FROM GRAMMAR DELTAS TO TOKEN MASKS: INCREMENTAL CONSTRAINED DECODING UNDER RUNTIME GRAMMAR EVOLUTION

Anonymous authors
Paper under double-blind review

ABSTRACT

Grammar-constrained decoding enables large language models to generate outputs that satisfy formal constraints, but in code and domain-specific language generation, the relevant constraints are not always fully available before decoding and may emerge progressively from new declarations, inferred type information, or runtime analysis feedback. Existing structured-decoding engines efficiently handle fixed, predeclared, or previously seen constraint structures; however, when a first-seen grammar update changes the parser state induced by an already committed prefix, the straightforward solution reconstructs the constraint state under the updated grammar, repeating parsing work already performed over the prefix. We introduce **GramDelta**, an incremental framework for constrained decoding with append-only runtime grammar updates. GramDelta maintains parse state over the committed prefix and propagates only changes induced by newly added grammar rules through historical dependencies, avoiding prefix replay. It further incrementally maintains next token validity from these parser state changes, reusing an unchanged mask or rechecking only affected tokenizer-trie regions when the impact is localized. GramDelta therefore preserves the result of full reconstruction while making update cost sensitive to the actual impact of a grammar change. Experiments across four model configurations show that GramDelta reduces median first-seen grammar-update latency by 1.33–$1.69\times$ relative to XGrammar and reduces the additional TPOT overhead over unconstrained serving by 83.9–87.3%. These results enable efficient constrained decoding when constraints progressively emerge and evolve with the generated prefix.

1 INTRODUCTION

Grammar-constrained decoding Geng et al. (2023); Scholak et al. (2021); Dong et al. (2025); Hua et al. (2026) enables large language models (LLMs) to reliably generate structured outputs that satisfy formal constraints by restricting invalid tokens during decoding. However, in code and domain-specific language generation, the effective constraints are not always fully determined before decoding begins and may instead emerge progressively as generation proceeds Li et al. (2026). A newly generated declaration can introduce identifiers and scopes that did not previously exist Zhang et al. (2026); type inference can further restrict admissible expressions Mündler et al. (2025); and static or runtime analyses Li et al. (2025) can determine which fields, APIs, or program actions are valid for the current partial program. The common feature of these settings is that the information determining valid continuations becomes available only as the generation context is established.

Recent work supports such runtime-dependent constraints from several directions. Structured-generation engines can dispatch among dynamically selected grammar components and reuse computation across related constraint structures (Li et al., 2026), while code-generation methods can instantiate or derive constraints from an evolving program environment (Mündler et al., 2025; Zhang et al., 2026). These systems show that constrained decoding need not operate over a language whose effective structure is fully fixed before generation.

However, they primarily address how to select, construct, or apply the constraint needed at runtime, rather than how to incrementally update a general grammar-constrained decoder that has already

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.

processed the current prefix when new grammar structure is introduced for the first time. At that point, the decoder already holds parsing state and next-token validity information computed under the previous grammar, while the newly introduced structure may affect part of this existing computation.

Concretely, at decoding step t, the constrained decoder has already processed a committed prefix $x_{<t}$ under the current grammar G_t, maintaining the current parsing state and the set of tokens admissible next. Suppose an additive grammar update ΔG_t is then revealed for the first time, extending the grammar to $G_t \cup \Delta G_t$. The generated tokens remain unchanged, but the decoder must obtain the same constraint state that would result from processing the same prefix under the updated grammar. The straightforward approach is to reconstruct the relevant parsing state under the new grammar, reprocess the existing prefix, and recompute next-token validity. However, adding grammar structure does not invalidate previously established parsing results wholesale: the new rules introduce additional parsing consequences only through relevant dependencies, and only a subset of these consequences can affect the current decoding position. Full reconstruction therefore repeats substantial computation that is unaffected by the grammar update.

To avoid this redundant work, we propose **GramDelta**, an incremental constrained-decoding framework for additive runtime grammar updates. GramDelta retains the Earley state for the committed prefix. When a new production arrives, it seeds that production at the historical waiters that can activate it, propagates only newly enabled deductions, and repairs the token mask only where current validity may change. These parser changes are then projected to next-token validity: unaffected masks are reused, localized effects require rechecking only relevant tokenizer-trie regions, and broad effects retain a full projection path.

GramDelta addresses this problem through three components:

- We characterize the historical entry points of additive grammar updates and prove that charts preceding the earliest entry remain unchanged, enabling direct update seeding without replaying the generated prefix.
- We derive semi-naive delta rules for Earley Predict, Scan, and Complete that propagate only computation involving new facts, and prove that their fixed point exactly matches full reconstruction under the updated grammar.
- We project only parser changes that can affect the current decoding boundary to token validity, reusing unaffected masks, re-evaluating localized tokenizer-trie regions, and retaining a full path for broad effects.

We evaluate GramDelta across multiple model families, grammar structures, and runtime-update workloads. GramDelta preserves parser-state and token-validity equivalence to full reconstruction in all evaluated cases, while reducing first-seen grammar-update latency by 1.33–$1.69\times$ and the additional end-to-end decoding overhead of runtime grammar evolution by $\mathbf{83.9} - \mathbf{87.3}\%$.

2 Background and Related Work

2.1 Grammar-Constrained Decoding

Grammar-constrained decoding restricts the candidate tokens at each generation step so that the generated output remains compatible with a formal language Willard & Louf (2023); Koo et al. (2024). Given a grammar G, the byte sequence $x_{<t}$ induced by the current generated prefix, and tokenizer vocabulary $\mathcal{V}$, let $b(v)$ denote the byte sequence of token v. The token mask $M_t \in \{0,1\}^{|\mathcal{V}|}$ is defined as

$$M_t[v] = 1 \iff \exists y, \quad x_{<t} b(v) y \in L(G)$$

The central runtime task is therefore to determine valid next tokens from the current prefix. Existing systems accelerate this operation through different combinations of automata or parser state, tokenizer-aware preprocessing, trie traversal, caching, and grammar compilation. For example, automata-based methods precompute tokenizer–language compatibility, XGrammar compiles grammar structure for efficient runtime mask generation Dong et al. (2025); Li et al. (2026), Pre^3 Chen et al. (2025) exploits deterministic pushdown transitions for LR(1) grammars, and Earley-based

systems such as Formatron Sun et al. (2025) and llguidance Guidance AI (2025) construct token validity from incrementally advanced parser state . These approaches substantially reduce constrained-decoding overhead when the relevant grammar is fixed or already available.

We instead study what happens when new grammar structure becomes available only after the current prefix has already been processed under an earlier grammar.

2.2 Runtime Grammar Evolution and State Maintenance

Recent constrained-decoding systems support dynamic structure at several levels. Llguidance uses an Earley recognizer together with a derivative-based lexer and tokenizer-trie traversal to compute token masks for a supplied grammar (Guidance AI, 2025). XGrammar-2 targets dynamic agentic workloads: TagDispatch switches among grammar components during generation, while cross-grammar caching, adaptive Earley token-mask caching, and just-in-time compilation reuse work across related structures (Li et al., 2026). Decode-Time Grammars provide a complementary form of dynamism by instantiating grammar fragments from an evolving runtime environment Γ, allowing the constraint applied to later regions to depend on declarations already generated (Zhang et al., 2026). Other code-generation methods similarly derive context-sensitive constraints from type, scope, or program state (Mündler et al., 2025; Li et al., 2025). Collectively, these methods make generation-time constraints more efficient to construct, select, or execute.

A different dimension of change arises when grammar delta is introduced only after the prefix $x_{<t}$ has already been processed under the current grammar G. Such an update can change not only the constraint to be executed subsequently, but also the parsing state induced by the *same prefix* under $G \cup \Delta G$. Existing reuse mechanisms do not directly maintain this historical state across grammar versions: XGrammar2 reuses grammar substructures and token-mask caches, Llguidance advances its recognizer over committed tokens under a supplied grammar, and Decode-Time Grammars instantiate fragments for the corresponding later generation regions. Under such a late grammar update, obtaining the state equivalent to processing the same prefix under the updated grammar therefore still requires re-establishing its parsing computation, repeating work already performed.

Parsing research provides closely related forms of incremental reuse. Incremental parsing reuses parsing results across input changes (Wagner & Graham, 1998); Reflective Grammars introduce scoped syntax extensions during parsing (Stansifer & Wand, 2011); and DEXTER incrementally updates an LR goto graph as productions are added or removed (Cazzola & Vacchi, 2014). These approaches incrementally maintain either input-dependent parsing results or grammar-dependent parser structure, but not prefix-dependent historical state that must be revised by a later production addition while the input itself remains fixed. They therefore do not directly eliminate the historical recomputation induced by the late-update setting above.

3 GramDelta

Given a current grammar G, a committed prefix $x_{<t}$, and the parsing state already materialized for that prefix, an additive grammar update ΔG may become available at runtime. Our objective is to obtain the parser state and next-token mask induced by the updated grammar

$$G' = G \cup \Delta G$$

without changing the committed prefix.

Figure 1 contrasts two ways of reaching this same target. The reference reconstruction path builds the updated recognizer under G', replays the committed prefix from the beginning, and then projects the reconstructed parser state to the next-token mask. GramDelta instead reuses the materialized historical state and updates it in three stages.

First, *Historical Delta Localization* queries the persistent historical waiting dependencies to identify the positions at which newly added productions can enter the old Earley closure. The earliest such position j_Δ gives a left boundary on historical change, while all matching historical waiters contribute to the seed set S_Δ. Second, *Delta Earley Closure* propagates only newly enabled Predict, Scan, and Complete deductions through an event-driven worklist, producing the updated Earley state without replaying the prefix. Third, *Token-Mask Repair* projects the updated frontier through the to-

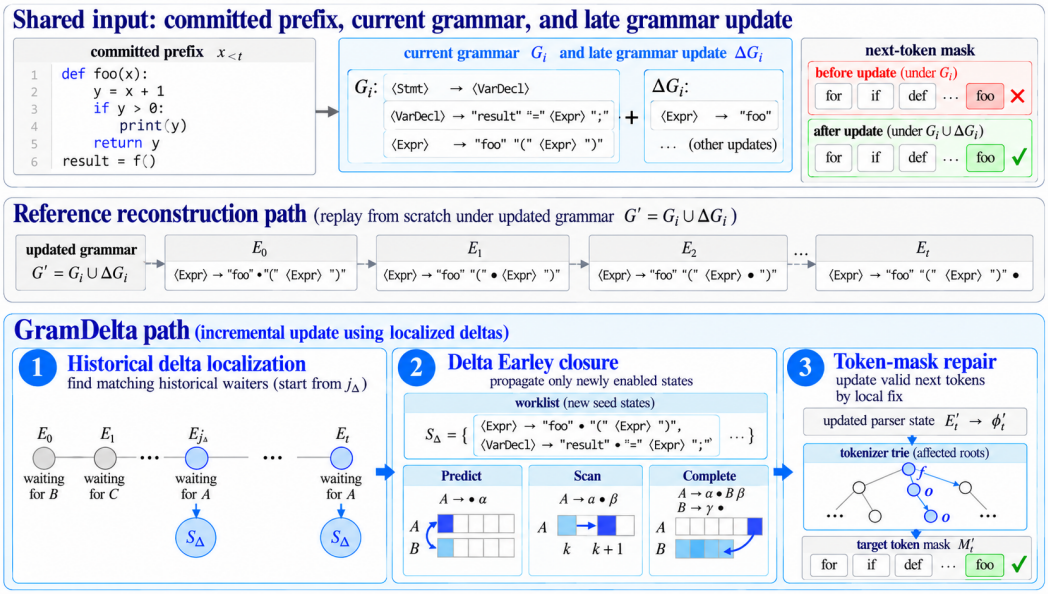


Figure 1: Overview of runtime grammar evolution and GramDelta.

kenizer trie and re-evaluates only affected regions when the change is localized, while retaining a full-projection fallback when necessary.

3.1 Runtime Update Model

Let G denote the current context-free grammar and $x_{<t}$ the byte sequence induced by the committed generation prefix. GramDelta retains the historical Earley state

$$E_{0:t} = (E_0, \ldots, E_t)$$

where an item

$$[A \to \alpha \bullet \beta, o]_k \in E_k$$

states that production $A \to \alpha\beta$ started at position o and that α has matched $x_{o:k}$. We use the standard Predict, Scan, and Complete deductions Earley (1970).

A previously unseen additive grammar update ΔG may become available only after $x_{<t}$ has been committed. We consider production additions without deleting or modifying existing productions

$$G' = G \cup \Delta G \qquad x_{<t} \text{ remains fixed}$$

The post-update target is

$$E'_{0:t} = \mathrm{Earley}(G', x_{<t})$$

together with the next-token mask M'_t induced by this updated state.

Because Earley deduction is monotone under production addition

$$E_k \subseteq E'_k \qquad \Delta E_k = E'_k \setminus E_k$$

GramDelta treats runtime grammar evolution as incremental maintenance of the already materialized fixed point: it computes only the new facts $\Delta E_{0:t}$ required to obtain $E'_{0:t}$, while preserving the committed prefix and existing historical state.

3.2 Historical Delta Localization

When an update arrives, the old Earley closure has already been materialized. GramDelta first determines where a newly added production can enter this historical computation.

Let ΔG denote the newly introduced productions. For a nonterminal A, define its historical waiting positions

$$\mathcal{W}_E(A) = \{k \mid \exists [B \to \beta \bullet A\eta, o]_k \in E_k\}$$

For a new production $p : A \to \gamma$, the corresponding delta seeds are

$$\mathrm{Seed}(p, E) = \{[A \to \bullet\gamma, k]_k \mid k \in \mathcal{W}_E(A),\ [A \to \bullet\gamma, k]_k \notin E_k\}$$

and

$$S_\Delta = \bigcup_{p \in \Delta G} \mathrm{Seed}(p, E)$$

S_Δ is exact rather than heuristic: it characterizes the entry points through which newly added productions can first leave the old Earley closure.

Proposition 1 (First-new-fact characterization). Every

$$e \in E'_{0:t} \setminus E_{0:t}$$

admits an Earley derivation whose first fact outside the old closure belongs to S_Δ. Consequently

$$S_\Delta = \varnothing \quad \Longrightarrow \quad E'_{0:t} = E_{0:t}$$

The proof and the corresponding historical left-boundary property are given in Appendix A.2.

To avoid scanning historical charts when an update arrives, GramDelta maintains a nonterminal-indexed relation $\texttt{HistWait}[A]$ during normal decoding. Whenever position k contains an item waiting for A, k is registered in this relation. A newly introduced production $A \to \gamma$ can therefore instantiate its seeds directly from $\texttt{HistWait}[A]$.

The retained dependency is semantic rather than representational: historical state may be compacted or sparsified, but a dependency that can be reactivated by a future grammar addition cannot be irreversibly discarded.

3.3 Incremental Earley Closure

The seed set S_Δ identifies where a grammar update first enters the historical computation. GramDelta then treats $E_{0:t}$ as a materialized Earley fixed point Pereira & Warren (1983) and propagates only deductions newly enabled by these seeds Bancilhon & Ramakrishnan (1986).

Factorized Predict. Let U encode active nonterminal waits

$$U(k, A) = 1 \iff E_k \text{ contains an item waiting for } A$$

and let V map nonterminals to production-entry states

$$V(A, q_p^0) = 1 \iff q_p^0 \text{ is the entry state of } p : A \to \gamma$$

Predict can then be written as the factorized relation

$$\mathsf{P} = UV$$

A grammar update extends V by ΔV, while incremental propagation may introduce additional waits ΔU. The newly required Predict work is therefore

$$\Delta\mathsf{P} = U\Delta V \cup \Delta U V' \qquad V' = V \cup \Delta V$$

The first term is exactly the historical seed relation from Section 3.2; the second propagates waits created during the incremental closure.

Sparse Scan. Let $\mathsf{S}_a(q, q^+)$ denote a terminal transition from dotted state q to its successor q^+ after consuming terminal or lexical symbol a. Let ΔT_k be the sparse relation over dotted states and origins defined by

$$\Delta T_k(q, o) = 1 \iff (q, o, k) \in \Delta E_k \text{ and } q \text{ expects a terminal or lexical transition}$$

Because the committed input is unchanged, terminal-expecting facts already present before the grammar update have already been scanned. Only ΔT_k therefore requires new Scan work

$$\Delta E_{k+1}^{\mathrm{scan}} = \Delta T_k \, \mathsf{S}_{x_k}$$

Algorithm 1 GramDelta recognizer update

Require: Persistent state $\mathcal{R}$, committed prefix $x_{<t}$, new productions ΔG
Ensure: Updated Earley state

for all $p : A \to \gamma \in \Delta G$ **do**
  APPENDPRODUCTION(p)
  for all $k \in \texttt{HistWait}[A]$ **do**
    INSERT($[A \to \bullet\gamma, k]_k$)
  end for
end for
while WORKLISTNONEMPTY **do**
  $e \leftarrow$ POPWORKLIST
  if e waits for nonterminal B at position k **then**
    register k in $\texttt{HistWait}[B]$ and e in $\texttt{WaitAdv}[(k, B)]$
    INSERT(all Predict successors for B)
    INSERT(all joins with $\texttt{CompletedByStart}[(k, B)]$)
  else if e waits for terminal τ at $k < t$ **then**
    if MATCHES(τ, x_k) **then**
      INSERT(the scanned successor of e)
    end if
  else if e completes B from o **then**
    register e in $\texttt{CompletedByStart}[(o, B)]$
    join e with $\texttt{WaitAdv}[(o, B)]$
  end if
end while

Origin-sensitive Complete. Complete joins a waiter for B at position j with a completed B-constituent that starts at j. GramDelta maintains $\texttt{WaitAdv}[(j, B)]$ and $\texttt{CompletedByStart}[(j, B)]$ for the two sides of this join. A newly materialized waiter queries existing completions, while a newly materialized completion queries existing waiters. Whichever side arrives second therefore triggers the missing deduction without rescanning historical charts.

Incremental fixed point. Every candidate fact passes chart-membership deduplication. An unseen fact is inserted once, updates the corresponding sparse dependency relations, and may trigger further Predict, Scan, or Complete consequences. Propagation terminates when the worklist becomes empty.

The three delta operators above are combined in the event-driven procedure in Algorithm 1. A grammar update first inserts the exact historical seeds identified in Section 3.2. Thereafter, every newly inserted fact triggers only the Predict, Scan, and Complete consequences for which it supplies a previously missing premise. Membership deduplication ensures that each new Earley fact is processed at most once by the worklist.

The two Complete directions in Algorithm 1 are essential: a new waiter must join previously materialized completions, while a new completion must join previously materialized waiters. Thus the procedure performs dependency-driven propagation from S_Δ rather than replaying the prefix from an affected position.

Theorem 1 (Exact incremental closure). If

$$E_{0:t} = \mathrm{Earley}(G, x_{<t})$$

before the update and $G' = G \cup \Delta G$, then the state obtained at termination satisfies

$$\widehat{E}_{0:t} = \mathrm{Earley}(G', x_{<t}) = E'_{0:t}$$

The complete delta derivation, proof, and event-driven procedure are given in Appendices A.3 and B.1.

3.4 Exact Incremental Token Projection

Incremental closure recovers the updated Earley state $E'_{0:t}$, but only the state relevant to the current generation boundary can affect the next token. Let

$$F_t(E) = \{e \in E_t \mid e \text{ next expects a terminal or lexical transition}\}$$

denote the scannable parser frontier. Token projection may additionally depend on whether the current prefix is accepting and on lexical continuation state maintained by the projector. We therefore define the projection-relevant state

$$\Phi_t(E) = \big(F_t(E), a_t(E), \ell_t(E)\big)$$

where a_t denotes acceptance and ℓ_t denotes any auxiliary lexical state required by token projection.

If the grammar update leaves this state unchanged

$$\Phi_t(E') = \Phi_t(E)$$

then next-token validity is unchanged and the existing mask M_t is reused.

Otherwise, GramDelta identifies the tokenizer-trie regions whose validity may have changed. Let

$$C_\Delta \subseteq \mathcal{V}$$

denote the tokens covered by these regions. The affected set is conservative:

$$v \notin C_\Delta \quad \Longrightarrow \quad \mathrm{Valid}(\Phi'_t, v) = \mathrm{Valid}(\Phi_t, v)$$

where $\Phi'_t = \Phi_t(E')$. Exact projection is therefore repeated only for the affected trie regions, while all other mask entries are reused.

Because the grammar update is additive

$$L(G) \subseteq L(G \cup \Delta G)$$

and for the fixed committed prefix token validity is monotone

$$M_t \subseteq M'_t$$

Define

$$\Pi(\Phi'_t, C_\Delta) = \{v \in C_\Delta \mid \mathrm{Valid}(\Phi'_t, v)\}$$

The updated mask is

$$M'_t = M_t \cup \Pi(\Phi'_t, C_\Delta)$$

GramDelta consequently has three projection paths: reuse the existing mask when Φ_t is unchanged, repair only affected trie regions when the change is localized, and invoke full projection when a safe local affected set cannot be established. The full path uses the same projector and therefore changes performance, not semantics.

Proposition 2 (Exact incremental projection). If C_Δ covers every token whose validity may change under $\Phi_t \to \Phi'_t$, the repaired mask is identical to full projection from the updated state.

The construction, implementation details, and proof are given in Appendices A.4 and B.2.

4 Experiments

We evaluate whether GramDelta reduces the cost of first-seen runtime grammar evolution and whether this reduction translates to lower overhead in complete serving requests. All methods are evaluated within the same SGLang Zheng et al. (2024) serving stack under a common late-update protocol.

4.1 Experimental Setup

Baselines. We compare GramDelta against XGrammar Li et al. (2026), which are optimized constrained-decoding systems. Under our first-seen late-update protocol, a newly revealed grammar version is handled by instantiating the corresponding XGrammar constraint backend and reprocessing the already committed prefix. GramDelta instead applies the grammar delta directly to the persistent request-local recognizer state. This reconstruct-and-replay path describes how the baselines are exercised under our protocol rather than a general characterization of their systems.

Table 1: Main serving results on Qwen3-8B with BFCL-derived growing tool/schema traces. Update Latency is the median wall-clock latency from a first-seen grammar update until the updated token mask becomes available.

Method	Update Lat. (ms) ↓	TPOT (ms/tok) ↓	Overhead (%) ↓
Unconstrained	–	18.455	0.0
XGrammar	9.059	193.081	946.2
GramDelta	**5.351**	**40.577**	**119.9**

Models and workloads. We evaluate Qwen3-8B Yang et al. (2025), Qwen3-32B, Granite-4.1-3B IBM Research (2026), and Granite-4.1-8B, covering multiple model scales and tokenizer families. The benchmark contains two workload classes. Controlled dynamic traces exercise recursive dependencies, subgrammar extension, and declaration/type-dependent grammar extension. We additionally derive growing tool/schema traces from BFCL tool-calling schemas Patil et al. to provide realistic nested constraint structures. Each grammar delta becomes visible only at its designated runtime position; future grammar fragments are never preloaded. All models are executed in BF16 on a single NVIDIA RTX PRO 6000 Blackwell GPU. Further workload and system details are provided in Appendix C.

Measurement protocol. All measurements are collected within the same SGLang serving stack, but Update Latency and TPOT (Time Per Output Token) use different timing scopes. For a first-seen grammar update, *Update Latency* is the wall-clock interval from the moment ΔG_i becomes visible to the constraint backend until the updated token mask is available for the next decoding step. This interval excludes model forward execution, but includes all constraint-side work required by the update: backend construction or incremental state maintenance, committed-prefix reprocessing when required, and token projection.

TPOT (Time Per Output Token) is measured over complete serving requests and therefore includes model execution, steady-state constrained decoding, and runtime-update handling. We report TPOT Overhead relative to unconstrained serving

$$\text{TPOT Overhead} = \frac{\text{TPOT}_{\text{method}} - \text{TPOT}_{\text{unconstrained}}}{\text{TPOT}_{\text{unconstrained}}} \times 100\%$$

Headline serving comparisons use a shared-token protocol so that all constrained backends follow the same committed token sequence and update chronology. This removes backend-specific tokenization policy from the TPOT comparison; the corresponding token-policy details are given in Appendix C.2.

4.2 Main Results

All results use the common SGLang serving configuration described above. Update Latency isolates the constraint-update interval inside this serving stack, whereas TPOT is measured over complete serving requests. For every first-seen grammar update, all constrained backends receive the same grammar-update sequence and committed prefix.

First-seen update latency. Table 1 reports the main results on the BFCL-derived growing tool/schema workload with Qwen3-8B. GramDelta reduces median first-seen Update Latency from 9.059 ms with XGrammar to 5.351 ms, a $1.69\times$ reduction in update time. The metric covers the constraint-side interval from grammar revelation to availability of the updated token mask, including state maintenance or reconstruction, committed-prefix reprocessing when required, and token projection.

End-to-end serving cost. GramDelta reaches 40.577 ms/token, corresponding to 119.9% overhead over unconstrained serving, compared with 193.081 ms/token and 946.2% overhead for XGrammar. Thus, although runtime grammar evolution remains more expensive than unconstrained decoding, incremental state maintenance substantially reduces the additional serving cost caused by first-seen grammar updates.

Generalization across models. The advantage of incremental maintenance persists across all four evaluated model/tokenizer configurations; complete cross-model results are reported in Appendix D.

4.3 Analysis and Discussion

Update latency versus end-to-end serving cost. Update Latency and TPOT capture different aspects of runtime grammar evolution. GramDelta directly reduces the constraint-side work triggered by a first-seen grammar update, whereas TPOT also includes model execution and steady-state constrained decoding. Consequently, improvements in Update Latency need not translate proportionally to TPOT. Across the evaluated models, however, GramDelta reduces both metrics relative to reconstruction-based handling, indicating that avoiding repeated historical reprocessing remains visible at the request level.

Why incremental maintenance helps. The benefit of GramDelta comes from reusing computation that remains valid after an additive grammar update. Historical localization restricts where newly introduced productions can enter the materialized Earley state, incremental closure propagates only facts that become newly derivable, and token projection can reuse mask entries whose validity is unaffected. These mechanisms make the update cost depend on the impact of the grammar change rather than on the full length of the committed prefix.

Dependence on update locality. The advantage of incremental maintenance is inherently workload dependent. When a grammar delta activates only a small part of the historical dependency structure and changes a localized portion of the current projection state, most previous computation can be reused. As the affected region grows, the amount of incremental work can approach reconstruction. This affects performance rather than semantics: GramDelta still targets the same parser state and token mask induced by processing the committed prefix under the updated grammar.

Scope and limitations. Our current implementation targets additive grammar evolution. Supporting production deletion or replacement would require invalidating previously materialized parsing facts and cached token-validity information, rather than only propagating newly enabled state. GramDelta also retains historical dependency information so that later grammar updates can reactivate earlier parser states, introducing a memory–update-cost tradeoff that we leave for future work.

5 Conclusion

We introduced GramDelta for additive grammar updates revealed after a generation prefix has already been processed. GramDelta localizes the historical dependencies activated by new productions, propagates only newly enabled Earley facts, and incrementally repairs next-token validity, while remaining equivalent to processing the same prefix under the updated grammar. Its benefit depends on update locality: broad changes may approach reconstruction cost, and supporting non-monotonic grammar changes would additionally require invalidating previously derived state. Extending these ideas to deletion/replacement updates and reducing persistent-state overhead are natural directions for future work.

Reproducibility Statement

We provide an anonymized supplementary code artifact containing the GramDelta implementation and SGLang integration, benchmark traces, experiment configurations and scripts, and the processing code used to generate the reported tables and figures. Section 3 specifies the runtime-update semantics and incremental algorithms, while the appendix provides formal proofs, implementation details, workload construction, and the complete first-seen update protocol. The supplementary artifact contains the exact traces and configurations used for evaluation; future grammar fragments are not exposed to a backend before their designated reveal events.

AI USE STATEMENT

Generative AI tools were used during this work to assist with literature search and organization, brainstorming and refinement of the problem formulation, implementation and debugging, experiment scripting and result analysis, and polishing the manuscript. They were also used to assist in developing intermediate mathematical formulations and proof sketches, and in the design and refinement of scientific figures.

All algorithmic choices, mathematical claims, proofs, experimental protocols, and reported conclusions were reviewed and verified by the authors. AI-assisted code was inspected and tested before use in experiments, and all numerical results reported in the paper were obtained from executed experimental artifacts rather than generated by AI tools. Literature-derived claims and citations were checked against their original sources. The authors take responsibility for the final manuscript, code, figures, claims, and results.

REFERENCES

Francois Bancilhon and Raghu Ramakrishnan. An amateur's introduction to recursive query processing strategies. In *Proceedings of the 1986 ACM SIGMOD international conference on Management of data - SIGMOD '86*, SIGMOD '86, pp. 16–52. ACM Press, 1986. doi: 10.1145/16894.16859. URL http://dx.doi.org/10.1145/16894.16859.

Walter Cazzola and Edoardo Vacchi. On the incremental growth and shrinkage of lr goto-graphs. *Acta Informatica*, 51(7):419–447, 2014. ISSN 1432-0525. doi: 10.1007/s00236-014-0201-2. URL http://dx.doi.org/10.1007/s00236-014-0201-2.

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 Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar (eds.), *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 11253–11267, Vienna, Austria, July 2025. Association for Computational Linguistics. ISBN 979-8-89176-251-0. doi: 10.18653/v1/2025.acl-long.551. URL https://aclanthology.org/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. *Proceedings of Machine Learning and Systems*, 7, 2025.

Jay Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102, February 1970. ISSN 1557-7317. doi: 10.1145/362007.362035. URL http://dx.doi.org/10.1145/362007.362035.

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*, pp. 10932–10952, 2023.

Guidance AI. LLGuidance: Super-fast structured outputs. https://github.com/guidance-ai/llguidance, 2025. Software repository.

Hantao Hua, Jiming Su, Hao Tang, Yiping Yao, and Feng Zhu. Gram2token: Enabling run-time GPU-native grammar-constrained decoding for LLMs. In *Forty-third International Conference on Machine Learning*, 2026. URL https://openreview.net/forum?id=h3K23f6tLU.

IBM Research. Granite 4.1 language models. https://huggingface.co/blog/ibm-granite/granit-4-1, 2026. Accessed: 2026-04-28.

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

Lingxiao Li, Salar Rahili, and Yiwei Zhao. Correctness-guaranteed code generation via constrained decoding. *arXiv preprint arXiv:2508.15866*, 2025.

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*, pp. 1009–1022, 2026.

Niels Mündler, Jingxuan He, Hao Wang, Koushik Sen, Dawn Song, and Martin Vechev. Type-constrained code generation with language models. *Proceedings of the ACM on Programming Languages*, 9(PLDI):601–626, 2025.

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 *Forty-second International Conference on Machine Learning*.

FCN Pereira and DHD Warren. Parsing as deduction. acl proceedings. *21st Annual meeting*, 1983.

Torsten Scholak, Nathan Schucher, and Dzmitry Bahdanau. PICARD: Parsing incrementally for constrained auto-regressive decoding from language models. In Marie-Francine Moens, Xuanjing Huang, Lucia Specia, and Scott Wen-tau Yih (eds.), *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pp. 9895–9901, Online and Punta Cana, Dominican Republic, November 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.emnlp-main.779. URL https://aclanthology.org/2021.emnlp-main.779/.

Paul Stansifer and Mitchell Wand. Parsing reflective grammars. In *Proceedings of the Eleventh Workshop on Language Descriptions, Tools and Applications*, LDTA '11, pp. 1–7. ACM, March 2011. doi: 10.1145/1988783.1988793. URL http://dx.doi.org/10.1145/1988783.1988793.

Xintong Sun, Chi Wei, Minghao Tian, and Shiwen Ni. Earley-driven dynamic pruning for efficient structured decoding. In Aarti Singh, Maryam Fazel, Daniel Hsu, Simon Lacoste-Julien, Felix Berkenkamp, Tegan Maharaj, Kiri Wagstaff, and Jerry Zhu (eds.), *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 57773–57789. PMLR, 13–19 Jul 2025. URL https://proceedings.mlr.press/v267/sun25v.html.

Tim A. Wagner and Susan L. Graham. Efficient and flexible incremental parsing. *ACM Transactions on Programming Languages and Systems*, 20(5):980–1013, 1998. ISSN 1558-4593. doi: 10.1145/293677.293678. URL http://dx.doi.org/10.1145/293677.293678.

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

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

Shuoming Zhang, Ruiyuan Xu, Haofeng Li, Qiuchu Yu, Yangyu Zhang, Chunwei Xia, Xiaobing Feng, Chenxi Wang, Huimin Cui, and Jiacheng Zhao. Decode-time grammars: Constrained llm generation over a refinement order of grammar fragments. *arXiv preprint arXiv:2607.18357*, 2026.

Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody H Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E Gonzalez, et al. Sglang: Efficient execution of structured language model programs. *Advances in neural information processing systems*, 37:62557–62583, 2024.

A Formal Details and Proofs

A.1 Earley Deduction System

We represent the runtime constraint grammar as a context-free grammar

$$G = (N, \Sigma, P, S)$$

where N is the set of nonterminals, Σ the terminal alphabet, P the production set, and S the start symbol. Given a committed input prefix

$$x_{<t} = x_0 x_1 \cdots x_{t-1}$$

which is fixed after generation, the Earley recognizer maintains a chart E_k at each input position $k \in \{0, \ldots, t\}$.

An Earley item is written as

$$[A \to \alpha \bullet \beta, o]_k$$

meaning that production $A \to \alpha\beta$ started at input position o, and α has matched the interval $x_{o:k}$. We refer to o as the *origin* and k as the current chart position. Charts use set semantics, so each item is materialized at most once.

For a fixed grammar and input, the Earley state is the least closure under the following three standard inference rules.

Predict. If an item in E_k expects a nonterminal B, every production of B introduces an initial item at the same position:

$$\frac{[A \to \alpha \bullet B\beta, o]_k \qquad (B \to \gamma) \in P}{[B \to \bullet\gamma, k]_k}$$

Scan. If an item at position k expects terminal a and the current input symbol matches a, the dot advances into the next chart:

$$\frac{[A \to \alpha \bullet a\beta, o]_k \qquad x_k = a}{[A \to \alpha a \bullet \beta, o]_{k+1}}$$

For lexical terminals, $x_k = a$ can be replaced by the corresponding lexical matching relation. The arguments below rely only on the fact that Scan advances monotonically from k to $k+1$, rather than on a particular lexical representation.

Complete. If a constituent B that started at position j completes at position k, every item in E_j waiting for B can advance:

$$\frac{[A \to \alpha \bullet B\beta, o]_j \qquad [B \to \gamma\bullet, j]_k}{[A \to \alpha B \bullet \beta, o]_k}$$

Thus, Complete is keyed by (j, B): the origin j of the completed constituent must coincide with the chart position containing its waiter. This origin-sensitive dependency is used later to construct the incremental completion indices.

We denote the least fixed point generated from the augmented start item

$$[S' \to \bullet S, 0]_0$$

as

$$E_{0:t} = \mathrm{Earley}(G, x_{<t})$$

The prefix is accepted as a complete string when

$$[S' \to S\bullet, 0]_t \in E_t$$

For constrained decoding, however, we additionally retain incomplete states in E_t that admit further terminal continuations, as these states determine next-token validity.

We consider additive grammar updates of the form

$$G' = (N', \Sigma', P \cup \Delta P, S)$$

while keeping $x_{<t}$ unchanged. Since adding productions cannot remove an existing derivation, Earley closure is monotone:

$$E_k \subseteq E'_k, \qquad E'_{0:t} = \mathrm{Earley}(G', x_{<t})$$

We can therefore write

$$E'_k = E_k \cup \Delta E_k, \qquad \Delta E_k = E'_k \setminus E_k$$

and characterize the update entirely in terms of the new Earley facts induced by ΔP.

A.2 Proofs for Historical Delta Localization

This section provides the proofs omitted from Section 3.2. We reuse the notation defined there.

Proof of the First-New-Fact Characterization. Consider a finite Earley derivation of any

$$f \in E' \setminus E$$

and choose a new fact $g \notin E$ of minimum derivation depth. By minimality, every Earley premise of g already belongs to E.

If g is produced by *Scan*, its only premise belongs to E. The grammar update does not modify the committed prefix, and E is already closed under Scan for this fixed input. Hence the same deduction must already have produced g in the old closure, contradicting $g \notin E$.

If g is produced by *Complete*, both its waiting and completed premises belong to E. Since the old Earley state is closed under Complete, their conclusion must also belong to E, yielding the same contradiction.

Thus g must be produced by *Predict*. Let

$$g = [A \to \bullet\gamma, k]_k$$

be derived from a waiter

$$[B \to \beta \bullet A\eta, o]_k \in E_k$$

If $A \to \gamma \in P$, the same Predict deduction was already available in the old grammar, and therefore $g \in E$. Hence

$$A \to \gamma \in \Delta G$$

Because the waiter occurs in E_k, we have $k \in \mathcal{W}_E(A)$, so $g \in S_\Delta$. Therefore every derivation that produces a new Earley fact must first leave the old closure through one of the seeds defined in Section 3.2.

Historical left boundary. Assume $S_\Delta \neq \varnothing$ and define the earliest seeded chart position

$$j_\Delta = \min\{k \mid [A \to \bullet\gamma, k]_k \in S_\Delta \text{ for some } A \to \gamma \in \Delta G\}$$

Corollary 1 (Historical left boundary). No chart preceding j_Δ changes under the additive update

$$\forall k < j_\Delta \qquad E'_k = E_k$$

Proof. Every new derivation begins from a seed in S_Δ. Predict preserves the current chart position, Scan advances to the next position, and Complete produces its consequence at the end position of the completed constituent. None of the three Earley deductions can propagate a newly derived fact to a chart earlier than the position at which the new derivation entered the old closure. Since j_Δ is the earliest such entry position, no new fact can occur in any E_k with $k < j_\Delta$.

Proof of the Empty-Seed No-Op. Assume $S_\Delta = \varnothing$ but $E' \setminus E \neq \varnothing$. The First-New-Fact Characterization would require the derivation of some new fact to contain a first new fact in S_Δ, a contradiction. Thus $E' \setminus E = \varnothing$. Together with monotonicity under additive updates, $E \subseteq E'$, this gives

$$E' = E$$

Why the earliest seed alone is insufficient. The position j_Δ is only a left boundary on historical change. For every

$$k \in \mathcal{W}_E(A)$$

if

$$[A \to \bullet\gamma, k]_k$$

was not already present in E, the newly added production $A \to \gamma$ makes this item itself a member of $E' \setminus E$ through Predict. Exact full-chart equivalence therefore requires retaining all seeds, rather than only the earliest one.

A.3 Proofs for Incremental Earley Closure

This section provides the derivations and correctness arguments omitted from Section 3.3, using the relational notation introduced there.

Incremental Predict. The Predict relation is written as

$$\mathsf{P} = UV$$

where $U(k, A)$ records that position k contains a waiter for A, and $V(A, q_p^0)$ maps a nonterminal to the entry state of one of its productions. A grammar update and subsequent closure propagation yield

$$V' = V \cup \Delta V, \qquad U' = U \cup \Delta U$$

Hence

$$\begin{aligned}\mathsf{P}' &= U'V' \\ &= (U \cup \Delta U)(V \cup \Delta V) \\ &= UV \cup U\Delta V \cup \Delta UV \cup \Delta U\Delta V\end{aligned}$$

Since $UV = \mathsf{P}$ and

$$\Delta UV' = \Delta UV \cup \Delta U\Delta V$$

every potentially new Predict conclusion is generated by

$$U\Delta V \cup \Delta UV'$$

Set-membership checks remove conclusions that have already been materialized. The first term corresponds to existing historical waiters activated by newly added productions, while the second propagates Predict from waiters that themselves arise during the incremental closure.

Incremental Scan. The lexical Scan relation is unchanged by the grammar update. If a Scan conclusion becomes new after the update, its terminal-expecting premise must also be new; otherwise the same deduction would already have been available in the old closure for the unchanged committed prefix. Therefore,

$$\Delta E_{k+1}^{\text{scan}} = \Delta T_k \, \mathsf{S}_{x_k}$$

where S_{x_k} denotes the Scan relation induced by the fixed input symbol x_k.

Incremental Complete. Let $W(j, B)$ denote items at position j waiting for B, and let $C(j, B)$ denote completed B-constituents whose origin is j. Old Complete conclusions arise from

$$J = W \bowtie C$$

With

$$W' = W \cup \Delta W, \qquad C' = C \cup \Delta C$$

all potentially new joins are generated by

$$\Delta J = W \bowtie \Delta C \;\cup\; \Delta W \bowtie C'$$

The first term handles a newly completed constituent against existing waiters; the second handles a newly materialized waiter against existing or newly materialized completions. Any duplicate candidate is removed by membership checking. The keyed structures described in Section 3.3 implement these two cases by joining whichever side arrives later with the already-materialized facts sharing the same (j, B) key.

Exactness of incremental closure. Let $\widehat{E}$ denote the fixed point obtained by starting from S_Δ and repeatedly applying the incremental Predict, Scan, and Complete rules above. Then

$$\widehat{E} = E' = \text{Earley}(G', x_{<t})$$

Soundness. Every seed is a valid Predict conclusion under G'. Every subsequently materialized fact is produced by a standard Earley rule from premises already valid under G'. Hence

$$\widehat{E} \subseteq E'$$

Completeness. Consider any

$$f \in E' \setminus E$$

By the First-New-Fact Characterization, some derivation of f first leaves the old closure through a fact in S_Δ, which is inserted into the initial worklist.

Proceed inductively along the remainder of this derivation. If a new fact is produced by Predict, it is either an initial $U\Delta V$ seed or is triggered by a newly materialized waiter through $\Delta UV'$. If it is produced by Scan, its premise must be new and is therefore propagated by the incremental Scan rule. If it is produced by Complete, at least one of its two premises must be new; otherwise the old closure would already contain the conclusion. A new completion is joined through $W \bowtie \Delta C$, while a new waiter is joined through $\Delta W \bowtie C'$. In every case, the new conclusion is eventually materialized.

Thus every fact in $E' \setminus E$ is produced, giving

$$E' \subseteq \widehat{E}$$

Together with soundness,

$$\widehat{E} = E'$$

For a fixed prefix and finite grammar, the Earley item universe is finite. Since each item enters the worklist only on its first insertion, the incremental closure also terminates.

A.4 Proofs for Exact Incremental Token Projection

This section proves the correctness claims used in Section 3.4. Let Φ_t and Φ'_t denote the projection-relevant states before and after the grammar update, respectively. Recall that Φ_t contains the current scannable parser frontier, acceptance state, and any auxiliary lexical state required by the exact token projector.

For tokenizer vocabulary $\mathcal{V}$, define

$$M_t = \{v \in \mathcal{V} \mid \mathrm{Valid}(\Phi_t, v)\}$$

and

$$M'_t = \{v \in \mathcal{V} \mid \mathrm{Valid}(\Phi'_t, v)\}$$

where $\mathrm{Valid}(\Phi, v)$ means that the byte sequence represented by token v is admissible from projection state Φ. The acceptance component of Φ also determines the admissibility of EOS or other configured stop tokens.

Monotonicity under additive updates. The runtime update only adds grammar structure and leaves the committed prefix unchanged. Every continuation admitted before the update therefore remains derivable after the update. For every $v \in \mathcal{V}$

$$\mathrm{Valid}(\Phi_t, v) \Longrightarrow \mathrm{Valid}(\Phi'_t, v)$$

An additive update likewise cannot turn an accepting prefix into a non-accepting one. Hence

$$M_t \subseteq M'_t$$

Incremental projection therefore needs only to discover newly valid tokens; previously valid entries never need to be removed.

Exactness of affected-token repair. Let $C_\Delta \subseteq \mathcal{V}$ be any conservative affected-token set satisfying

$$v \notin C_\Delta \Longrightarrow \mathrm{Valid}(\Phi'_t, v) = \mathrm{Valid}(\Phi_t, v)$$

The set need not be minimal, but it must contain every token whose validity can change.

Define exact reprojection over the affected set as

$$\Pi(\Phi'_t, C_\Delta) = \{v \in C_\Delta \mid \mathrm{Valid}(\Phi'_t, v)\}$$

We show

$$M'_t = M_t \cup \Pi(\Phi'_t, C_\Delta)$$

First, consider any

$$v \in M_t \cup \Pi(\Phi'_t, C_\Delta)$$

If $v \in M_t$, monotonicity gives $v \in M'_t$. If $v \in \Pi(\Phi'_t, C_\Delta)$, membership in M'_t follows directly from the definition of Π. Therefore

$$M_t \cup \Pi(\Phi'_t, C_\Delta) \subseteq M'_t$$

Conversely, let $v \in M'_t$. If $v \in C_\Delta$, then $v \in \Pi(\Phi'_t, C_\Delta)$. If $v \notin C_\Delta$, conservativeness gives

$$\mathrm{Valid}(\Phi'_t, v) = \mathrm{Valid}(\Phi_t, v)$$

and therefore $v \in M_t$. Hence

$$M'_t \subseteq M_t \cup \Pi(\Phi'_t, C_\Delta)$$

The two inclusions establish the desired equality.

Reuse condition. If the projection-relevant state does not change

$$\Phi'_t = \Phi_t$$

then for every $v \in \mathcal{V}$

$$\mathrm{Valid}(\Phi'_t, v) = \mathrm{Valid}(\Phi_t, v)$$

and therefore

$$M'_t = M_t$$

This justifies direct mask reuse without traversing the tokenizer trie.

Exact fallback. Correctness depends on coverage of C_Δ, not on its size. Whenever a smaller conservative affected region cannot be established, the projector may use

$$C_\Delta = \mathcal{V}$$

in which case

$$\Pi(\Phi'_t, C_\Delta) = M'_t$$

and the procedure reduces to full exact projection. The fallback therefore changes only the amount of projection work, not the resulting token-mask semantics.

B Algorithms and Implementation

B.1 Event-Driven Delta Update

Algorithm 1 in the main text gives the complete recognizer update procedure. Here we describe the persistent representation used to implement its operations.

For a production

$$p : A \to X_1 \cdots X_m$$

GramDelta assigns stable dotted-state identifiers

$$q_p^0, \ldots, q_p^m$$

An Earley fact is stored as

$$(q_p^d, o, k)$$

where o is the origin position and k is the current chart position. Runtime grammar additions append new production and dotted-state metadata; identifiers associated with previously materialized facts are never renumbered.

In addition to request-local chart membership, GramDelta maintains three reverse indices

$$\texttt{HistWait}[A]$$

$$\texttt{WaitAdv}[(k, A)]$$

$$\texttt{CompletedByStart}[(k, A)]$$

`HistWait`$[A]$ stores the deduplicated chart positions containing at least one item that waits for A. `WaitAdv`$[(k, A)]$ stores the corresponding waiting items themselves, while `CompletedByStart`$[(k, A)]$ stores completed A-constituents whose origin is k. The latter two structures implement the two directions of the incremental Complete join: a new waiter queries already materialized completions, and a new completion queries already materialized waiters.

All derived facts pass through a single INSERT operation. For a candidate fact f, INSERT checks request-local chart membership. If f is already present, no further work is performed. Otherwise, f is materialized and placed on the worklist exactly once. The worklist processing in Algorithm 1 subsequently updates the relevant reverse indices and triggers Predict, Scan, or Complete successors.

For a newly revealed production $A \to \gamma$, seed construction queries only `HistWait`$[A]$. No historical chart scan is required on this path. Newly produced waiters and completions then propagate through the persistent indices until the worklist becomes empty. Thus the implementation follows the dependency-driven update characterized in Section 3.3 rather than replaying the committed prefix from an affected position.

B.2 Incremental Token Projection

GramDelta uses a byte-level tokenizer trie shared across requests for the same tokenizer. Token-bearing trie nodes store the corresponding token IDs, while each request retains its current projection-relevant state Φ_t and packed token mask M_t.

After an incremental recognizer update, the projector first compares Φ_t with the updated state Φ'_t. If

$$\Phi'_t = \Phi_t$$

the existing mask is reused directly and no tokenizer-trie traversal is required.

Otherwise, the projector computes a conservative set of affected trie roots R_Δ whose descendants cover every token that may have changed validity. Exact continuation checking is rerun only below these roots. During traversal, parser and lexical continuation states are propagated along token bytes, and a subtree is pruned once no valid continuation remains.

Let

$$C_\Delta = \bigcup_{r \in R_\Delta} \mathrm{TokensBelow}(r)$$

The implementation need not materialize C_Δ explicitly; the affected roots define the same candidate region implicitly. Local repair updates only packed mask words reached by the affected traversal.

If the affected region is broad, or if the implementation cannot establish a conservative local root set, the same continuation checker is invoked from the full tokenizer trie. This is the full-projection path proved exact in Appendix A.4.

The serving path therefore retains a single packed mask representation across all three cases: unchanged masks are reused, localized updates patch affected words, and full projection rebuilds the complete packed mask.

C Experimental Details

C.1 Workloads and System Setup

We evaluate two classes of runtime grammar workloads.

First, we construct controlled dynamic grammars covering three dependency patterns: recursive updates, subgrammar extension, and type/declaration-dependent updates. These families exercise recursive historical propagation, attachment of newly revealed subgrammars to previously materialized parser states, and runtime growth of symbol or schema alternatives.

Second, we derive growing tool/schema traces from the tool and function schemas provided by BFCL. BFCL supplies realistic function names, argument fields, types, and nested schema structures. We translate each selected instance into a shared base grammar and a sequence of appendable application-specific grammar fragments that are progressively revealed during generation. This

Table 2: Hardware and software configuration used for the serving experiments.

Component	Configuration
GPU	NVIDIA RTX PRO 6000 Blackwell, single GPU
Precision	BF16
SGLang	0.5.20
XGrammar	0.2.7

progressive-reveal protocol is introduced by our evaluation and is not part of the original BFCL benchmark.

We evaluate Qwen3-8B, Qwen3-32B, Granite-4.1-3B, and Granite-4.1-8B. All models are executed in BF16 on a single NVIDIA RTX PRO 6000 Blackwell GPU. GramDelta, XGrammar are integrated into the same SGLang serving stack and use identical model weights, tokenizers, and decoding configurations for a given model.

C.2 First-Seen Update Protocol

Each trace starts from an initial grammar G_0 and contains a sequence of runtime grammar deltas

$$\Delta G_1, \ldots, \Delta G_m$$

Before event i, a backend has access only to the current grammar G_{i-1}. Future grammar fragments are not registered or compiled in advance. At the reveal event, ΔG_i becomes visible for the first time and the active grammar becomes

$$G_i = G_{i-1} \cup \Delta G_i$$

Updates occur after the current token has been committed, so all methods process the same grammar change against the same historical prefix $x_{<t}$.

GramDelta applies ΔG_i directly to its persistent request-local recognizer state. For XGrammar, we use their public constraint interfaces to instantiate the backend state corresponding to G_i and reprocess the same committed prefix. This reconstruct-and-replay path describes their handling under our first-seen late-update protocol rather than a general characterization of those systems.

Update Latency is measured from the time ΔG_i becomes visible to the constraint backend until the updated token mask is available for the next decoding step. Model forward execution is excluded from this event-level interval, while constraint-state maintenance, required prefix reprocessing, and token projection are included. Each backend is independently warmed up before formal measurement, without exposing grammar fragments belonging to future formal events.

The main serving comparison uses a shared-token protocol so that all backends follow the same committed token sequence.

Token-policy alignment. For GramDelta–XGrammar mask comparisons, we exclude empty-payload special tokens and admit configured stop tokens only in accepting states. We therefore use normalized-mask comparison for GramDelta and XGrammar.

D Extended Experimental Results

Table 3 reports the complete serving results across all four evaluated models. We report the median first-seen Update Latency together with request-level TPOT and TPOT Overhead.

Table 3: Serving results across model configurations. Update Latency is the median over first-seen runtime grammar updates. TPOT Overhead is relative to unconstrained serving for the same model.

Model	Method	Median Update (ms) ↓	TPOT (ms/token) ↓	TPOT Ovhd. (%) ↓
Qwen3-8B	Unconstrained	–	18.455	0.0
	XGrammar	9.059	193.081	946.2
	GramDelta	**5.351**	**40.577**	**119.9**
Qwen3-32B	Unconstrained	–	56.278	0.0
	XGrammar	7.225	221.447	293.5
	GramDelta	**5.417**	**77.947**	**38.5**
Granite-4.1-3B	Unconstrained	–	18.062	0.0
	XGrammar	7.278	168.548	833.2
	GramDelta	**4.404**	**40.388**	**123.6**
Granite-4.1-8B	Unconstrained	–	20.616	0.0
	XGrammar	7.204	158.343	668.1
	GramDelta	**4.365**	**42.852**	**107.9**