000000 | 1 |
| $q_1$ | 1 | 000000 | 0 |
| $q_1$ | $\geq 2$ | 000001 | 0 |
| $q_R$ | 0 | 000000 | 0 |
| $q_R$ | $\geq 1$ | 000001 | 0 |

The omitted single-byte token `)` is enabled in $q_1$ at positive depth and in $q_R$ at every displayed depth. Other omitted byte tokens are invalid for this grammar. The final state $q_f$ permits EOS and no ordinary token. Empty-suffix viability is true at all displayed configurations, including $(q_1, \gamma_p^d\bot)$ for positive $d$, because closers can still be generated; this differs from EOS.

The response queries include every suffix of these tokens. Their masks and EOS distinguish $d = 0$, $d = 1$, and $d \geq 2$, and no query distinguishes greater depths. Thus there are three response classes on the family $\{\gamma_p^d\bot : d \geq 0\}$. The full compiler also starts from the empty stack and considers arbitrary push strings, so this illustration does not assert a global three-class compilation result. Pushes of

$\gamma_p$ preserve the displayed equivalence. Pops do not: depths two and three share a class but lead to different predecessor classes. Recording the signature below each pushed frame provides the correct restoration without a stack traversal.

## H Comparison with Prior Work

Table 9: Preprocessing and online work in related GCD systems. These are algorithmic descriptions from the cited versions, not matched performance measurements. Grammar-front-end support is separate from parser correctness.

| System | Parsing/tokenization model | Precomputed object | Online work and scope |
|---|---|---|---|
| PSC (Li et al., 2026b) | Lexing transducer; terminating deterministic PDA | Stack-classifier FSA and token masks | Traverse the current stack to select a mask; advance lexer/parser. The cited construction assumes termination and determinism. |
| GreatGramma (Park et al., 2025) | Regex lexer transducer and parser PDA | Lexer-state-dependent inverse token-spanner table; parser classifications | Check unresolved terminal sequences against the parser stack and combine token sets. The presentation assumes a lexer with one-character lookahead. |
| SynCode (Ugare et al., 2025) | Regex DFAs; incremental LR(1)/LALR(1) parser | DFA mask store indexed by lexical state and terminal sequences | Compute accept sequences/remainder and union stored masks. Precision depends on lookahead; the implementation uses short accept sequences. |
| XGrammar-2 (Li et al., 2026a) | Grammar FSMs; Earley parser in the cited version | Adaptive masks with cross-grammar reuse; optional just-in-time construction | Retrieve scannable-state caches and check context-dependent tokens using Earley context; cache misses may compile during generation. |
| llguidance (guidance-ai, 2026) | Earley parser; regex-derivative lexer | Compiled grammar/lexer structures and vocabulary trie | Traverse the token trie with lexer/parser checks. Runtime work and supported schema conversion depend on the implementation and input grammar. |